



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

OPTIMIZATION OF REAL-TIME ROUTE CALCULATION ALGORITHM

Autor: Blanca Serrano Marfil

Director: Miguel García Domínguez

Madrid

Junio 2018

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Optimization of Real-Time Route Calculation Algorithm

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2017/18 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.



Fdo.: Blanca Serrano Marfil

Fecha: 11/ 06/ 2018

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Fdo.: Miguel García Domínguez

Fecha: 11/ 06/ 2018

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESIS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. _____ Blanca Serrano Marfil _____
DECLARA ser el titular de los derechos de propiedad intelectual de la obra: _____ Optimization of Real-Time Route Calculation Algorithm _____, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

- a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar “marcas de agua” o cualquier otro sistema de seguridad o de protección.
- b) Reproducir la en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
- c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
- d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
- e) Asignar por defecto a estos trabajos una licencia Creative Commons.
- f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente*).

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

- a) Que la Universidad identifique claramente su nombre como autor de la misma
- b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
- c) Solicitar la retirada de la obra del repositorio por causa justificada.
- d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

- a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
- b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
- c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.
- d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción

de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusiva del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a 11 de Junio de 2018.

ACEPTA

Fdo. Blanca Serrano Marfil



Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

OPTIMIZATION OF REAL-TIME ROUTE CALCULATION ALGORITHM

Autor: Blanca Serrano Marfil

Director: Miguel García Domínguez

Madrid

Junio 2018

Acknowledgements

I want to thank the company Baolau and its members for the amazing opportunity they have offered to me. Their passion for what they do and their forward-thinking have been a great motivation to me.

Blanca Serrano Marfil

OPTIMIZACIÓN DE ALGORITMOS DE CÁLCULO DE RUTAS EN TIEMPO REAL

Autor: Serrano Marfil, Blanca.

Director: García Domínguez, Miguel.

Entidad Colaboradora: Baolau PTE. LTD.

RESUMEN DEL PROYECTO

Este proyecto busca optimizar el procedimiento de cálculo de rutas múltiples de la empresa Baolau. Esto implica la mejora del acceso a la base de datos de las rutas así como la selección, desarrollo e implementación de un algoritmo de cálculo de rutas múltiples para sustituir al que está actualmente en uso (Dijkstra), y poder comparar el rendimiento de ambos.

Palabras clave: IEA, Dijkstra, rutas múltiples, algoritmo.

1. Introducción

Baolau es una compañía de búsqueda multi-transporte en el sureste asiático que gestiona la información sobre las rutas de interconexión de ciudades, las reservas de viajes y los pagos. Facilita tanto las rutas directas entre dos puntos como las combinadas que las unen. Dijkstra ha sido el algoritmo empleado para ello hasta ahora. Sin embargo, debido a la expansión de la compañía a otros países y su consecuente incremento de los nodos en su sistema, este algoritmo no alcanza a enfrentarse a los nuevos retos que han surgido.

2. Definición del Proyecto

El principal objetivo de este proyecto es mejorar el método de la empresa para el cálculo de rutas múltiples. Para ello, el algoritmo actual ha de ser sustituido por uno que pueda cumplir con las nuevas expectativas. Se ha llevado a cabo una investigación detallada para seleccionar el sustituto óptimo: el Algoritmo de Enumeración Iterativa (IEA). Se ha procedido con su desarrollo e implementación en el sistema de la compañía así como con el diseño del nuevo acceso a la base de datos de rutas.

3. Descripción del modelo/sistema/herramienta

El propósito de encontrar un sustituto al algoritmo de Dijkstra es emplear uno capaz de proporcionar más de una sola solución y suficientemente eficiente para trabajar con grandes cantidades de información. Dijkstra es un algoritmo que obtiene el camino más corto desde un origen y que necesita ser ejecutado tantas veces como resultados adicionales se deseen, eliminando cada vez un enlace entre nodos (Algoritmo de Yen). El proceso de encontrar rutas combinadas resulta ser arbitrario, ineficiente y extenso.

IEA se ha seleccionado tras una comparación de los pros y contras de los diferentes algoritmos de la investigación. Algoritmos de búsqueda como Breadth y Depth-First Search son demasiado simples y no siempre proporcionan una solución. Bellman-Ford trabaja con pesos negativos y Floyd-Warshall o Johnson generan el camino más corto entre todos los pares de nodos, características no deseadas en este caso. El algoritmo de programación dinámica A* es rápido y eficiente gracias a la función heurística que aplica, cuya complejidad puede llegar a ser su mayor desventaja. Los algoritmos que proporcionan los k caminos más cortos, como Yen o Eppstein, podrían haber sido

escogidos si no fuese por la incertidumbre del primero y la complejidad del último. IEA también pertenece a esta categoría y, gracias a su estructura de etiquetas y predictibilidad, ha sido el seleccionado.

La operación de IEA se basa en el uso de etiquetas asignadas a cada nodo y que recogen información acerca del camino empleado para llegar a dicho nodo. Una etiqueta corresponde con una determinada ruta que une dos puntos directamente. Estos datos incluyen el peso acumulado del camino así como los nodos por lo que pasa, el índice de la etiqueta dentro de dicho nodo y si la etiqueta ha sido considerada hasta ahora o no. La Ilustración 1 representa el funcionamiento del algoritmo a partir de estas etiquetas.

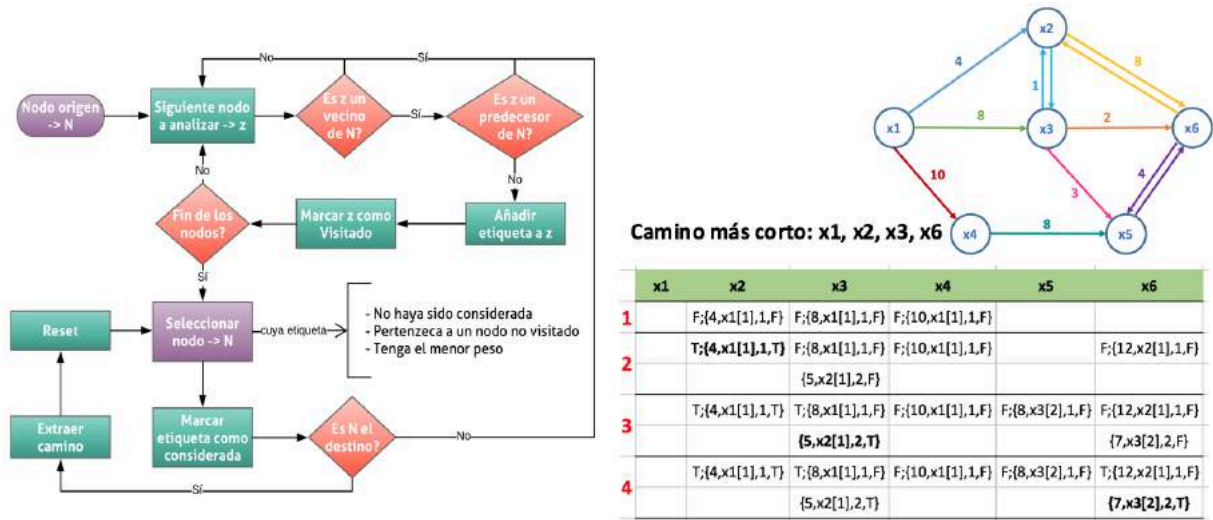


Ilustración 1 – Diagrama de flujo de operación y estructura de etiquetas de IEA.

El primer nodo en el proceso es el punto de origen. Las etiquetas solo serán extendidas a sus vecinos y éste será el predecesor incluido en cada una de dichas etiquetas (paso 1). Después, el nodo que contenga la etiqueta con menor peso acumulado será seleccionado para ocupar el lugar del nodo inicial. La etiqueta se marcará como considerada para que no esté disponible en el futuro. También el nodo se marca como visitado para que sus etiquetas ya no sean estudiadas en esta iteración (paso 2). El proceso se repite (pasos 3 y 4), hasta el nodo seleccionado sea el destino, evitando en cada iteración extender etiquetas a los nodos que sean predecesores del nodo seleccionado en el momento. A continuación se obtiene el primer camino a partir de los predecesores del nodo en el que se encuentran la etiqueta. Los nodos pertenecientes al camino se “resetearán”, esto es, se marcarán como no visitados para que sus etiquetas disponibles puedan ser consideradas en el futuro.

El proceso comienza de nuevo obteniendo un camino diferente cada vez (k caminos más cortos). Una ventaja de IEA es que su estructura de etiquetas se puede adaptar fácilmente a escenarios dependientes del tiempo (*transportation networks*), donde las rutas están sujetas a horarios. La modificación simplemente consiste en añadir dos campos a las etiquetas: el tiempo de salida de la ruta y de llegada al nodo al que pertenece la etiqueta. De esta manera solo las etiquetas que cumplen con las restricciones oportunas podrán añadirse a los nodos. Si una ruta cuyo tiempo de salida es previo al tiempo de llegada de la anterior a dicho nodo, la etiqueta no se creará. Esto resulta ser una gran ventaja para la elección del algoritmo, ya que otros como Eppstein no pueden ajustarse a este tipo de cambios.

4. Resultados

En este proyecto IEA se ha implementado en PHP y se ha adaptado para poder trabajar con la estructura de rutas de Baolau. También el acceso a la base de datos de rutas se ha modificado para poder optimizar todo el proceso de cálculo de rutas múltiples. El rendimiento global se ha comparado con el de Dijkstra en cuanto a tiempo de ejecución y eficiencia en un escenario (entre otros) donde ambos han de calcular el mismo número de caminos entre dos determinados puntos del grafo. Cada ejecución se ha repetido 10 veces para un cierto número de caminos.

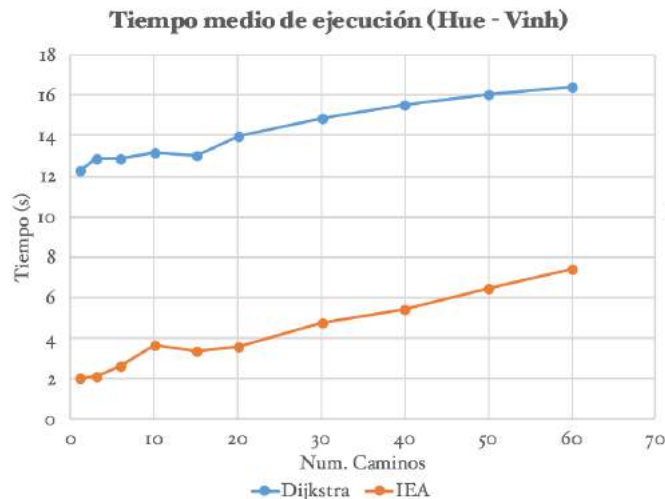


Ilustración 2 – Comparación de rendimiento de IEA y Dijkstra.

Tal y como muestra la Ilustración 2, Dijkstra junto al acceso a la base de datos, resulta en tiempo de ejecución sumamente alto comparado con el de IEA, el cual está incluso operando con un número mayor de rutas debido a la nueva consulta diseñada.

5. Conclusiones

Con la investigación que se ha llevado a cabo en este proyecto se ha considerado a IEA como el mejor algoritmo de sustitución de Dijkstra debido a la ausencia de incertidumbre que su estructura de etiquetas proporciona, ya que ayuda a mantener un registro de todos los posibles caminos desde cada nodo. Además se ha probado cómo el tiempo de ejecución para el cálculo de rutas múltiples se ha reducido y la eficiencia incrementado gracias a su capacidad de trabajar con mayor cantidad de rutas.

6. Referencias

- [1] Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein. *Introduction to Algorithms. 3rd Edition*. MIT Press, Cambridge, Massachusetts London, England. 2009.
- [2] Grégoire Scano, Marie-José Huguet and Sandra Ulrich Ngueveu. *Adaptations of k-Shortest Path Algorithms for Transportation Networks*. 6th IESM Conference, October 2015, Seville, Spain.
- [3] David Eppstein. *Finding the k Shortest Paths*. March 31, 1997. Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, eppstein@ics.uci.edu, <http://www.ics.uci.edu/~eppstein/>.

OPTIMIZATION OF REAL-TIME ROUTE CALCULATION ALGORITHM

Author: Serrano Marfil, Blanca.

Supervisor: García Domínguez, Miguel.

Collaborating Entity: Baolau PTE. LTD.

ABSTRACT

This project aims at optimizing the multiple-route calculation procedure used so far in the company Baolau. This implies improving the route database access as well as selecting, developing and implementing a multiple-route calculation algorithm in order to substitute the currently used Dijkstra's algorithm and compare both algorithm's performance.

Keywords: IEA, Dijkstra, multiple-route, algorithm.

1. Introduction

Baolau is a multi-transport search company in South-East Asia that provides information about city interconnection routes, and manages trip reservations and payments. Apart from supplying direct routes between two points, it also displays combinational routes to link them. Dijkstra has been the algorithm used to do so, so far but, due to the expansion of the company to other countries and the consequent increase of nodes in their system, this algorithm falls short on meeting the new challenges aroused.

2. Project definition

The main goal of this project is to improve the multiple-route calculation methodology the company has already developed. For this purpose, the current algorithm in use must be replaced by one that can meet the company's new expectations. A detailed research on the different possibilities has been carried out to select the optimal substitute: Iterative Enumeration Algorithm (IEA). Its development and implementation have followed as well as its adaptation to the company's system and new access to the database.

3. Model description

The principal purpose for finding a replacement for Dijkstra's algorithm is to employ one capable of finding more than one solution (the shortest) and efficient enough to work with large amounts of data. Dijkstra is a single-source shortest path algorithm which needs to be executed as many times as the desired number of different results, eliminating each time an edge of the graph (Yen's algorithm). This renders the multiple-route calculation process arbitrary, inefficient and time-consuming.

Selecting IEA among the other considered possibilities is the result of a thorough research that has helped to determine the pros and cons of all of them. The studied search algorithms, such as Breadth-First and Depth-First Search, are too simple and not always provide a solution. Bellman-Ford works with negative weights, and Floyd-Warshal or Johnson are examples of all-pairs shortest-path graph algorithms, all too complex and with additional features not interesting for this project. The dynamic-programming algorithm A* is fast and efficient thanks to the heuristic function it applies, whose complexity can become also its drawback. K-shortest paths algorithms, such as Yen and

Eppstein, could have become potential replacements for Dijkstra if not for the uncertainty of the first one and great complexity of the latter. IEA also provides k-shortest paths and, thanks to its labeled structure and unpredictability avoidance, has been the best option chosen to be implemented.

IEA's operation is based on labels assigned to each node and that store information about the path used to reach that node. A label corresponds to a certain route connecting two specific nodes. This data includes the accumulated weight and the nodes belonging to that path, the index of the label in the node's label set and finally, if the label has been employed or not for future calculations. Illustration 1 depicts the algorithm's functioning by means of these labels.

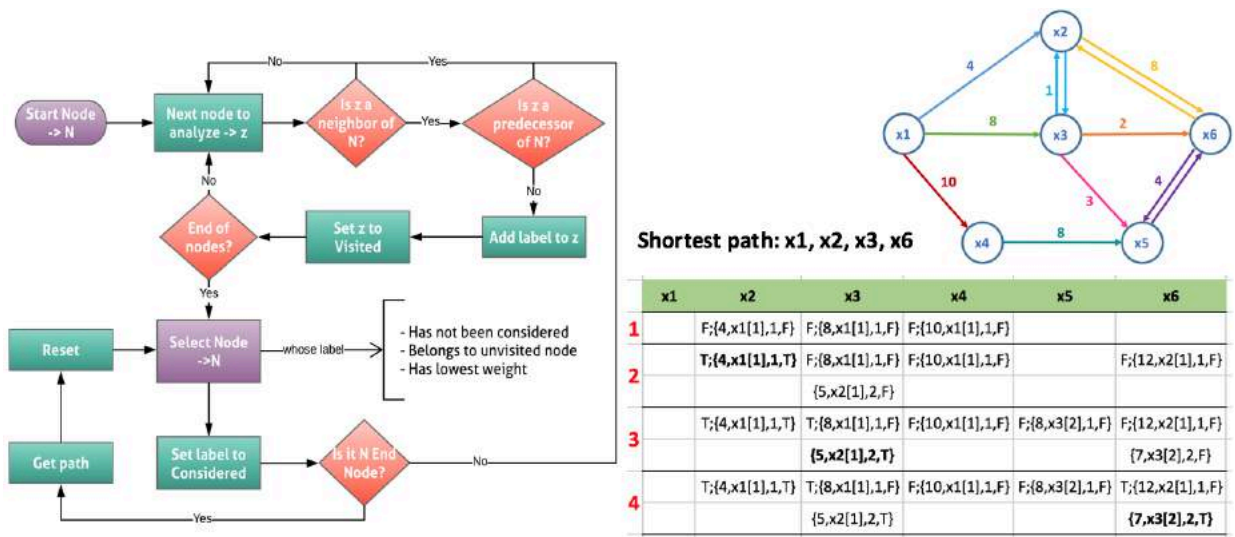


Illustration 1 – Operation flowchart and IEA label structure.

The first node to be considered is the start point. Only labels will be extended to this node's neighbors and the start node will be set as the predecessor of these nodes and stated this way in each label (step 1). Then, the node whose label owns the lowest accumulated weight will be selected and will occupy the beginning node's place. As the label has been considered it will no longer be available in future iterations. Also this node is set to visited so its labels will no longer be studied in this iteration (step 2). The process is reiterated (steps 3 and 4), until the chosen node is the destination point, now avoiding to extend labels to nodes that are predecessors of the selected node. Next, the first path can be extracted by the predecessor's node stored in each label. Afterwards, the nodes belonging to the last extracted path will be "reset", that is, set to unvisited so their available labels can be considered in the future.

The process begins once again, obtaining a different path in each iteration and achieving this way the k-shortest paths extraction so distinctive of this algorithm. One particular advantage of its labeled structure is that it can be easily adapted to time-dependent scenarios (*transportation networks*) where the routes are subjected to timetables. The modification introduced simply consist on adding two fields to the labels: the route's departure time and the arrival time to the node to which the label belongs. This way, only the labels that comply with the time restrictions will be extended in each node. This means that a route whose departure time is earlier than the previous arrival to that node will not be created. This proves to be a great advantage as other algorithms, such as Eppstein's, could not be submitted to any adjustment of this kind.

4. Results

In this project, IEA has been implemented in PHP. It has been adapted to operate with Baolau's own route structure. Also the access to the routes database has been changed in order to optimize the whole multi-route searching process. The overall performance has been compared to that of Dijkstra in terms of execution time as well as efficiency in a scenario (among others) where both calculate an equal number of paths between the same two points. Each execution has been repeated 10 times for each number of paths.

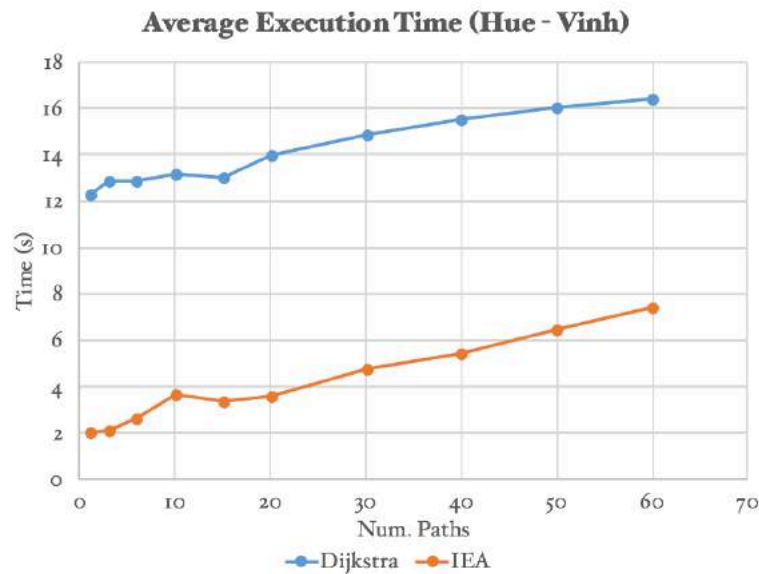


Illustration 2 – Performance comparison between IEA and Dijkstra.

As Illustration 2 shows, Dijkstra execution together with the former database access procedure, results in an extremely high execution time compared to IEA, which is operating with a higher amount of routes due to the new database query.

5. Conclusions

The detailed research carried out in this project deemed IEA the best algorithm to replace Dijkstra, thanks to the uncertainty avoidance provided by its labeled structure which allows to keep record of all possible paths from every node. Moreover, it has been proven how the execution time has been reduced and the efficiency increased, due to its capability of operating with a higher number of routes.

6. Bibliography

- [1] Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein. *Introduction to Algorithms. 3rd Edition*. MIT Press, Cambridge, Massachusetts London, England. 2009.
- [2] Grégoire Scano, Marie-José Huguet and Sandra Ulrich Ngueveu. *Adaptations of k-Shortest Path Algorithms for Transportation Networks*. 6th IESM Conference, October 2015, Seville, Spain.
- [3] David Eppstein. *Finding the k Shortest Paths*. March 31, 1997. Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, eppstein@ics.uci.edu, <http://www.ics.uci.edu/~eppstein/>.

Index

Chapter 1. Introduction.....	9
1.1 Motivation	11
Chapter 2. Resources Description	14
Chapter 3. State of the Art	15
Chapter 4. Project Definition	18
4.1 Justification	18
4.2 Objectives	19
4.3 Methodology.....	20
4.4 Planning and Economic Estimation.....	21
Chapter 5. Research on Algorithms	23
5.1 Algorithm classification	23
5.2 O-Big Notation.....	25
5.3 Design	26
5.4 Current Algorithm in use: Dijkstra.....	30
5.4.1 Original implementation	30
5.4.2 Modified implementation: the removal of edges	33
5.4.3 Advantages and Disadvantages.....	36
5.5 Algorithms included in the research.....	37
5.5.1 Search Algorithms.....	37
5.5.2 Dynamic Programming Algorithms.....	49
5.5.3 Graph Algorithms.....	57
5.5.4 K-Shortest Paths.....	71
5.6 Algorithms Comparison.....	101
Chapter 6. Implementation of IEA Algorithm	104
6.1 The role of IEA	104
6.2 Problems solved by IEA.....	106

6.3	Original Implementation	108
6.3.1	Adaptation of IEA to time-dependent structure	109
6.3.2	Modeling of IEA adjusted to data structure	112
6.3.3	Overcoming the drawbacks	126
6.4	New Data Structure.....	128
6.5	Additional modifications.....	130
6.6	Interface modifications	133
Chapter 7.	Results Analysis.....	138
7.1	Comparison between Dijkstra and IEA	138
7.1.1	Scenario I: Execution with the same number of routes.....	139
7.1.2	Scenario II: Execution on the same set of routes	145
7.1.3	Conclusions	151
7.2	IEA performance analysis	152
7.2.1	Option I: Fixed number of paths	153
7.2.2	Option II: Variable number of paths	154
7.2.3	Option III: Fixed execution time	155
7.2.4	Conclusions on IEA performance analysis	157
Chapter 8.	Conclusions and Future Contributions.....	158
Chapter 9.	Bibliografía	¡Error! Marcador no definido.
Appendix A.	165

Figure Index

Figure 1. Algorithms' time complexity in O-Big Notation [80].....	26
Figure 2. Simplified example of a nodes graph.	27
Figure 3. Example of a graph with its nodes and edges.....	32
Figure 4. Steps for Dijkstra Algorithm.	32
Figure 5. Example of a graph with and edge removed.	34
Figure 6. Steps for Dijkstra Algorithm eliminating edges.....	34
Figure 7. Diagram without weights.	39
Figure 8. Steps for Breadth-Search Algorithm.	40
Figure 9. Breadth-Search Algorithm with adjacency list.....	41
Figure 10. Tree for Depth-First Search algorithm.	44
Figure 11. Depth-First Search Algorithm.	45
Figure 12. Depth-Search Algorithm with stack.	46
Figure 13. Manhattan distance.....	51
Figure 14. Diagonal distance.	52
Figure 15. Euclidean distance.....	52
Figure 16. Graph for A* with h(n).....	54
Figure 17. Steps for A* algorithm.	55
Figure 18. Steps for Bellman-Ford Algorithm.....	60
Figure 19. Initialization for Floyd-Warshall algorithm.	65
Figure 20. Distance matrix for Floyd-Warshall.....	65
Figure 21. Last iteration of Floyd-Warshall algorithm.....	66
Figure 22. Modification of Floyd-Warshall Algorithm.....	66
Figure 23. Next Matrix for Floyd-Warshall Algorithm.....	67
Figure 24. Last iteration of the modified Floyd-Warshall algorithm.....	68
Figure 25. Reversed graph for Breadth-First search in Eppstein's algorithm.	75
Figure 26. Tree for reversed Breadth-First Search in Eppstein's algorithm.....	75

Figure 27. Graph with shortest paths in Eppstein’s algorithm.....	76
Figure 28. Calculation of sidetrack edges.....	77
Figure 29. Tree of sidetracks.	78
Figure 30. Heaps for the sidetracks.....	79
Figure 31. Final arrangement of heaps.	79
Figure 32. Breadth-First Search on Eppstein’s Algorithm with heaps.	81
Figure 33. Graph where IEA is applied.	87
Figure 34. First step labels on IEA.	87
Figure 35. Label structure in IEA.	88
Figure 36. Second step labels on IEA.....	88
Figure 37. Label structure in second step in IEA.....	89
Figure 38. First iteration labels of IEA.	89
Figure 39. Building of path in first iteration in IEA.	90
Figure 40. Resetting of labels after first iteration in IEA.	91
Figure 41. Labels for second iteration of IEA.	92
Figure 42. New labels extended to nodes in step 3 in first iteration of IEA.....	95
Figure 43. First iteration labels for IEA without cycles.....	95
Figure 44. Second iteration without cycles IEA.....	96
Figure 45. Building of path in first iteration in IEA without cycles.....	97
Figure 46. Third iteration IEA without cycles.....	98
Figure 47. IEA operation flowchart.....	99
Figure 48. Calculation of routes flowchart.....	105
Figure 49. Simple time-dependent graph.....	110
Figure 50. First step of IEA with time dependency.....	110
Figure 51. Label structure in IEA with time-dependency.....	111
Figure 52. Second and third steps of IEA with time-dependency.....	111
Figure 53. First iteration of IEA with time-dependency.....	112
Figure 54. Structure of label when implementing IEA.....	114
Figure 55. UML class diagram for IEA internal structure.....	119
Figure 56. Query to obtain routes for combined results.....	121

Figure 57. Multiple-access to combined results flowchart.	131
Figure 58. Route-search form in baolau.com.....	134
Figure 59. Transport filters.	135
Figure 60. Search results filters.	135
Figure 61. Unfiltered results Vientiane – Luang Prabang.	136
Figure 62. Filtered results Vientiane – Luang Prabang.	137
Figure 63. Load-more button hidden when there are no results.	137
Figure 64. Scenario I – Dijkstra average execution time.....	140
Figure 65. Scenario I – Dijkstra number of executions vs. number of paths.....	141
Figure 66. Scenario I -IEA average execution time.....	142
Figure 67. Scenario I – IEA number of executions vs. number of paths.....	143
Figure 68. Scenario I – Performance time comparison between Dijkstra and IEA.....	144
Figure 69. Scenario I – Average total time of execution comparison between Dijkstra and IEA.....	145
Figure 70. Scenario II -Dijkstra average execution time.	146
Figure 71. Scenario II – Dijkstra number of executions vs. number of paths.	147
Figure 72. Scenario II – IEA average execution time.....	148
Figure 73. Scenario II – IEA number of executions vs. number of paths.....	149
Figure 74. Scenario II – Performance time comparison between Dijkstra and IEA.....	150
Figure 75. Scenario II – Average total time of execution between Dijkstra and IEA.	151
Figure 76. IEA average execution time for different set of routes.	153
Figure 77. IEA average number of iterations with a fixed execution time.....	156
Figure 78. IEA complete label structure without cycles.....	170
Figure 79. IEA complete label structure in time-dependent scenario.....	171

Table Index

Table 1. Project time line.	21
Table 2. Project cost estimation.	22
Table 3. Algorithm classification, definition and examples.	25
Table 4. Dijkstra’s specifications.	30
Table 5. Breadth-First Search specifications.	38
Table 6. Depth-First Search Specifications.	43
Table 7. Breadth-First and Depth-First comparison.	49
Table 8. A* Specifications.	49
Table 9. Bellman-Ford specifications.	57
Table 10. Floyd-Warshall specifications.	62
Table 11. Johnson specifications.	69
Table 12. Yen specifications.	72
Table 13. Eppstein specifications.	73
Table 14. IEA specifications.	83
Table 15. Algorithms comparison.	102
Table 16. Weights passed into Dijkstra.	107
Table 17. Weights passed to IEA.	107
Table 18. Information returned for each path by IEA.	125
Table 19. Scenario I – Recorded data for Dijkstra execution.	139
Table 20. Scenario I – Recorded data for IEA.	142
Table 21. Scenario II – Recorded data for Dijkstra execution.	146
Table 22. Scenario II – Recorded data for IEA.	148

Formula Index

Formula 1. Dijkstra's time complexity.....	33
Formula 2. Time complexity of Breadth-First Search Algorithm.....	42
Formula 3. Time complexity of Depth-First Search Algorithm.....	47
Formula 4. Function for A* algorithm.....	50
Formula 5. Manhattan distance.....	51
Formula 6. Diagonal distance.....	52
Formula 7. Euclidean distance.....	52
Formula 8. Heuristics condition for A* Algorithm.....	56
Formula 9. Approximated time complexity of Bellman-Ford.....	61
Formula 10. Time Complexity for Floyd-Warshall Algorithm.....	68
Formula 11. Time complexity for Johnson's Algorithm.....	70
Formula 12. Time Complexity for Yen's Algorithm.....	72
Formula 13. Calculation of Sidetrack value.....	76
Formula 14. Time complexity for unbounded heap for Breadth-First Search for Eppstein's Algorithm.....	82
Formula 15. Time complexity for bounded heap for Breadth-First Search for Eppstein's Algorithm.....	82
Formula 16. Time complexity for IEA.....	99

Pseudocode Index

Pseudocode 1. Dijkstra's Algorithm.....	31
Pseudocode 2. Breadth-First Search Algorithm.....	38
Pseudocode 3. Depth-First Search Algorithm.	44
Pseudocode 4. A* Algorithm.....	53
Pseudocode 5. Bellman-Ford Algorithm.	59
Pseudocode 6. Floyd-Warshall Algorithm.....	63
Pseudocode 7. Floyd-Warshall Algorithm with path reconstruction.....	64
Pseudocode 8. Eppstein's Algorithm.....	74
Pseudocode 9. Breadth-First Search with heaps for Eppstein's Algorithm.....	80
Pseudocode 10. IEA.....	86
Pseudocode 11. IEA without cycles.....	94

Chapter 1. INTRODUCTION

Tourism is an activity that has been growing for the past few years and Asia has been one of the main destinations among international travelers. Specifically, 1.24 billion tourists were arriving in Asia by 2016 [70]. Consequently, more companies and travel service providers have been offering a wide range of new connections and means of transportation to journey between cities and sightseeing sites in these regions.

A few years ago, search machines programs and booking companies encountered here a market niche that has been deeply explored since then. The extensive travel connections, timetables and fares conform a broad arrangement of trip possibilities each company manages differently in terms of efficiency, readiness and price. Baolau is one of these companies which immersed its activities in this business nook.

Baolau is a multi-transport search company in south-east Asia. It applies a search engine in its web page www.baolau.com [69]. Its beginnings date back to 2013 in Vietnam being established now both in Vietnam and Singapore. Since then, the business has extended its services to countries such as Cambodia, Laos and Thailand apart from Vietnam, where it grew since its dawn. Its employees conform an intercultural team gathering people from Vietnam, Spain, Singapore, Italy and Japan.

Among the services offered by the company, it provides the possibility of accessing to connection routes between different destination points and to information about the providers supplying their services. Also they offer the available timetables and the reservation of combined trips by bus, train, plane and ferry, helping this way to the planning of journeys in these countries.

In this context, the company has employed Dijkstra's algorithm for the combined routes calculation between two cities and regions. Dijkstra is a graph algorithm that calculates the shortest path between two nodes located in this graph. This a very extended algorithm for

multi-route calculation applied in numerous situations nowadays thanks to its simplicity and efficiency [59]. The algorithm provides the shortest path between one node and the rest in the graph and, therefore, its capability is restricted to finding just one connection between two certain nodes. In order to be able to obtain more connections, Dijkstra needs to be run more times, each time removing a different edge so a different solution is forced out from the algorithm's usual functioning (Yen's algorithm [26]).

At the beginning, when the number of nodes conforming the transport graph managed by Baolau was moderately small, Dijkstra could show all the optimal connection results between two points in a fast and effective way thanks to its simplicity and efficiency. Thus, running the algorithm several times removing edges in each iteration so new results could be obtained, did not imply an overload problem for the system. However, due to the company's expansion to new regions and countries, the transport network has grown into a more complex mesh where nodes have increased in number as well as the routes interconnecting them. The implications of this growth have affected directly the algorithm application into the system. The amount of nodes over which the algorithm must iterate is indeed too extensive, having a negative effect on the time necessary to obtain the results. This prolongation in the execution time has even impacted the whole system's integral operation.

Optimization of Real-Time Route Calculation Algorithm is a project in collaboration with Baolau, and whose main objective is the study, selection, development and implementation of a new route calculation algorithm. It must be able to procure several connection solutions between two nodes in the graph in a quick and proficient way taking into account the vast amount of information with which the company works now. Its application into the system must provide a solution to these problems it is currently facing.

It is expected that the execution of this new algorithm will not only ensure the correct system integral functioning, but also provide a collection of different routes connecting two nodes in a shorter overall execution time and owning a variety of combinations Dijkstra was not capable of obtaining.

1.1 MOTIVATION

As mentioned before, Dijkstra is an algorithm very much extended in applications and systems whose main objective is finding the shortest path between two nodes. However, this shortest path can be found according to different criteria: time, price, waiting time... Some users may prefer one of these measures but other may choose another one entirely. Consequently, it is in the company's best interest to provide different solutions regarding these different choices. On the other hand, Baolau, as well as similar companies, keeps growing and expanding to the countries it wishes to offer its transport organization services. This growth implies, not only the inclusion of the new routes acquired by the different providers but also the integration of the information regarding time schedules, prices and trip conditions.

The provision of a new algorithms is based in two solid grounds: the need to provide more than one solution and the ability to work with such loads of information as these kind of companies are expected to operate with.

Then, it has been proven that all the amount of information the company needs to handle exceeds Dijkstra's capabilities in its current implementation. Specifically, the algorithm was executed at least three times in which the edges connecting the nodes were weighted in terms of time, price and waiting time. Afterwards, if more results were needed, edges began to be removed from the graph. This is the philosophy followed by Yen's algorithm: the application of a single-source shortest path algorithm (such as Dijkstra) several times, removing an edge each time so solutions differ.

The main disadvantage of applying this procedure is that no criteria for removing edges could be found. This lead to the elimination of edges that did not affect to the solution obtained, making the reiteration of the algorithm necessary. All this process can be summarized in an unneeded loss of time and resources. Another problem encountered is the uncertainty faced when removing the edges for new solutions, as finding the next optimal path cannot be in any way ensured. As a consequence, pivot nodes had to be designated.

These nodes are some important cities or regions through which some routes were forced to go in order to ensure the algorithm provided at least some coherent results.

As a summary, these are the main problems encountered when executing Dijkstra's current implementation in the company:

- The algorithm's inherent design forced its execution as many times as different paths wanted to be found. Sometimes even more times, as the paths obtained could have been repeated before.
- This previous practice resulted in an unnecessary consumption of time and resources essential for the system's correct functioning. As a result, the system itself collapsed due to the excessive use of its assets.
- There was no possibility of charging more results added to the ones already obtained as no methodology to store which edges have been removed had been implemented. Thus, the algorithm needed to be executed once again but set to obtain a larger number of paths, wasting time and resources in the process.
- Pivot nodes were needed in the graph. Pivot nodes are certain points in the graph that have been selected due to its importance in terms of high influx of routes, and that are chosen to force some combined trips to go through them. Hence, Dijkstra's approach uncertainty due to the random removal of edges could be reduced knowing the paths obtained were going to travers certain nodes were routes are known to already exist.

Therefore, this project seeks to avoid all these difficulties and provide a solution for all the problems confronted so far. A thorough research on the different available and applicable algorithms is needed in the first place. After analyzing the advantages and disadvantages of each one of these algorithms, their suitability for the company's system structure and preference must be studied in order to estimate the effects its development and execution may have.

The main aspects the new algorithm must include are speed and effectiveness in finding several combined routes between two nodes. After the research that will conclude with the selection of the most suitable algorithm, the next step would be its implementation in the system in the most efficient way. The overall result would be the provision of more information to the user in less time but also with more accurate data than before and with absolutely no danger for the system operation.

Last but not least, the development of this project may have multiple future applications. For instance, the possibility of finding more than one solution in a fast and effective way may be of great interest to any route-calculation application or system whose main goal is to provide the best and most accurate information to the user. Finding and developing such algorithm could mean the extension of its capabilities to other fields and applications different from the current ones.

All in all, all these aspects are what render this project complete and challenging enough to make it attractive for further study and development in its applicability in different grounds and fields of expertise.

Chapter 2. RESOURCES DESCRIPTION

The main resource used in this project is the programming language PHP version 7.2.2 [71]. Baolau's web page has been written in the same language and it will be the same used for the implementation and development of the new algorithm.

The tests applied on the algorithm's functioning will be performed locally in order to avoid affecting the real search structure in the server. Consequently, an Apache local server will be employed to execute the code while in development and testing, so the new features and modifications applied to the code can be checked without the risk of endangering the real operations.

The Apache server connection is provided by MAMP [72], a software tool designed specifically for Mac for this purpose. MAMP also includes a link to a MySQL [73] server that provides a connection to the database. This database is stored locally and previously populated with the chosen tables and fields selected from the real databases.

Baolau's successful activities depend on the coordinated work of its employees. Anyone of them can introduce changes to the code, adding and removing lines depending on the task at hand. In order to ensure this process is carried out smoothly and without the risk of overwriting legitimate code, a web-based repository called Bitbucket [74] is used. This service allows to upload the changes made in the code locally only after the ones applied before by one of the parties has already been integrated. Also, different branches can be opened in order to merge gradually the changes to the main branch and make all this process neat and simple.

On the other hand, Sourcetree [75]. is the interface that allows to work with Bitbucket's version-control service.

Chapter 3. STATE OF THE ART

Since a few years, a myriad of web-based applications based on search engines have been developed. Several are focused on reservations such as Booking, Trivago or Kayak, others provide flight booking like Skyscanner, Momondo or Google Flights, and so on. Baolau is included in this last classification, and as such, is in need of an algorithm that provides the different connections when a direct solution is not possible.

Baolau, like the other companies mentioned above, bases its success on expanding to other regions and countries. This implies an increase in the information and data to be managed, and so, the algorithm receiving the increasing material needs to be able to handle it as well as the complexity of the input received.

An algorithm is a computational procedure which, taking a series of input values, is capable of obtaining output data following certain computational steps [1]. Algorithms existed much earlier than the development of computers that can now easily execute them. As a result, computers have implied an extended generation of algorithms now more complex and effective than before.

However, Dijkstra, which was developed in 1956 [76], is said to be used in most applications that find combined routes between two points. These include geographical maps, telephone work and also in IP routing [59]. Its simplicity and easy implementation has allowed the algorithm to become one of the most widely used around the world for route-calculation problems. Baolau chose this algorithm to calculate the combined routes that connect two cities or regions and is now facing problems due to some of the algorithm's limitations.

Apart from Dijkstra, there are several options of route calculation algorithms that could be applied to the problem at hand. They can be classified based on different criteria according to their implementation, the paradigm design they deal with, the optimization of their solution or their complexity [77]. Taking into account the characteristics of the problem that

needs to be dealt with, only a certain type of these algorithms will be considered in this work. These are included in graph, greedy, recursive, search and dynamic programming algorithms.

Dijkstra is a graph algorithm as it works inside a graph conformed of nodes and edges. Johnson's algorithm can also be included in this category. Dijkstra is also classified as a greedy algorithm that chooses the best solution in each internal iteration with hopes of finding the optimal solution at the end [81]. It can also be considered a recursive algorithm [82] as its executions involves the repetition of several procedures until a condition is met.

Search algorithms also include this recursive feature, as they must perform the same operations in each node in order to obtain the next node to analyze, until a final solution is reached. Breadth-first search and Depth-First search are two examples of search algorithms [68].

Dynamic programming algorithms [24] also incorporate a recursive behavior as their procedure for finding the optimal solution consists on dividing the main problem at hand in different sub-problems, and using in each one of them information extracted from the previous solutions. Floyd-Warshall, Bellman-Ford and A* are examples of this type of algorithms.

All of the algorithms mentioned above have a common feature: they provide just one path as a solution. There is another type of algorithms that have not been mentioned before and these are the k-shortest path algorithms [38]. As their category name indicates, these algorithms provide not only one solution but k solutions so they fall into a completely different group of algorithms. Yen [26], Eppstein [3] or Iterative Enumeration Algorithm (IEA) [2] are examples classified as such.

In particular, IEA can be applied in the denominated *transportation networks* [9]. These are networks oriented to time-dependent transport where timetables, arrival and departure times are inherent to the routes, such as road networks or timetable networks of public transportation. These networks are usually too large for Dijkstra's algorithm, whose

execution takes up too much time. Other algorithms seem to be better suited, but still they only work for positive and real-weighted edges in a graph. However, there are real-life cases in which problems are not that simple: edges are time-dependent, there are road constrictions or multiple source and destination nodes to be taken into account [8].

There are many techniques to overcome these difficulties [11]. For instance, Contraction Hierarchies (HC) work to contract the graph in a set of nodes in order to simplify the problem instead of performing all operations with all the nodes, which would take a lot of time a resources. Multi-Level Overlays is a method that precomputes solutions to the problem in the form of partial paths to the destination. This technique is based on the idea that many long-distance share large sub-paths. On the other hand, Hub Labeling [13] is a method similar to IEA in that they employ labels attached to each node that store information about the distance separating them, among other pieces of information. The main difference is that Hub Labeling is more oriented to broad graphs with large quantities of nodes and edges.

All in all, transportation networks algorithms are better suited for large graphs where the amount of information needed to be stored far exceeds the one stocked by simpler graphs. For instance, finding the shortest path between two points by car where the possibilities of traversing the city's streets are plentiful. The same does not happen for graphs such as the one Baolau owns. They are in charge of providing connection between discrete and fixed points defined by the providers with whom they are partnered. Therefore, the nodes and edges to be considered and entered into the algorithm are far less and their connection less complex than that of a transportation network. What Baolau's graph and that of transportation networks may have in common is the necessity of adjusting their routes to time dependent lines and connections that affect the different links between nodes.

As a result of this simplified analysis of the state of the art regarding route-calculation algorithms, it can be concluded that the research and study of this project must be centered in simple but efficient algorithms, instead of those too complex and more oriented to far more elaborate graphs, such as those classified for transportation networks. Chapter 5. will be centered in presenting a thorough analysis on the possible algorithms to adopt.

Chapter 4. PROJECT DEFINITION

4.1 JUSTIFICATION

Baolau is a growing enterprise who seeks to gradually expand its business to different regions and countries. As the graph conformed by the cities, territories and routes interconnecting them increments in size and complexity, the algorithm that calculates different paths between two points in the mesh (Dijkstra), fails to operate fast and efficiently.

This project aims to conduct the research necessary to select the algorithm among the ones mentioned in Chapter 3. , most suitable for the company. This algorithm must be able to overcome the difficulties Dijkstra's algorithm was facing due to the increasing graph's intricacy as well as provide new functionalities that were previously prevented from being adopted due to Dijkstra's limitations.

Just as Baolau keeps expanding, so have other companies and search platforms who have included the possibility of obtaining more than one result when finding the shortest route between two points in a graph. Consequently, this has been considered one essential feature the new algorithm must be able to acquire. Additionally, it has been determined that it is in the user's best interest to have as many route interconnection possibilities at his disposal as possible. This leads to consider which characteristics the algorithm must include in order to be able to keep loading new routes even if some have already been displayed.

