A Graph Model of Intermodal Transportation Networks

Rui P. P. Cardoso, Antonio D. Casimiro, and Rosaldo J. F. Rossetti, *Member, IEEE*
Artificial Intelligence and Computer Science Lab
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Rua Dr Roberto Frias, s/n, 4200-465, Porto, Portugal
{up201305469, up201305244, rossetti}@fe.up.pt

***Abstract*—The aim of this research is to address the problem of planning a trip between two points of an intermodal public transportation network, i.e., a network consisting of several different means of transportation which are often sought in parallel by users. An example of such a network is one which comprises bus and metro lines in a large city. If the network is abstracted by a graph representation, a possible approach to this problem consists in finding an optimal path between the source and destination nodes of that graph. There might be several criteria for selecting an optimal path in such a network. This text explores some renowned algorithms for finding an optimal path between two nodes of a graph, namely Dijkstra's algorithm and the A* algorithm, suggesting improvements whenever appropriate.**

***Keywords*—graph, network, intermodal, transportation, best path, Dijkstra, A*, Fibonacci.**

I. Introduction

This paper seeks to address the problem of finding an optimal path[1] between two points of an intermodal transportation network. A network consists of a set of points, also called nodes, and a set of links between pairs of points, also called arcs. It is therefore often abstracted by the mathematical concept of graph[5]. Each node in the network represents a station or a stop of a public transportation line, has an associated position in the xOy-plane, and an *average wait time*. Each arc represents a route between two nodes. Such a route could either be associated with a given transportation line or mean that the traveller is walking from one node to another in order to change lines. In either case, an arc has an *average travel time* and a *travelled distance*, both of which may be considered as parameters for calculating the *cost* of traversing the network. Derived from these, an arc also has an *average speed* of traversal. This text deals exclusively with *directed* networks, i.e., networks consisting only of directed arcs between pairs of nodes. The reason for this is that a round trip does not necessarily imply that a given node should host both directions of a transportation line. This is particularly true in the case of buses which travel one-way streets.

This work contributes with a practical perspective for modelling multimodal transportation networks, emphasising on implementation aspects and the computational efficiency of shortest path algorithms essential to support, for instance, multimodal traffic assignment routines and simulation procedures. The remainder of this paper is structured as follows. Section II covers preliminary aspects and definitions needed to support all discussions throughout the paper. Section III discusses on Dijkstra's algorithm whereas Section IV does on A*. The performance of the algorithms is analysed in Section V, and finally Section VI draws main conclusions and paves the way for future directions in this research.

[1]One may also refer to an *optimal path* as the *shortest path*. Throughout this paper, the authors shall make no distinction between these two terms.

II. Preliminaries

Before going further into detailing the proposed approach, it is important to establish the definition grounds on which discussion will develop, as follows:

- G represents the graph which abstracts the network.
- V represents the set of nodes (or vertices) in the network.
- E represents the set of arcs (or edges) in the network.
- ν_i represents a node (or vertex) in the network.
- (x_i, y_i) represent the xy-coordinates of node ν_i.
- $\epsilon_{i,j}$ represents the directed arc (or edge) connecting node ν_i to node ν_j.
- $l_{i,j}$ represents the transportation line, if any, associated with the directed arc which connects node ν_i to node ν_j.
- t_{ν_i} represents the average wait time associated with node ν_i.
- $t_{i,j}$ represents the average time necessary to traverse the arc $\epsilon_{i,j}$, which is ∞ if there is no such arc.
- $d_{i,j}$ represents the length of the arc $\epsilon_{i,j}$, which is ∞ if there is no such arc. Note that $d_{i,j} \geq \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}, \forall \epsilon_{i,j} \in G$.
- $\bar{\text{v}}_{i,j}$ represents the average speed when traversing the arc $\epsilon_{i,j}$.
- $\bar{\text{v}}_{max}$ represents the greatest average speed among all arcs in the network.
- $c_{i,j}$ represents the number of times the passenger changed transportation lines when travelling between

consecutive nodes ν_i and ν_j through arc $\epsilon_{i,j}$ (possible values are 0 and 1).

- $A_{i,j}$ represents the set of nodes in a certain path connecting node ν_i to node ν_j.
- $B_{i,j}$ represents the set of arcs in a certain path connecting node ν_i to node ν_j.
- $dist_{\nu_i}$ represents the distance from a certain start node to node ν_i along an optimal path connecting these two nodes.
- $path_{\nu_i}$ represents the node which precedes node ν_i in an optimal path connecting a certain start node to node ν_i.