All in all, the development and implementation of this new algorithm into the company's system can eventually provide considerable changes that would become an important boost for the company's success. The system will no longer be in danger of failing and being rendered inaccessible due to an over-consumption of resources by an outdated algorithm. Moreover, new features will be included into the user's search experience that will increase its appeal and affect positively to its expansion to more users and even more countries and areas.

4.2 OBJECTIVES

The main objective of this project is to modify the route-calculation procedure so searching for combined results between two points in the graph can become an efficient process. This procedure will be constituted by two main blocks: finding a new algorithm that substitutes the one currently at use and handling the routes database as proficiently as possible.

The first block will imply the analysis, development and implementation of the algorithm whose characteristics recommend it as the optimal candidate for substituting the current algorithm in use in Baolau's search engine: Dijkstra.

For this, an exhaustive and extensive research about the algorithms presented in Chapter 3. needs to be carried out. Furthermore, a thorough study must be performed on each one of them in order to determine which one will be the most suitable to be included into the company's system.

Once the algorithm has been selected, it will be developed and implemented using the web page's programming language PHP. Afterwards, its performance needs to be measured and compared to that of Dijkstra in order to evaluate the effectiveness and operation advantages over those of the previous algorithm in use.

If the results are acceptable enough, the next step consists con adding the incorporating features the company seeks to include, so the user's searching experience is enhanced in the best possible way. One of this features is the possibility of loading more results once some of them have already been displayed. Also some search filters are intended to be added regarding time, prices, waiting times and the trip's number of steps. This would imply modifying the algorithm so it can work with this new arrangement.

For the second block mentioned above, that of accessing the available routes, a new query needs to be designed so that not only time is saved when retrieving the routes with which the algorithm is going to work, but also to obtain as many routes as feasible so all possible routes connecting two points can be procured and passed on to the algorithm.

The combination of these two accomplishments will provide an entirely new search mechanism that, while remaining completely transparent to the user, its arrangement will imply important savings in time and resources.

4.3 METHODOLOGY

The first task to be performed in this project is the research on the algorithms that could become potential substitutes of Dijkstra. This research will comprise of a rigorous study on the algorithm's characteristics and the advantages and disadvantages its application would bring into the company's operation.

Once the algorithm has been chosen, it will be implemented in PHP and tested on a simplistic scenario with non-time-dependent routes and sparse nodes. If the results are satisfactory enough, then the algorithm will be tested on more complex graphs and with time-dependent routes until its degree of development allows to apply it on the real graph with the actual routes used in the company.

The algorithm must be modified each time so it can adjust to the scenarios at hand until it is completely accommodated to the company's data structure (nodes and routes). Once the algorithm has been proven to work correctly with the original data structure employed by the company, the next steps consists on reformulating the query that will change the access to the routes table and therefore, provide an entirely new set of routes passed on to the algorithm. On its part, the algorithm must be modified again to assimilate these changes and still work correctly.

Finally, when this process has been finished and successfully accomplished, the additional features thought to make the user experience more friendly and easy to manage (filters and loading of more results) will be developed and integrated into the system.

4.4 *PLANNING AND ECONOMIC ESTIMATION*

This project has been developed in an internship in Baolau. The duration of the internship as well, as that of the project itself, has been of 5 months, from January 5th 2018 to June 5th 2018. The procedure followed has been explained in Section 4.3 and is now represented in Table 1.

Months / Activities	January				February				March				April				May				June			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Research																								
Coding																								
- Simple graph (4 nodes)																								
- Simple graph (cycles)																								
- Adding time-dependency																								
- Reduced database																								
- Real database																								
- Adding modifications to code																								
Modified Access to Database																								
- Query design																								
- Implementation																								
- Modifications																								
- Analysis of results																								
Interface modifications																								
- Design																								
- Implementation																								
- Modifications																								
- Analysis of results																								
Documentation																								

Table 1. Project time line.

The project economic estimation results to be straightforward as all tools used are open source and the licenses employed, free.

The calculations required by the algorithm are not too many nor too complex, just repetitive. Therefore, a mid-range computer has been sufficient when implementing the algorithm. The

average price of this computer is around 1000€. The model is a MacBook Pro with a 2,7 GHz Intel Core i5 microprocessor. It uses 8 GB 1867 MHz DDR3 of memory.

On the other hand, the working hours during this five-month internship would cost around 6000€, conforming the majority of the budget.

Taking into account that tools such as MAMP, Sourcetree, Bitbucket or PHP are completely free and therefore its used can be excluded from this economic estimation. Table 2 presents a summary of the project overall cost.

Element	Estimated cost
Computer	1000€
Working hours	6000€
Software Tools	0€
Total	7000€

Table 2. Project cost estimation.

Chapter 5. RESEARCH ON ALGORITHMS

An algorithm is a series of specifications that indicate a set of steps and the order in which they need to be executed in order to reach a certain solution, solving this way the problem at hand. Depending on the problem and the variables to take into account, there are many different ways to solve a problem, leading to a wide range of algorithms available to find the solution. But first of all, it is necessary to present the problem at hand.

5.1 ALGORITHM CLASSIFICATION

A list of different algorithms classes is presented in the table below, providing a short description of each one of them. They are organized by implementation, design paradigm, optimization and complexity (10).

Algorithms	Examples	
By implementation		
Recursive	The algorithm repeats certain directives obtaining a result when a certain condition is reached	Dijkstra, Dynamic Programming...
Logical	Algorithms controlled by logical inference captured in computational axioms	Abductive logic programming, constraint logic programming...
Serial, parallel or distributed	Depending on how the instructions are executed: one at a time, several simultaneously in the same or different hardware architecture, respectively	Newton's method, Paxos algorithm...
Deterministic/non-deterministic	Depending on which type of decisions are taken: an exact one or one made by guesses (heuristically at best), respectively	Gompertz curve, PCA...

Exact or approximate	Depending on the solutions given at the end	Lenstra, Shmoys, Tardos...
Quantum	Used in quantum computation	Quantum computation
By Design Paradigm		
Brute force	It consists on trying every possible solution	Brute-force sear
Divide and conquer	These algorithms splits the problem in smaller sections, recursively sometimes, that can be solved more easily	Divide and conquer algorithm, merge sorting
Search	These are performed in graphs where rules to move around are specified	Breadth-First-Search, Depth-First-Search...
Randomized	The algorithm makes choices in a random way	Monte Carlo, Las Vegas
Reduction of complexity	It involves transforming the problem so simpler algorithms can be used to solve it	Selection algorithm, Asymptotically optimal algorithms...
Optimization algorithms		
Linear programming	Used to solve linear problems in which the problem's own constraints can be used to obtain the optimal solutions	Simplex algorithm, maximum flow algorithm...
Dynamic programming	The optimal solution is obtained by optimal solutions of sub-problems procured from the main one. In this case, the sub-problems are related whereas in Divide and Conquer, they are not	Floyd-Warshall, Bellman-Ford Algorithm, A*...
Greedy algorithms	They also play with substructures but in this case of the solution instead of the problem (see dynamic programming)	Dijkstra, Huffman Coding, Kruskal's and Prim's Minimum Spanning Tree Algorithm

The heuristic method	They find the optimal solution closest to the actual one, as finding the real would be unfeasible	Genetic algorithms, simulated annealing...
By complexity: time to run		
<ul style="list-style-type: none"> • Constant time • Linear time. • Logarithmic time. • Polynomial time. • Exponential time. 		Binary search algorithm, bubble sort...

Table 3. Algorithm classification, definition and examples.

5.2 O-BIG NOTATION

When comparing different algorithms' performance, a specific notation is used: O-Big notation. This notation defines the time-complexity of an algorithm. It is a mathematical notation that indicates the limiting performance of a function when the argument tends to infinity. The best way to understand the meaning of this notation is with some examples [79].

- $O(1)$ defines an algorithm whose execution is always the same independently of the input size.
- $O(N)$ describes an algorithm whose execution time will grow linearly depending on the input size.
- $O(N^2)$ refers to an algorithm that takes a time proportional to the square of the input size. The same happens with $O(N^3)$, $O(N^4)$, etc.
- $O(\log N)$ is applied to algorithms whose time complexity that has a peak in the beginning and slowly flattens out, as it happens with the binary search.

All in all, O-Bog notation is used to asymptotically bound the time growth of an algorithm depending on the input size [80].

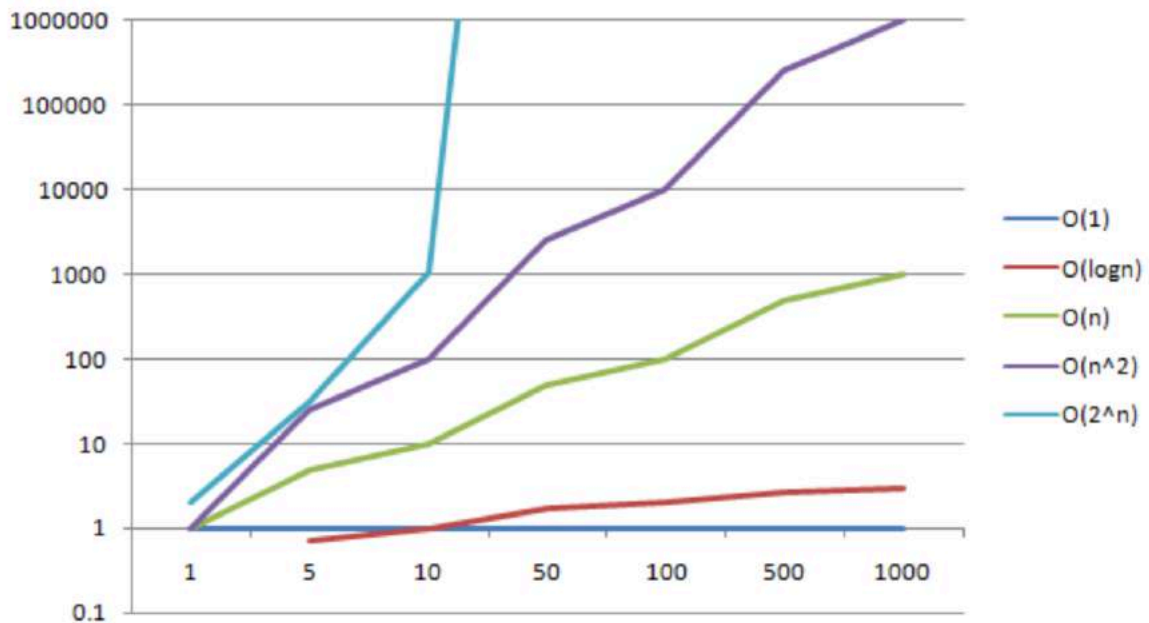


Figure 1. Algorithms' time complexity in O-Big Notation [80].

5.3 DESIGN

Baolau possesses a database with nodes and routes or edges that interconnect them in structure called a graph. The main objective is, as clarified before, to find the shortest paths from one node to another that the user has already specified. Given these conditions, there are some algorithms incompatible with this layout that can be discarded since the beginning. Due to the accuracy needed in the results and the large amount of information that has to be handled, the randomized or brute force algorithms are completely left out of consideration.

As Baolau looks for different optimal solutions to represent the various ways to reach a certain destination given different nodes and routes available, the basic problem on which the algorithm must work is this type of graph. Also, routes are only available at fixed dates and hours, which another feature that must be taken into account. Figure 2 shows the concept of a graph.

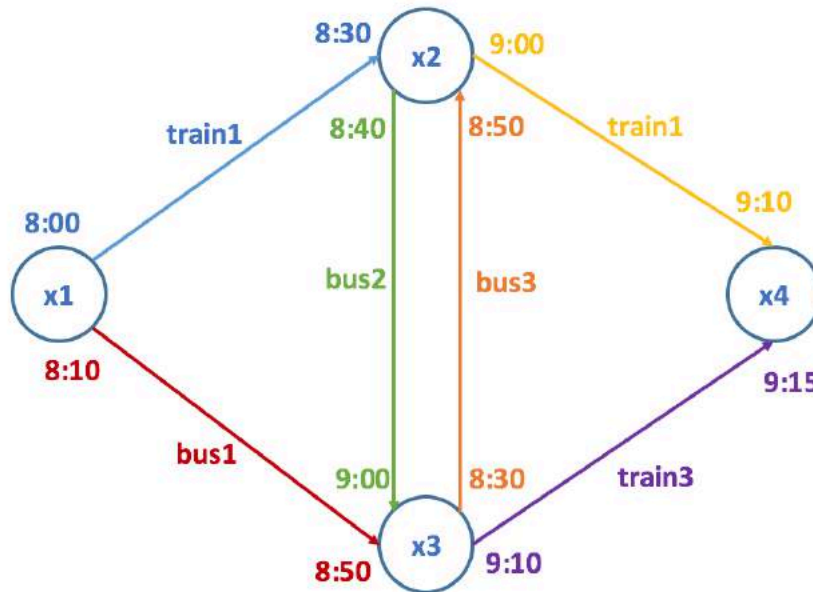


Figure 2. Simplified example of a nodes graph.

Therefore, the graph with which Baolau works has the following characteristics:

- It is directed. It means that the edges that connect the different nodes have a direction specified.
- It is positive weighted. Each one of the edges has an associated positive weight that is a combination of the time that it takes to travel through that edge, the price and the waiting time, although these specifications will be analyzed deeper later on.

Considering then the main problem at hand in Baolau, only some types of algorithms had been chosen for a deeper analysis: the recursive, search, dynamic programming and greedy algorithms.

- **Recursive algorithms**

A recursive algorithm consist on the iterative execution of instructions until a certain condition is met or the solution has been found. These are very interesting in graph algorithms where rules need to be applied if specific conditions are met, advancing throughout the whole graph until the destination is found.

A lot of the algorithms that will be studied have this characteristic, such as Depth-first search or the Iterative Enumeration Algorithm.

- **Search algorithms**

These algorithms are used to retrieve some information stored in a data structure or problem domain, such a search tree or a linked list. This term is also used for graph exploration techniques, such as Depth-First and Breath-first search, which also apply recursive techniques. In fact, there are a great amount of graph algorithms apart from these two, considered elementary graph algorithms.

- **Dynamic Programming**

Dynamic programming algorithms find the optimal solution to a problem by obtaining optimal solutions to parts of the problem or sub-problems and using the intermediate results as input for the following iterations. There is an important difference between dynamic programming algorithms and divide and conquer algorithms, as these last ones obtain intermediary solutions that are independent from each other.

Dynamic programming counts also with an important contribution of recursive mechanisms, but the main difference lies in the fact that dynamic programming caches or stores the results of the recursive calls, reducing the complexity of the problem. This is only possible to apply in problems that count with substructures dependent on each other and where repetition is required to obtain an optimal solution. Floyd–Warshall, Bellman–Ford Algorithm and A* algorithm are examples of dynamic programming.

- **Greedy algorithms**

Greedy algorithms work similarly to dynamic programming algorithms. The main distinction is that the first ones work on solutions of substructures of previous solutions and not of sub-problems, as dynamic programming algorithms do. This implies that the algorithms start obtaining one solution and make modifications to the structure in order to obtain better results. In this classification Dijkstra, Huffman Tree, Kruskal, or Prim can be found.

- **Graph algorithms**

Given the problem the company needs to resolve, this project is going to be focused on the so-called *graph algorithms*, that find a solution for connecting two nodes in a graph. Graph algorithms include some belonging to the previous classes such as search and greedy algorithms. This will be explained later more in detail.

Minimum Expanding Trees and algorithms that find the shortest path between two nodes in a graph also belong to these group. In fact, an additional classification can be applied for these last ones.

- *Single-source shortest paths algorithms.*

They find the shortest path from one single node to the rest of them in the graph. Dijkstra and Bellman-Ford provide this type of solutions.

- *Single-destination shortest paths algorithm.*

In this case, paths are found from each node to a single defined one in the graph. An example of these algorithms is the Multiple Sources Single-Destination (MSSD) algorithm.

- *All-pairs shortest path algorithm.*

The algorithms find the shortest path between all pair of nodes in the graph. Floyd–Warshall algorithm, Seidel's algorithm, Williams, Pettie & Ramachandran and Thorup can be considered examples of this type.

- *Single pair shortest path algorithm.*

Only the shortest solution between two fixed nodes in the graph is found. A* algorithm is an example of these.

Also, due to the approach dynamic programming algorithms take, these are also interesting to analyze in this project.

The following section will present the current algorithm in use in the company's web and its main characteristics, as well as the problems its application comes with.

5.4 CURRENT ALGORITHM IN USE: DIJKSTRA

Dijkstra	
Classification	Greedy/Graph Algorithm
Solutions	Single-Source Shortest Path
Graph	Directed and Weighted
Time	$O(V ^2)$

Table 4. Dijkstra's specifications.

Edsger Dijkstra (1930-2002) was born in the Netherlands where he was employed as a university research professor till 1984 [76], when he moved to the United States. During the so called Structured Programming movement, which he set up himself, he developed a methodology for building computer programs based on nested loops and sequences that avoided the arbitrary jumps wide extended by then.

It was in 1959 when his famous Dijkstra algorithm was first published [78]. The algorithm provides the shortest path from a single source to all the different nodes in the same graph in a single execution. The functioning consists on building up a priority queue that stores in each node the accumulated weight a specific path has brought from the start point. Then the next node to be visited will be the one with the smallest accumulated weight.

5.4.1 ORIGINAL IMPLEMENTATION

At the beginning, the start point has an accumulated weight of 0 as the distance to itself is null. Then, all the nodes that are directly connected to the start point are updated with the weight of the edge that connects them to it. The node with the smallest accumulated weight will be picked and considered visited so it is not analyzed again. All the nodes connected with a single step to this node will be studied, updating their accumulated weight. The one with the smallest weight will be marked as visited and the process continues until all nodes have been visited [31].

In the company a new feature to the algorithm was added so it could be functional with timetables and routes fixed by the time of departure. For that to be possible, the time that it takes to reach each node by each one of the routes is stored. Consequently, only routes or edges with a departure time later than the arrival time at each node are considered as potential future paths.

5.4.1.1 Pseudocode

In the following lines, the pseudocode for Dijkstra algorithm as it was used in the company is presented.

```

function Dijkstra(G,source,destination):
  distance[source] = ∞;
  Q = PriorityQueue;
  Q.queue(source,distance[source]);           //The original node is inserted in the queue

  while(size(Q) > 0)
    v = Q.dequeue();                          //The last node inserted is dequeued from Q
    visited[v] = true;

    for all neighbours w of v in G:          //Every neighbor of v is analyzed
      distance[w] = distance[v] + weight(v,w); //The weight is updated
      newArrival = arrival[v] + duration;     //The arrival at the end of e is the time at v
                                              //plus the duration of the trip

      if(distance[w] < distance[v])
        previous[w] = v;                     //To retrace the path from the destination
        arrival[w] = newArrival;
        Q.queue(w,distance[w]);

      end if

    end for

    getPath(destination);                    //Using array previous(destination)

  end while
  
```

Pseudocode 1. Dijkstra's Algorithm.

5.4.1.2 Operation

Let us consider the following graph (Figure 3) with 5 nodes and the edges connecting them with their weights.

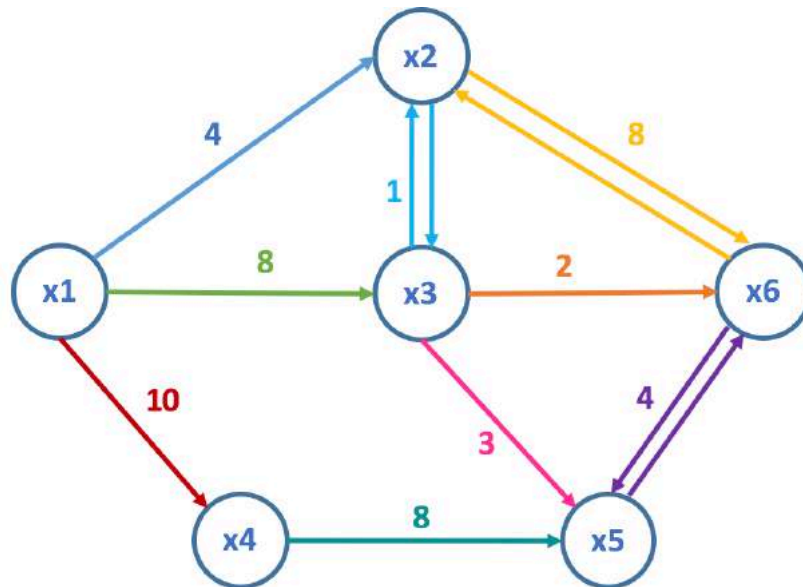


Figure 3. Example of a graph with its nodes and edges.

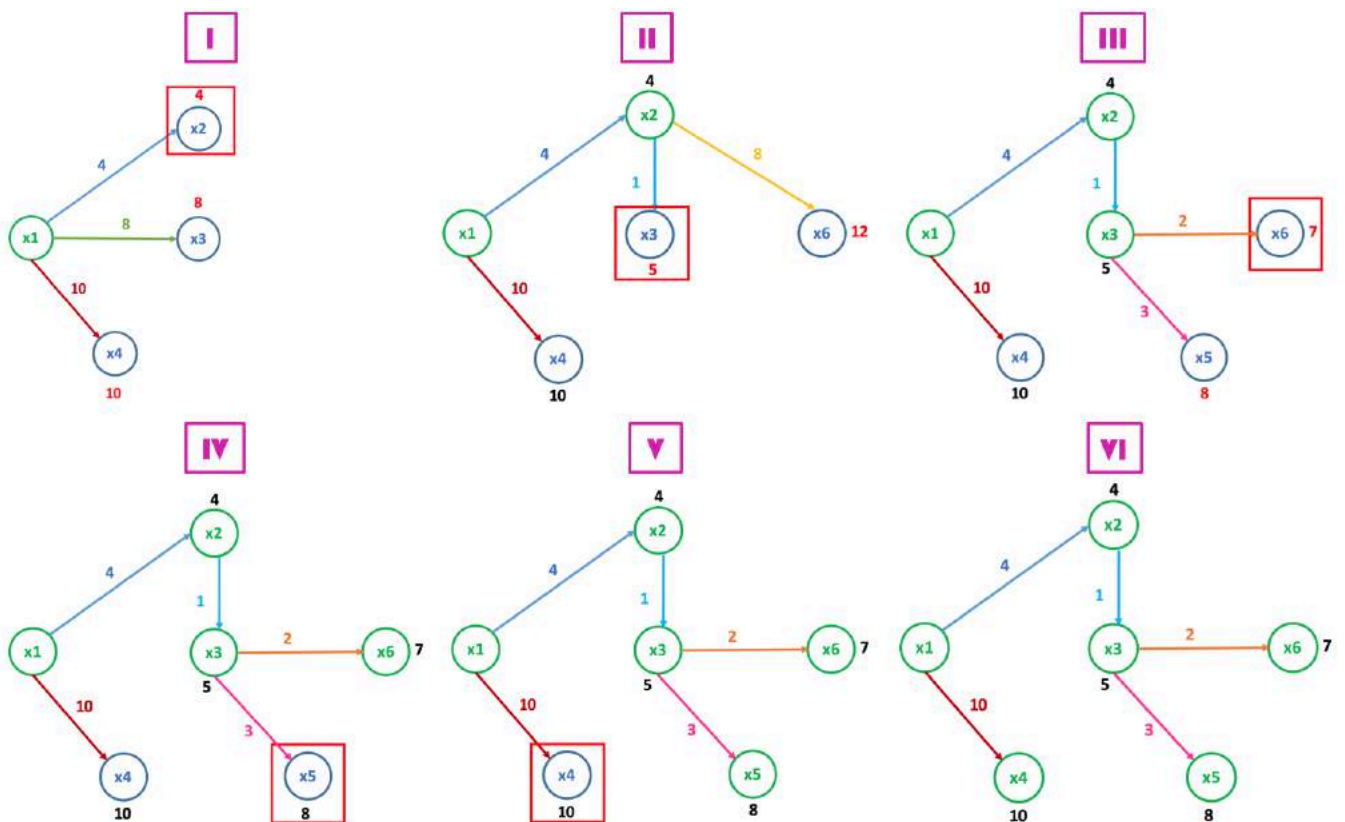


Figure 4. Steps for Dijkstra Algorithm.

Figure 4 shows the process Dijkstra algorithm follows. In the first place, x_1 is marked as visited as it is the start node. Then all nodes directly connected to x_1 are updated with the weight of the connection edge (I). x_2 is chosen as the next node to be analyzed thanks to its lowest accumulated weight. From this node, x_3 and x_6 can be accessed. As the accumulated weight to x_3 going through x_2 is lower than going directly from x_1 , the edge from x_1 to x_3 is no longer considered (II). The same happens in III when the accumulated weight from x_1 to x_6 going through x_2 and x_3 is lower than going only through x_2 . The graph shown in VI shows the shortest path from x_1 to any other node in the graph.

5.4.1.3 Time complexity

Following the Bog-O notation described early, and supposing a graph of V nodes and E vertexes, the time complexity of this algorithm can be simplified to:

$$O(|V|^2)$$

In fact, the actual complexity depends on the nature of the priority queue Q applied in the algorithm for storing the vertexes. In the existing implementation of the algorithm a simple priority queue is used, which gives a time complexity of:

$$O(|E| + |V|^2) \cong O(|V|^2)$$

Formula 1. Dijkstra's time complexity.

5.4.2 MODIFIED IMPLEMENTATION: THE REMOVAL OF EDGES

Given this functioning, Dijkstra only procures a single path to a certain destination node. In case more paths are wanted to be found, the algorithm needs to be run again but eliminating one or more edges, so the algorithm is forced to pick a different path. This is Yen's algorithm philosophy: to use a single-source shortest path algorithm and apply it repeatedly removing edges each time. In Figure 5 an example is shown on how the algorithm would be applied to a scenario where an edge has been removed. Figure 6 shows the algorithms procedure.

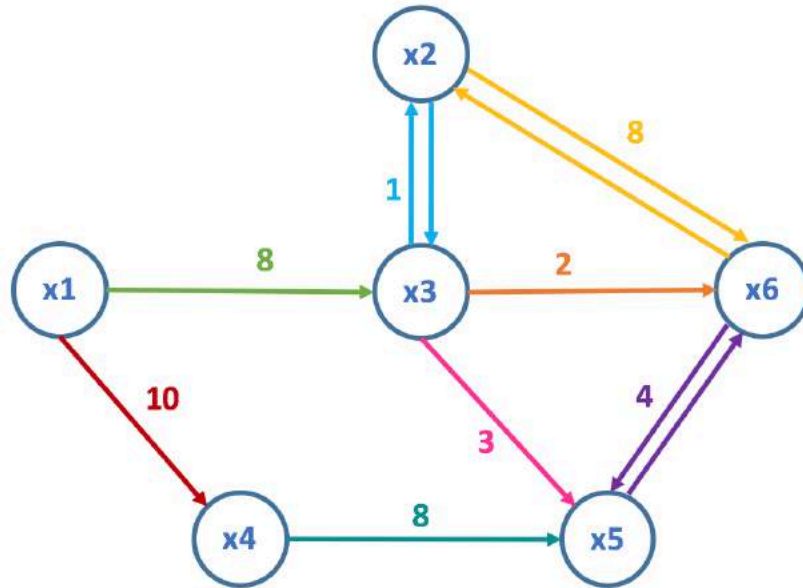


Figure 5. Example of a graph with and edge removed.

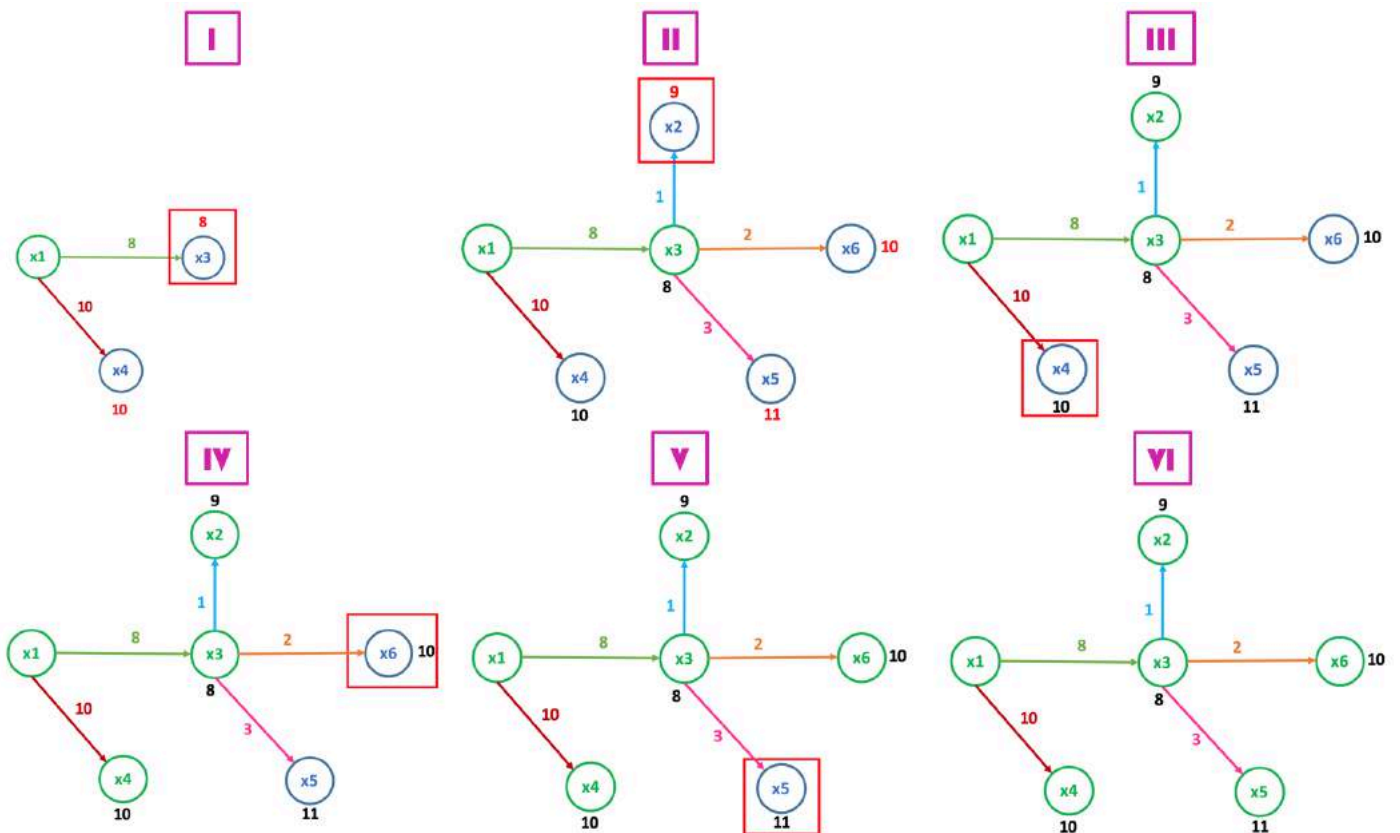


Figure 6. Steps for Dijkstra Algorithm eliminating edges.

In this case, the first node that is visited after x_1 is x_3 instead of x_2 . Following the same steps as before, it can be seen that the shortest path to the rest of the nodes are different and with a higher accumulated weight. If this process is executed several more times, each time removing a different edge, differing results will be obtained.

This is basically the process the company is following in order to obtain different results when a user inputs an origin point and a destination. Even though this is a very extended practice when looking for more than one path between two nodes in a graph when applying a single-source shortest-path search algorithm, it comes with some unavoidable problems.

- As the network increases its dimensions, the number of nodes and edges to be considered increases, making the **algorithm more difficult to run**. This affects directly the time that it takes to run the algorithm only just once. If it needs to be called more than once, the effects are very severe.
- As the company has been recently expanding to more areas and countries, the mesh of nodes and edges has considerably increased. Executing the algorithm only once **would take too much time**, and the subsequently calls would have a great impact on the system, driving it to a collapse sometimes.
- When removing the edges, no special criteria is followed. This means that the additional results obtained when applying this scheme are **not necessarily** the next **optimal** ones, that is, the next shortest or less expensive ones. Only when the required amount of results has been procured, then they are independently sorted.

In a general perspective, we can relate to the other problems already commented at the beginning.

- Baolau is facing some difficulties regarding the **time** it takes the algorithm to **execute just once** due to the higher amount of nodes the company is dealing with now.
- The current implementation of the **algorithm requires at least three executions** in order to give more significance to time, price and waiting time in each connection. That is, the edges are modified each one of these three times so the paths obtained are the fastest, cheapest and more convenient, respectively.
- Some **pivot nodes** are required in the current code. Pivot nodes are points some routes are forced to run through due to some characteristics that render them more desirable than others. They are chosen by the company as they believe those routes

are the best ones and do not want to risk the algorithm not showing them. The need to implement these pivots nodes comes from the **uncertainty** of the algorithm, as well as the act of removing random edges.

Given this situation where a lot of obstacles prevent a smooth calculation of different paths between two points in a graph, it is more than necessary to find and implement a new algorithm good enough to avoid these difficulties. In the following subsections, a series of algorithms will be presented and analyzed in order to determine if they are plausible and potential substitutes for Dijkstra's algorithm.

5.4.3 ADVANTAGES AND DISADVANTAGES

Advantages

One of the main advantages of using Dijkstra is the simplicity of its structure and implementation. The use of a priority list simplifies the whole process of finding the shortest path between two given nodes in a graph.

It is one of the most extended algorithms used in a wide range of fields of study around the world. It is applied geographical maps as well as in networks.

It can be modified to obtain more than one result by eliminating some edges in the graph.

Disadvantages

The main disadvantage of this algorithm is the blind search it applies which results in a waste of time and resources when finding the shortest paths between two nodes. In fact, this can be seen as the algorithm calculates the shortest path between the source and any node in the graph, while the optimal solution would be just obtaining a single result.

This algorithm cannot handle negative weights, which can lead to problems in other situations where the path that needs to be found is not the shortest but the one that has the lowest accumulated weight (networks). Specifically, when negative weights are being used, low-weight shortest paths usually have more edges than higher-weight paths.

This features lead to acyclic graphs that prevent the algorithm from finding the correct shortest path.

5.5 ALGORITHMS INCLUDED IN THE RESEARCH

This section provides an introduction to the various algorithms that have been considered as potential candidates to substitute the present one at use: Dijkstra's algorithm. They will be presented following the classification applied in the previous section. Their characteristics will be described and their functioning demonstrated with the same graph with which Dijkstra was displayed. Afterwards, the reasons for considering or not the algorithm as a substitute will be explained. Finally, after all algorithms have been examined, a table will show the comparison between their most important characteristics.

5.5.1 SEARCH ALGORITHMS

First of all, two elementary graph algorithms will be studied: Breadth-first and Depth-First search algorithms. As mentioned before, search algorithms look for extracting information from a certain structure. In this case, they can be applied to graphs. The process they follow is also called graph traversing, which refers to the process of searching through or traversing through the data in a graph [22],[23].

One thing to consider about these two algorithms is that they work with unweighted graphs, that is, the edges do not have any kind of weight. Although this characteristic renders them completely unsuitable for the problem at hand, it is important to analyze their functioning as they are used by other algorithms, such as Eppstein's algorithm. This one will be studied further on (see section 5.5.4.2). Also, it could be possible to use these algorithms if they are modified so they can operate with weights.

Breadth-search and Depth-search algorithm both work recursively. The main difference between them is the type of the solution they offer and how they obtain it.

5.5.1.1 Breadth-First Search Algorithm

Breadth-First Search	
Classification	Search Algorithm
Solutions	Elementary Graph Algorithms
Graph	Directed and Unweighted
Time	$O(V + E)$

Table 5. Breadth-First Search specifications.

This algorithm, also known as BFS, indicates which paths are shortest connecting a node with the rest in a graph. Its procedure consists on analyzing one node and all the neighbors surrounding it, instead its children. This way, it works very similarly to Dijkstra with the exception that the weights are substituted by jumps to the starting node. If a node is reached twice, the value that it will keep will be the first one as it is the result of less jumps.

Pseudocode

The pseudocode for Breadth-First algorithm is specified as follows in the following lines:

```

function BFS (G, source):
  Q = PriorityQueue;
  Q = source;
  visited[source] = true;           //Initial node is marked as visited

  while ( size(Q) > 0)
    v = Q.dequeue( )               //The last node inserted is dequeued from Q

    for all neighbours w of v in G: //Every neighbour of v is analyzed

      if (visited[v]==false)
        Q.enqueue( w )             //Only if the node has not been visited the node is
        visited[w] = true;         considered
      end if

    end for

  end while
  
```

Pseudocode 2. Breadth-First Search Algorithm.

Operation

Considering the initial graph with which Dijkstra was demonstrated, the following sequence shows the functioning of breadth-search algorithm. No weights are shown.

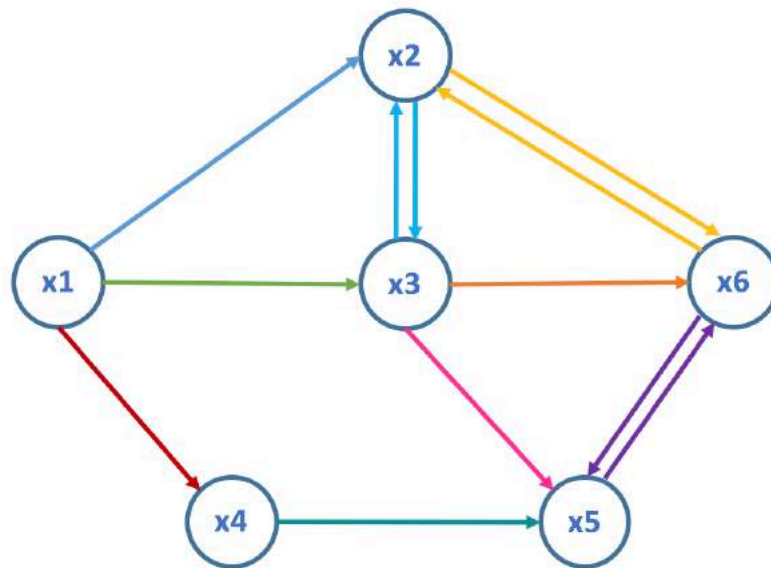


Figure 7. Diagram without weights.

In Figure 8 the procedure followed by the algorithm is shown. In first place, node x1 has a distance 0 from the origin as it is the starting point. Then, x2, x3 and x4 are updated with a distance 1 from x1. X2 is chosen then and x6 is updated to 2 jumps. X3 is not modified because it has a direct access to x1. Then x3 is analyzed and x6 is not modified because the value is the same. However, x5 is visited and updated with 2 jumps. With x4, something similar happens and x5 is not modified. The process continues until all nodes have been selected. The result shows the distance of each one of the nodes to the origin. However, it does not show which is the path that needs to be taken in order reach each one of these nodes.

This path can be known thanks to the tree that is shown next to the graph in Figure 8. The tree represents the children from each node and the order in which they are visited. At the end we have enough information to know what is the distance between the origin and destination point and which is the path that has been followed. For example, from x1 to x6,

the final image in step 6 shows that x_6 is 2 jumps from x_1 and that the path followed to reach it is x_1 - x_2 - x_6 , thanks to the tree besides the graph.

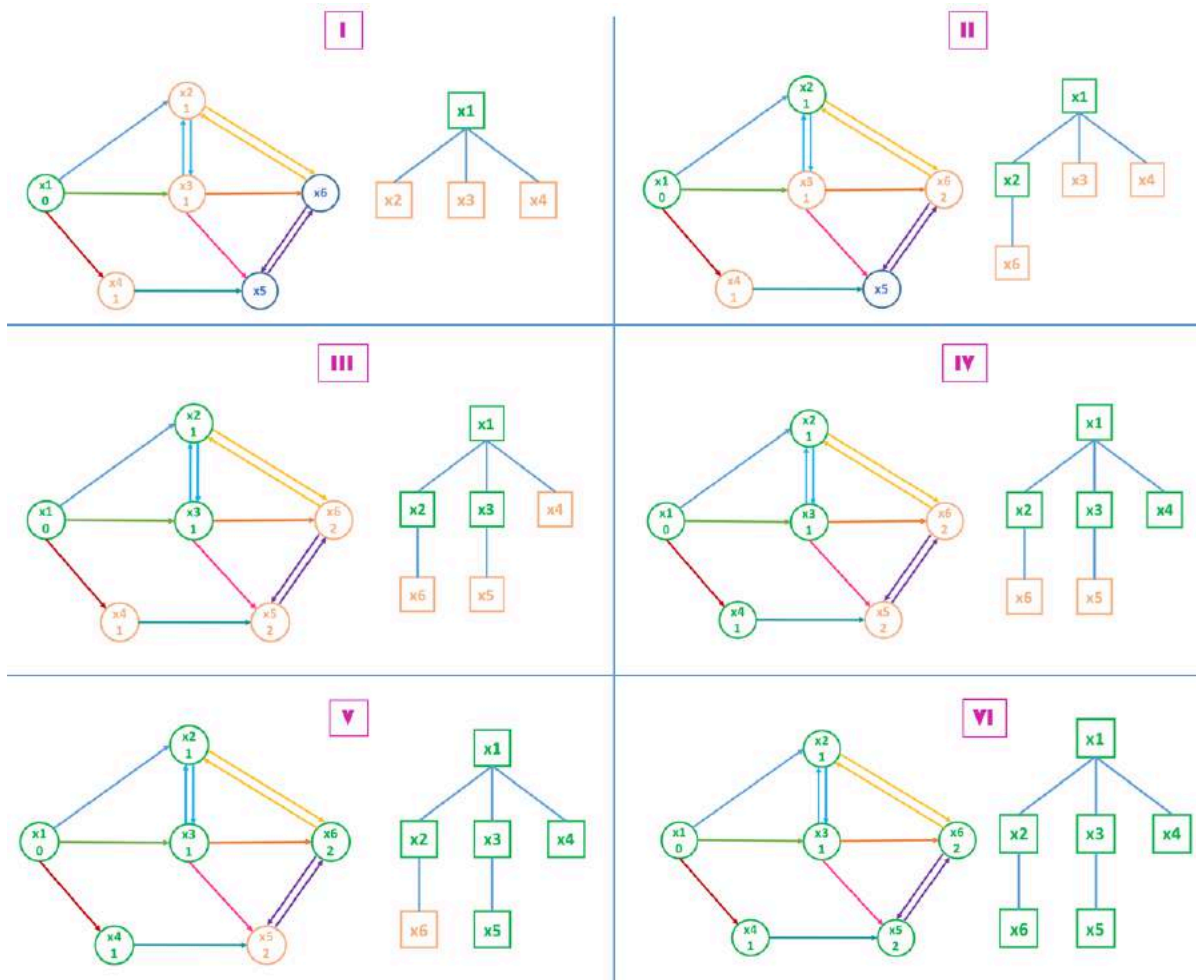


Figure 8. Steps for Breadth-Search Algorithm.

A modification can be applied to the algorithm in order to store the path information as the process takes place. This way the distance to the origin can be known as well as the path needed to be taken. For this, an adjacency list could be used.

An adjacency list is a group of unordered lists that help to represent a finite graph. Supposing a graph with V nodes and E edges, when $|E|$ is much lower than $|V|^2$ then an adjacency list can be used. When $|E|$ is closer to $|V|^2$ then it is better to use an adjacency matrix. This is a matrix whose rows and columns represent graph vertices. If a single graph does not have

loops, then the adjacency matrix has 0 in its diagonal. For undirected graphs, it is symmetric [49].

In this case, an adjacency list can be used as follows.