A. *Input*

The input to this model shall be a graph representation of the network which is to be analysed, as described before. This graph is likely to need preprocessing so as to allow the application of the shortest path algorithms which are considered in this text, as described in subsection *D*.

B. *Output*

The output of the model is a sequence of nodes which correspond to an optimal path in the network. There might be several criteria for choosing an optimal path in the network, and this text deals with three such criteria, namely the path which represents the least travelled distance, the path which takes the least time to travel, and the path requiring the least number of changes between transportation lines. Note that it would also make sense to include the path which costs the least money to travel. However, such a criterion would be too case-specific. Nevertheless, for most intermodal transportation systems, the fewer the time one spends travelling, the fewer the amount of money one must pay for the trip, which is why finding the cheapest path might be deemed equivalent to finding the fastest path. Thus, there are three possible definitions for the *objective function*.

C. *Objective function*

The goal is to find an optimal path which connects node ν_i to node ν_j. As mentioned in the previous subsection, this text considers the following three possible definitions for the objective function:

- Minimize $\sum_{\epsilon_{k,k+1} \in B_{i,j}} d_{k,k+1}$, i.e., minimize the total travelled distance.
- Minimize $\sum_{\nu_k \in A_{i,j}} t_{\nu_k} + \sum_{\epsilon_{k,k+1} \in B_{i,j}} t_{k,k+1}$, i.e., minimize the total time necessary to go from node ν_i to node ν_j.
- Minimize $\sum_{\epsilon_{k,k+1} \in B_{i,j}} c_{k,k+1}$, i.e., minimize the total number of changes between transportation lines required to travel from node ν_i to node ν_j.

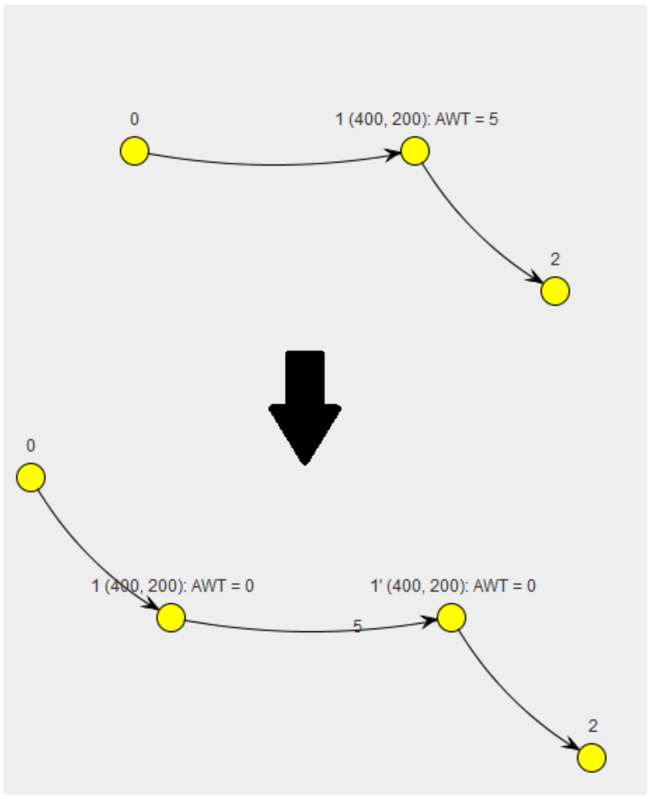

Fig. 1. Collapsing a weighted node into two unweighted nodes.

D. *Reducing the problem to a shortest path instance*

If either the first or the third definition of the objective function is chosen, i.e., if our aim is to minimize the total travelled distance, or the total number of changes between transportation lines, it turns out that the graph representing the network, just as it was defined at the beginning of this section, may already be used as input to the shortest path algorithms which were taken into account in this paper. However, this is not the case should we seek to find the path which results in the least travel time. The reason for this is that we defined t_{ν_i} as the average wait time associated with each node ν_i. The algorithms which were tested do not accept graphs with weighted nodes as input. We must thus reduce the problem at hands so as to make the algorithms applicable. This can be obtained by preprocessing the network in order to eliminate weighted nodes, while at the same time keeping the meaning of the problem unaltered. This is quite easy to achieve. Note that a node ν_i with an average wait time t_{ν_i} greater than zero can be collapsed into two nodes, ν_i' and ν_i'', which share the same position in the xOy-plane, each with a null average wait time, and connected by an arc of length zero and travel time $t_{i',i''} = t_{\nu_i}$, as illustrated in **Figure 1**.

Algorithm 1 describes a pseudocode implementation of the algorithm which preprocesses the network in order to eliminate weighted nodes.

III. Dijkstra's Algorithm

The first shortest path algorithm which was taken into account in this paper was the renowned Dijkstra's algorithm[2]. **Algorithm 2** describes this algorithm in pseudocode.

Algorithm 1 Algorithm for preprocessing a graph with weighted nodes.

```
function preProcessGraph
    for all $\nu_i \in V$ do
        if $t_{\nu_i} > 0$ then
            $G$.addNode($\nu'_i$)
            $(x'_i, y'_i) \leftarrow (x_i, y_i)$
            $t_{\nu'_i} \leftarrow 0$
            for all $\nu_k$ adjacent to $\nu_i$ do
                $G$.addEdge($\epsilon_{i',k}$)
                $l_{i',k} \leftarrow l_{i,k}$
                $t_{i',k} \leftarrow t_{i,k}$
                $d_{i',k} \leftarrow d_{i,k}$
                $\bar{\mathrm{v}}_{i',k} \leftarrow \bar{\mathrm{v}}_{i,k}$
                $c_{i',k} \leftarrow c_{i,k}$
                $G$.removeEdge($\epsilon_{i,k}$)
            end for
            $G$.addEdge($\epsilon_{i,i'}$)
            $l_{i,i'} \leftarrow$ null
            $t_{i,i'} \leftarrow t_{\nu_i}$
            $d_{i,i'} \leftarrow 0$
            $\bar{\mathrm{v}}_{i,i'} \leftarrow 0$
            $c_{i,i'} \leftarrow 0$
            $t_{\nu_i} \leftarrow 0$
        end if
    end for
end function
```

Algorithm 2 Dijkstra's Algorithm (modified)

```
1:  function dijkstra(Node $start$, Node $destination$)
2:      Let $w_{i,j}$ be the cost of the arc $\epsilon_{i,j}$ (either $d_{i,j}$, $t_{i,j}$ or $c_{i,j}$).
3:      Let $C$ be the set of nodes $\nu_i$ yet to be processed.
4:      for all $\nu_i \in V$ do
5:          Mark $\nu_i$ as unknown
6:          if $\nu_i = start$ then
7:              $dist_{\nu_i} \leftarrow 0$
8:          else
9:              $dist_{\nu_i} \leftarrow \infty$
10:         end if
11:         $path_{\nu_i} \leftarrow null$
12:         $C \leftarrow C \cup \{\nu_i\}$
13:     end for
14:     while $C \neq \emptyset$ do
15:         Node $\nu_i \leftarrow$ extractNodeWithLeastDistance($C$)
16:         Mark $\nu_i$ as known
17:         if $\nu_i = destination$ then
18:             return $\nu_i$
19:         end if
20:         for all $\nu_j$ adjacent to $\nu_i$ do
21:             if $\nu_j$ is unknown then
22:                 if $dist_{\nu_i} + w_{i,j} < dist_{\nu_j}$ then
23:                     $dist_{\nu_j} \leftarrow dist_{\nu_i} + w_{i,j}$
24:                     $path_{\nu_j} \leftarrow \nu_i$
25:                     decreaseKey($C, \nu_j$)
26:                 end if
27:             end if
28:         end for
29:     end while
            return failure
30: end function
```

A. Time and Space Efficiency

Dijkstra's algorithm keeps a set of all nodes yet to be processed. The size of this set is obviously linear on the number of nodes in the network, $|V|$. The algorithm does keep other local variables, but their size is constant with respect to both $|V|$ and $|E|$. The space efficiency of Dijkstra's algorithm is thus $O(|V|)$.

The algorithm is usually implemented using a binary min-heap to sort the nodes yet to be processed by the least distance from the start node to each of those nodes. The **insertion** of a new element in a binary heap can be implemented in $O(\log(N))$ time, N being the number of elements in the heap. The **decrease key** operation decreases the key of an element in the heap after it has been updated. This operation assumes that the new key value is less or equal than the current one, and also runs in $O(\log(N))$ time. Selecting the minimum element in the heap is done in constant time, $O(1)$, and extracting it from the heap takes $O(\log(N))$ time[1].