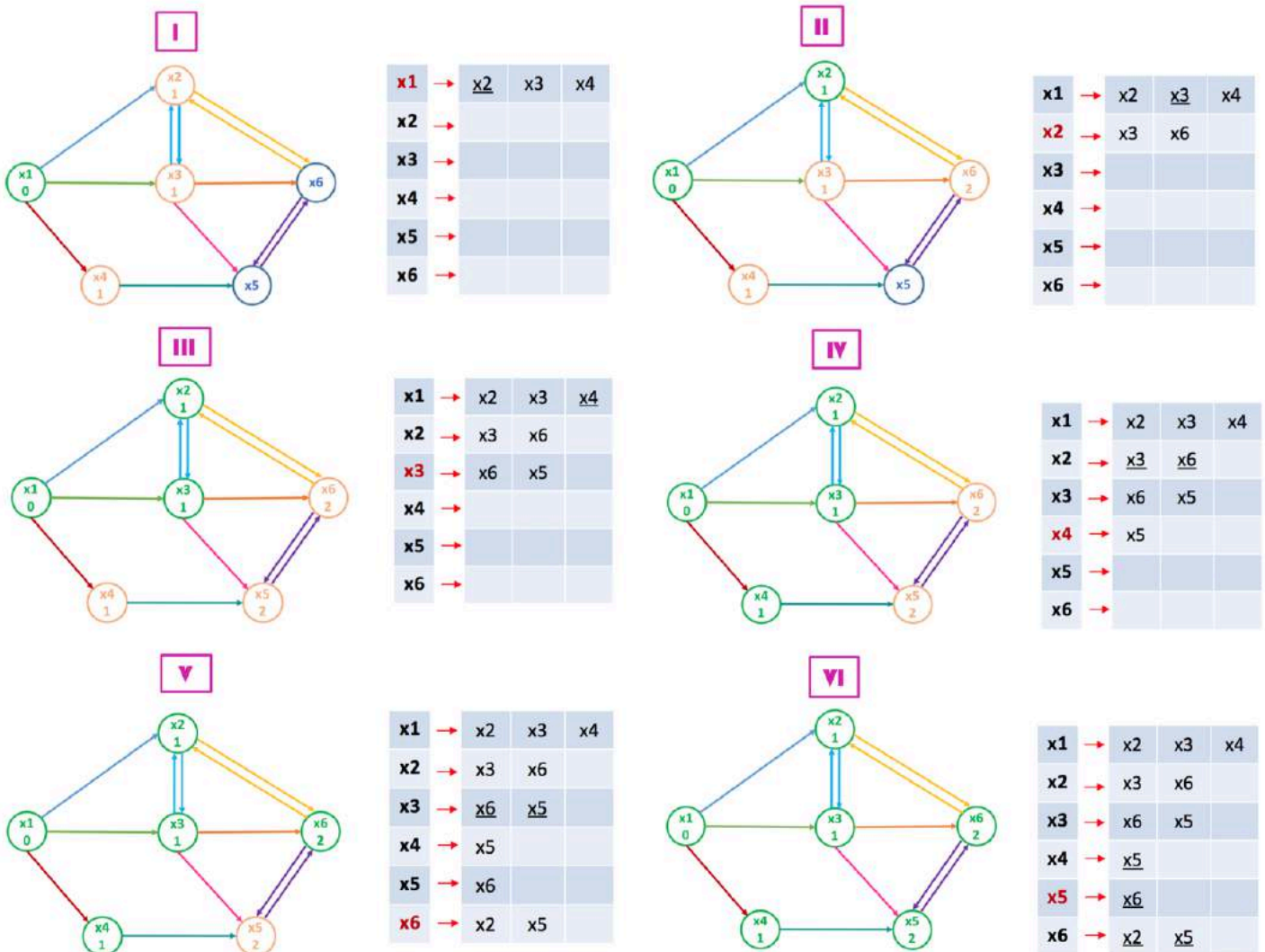


Figure 9. Breadth-Search Algorithm with adjacency list.

As Figure 9 shows, the list is filled for each node with the nodes adjacent to it. It helps to see which node needs to be visited next. For example, for x1, the adjacency list has stored x2, x3 and x4 in that particular order. Then, these nodes need to be visited in that same sequence. The node being analyzed is marked un the first column in red and the next node

to be visited is underlined. If an underlined node has already been visited, then the next underlined node will be the one considered. So, x2 is visited next and its adjacent nodes are stored in the list (x3 and x6). In step IV the next node that is supposed to be visited is x3 but it has already been considered, so the next node will be x6, that is why both are underlined. The process follows until all nodes in the list have been underlined and considered for analysis.

Time complexity

The time complexity depends of the nodes and the edges connecting them. Every node needs to be visited and the number of edges depends on the graph being directed or undirected, being the amount $2|E|$ or $|E|$, respectively.

In any case, the time complexity of this algorithm is covered by:

$$O(|V| + |E|)$$

Formula 2. Time complexity of Breadth-First Search Algorithm.

Advantages and Disadvantages

Advantages

The existence of a solution means the algorithm will always find it. It may not exist, though. If more than one solution exists, then the algorithm will find the one that requires a less number of steps.

Thanks to the explorations of each child instead of going deep into the graph (depth-first search), the algorithm will never get trapped in a never ending path.

Disadvantages

The memory that it requires is one of the main disadvantages of the algorithm, as explained before. This happens because the algorithm saves the nodes it studies in each level, increasing the memory usage.

If the solution is not near enough from the source point, the time consumption of the algorithm can be high.

The algorithm works with unweighted graphs.

5.5.1.2 Depth-First Search Algorithm

Depth-First Search	
Classification	Search Algorithm
Solutions	Elementary Graph Algorithms
Graph	Directed and Unweighted
Time	$O(V + E)$

Table 6. Depth-First Search Specifications.

This algorithm (also known as DFS), like Breadth-First algorithm works with an unweighted graph. The main difference between them is that, while Breadth-First algorithm determines the shortest path between two nodes, Depth-First will dictate if a path exists or not between them. In order to achieve this, the algorithm traverses the graph analyzing a node and its children before reaching out to its neighbors.

Pseudocode

Next the pseudocode for Depth-First search algorithm is presented.

```

function DFS (G,source):
  create S;
  S.push(source);
  visited[source] = true;           //Initial node is marked as visited

  while (size(S) > 0):
    v = S.top()
    S.pop()                         //The last node inserted is taken from S and
    //eliminated from the stack

    for all neighbours w of v in G:
      if(visited[v]==false):
        S.push(w)                   //Only if the node has not been visited the node is
        //considered
        visited[w] = true;
  
```

end if
end for
end while

Pseudocode 3. Depth-First Search Algorithm.

Operation

In the following figures, the functioning of the algorithm can be seen. One feature of this algorithm its heavy reliability on recursion. When a branch is found to be connectionless for two particular nodes, then the beginning of that branch needs to be reconsidered.

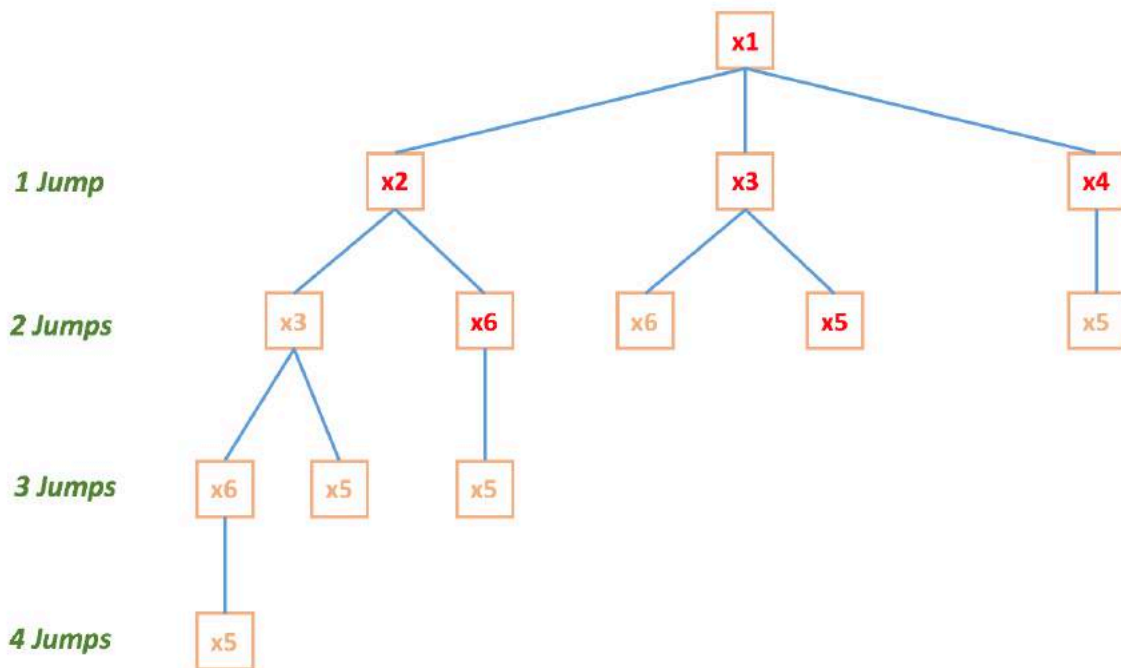


Figure 10. Tree for Depth-First Search algorithm.

In Figure 10, the tree formed using the same graph as in Breadth-First search, is traversed from the top to the bottom, analyzing children first rather than neighbors. All the branches are all the possible ways to reach a certain node without entering into a loop. This way, the shortest path to each one of the nodes can be obtained. In Figure 10 they are marked in red.

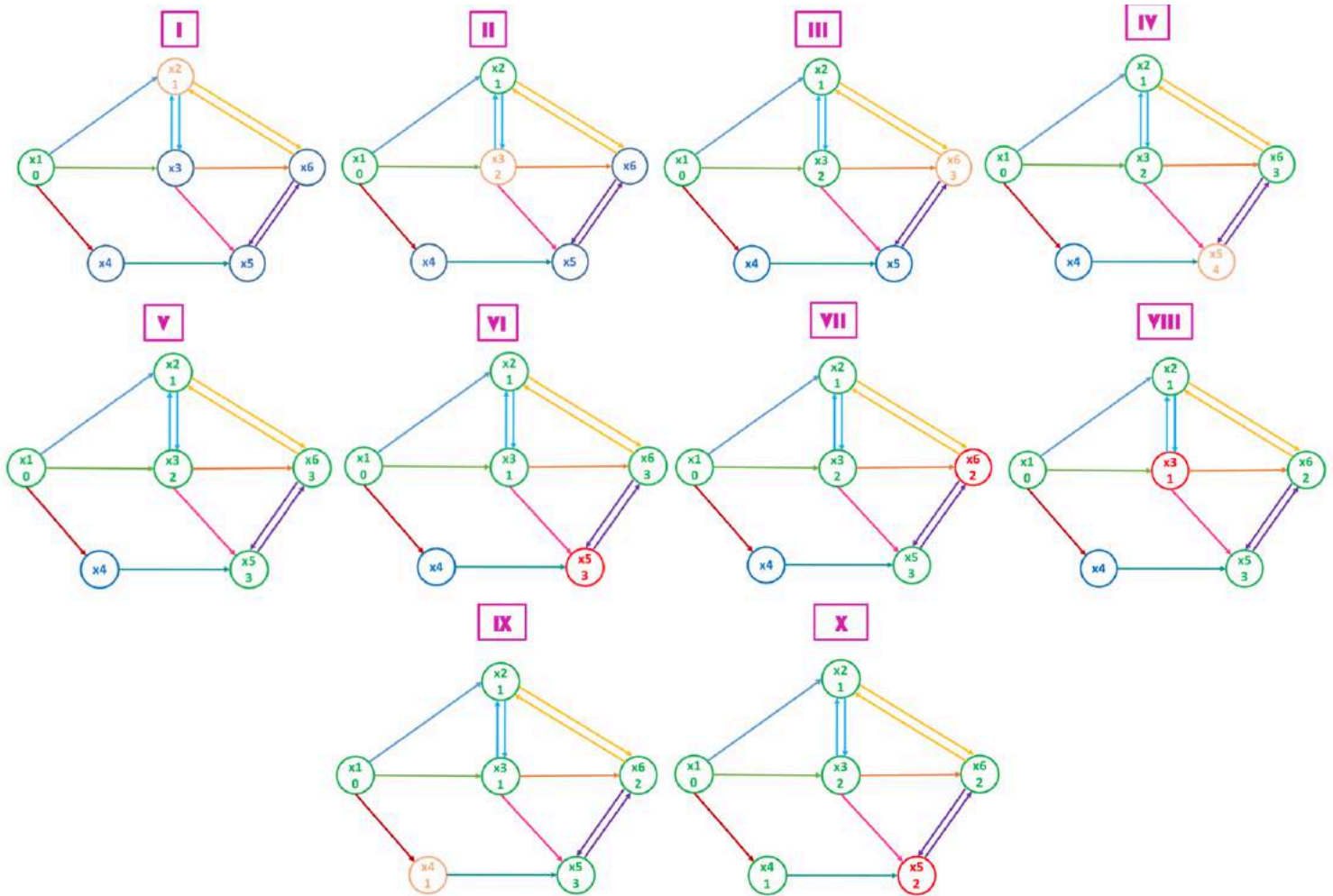


Figure 11. Depth-First Search Algorithm.

Figure 11 follows the previous tree but in the graph being used. As it can be seen, nodes are visited in a chain fashion. Supposing our starting point is x_1 , the next node that is visited will be x_2 . Afterwards, the next node will be one of x_2 's children (x_3 or x_6); x_3 is chosen. Then one of x_3 's children (x_5 or x_6) will be considered and so on until step 5 is reached. Here, an end of the chain is met because the only node that can be visited from x_5 is x_6 , which has already been visited. So now the branch needs to be retraced until a node that has a child that has not been considered yet. Going back to 3 jumps in the tree, x_6 has x_2 as a child but x_2 has already been visited and the path through x_6 is not shorter than the one already been found to x_2 . So the branch keeps being traced backwards to 2 jumps in the tree and x_3 is reached. Here x_3 has a child (x_5) that has already been considered but a shorter

path can be found going directly through x3 than through x6 like before. So x5 is updated with the new distance from x1 (step VI). But now the same end is found so this branch needs to be traced backwards until x2 is found. Apparently no other path can be found to its children that are shorter than the ones already found. In step VII node x4 is visited and in step VIII x5 distance to x1 is updated.

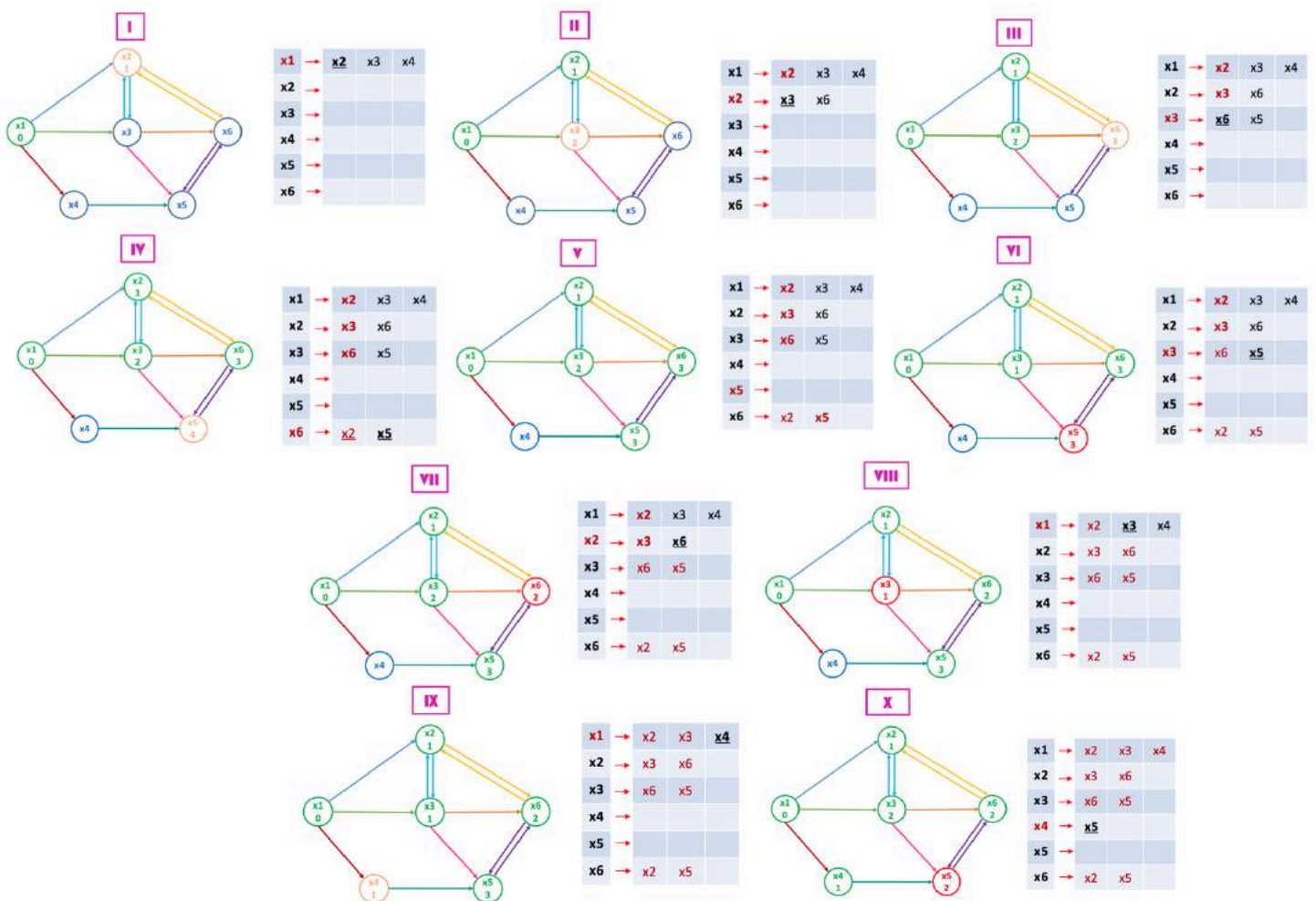


Figure 12. Depth-Search Algorithm with stack.

In order to keep track of the parent of each node to find the shortest path something similar to Breadth-First search can be applied. In this case, instead of using a list or queue, a stack will be more suitable for the algorithm's features. The main difference between a queue and stack is that the first owns a FIFO (First In First Out) quality while a stack gets to be LIFO

(Last In Last Out). And this is the main difference in both pseudocodes from Depth-First and Breadth-First search. In the given graph the contrast between their functioning can be seen and Figure 12 shows how the stack works.

First of all, when a node is considered, it is marked in red in the first column and its children are inserted in the matrix. In this case, nodes in bold mean they are maintained “on-hold”, that is that they are going to be traced back when a dead end is met on the chain. Nodes are marked in red when they have already been considered in the study. In step 5 a dead end has been found because node x6 has already been visited. In step 6 the path that has been followed so far is traced back to the first node that has another child that has not been considered yet, even if it has been visited. In this case, this node is x3 that had, apart from x6, a child in x5. So x5 is updated. This process is carried out until all nodes have been considered and visited.

Time Complexity

The time complexity for Depth-First search is the same as for Breadth-First search:

$$O(|V| + |E|)$$

Formula 3. Time complexity of Depth-First Search Algorithm.

The reason is because in both cases all nodes and edges need to be analyzed even if the followed pattern differs.

Advantages and Disadvantages

Advantages

The memory usage is linear with respect to the number of nodes. This is because only the stack of nodes from the path to the node being analyzed need to be stores instead of all the levels of the tree (Breadth-First search).

If the solution that is being looked for is found relatively soon, then the time and space needed are much lower than in Breadth-First search. This is thanks to the fact this algorithm analyzes the children of a node instead of going through all the levels.

Disadvantages

This algorithm does not guarantee finding the solution even if it exists. And if more than one exists and one has been found, nothing ensures that this solution is the optimal one. This characteristic renders the algorithm practically implausible for applying it to the problem at hand. Also the fact that it can only be applied in unweighted graphs implies that this algorithm cannot be considered as a potential substitute for the actual implementation of Dijkstra.

There is the possibility of the algorithm entering into an infinite loop as, even if the graph is finite, the tree obtained might be infinite. Although there is a solution to this problem: cutting the exploration of a certain branch given a desired level of depth.

5.5.1.3 Comparison between Breadth-First and Depth-First Search

As mentioned before, as these two algorithms work with an unweighted graph, their use cannot be applied directly to the problem at hand. However, they can become useful when other analyzed algorithms have a need of them. Breadth-First search helps to find the shortest path between two nodes. However, its neighborly approach implies that more information than necessary is stored as the result will come when all nodes are analyzed. This can be an inconvenience when the graph is very large. On the other hand, Depth-First search may be faster in finding a solution but it may result to be not the shortest one, and even the longest. Nevertheless, the fact that some results can be found analyzing just the children of a node and the whole matrix implies a saving time and memory.

Main differences:

Breadth-First Search Algorithm	Depth-First Search Algorithm
Analyzes neighbors before children	Analyzes children before neighbors
Uses a queue structure	Uses a stack structure
Good for finding the shortest path	Good to know if the shortest paths exists
May use memory unnecessarily	Good for saving memory if a path has already been found.

Table 7. Breadth-First and Depth-First comparison.

5.5.2 DYNAMIC PROGRAMMING ALGORITHMS

Dynamic Programming algorithms seek to obtain the optimal solution to a problem by finding the optimal solutions to sub-problems constructed from the principal one. This is possible thanks to the recursion employed by these algorithms. The results obtained in each one of the iterations are passed as inputs to the next iterations, so information is reused in order to acquire the optimal solution to each one of the subparts of the problem.

Floyd–Warshall, Bellman–Ford Algorithm are examples of dynamic programming algorithms but they will be studied later as graph algorithms. In this section the A* algorithm will be analyzed.

5.5.2.1 A* Algorithm

A*	
Classification	Dynamic Programming/Graph Algorithm
Solutions	Single-Pair Shortest Path
Graph	Directed and Weighted
Time	$O(b^d)$

Table 8. A Specifications.*

A* (A Star) algorithm was first described by Peter Hart, Nils Nilsson and Bertram Raphael in 1968 [65]. It is a computer algorithm that has been widely used in graph traversal and pathfinding since its beginnings thanks to its flexibility and applicability to different contexts. It can be defined as an informed search algorithm or best-first search as it always looks for the optimal solution among all the possible paths. This is done by choosing in each iteration the one that incurs the smallest cost (whether is distance, time or price). And among all these possibilities, it chooses the one that will lead faster to the solution.

A* algorithm can also be classified as a graph algorithm as it helps to find a path between two nodes in a graph. Specifically, it is a single-pair shortest path algorithm. It basically follows the same idea as Dijkstra's algorithm, but the main difference lies in the use of heuristics to guide the decisions the algorithm makes in each iteration. Specifically, A* decides on the path that minimizes the following formula:

$$f(n) = g(n) + h(n)$$

Formula 4. Function for A algorithm.*

Where n is the last node in the graph being considered.

- ◆ $g(n)$ is the total cost from the origin to node n .
- ◆ $h(n)$ is the heuristic function applied to estimate the cheapest path from node n to the destination point.

This formula is applied in each node that is being chosen to be analyzed until the destination point is reached.

Heuristics

In computer science, heuristics refers to methods that provide a faster solution than traditional methods. They are also used to find an approximate solution when the traditional approaches fail to provide an exact one [83]. Specifically, in search algorithms, heuristics are used to evaluate each possibility at each step with the available information in order to choose the option that will provide a faster and more accurate solution.

When applying Formula 4 in a graph, two situations may be presented:

- **The exact value of h can be calculated.**

A pre-computation is needed in order to calculate the distance between the different node in a graph before running the algorithm. In case there are no obstacles between origin and destination, this pre-computation is not necessary as the formula for calculating the Euclidean Distance between these two points can be applied.

- **An approximation of h is needed using heuristics.**

As h in Formula 4 refers to the physical distance between a node and the destination point, this distance can be calculated in three different ways (25):

- Manhattan distance

h can be calculated by summing the absolute value of the difference between x and y coordinates of the current node being analyzed and the destination point.

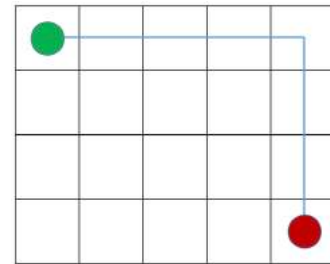


Figure 13. Manhattan distance.

$$h = |current_{node}.x - dest.x| + |current_{node}.y - dest.y|$$

Formula 5. Manhattan distance.

This approach is adequate when movement in the four principal directions is allowed.

- Diagonal distance.

This method takes the largest difference between the coordinates of the current node and the destination point, as Formula 6 indicates.

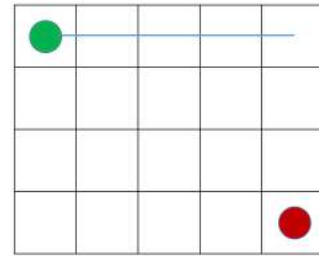


Figure 14. Diagonal distance.

$$h = \max \{|current_{node}.x - dest.x| + |current_{node}.y - dest.y|\}$$

Formula 6. Diagonal distance.

This method should be used when movement is allowed in eight directions (similar to the King in Chess).

- Euclidean distance.

This consists in the Euclidean computation of the distance between two nodes, as the length of the line that connects them directly.

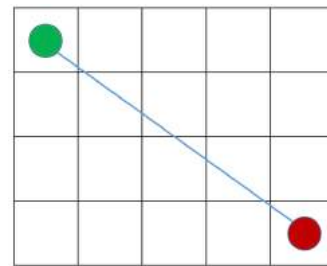


Figure 15. Euclidean distance.

$$h = \sqrt{|current_{node}.x - dest.x|^2 + |current_{node}.y - dest.y|^2}$$

Formula 7. Euclidean distance.

When movement in every direction is permitted, this method is the most suitable for use.

Pseudocode

The following lines show the pseudocode for A* algorithm. It is very similar to that of Dijkstra, the main difference lying in the fact that the accumulated weight of each node adds the value of the $h(n)$ function.

```

function A*(G,source,destination):
  distance[source] = ∞;
  Q = PriorityQueue;
  Q.queue(source,distance[source]); //The original node is inserted in the queue

  while(size(Q) > 0)
    v = Q.dequeue(); //The last node inserted is dequeued from Q
    visited[v] = true;

    for all neighbors w of v in G: //Every neighbor of v is analyzed
      distance[w] = distance[v] + weight(v,w) + h(v); //The weight of an edge is the accumulated
      //weight plus its own.

      if(distance[w] < distance[v])
        previous[w] = v;
        Q.queue(w,distance[w]);
      end if

    end for

    getPath(destination); //Using array previous(destination)

  end while
  
```

Pseudocode 4. A Algorithm.*

Operation

The procedure that A* algorithm chooses to find the optimal path to the destination point is very similar to Dijkstra. In fact, Dijkstra is a specific case of A* where $h=0$ for all nodes.

Figure 16 shows the graph used in the previous examples and the one employed to understand A* functioning. The origin is x1 and the destination is x6. Also another difference between this graph and the previous graphs is that h has obtained a value for each node. In this case $h(n)$ is the supposed Euclidean distance between each one of the nodes and the destination point (x6) in this case.

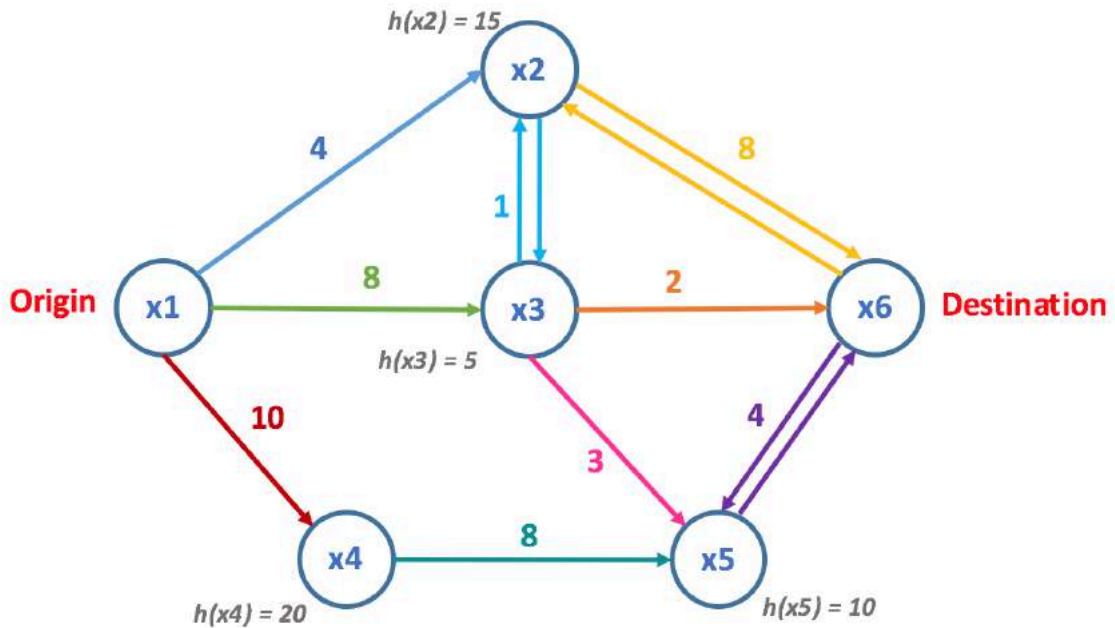


Figure 16. Graph for A* with $h(n)$.

In Figure 17 the functioning of A* is depicted. In step 1, in order to see the cost to reach x2, x3 and x4, not only does the weight of each route need to be taken into account but also the $h(n)$ function in each one of this nodes, that is, the physical distance between them and the destination point x6.

Opposite to Dijkstra, A* chooses the following node x3 instead of x2, as the cost to reach x2 is 29 and to x3 13 (step I). Even though x4 is chosen as the next node to be analyzed (step II), in step III the final node is reached and so the algorithm stops to run in step IV when x6 is selected.

The number of iterations is cut short to 4, instead to the six steps needed for Dijkstra in this same graph.

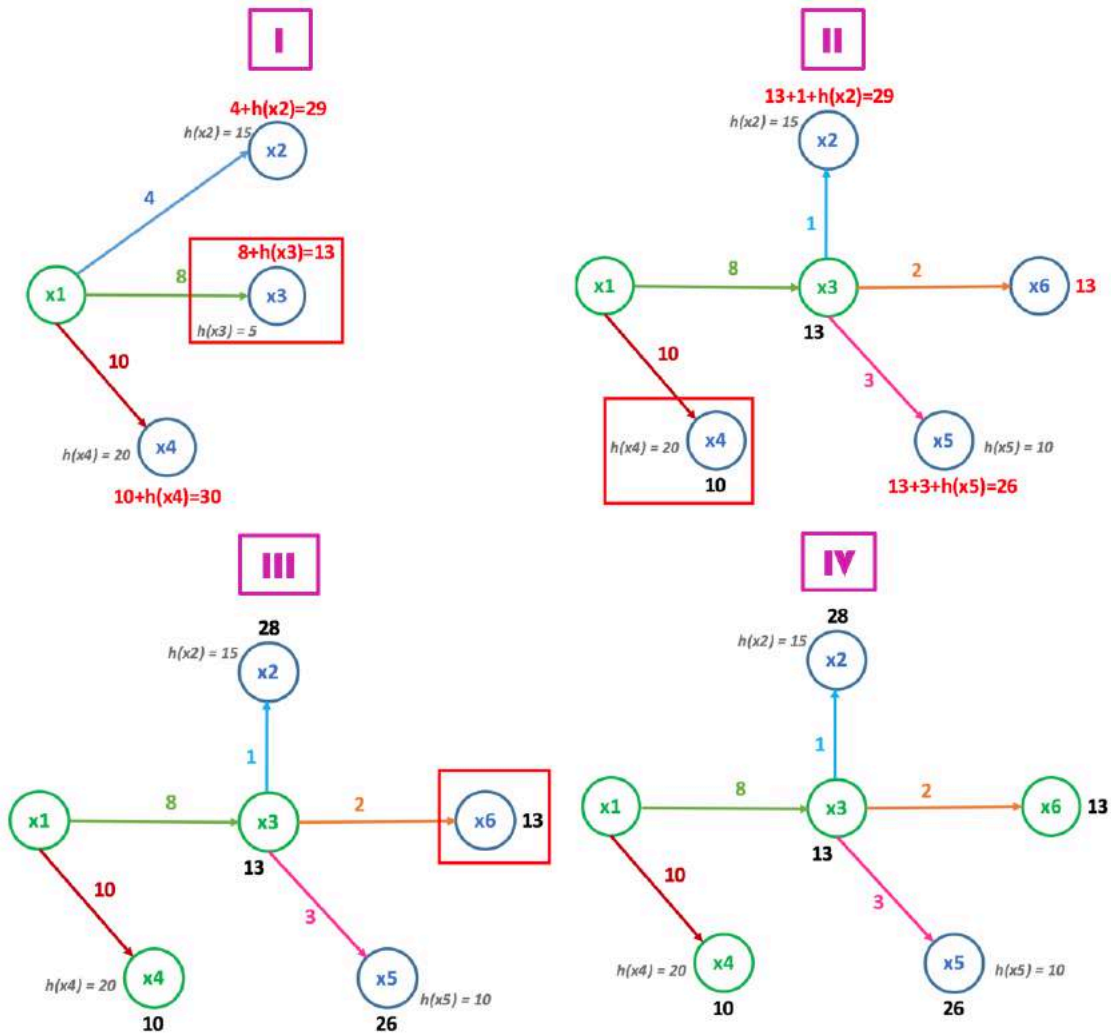


Figure 17. Steps for A* algorithm.

Time Complexity

The time complexity of A* algorithm depends on the heuristic function $h(n)$ that is being used. In an unbounded search space, this complexity can be calculated as: $O(b^d)$, where d is the shortest path to be found and b is the branching factor, that is, the number of children of a single node. One of the advantages of A* is that the algorithm prunes a considerable amount of nodes another method would expand to. As this is a direct result of using heuristics, the best heuristics that could be used are those that would lower the effective branching factor.

As a conclusion, the time complexity of A* algorithm is **polynomial** if the following conditions are met:

- The search space is a tree.
- Only a goal exists.
- The heuristic function h follows the condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

Formula 8. Heuristics condition for A Algorithm.*

Being $h^*(x)$ the optimal heuristic that obtains the exact cost of reaching the destination point.

An algorithm with a polynomial time complexity means that its running time is upper bounded by a polynomial expression that depends on the input of the algorithm, being the input in this case the graph, with its nodes and edges.

Advantages and Disadvantages

Advantages

The algorithm is very fast as it does not consider useless braches in its development as Dijkstra and BFS do. This means that it does not apply a blind search. How fast it can become or the memory needed depends on the heuristic function that is being used.

The algorithm is complete and optimal. This means that if a solution exists, the algorithm is going to find it and it will be the optimal one.

Its efficiency makes it very suitable for complex problems where securing a solution is time bounded. Also it is optimally efficient, which means that there is no other algorithm that uses a lower number of nodes to expand in order to find the optimal solution.

These advantages make A* algorithm very suitable for the problem at hand and one that will be as a potential substitute for the actual Dijkstra implemented in the company.

Disadvantages

The algorithm is actually complete if the branching factor b is a finite number and a fixed cost is associated to each action.

The speed of the execution of the algorithm depends entirely on the heuristic function that is being used. Also the memory usage can become a limitation for the algorithm when the problem is very complex.

5.5.3 GRAPH ALGORITHMS

As explained in section 5.1, graph algorithms are applied to find the connecting path between two nodes in a graph. Some of the algorithms that are going to be studied in this section can be classified otherwise, such as Floyd-Warshal which is an example of a dynamic programming algorithm or Dijkstra, which is a greedy algorithm. The way these algorithms work places them in these different categories but as they can also be used to find the shortest path connecting two nodes in a graph, they can also be included in the graph algorithm classification.

As stated before, graph algorithms can be organized in four groups: single-source, single-destination, all-pairs, and single pair shortest path algorithms. They are explained in section 5.1. The ones that are going to be analyzed in this section are Bellman-Ford, Floyd-Warshal and Johnson algorithm.

5.5.3.1 Bellman-Ford Algorithm

Bellman-Ford	
Classification	Dynamic Programming/Graph Algorithm
Solutions	Single-Source Shortest Path
Graph	Directed and positive/negative Weighted
Time	$O(V \cdot E)$

Table 9. Bellman-Ford specifications.

Bellman-Ford algorithm was initially developed by Alfonso Shimbel in 1955 but its name comes from Richard Bellman and Lester Ford who published its functioning in 1958 and 1956, respectively [62]. Also Edward F. Moore published the algorithm in 1957, so it is sometimes known as Bellman-Ford-Moore Algorithm.

Apart from being a graph algorithm, Bellman-Ford can also be categorized as a dynamic programming algorithm thanks to its iterative approach and the division in subparts of the problem.

This algorithm calculates the shortest path between one node and the rest in a weighted graph, just as Dijkstra in what is referred to as *single-source shortest-path problem*. But it is more versatile than Dijkstra as it is compatible with negative weights on the edges. It returns a Boolean value determining the presence of negative-weighted cycle reachable from the source. If this cycle exists, the algorithm indicates that no solution exists for the problem. Also, the number of iterations needed are $|V|-1$, as it will be seen in the example.

As stated before, the fact of working with negative weights is not included in the scope of this project. However, the study of the functioning of the algorithm when all weights are positive may be interesting in order to decide which could be the advantages of applying this algorithm to the problem at hand.

Pseudocode

The following lines present the pseudocode for Bellman-Ford algorithm.

```

function Bellman-Ford (G,source,destination):
  for all v in V:
    distance[v] = ∞;                                     //All nodes are initialized with a distance of ∞
  end for

  distance[source] = 0;                                  //Only the source has distance 0

  for v in |V|-1:                                       //Number of iterations = |V|-1

    for all edges e(v,w) in E:

      if(distance[v] > distance[w] + weight(e(v,w)))    //Only when both ends of an edge have a
                                                         distance different from ∞ will be the distance
                                                         of a node be updated if the condition is met

```

```

        distance[w] = distance[w] + weight(e(v,w));
    end if
end for
end for
getPath(destination);

```

Pseudocode 5. Bellman-Ford Algorithm.

The main procedure in the algorithm, as it will be seen in the following subsection, is iterated a number of times equivalent to the total numbers of nodes in the graphs minus 1, even if the last iterations probably won't change the result at all in most of the cases. In each iteration, every edge in the graph will be analyzed and only if the nodes in both ends of the edge have a distance set different from ∞ , then the distance of the nodes to the source could be changed if the condition is met.

This process of analyzing the edges is named *relaxation*. Bellman-Ford provides a relaxation process on all the edges and $|v|-1$ times, while Dijkstra only applies it on the edges of the node that is being considered.

Operation

Following the pseudocode in the previous subsection, the next image shows the functioning of the algorithm in the graph that has been being used in the previous algorithms. As said before, this algorithm has the advantage of being capable of working with negative weights but in this case the algorithm will be applied in the usual graph where all the edges have positive weights.

As the graph has 6 nodes, the algorithm needs to apply 5 iterations. In the first iteration, when considering x_1 , the distances to x_2 , x_3 and x_4 are 4, 8 and 10, respectively. However, in this same iteration but considering x_2 , the distance to x_3 gets changed to 5. And it can be changed because both ends of this edge are known. x_6 remains as ∞ because that end was unknown. This process is followed in the rest of iterations.

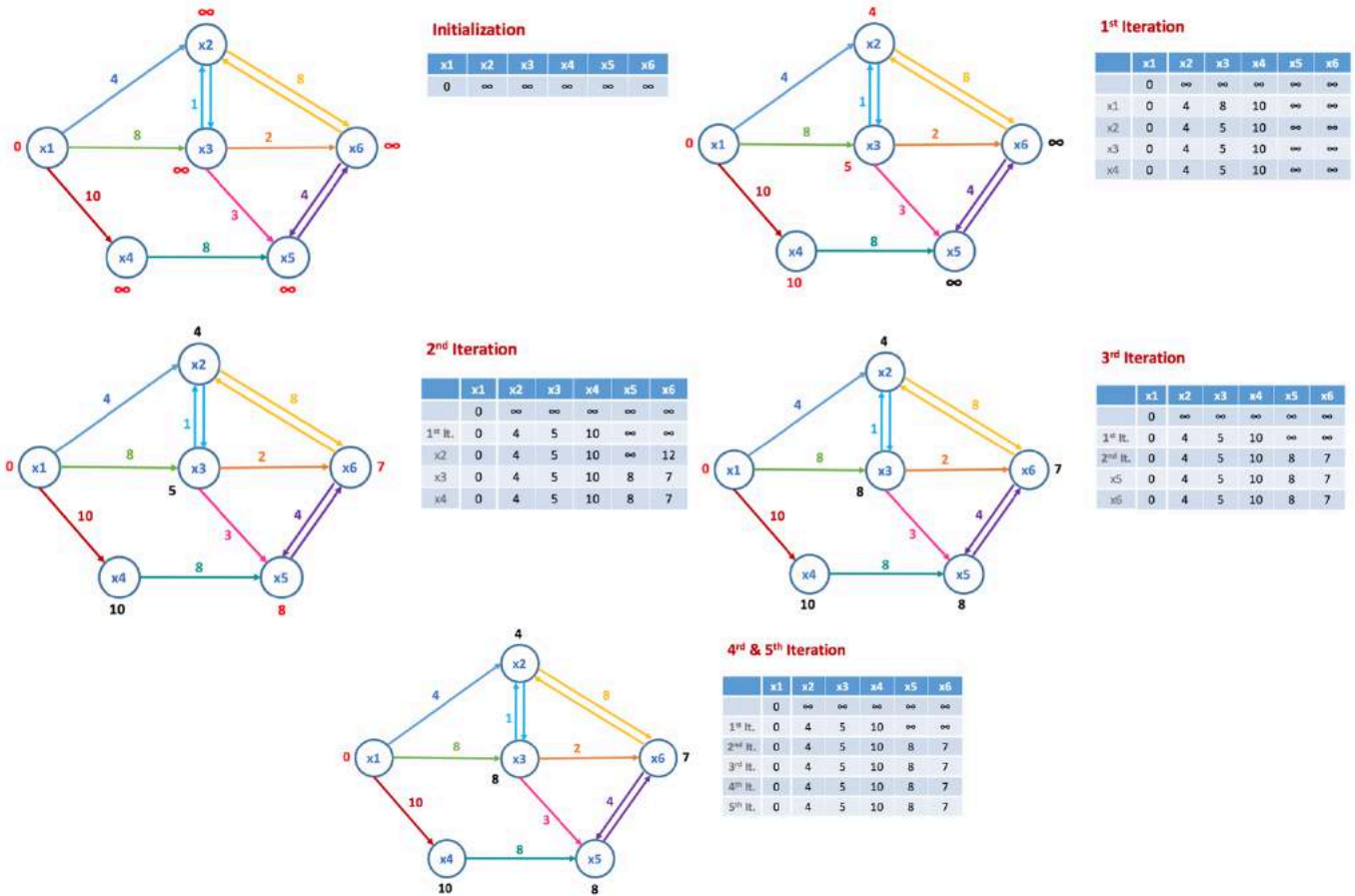


Figure 18. Steps for Bellman-Ford Algorithm.

Time Complexity

The time complexity of this algorithm can be stated as follows:

$$O(|V| \cdot |E|)$$

Usually, the value of E can be represented as:

$$|E| \approx |V|^2$$

Which will result in a time complexity for the whole algorithm expressed as:

$$O(|V| \cdot |E|) \approx O(|V|^2)$$

Formula 9. Approximated time complexity of Bellman-Ford.

Advantages and Disadvantages

Advantages

Given the previous algorithms that have been studied, one of the main advantages of this algorithm is its capability of working with negative weights. It is already known that the graph the company works with lacks of this type of edges, where the weight might be negative, so this feature hardly comes as a plus for considering Bellman-Ford algorithm as a potential substitute.

When a solution does not exist, the algorithm indicates this fact.

The relaxation process the algorithm applies to all the edges in each one of the iterations allows for a wider class of inputs to be introduced to the algorithm, compared to those allowed in Dijkstra, as its relaxation process only cover the edges of the node being analyzed.

Disadvantages

The main the disadvantage encountered when considering this algorithm as a substitute for Dijkstra is that it is actually more complex. The time complexity is $O(|V| \cdot |E|)$ while Dijkstra's is $O(|V|^2)$. This is because of the relaxation process which is more elaborated than that of Dijkstra.

As stated before, the fact that this algorithm is more suitable than Dijkstra in the case the weights are negative leads to discard it as substitute of the latter.

5.5.3.2 Floyd-Warshall Algorithm

Floyd-Warshall	
Classification	Dynamic Programming/Graph Algorithm
Solutions	All-Pairs Shortest Path
Graph	Directed and positive/negative Weighted
Time	$O(V ^3)$

Table 10. Floyd-Warshall specifications.

This algorithm is also categorized as a dynamic programming algorithm. It was first published as it is currently known in 1962 by Robert Floyd. However, publications by Bernard Roy in 1959 and Stephen Warshall in 1962 were also recorded. Hence the various names assigned to this algorithm: Floyd's algorithm, Roy–Warshall algorithm or Roy–Floyd algorithm [63].

This algorithm is capable of finding the shortest paths between every pair of nodes in a weighted graph, so it is classified as an *all-pairs shortest path algorithm*. The edges weights can be either positive or negative, but it cannot work with cycles, opposite to Bellman-Ford algorithm.

The original development of the algorithm does not provide details on the paths found between each pair of nodes in the graph, but a modification can be applied in order to obtain this information.

Pseudocode

```

function Floyd-Warshall(G):
  dist = VxV matrix -> ∞           //dist is a matrix initialized to ∞

  for all v in V
    dist[v][v] = 0;                 //diagonal set to 0
  end for

  for all edges e(u,v) in E:
    dist[u][v] = weight(u,v);      //Every edge is updated with its weight
  end for
  
```



```

for k in range(1,V):
    for i in range(1,V):
        for j in range(1,V):
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] = dist[i][k] + dist[k][j]; //The matrix is uptaded
            end if
        end for
    end for
end for

```

Pseudocode 6. Floyd-Warshall Algorithm.

In the first place a matrix is initialized with all its values to ∞ . Then the elements in the diagonal are set to 0, and the edges to the value of the corresponding weights. Then the next three nested for-loops are executed and the values updated if the condition exposed is met.

The following pseudocode shows a way of obtaining information about the paths between every pair of nodes.

```

function Floyd-Warshall(G):
    dist = VxV matrix ->  $\infty$  //dist is a matrix initialized to  $\infty$ 
    next = VxV matrix -> null

    for all v in V
        dist[v][v] = 0; //Diagonal set to 0
    end for

    for all edges e(u,v) in E:
        dist[u][v] = weight(u,v); //Every edge is updated with its weight
        next[u][v] = v; //The value of next for each edge is the
    end for //destination node of the route

    for k in range(1,V):
        for i in range(1,V):
            for j in range(1,V):
                if dist[i][j] > dist[i][k] + dist[k][j]
                    dist[i][j] = dist[i][k] + dist[k][j]; //The matrix is uptaded
                    next[i][j] = next[i][k];
                end if
            end for
        end for
    end for

```

```

        end for
    end for
end for

function CalculatePath(u,v):
    if (next[u][v] == null)
        return []; //There is no path between them
    end if

    path = u;

    while(u≠v)
        u = next[u][v];
        path .= u;
    end while

    return path;

```

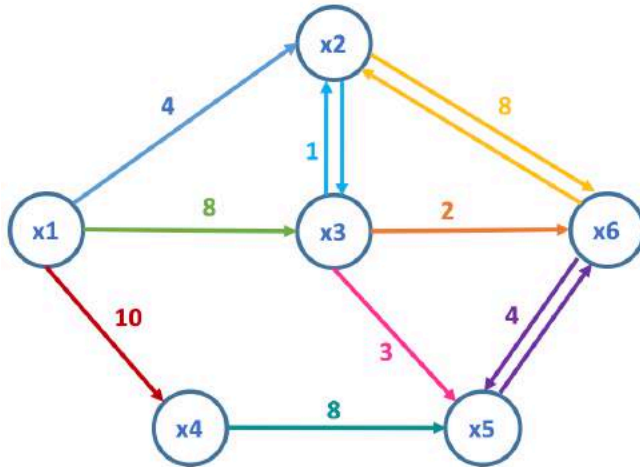
Pseudocode 7. Floyd-Warshall Algorithm with path reconstruction.

With the pseudocode provided, the path between two nodes can be computed by means of another matrix that stores the information computed in the execution of the algorithm.

Operation

The functioning of this algorithm relies on the presence of a matrix that stores the distances between the different nodes. The updating of this matrix is done exactly following the condition stated in the pseudocode.

Figure 19 depicts the initialization format of the matrix that stores the distances between each node. The diagonal has a value of 0 because it is the distance to go from one node to the same one. Then the origin is read on the row indexes, and the destination on the column's. For instance, the distance from node x1 to x3 is 8, which it is already known because it is a direct route.



Indexes

k	x1	x2	x3	x4	x5	x6
i	x1	x2	x3	x4	x5	x6
j	x1	x2	x3	x4	x5	x6

Matrix dist.

		destination					
		x1	x2	x3	x4	x5	x6
origin	x1	0	4	8	10	∞	∞
	x2	∞	0	1	∞	∞	8
	x3	∞	1	0	∞	3	2
	x4	∞	∞	∞	0	8	∞
	x5	∞	∞	∞	∞	0	4
	x6	∞	8	∞	∞	4	0

Figure 19. Initialization for Floyd-Warshall algorithm.

Figure 20 shows the distance matrix for some cases in which there are variations in the matrix, that is, when the condition determined in the pseudocode has been met.

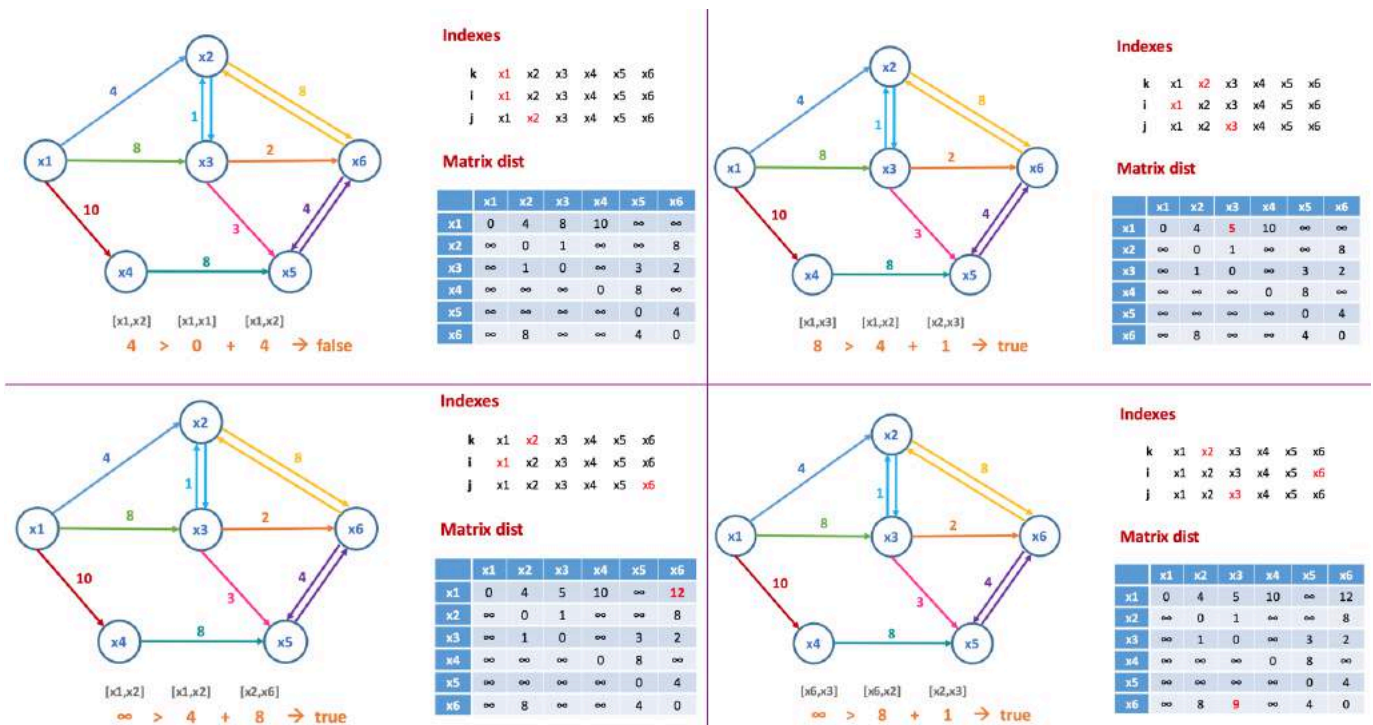
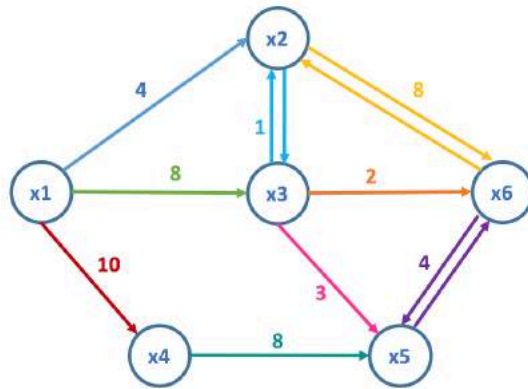


Figure 20. Distance matrix for Floyd-Warshall.

Figure 21 represents the distance matrix in the last iteration, showing the shortest paths between each pair of nodes.



Indexes

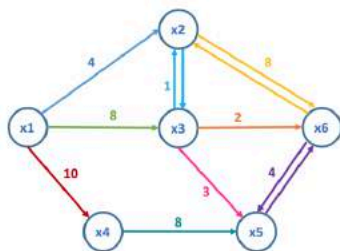
k	x1	x2	x3	x4	x5	x6
i	x1	x2	x3	x4	x5	x6
j	x1	x2	x3	x4	x5	x6

Matrix dist

	x1	x2	x3	x4	x5	x6
x1	0	4	5	10	8	7
x2	∞	0	1	∞	4	3
x3	∞	1	0	∞	3	2
x4	∞	20	21	0	8	12
x5	∞	12	13	∞	0	4
x6	∞	8	9	∞	4	0

Figure 21. Last iteration of Floyd-Warshall algorithm.

For instance, the shortest path from x5 to x3 has a total accumulated weight of 13. The path itself is unknown but it can be computed the following way: $x5 \rightarrow x6, x6 \rightarrow x2$ and $x2 \rightarrow x3$. As stated before, there is a way to modify this algorithm in order to store the path chosen to obtain each one of the values in the distance matrix. This is analyzed in the following images.



Next Matrix

	x1	x2	x3	x4	x5	x6
x1	null	x2	x3	x4	null	null
x2	null	null	x3	null	null	x6
x3	null	x2	null	null	x5	x6
x4	null	null	null	null	x5	null
x5	null	null	null	null	null	x6
x6	null	x2	null	null	x5	null

Indexes

k	x1	x2	x3	x4	x5	x6
i	x1	x2	x3	x4	x5	x6
j	x1	x2	x3	x4	x5	x6

Matrix dist

	x1	x2	x3	x4	x5	x6
x1	0	4	8	10	∞	∞
x2	∞	0	1	∞	∞	8
x3	∞	1	0	∞	3	2
x4	∞	∞	∞	0	8	∞
x5	∞	∞	∞	∞	0	4
x6	∞	8	∞	∞	4	0

Figure 22. Modification of Floyd-Warshall Algorithm.

Next Matrix stores the predecessors for each node. At the beginning, in Figure 22, only the known predecessors are written in the table, the rest are set to null. Figure 23 shows the updating of this matrix for the same iterations as in Figure 20.

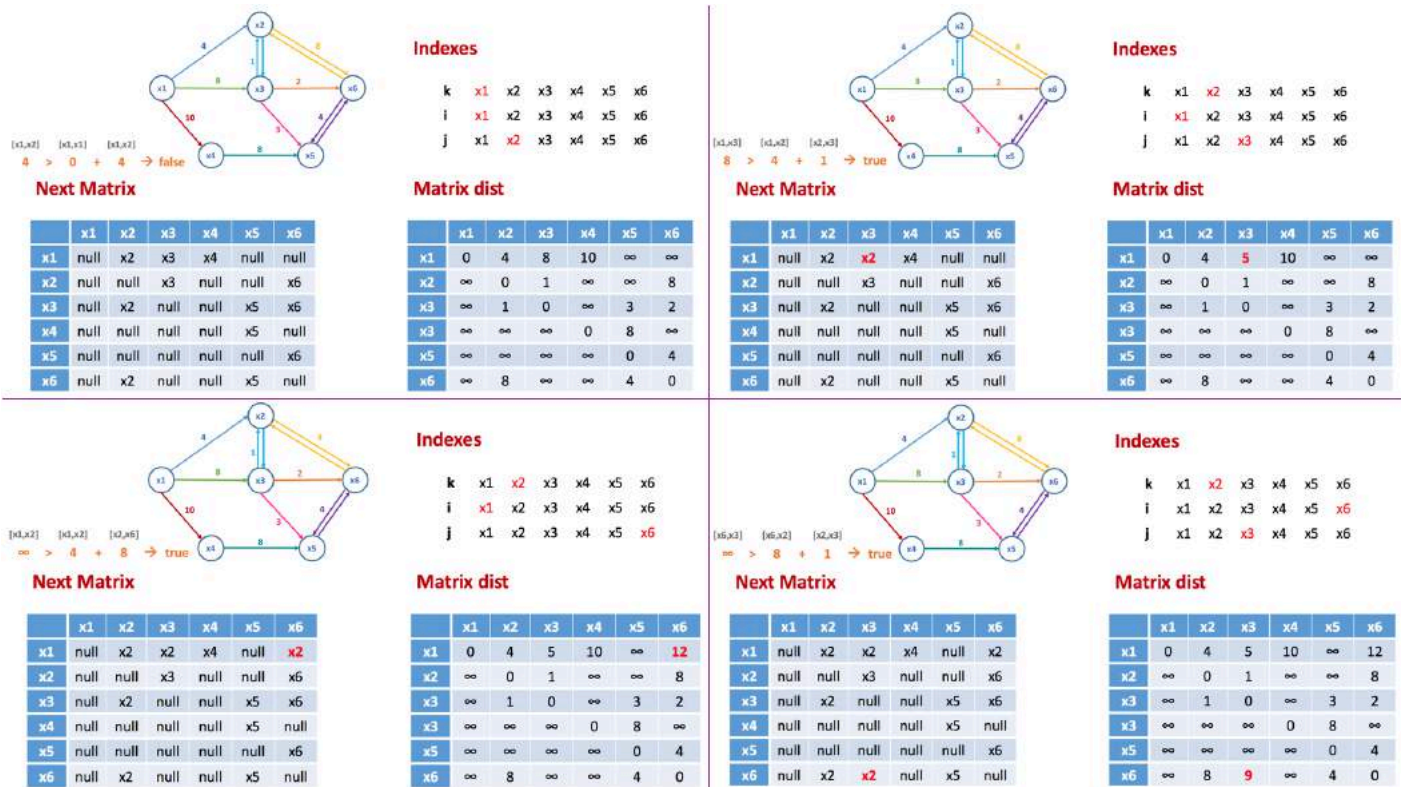
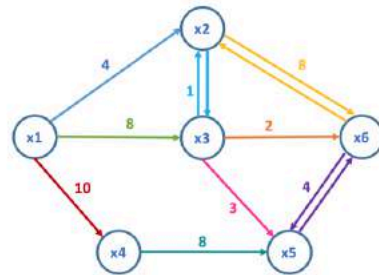


Figure 23. Next Matrix for Floyd-Warshall Algorithm.

Figure 24 shows the final version of *Next Matrix* where the predecessors of each node for the shortest paths between each pair has been calculated. With this last version the calculation of the shortest path between every pair of nodes is possible. It can be known, not only the length of the shortest path, but also its details. For example, the shortest path between x1 and x6 can be calculated as the formula on the pseudocode states and it is explained as follows:

1. The first position of the path is the origin: $\text{path}[1] = x1$.
2. In *Next Matrix* the value of $[x1,x6]$ is taken as $\text{path}[2] = x2$.
3. Now the value of $[x2,x6]$ is searched in the matrix, $\text{path}[3] = x3$.
4. The process repeats. $\text{path}[4] = [x3,x6] = x6$.
5. Now the destination point has been reached.

So the shortest path between x_1 and x_6 is: $[x_1, x_2, x_3, x_6]$, with an accumulated weight of 7, which is known thanks to the distance matrix.



Indexes

k	x1	x2	x3	x4	x5	x6
i	x1	x2	x3	x4	x5	x6
j	x1	x2	x3	x4	x5	x6

Next Matrix

	x1	x2	x3	x4	x5	x6
x1	null	x2	x2	x4	x2	x2
x2	null	null	x3	null	x3	x3
x3	null	x2	null	null	x5	x6
x4	null	x5	x5	null	x5	x5
x5	null	x6	null	null	null	x6
x6	null	x2	x2	null	x5	null

Matrix dist

	x1	x2	x3	x4	x5	x6
x1	0	4	5	10	8	7
x2	∞	0	1	∞	4	3
x3	∞	1	0	∞	3	2
x4	∞	20	21	0	8	12
x5	∞	12	13	∞	0	4
x6	∞	8	9	∞	4	0

Figure 24. Last iteration of the modified Floyd-Warshall algorithm.

Time complexity

The three nested for-loops provide Floyd-Warshall algorithm of a time complexity of

$$O(|V|^3)$$

Formula 10. Time Complexity for Floyd-Warshall Algorithm.

This time complexity is higher than that of Dijkstra or Bellman-Ford algorithms but the information obtained when running this algorithm is higher, as it provides the shortest paths between every pair of nodes.

Advantages and Disadvantages

Advantages

The main advantage of this algorithm is that it provides the shortest path between every pair of nodes in a positive or negative weighted graph. Also, it can be easily modified in order to find the details of the paths it finds.

Only one execution is needed to find all the solutions. Also the structure and functioning of the algorithm is very simple and very easy to implement.

Disadvantages

The original implementation of the algorithm does not include information about the shortest paths found for each pair of nodes. However, it has been proved that modifying the algorithm to include this feature is a fairly easy task.

The shortest paths can only be found when there are no negative cycles. Also this algorithm may run slower than others that provide the same results (such as Johnson's, see section 5.5.3.3).

All in all, this algorithm works fine when the objective is to find the shortest paths between every pair of nodes in the graph with just one iteration. Nevertheless, this is not the main goal of this project, which is to find several paths between the same pair of nodes.

5.5.3.3 Johnson's Algorithm

Johnson	
Classification	Graph Algorithm
Solutions	All-Pairs Shortest Path
Graph	Directed and positive/negative Weighted
Time	$O(V ^2 \log V + V \cdot E)$

Table 11. Johnson specifications.

This algorithm receives its name from Donald B. Johnson, who first published it in 1977. Like Floyd-Warshall algorithm, Johnson's algorithm provides the shortest path between all pair of nodes in a positive or negative weighted and directed graph [64].

It basically consists on employing Bellman-Ford algorithm to remove the negative weights in a graph and then applying Dijkstra to obtain the final results, as this algorithm only works with positive weights.

This algorithm focuses in two main facts: the possibility to work with negative weights in a graph and finding the shortest paths between all nodes in a graph. For these two reasons it is not the most suitable one for the problem analyzed in this project. On one hand, it has already been stated that negative weights will never be consider in a graph as the one provided by the company, and on the other hand, algorithms that provide more than one result in finding the shortest path between a single pair of nodes are more interesting than those that make use of time and resources to calculate what is insubstantial information for the problem at hand. Therefore, the operation or pseudocode of this algorithm is not even considered in this project.

Time complexity

The time complexity of Johnson's algorithm consists of the combination of the execution of Bellman-Ford and Dijkstra's algorithms. So in first place, Bellman-Ford algorithm provides an execution duration of:

$$O(|V| \cdot |E|)$$

While all $|V|$ executions of Dijkstra's algorithm to find the shortest path between every pair of nodes results in:

$$O(|V| \log|V| + |E|)$$

Then, the total time that it takes Johnson's algorithm to be executed results in:

$$O(|V|^2 \log|V| + |V| \cdot |E|)$$

Formula 11. Time complexity for Johnson's Algorithm.

This makes Johnson's algorithm faster than Floyd-Warshall's for sparse graphs, that is, where a lower number of edges can be found. This is because Johnson's algorithm depends on the edges whereas Floyd-Warshall's does not.

Advantages and Disadvantages

Advantages

Johnson's algorithm is faster than Floyd-Warshall's algorithm in sparse graphs and provides the same results than the latter.

It can work in negative weighted graphs, which gives it an advantage over Dijkstra's algorithm, which was limited by this fact.

Disadvantages

The algorithm may not work as efficiently when the graph contains a lot of edges, as its execution time depends entirely on this fact.

On the other hand, as explained before, the features that make this algorithm so desirable in certain situations are not essential for the problem studied in this project. Therefore, the algorithm will not be further considered.

5.5.4 K-SHORTEST PATHS

In this section, a new range of algorithms are considered. They differ from the ones studied previously in that they find, not only *the* shortest path between two nodes, but *the k shortest* paths that interconnect them. Of course only one will be optimal, but sometimes more than one option to reach the destination point is desired, as more than one variable is taken into account, such as time or money.

Therefore, three algorithms will be analyzed in this section: Yen's algorithm, Eppstein algorithm and Iterative Enumeration Algorithm (IEA).

5.5.4.1 Yen's Algorithm

Yen	
Classification	Graph Algorithm
Solutions	K-Shortest Path
Graph	Directed and Weighted
Time	$O(KN \cdot V ^2)$

Table 12. Yen specifications.

This algorithm was first published by Jin Y. Yen in 1971 [26]. The procedure it applies consists on calculating the shortest path between two nodes using a single-source shortest path algorithm, and then removing edges in order to find k-1 different paths from the same origin to the same destination.

If the initial algorithm applied to find the shortest path is chosen to be Dijkstra then it can be concluded that the company has been applying an implementation of Yen's algorithm.

Time complexity

As the algorithm depends on the algorithm applied to find the shortest path between two nodes, in order to analyze the time complexity of Yen's algorithm, Dijkstra is assumed to be used. This way, the time complexity of the actual implementation in the company code can be computed.

As seen in section 5.4.1.3 the time complexity of Dijkstra's algorithm is

$$O(|V|^2)$$

Considering that, Yen's algorithm calls k·N times, being N the worst case of spur paths. Therefore, the total time complexity for Yen's algorithm in the worst case is:

$$O(KN \cdot |V|^2)$$

Formula 12. Time Complexity for Yen's Algorithm.

Advantages and Disadvantages

Advantages

One of the main advantages of this algorithm is that it provides more than one result when finding the shortest path between two nodes. It is true that only the first one, calculated without removing any edges, provides the optimal solution. But the possibility of finding more results may provide more options that before were not contemplated and that may be even more desirable in terms of time or money or any other feature that is being considered.

Disadvantages

The principal disadvantage was stipulated in section 5.4.2, where it was mentioned that removing edges in an arbitrary way may not lead to the best solutions after the optimal solution has been found. Maybe the edge that has been removed is the one that provides the best path. Uncertainty is the main disadvantage of this algorithm.

5.5.4.2 Eppstein's Algorithm

Eppstein	
Classification	Graph Algorithm
Solutions	K-Shortest Path
Graph	Directed and Weighted
Time	$O(k + k \cdot \log k)$

Table 13. Eppstein specifications.

David Eppstein is a mathematician and computer scientist from the United States that developed the algorithm that carries his own name in 1997 [84]. This algorithm helps to find the k-shortest paths between two nodes in a weighted and directed graph. Origin and destination points need to be known.

The basic functioning of this algorithm is as follows. In the first place, Breadth-First search algorithm needs to be applied to the reversed graph, that is, with the edges' orientation

changed. This way the shortest paths to the destination can be found. Then, *sidetrack edges* are defined. These edges are those not included in the Breadth-First algorithm results and represent the added additional distance if they were taken instead of the ones obtained by the algorithm. In the end, the different k paths to the destination will be obtained.

Pseudocode

```

function Eppstein(G,origin,destination):
  T = tree that results on computation of reversed Breadth-First Search on G;

  for all nodes v in V:
    compute d(n,destination) following T;           //Distance from one node to the destination
  end for

  for all edges e in T:
     $\delta(e) = 0$ ;                               // $\delta(e)$  measures the additional cost of taking that edge
  end for

  S = G - T;
  for all edges (e) in S:
     $\delta(e) = \text{length}(e) + d(\text{head}(e),\text{destination}) - d(\text{tail}(e),\text{destination})$ ;
  end for

  build ST                                         //Tree of sidetracks

  for each node in ST:
    H[node] = heap(children(node));                //For each node in ST with children heap is built with its children
  end for

  record incremental costs;

  for each node in ST:
    H[node].append(H[parent(node)]);              //All heaps are appended
  end for

  start_node = heap(0);
  H_final = start_node.append(H[root]);           //A node with cost 0 is appended at the top of the total heap
  
```

Pseudocode 8. Eppstein's Algorithm.

Operation

- *Reversed Breadth-First Search algorithm*

Eppstein's algorithm applies a reversed Breadth-First search on the destination, so the result includes all the paths that reach the destination. Figure 25 shows the reversed graph on which Breadth-First algorithm should be applied.

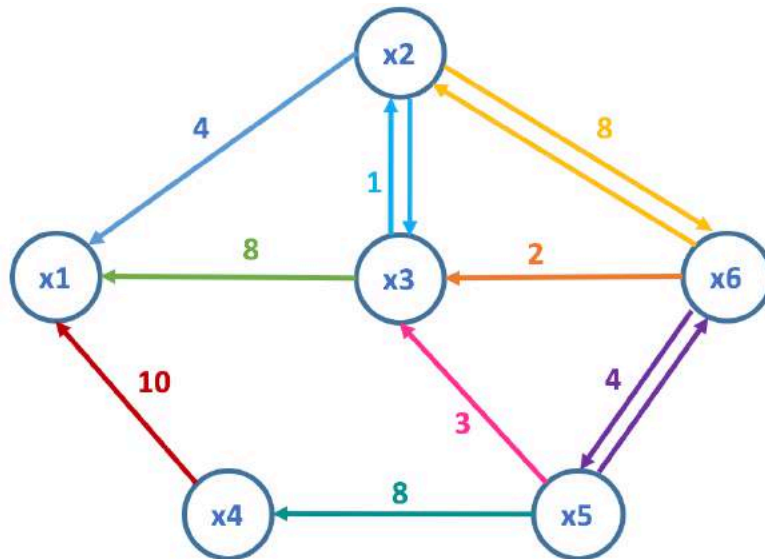


Figure 25. Reversed graph for Breadth-First search in Eppstein's algorithm.

Figure 26 shows the tree used to execute Breadth-First algorithm. As weights need to be considered to find the shortest path to the nodes, the whole tree that relates the interconnections between the nodes is shown on the left. Then, the final tree where the shortest paths can be found is provided on the right.

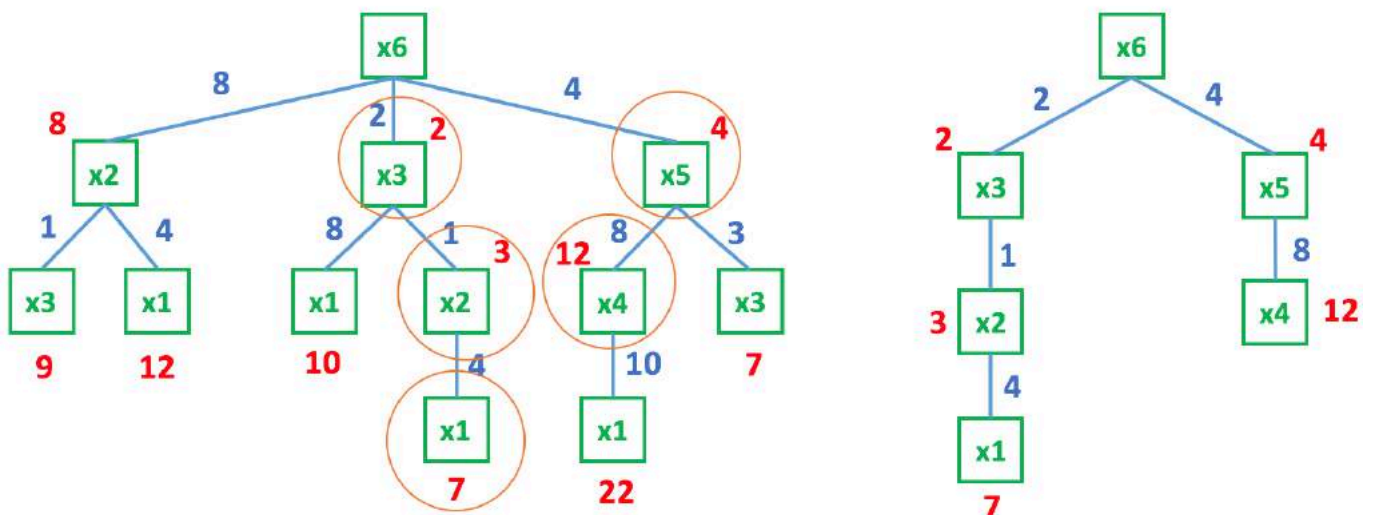


Figure 26. Tree for reversed Breadth-First Search in Eppstein's algorithm.

The values in blue represent the weights on the edges while the values in red represent the accumulated weight to each node. The nodes that have been circled on the left are the ones with a lowest accumulated weight and the ones chosen for the tree on the right.

Figure 27 shows the resulting graph once the tree on the right side of the figure above has been applied to the original one.

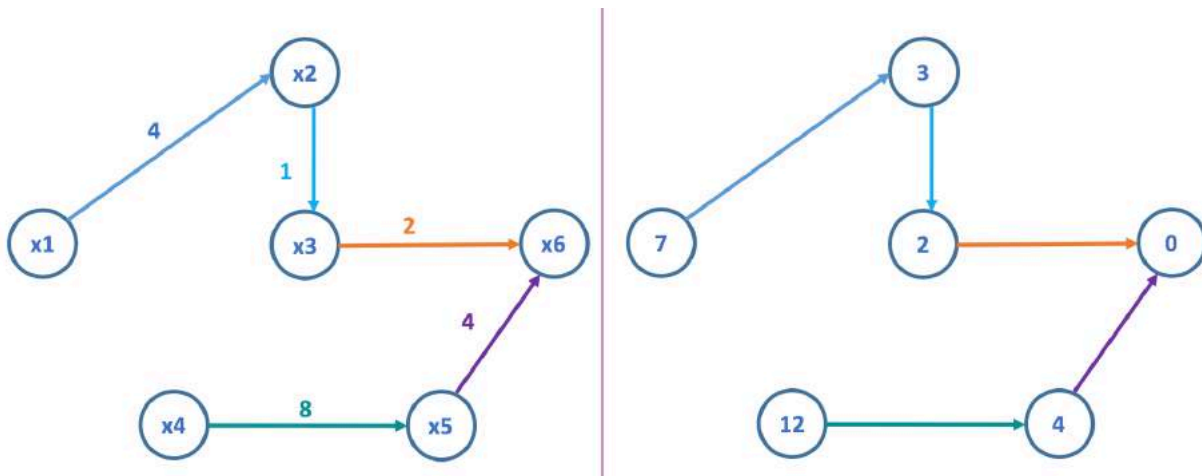


Figure 27. Graph with shortest paths in Eppstein's algorithm.

On the left side of the image, it can be seen that two entire paths can be built that reach the destination. On the right side, the same graph is represented but with the accumulated distance to the destination inside each node. This will facilitate the following explanations.

- *Calculation of sidetracks*

Sidetracks are the edges that are not included in the graph of Figure 27 and that will add an additional cost when using them to reach the destination. The weight of these sidetracks can be calculated as follows:

$$\delta(e) = \text{length}(e) + d(\text{head}(e), \text{destination}) - d(\text{tail}(e), \text{destination})$$

Formula 13. Calculation of Sidetrack value.

In Formula 13, $length(e)$ refers to the weight of the edge. The *head* of the edge is the node to which the head of the arrow aims, while its *tail* is the opposite side. Then $d(head(e), destination)$ is the distance between the node to which the head of the edge aims and the destination.

Figure 28 shows the sidetracks with dotted arrows. For instance, in the red sidetrack, $\delta(e)$ would be calculated as:

- $length(e) = weight(e) = 10$.
- Head = value at the head of the arrow = 12.
- Tail = value at the tail of the arrow = 7.

$$\delta(e) = 10 + 12 - 7 = 15$$

This means that going from node x1 to x6 through node x4 would increment the total cost in 15.

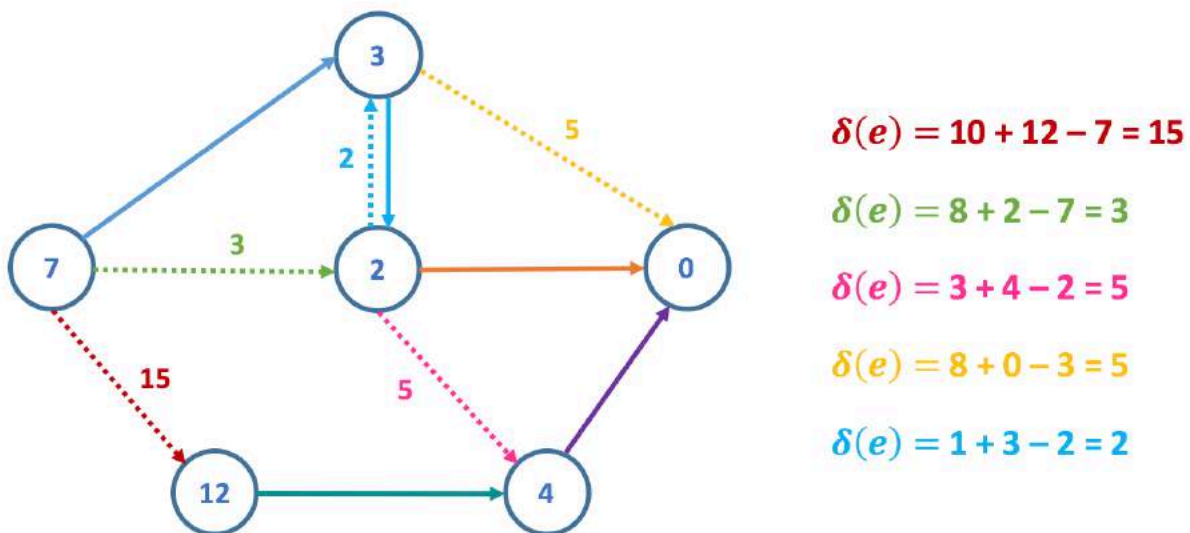


Figure 28. Calculation of sidetrack edges.

The only sidetrack edges that have not been calculated are those that have their origin in x6, which is the final destination. The additional cost they provided was that of going to the nearest node and come back.

- *Tree of sidetrack sequences*

The next step is the arrangement of the sidetracks in a tree that provides the interdependencies between them. The first level of the tree includes the sidetracks that are directly connected to one of the direct paths to the destination.

If the sidetrack 3 is chosen, then other sidetracks can also be selected to reach the destination, such as 2 and then the yellow 5, or just the pink 5. This tree represents which sidetracks need to be taken to reach the destination through the two direct paths depicted in the figures.

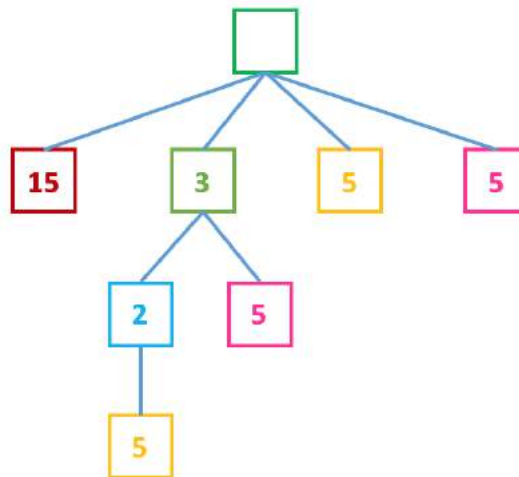


Figure 29. Tree of sidetracks.

- *Heap of sidetracks*

Once the tree of sidetracks has been procured, a heap of its branches shall be built. A heap is a binary tree, which means that every node has only two children. And also, the disposition of the nodes is that they are ordered by weight. A heap needs to be built from each level of the tree. So in this case, three heaps will be procured.

Figure 30 shows this disposition.

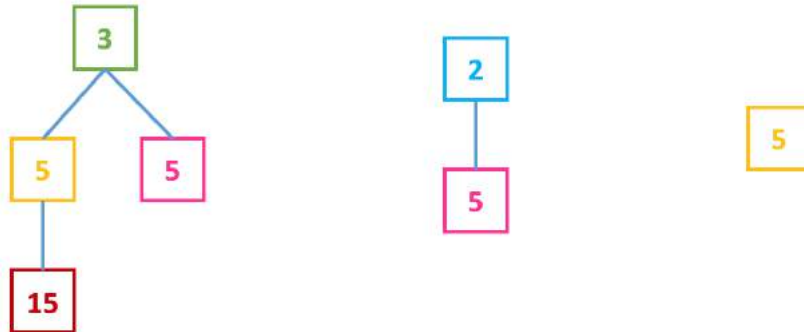


Figure 30. Heaps for the sidetracks.

- *Arrangement of heaps*

This process includes several steps. First, the incremental costs in each branch of the heaps needs to be recorded. Afterwards, each sidetrack node at the top of the head needs to be connected to the heap of its tail in the original tree. Finally, the value of the nodes is eliminated so the incremental cost to reach them is the only information remaining.

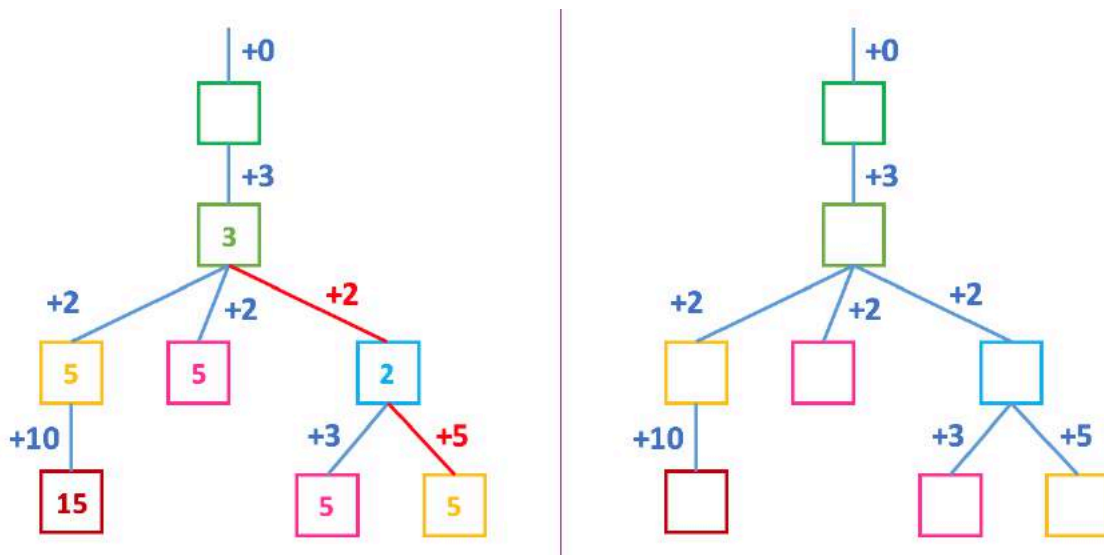


Figure 31. Final arrangement of heaps.

- *Execution of Breadth-First Search using heaps*

The purpose of arranging the sidetracks in a heap was to reduce the out-degree of the initial tree. The out-degree is the maximum number of children a node in a tree has. In the initial tree, the out-degree was 4 but with the new arrangement this number gets lowered to 3, which makes the execution of Breadth-First search faster.

Breadth-First search algorithm is going to be executed now using heaps. Afterwards, the k-paths will be obtained. The pseudocode of this process is included bellow.

```

function BFS-Heap(H_final,min_cost):
  push root(H_final).0 onto heap H;

  while(H != empty):

    pop n.c form top of H;           //The node on top of H is taken out
    path_cost[i] = min_cost + c1;   //The cost of each path is recorded

    for each child s of n with cost c':
      push s.c' onto H;             //H is updated
    end for

    i++;

  end while

  print path_cost;
  
```

Pseudocode 9. Breadth-First Search with heaps for Eppstein's Algorithm.

min_cost refers to the minimum cost from any node to the destination. In this case is 7 from x1. The images bellow will explain the functioning specified in the pseudocode. The basic idea is to go through the heap tree obtained before and building and updating a new heap tree with the children of its nodes.

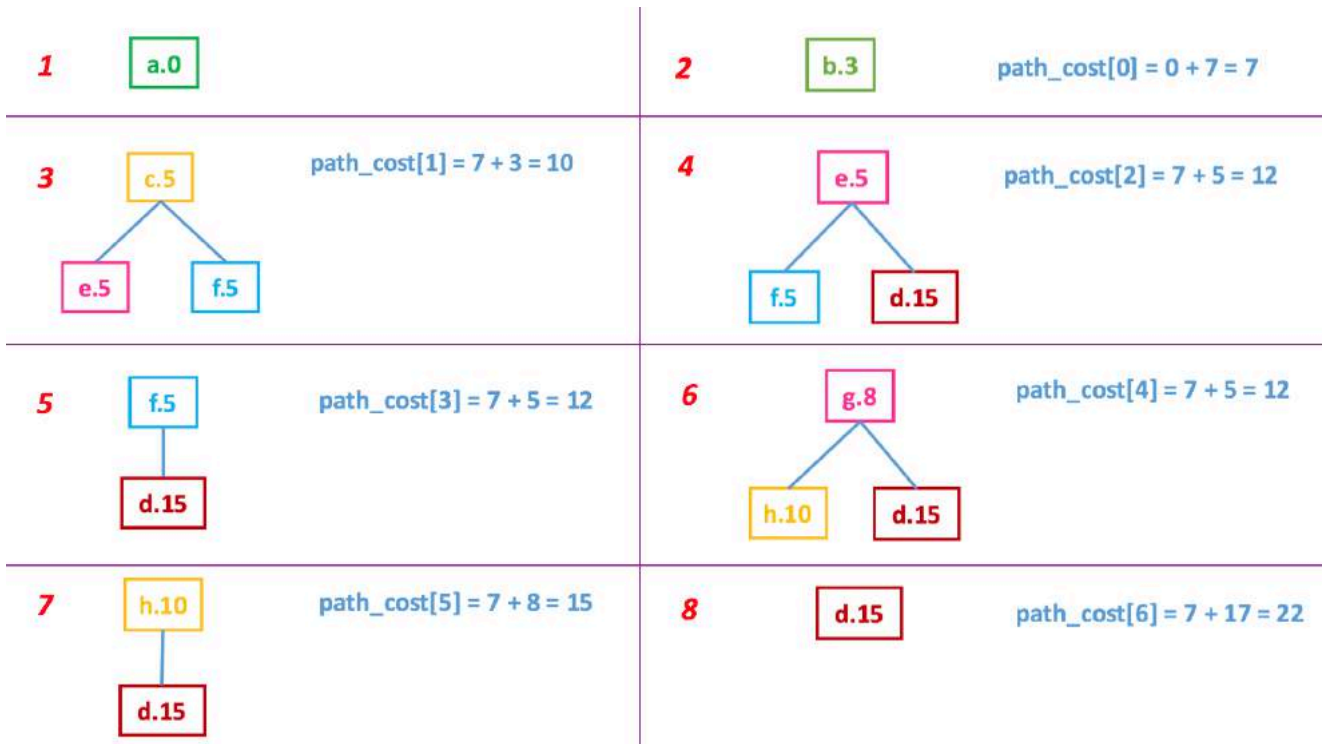


Figure 32. Breadth-First Search on Eppstein's Algorithm with heaps.

Figure 32 shows the different paths that can be followed until the destination is reached and which is the cost of any of them. The nodes that are being traversed to obtain that cost can be extracted with the association of the sidetrack assigned to each heap. For instance, the path with cost 22 is the one that starts in x1 and goes through x4 and x5 to reach x6. It corresponds with the heap d.15 which is the sidetrack that needs to be traversed in order to follow that path. Even though finding the nodes included in each path, this seems to be a complex way of keeping track of the different paths found.

Time Complexity

First of all, the time complexity needed to execute the Breadth-First search algorithm for the first time is:

$$O(|E| + |V| \log(|V|))$$

where E are the edges and V, the nodes.

When trying to compute the execution time with an unbounded tree of sidetracks the time complexity can be calculated as:

Unbounded tree Time Complexity analysis:

Total pops = size of Heap k
 Total pushes = $k \cdot E$
 Total heap size = $O(k \cdot E)$
 Cost of pop = $O(\log(k \cdot E)) = O(\log k + \log E)$
 Cost of push = 1

Total cost = Total heap size + Cost of pop + Cost of push =
 $= O(k \cdot E) + O(\log k + \log E) + 1 =$
 $= O(k \cdot E + k \cdot \log k + k \cdot \log E) =$
 $= O(k \cdot E + k \log k)$

Formula 14. Time complexity for unbounded heap for Breadth-First Search for Eppstein's Algorithm.