The **for** loop in lines 4 to 13 executes $|V|$ times, as it goes through all nodes in the network. Each node is inserted in set C, which is a min-heap. Thus, insertion time is $O(\log(|V|))$, and, consequently, the loop runs in $O(|V| \cdot \log(|V|))$ time. The **while** loop in lines 14 to 29 is executed at most once for each node in set C, which has size $|V|$, as no node may be inserted twice in the heap. As mentioned above, the next node to be processed, which is the node at the least distance from the start node, is selected and extracted in $O(\log(|V|))$ time. For each node ν_i being processed, all of the arcs leaving it are explored in the inner **for** loop in lines 20 to 28. For each arc being explored, the **decrease key** operation in line 25 may or may not be executed. In the worst case, it is always executed, and the body of the **for** loop takes $O(\log(|V|))$ time. One might be tempted to say that the inner **for** loop is executed at most $|V| \cdot |E|$ times, as all of the arcs leaving a node are explored, and, in the worst case, the process is repeated for each node. Note, however, that each arc is explored at most once, which is when its source node is extracted from the heap, if that event ever happens. Therefore, the loop is executed, in fact, at most $|E|$ times. The running time of the algorithm is thus,

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

If $|E| > |V|$, the running time is:

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

Note that, should the network be dense ($|E| = O(|V|^2)$), then the original Dijkstra's algorithm, which doesn't mention any priority queue, is faster, as it runs in $O(|V|^2)$ time. In this paper, it is assumed that the networks are preferably sparse ($|E| = O(|V|)$).

B. Fibonacci Heaps

In the implementation of Dijkstra's algorithm which was presented in this section, the **decrease key** operation happens more frequently than the **extract minimum** operation, if we assume that $|E| > |V|$, as is usually verified. In cases like this, using a Fibonacci heap instead of a binary heap might lead to better performance[1]. Fibonacci min-heaps implement the **insertion** and **decrease key** operations in constant amortized time, making them an interesting data structure to consider when implementing Dijkstra's algorithm. If we follow this approach, the initial **for** loop in lines 4 to 13 can be performed in $O(|V|)$ time, as the amortized running time of the **insertion** operation is $O(1)$. The inner **for** loop in lines 20 to 28 is now performed in $O(|E|)$ time, as the **decrease key** operation is also executed in constant amortized time. The running time of the algorithm using Fibonacci heaps is thus,

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

If the network is sparse, i.e., if $|E| = O(|V|)$, then the running time of the algorithm is

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

If it verifies that $|E| > |V|$, this is an improvement over the running time which can be achieved with binary heaps. From a theoretical point of view, Fibonacci heaps are more desirable than binary heaps when the number of **extract minimum** and **delete** operations is small compared to the number of **decrease key** operations. In practice, however, the programming complexity of Fibonacci heaps is usually a deterrent to their use in most applications[1]. Note also that the constant time of the **insertion** and **decrease key** operations is an amortized time, meaning that it only describes accurately how the algorithm will perform in the average case. The $O(\log(|V|))$ time for the same operations on a binary heap corresponds to a worst-case scenario. Although they are theoretically efficient, Fibonacci heaps also have some overhead, making it unclear whether they perform better in practice than binary heaps in Dijkstra's algorithm[6].

IV. The A* Algorithm

Dijkstra's algorithm is a single-source shortest path algorithm, which means that it finds the shortest path connecting a source node to all other nodes in the network, if there is such a path. In fact, we are interested only in finding the shortest path between two nodes in the network, so, although the version of Dijkstra's algorithm presented in this paper stops as soon as it marks the destination node as known, it seems like it performs extra work which is not used in the end. An alternative to Dijkstra's algorithm is the A* search algorithm. This is a goal-directed search algorithm which makes use of heuristics, and the main idea behind it is that a shortest path between a source node and a destination node should lead in the general direction of the destination[3]. If the algorithm expands nodes which obviously can't be in the shortest path between the source and destination nodes, it is wasting resources. On the other hand, if it overlooks nodes which might be in the shortest path between the two nodes, then it is no longer *admissible*, i.e., it is no longer guaranteed to find an optimal path between the nodes[4]. The algorithm makes use of an evaluation function, $f^*(\nu_i)$, which estimates the cost of an optimal path from the start node to the goal node, *constrained* to go through node ν_i. **Algorithm 3** describes the A* algorithm in pseudocode.