When applying the modifications specified above in order to obtain a bounded tree, the time complexity that results in executing the Breadth-First algorithm to a bounded tree can be calculated as:

Bounded tree Time Complexity analysis:

Total pops = size of Heap k
 Total pushes = $k \cdot \text{constant}$
 Total heap size = $O(k)$
 Cost of pop = $O(\log k)$
 Cost of push = 1

Total cost = Total heap size + Cost of pop + Cost of push =
 $= O(k) + O(\log k) + 1 =$
 $= O(k + k \log k)$

Formula 15. Time complexity for bounded heap for Breadth-First Search for Eppstein's Algorithm.

The key factor is that in this last formula, the total time for computation does not depend additionally on the number of edges and therefore, on the number of nodes, although the size of the heap depends on them. All in all, this reduces considerably the time complexity.

Advantages and Disadvantages

Advantages

Eppstein’s algorithm provides a set of different paths to a certain destination, which makes it suitable for the purpose of this project.

Also, it is provided with an asymptotic performance that makes it fit for situations where the main objective is obtaining a large number of paths.

Disadvantages

The implementation of this algorithm, simple as it seems explained, would not be possible for the problem this projects wants to solve. The reason is that Eppstein’s algorithm cannot be applied for time dependent problems, which is the type the company faces.

This algorithm needs to apply a reverse Breadth-First search on the graph at the beginning, which is impossible when the edges are subject to time dependencies. Specifically, this cannot be done when the edges that connect the different nodes are routes with a start and end time. This way, as the time the destination is reached cannot be predicted if the starting time is unknown, the routes that actually reach it are uncertain.

This is the main reason this algorithm is ruled out from being considered as potential substitute of Dijkstra’s algorithm.

5.5.4.3 Iterative Enumeration Algorithm (IEA)

IEA	
Classification	Enumeration Algorithm
Solutions	K-Shortest Path
Graph	Directed Weighted
Time	$O(E + k \cdot V \cdot \log V)$

Table 14. IEA specifications.

The Iterative Enumeration algorithm or IEA, is an algorithm that computes the k-shortest paths from a single source to all the rest of the nodes in a graph. It has been designed to be applied in Transportation Networks. These networks are graphs whose edges or routes are time dependent and subject to the type of transportation used in each route (bus, train, car...). Also, they are designed to avoid cycles in the solutions even if this increase their time complexity, as users are not willing to accept this type of routes.

IEA, as well other algorithms applied on Transportation Networks, are characterized by the employment of labels associated to each route and stored on each node. This is different from the rest of the algorithms covered in this study, that made used of a static graph. These labels have information regarding times and means of transportations. But one characteristic that differentiates this algorithm from others included in Transportation Networks is that IEA provides k different paths and not just one.

As its name indicates, IEA has an important factor based on **iteration** of different operations to reach the different solutions. The labels included in IEA for each route cover the following information:

- If the node has been visited or not.
- A set of sublabels obtained from each route that include:
 - Accumulated cost if that route would be taken.
 - Predecessor node, which is the start of the route.
 - The predecessor's label used to obtain the next node.
 - Order number associated to the label.
 - If the label has been considered or not.

There are two ways of implementing this algorithm: one admitting cycles in the solutions and another one, avoiding them. The first one comes with the simplest implementation, while the second one requires additional settings. They both will be analyzed in the following sections.

Original Implementation

Provided with the most straightforward structure, this implementation is the simplest one but one that could include cycles in the solutions and additional memory usage because of them.

- **Pseudocode**

The following lines present the pseudocode that gives an indication of the overall functioning of the algorithm.

```

function IEA(V,R,source,destination):

    labels = Array(); //Array of labels associated to each node.
    nodes = Array();

    for each node n in V:
        nodes[n] = [visited,labels]; //nodes array hold information about if it
                                     //has been visited and the labels
                                     //associated to it.

    end for

    next_node = source;

    while(next_node != destination)

        for each route r in R whose start = next_node:
            label = new createLabel(r);
            nodes[end(r)][labels].append(label); //A new label is created for the node at
                                                    //the end of the route.

        end for

        all_labels = Array(); //Temporal

        for each node n in V:
            for each label l in nodes[n][labels]:
                all_labels.add(l);
                next_label_considered = min(all_labels.acc_weight); //From all labels from all nodes, the
                                                                    //label with less accumulated cost is taken.
            end for
        end for

        next_node = next_label_considered.getNode()

        nodes[next_node].visited = true; //The node to which the label considered
                                         //belongs to is set to visited.
        next_label_considered.label_considered = true; //The label is marked as considered.

    end while

    //Now next_node = destination

    path = new Array();
    path.append(next_node);

    predecessor = next_node;

    while(predecessor != source)

        predecessor = next_label_considered.getPredecessor();
        next_label_considered = predecessor.getLabelConsidered(); //This takes into account the predecessor
                                                                    //and the label needed to check the next
                                                                    //predecessor.

        path.append(predecessor);

    end while
```

```
path = reverse(path);  
reset_for_next_iteration();
```

Pseudocode 10. IEA.

The process of creating a label needs the route's information, such as the weight of the route. The first node to be selected is the origin point. When a node is selected, labels are assigned to the nodes attached to it, that is, its neighbors. A label stores information about the predecessor node and the label by which the current node has been considered. The next node to be selected is the one with the unconsidered label with the lowest accumulated weight. If this node is the end point, it means the destination has been reached. Then, the next step would be to extract the path by which the destination has been reached. The process would check the predecessor node indicated in this label and the label of this predecessor that has been used to reach the destination. So if the predecessor is shown as x2(3) it means that the third label of node x2 was the one by which the destination was reached. So the predecessor of the third label is the one that needs to be analyzed in order to find the list of predecessors until the source is reached. Finally, the path will be obtained is this chain of predecessor is reversed.

The following section shows this process with the graph being employed by the previous algorithms.

- **Operation**

The graph employed to apply the IEA algorithm is the one presented below, as the previous sections show.

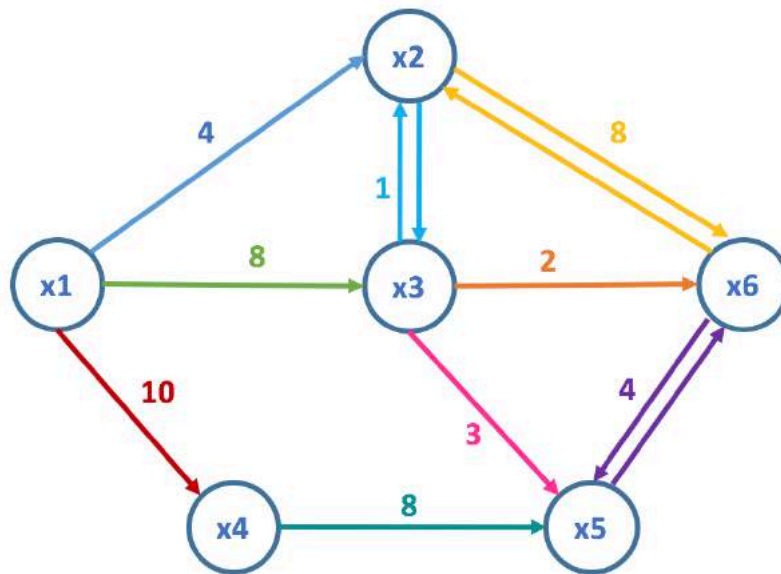


Figure 33. Graph where IEA is applied.

The source will be x_1 and the destination, x_6 , so the functioning of this algorithm can be shown. In this case, the development of the labels associated to each node will be provided, instead of the progress of the algorithm on the graph.

◆ **Initiation**

As the pseudocode indicates, the first step consists on populating the nodes adherent to the source (x_1) with the first label (Figure 34). As it can be seen, x_1 does not have any label assigned and it will remain so for the rest of the algorithm development.

	x1	x2	x3	x4	x5	x6
1		$F; \{4, x_1[1], 1, F\}$	$F; \{8, x_1[1], 1, F\}$	$F; \{10, x_1[1], 1, F\}$		

Figure 34. First step labels on IEA.

Nodes x_2 , x_3 and x_4 are the ones that connect directly to x_1 , so they will be the ones to receive a label first. As the accumulated weight is 0 so far, the weight associated to each label corresponds to the exact weight of the route from x_1 to each one of them.

Figure 35 provides a detailed explanation of the structure of a label.

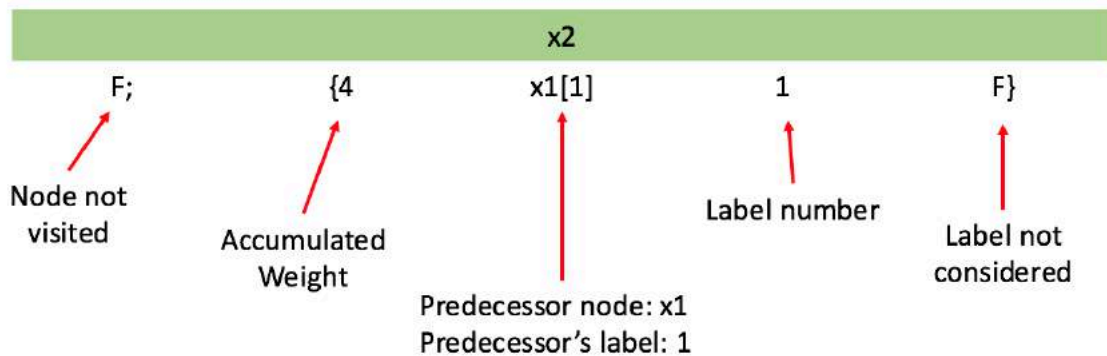


Figure 35. Label structure in IEA.

◆ **Second step**

The next step consists on choosing from all this labels the one that has less accumulated weight. In this example, this label belongs to x2, so it is highlighted in bold. Also, this node is set to visited and the label is also marked as considered so it is not used in the future. Figure 36 depicts this, and also the new labels that are associated to nodes x3 and x6, as these are the nodes connected directly to x2.

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],1,F}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		
2		T;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		F;{12,x2[1],1,F}
			{5,x2[1],2,F}			

Figure 36. Second step labels on IEA.

The new labels associated to x3 and x6 (marked in blue) have an accumulated weight calculated as sum of the weight of the label just chosen (the one belonging to x2), which is 4, and the weight of the route that connects x2 to each one of them. For instance, the accumulated weight on x3's second label is:

$$4(\text{weight on x2 label}) + 1(\text{weight on route from x2 to x3}) = 5$$

Figure 37 shows a detailed analysis con the second label of node x3.

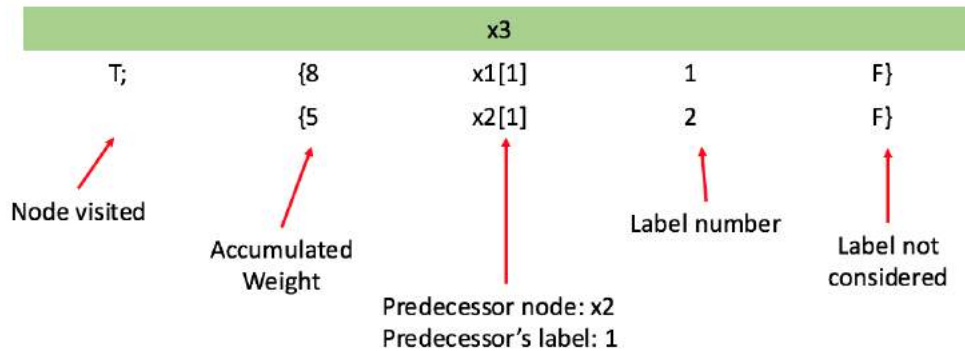


Figure 37. Label structure in second step in IEA.

An important fact to take into account is the information the predecessor gives as it will prove crucial when obtaining the path to the destination. In this second step, both new labels in x_3 and x_6 included as a predecessor x_2 and the label by which they have been assigned is the first one.

◆ Choosing the next node

The node to be selected is the one which has not been visited yet and has the unconsidered label with the lowest accumulated weight.

Taken this into account Figure 38 provides the labels created for the next steps of the first iteration of the algorithm.

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],1,F}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		
2		T;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		F;{12,x2[1],1,F}
3		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
4		T;{4,x1[1],1,T}	{5,x2[1],2,T}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	T;{12,x2[1],1,F}
			{5,x2[1],2,T}			{7,x3[2],2,T}

Figure 38. First iteration labels of IEA.

Step 3 functions the same way as step 2. The second label in x_3 is chosen and new labels are included in x_2 , x_6 and x_5 . These labels have as predecessor x_3 and the label used to obtain them, the second of x_3 .

Finally, in step 4 the next node chosen is x_6 , which is the destination node. Then the first iteration of the algorithm has finished and no new labels are added. The first path to the destination can be obtained thanks to the information stored in the labels.

◆ Obtaining the path

In order to find the path to the destination, the first thing that needs to be done is to analyze the label that has been selected on the destination. It needs to be marked as considered, and the node as visited. Then the predecessor and its label need to be taken into account. In this case, the label indicates x_3 is the predecessor of x_6 and that its second label is the one that must be analyzed afterwards.

Node x_3 's second label states that its predecessor is x_2 and its label to be analyzed is the first one. When considering this label, the predecessor found is x_1 , so the first path, and the shortest one, has been found. The total cost is shown in the label selected in x_6 .

$$\text{Path 1} = [x_1, x_2, x_3, x_6] \quad \text{Cost} = 7$$

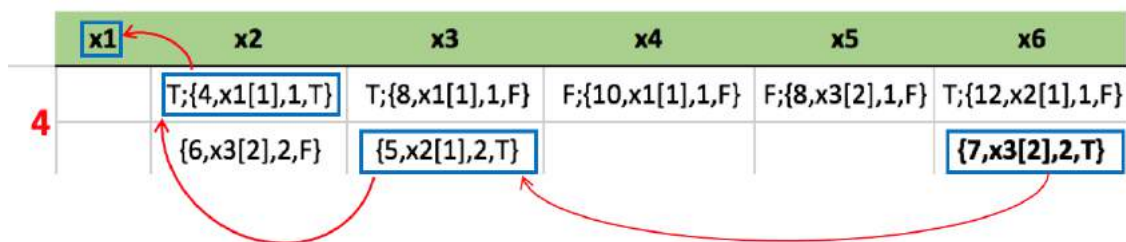


Figure 39. Building of path in first iteration in IEA.

◆ Resetting for next iteration

As stated before, this algorithm relies entirely on repetition using information stored previously, specifically the one provided by the labels added before and that have not been

considered yet. But in order to make use of this labels, it is necessary to prepare the stored information.

To be precise, those nodes that have been included in the path that has just been found need to be marked as unvisited. The labels that have been used so far, that is, the ones that have been marked as considered, remain unchanged, still considered.

Figure 40 shows the result of applying this to the stored labels.

	x1	x2	x3	x4	x5	x6
5		F;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
		{6,x3[2],2,F}	{5,x2[1],2,T}			{7,x3[2],2,T}

Figure 40. Resetting of labels after first iteration in IEA.

Coincidentally, the nodes that have been included in the first path obtained were the ones visited during the first iteration. However, this not always happens. What needs to be noticed is that a node that is visited during a iteration does not need necessarily be included in the path resulting from that iteration.

This can be seen in the following iteration.

◆ Second iteration

Figure 41 shows the labels that have been assigned to each node during all the steps of the second iteration.

	x1	x2	x3	x4	x5	x6
5		F;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
		{6,x3[2],2,F}	{5,x2[1],2,T}			{7,x3[2],2,T}
6		T;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
		{6,x3[2],2,F}	{5,x2[1],2,T}			{7,x3[2],2,T}
			{7,x2[2],3,F}			{14,x2[2],3,F}
7		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
		{6,x3[2],2,F}	{5,x2[1],2,T}		{10,x3[3],2,F}	{7,x3[2],2,T}
		{8,x3[3],3,F}	{7,x2[2],3,T}			{14,x2[2],3,F}
						{9,x3[3],4,F}
8		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{6,x3[2],2,F}	{5,x2[1],2,T}		{10,x3[3],2,F}	{7,x3[2],2,T}
		{8,x3[3],3,F}	{7,x2[2],3,T}			{14,x2[2],3,F}
			{11,x5[1],4,F}			{9,x3[3],4,F}
						{12,x5[1],5,F}
9		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	T;{8,x3[2],1,T}	T;{12,x2[1],1,F}
		{6,x3[2],2,F}	{5,x2[1],2,T}		{10,x3[3],2,F}	{7,x3[2],2,T}
		{8,x3[3],3,F}	{7,x2[2],3,T}			{14,x2[2],3,F}
			{11,x5[1],4,F}			{9,x3[3],4,F}
						{12,x5[1],5,F}

Figure 41. Labels for second iteration of IEA.

The path that is obtained after this iteration is:

$$path\ 2 = x1, x2, x3, x2, x3, x6 \quad Cost = 9$$

Obviously, the algorithm is providing solutions with cycles, which is completely undesirable in transportation networks. Additionally, the presence of cycles implies a higher number of labels that need to be stored, which results in an increase of memory usage required to execute the algorithm.

The following section provides a modification in the algorithm to avoid the presence of loops and cycles.

Cycle Avoidance Implementation

In order to avoid the problem faced in the previous example, that of cycles appearing in the solutions as well as an unnecessary memory consumption, this second implementation provides the necessary mechanisms that will ensure a more efficient performance.

The main idea consists on adding a condition to the process of extending labels to each node. Specifically, a label will not be extended to a node if this node is included in the predecessors list of the label being considered.

The pseudocode will be as follows:

- Pseudocode

```

function IEA_NoCycles(V,R,source,destination):

  labels = Array(); //Array of labels associated to each node.
  nodes = Array();

  for each node n in V:
    nodes[n] = [visited,labels]; //nodes array hold information about if it
    //has been visited and the labels
    //associated to it.

  end for

  next_node = source;

  while(next_node != destination)

    for each route r in R whose start = next_node:
      if(nodes[end(r)] not in next_node.get_predecessor()) //Only if the node it is not included in the
      //predecessors list of next_node.

        label = new createLabel(r);
        nodes[end(r)][labels].append(label); //A new label is created for the node at
        //the end of the route.

      end if
    end for

    all_labels = Array(); //Temporal

    for each node n in V:
      for each label l in nodes[n][labels]:
        all_labels.add(l);
        next_label_considered = min(all_labels.acc_weight); //From all labels from all nodes, the
        //label with less accumulated cost is taken.

      end for
    end for

    next_node = next_label_considered.getNode()

    nodes[next_node].visited = true; //The node to which the label considered
    //belongs to is set to visited.

    next_label_considered.label_considered = true; //The label is marked as considered.
  
```

```

end while

//Now next_node = destination

path = new Array();
path.append(next_node);

predecessor = next_node;

while(predecessor != source)

    predecessor = next_label_considered.getPredecessor();
    next_label_considered = predecessor.getLabelConsidered(); //This takes into account the predecessor
                                                                and the label needed to check the next
                                                                predecessor.

    path.append(processor);

end while

path = reverse(path);
reset_for_next_iteration();
  
```

Pseudocode 11. IEA without cycles.

The if condition highlighted in the code implements the idea explained before. In the following subsection the result of applying this modification will be shown.

- **Operation**

◆ **First iteration**

The operation of the algorithm is the same as in the previous implementation. The first difference comes in step 3. The node selected is x3 by means of its second label, which indicates the predecessor of x3 if that label is used is x2, and that of x2 using that particular label is x1. And so, as x2 is in the predecessor list of x3 [x1, x2], the label that had been previously extended to x2 now it is not.

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],1,F}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		
2		T;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		F;{12,x2[1],1,F}
			{5,x2[1],2,F}			
3		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
		{6,x3[2],2,F}	{5,x2[1],2,T}			{7,x3[2],2,F}

Figure 42. New labels extended to nodes in step 3 in first iteration of IEA.

The labels marked in blue are the new labels extended in step 3, and the label crossed out in orange is the label extended to x2 in the previous implementation. This label will no longer be taken into account.

◆ **First path**

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],1,F}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		
2		T;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		F;{12,x2[1],1,F}
			{5,x2[1],2,F}			
3		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
			{5,x2[1],2,T}			{7,x3[2],2,F}
4		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	T;{12,x2[1],1,F}
			{5,x2[1],2,T}			{7,x3[2],2,T}

Figure 43. First iteration labels for IEA without cycles.

For the first iteration, the removal of the second label of x2 does not affect the path obtained. It is the same:

$$Path\ 1 = [x1, x2, x3, x6] \quad Cost = 7$$

However, the difference will be found in the second iteration where the number of labels extended is going to get considerably reduced and no cycles will appear in the second path obtained.

◆ Second iteration

	x1	x2	x3	x4	x5	x6
5		F;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
			{5,x2[1],2,T}			{7,x3[2],2,T}
6		F;{4,x1[1],1,T}	T;{8,x1[1],1,T}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
		{9,x3[1],2,F}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
						{10,x3[1],3,F}
7		F;{4,x1[1],1,T}	T;{8,x1[1],1,T}	F;{10,x1[1],1,F}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{9,x3[1],2,F}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
						{10,x3[1],3,F}
						{12,x5[1],4,F}
8		T;{4,x1[1],1,T}	T;{8,x1[1],1,T}	F;{10,x1[1],1,F}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
						{10,x3[1],3,F}
						{12,x5[1],4,F}
						{17,x2[2],5,F}
9		T;{4,x1[1],1,T}	T;{8,x1[1],1,T}	T;{10,x1[1],1,T}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
					{18,x4[1],3,F}	{10,x3[1],3,F}
						{12,x5[1],4,F}
10		T;{4,x1[1],1,T}	T;{8,x1[1],1,T}	T;{10,x1[1],1,T}	T;{8,x3[2],1,T}	T;{12,x2[1],1,F}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
					{18,x4[1],3,F}	{10,x3[1],3,T}
						{12,x5[1],4,F}
						{17,x2[2],5,F}

Figure 44. Second iteration without cycles IEA.

Even though the number of labels in node x6 is the same as in the previous case, those of the rest of the nodes have been reduced. This is very advantageous in terms of performance as the memory usage decreases.

◆ **Second path**

The second path obtained in this second iteration is extracted by means of the labels attached to each node. The label selected in x6 indicates its predecessor is x3. Label 1 in x3 determines the predecessor is x1. The cost is 10.

$$\text{Path 2} = [x1, x3, x6] \quad \text{Cost} = 10$$

	x1	x2	x3	x4	x5	x6
		T;{4,x1[1],1,T}	T;{8,x1[1],1,T}	T;{10,x1[1],1,T}	T;{8,x3[2],1,T}	T;{12,x2[1],1,F}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
10					{18,x4[1],3,F}	{10,x3[1],3,T}
						{12,x5[1],4,F}
						{17,x2[2],5,F}

Figure 45. Building of path in first iteration in IEA without cycles.

The cost is higher than the second path obtained with a cycle. Also, it can be seen that even if all nodes were visited in the process, not all of them are included in the path.

◆ **Third Iteration**

After finding the second path, the third iteration starts with a reset of the labels. As only x3 and x6 were part of the path apart from the source, only these two nodes will be set to unvisited. Then, x3 and x6 are the only ones whose unconsidered labels can be selected in the future. All belonging to x3 have already been selected so x6's labels are the only ones that could be chosen now.

When there are two labels with the same weigh, it does not matter which one is selected. In this example, the first to be analyzed is the one that is chosen.

	x1	x2	x3	x4	x5	x6
11		T;{4,x1[1],1,T}	F;{8,x1[1],1,T}	T;{10,x1[1],1,T}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
					{18,x4[1],3,F}	{10,x3[1],3,T}
						{12,x5[1],4,F}
						{17,x2[2],5,F}
12		T;{4,x1[1],1,T}	F;{8,x1[1],1,T}	T;{10,x1[1],1,T}	T;{8,x3[2],1,T}	T;{12,x2[1],1,T}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
					{18,x4[1],3,F}	{10,x3[1],3,T}
						{12,x5[1],4,F}
						{17,x2[2],5,F}

Figure 46. Third iteration IEA without cycles.

After one label selected, another path is found.

◆ **Third path**

Now the path has a cost of 12 and the nodes involved are:

$$Path\ 3 = [x1, x2, x6] \quad Cost = 12$$

◆ **End of iterations**

The iterations in IEA continue until no new labels are added and all of them have been considered. In very large graphs, this number could get too high, that is why it would be better to outbound the maximum number of solutions. That is, the k solutions to be found.

In this particular example, the number of iterations, that is, the number of paths found have been 7. In Appendix A the entire solution is shown.

Figure 47 shows IEA operation without cycles.

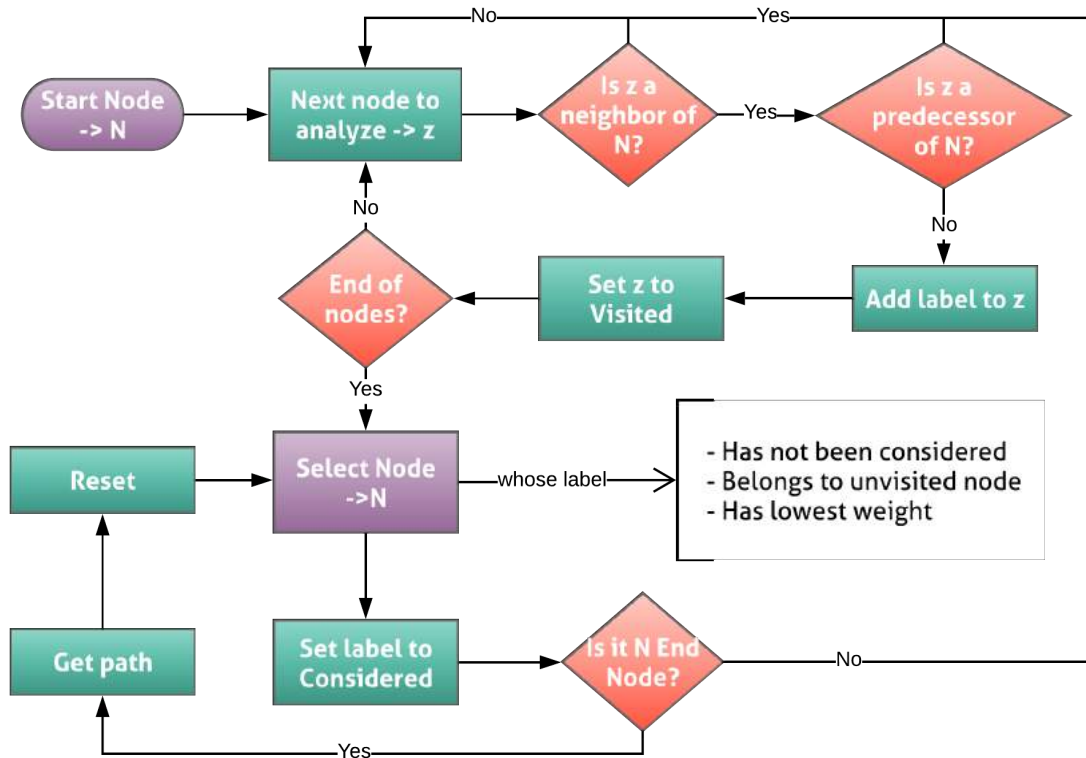


Figure 47. IEA operation flowchart.

Time complexity

The time complexity of this algorithm is:

$$O(|E| + k \cdot |V| \cdot \log|V|)$$

Formula 16. Time complexity for IEA.

Where k is the number of solutions expected to be found. This formula is very similar of that of Breadth-First search studied for Eppstein's algorithm (Formula 15), but adding the k factor, that is, the number of different paths to be obtained.

Advantages and Disadvantages

Advantages

This algorithm provides k solutions for finding the shortest path between two nodes in a single execution of the algorithm. The inner iterations the algorithm applies are equivalent to executing Dijkstra or another single-path-algorithm a certain number of times. Therefore, the simplicity of this solution makes it very attractive for solving the problem this project faces.

Its time complexity is similar to that of Breadth-First search but worse than that of Eppstein's. However, it has been proved in [2] that IEA running time is better in practice.

The fact that this algorithm has been built with Transportation Networks in mind, implies that it can be easily adapted to time-dependent and transportation modulated scenarios. This is a great advantage in what this project is concerned.

Disadvantages

As stated by Formula 16, the time complexity of the algorithm relies heavily on the number of paths that need to be found. If this number is too high, the algorithm might cause memory usage problems due to the increasing number of labels that need to be stored for future implementations.

On the other hand, the relatively newness of its implementation and scarce information about its performance results, leave blank spaces about how good it will do in other topologies, and therefore, how well it could be implemented in the graph provided by the company.

Finally, the use of the modified version of the algorithm in order to avoid cycles, does not ensure that the first results to obtained are the shortest ones. This is an important drawback that could affect the company's performance.

5.6 ALGORITHMS COMPARISON

All the algorithms that have been studied in the previous section were analyzed in order to provide the advantages and disadvantages of employing them as substitute of Dijkstra's algorithm, currently being used in the company's solution.

Table 15 provides a summary of the more important characteristics to be taken into account in order to choose the more suitable algorithm for the job of substituting Dijkstra.

Algorithm	Time Complexity	Advantages	Disadvantages
Breadth-First	$O(V + E)$	- Finds the best solutions.	- Memory usage. - Unweighted graph.
Breadth-First	$O(V + E)$	- Shorter time and less memory usage.	- Might not find a solution. - Infinite loops.
A*	$O(b^d)$	- Fast and efficient. - Good for complex problems.	- Dependency on heuristic function.
Bellman-Ford	$O(V \cdot E)$	- Negative weights. - Wider inputs can be introduced.	- More complex than Dijkstra.
Floyd-Warshall	$O(V ^3)$	- All-pairs shortest path in one execution.	- No information about paths (original). - Slower than other providing same results.
Johnson	$O(V ^2 \cdot \log V + V \cdot E)$	- All-pairs shortest path in one execution. - Faster than Floyd-Warshall. - Negative weights.	- Execution depends highly on number of edges.
Yen	$O(KN \cdot V ^2)$	- Provides more than one result.	- Uncertainty when removing edges arbitrarily.
Eppstein	$O(k + k \cdot \log k)$	- Provides more than one result. - Good when the number of paths needs to be high.	- Complex implementation. - Not suitable for time-dependent situations.
IEA	$O(E + k \cdot V \cdot \log V)$	- Provides more than one result with just one execution.	- Complexity depends on number of paths. - Memory usage depends on labels.

		<ul style="list-style-type: none"> - Good performance in practice. - Applicable to time-dependent situations. 	<ul style="list-style-type: none"> - Scarce performance applicability and performance results.
--	--	---	---

Table 15. Algorithms comparison.

In order to pick the best algorithm to substitute Dijkstra, it is necessary to highlight what are the main objectives the company wants to obtain. Then, given the specifications, some algorithms can be ruled out.

- First of all, the company works with a weighted graph.
 - Algorithms such as Breadth-First and Depth-First search are not taken into account. Even if they could be modified to be applied to a weighted graph, their drawbacks far surpass the effort of applying the necessary modifications.
- The weights in the graph are always positive. Then, those algorithms whose greatest advantage is that of providing a shortest path between two nodes in graph with negative weights are not the ones needed at the moment.
 - Therefore, Bellman-Ford algorithm is not going to be further examined.
- Also, the main goal is to find different routes between two nodes in a graph. As a result, the company is not interested in multiple solutions between all the nodes in a graph. This is because a user usually looks for a certain itinerary between only two points in the graph.
 - Then, algorithms such as Floyd-Warshall and Johnson get rejected.
- The possibility of providing the user with more than just one result of connections between two nodes in the graph is also being considered. Therefore, those algorithms that provide k solutions are very attractive at this point.
 - Such as Yen, Eppstein and Iterative Enumeration Algorithm (IEA).
- A better, more efficient and faster solution is being looked for, given the problems Dijkstra's implementation is causing.
 - A* algorithm would be a good option regarding these characteristics.
 - Not Yen's algorithm though, as the uncertainty it shares with the current implementation of Dijkstra is one of the principal reasons to be avoided.

- Eppstein's algorithm should also be ruled out at this point due to the complexity of its implementation.

As a conclusion, the algorithms that are worth to be taken into account are A* and IEA. A* has been proven to be very fast and efficient and, given a correct heuristic function, it could provide high-quality solutions. However, it still calculates only one result.

On the other hand, IEA has the characteristics that render it more desirable than the rest. Not only it provides more than one result with only one execution but it also does this within a good time range. Even though there are some unknown aspects about its performance in certain types of graphs, the relatively simplicity of the one used in the company could become the perfect scenario to prove its efficiency.

For these reasons, it can be concluded that IEA is the algorithm chosen and the one to be applied in the company's graph.

Chapter 6. IMPLEMENTATION OF IEA ALGORITHM

The Iterative Enumeration Algorithm (IEA) is the algorithm chosen among all the ones studied in the previous section thanks to its qualities and potential solution of the problems being encountered by the company.

This section will describe at the beginning what is the position of this algorithm in the overall picture of the procedure for providing routes between two points. Then, it will be explained the way IEA might supply a solution for the problems enumerated and explained in section 5.4.2. Then, it will be specified what are the necessary modifications to be added to the algorithm to make it time-dependent and suitable for finding the correct routes connecting two points given the time of traveling.

Afterwards, a detailed explanation of the implementation of the algorithm will be provided in order to show how it has been modeled to the internal structure to which Dijkstra was subjected to. First, this implementation will be applied with the original data structure, followed by its adaptation to the enhanced data access arrangement. The results that compare both algorithms' performance will be displayed and discussed in Chapter 7.

Finally, some additional features added to the algorithm will be explained and how they have been developed.

6.1 THE ROLE OF IEA

As explained before, IEA will be the algorithm in charge of providing combined routes between two cities or regions. First of all, it is necessary to specify its position among the rest of functions and procedures that also help to procure a connection between them.

Figure 48 depicts the flow of actions necessary to extract all the information about the routes that connect two points. In the first place, after the user has entered the origin and destination, a function is called to check if there are direct results between these points as they are the

ones usually preferred. These results are obtained directly from the routes table in the database but they might not match the current price. This may happen because the providers might have made changes after the routes have been inserted into the table. That is why scraping is necessary. This action allows to ask for the newest routes characteristics so an accurate description can be provided to the user.

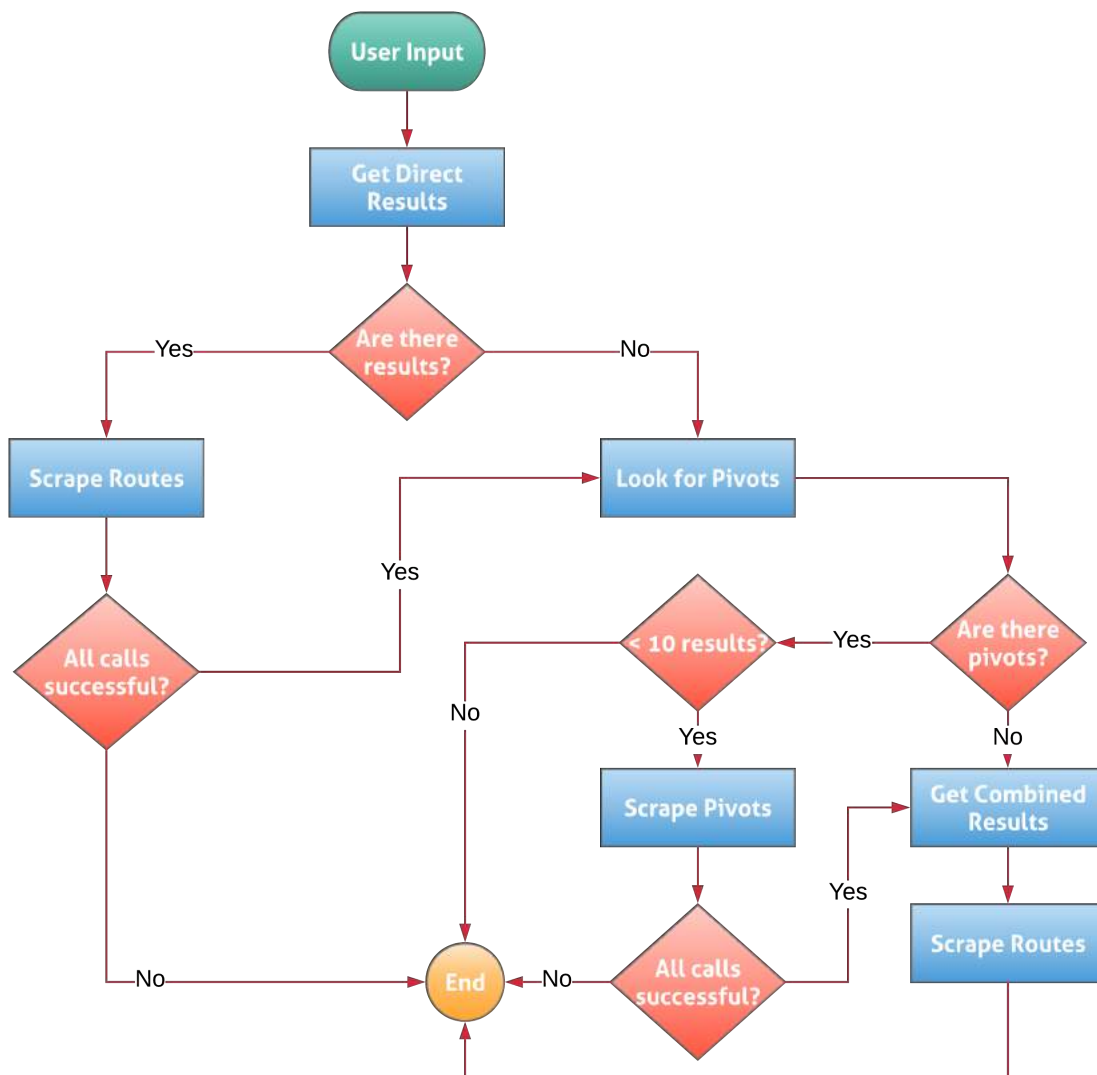


Figure 48. Calculation of routes flowchart.

If all these scrapings have been successful, that is, if all the information has been successfully retrieved, then pivots are looked for. Pivot nodes are certain points in the graph that have been selected due to its importance in terms of high influx of routes, and that are chosen to force some combined trips to go through them. Therefore, if no pivots could be found then the algorithm to calculate the combined routes is called. Dijkstra took this responsibility before but now IEA will take the job.

In case less than 10 pivots have been found then some scraping will be done again, but this time looking for pivots instead of direct results. Again, if all calls have been successful, then more combined results can be calculated and so IEA will be called again.

6.2 PROBLEMS SOLVED BY IEA

As explained in section 1.1, the company faces certain problems that need to be solved with the implementation of the new algorithm found. In this section, the solutions proposed by the algorithm will be specified and how they may help overcome the difficulties the company is facing.

Time complexity and system collapse

It was found that Dijkstra consumed a lot of resources when being executed and that lead to an inevitable system collapse every time more than three or four combined results were found between two nodes in the graph. This happened due to the continuous calls that needed to be made to the algorithm every time a new result had to be found, as the algorithm only provides one solution at a time.

Additional calls to the algorithm

In order to obtain solutions that were different in terms of time of traveling, cost or waiting time, an array of different weights was created and passed into Dijkstra implementation.

Time	Cost	Waiting Time
0.5	0.25	0.25

Table 16. Weights passed into Dijkstra.

With the example shown in the previous table, the path that Dijkstra would supply would certainly be the shortest as more weight is applied in the first element of the array. Specifically, Dijkstra was called up to three times using a different set of this array every time so different types of paths could be obtained. For example, the first path procured could be the shortest, the next the cheapest one and so on.

With IEA the presence of more than one of these arrays is not needed and the algorithm is called just once. The reason is because the algorithm outputs several solutions with just one execution. These solutions are bound to be different in terms of time and cost and obtained without the uncertainty of eliminating edges, as it was done with the current implementation of Dijkstra. Therefore, the algorithm just receives one array with the equivalent cost of the time traveling.

Time	Cost	Waiting Time
<i>cost_per_hour</i>	1	<i>cost_per_connection</i>

Table 17. Weights passed to IEA.

The *cost_per_hour* is a value the company has assigned to the total time spent on a vehicle. Then, the price of the travel is assigned the maximum weight. Finally, to the waiting time a *cost_per_connection* is assigned. With this structure, the variety of the solutions is ensured.

Impossibility of charging more results

With the current implementation of Dijkstra it was not possible to load more results once the algorithm had already been executed. This is due to an absence of a structure able to store the information extracted already by the algorithm. This implies that the same results are not obtained if the algorithm is executed more times.

The structure of IEA based on labels assigned to each node allows for a simple solution to this problem. Every time the algorithm is finished with its execution – limited by time or number of extracted results –, the labels assigned to each node can be stored so the ones that have not been considered yet can be used in a future execution. This way more paths can be obtained without having to execute the algorithm since the beginning.

This can be very useful if an infinite scroll is implemented as the algorithm just receives the overall label state of the last iteration and only needs to carry out the previous execution as it had never been stopped.

The reason why the algorithm is stopped before it calculates all possible solutions is because, in this type of graphs where the number of nodes and edges is considerably high, the amount of solutions to be found is too elevated and getting all of them would take too much time.

Using pivot nodes

Due to the uncertainty generated by randomly removing the edges from the graph in Dijkstra's implementation, pivot nodes were needed, as mentioned in section __. Now that IEA is known to procure all the existing paths eventually without the risk of repeating results or omitting some, pivot nodes are found to be unneeded. The routes returned by the algorithm will be the best obtained and if they are good enough they will go through the important cities and regions without the need of forcing them to do so.

6.3 ORIGINAL IMPLEMENTATION

IEA has been first implemented with the data structure provided already by the company. This data structure refers to the databases used and the entries in each one of the tables. This way it can be possible to analyze the IEA performance compared to the current implementation of Dijkstra in the same circumstances.

The main information of this data structure to be taken into account is the specifications of the routes that are introduced into the algorithm. This information includes:

- **Route Identification Number.** This number uniquely identifies each route. It is generated at the time the route is inserted into the table.
- **Origin/Destination Identification Number.** Every node in the graph has an assigned id that helps to identify them.
- **Start/End Date.** This information refers to the day, month and year the route starts and ends.
- **Start/End Time.** This alludes to the time in hours, minutes and seconds the route departs and reaches its destination, respectively.
- **Duration.** The total trip time in minutes.
- **Price.** It is included in the currency of the country the route belongs to.
- **Vessel.** This identifies the vehicle used in the route: plane, bus, train or taxi.
- **Origin/Destination Country.** Knowing this information allows to take different time-zones into consideration.
- **Company.** It provides the name of the company to which the trip belongs to.

All of this information is going to be used by the algorithm. When the modified data structure is finally applied (section 6.4), some of this data will no longer be necessary and, therefore, not fed in into the algorithm. But for now, this is everything that is needed to make IEA work with the original data structure.

In this original structure, the information about the routes is taken from different tables and grouped in a MySQL view so it can be accessed directly, instead of launching a query to all of these tables. The query launched to this view extracts all the routes in the next 2 days since the travel date entered by the user and they are introduced into IEA as input.

6.3.1 ADAPTATION OF IEA TO TIME-DEPENDENT STRUCTURE

As stated before, a time-dependent structure is one that is subjected to certain times of arrival and departure, and that affects the routes available at each node in a particular time.

The original development of IEA does not take these aspects into account, but thanks to its labeled structure, the incorporation of the time a route starts and when a node will be reached,

can be easily added to the data already stored in the label. This way, only the routes that depart from this particular node after the calculated time of arrival will be considered to extend labels to the corresponding nodes.

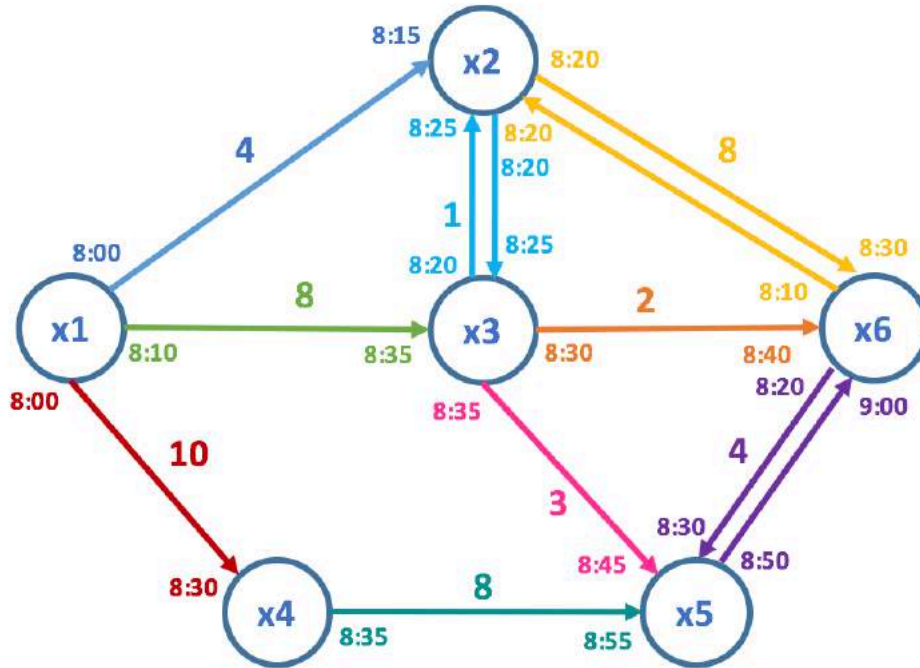


Figure 49. Simple time-dependent graph.

Figure 49 shows the graph that has been used to analyze every potential candidate as Dijkstra's substitute. The difference is that now every route includes a start and end time of departure and arrival, respectively. Given the additional time conditions implemented in IEA, the labels employed by the algorithm will have the appearance showed in Figure 50.

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],8:00,8:15,1,F}	F;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}		

Figure 50. First step of IEA with time dependency.

The structure of the label is shown in Figure 51:

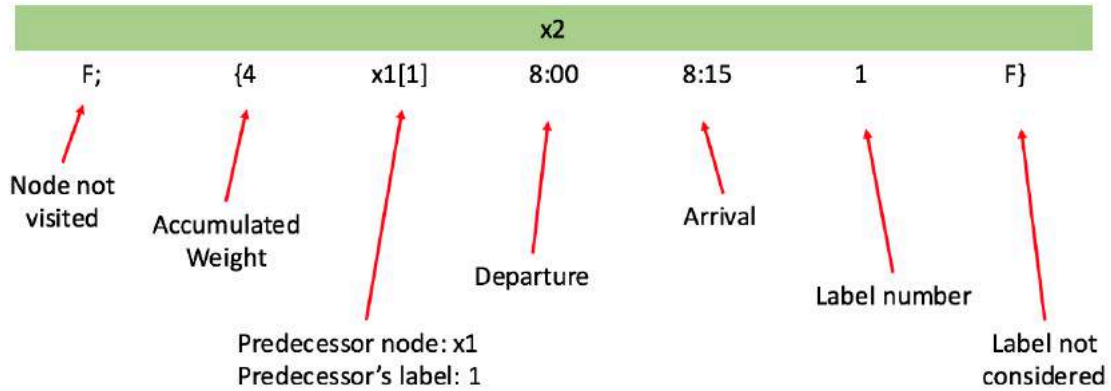


Figure 51. Label structure in IEA with time-dependency.

In the second iteration, x_2 is chosen as the next node as it has less weight (Figure 52). Labels are extended to x_3 and x_6 because the departure time from x_2 is later than arrival time at the same node. On step 3, however, the route from x_3 to x_6 is not considered as the departure time is earlier than the arrival to this same node. Only the label to node 5 is extended.

	x_1	x_2	x_3	x_4	x_5	x_6
1		F;{4,x1[1],8:00,8:15,1,F}	F;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}		
2		T;{4,x1[1],8:00,8:15,1,T}	F;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}		F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,F}			
3		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}	F;{8,x3[2],8:35,8:45,1,F}	F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,T}			

Figure 52. Second and third steps of IEA with time-dependency.

Figure 53 shows the number of steps of the first iteration. There are a few more than in the previous case, and also the path obtained is a different one and with a different cost from the

one extracted without time dependency. This is because some routes could not be considered due to the time they take place.

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],8:00,8:15,1,F}	F;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}		
2		T;{4,x1[1],8:00,8:15,1,T}	F;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}		F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,F}			
3		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}	F;{8,x3[2],8:35,8:45,1,F}	F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,T}			
4		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}	T;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,T}			{12,x5[1],8:50,9:00,2,F}
5		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,F}
6		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	T;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,F}

Figure 53. First iteration of IEA with time-dependency.

Also, with the information of the last label, it can be known the time of arrival to the destination, which is 8:30 in the morning. Appendix A shows the complete number of steps for the time-dependency version of the graph.

6.3.2 MODELING OF IEA ADJUSTED TO DATA STRUCTURE

When describing the modeling of the algorithm, it is necessary to consider the programming language used to implement it. In this case it has been PHP 7.2.2 [71].

The modeling of the algorithm consists on defining what are the inputs it will receive and the outputs to be obtained when executing it. Then, the internal structure of the implementation will be specified, explaining the classes and objects created and defined to organize the internal structure. Finally, the steps needed to put into action the whole performance of the algorithm will be identified and explained.

6.3.2.1 Inputs

The inputs the algorithm receives are the route attributes, the number of iterations, the start time and the origin and destination points.

As stated in section 6.3, the main **route attributes** to be taken into account are:

- Route Identification Number.
- Origin/Identification Number.
- Start/End Date.
- Start/End Time.
- Duration.
- Price.
- Vessel.
- Origin/Destination Country.
- Company.

The **start time** is the time at which the user wants to start looking for trips. What actually happens is that the user only enters into the search machine the day desired to be the first of the journey, not the time. Then, the first hour of departure is defined internally in the code and not by the user. Also, as explained before, routes starting this date and 2 days onwards are passed on to the algorithm.

The **origin** and **destination** are the two points entered by the user into the search machine.

6.3.2.2 Outputs

The output of the algorithm are all the **paths** that it has found. As it will be explained later, the number of paths will be limited by the execution time needed to calculate them. In case only few number of paths are available, the execution will finish when all the paths are found.

6.3.2.3 Internal Structure

PHP is a programming language that allows to apply Object Oriented techniques. Thanks to that fact, the internal structure of the algorithm when implemented in PHP relies on the definition and construction of Objects.

◆ **Edge**

For each route extracted from the query for combined results, an Object belonging to the class called Edge is created. The information included in all Objects created from this class include the following attributes.

Attributes

- Route id.
- Start and end points of the route.
- Weight or price assigned to that route.
- Additional data associated to the route.

Methods

No methods are included in this class except for the constructor.

◆ **Label**

A class representing the labels used by the algorithm has been also created. The information associated to the labels is the same specified in the examples in section 5.5.4.3.

	x1	x2	x3	x4	x5	x6
1		F; {4,x1[1],8:00,8:15,1,F}	F; {8,x1[1],8:10,8:35,1,F}	F; {10,x1[1],8:00,8:30,1,F}		
2		T; {4,x1[1],8:00,8:15,1,T}	F; {8,x1[1],8:10,8:35,1,F}	F; {10,x1[1],8:00,8:30,1,F}		F; {12,(x1[1],x2[1]),8:20,8:30,1,F}
			{5,(x1[1],x2[1]),8:20,8:25,2,F}			
3		T; {4,x1[1],8:00,8:15,1,T}	T; {8,x1[1],8:10,8:35,1,F}	F; {10,x1[1],8:00,8:30,1,F}	F; {8,(x1[1],x2[1],x3[2]),8:35,8:45,1,F}	F; {12,(x1[1],x2[1]),8:20,8:30,1,F}
			{5,(x1[1],x2[1]),8:20,8:25,2,T}			
4		T; {4,x1[1],8:00,8:15,1,T}	T; {8,x1[1],8:10,8:35,1,F}	F; {10,x1[1],8:00,8:30,1,F}	T; {8,(x1[1],x2[1],x3[2]),8:35,8:45,1,T}	F; {12,(x1[1],x2[1]),8:20,8:30,1,F}
			{5,(x1[1],x2[1]),8:20,8:25,2,T}			{12,(x1[1],x2[1],x3[2],x5[1]),8:50,9:00,2,F}
5		T; {4,x1[1],8:00,8:15,1,T}	T; {8,x1[1],8:10,8:35,1,F}	T; {10,x1[1],8:00,8:30,1,T}	T; {8,(x1[1],x2[1],x3[2]),8:35,8:45,1,T}	F; {12,(x1[1],x2[1]),8:20,8:30,1,F}
			{5,(x1[1],x2[1]),8:20,8:25,2,T}		{18,(x1[1],x4[1]),8:35,8:55,2,F}	{12,(x1[1],x2[1],x3[2],x5[1]),8:50,9:00,2,F}
6		T; {4,x1[1],8:00,8:15,1,T}	T; {8,x1[1],8:10,8:35,1,F}	T; {10,x1[1],8:00,8:30,1,T}	T; {8,(x1[1],x2[1],x3[2]),8:35,8:45,1,T}	T; {12,(x1[1],x2[1]),8:20,8:30,1,T}
			{5,(x1[1],x2[1]),8:20,8:25,2,T}		{18,(x1[1],x4[1]),8:35,8:55,2,F}	{12,(x1[1],x2[1],x3[2],x5[1]),8:50,9:00,2,F}

Figure 54. Structure of label when implementing IEA.

Attributes

- Accumulated weight when the label is used.

- The predecessor nodes by which that label has been created. These predecessors have a two-field array structure. The first field corresponds to the name of the predecessor and the second one to that predecessor's label used to obtain the following labels.

This differs from the label structure presented in the previous examples, which only included the previous predecessor. When implementing the algorithm, it was found that it was much more efficient to include an array of all the predecessor and the labels associated to each one of them. In the Figure 54, an example is shown using the first steps of the first iteration of time-dependent IEA.

In step 2, the second label of x3 does not only include x2[1] as a predecessor together with its label, but also x1[1] as it was the predecessor of x2[1]. This way, when a label from x6 is chosen in step 6, the path that has been obtained by that label can be automatically known thanks to this array of predecessors. In this case, the path is x1, x2 and x6. The presence of the label associated to each predecessor would seem useless now that the path is being stored in each label, as that was its main purpose. However, it is useful to know which label has been chosen if additional information such as arrival or departure time, wants to be known.

- The arrival and departure time stored in the label.
- All the information associated to the route.
- If the label has been considered or not.

Methods

This class has not methods associated but includes logic in its constructor function regarding the predecessor nodes.

Constructor

The information used to create a label, that is, the inputs the constructor method receives are the accumulated weight, the arrival and departure time and the route information. It also receives all the predecessors of the node for which this label is being created, the new predecessor that needs to be included in this list and the label index of this predecessor.

In first place, the predecessors input is assigned to the predecessors' attribute to be included into the label being created. Then the last predecessor to be found is appended to this array together with the label index associated to it. This can be seen in the example shown above.

There is an exception to this procedure and that is when the label to be created has as predecessor the initial node. This is because no information about predecessors can be included as the first node does not have any predecessors itself.

◆ **Node Holder**

A class has been created to represent the nodes that will include labels assigned to each one of them. The “*holder*” in the class’ name is due to the fact the Objects of this class are not the nodes themselves, but a recipient that includes the following information.

Attributes

- Identification number of the node.
- Name of the node. It is necessary include both id and name because, when the node is the origin or the destination introduced as inputs, the identification of the node is by name rather than by id.
- If the node has been visited or not.
- The set of labels associated to this node. This set is an array with Label Objects specified in the previous point.

Constructor

The constructor of this class receives as inputs just the node id and name. It is automatically set to unvisited and the set of labels is created but it is empty.

Methods

This class includes some methods.

- Set the node to visited.
- Set the node to unvisited.
- Add a label to the array of labels assigned to the node. For this process, first an Object of label is created by means of the constructor of the Label Class and then added to the array belonging to the node.

◆ Node Holder Finder

This class creates a single object when executing the algorithm. This object is the one in charge of storing the list of all the Node Holders that have been created and to update it each time a modification is made. It is the internal motor of the algorithm.

Attributes

- List of Node Holders participating in finding paths between origin and destination.
- The accumulated weight of the path being considered at the moment.

Constructor

This class has a simple constructor function that creates only the empty list of the Node Holders that will be later populated.

Methods

A relatively large part of the algorithm's logic is built in this class by means of methods.

- Add Node Holder. As the list created in the constructor is empty at the beginning, one of the methods to be called is the one that adds a Node Holder that has been created.
- Update the list. Every time a change is made in the Node Holders (a node is set to visited or unvisited, or new labels have been included to a node) the list needs to be updated with the latest versions of the Node Holders.
- Find by Id or by Name. As a Node Holder gets identified by either the id or the name or both, it is necessary to implement a function that finds a Node Holder when one of them is provided.
- Find if the node is in the predecessor's list assigned to a label. This function takes as inputs the predecessors' list of a certain label and the node id and name of the node to which a label needs to be extended. Then, it iterates through this list and determines if the node is inside or not.

Find the next node. One of the main tasks of the algorithm is to find the next node to be analyzed when new labels are added to the Node Holders, as it was specified in section 5.5.4.3. As the Node Holder Finder is the one that has all the information about the Node Holders and their labels, to incorporate the logic to this class and not outside is the sensible thing to do.

No inputs are needed when this function is called as all the information needed is already stored in the class. The list kept by the Node Holder Finder is going to be iterated and each one of the Node Holders is going to be analyzed.

- Reset for next path. As explained in the previous section, when a path has been found, some modifications need to be applied in the list that now holds the Node Holder Finder. These changes are to mark as unvisited the nodes belonging to the path that has just been found and to reset the accumulated weight to 0.

◆ IEA Class

For the algorithm overall operation, a class will be created with its same name. This class has the following information.

Attributes

- A list of all the nodes so the list of Node Holders can be created and stored inside the Node Holder Finder instance.
- A list of the edges emerging from each one of the nodes. The creating of this array will be explained in more detail further on.
- A Node Holder Finder instance.

Constructor

The constructor initializes the Node Holder Finder instance and also populates the nodes attribute with the nodes passed as an input.

Methods

The methods used by this algorithm are:

- Add an edge to a node. For each one of the routes passed into IEA, the class is going to receive as inputs the route id, the start and end point, the weight of the route and the route's complete information. Now, every time an Edge Object is created it is included into an array identified by the origin id of the route. This way, the key of an element in the array is the node id and the value is the array of all the routes that have as an origin the node specified in the key.
- Remove an edge. It will be identified by the start point of the edge and the identification of the route.
- Get IEA instance. This method allows to extract the array of all the edges identified by their origin as well as the Node Holder Instance.
- Updating the Node Holder Finder, as mentioned above.

- Create or search a node. This will depend on the node having been already created or not.
- Get paths. This method includes all the logic of the algorithm.

Figure 55 shows the relationships between these classes by means of an UML class diagram of IEA internal structure.

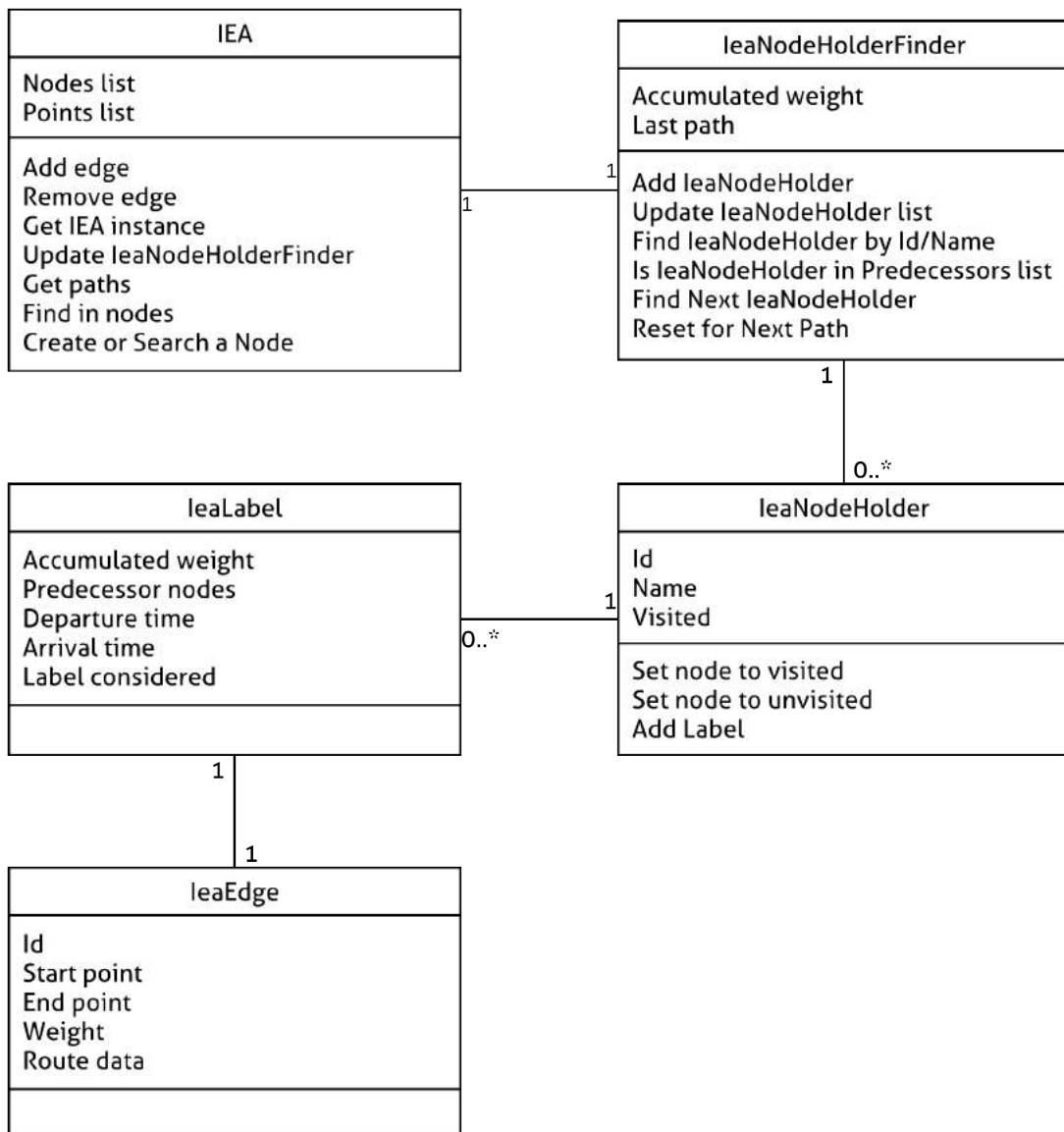


Figure 55. UML class diagram for IEA internal structure.

6.3.2.4 Operation

The basic operation of this algorithm is the same as explain in section 5.5.4.3. However, it was necessary to modify some of its features in order to embed it inside the internal code of the company.

In this section it will be specified how the algorithm was implemented when adjusted to the internal data and code structure as well as the internal modifications needed to adapt it to the time-dependency adopted by the company.

◆ Getting combined results function

This function outputs the combined results obtained by the algorithm, receives the inputs of the user's request, obtains the appropriate routes to insert to the algorithm, calls the algorithm and obtains, evaluates, sorts, stores and outputs its results.

Inputs

This function receives as an input the parameters of the search inserted by the user, which can be summarized as the origin, destination and the date of the routes that want to be found.

Queries

The function needs to prepare a query with the parameters received as input. This query is emitted in MySQL language, and has the main purpose of obtaining the appropriate routes from the database routes so the algorithm can work to obtain the results.

In first place, the query looks for all the routes that depart from the start point and those that reach the destination stated both by the user. Then it will add a condition to obtain the routes departing in the range of time between the date specified by the user and 2 days later. Also the routes that do not have a start date will be also required. These need to be considered because they can be adapted to the traveling time. For instance, taxi routes are not subjected to any schedule most of the times. Also, the expired routes will not be taken into account as

well as those routes that only operate during certain days of the week and do not match the traveling date specified by the user.

From all the columns belonging to the routes table, only those matching the attributes used by the algorithm and specified in section 5.5.4.3 will be requested, that is:

- Route Identification Number.
- Origin/Identification Number.
- Start/End Date.
- Start/End Time.
- Duration.
- Price.
- Vessel.
- Origin/Destination Country.
- Company.

Then another query is needed in order to obtain from all these routes that have been just obtained, the routes that connect origin and destination with just one jump in the middle. Figure 56 shows how routes are selected with these two queries.

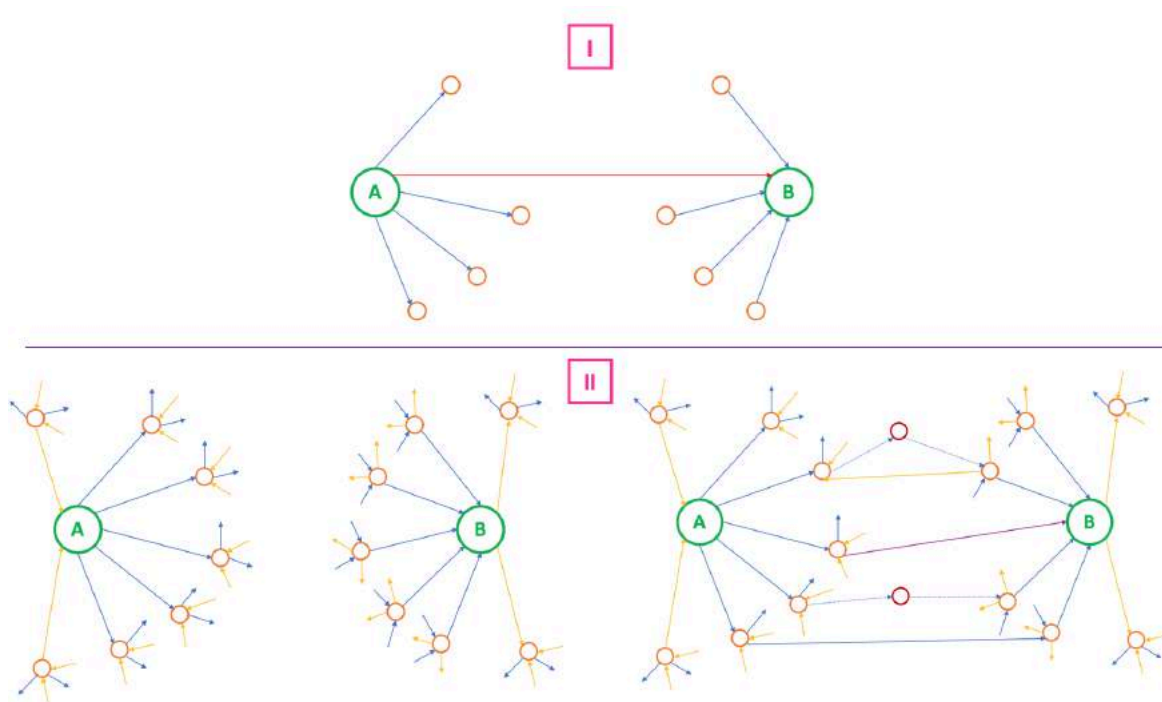


Figure 56. Query to obtain routes for combined results.

The first query obtains the routes that have as origin the start point specified by the user and the destination any node. The same happens with the routes that have as a destination the end point indicated by the user. Some of the routes may be direct, that is, their origin is the starting point and their destination is the end point at the same time.

For the second query, these routes have been omitted in the picture. In this case, more routes are being included in the results obtained from the routes database. These routes are those that have their origin or destination in any point depicted in the first part of the figure. What can happen is that routes that have their origin in a second-level node and their destination also in a second-level point will be found. But also routes that are connected through a second node. Also routes with their origin in a second-level node and their destination in the final destination will be obtained.

The problem of this query is that it provides more routes than the ones needed and which are quite useless, causing a considerable delay in the execution of the whole algorithm. The improvement of this query will be one the main objectives after the implementation of the algorithm.

◆ **Creation of Edges**

The next step is the creation of the Objects of class Edge included in the algorithm's class: IEA class.

Now that all the needed routes have been obtained from the database, only those routes that are not direct will be used to create the objects of Edge. The procedure is as follows.

In first place, the edges radiating from each node are created as explained in the previous point regarding IEA Class methods. The result will be an array whose keys are the node ids, and whose values are arrays with the routes that have their origin id in the node specified in each key.

◆ Getting the paths

Next, the IEA method that gets the paths is called and the result it returns will be all the paths the algorithm calculates. As stated before, the input of this method are the origin and destination specified by the user, the time of departure and the array of weights explained in section 6.2.

Initial step

First of all, it is necessary to set the node representing the origin to visited. It needs to be taken into account that if the user entered the name of a town, several nodes can be found with a name equal to the town name but with a different identification number. For example, if the town specified by the user was Ho Chi Minh, several nodes can be found inside the city, such as the railway station, the airport and various bus stations. All of them will own the name of Ho Chi Minh but their id will differ.

Therefore, if the user entered a town name in his search, it is necessary to find all the nodes inside this town and set them to visited. This is because only the routes whose destination is outside the town will be of interest, as several jumps between bus and train inside the same city is not desirable, when the final destination is another city.

Iterative steps

The next steps are repeated conforming the algorithm's iterative features analyzed in the previous section.

First steps

The first time the loop is executed, the node to be considered is the origin entered as an input to the function, the same the user chose. Then every edge parting from this node will be analyzed.

Now the arrival time of the user to the initial node is specified by the one included in the route or the one passed as an input. Next, this arrival time at the origin will be compared to the departure time of the route. This way it will be validated because it can happen that that specific trip has already been missed.

Finally, a label will be added to the end point of the first route considered and that departs from the origin, that is, to the first edge belonging to the origin point being analyzed. Now, for the few following iterations, the rest of the edges departing from the origin point will be studied and the same logic explained here will be executed.

Next steps

When all the edges starting from the origin have been already analyzed then the node with the label with the lowest weight needs to be chosen. For that, the method included in the Node Holder Finder instance will be applied.

When this node has been found then it will take the place the origin node had in the previous case. That is, all the edges departing from it will be analyzed as before. However, there are some aspects to be taken into account now that these edges are connections. One of them is that in each connection, a waiting time between each trip needs to be considered. This include the time that it takes to collect the luggage or exit the airport in case of planes. This waiting time has been estimated by Baolau or specified by the provider instead. The result of applying it will result in ruling out some routes departing too early from the same node.

Finally, labels will be extended to the nodes that are reached by these routes. Then the next node needs to be found. If this node is different from the destination, then all this process starts all over again. If it is the same, then a path can be extracted.

Obtaining a path

When the node returned by the Node Holder Finder's method of finding the next node is the same as the destination passed as an input to the algorithm class, that is, the same as the user specified, then a path can be obtained.

The path can be extracted using the predecessors' field in the label selected from the destination node just chosen as the next node. Also, as this predecessors' field is iterated in order to find the predecessors, the data (or route information) stored in each of the predecessors' labels is extracted and stocked in an array.

The information presented in this array includes:

- The node id through which the path goes through.
- The route information with origin in the node specified before.
- The departure time of the route.
- The arrival time of the route.

Step	Information
1st Step	Id: 364
	Origin id: 364
	Destination id: 370
	Data: Name: Nha Trang
	Start date, end date, start time, end time, price...
	Start time: 12:00
	End time: 15:00
2nd Step	Id: 370
	Origin id: 370
	Destination id: 395
	Data: Name: Da Nang
	Start date, end date, start time, end time, price...
	Start time: 16:00
	End time: 18:00
3rd Step	Da Nang

Table 18. Information returned for each path by IEA.

Reset

Once a path has been obtained, all the nodes belonging to that path will be marked as unvisited so their unconsidered labels can be evaluated in future iterations.

End of the iterations

The loop is left when the maximum number of iterations or maximum execution time (this will be decided in section 7.2.4) have been reached or when the value returned by the Node Holder Finder's method of finding the next node is null, which means that all possible labels have been considered.

6.3.3 OVERCOMING THE DRAWBACKS

As mentioned in section 5.5.4.3, one of the main disadvantages of using IEA with cycle avoidance is that the results are no longer extracted in order. That is, it may happen that the first results to be obtained are not the best ones and this is due to the modification applied to the original implementation of the algorithm to avoid loops in the solutions. When including the condition of forbidding the extension of labels to nodes that are included in the predecessors' list of a certain label, it may occur that the path that was being followed and that could be the next shortest one, is discarded for the moment and another with worse characteristics could be chosen before.

Some options to solve this problem were proposed:

- ◆ **Execute algorithm always with higher number of iterations than needed**

The algorithm is set to provide more results (more iterations) than the solutions to be displayed for the user. For example, the algorithm could work internally with a maximum number of 20 paths but show only 5 in the page.

This procedure can be carried out because the time that it takes the algorithm to obtain 20 paths does not defer much to that of obtaining just 5 or 10 for a low number of routes (see section 7.2.1). However, Baolau was choosing to show few results because of Dijkstra's implementation limitations of time and memory. If IEA algorithm offers better results with

20 solutions and they can be obtained in a considerable time span, then these 20 paths will be shown, as useful information would be being withheld with no reason at all otherwise. With this amount of results, the best one among all of them will have more probabilities of being displayed.

◆ **Continuous execution of the algorithm**

Another possibility would be to manage the execution of the algorithm at the same time the results are being shown. This way, a short execution of 3 or 4 results could be performed followed for more paths obtained afterwards. When these had been acquired, they could be added to the table and rearrange it dynamically so the newest additions that provide better results would be shown higher in the results table.

This comes with a serious problem and that is the friendly usage experience provided to the user. Even if it would be more efficient to show the user the best results at the beginning of the table, the dynamic arrangement could end up being an inconvenience. For instance, if the user had already located a good enough option in which he is interested, if this option was found to be worse in terms of duration than a newest one just added, then the path would be moved to an another location in the table, and there would not way to indicate the user to which place it has been moved.

Therefore, this option gets discarded.

◆ **Fixed execution time**

The maximum time the algorithm is going to be working can also be fixed. After the time has been spent then the results obtained will be output. This option may come in handy when considering the variability of the amount of routes that can be passed on to algorithm, which condition the execution time of the algorithm.

Also, if this time is fixed high enough to obtain a considerable amount of paths then there are possibilities of finding the best ones among them. This option will be further studied in section 7.2. The final selection among these possibilities will be presented in section 7.2.4.

6.4 NEW DATA STRUCTURE

A new data structure was decided by the company to be implanted when retrieving the routes from the database to calculate the combined paths.

First of all, due to the long time that the query to the original data structure took, a new data structure was decided to be implemented together with the update to a new algorithm for path-searching. As seen in section 6.3, the original query obtained routes differentiated by the date they took place. In the database, even old routes are stored but they are periodically deleted. However, the amount of information accessed by the query was fairly large causing the query to take too much time, more than the execution of the algorithm itself.

Additionally, the user was shown a list of routes that had already been consulted by other users. This has proven to be sometimes deceitful as the route was later found to be unavailable. The new data structure provides a solution to this problem.

Taking this into account, a new access to the route database was determined. Instead of accessing with the query directly to the routes table, a MySQL view has been defined and it includes all the routes: those already outdated and those programmed for the near future. The main difference is that no date is included in the view, just the schedule for this routes. The idea behind this is that it is probable that the routes are repeated over time and the previous departure and arrival times could match those of the future routes.

Also, the price stored for these routes is an average of the prices of those routes that have the same origin, destination and vessel and have been found in the routes table. Consequently, when launching a query to the database, only the origin and destination are going to be defined and so the results are going to include much more routes than with the last query. For instance, when searching for routes between Ho Chi Minh and Da Nang, all the routes that have been registered in the routes table and that are now stored in the view without date are going to be returned. With the former query the result was limited by the travel date chosen by the user and introduced in the query.

Thanks to this new view, routes can be shown instantly to the user with an approximate value of the real price. On the other hand, when the algorithm calculates the different paths between origin and destination, these routes are first checked with those in the route table in order to see if they actually are available or if scraping is going to be needed.

Changes introduced in IEA

One of the consequences of using the new view is that the routes cannot longer be identified by an id as it is possible that they do not actually exist in the routes table. Therefore, IEA needs to be modified in order to operate without identifying the routes by id.

Also, all operations regarding the date of the routes must be removed as the new routes do not have departure nor arrival date. However, it is necessary to take into account the trip time in order to see if the routes were available at a certain time of the day.

Afterwards, the routes that conformed the paths obtained by IEA need to be identified by their origin, destination and vessel employed so they can be somehow looked for in the routes table and check if they really exist. Those that have been found in the table will be shown directly to the user while those that have not will be set to scraping (see Figure 48). Therefore, all paths found by the algorithm and currently available will be displayed to the user one way or another. This provides more flexibility to the user as he can now see routes that previously existed and that have been updated with this last access to the routes table. With the former query, the routes presented were only the ones that the routes table provided for the future.

As a consequence, the amount of routes returned by the query is going to be much larger than with the former query. Therefore, it is necessary to prove IEA can work with so many routes. IEA was proven to be effective with the quantity of routes returned by the previous query, contrary to Dijkstra that constantly made the system collapse.

The next step would be to analyze IEA's performance with this new amount of queries obtained with the new query, and what are the advantages of the combination of the new data structure and the new algorithm in use. This will be studied in Chapter 7.

6.5 ADDITIONAL MODIFICATIONS

Once the complete functionality of the algorithm has been implemented, some modifications have been introduced into the algorithm. These adjustments are:

- Loading more results after some have been shown to the user.
- Allowing different users access the information already calculated before.
- Avoiding parallel access to common resources.

Fortunately, for the first two features only one change in the structure needed to be made. As for the third modification, semaphores have been needed in order to implement it.

Loading more results

As specified before, the execution of IEA can either be time-limited or result-bound, depending on the chosen approach. Either way, not all the results available are shown after the first execution. That is why a button has been added in the interface so the user has the possibility of asking for more results.

During the first iteration, IEA has calculated labels that have not been used yet but that will allow the calculation of future paths, that is, they would be helpful if IEA could continue its execution from the point it was left on the first time. Therefore, the solution has been to extract and store IEA label structure every time an iteration is finished whether because it has exceeded the selected time or the maximum number of paths to find. This IEA structure is identified by the origin and destination points as well as the travel date.

When the user wishes to load more results, as the origin, destination and travel date are the same, the IEA structure is retrieved and IEA can continue its execution where it was left before. This procedure reduces the time taken to obtain more results and avoids wasting resources repeating the operations that otherwise would need to be done again.

Multi-access to already calculated combined routes

As an IEA structure is stored in case the user wants to load more results, this structure could be also used by other users interested in the same itinerary and in the same date. However,

it is important to keep track of the information that has been shown to each user so there is no overlapping on the routes displayed to each one of them. For example, if a user is shown 5 combined results and another user a few seconds after looks for the same itinerary and date, then this user will be shown this same 5 results. Now if the second user wished to load more results another 10 results are shown, for instance. If the first user now presses the button for more paths, these 10 results need to be shown before more results are loaded, otherwise only the new paths generated by IEA will be directed to the user and not the previous ones.

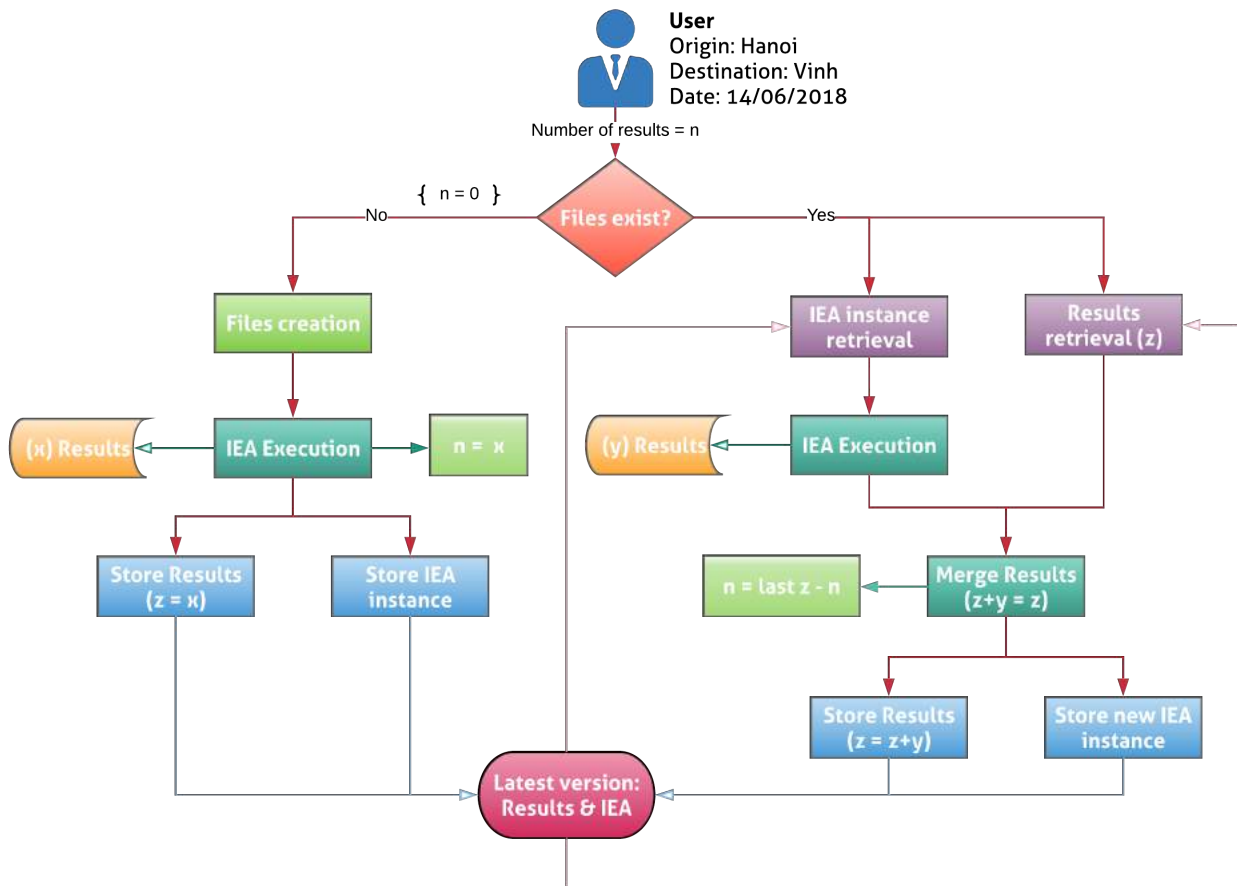


Figure 57. Multiple-access to combined results flowchart.

The solution to this dilemma consists on always storing in files the latest version of the IEA structure as well as all the results obtained from every user so far. But also, each user needs

a counter of the exact number of results he is being shown to avoid the problem mentioned earlier. Figure 57 explains this procedure for routes searched between Hanoi and Vinh on June 14th 2018.

Scenario 1: A user access for the first time the itinerary.

- The value of the results being shown to him is 0 ($n = 0$), and obviously no files will exist as no one has created them before.
- The files are created and IEA is executed in order to obtain x results. Let us say $x = 10$ results.
- These 10 results and the IEA instance are stored in the files.
- Also the result counter stored on the client side is updated to the new results being displayed ($n = 10$).

Scenario 2: The same user loads more results.

- Now n has a value of 10 and the results file as well as that containing an IEA instance exist, and they do so because the itinerary is the same as well as the date as the user just pressed the button to load more results.
- The results as well as the IEA instance are retrieved from the files.
- IEA is executed, and let us say that this time it provides 5 results ($y = 5$).
- These 5 results are merged to those obtained from the results file. Now the total number of results is 15.
- From these 15 results (it is essential they are not reordered once they merge) the user already has the first 10 ($n = 10$). Therefore, he will only need the last $15 - 10 = 5$ results, which match the new results obtained.
- Now n is updated to 15.

Scenario 3: A new user access the itinerary.

- For this user the files also exist and he retrieves the 15 results that have already been calculated by the other user.

- The procedure is the same as in Scenario 2 but now $n = 0$, so all these results will be passed back to the user.

This way it can be ensured all users are shown every result obtained so far without missing one.

Avoiding parallel access to common resources

The arrangement shown in Figure 57 allows different users to access the same resources, three files specifically: one for routes, another for results and then another one that stores the IEA instance to be used on future iterations. Each user will be performing reading and writing tasks. The fact that two users read simultaneously a file does not imply any inconsistencies. However, writing simultaneously does. Therefore, it is necessary to employ some mechanisms, such as semaphores, that prevent two users from accessing the files at the same time.

A semaphore simply locks a resource once it has been accessed by one user. If another user tries to access that same resource, the semaphore detects it is already locked and, in this case, sends the second user to sleep for a certain time after which he can try again. If the resources are still being used, then the same process is repeated. If not, the user is free now to access them now that the other one has already released them.

6.6 INTERFACE MODIFICATIONS

One of the additional goals of this project was to improve the route-searching interface appearance. Therefore, some filters have been added in order to provide the user more possibilities when looking for a certain trip.

There were some filters already implemented regarding means of transportation. They can be used to display results depending on the vehicle used in each one of them. What has been added is the possibility to filter some trips depending on the following features:

- Departure time lower limit: what is the earliest hour the user wishes to start his trip.

- Departure time upper limit: when is the latest hour the user wishes to depart.
- Maximum overall trip duration.
- Maximum number of steps: how many vehicle transfers the user is willing to take.
- Companies providing the different routes.

What all these filters have in common is that they are applied on the already calculated results, that is, the ones that are being displayed to the user. Only when the user has entered the origin, destination and date of his trip and the available results have been shown, will the filters be displayed. This implies that the IEA implementation has not been modified in order to design these filters.

The filters have been implemented in the JavaScript file that allows to make changes on the client side rather than the server, rendering this process much more agile and versatile.

The image shows the Baolau website's search interface. It features a dark-themed header with the Baolau logo and navigation links. The main content area has a cityscape background and a large text overlay. Below the header, there are several input fields for search criteria: origin (DE), destination (A), trip type (VIAJE), departure date (IDA), and return date (VUELTA). The number of passengers (PASAJEROS) is set to 1. A green search button labeled 'BUSCAR' is positioned to the right of the date fields.

Figure 58. Route-search form in baolau.com.

Figure 58 shows the route-search form in the initial page of baolau.com. As mentioned before, no filters can be applied before the results are shown. The transport filters can be seen in Figure 60.

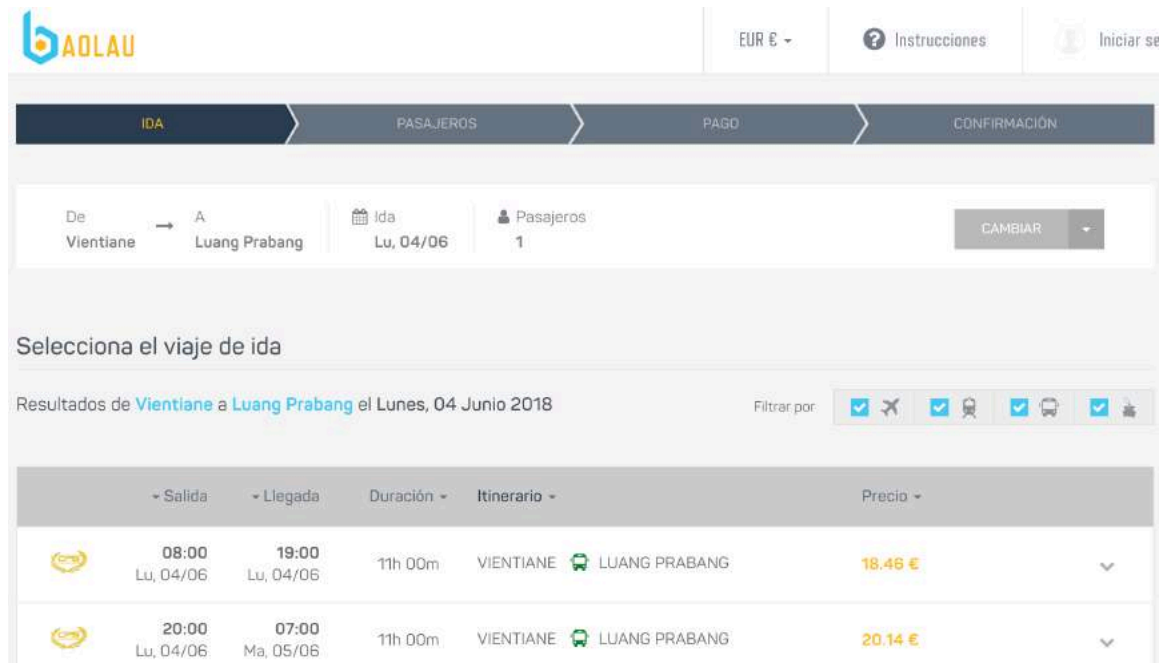


Figure 59. Transport filters.