Algorithm 3 The A* Algorithm

```
1:  function A_STAR(Node $start$, Node $goal$)
2:      Let $w_{i,j}$ be the cost of the arc $\epsilon_{i,j}$.
3:      Let $A$ be the open set of nodes $\nu_i$ yet to be evaluated.
4:      Let $f^*(\nu_i)$ be an evaluation function of the cost of
        an optimal path from the start node to the goal node,
        constrained to go through node $\nu_i$.
5:      Let $f_{\nu_i}$ be the previous value of the evaluation function
        for node $\nu_i$ (f-score).
6:      for all $\nu_i \in V$ do
7:          $f_{\nu_i} \leftarrow \infty$
8:          $dist_{\nu_i} \leftarrow \infty$
9:          $path_{\nu_i} \leftarrow null$
10:     end for
11:     $dist_{start} \leftarrow 0$
12:     $f_{start} \leftarrow f^*(start)$
13:     $A \leftarrow A \cup \{start\}$
14:     while $A \neq \emptyset$ do
15:         Node $\nu_i \leftarrow$ extractNodeWithLeastFScore$(A)$
16:         Mark $\nu_i$ as closed
17:         if $\nu_i = goal$ then return $\nu_i$
18:         end if
19:         for all $\nu_j$ adjacent to $\nu_i$ do
20:             if $\nu_j$ is not marked as closed then
21:                 $dist_{\nu_j} \leftarrow dist_{\nu_i} + w_{i,j}$
22:                 $path_{\nu_j} \leftarrow \nu_i$
23:                 $f_{\nu_j} \leftarrow f^*(\nu_j)$
24:                 if $\nu_j \notin A$ then
25:                     $A \leftarrow A \cup \{\nu_j\}$
26:                 else
27:                     decreaseKey$(A, \nu_j)$
28:                 end if
29:             else
30:                 if $f^*(\nu_j) < f_{\nu_j}$ then
31:                     Unmark $\nu_j$ as closed
32:                     $dist_{\nu_j} \leftarrow dist_{\nu_i} + w_{i,j}$
33:                     $path_{\nu_j} \leftarrow \nu_i$
34:                     $f_{\nu_j} \leftarrow f^*(\nu_j)$
35:                     $A \leftarrow A \cup \{\nu_j\}$
36:                 end if
37:             end if
38:         end for
39:     end while
         return failure
40: end function
```

A. The Evaluation Function

Let $f(\nu_i)$ be the actual cost of an optimal path connecting a start node s to a certain goal node, and *constrained* to go through node ν_i. $f(s)$ is the actual cost of an *unconstrained* optimal path connecting the start node s to a certain goal node. Note that $f(\nu_i) = f(s)$ for every node ν_i on an optimal path between s and a goal node, and $f(\nu_j) > f(s)$ for every node ν_j not on an optimal path[4]. We can then define $f(\nu_i)$ as a sum of two parts:

$$f(\nu_i) = g(\nu_i) + h(\nu_i)$$

where $g(\nu_i)$ is the actual cost of an optimal path from the start node s to node ν_i and $h(\nu_i)$ is the actual cost of an optimal path from node ν_i to a goal node. An estimate of $f(\nu_i)$ may be used as the evaluation function of the cost of an optimal path from s to a goal node[4]. This evaluation function will be called $f^*(\nu_i)$, and can be defined as:

$$f^*(\nu_i) = g^*(\nu_i) + h^*(\nu_i)$$

where $g^*(\nu_i)$ is an estimate of $g(\nu_i)$. If we define $g^*(\nu_i)$ as the cost of the path between the start node s and node ν_i with the least cost found so far, it follows that $g^*(\nu_i) \geq g(\nu_i)$[4]. The estimate of $h(\nu_i)$ is the heuristic function $h^*(\nu_i)$. The authors of this paper have decided to use the A* algorithm to address the problems of finding the paths with the least travelled distance and the least travel time in the network, as there is little or no information which can be extracted from the problem domain in order to reasonably estimate the total number of changes between transportation lines on an optimal path from node ν_i to a goal node[2]. The admissibility and the optimality of the A* algorithm depend on the choice of this heuristic function, as discussed in the next subsections.

B. The admissibility of the A* algorithm

It can be proved that, if

$$h^*(\nu_i) \leq h(\nu_i), \forall \nu_i \in V \tag{1}$$

i.e., if $h^*(\nu_i)$ is always a lower bound of $h(\nu_i)$, then the A* algorithm is admissible, i.e., it is guaranteed to find an optimal path from the start node to a goal node. Let us define two heuristic functions, one for the application of the A* algorithm for finding the path with the least travelled distance, and another for using the algorithm for finding the path with the least travel time.

Let $d^*(\nu_i)$ be defined as the Euclidean distance, i.e., the straight-line distance, between node ν_i and a goal node of the start node s, and let $d(\nu_i)$ be the actual distance between those two nodes on an optimal path. By definition, the Euclidean distance between two points in a metric space is the shortest possible distance of any path connecting them. Then, it is true that $d^*(\nu_i) \leq d(\nu_i), \forall \nu_i \in V$, as $d^*(\nu_i)$ never overestimates $d(\nu_i)$. Therefore, condition **(1)** is satisfied, and the admissibility of the A* algorithm is guaranteed when such a heuristic is used.

Let $t^*(\nu_i)$ be an estimate of the travel time between node ν_i and a goal node of s on an optimal path, which will be called $t(\nu_i)$. If $t^*(\nu_i)$ is estimated based on the Euclidean distance between those two nodes, $d^*(\nu_i)$, and on the *average speed of the fastest road* in the network, denoted by $\bar{v}_{max}$, then $t^*(\nu_i) = \frac{d^*(\nu_i)}{\bar{v}_{max}}$. If $\bar{v}(\nu_i)$ is the average speed of the optimal path in question, the actual travel time is given by $t(\nu_i) = \frac{d(\nu_i)}{\bar{v}(\nu_i)}$. Since $\bar{v}_{max} \geq \bar{v}(\nu_i), \forall \nu_i \in V$, and $d^*(\nu_i) \leq d(\nu_i), \forall \nu_i \in V$, as discussed in the previous paragraph, then $t^*(\nu_i) \leq t(\nu_i), \forall \nu_i \in V$, and condition **(1)** is verified. The admissibility of the A* algorithm is thus also guaranteed in this case.

C. The optimality of the A* algorithm

As referred in the previous subsection, if $h^*(\nu_i)$ is any lower bound on $h(\nu_i)$, then A* is admissible. A possible lower bound could be $h^*(\nu_i) = 0, \forall \nu_i \in V$, as there are no negative arc costs in the network. This heuristic is equivalent to assuming that any open node ν_i might be arbitrarily close to a goal node. If a higher lower bound of $h(\nu_i)$ is used for $h^*(\nu_i)$, then A* is still admissible, but fewer nodes will generally be expanded[4]. We start by making the following consistency assumption:

$$h(\nu_i, \nu_j) + h^*(\nu_j) \geq h^*(\nu_i) \tag{2}$$

This assumption means that any estimate $h^*(\nu_i)$, calculated from data provided by the situation which node ν_i represents, would not be improved by using the same data from situations represented by other nodes[4]. We begin by proving that both heuristics $d^*(\nu_i)$ and $t^*(\nu_i)$, defined in the previous subsection, verify this assumption.

Let $d(\nu_i, \nu_j)$ be the Euclidean distance between the nodes ν_i and ν_j in the network. Since, as described in the previous subsection, $d^*(\nu_i)$ is the Euclidean distance between node ν_i and a certain goal node of ν_i, and thus the shortest, it is true that

$$d(\nu_i, \nu_j) + d^*(\nu_j) \geq d^*(\nu_i) \tag{3}$$

Note that $h(\nu_i, \nu_j)$ is the actual cost of an optimal path from node ν_i to node ν_j, and therefore $h(\nu_i, \nu_j) \geq d(\nu_i, \nu_j)$. It is then also true that

$$h(\nu_i, \nu_j) + d^*(\nu_j) \geq d^*(\nu_i) \tag{4}$$

The consistency assumption **(2)** is therefore verified for $d^*(\nu_i)$.

Let $t(\nu_i, \nu_j)$ be the travel time between the nodes ν_i and ν_j based on the Euclidean distance between those nodes and on the average speed of the fastest road in the network, $t(\nu_i, \nu_j) = \frac{d(\nu_i, \nu_j)}{\bar{v}_{max}}$. Since, as described in the previous subsection, $t^*(\nu_k) = \frac{d^*(\nu_k)}{\bar{v}_{max}}, \forall \nu_k \in V$, dividing inequation **(3)** by $\bar{v}_{max}$ results in:

$$\frac{d(\nu_i, \nu_j)}{\bar{v}_{max}} + \frac{d^*(\nu_j)}{\bar{v}_{max}} \geq \frac{d^*(\nu_i)}{\bar{v}_{max}} \Leftrightarrow t(\nu_i, \nu_j) + t^*(\nu_j) \geq t^*(\nu_i) \tag{5}$$

Since $h(\nu_i, \nu_j)$ is the actual cost of an optimal path from node ν_i to node ν_j, and because $t(\nu_i, \nu_j)$ is an estimate of $h(\nu_i, \nu_j)$ based on the Euclidean distance between those nodes and on the greatest average speed of all arcs in the network, then $h(\nu_i, \nu_j) \geq t(\nu_i, \nu_j)$, and thus,

[2]Actually, the heuristic function $h^*(\nu_i) = 0$ could be used for every node ν_i in the network. This heuristic satisfies both the admissibility constraint, and the consistency assumption, as should be made clear by the next two subsections. However, as Hart and Nilsson discuss, such a heuristic would lead to the expansion of more nodes. In the opinion of the authors of this paper, this heuristic is not of academic interest, and would most likely not lead to an improvement over Dijkstra's algorithm.

$$h(\nu_i, \nu_j) + t^*(\nu_j) \geq t^*(\nu_i) \tag{6}$$

The consistency assumption **(2)** is verified for $t^*(\nu_i)$.