Figure 60 shows on the top the initial appearance of the newly added filters, and which will be later modified before the company uploads the code to production. The bottom image depicts the interface after the user moves the sliders and clicks the button.

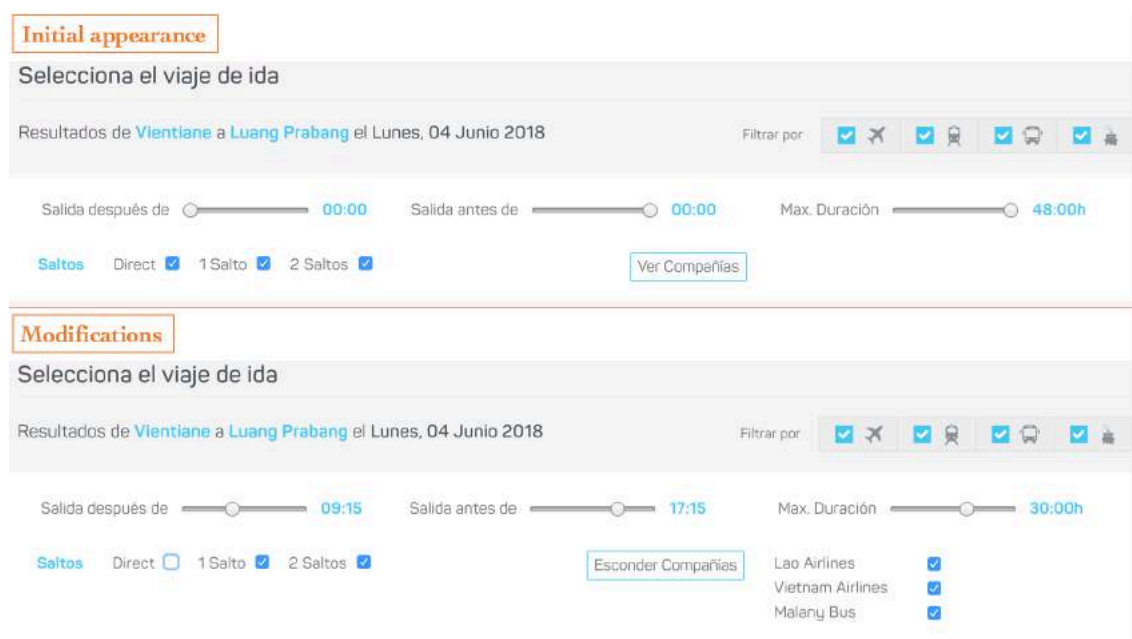


Figure 60. Search results filters.

One of the characteristics of these filters is that they update the results in real time and there is no button that needs to be clicked to show the filtered results. Figure 61 depicts all the unfiltered results between Vientiane and Luang – Prabang.

Selecciona el viaje de ida

Resultados de **Vientiane a Luang Prabang** el Lunes, 04 Junio 2018 Filtrar por ✈ 🚗 🚚 🚚 🚚

Salida después de Salida antes de Max. Duración

Saltos Direct 1 Salto 2 Saltos

Esconder Compañías

- Lao Airlines
- Vietnam Airlines
- Bangkok Airways
- Thai Airways
- Malang Bus
- King of Bus

	- Salida	- Llegada	Duración -	Itinerario -	Precio -
	08:00 Lu, 04/06	19:00 Lu, 04/06	11h 00m	VIENTIANE 🚗 LUANG PRABANG	18,45 €
	20:00 Lu, 04/06	07:00 Ma, 05/06	11h 00m	VIENTIANE 🚗 LUANG PRABANG	20,12 €
	11:30 Lu, 04/06	12:15 Lu, 04/06	0h 45m	VIENTIANE ✖ LUANG PRABANG	141,72 €
	13:00 Lu, 04/06	13:45 Lu, 04/06	0h 45m	VIENTIANE ✖ LUANG PRABANG	141,72 €
	17:00 Lu, 04/06	17:45 Lu, 04/06	0h 45m	VIENTIANE ✖ LUANG PRABANG	141,72 €
	09:10 Lu, 04/06	10:00 Lu, 04/06	0h 50m	VIENTIANE ✖ LUANG PRABANG	141,72 €
	14:40 Lu, 04/06	20:30 Lu, 04/06	5h 50m	VIENTIANE ✖ HANOI ✖ LUANG PRABANG	243,72 €
	14:40 Lu, 04/06	20:30 Lu, 04/06	5h 50m	VIENTIANE ✖ HANOI ✖ LUANG PRABANG	260,80 €
	14:40 Lu, 04/06	19:50 Lu, 04/06	5h 10m	VIENTIANE ✖ HANOI ✖ LUANG PRABANG	266,40 €
	19:35 Lu, 04/06	12:15 Ma, 05/06	16h 40m	VIENTIANE ✖ BANGKOK ✖ LUANG PRABANG	212,17 €
	20:00 Lu, 04/06	19:50 Ma, 05/06	23h 50m	VIENTIANE ✖ HANOI ✖ LUANG PRABANG	297,59 €

LOAD MORE RESULTS

Figure 61. Unfiltered results Vientiane – Luang Prabang.

Figure 62 shows how the results change after some filters have been modified.

IMPLEMENTATION OF IEA ALGORITHM

Selecciona el viaje de ida

Resultados de **Vientiane a Luang Prabang** el Lunes, 04 Junio 2018 Filtrar por ✕ ✕ ✕ ✕ ✕

Salida después de: Salida antes de: Max. Duración:

Salto Direct 1 Salto 2 Salto

Vietnam Airlines
Lao Airlines

	- Salida	- Llegada	Duración	Itinerario	Precio
	11:30 Lu, 04/06	12:15 Lu, 04/06	0h 45m	VIENTIANE ✕ LUANG PRABANG	141.35 €
	13:00 Lu, 04/06	13:45 Lu, 04/06	0h 45m	VIENTIANE ✕ LUANG PRABANG	141.35 €
	17:00 Lu, 04/06	17:45 Lu, 04/06	0h 45m	VIENTIANE ✕ LUANG PRABANG	141.35 €
	14:40 Lu, 04/06	20:30 Lu, 04/06	5h 50m	VIENTIANE ✕ HANOI ✕ LUANG PRABANG	243.72 €
	14:40 Lu, 04/06	20:30 Lu, 04/06	5h 50m	VIENTIANE ✕ HANOI ✕ LUANG PRABANG	260.80 €
	14:40 Lu, 04/06	19:50 Lu, 04/06	5h 10m	VIENTIANE ✕ HANOI ✕ LUANG PRABANG	266.40 €

Figure 62. Filtered results Vientiane – Luang Prabang.

Another feature has been added in case the filtered results are non-existent. For example, in case the user inputs a maximum trip duration of 0h, no results will be shown. In this case, the load-more button will be hidden (Figure 63).

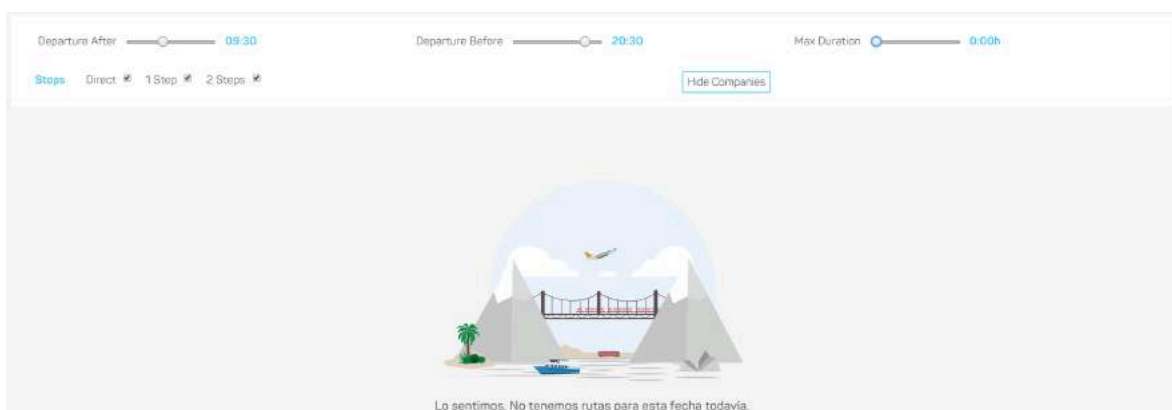


Figure 63. Load-more button hidden when there are no results.

Chapter 7. RESULTS ANALYSIS

This chapter will be focused on analyzing the results obtained with the implementation of the Iterative Enumeration Algorithm (IEA). Its performance will be compared to that of Dijkstra's algorithm as it was developed in Baolau's code. Afterwards a thorough analysis will be carried out regarding only IEA's execution.

The performance comparison will be carried out by contrasting the time execution needed by both Dijkstra and IEA to obtain the required paths between two nodes, and also the amount of paths obtained. This comparison will be done between the original implementation of Dijkstra's algorithm and IEA's implementations using the new access to the database. Afterwards the conclusions acquired with the tests and comparisons will be presented and the arguments determining the best algorithm performance will be explained. Consequently, the main objective of this section is comparing, not only the performance of both algorithms, but also the efficiency of the new access to the database to obtain the routes the algorithms employ.

Finally, with IEA's individual operation analysis results, the best configuration when running the algorithm will be specified in order to obtain the best possible performance.

The execution of all tests has been conducted in in a MacBook Pro computer with a 2,7 GHz Intel Core i5 microprocessor. It uses 8 GB 1867 MHz DDR3 of memory.

7.1 COMPARISON BETWEEN DIJKSTRA AND IEA

Two scenarios will be presented in order to test the effectiveness of the algorithm itself (Scenario I) and the benefits of using the new query to access the route database.

- **Scenario I:** Execution of both algorithms using each one a different set of routes whose number is similar to one another.

- **Scenario II:** Execution of both algorithms using the same set of routes between the cities of Hue and Vinh in Vietnam.

7.1.1 SCENARIO I: EXECUTION WITH THE SAME NUMBER OF ROUTES

The main goal of this scenario is to analyze the efficiency of IEA compared to that of Dijkstra under similar circumstances. For that reason, each algorithm will be executed using a similar amount of routes, as the factor that determines the speed of the algorithms is the quantity of routes introduced.

In order to find the same amount of routes for each one of the algorithms, it is necessary to apply different origin and destination points for each algorithm. The reason is because, as explained in section 6.4, the new query structure included by IEA's implementation always returns more results than the previous query to the database.

Dijkstra's execution

As explained before, the algorithm currently use in Baolau is Dijkstra using Yen's implementation of removing edges when more results want to be obtained. The query chosen for Dijkstra's execution is between the Vietnamese cities of Hanoi and Dong Hoi, resulting in 3100 routes.

Iterations	Avg. Time Query	Avg. Time Dijkstra	Avg. Total Time	Num. Paths	Num. Executions
1	6,554	0,131	6,726	1	1
3	10,696	1,409	12,449	3	6
6	7,014	1,406	8,464	6	9
10	7,061	2,075	9,185	10	13
15	7,105	3,015	10,167	15	18
20	7,004	3,918	10,989	20	23
30	7,560	5,692	13,328	30	33
40	7,392	7,183	14,624	40	43
50	7,369	9,189	16,610	50	53
60	7,072	10,179	17,297	60	63

Table 19. Scenario I – Recorded data for Dijkstra execution.

Table 1 shows the average time of the execution of the algorithm for a certain number of iterations with 10 executions every time (to obtain the average value). In this case, when applying 10 iterations the algorithm is called 10 times; with 20 iterations, 20 times and so on. This average time increases with the number of results to be obtained. However, it was found that three more executions were needed in order to reach the specified number of results, as it can be seen in columns *Num. Paths* and *Num. Executions*, which refer to number of paths found and the number of executions needed to acquire them, respectively. For instance, 6 executions were needed to obtain 3 paths. Nevertheless, the positive aspect is that this span seems to remain constant. The execution time is linear as Figure 64 shows, and this is because the algorithm provides a new path every time that it is called and it takes practically the same time in each execution. So the accumulated time follows that pattern.

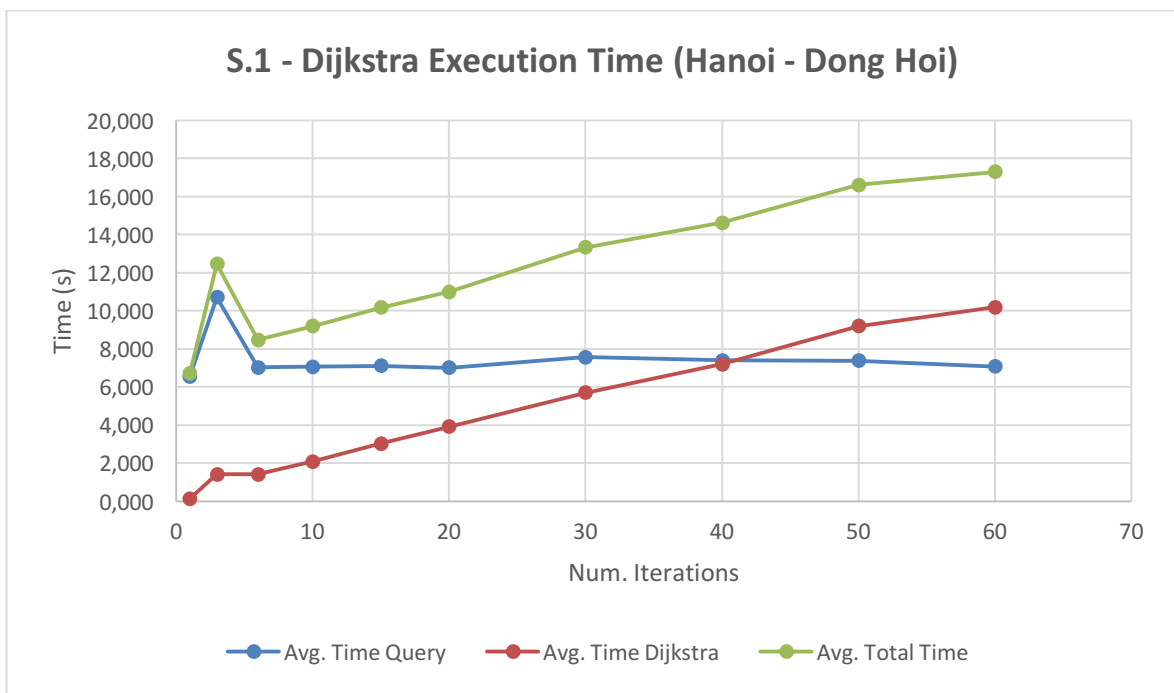


Figure 64. Scenario I – Dijkstra average execution time.

As Figure 64 shows, when the number of iterations is low enough, the actual time for launching the query is higher than that of the execution of the algorithm. This is highly inefficient as the algorithm itself is quite fast. However, as the number of paths to be found increments, the execution time of the algorithm surpasses that of the query, which should be

the usual behavior, as MySQL queries are normally found to be quite quick. The peak found in one of the first queries was probably caused by some glitch in the process and nothing too significant.

Figure 65 shows that the additional three executions needed to obtain the required number of paths remains constant over the number of iterations.

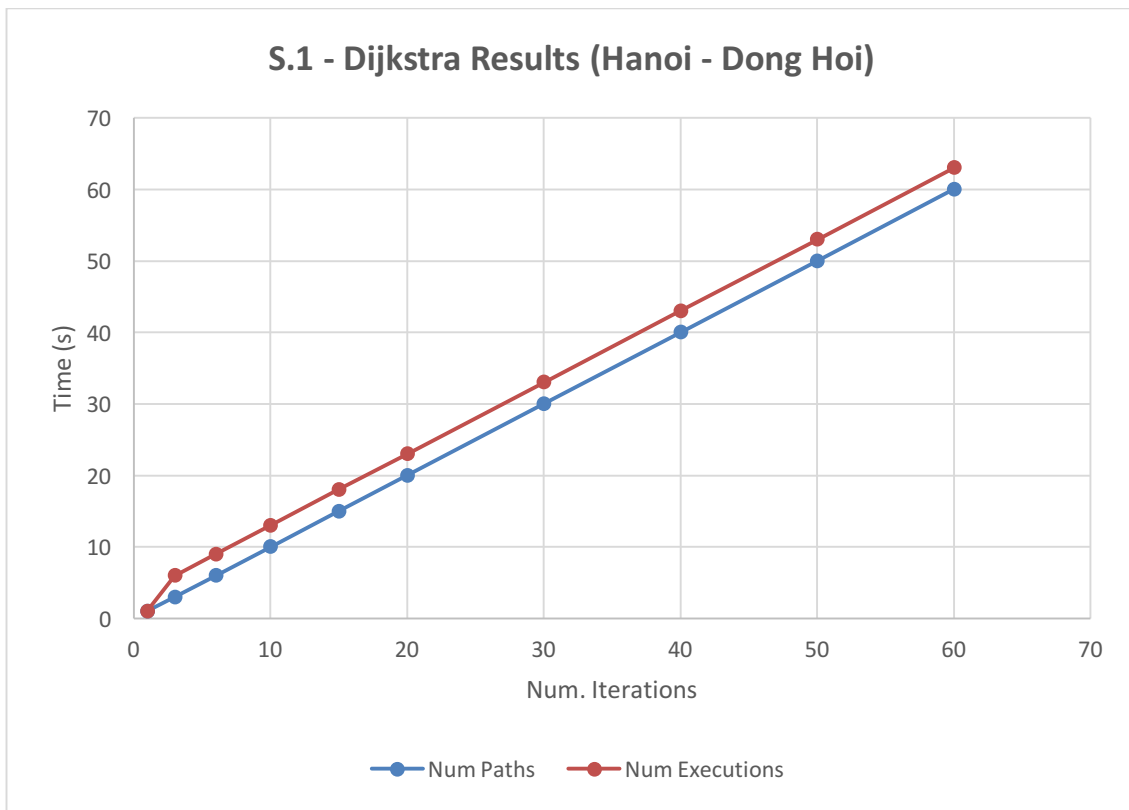


Figure 65. Scenario I – Dijkstra number of executions vs. number of paths.

IEA's execution

In order to operate with an amount around 3000 routes, IEA's performance was tested by introducing the routes between Nha Trang and Quy Nhon, two Vietnamese cities. Supposing the routes' complexity was similar to that of the routes applied to Dijkstra, the results obtained are presented below.

Iterations	Avg. Time Query	Avg. Time Algorithm	Avg. Total Time	Num. Paths	Num. Executions
1	0,187	0,722	0,994	1	1
3	0,156	0,702	0,997	3	3
6	0,136	0,798	1,010	6	6
10	0,127	0,808	1,010	10	10
15	0,129	0,994	1,299	15	15
20	0,123	1,096	1,390	20	20
30	0,124	1,551	1,929	30	30
40	0,127	1,543	1,948	40	40
50	0,122	2,495	2,943	50	50

Table 20. Scenario I – Recorded data for IEA.

Similar to Dijkstra, the average time for the execution of the algorithm is also quite low, thanks to the reasonable amount of routes introduced, as Table 20 shows. On the other hand, the algorithm returns the exact amount of results it was supposed to provide, contrary to Dijkstra, which sometimes provided the same paths and the execution needed to be redone in order to find a different result.

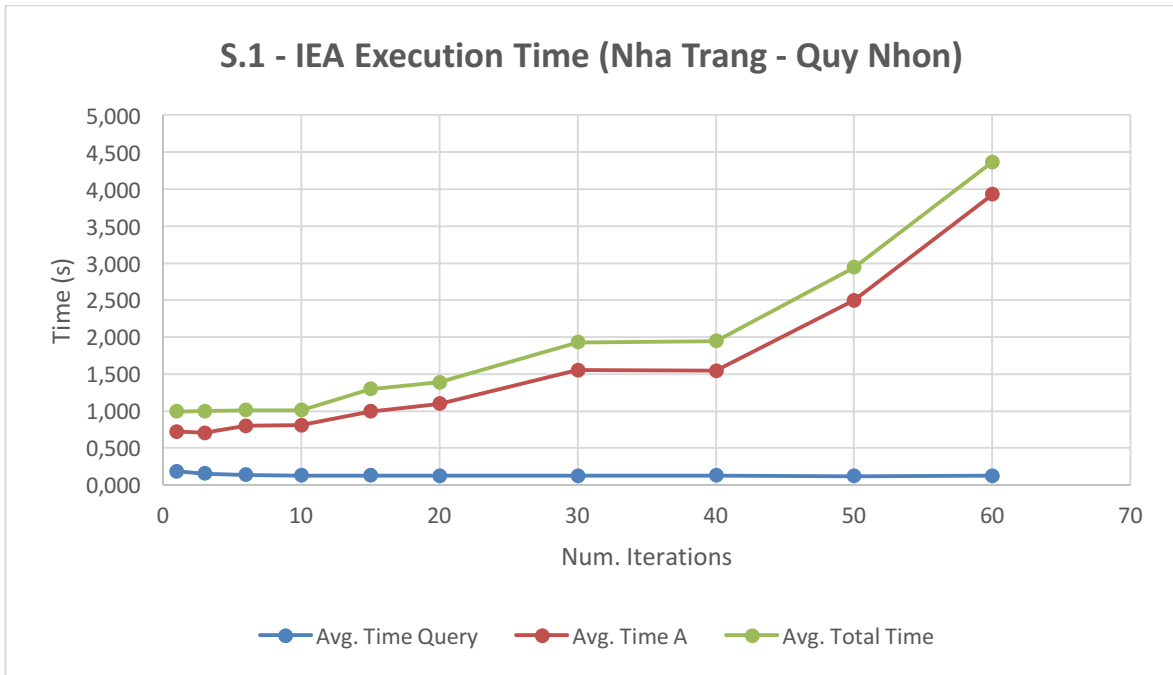


Figure 66. Scenario I -IEA average execution time.

However, the most interesting aspect is the amount of time needed to launch the query and receive its results, which is almost 12 times lower than the former query used with Dijkstra. The information provided in the table is represented in Figure 66

Compared to Dijkstra, the time required to access the database to obtain the routes is inessential, which is a very important advantage of the new query structure. More routes can be obtained with a fast query, promoting the effectiveness not only of the whole route-searching process but also of the algorithm's path calculation procedure. This way, the total time for obtaining the routes and calculating the different routes depends mostly on the algorithm's efficiency rather than in procuring the routes with which the algorithm can work.

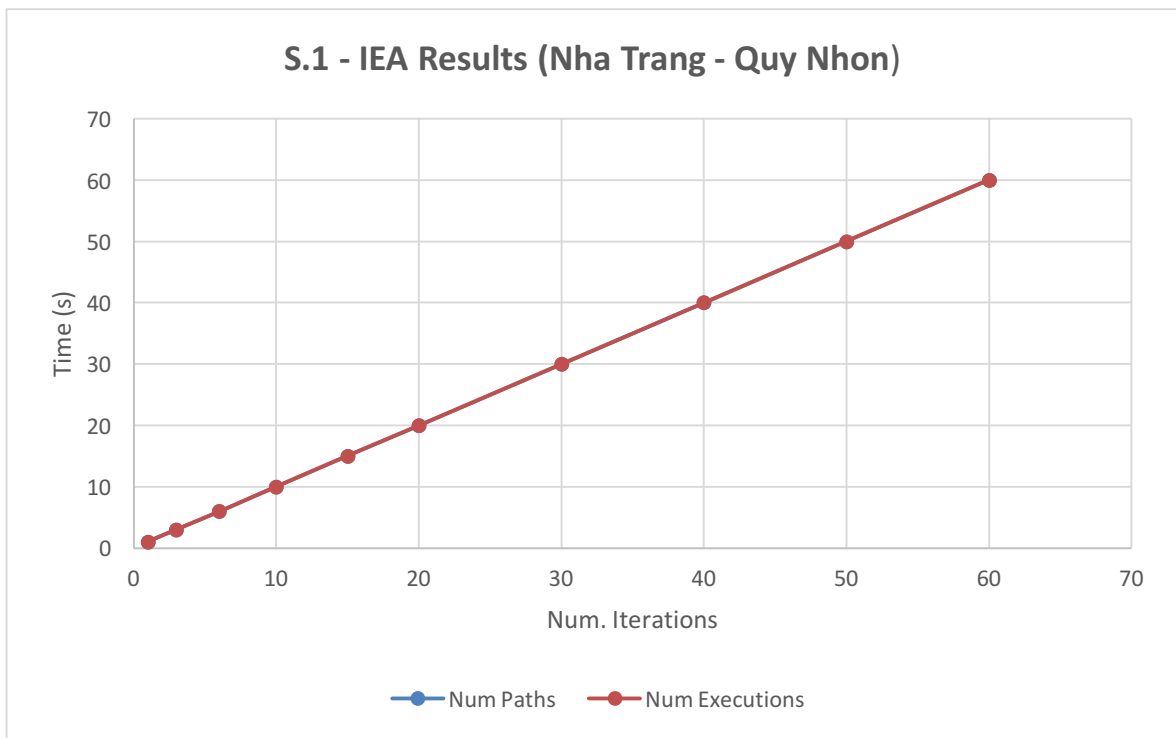


Figure 67. Scenario I – IEA number of executions vs. number of paths.

Also, as mentioned before, IEA always provides the exact number of paths that were required (Figure 67), as its internal structure ensures none of them are repeated. This is possible because IEA is executed only once while Dijkstra was called as many times as the number of paths required, so the algorithm itself could not keep track of the paths already calculated.

Figure 68 depicts the comparison between the time spent by both queries and the algorithms' execution times.

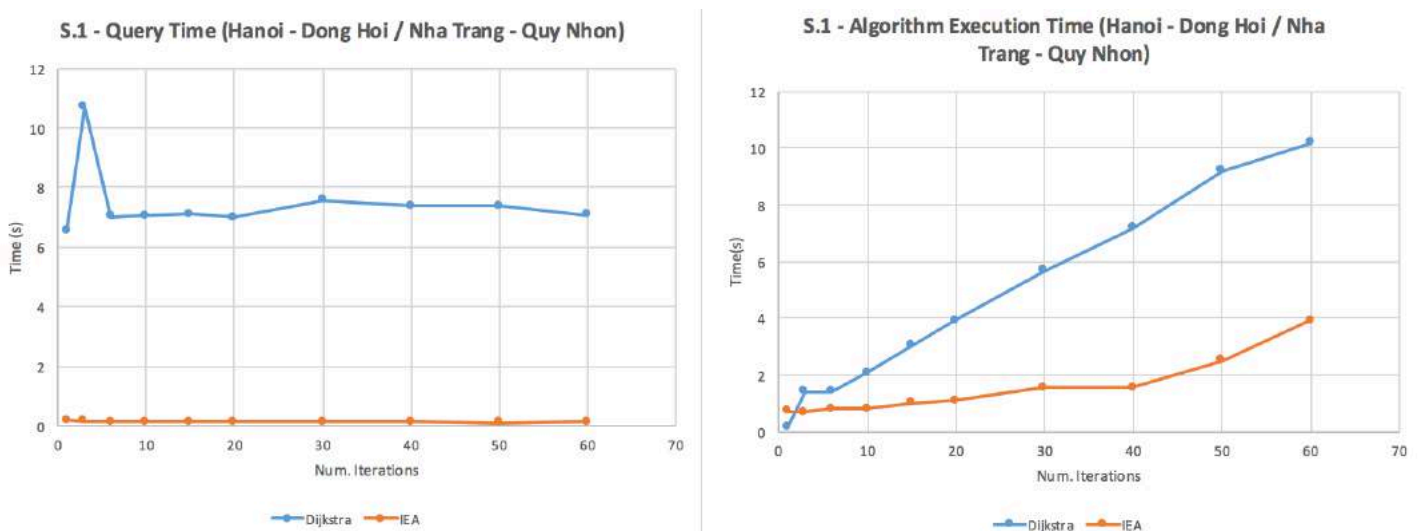


Figure 68. Scenario I – Performance time comparison between Dijkstra and IEA.

The right side of the image shows the comparison between the time needed for launching the query and receiving its output. It can be seen that the new query is six times lower as that employed in Dijkstra.

On the other hand, the right side of the image provides the different execution times of both algorithms. While Dijkstra manages to be a little bit faster when obtaining a single path, IEA takes much less time for the rest of the number of iterations needed.

This added to low time needed for obtaining the routes results in a total execution time much lower than Dijkstra, as Figure 69 shows.

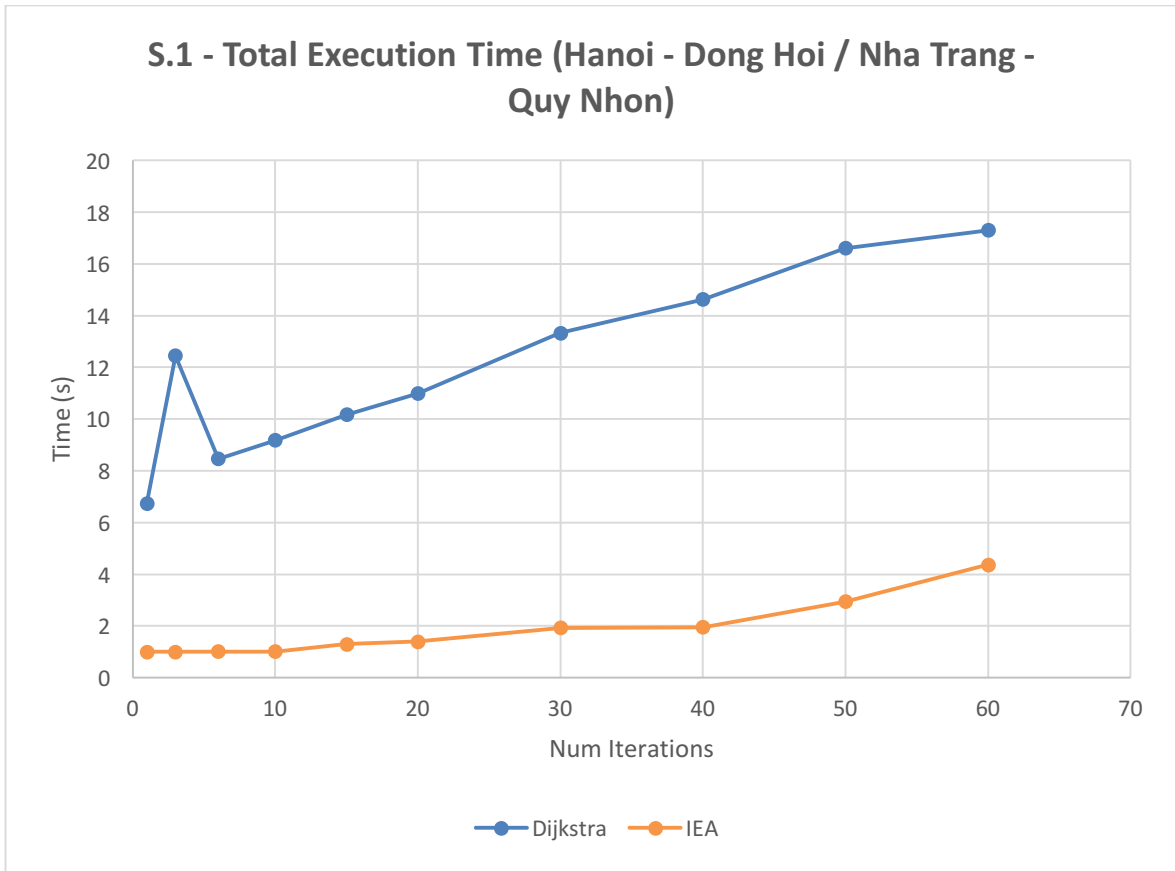


Figure 69. Scenario I – Average total time of execution comparison between Dijkstra and IEA.

7.1.2 SCENARIO II: EXECUTION ON THE SAME SET OF ROUTES

Both Dijkstra and IEA will be tested when calculating the paths between Hue and Vinh, two Vietnamese cities. The main difference is that Dijkstra is going to operate with 2000 routes approximately while IEA will function with about 5000. This is because the new query defined for IEA has proven to be more complete as well as more direct. Therefore, it needs to be taken into account that, theoretically, IEA will operate at a disadvantage, as the execution of both algorithms depends on the number of routes entered into the system.

Dijkstra's execution

The algorithm is quite fast when calculating each one of the results, as Table 21 shows on the column *Avg. Time Algorithm*.

However, the problem comes with the time spent on launching the query which eventually saturates the system causing its collapse.

Iterations	Avg. Time Query	Avg. Time Algorithm	Avg. Total Time	Num. Paths	Num. Executions
1	12,159	0,081	12,292	1	1
3	12,444	0,385	12,857	3	5
6	12,278	0,611	12,921	6	8
10	12,268	0,905	13,203	10	12
15	11,827	1,211	13,069	15	17
20	12,311	1,687	14,030	20	22
30	12,438	2,383	14,852	30	32
40	12,606	2,922	15,573	40	42
50	12,169	3,857	16,076	50	53

Table 21.Scenario II – Recorded data for Dijkstra execution.

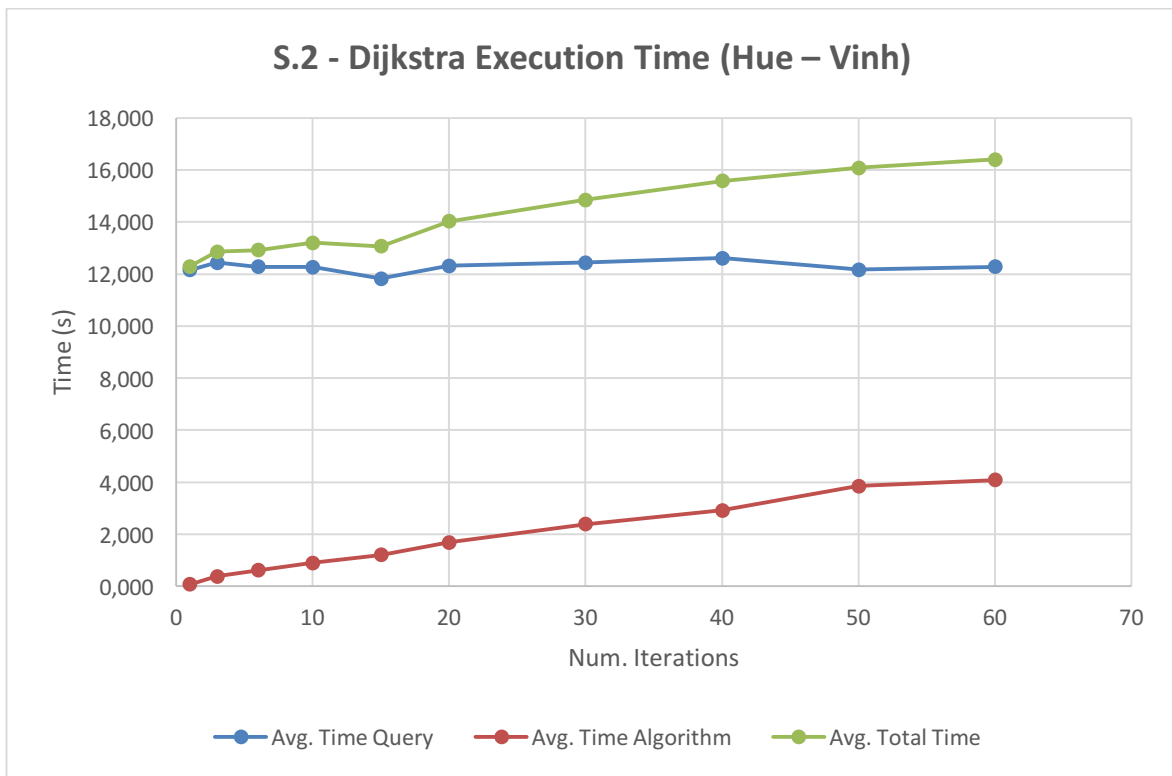


Figure 70. Scenario II -Dijkstra average execution time.

As Figure 70 shows, the time the algorithm takes to obtain all the paths is actually very short. However, the query to the database takes up too much time, rendering the algorithm's efficiency almost useless.



Figure 71. Scenario II – Dijkstra number of executions vs. number of paths.

As mentioned before, another drawback of Dijkstra implementation is that the extraction of one of the edges each time does not avoid obtaining the same path twice. In this case, in order to obtain the required number of paths, the algorithm needs to be executed once again. However, the good thing is that this difference does not increase with the number of results to be obtained (Figure 71).

IEA execution

For this same scenario, where routes between Ho Chi Minh and Da Nang must be found, IEA shows a quite different performance from Dijkstra. Again, the algorithm was

programmed to obtain a different set of results every time. And it was executed 10 times for each set in order to procure accurate results of the average performance time. Table 22 shows the information about the algorithm's operation.

Iterations	Avg. Time Query	Avg. Time Algorithm	Avg. Total Time	Num. Paths	Num. Executions
1	0,207	1,710	2,027	1	1
3	0,165	1,846	2,129	3	3
6	0,234	2,278	2,658	6	6
10	0,203	3,247	3,712	10	10
15	0,180	3,034	3,416	15	15
20	0,154	3,249	3,601	20	20
30	0,163	4,370	4,787	30	30
40	0,153	5,018	5,478	40	40
50	0,159	5,983	6,480	50	50

Table 22. Scenario II – Recorded data for IEA.

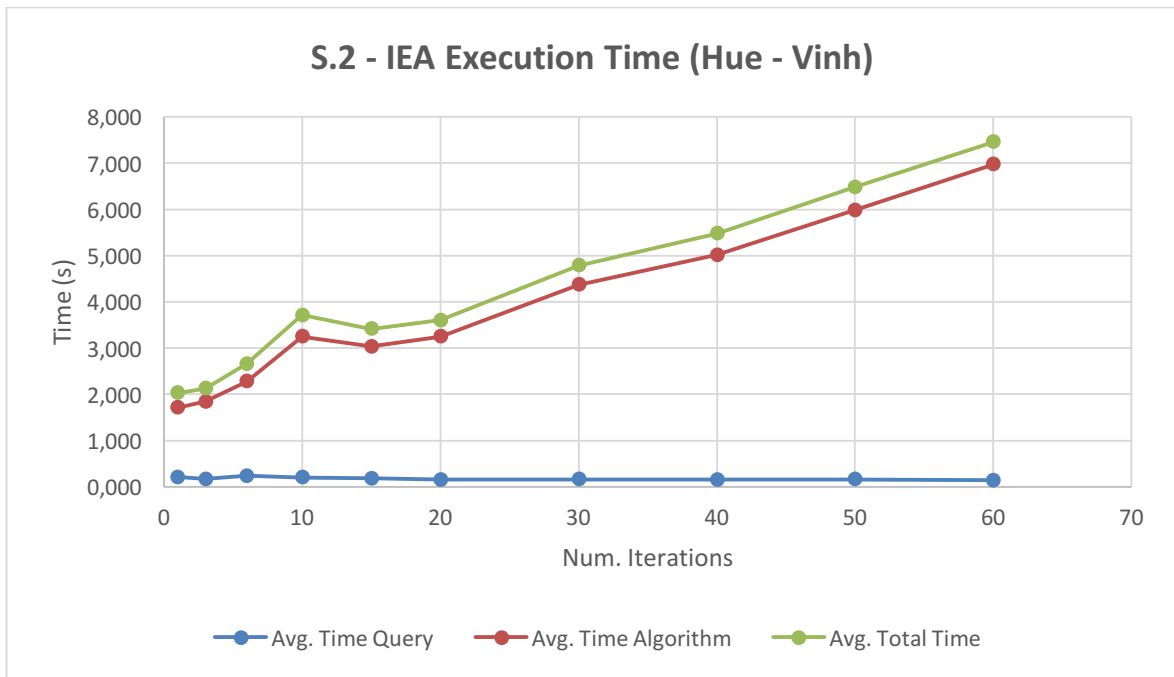


Figure 72. Scenario II – IEA average execution time.

IEA's execution time is higher than Dijkstra's. However, it must be taken into account that IEA has operated with more routes than Dijkstra. It can be seen that the execution time increases rather linearly with the amount of iterations performed. This is the logical output, similar to that of Dijkstra.

On the other hand, the average time for launching the query and receiving a response is substantially much lower than the previous data structure used with Dijkstra. This is what is going to allow the algorithm become much more efficient than Dijkstra as it is going to be able to operate with more routes. Also, more results will be obtained in a shorter period of time, even though the execution of the algorithm itself is a little bit slower than Dijkstra (Figure 72).

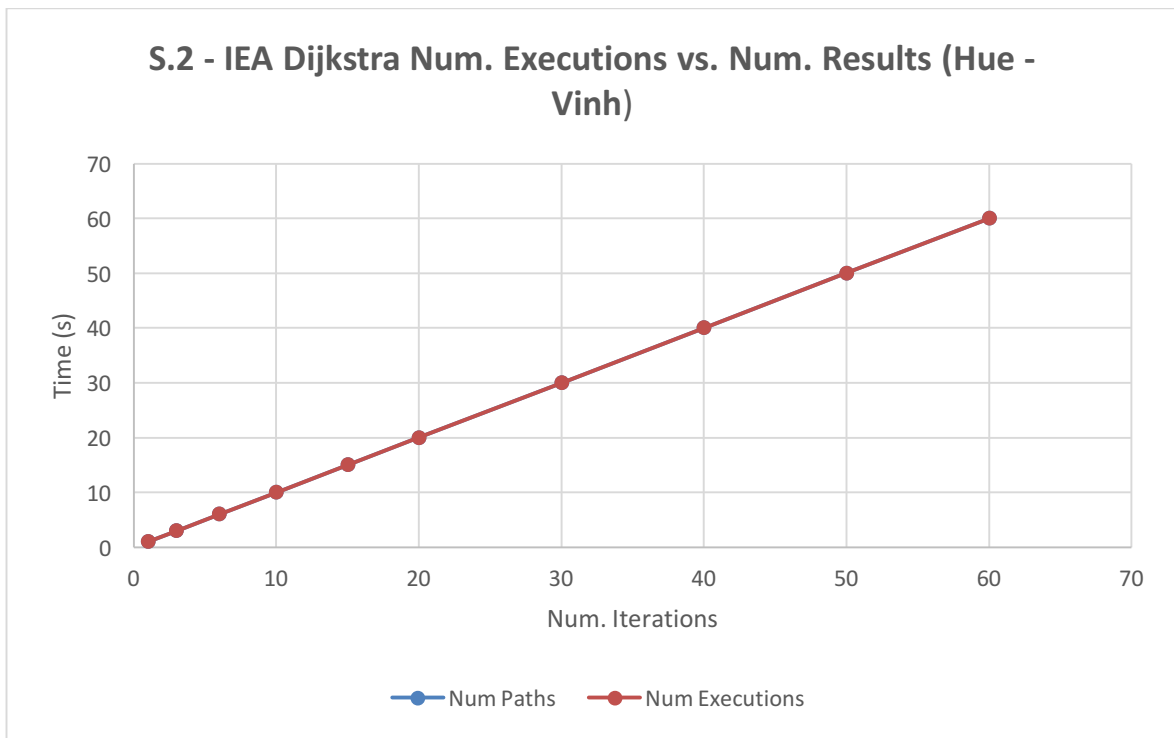


Figure 73. Scenario II – IEA number of executions vs. number of paths.

Regarding the number of results, IEA displays a much better performance than Dijkstra as it provides the exact number of results that it was programmed to obtain (Figure 73). This is a great advantage because, not only the company is going to be able to supply as many

results as it wants if they exist, but also the complexity of acquiring that many results can be known and therefore managed in the best possible way. That is, the company has knowledge about what are the performance consequences of asking the algorithm for more results. So they will be able to decide which is the optimal number of paths to extract every time, in case this is the approach chosen to be followed (see section 7.2.4).

Figure 74 shows the performance time comparison between Dijkstra and IEA for this scenario.