It can be proved that, if a heuristic satisfies the consistency assumption, then the A* algorithm never reopens a closed node, i.e., a node is never added again to the open set A, defined in **Algorithm 3**, after it has been extracted from it. This conclusion will be useful in our discussion of the complexity of A*. Under the same circumstances, as far as the heuristic function is concerned, it can also be shown that A* is optimal, in the sense that no other admissible algorithms expand fewer nodes, as long as they use no more information from the problem domain than A* does[4].

D. Time and Space Complexity of the A algorithm*

The complexity of A* clearly depends on the choice of the heuristic function. Not only might the calculation of the heuristic function contribute to the global complexity of the algorithm, but certain properties of that function may also have an influence on how the algorithm behaves, as explained in the previous subsection. In this subsection, we will estimate the time and space complexity of A* in a worst-case scenario.

A* keeps a set A of open nodes, usually a min-heap, from which it selects the node with the least value of the evaluation function. There can be at most $|V|$ nodes in the set, as no node appears twice in the set at any given time. The space complexity of every other operation is constant with respect to $|V|$ and $|E|$, including the calculation of the heuristic function, $d^*(\nu_i)$ or $t^*(\nu_i)$. Therefore, the space complexity of the A* algorithm is $O(|V|)$.

Let us first assume that a binary heap is used as the open set A of all nodes ν_i yet to be evaluated. The first **for** loop in lines 6 to 10 executes $|V|$ times. Since its body executes in constant time, the loop runs in $O(|V|)$ time. The **insertion** operation in line 13 runs in constant time, as the open set A was empty at the time. The algorithm as defined by **Algorithm 3** implies that a certain node ν_i may be added again to the open set after being marked as closed, which means that the same node might be expanded more than once. However, the occurrence of such repetition depends on the choice of the heuristic function. Recall from the previous subsection that, if the heuristic function verifies the consistency assumption, then the algorithm never reopens a closed node. It was also proved that both heuristics $d^*(\nu_i)$ and $t^*(\nu_i)$ verify the consistency assumption, which means that using either one of them in the algorithm guarantees that the **while** loop executes at most $|V|$ times, once for each node ν_i in the network. The **extract minimum** operation in line 15 runs in $O(\log(|V|))$, as the binary heap stores at most $|V|$ nodes at a time. The inner **for** loop in lines 19 to 38 is executed once for each node ν_j adjacent to the current node ν_i. As explained in the previous section, when analysing the complexity of Dijkstra's algorithm, the body of the inner **for** loop is executed at most $|E|$ times. The nested **if** construct means that either one of the **insert** or **decrease key** operations is performed, both of which running at $O(\log(|V|))$ time. The time complexity of the algorithm for a worst-case scenario is thus,

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

If $|E| = O(|V|)$ and $|E| > |V|$, then the complexity of the algorithm is:

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

If a Fibonacci heap is used to store the nodes in the open set, then the time complexity of both the **insert** and the **decrease key** operations is $O(1)$, and the complexity of the algorithm is thus,

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

If the graph is sparse ($|E| = O(|V|)$), then the complexity may be written as:

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

The complexities of the A* algorithm, using either one of the heuristics $d^*(\nu_i)$ and $t^*(\nu_i)$, and Dijkstra's algorithm for a worst-case scenario are the same. We began this discussion by claiming that Dijkstra's algorithm might perform extra work when searching for an optimal path between a pair of nodes in the network. However, we have reached no theoretical evidence that A* performs better than Dijkstra's algorithm. The performance of A* depends on the number of nodes which are expanded before the algorithm terminates, and such a number depends on the choice of the heuristic function, $h^*(\nu_i)$. As Hart and Nilsson point out, it is generally true that the higher the lower bound of $h(\nu_i)$ is, the fewer the nodes expanded by A*. In our case, $d^*(\nu_i)$ and $t^*(\nu_i)$ are the highest possible lower bounds of both $d(\nu_i)$ and $t(\nu_i)$, as the Euclidean distance between two nodes is the least possible distance between them, and a travel time based on the Euclidean distance between two nodes and on the greatest average speed is the least possible travel time between those two nodes. A* is therefore expected to perform well, although it is still not clear whether it performs better than Dijkstra's algorithm.