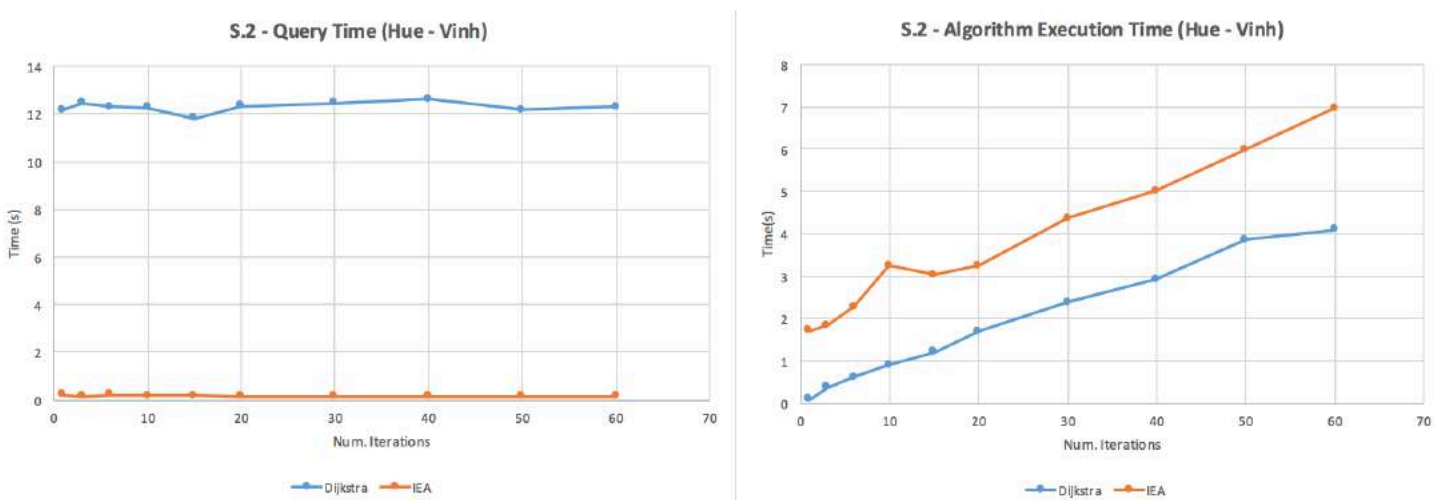


Figure 74. Scenario II – Performance time comparison between Dijkstra and IEA.

As the figure on the left shows, the previous query took up almost 12 times more time than needed with the new data structure. So even if, as the figure on the right shows, IEA's execution time is higher than that of Dijkstra's in this case, the average total time when applying IEA is lower, as Figure 75 shows.

Again, it is necessary to keep in mind that IEA is working with more routes than Dijkstra in this scenario, which results in an increase of its execution time.

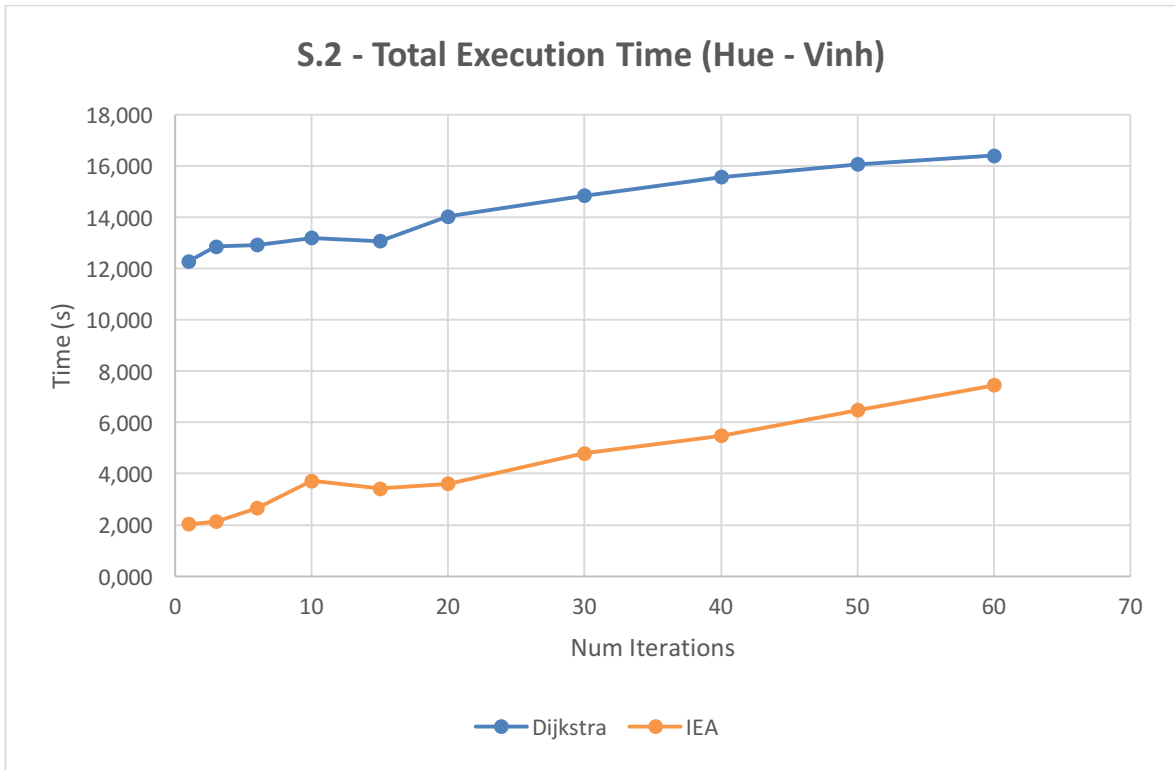


Figure 75. Scenario II – Average total time of execution between Dijkstra and IEA.

Therefore, it can be concluded that in this scenario where the previous query returned less results than the actual query, the average total time of execution is lower for IEA than for Dijkstra. The problem of Dijkstra lies on the fact that the query was not correctly structured and the internal operations took a long time to be executed.

7.1.3 CONCLUSIONS

In the previous sections two scenarios were analyzed in order to study the behavior of IEA together with the new data structure, and to compare its operation to that of Dijkstra and the current database access.

From the Scenario I it can be concluded that IEA provides more satisfactory results. Not only its performance is found to be more efficient in terms of time and results, but also the

data structure and the query used to access the database in order to obtain the routes used by the algorithm was found to play a major part on effectively reducing the entire process' time.

On the other hand, when considering Scenario II the real advantages provided by the new data structure can be appreciated. The new query provides more results than the former one even if the origin and destination are the same in both, as explained in section 6.4. What this implies is the necessity of an algorithm that can take that many routes and still operate as competitively as before, if so a little bit slower than before due to an increase of the number of internal iterations.

Dijkstra was proven to be completely inadequate and unprepared for a high number of routes, as the time it took to iterate over the array exceeded the default time out set by PHP. Therefore, the implementation of IEA was essential for the application of the new query.

Also, even if IEA ended up taking more time than Dijkstra on its execution due to the higher quantity of routes, the high amount of time spent by the former query when applying Dijkstra resulted in IEA's better performance, thanks to the speed of the new query as well as that of the algorithm.

Consequently, it can be concluded that IEA, together with the new data structure, provide a much more efficient and competent performance than the one already implemented in the company. And one that can be considered to substitute the previous path-searching arrangement.

7.2 IEA PERFORMANCE ANALYSIS

Once IEA was compared to Dijkstra and it was concluded that the new route-searching structure is more efficient, a thorough analysis was carried out in order to determine the most convenient configuration to use the algorithm. This study is essential when considering the increasing amount of time the algorithm takes to obtain an incremental number of paths.

Several possibilities were studied:

- **Option I:** A maximum number of paths is fixed for every search.
- **Option II:** The number of paths to be calculated each time varies depending on the number of routes.
- **Option III:** The algorithm execution time is fixed for every search.

7.2.1 OPTION I: FIXED NUMBER OF PATHS

This was the first approach chosen and the one used to develop the algorithm. However, the higher number of routes returned by the new query lead to analyze this possibility more in detail in order to study its advantages and disadvantages.

Figure 76 shows the behavior of the algorithm with an increasing number of routes as well as iterations. For this test, every execution was carried out 10 times and an average of the execution time was calculated.

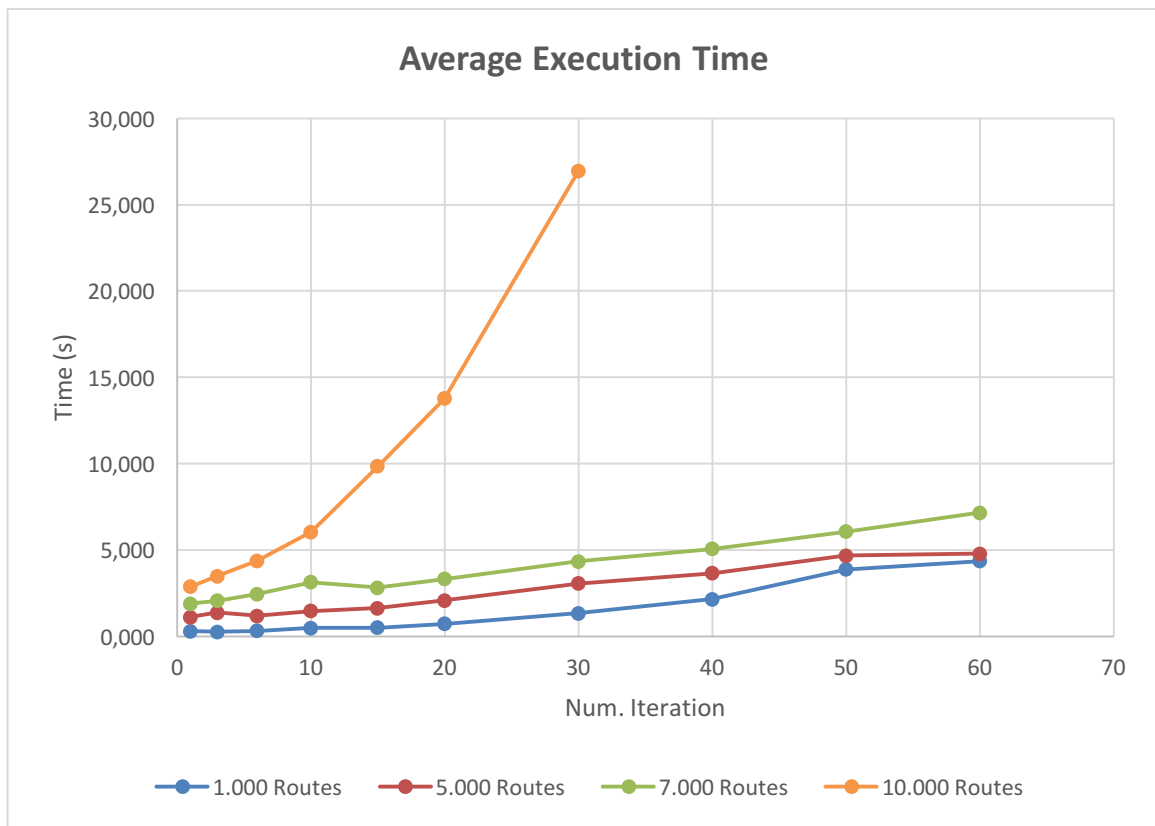


Figure 76. IEA average execution time for different set of routes.

As it can be seen, the algorithm average execution time increases with the number of paths to be obtained. Also, this number increments as the amount of routes passed on to the algorithm grows. The greatest difference among the performances represented is that one obtained when employing 10.000 routes. The time spent by the algorithm increases exponentially rather than linearly as in the rest of the cases. This entails a very serious consequence, which is the impossibility of calculating more than 30 results. This happens because the algorithm takes more time than the limit allowed by the system (around 20 seconds) causing its timeout.

Leaving this particular case aside, as it only happens when the amount of routs is fairly high, the algorithm has a good performance for the rest of the routes amounts. However, it must be taken into account that the cases where the algorithm might be rendered useless due to the high quantity of routes may happen more often than not. This is because that amount of routes are available when both origin and destination are concurrent nodes in the graph, just as Ho Chi Minh and Da Nang in Vietnam.

As a conclusion, fixing the amount of paths to be found every time should not be a possibility as the execution of the algorithm depends on the number of routes introduced. The consequence would be that the algorithm would return results some times when the amount of routes was not that high. If the number of routes was too high, it would time out providing no results, even if the labels have already been created. This is one of the drawbacks of the algorithm, as it can only return the paths once all the required ones have been calculated.

Therefore, this option is not considered.

7.2.2 OPTION II: VARIABLE NUMBER OF PATHS

Following the same reasoning as in Option I, one solution to the problem found would be to apply a variable number of paths to be obtained depending on the quantity of routes returned by the query.

For those trips where origin and destination are relevant enough for the query to provide a high amount of routes, then the maximum number of paths could be reduced. This way the

algorithm would not take too much time in its execution. On the other hand, for those destinations where not too many routes were found that connects them, more paths could be set to be obtained.

While this seems a very practical approach, it does not solve the problem of the routes interconnection complexity. Sometimes the algorithm has been found to take more time finding the correct interconnection between routes and this is due only to the routes' characteristics, and not because of the amount of them. This implies that the time the algorithm is going to calculate the paths is still unknown even if the quantity of routes has already been calculated. If the algorithm keeps conditioning its execution to the maximum number of paths to be obtained, it could happen that sometimes it needs more time, leading to a possible time out while in other situations the algorithm it works correctly.

Therefore, the uncertainty attached to this option due to the routes complexity as well as the unknown time the algorithm might take to calculate the routes leaves this option out of consideration.

7.2.3 OPTION III: FIXED EXECUTION TIME

As both Option I and Option II have already been discarded and both were set on fixing the number of paths the algorithm had to calculate, Option III is set on another approach: fixing the execution time rather than the number of paths.

It has been found that knowing the time a process is going to take provides a more user-friendly experience, as results are going to be shown after a certain and known amount of time. Hence, some tests were applied to this new modification of IEA's implementation where the number of paths to be obtained has not been determined, but rather the maximum time the algorithm must be running.

For the following test, the execution times were fixed to 2, 5, 10, 15 and 20 seconds and different routes set were applied:

- Vientiane –Luang Prabang: 1000 routes.
- Vinh – Dong Hong: 5000 routes.
- Hanoi – Vinh: 7000 routes.
- Ho Chi Minh – Da Nang: 10.000 routes.

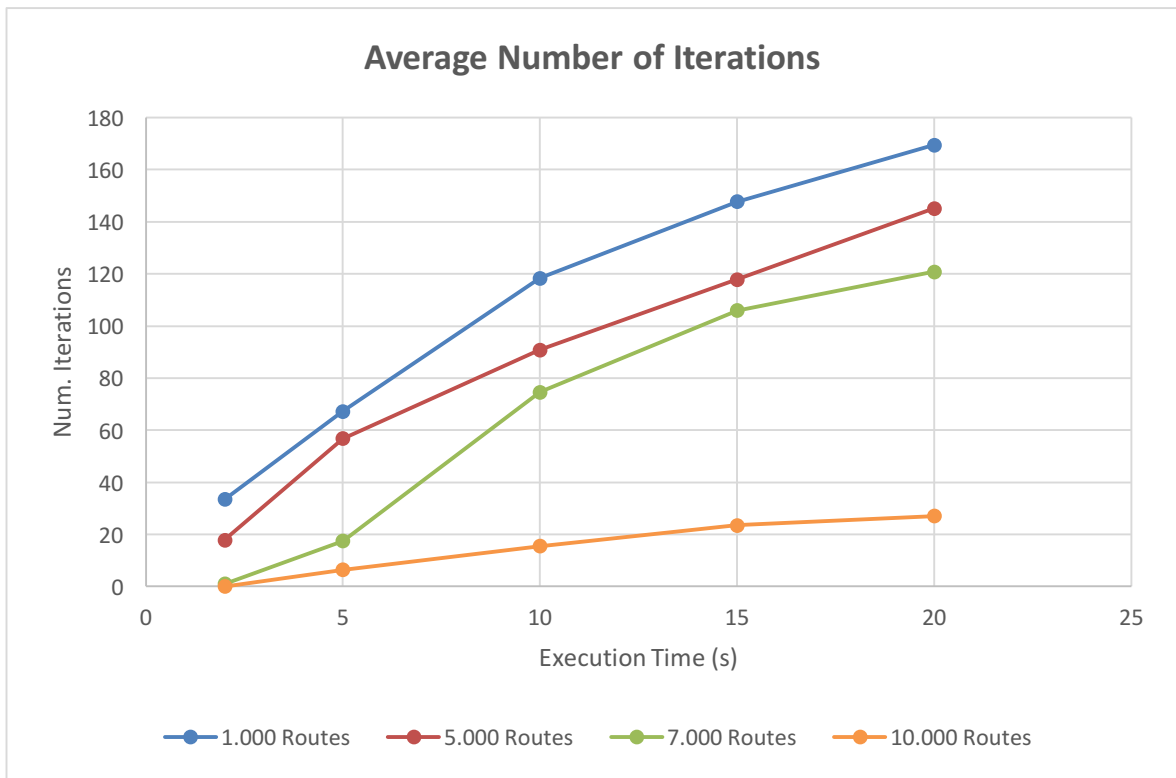


Figure 77. IEA average number of iterations with a fixed execution time.

Figure 77 depicts the average number of iterations obtained when fixing the execution time for each one of the routes set. When only 1000 routes have been found between Vientiane and Luang Prabang, IEA can obtain almost 40 paths with only 2 seconds of execution time which is very efficient. And as the maximum execution time increases, the amount of paths found also increments.

As the number of routes increments, the number of paths found is reduced. But still for 5000 and 7000 routes, this amount is fairly high. An evident change can be observed when 10.000

routes are introduced into the algorithm. However, with only 10 seconds of execution time almost 20 results can be found, which is more than necessary.

It needs to be taken into account that the results provided by the algorithm are for combined routes. The direct routes are extracted directly from the database and presented to the user after making sure they are still available. Usually, when more than 10.000 routes can be found between two points, it means these are important destinations. This also implies that a rather large amount of direct routes is going to be shown to the user before the combined results that will appear at the end of the list. Therefore, 10 seconds of execution time and almost 20 combined results to be shown provide a very good balance between efficiency and time in this case, when the user is not that interested in the combined options most of the times for these types of destinations.

7.2.4 CONCLUSIONS ON IEA PERFORMANCE ANALYSIS

As a result, it can be concluded that this approach is the one best suited for the company purposes and the one that is going to be adopted.

It is necessary to mention that, even if the algorithm is set to be time-limited, a condition will be added. The algorithm will keep working even if the execution time has already expired and no results have been found. In this case the algorithm must keep functioning until at least a result is found, or the end of iterations have been reached and no results have been obtained. This condition has been added because of those cases in which the number of routes is so high the algorithm takes more time than the limit introduced. This way it can be ensured that at least one result is going to be shown even if more time is needed. Moreover, when there are so many routes to be managed it means that a lot of direct routes will be shown and this will reduce the negative impact on the algorithm's extended execution time, as the user will already have several results to analyze.

Chapter 8. CONCLUSIONS AND FUTURE CONTRIBUTIONS

The main objective of this project is to improve the overall performance of the multiple-route search procedure implanted in Baolau. The first step has consisted on choosing and implementing a new algorithm that adjusts best to the company's needs and route structure. For that an extensive and detailed research was carried out in order to provide the best solution. The *Iterative Enumeration Algorithm* (IEA) was selected among all the possibilities studied in Chapter 5.

What earned **IEA** the possibility of becoming the potential substitute of Dijkstra's algorithm currently employed in the company, was the fact that IEA could be **easily adapted to time-dependent scenarios**. In these scenarios routes are subjected to timetables. Combinational routes must take this fact into account because some routes will no longer be available at a certain node if the arrival time has already surpassed the departure time of a certain trip. IEA effortlessly overcomes this difficulty thanks to its labeled structure that allows to store the schedule information of each route used to arrive at a certain node. Therefore, IEA was included into the system employing the former data structure in which the routes introduced into the algorithm were differentiated by its departure date. Moreover, only routes on the next 2 days after the travelling date entered by the user were selected and passed on to the algorithm, as explained in section 6.3.

The next step to keep improving the system's search performance was to **change all the data structure used to access the routes** introduced to the algorithm. It was found that the query launched to obtain the routes took up much time. Consequently, a new table was created in the database that included all the routes that had existed so far but with the peculiarity of not including the date, just the time of arrival and departure. This way, the algorithm has more possible routes to work with and so more result to show. Afterwards, the

actual routes table is accessed in order to check which of the paths procured by the algorithm actually exist and are available for the date introduced by the user into the system. With this new approach, the query launched was also modified and built to work in a much faster way than the previous one. However, **while the time spent in executing the query had been reduced, the algorithm needed more time to operate with the increased number of routes**. However, the global outcome still surpassed that of Dijkstra in terms of time and resources savings and quantity and amount of results obtained.

As a conclusion, not only was it possible to improve the multiple-route trip calculation by means of introducing a new algorithm (IEA), but also by modifying the data structure employed by that same algorithm. It was also proven that **this algorithm was perfectly suited for adopting the changes introduced in the system** by the use of this new data structure thanks to the labelled arrangement inherent to its internal anatomy. Also, this labelled structure was what allowed the algorithm to be **used to obtain additional solutions** once the algorithm had already been called and the results shown to the user. This has successfully been applied to allow the user to load more results after having examined the ones already displayed in the page. Also, the **new filters applied to these results**, which do not affect the algorithm in any way, could become an advantageous feature for the web-page as it allows a more customized searching experience for the user.

Internally, a future modification of this project would be its applicability to round-trip travels and adaptability to ensuing modifications included and applied into the web-page. Externally, some future contribution could be the extendibility of the algorithm to new scenarios and situations with time-dependent characteristics and in need of obtaining different paths from one source to a certain destination in a graph.

All in all, algorithms are gaining an increasing growth in computerized mechanisms extending to most of fields around the world. Thanks to their complexity now supported by the current powerful and developed computers, algorithms are becoming an important part in every-day business and careers. IEA could become the next extended algorithm for multiple-route calculation procedures.

Chapter 9. BIBLIOGRAPHY

- [1] Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein. *Introduction to Algorithms. 3rd Edition*. MIT Press, Cambridge, Massachusetts London, England. 2009.
- [2] Grégoire Scano, Marie-José Huguet and Sandra Ulrich Ngueveu. *Adaptations of k-Shortest Path Algorithms for Transportation Networks*. 6th IESM Conference, October 2015, Seville, Spain.
- [3] David Eppstein. *Finding the k Shortest Paths*. March 31, 1997. Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, eppstein@ics.uci.edu, <http://www.ics.uci.edu/~eppstein/>.
- [4] Víctor M. Jiménez and Andrés Marzal. *A Lazy Version of Eppstein's K Shortest Paths Algorithm*. 2003, DLSI, Universitat Jaume I, 12071 Castellón, Spain.
- [5] A. Marino. *Analysis and Enumeration*, 2015. Atlantis Studies in Computing 6. Algorithms for Biological Graphs.
- [6] Qiujin Wu, Joanna Hartley. *Using K-shortest paths algorithms to accomodate user preferences in the optimization of public transport travel*. School of Computing and Technology, The Nottingham Trent University, Burton Street, Nottingham, NG1 4BU, U.K.
- [7] Daniel Delling and Dorothea Wagner. *Time-Dependent Route Planning*. Universität Karlsruhe (TH), 76128 Karlsruhe, Germany.
- [8] Robert Geisberger. *Advanced Route Planning in Transportation Networks*. 4. Februar 2011. Fakultät für Informatik des Karlsruher Instituts für Technologie. Karlsruhe, Germany.
- [9] Konstantinos G. Zografos and Konstantinos N. Androutsopoulos. *Algorithms for Itinerary Planning in Multimodal Transportation Networks*. IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS, VOL. 9, NO. 1. March 2008.
- [10] Liviu Coftas, Andreea Diosteanu. *Public Transport Route Finding using a Hybrid Genetic Algorithm*. Informatica Economică vol. 15, no. 1/2011. Academy of Economic Studies, Bucharest, Romania.

- [11] Dorothea Wagner. *Route Planning Algorithms in Transportation Networks*. 7th International Network Optimization Conference. May 18, 2015, Warsaw, Poland. Karlsruher Instituts für Technologie. Karlsruhe, Germany.
- [12] Matthias Müller–Hannemann and Mathias Schnee. *Paying Less for Train Connections with MOTIS*. 5th Workshop on Algorithmic Methods and Models for Optimization of Railways, 2006. Darmstadt University of Technology, Computer Science, 64289 Darmstadt, Hochschulstraße 10, Germany.
- [13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. *A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks*. 2010. Microsoft Research Silicon Valley.
- [14] Tatsuya Ohshima. *A Landmark Algorithm for the Time-Dependent Shortest Path Problem*. February 2008. Department of Applied Mathematics and Physics Graduate School of Informatics, Kyoto University.
- [15] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner and Renato F. Werneck. *Route Planning in Transportation Networks*. April 17, 2015. Microsoft Research Silicon Valley.
- [16] Liang Zhao, Tatsuya Ohshima, Hiroshi Nagamochi. *A* Algorithm for the time-dependent shortest path problem*. School of Informatics Kyoto University Yoshidahonmachi, Sakyo, Kyoto, Japan.
- [17] Yansen Darmaputra. *An Application of Heuristic Route Search Techniques for a Scalable Flight Search System*. June 2008. Database and Artificial Intelligence Group Institute for Information Systems Faculty of Informatics Technische Universität Wien. Vienna, Austria.
- [18] *Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)*. 2018. 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305.
<https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>
- [19] Siddharth Mangalik Rahul Verma Joshua Ethridge Michael Witkovsky. *Pathfinding*.
<http://www3.cs.stonybrook.edu/~cse352/T5talk.pdf>
- [20] Vaidehi Joshi. *Finding The Shortest Path, With A Little Help From Dijkstra*. Medium, October 16, 2017. <https://medium.com/basecs/finding-the-shortest-path-with-a-little-help-from-dijkstra-613149fbc8e>
- [21] Vaidehi Joshi. *Demystifying Depth-First Search*. Medium, April 3, 2017.
<https://medium.com/basecs/demystifying-depth-first-search-a7c14cccf056>

- [22] Vaidehi Joshi. *Going Broad In A Graph: BFS Traversal*. Medium, September 18, 2017. <https://medium.com/basecs/going-broad-in-a-graph-bfs-traversal-959bd1a09255>
- [23] Vaidehi Joshi. *Deep Dive Through A Graph: DFS Traversal*. Medium, September 25, 2017. <https://medium.com/basecs/deep-dive-through-a-graph-dfs-traversal-8177df5d0f13>
- [24] Vaidehi Joshi. *Less Repetition, More Dynamic Programming*. Medium, October 24, 2017. <https://medium.com/basecs/less-repetition-more-dynamic-programming-43d29830a630>
- [25] John Hershberger* Matthew Maxel† Subhash Suri. *Finding the k Shortest Simple Paths: A New Algorithm and its Implementation*. <http://www.siam.org/meetings/alnex03/Abstracts/jhershberger.pdf>
- [26] *Yen's Algorithm*. Wikipedia, The Free Encyclopedia. May 2018. https://en.wikipedia.org/wiki/Yen%27s_algorithm#cite_note-6
- [27] Amit Patel. *Introduction to A**. Red Blob Games, From Amit's Thoughts on Pathfinding. April 2018. <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>
- [28] *Algorithms*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/fundamentals-of-algorithms/#DynamicProgramming>
- [29] *Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm)*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/>
- [30] *Greedy Algorithms | Set 5 (Prim's Minimum Spanning Tree (MST))*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/>
- [31] *Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>
- [32] *Greedy Algorithms | Set 8 (Dijkstra's Algorithm for Adjacency List Representation)*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/>

- [33] *Dynamic Programming | Set 23 (Bellman–Ford Algorithm)*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305.
<https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/>
- [34] *Dynamic Programming | Set 16 (Floyd Warshall Algorithm)*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305.
<https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/>
- [35] *Johnson’s algorithm for All-pairs shortest paths*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305.
<https://www.geeksforgeeks.org/johnsons-algorithm/>
- [36] *Shortest Path in Directed Acyclic Graph*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/>
- [37] *Some interesting shortest path questions | Set 1*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305.
<https://www.geeksforgeeks.org/interesting-shortest-path-questions-set-1/>
- [38] *Shortest path with exactly k edges in a directed and weighted graph*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305.
<https://www.geeksforgeeks.org/shortest-path-exactly-k-edges-directed-weighted-graph/>
- [39] *A* Search Algorithm*. GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/a-search-algorithm/>
- [40] Sahar Idwan and Wael Etaiwi. *Dijkstra Algorithm Heuristic Approach for Large Graph*. Journal of Applied Sciences, 11: 2255-2259. October 06, 2011.
<https://scialert.net/fulltext/?doi=jas.2011.2255.2259&org=11>
- [41] Kinght. *Eppstein k -best*. ISI. <https://www.isi.edu/natural-language/people/epp-cs562.pdf>
- [42] Yajun Yang, Hong Gao, Jeffrey Xu Yu, Jianzhong Li. *Finding the Cost-Optimal Path with Time Constraint over Time-Dependent Graphs*. Proceedings of the VLDB Endowment, Vol. 7, No. 9. Copyright 2014 VLDB Endowment 2150-8097/14/05. China.
- [43] *Iterative deepening A**. Wikipedia, The Free Encyclopedia. February, 2018.
https://en.wikipedia.org/wiki/Iterative_deepening_A*
- [44] Alan Mackworth. *Multiple Path Pruning, Iterative Deepening and IDA*. UBC CS 322 – Search 7. January 23, 2013.
- [45] Amit Patel. *Variants of A**. From Amit’s Thoughts on Pathfinding. 11 April 2018.
<http://theory.stanford.edu/~amitp/GameProgramming/Variations.html>
-

- [46] K Hong. *DIJKSTRA'S SHORTEST PATH ALGORITHM*. BogoToBogo, San Francisco.
- [47] *Time Complexity*. Wikipedia, The Free Encyclopedia. May, 2018.
https://en.wikipedia.org/wiki/Time_complexity
- [48] *Big O Notation*. Wikipedia, The Free Encyclopedia. May, 2018.
https://en.wikipedia.org/wiki/Big_O_notation
- [49] *Adjacency Matrix*. Wolfram Math World.
<http://mathworld.wolfram.com/AdjacencyMatrix.html>
- [50] Prateek Garg. *Breadth First Search*. Hackerearth.
<https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/>
- [51] Prateek Garg. *Depth Frist Search*. Hackerearth.
<https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/>
- [52] Harika Reddy. *PATH FINDING - Dijkstra's and A* Algorithm's*. December 13, 2013.
- [53] Merlin Puthuparampil. *Report Dijkstra's Algorithm*.
<https://cs.nyu.edu/courses/summer07/G22.2340-001/Presentations/Puthuparampil.pdf>
- [54] *What are advantage and disadvantage of Dijkstra's Algorithm?* GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. Practice Beta.
<https://practice.geeksforgeeks.org/problems/what-are-advantage-and-disadvantage-of-dijkstras-algorithm>
- [55] Robert Sedgewick. *Shortest Paths Graph Algorithms*. Pearson, InformIT. Mar 5, 2004.
<http://www.informit.com/articles/article.aspx?p=169575&seqNum=8>
- [56] Robin. *Breadth First Search*. Artificial Intelligence. December 16th, 2009.
<http://intelligence.worldofcomputing.net/ai-search/breadth-first-search.html>
- [57] Robin. *Depth First Search*. Artificial Intelligence. December 18th, 2009.
<http://intelligence.worldofcomputing.net/ai-search/depth-first-search.html>
- [58] Robin. *A Star Algorithm*. Artificial Intelligence. December 18th, 2009.
<http://intelligence.worldofcomputing.net/ai-search/a-star-algorithm.html>
- [59] *What are real life applications of Dijkstra's Algorithm?* GeeksforGeeks, 710-B, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. Practice Beta.
<https://practice.geeksforgeeks.org/problems/what-are-real-life-applications-of-dijkstras-algorithm>
- [60] *Shortest Path Problem*. Wkipedia, The Free Encyclopedia. May, 2018.
https://en.wikipedia.org/wiki/Shortest_path_problem

- [61] *Dijkstra's Algorithm*. Wikipedia, The Free Encyclopedia. April, 2018.
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- [62] *Bellman-Ford Algorithm*. Wikipedia, The Free Encyclopedia. March, 2018.
https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
- [63] *Floyd–Warshall algorithm*. Wikipedia, The Free Encyclopedia. April, 2018.
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm#Applications_and_generalizations
- [64] *Johnson's Algorithm*. Wikipedia, The Free Encyclopedia. December, 2017.
https://en.wikipedia.org/wiki/Johnson%27s_algorithm
- [65] *A* Search Algorithm*. Wikipedia, The Free Encyclopedia. April, 2018.
https://en.wikipedia.org/wiki/A*_search_algorithm
- [66] *Heuristic (Computer Science)*. Wikipedia, The Free Encyclopedia. April, 2018.
[https://en.wikipedia.org/wiki/Heuristic_\(computer_science\)](https://en.wikipedia.org/wiki/Heuristic_(computer_science))
- [67] *Best-first Search*. Wikipedia, The Free Encyclopedia. January, 2017.
https://en.wikipedia.org/wiki/Best-first_search
- [68] *Search Algorithm*. Wikipedia, The Free Encyclopedia. April, 2018.
https://en.wikipedia.org/wiki/Search_algorithm
- [69] *Baolau*. PDD Building, 162 Pasteur Street, Ward Ben Nghe, District 1, Ho Chi Minh, Vietnam. <https://www.baolau.com/>.
- [70] *International tourist arrival in Asia and the Pacific from 2010 to 2016, by región (in millions)*. The Scientific Portal. <https://www.statista.com/statistics/261703/international-tourist-arrivals-in-asia-and-the-pacific-by-region/>.
- [71] *PHP General Page*. PHP.net . <http://php.net/>
- [72] *MAMP & MAMP PRO*. MAMP. <https://www.mamp.info/de/>
- [73] *MySQL*. MySQL Home Page. <https://www.mysql.com/>
- [74] *Bitbucket*. Atlassian Bitbucket. <https://bitbucket.org/product>
- [75] *Sourcetree | Free Git GUI for Mac and Windows*. Sourcetree.
<https://www.sourcetreeapp.com/>
- [76] *Edsger Dijkstra*. The Centre for Computing History Rene Court Coldhams Road Cambridge CB1 3EW <http://www.computinghistory.org.uk/det/4179/Edsger-Dijkstra/>
- [77] *Algorithm Classification*. Wikipedia, The Free Encyclopedia. May, 2018.
<https://en.wikipedia.org/wiki/Algorithm#Classification>
-

- [78] Misa, Thomas J. *An Interview With Edsger W. Dijkstra*. Communications Of The ACM. 2010. <https://cacm.acm.org/magazines/2010/8/96632-an-interview-with-edsger-w-dijkstra/fulltext>
- [79] Thomas Cormen, Devin Balkcom. *Big-O notation*. Khan Academy. <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation>
- [80] Rob Bell. *A beginner's guide to Big O notation*. Rob Bell. June 23rd de 2009. <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>
- [81] *Basics of Greedy Algorithms*. Hackerearth. <https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms>
- [82] *Recursion*. GeeksforGeeks, Advant Navis Business Park, Sector-142, Noida, Uttar Pradesh – 201305. <https://www.geeksforgeeks.org/recursion/>
- [83] Amit Patel. *Heuristics*. From Amit's Thoughts on Pathfinding. 11 April 2018. <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
- [84] *David Eppstein*. Wikipedia, The Free Encyclopedia. May, 2018. https://en.wikipedia.org/wiki/David_Eppstein

APPENDIX A

COMPLETE SOLUTION TO IEA WITHOUT CYCLES

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],1,F}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		
2		T;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}		F;{12,x2[1],1,F}
			{5,x2[1],2,F}			
3		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
			{5,x2[1],2,T}			{7,x3[2],2,F}
4		T;{4,x1[1],1,T}	T;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	T;{12,x2[1],1,F}
			{5,x2[1],2,T}			{7,x3[2],2,T}
<i>Path 1: x1, x2, x3, x6</i>						
5		F;{4,x1[1],1,T}	F;{8,x1[1],1,F}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
			{5,x2[1],2,T}			{7,x3[2],2,T}
6		F;{4,x1[1],1,T}	T;{8,x1[1],1,T}	F;{10,x1[1],1,F}	F;{8,x3[2],1,F}	F;{12,x2[1],1,F}
		{9,x3[1],2,F}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
						{10,x3[1],3,F}
7		F;{4,x1[1],1,T}	T;{8,x1[1],1,T}	F;{10,x1[1],1,F}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{9,x3[1],2,F}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
						{10,x3[1],3,F}
						{12,x5[1],4,F}
8		T;{4,x1[1],1,T}	T;{8,x1[1],1,T}	F;{10,x1[1],1,F}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
						{10,x3[1],3,F}
						{12,x5[1],4,F}
						{17,x2[2],5,F}
9		T;{4,x1[1],1,T}	T;{8,x1[1],1,T}	T;{10,x1[1],1,T}	T;{8,x3[2],1,T}	F;{12,x2[1],1,F}
		{9,x3[1],2,T}	{5,x2[1],2,T}		{11,x3[1],2,F}	{7,x3[2],2,T}
					{18,x4[1],3,F}	{10,x3[1],3,F}
						{12,x5[1],4,F}
						{17,x2[2],5,F}

10		T; $\{4,x1[1],1,T\}$	T; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	T; $\{12,x2[1],1,F\}$
		$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,F\}$	$\{7,x3[2],2,T\}$
					$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
						$\{12,x5[1],4,F\}$
						$\{17,x2[2],5,F\}$
<i>Path 2: x1, x3, x6</i>						
11		T; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	F; $\{12,x2[1],1,F\}$
		$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,F\}$	$\{7,x3[2],2,T\}$
					$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
						$\{12,x5[1],4,F\}$
						$\{17,x2[2],5,F\}$
12		T; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	T;$\{12,x2[1],1,T\}$
		$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,F\}$	$\{7,x3[2],2,T\}$
					$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
						$\{12,x5[1],4,F\}$
						$\{17,x2[2],5,F\}$
<i>Path 3: x1, x2, x6</i>						
13		F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	F; $\{12,x2[1],1,T\}$
		$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,F\}$	$\{7,x3[2],2,T\}$
					$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
						$\{12,x5[1],4,F\}$
						$\{17,x2[2],5,F\}$
14		F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	T; $\{12,x2[1],1,T\}$
		$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,F\}$	$\{7,x3[2],2,T\}$
					$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
						$\{12,x5[1],4,T\}$
						$\{17,x2[2],5,F\}$
<i>Path 4: x1, x2, x3, x5, x6</i>						
15		F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	F; $\{8,x3[2],1,T\}$	F; $\{12,x2[1],1,T\}$
		$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,F\}$	$\{7,x3[2],2,T\}$
					$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
						$\{12,x5[1],4,T\}$
						$\{17,x2[2],5,F\}$

16	F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	F; $\{12,x2[1],1,T\}$
	$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,T\}$	$\{7,x3[2],2,T\}$
				$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
					$\{12,x5[1],4,T\}$
					$\{17,x2[2],5,F\}$
					$\{15,x5[2],6,F\}$
17	F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	T; $\{12,x2[1],1,T\}$
	$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,T\}$	$\{7,x3[2],2,T\}$
				$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
					$\{12,x5[1],4,T\}$
					$\{17,x2[2],5,F\}$
					$\{15,x5[2],6,T\}$
<i>Path 5: x1, x3, x5, x6</i>					
18	F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	F; $\{8,x3[2],1,T\}$	F; $\{12,x2[1],1,T\}$
	$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,T\}$	$\{7,x3[2],2,T\}$
				$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
					$\{12,x5[1],4,T\}$
					$\{17,x2[2],5,F\}$
					$\{15,x5[2],6,T\}$
19	F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	F; $\{8,x3[2],1,T\}$	T; $\{12,x2[1],1,T\}$
	$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,T\}$	$\{7,x3[2],2,T\}$
				$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
					$\{12,x5[1],4,T\}$
					$\{17,x2[2],5,T\}$
					$\{15,x5[2],6,T\}$
<i>Path 6: x1, x3, x2, x6</i>					
20	F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	F; $\{8,x3[2],1,T\}$	F; $\{12,x2[1],1,T\}$
	$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,T\}$	$\{7,x3[2],2,T\}$
				$\{18,x4[1],3,F\}$	$\{10,x3[1],3,T\}$
					$\{12,x5[1],4,T\}$
					$\{17,x2[2],5,T\}$
					$\{15,x5[2],6,T\}$

21	F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	F; $\{12,x2[1],1,T\}$
	$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,T\}$	$\{7,x3[2],2,T\}$
				$\{18,x4[1],3,T\}$	$\{10,x3[1],3,T\}$
					$\{12,x5[1],4,T\}$
					$\{17,x2[2],5,T\}$
					$\{15,x5[2],6,T\}$
22	F; $\{4,x1[1],1,T\}$	F; $\{8,x1[1],1,T\}$	T; $\{10,x1[1],1,T\}$	T; $\{8,x3[2],1,T\}$	T; $\{12,x2[1],1,T\}$
	$\{9,x3[1],2,T\}$	$\{5,x2[1],2,T\}$		$\{11,x3[1],2,T\}$	$\{7,x3[2],2,T\}$
				$\{18,x4[1],3,T\}$	$\{10,x3[1],3,T\}$
					$\{12,x5[1],4,T\}$
					$\{17,x2[2],5,T\}$
					$\{15,x5[2],6,T\}$
<i>Path 7: x1, x4, x5, x6</i>					

Figure 78. IEA complete label structure without cycles.

COMPLETE SOLUTION TO IEA IN TIME-DEPENDENT SCENARIO

	x1	x2	x3	x4	x5	x6
1		F;{4,x1[1],8:00,8:15,1,F}	F;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}		
2		T;{4,x1[1],8:00,8:15,1,T}	F;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}		F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,F}			
3		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}	F;{8,x3[2],8:35,8:45,1,F}	F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,T}			
4		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	F;{10,x1[1],8:00,8:30,1,F}	T;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,T}			{12,x5[1],8:50,9:00,2,F}
5		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,F}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,F}
6		T;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	T;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,F}
<i>Path 1: x1, x2, x6</i>						
7		F;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,F}
8		F;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,F}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	T;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,T}
<i>Path 2: x1, x2, x3, x5, x6</i>						
9		F;{4,x1[1],8:00,8:15,1,T}	F;{8,x1[1],8:10,8:35,1,F}	T;{10,x1[1],8:00,8:30,1,T}	F;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,T}
10		F;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,T}	T;{10,x1[1],8:00,8:30,1,T}	F;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,T}
					{11,x3[1],8:35,8:45,3,F}	
11		F;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,T}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,T}
					{11,x3[1],8:35,8:45,3,T}	{15,x5[3],8:50,9:00,3,F}
12		F;{4,x1[1],8:00,8:15,1,T}	T;{8,x1[1],8:10,8:35,1,T}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	T;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,T}
					{11,x3[1],8:35,8:45,3,T}	{15,x5[3],8:50,9:00,3,T}
<i>Path 3: x1, x3, x5, x6</i>						
13		F;{4,x1[1],8:00,8:15,1,T}	F;{8,x1[1],8:10,8:35,1,T}	T;{10,x1[1],8:00,8:30,1,T}	F;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,F}	{12,x5[1],8:50,9:00,2,T}
					{11,x3[1],8:35,8:45,3,T}	{15,x5[3],8:50,9:00,3,T}
14		F;{4,x1[1],8:00,8:15,1,T}	F;{8,x1[1],8:10,8:35,1,T}	T;{10,x1[1],8:00,8:30,1,T}	T;{8,x3[2],8:35,8:45,1,T}	F;{12,x2[1],8:20,8:30,1,T}
			{5,x2[1],8:20,8:25,2,T}		{18,x4[1],8:35,8:55,2,T}	{12,x5[1],8:50,9:00,2,T}
					{11,x3[1],8:35,8:45,3,T}	{15,x5[3],8:50,9:00,3,T}
<i>No Path</i>						


Figure 79. IEA complete label structure in time-dependent scenario.


SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)


Specification and Description Language (SDL) is an object oriented language defined by The International Telecommunications Union - Telecommunications Standardization Sector (ITU - T). It is used in event-driven and real-time situations where information is exchanged. Therefore, simulations are modeled by it.


The most common graphs used are the following:


- **Process**
Indicates the process that takes the action.



- **State**
Represents the state in which the process jumps into.


- **Input message**
Symbolizes the reception of a message.


- **Output message**
A message that is sent.


- **Timer**
Indicates that a timer was started.


- **Decision**
Identifies more than one possibility in the information flow.


- **Task**
Represents an activity that must be performed.