V. The Performance of the Algorithms Compared

In this section, the performance of the algorithms will be compared, so as to complement the theoretical evaluation of their time complexity. **Figure 2** and **Figure 3** show how the running time of the four algorithms, each of them executed one thousand times, grows as $|V|$ and $|E|$ grow, respectively. Each dot in the sequence represents the same network in both charts. Note that the behaviour of the curves in both cases is very similar, as the networks chosen to test the performance of the algorithms are sparse, i.e., $|E| = O(|V|)$. The implementation of a binary heap from the C++ STL, and the implementation of a Fibonacci heap from the Boost library were used for test purposes.

A first examination of both charts is surprising, although it confirms our suspicions about the performance of Fibonacci heaps in real-world applications. Although they appear quite efficient from a theoretical perspective, it looks like Fibonacci

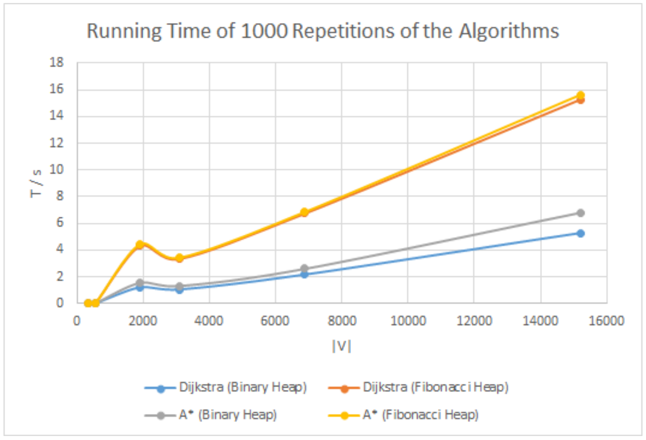

Fig. 2. Time complexity of the algorithms as $|V|$ grows.

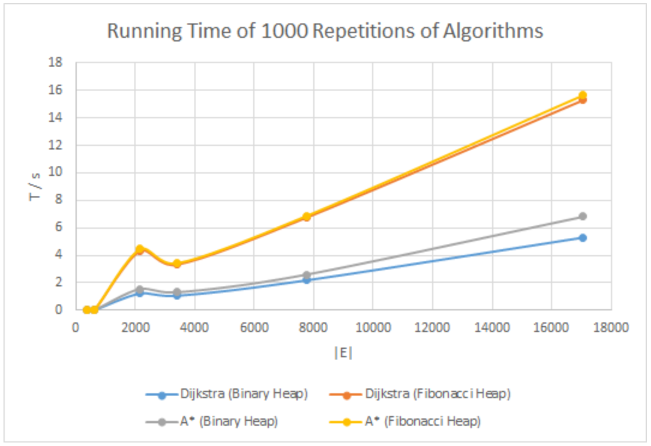

Fig. 3. Time complexity of the algorithms as $|E|$ grows.

heaps perform worse than binary heaps, in practice. In fact, using a binary heap instead of a Fibonacci heap produced arguably better results. This may due to the fact that Fibonacci heaps are much more complex data structures than binary heaps, and that their use causes some extra overhead, as discussed in previous sections.

As far as the algorithms are concerned, Dijkstra's algorithm performed slightly better than the A* algorithm as $|V|$ and $|E|$ grow. However, the running times are very close, and the results are thus inconclusive. After proving that the theoretical time bounds for both algorithms are the same, there is no empirical evidence that either A* or Dijkstra's algorithm performs better. Note also that the performance of the algorithms is related to how many nodes are expanded before finding the goal node, which depends greatly on each instance of the problem. Since Dijkstra's algorithm is more intuitive and easier to use in real-world applications, it is the judgement of the authors of this paper that this should be the usual choice for most situations. If, however, a heuristic leading to a better performance by A* is found, then this algorithm should be used instead. Whatever the choice, this should be as most informed a decision as possible.

VI. Conclusions and Future Work

The aim of this paper was to discuss a possible approach to the problem of planning a trip in an intermodal transportation network, along an optimal path connecting two points in such a network. A path is considered to be optimal if it meets a certain objective function. Three possible instances of the objective function were discussed in previous sections. This problem was reduced to a shortest path problem. When the subject involves a shortest path problem, the renowned Dijkstra's algorithm may never be overlooked, as it is one of the most efficient single-source shortest path algorithms. The motivation behind the research on the A* algorithm was the fact that Dijkstra's algorithm finds the shortest path from a source node to all the other nodes, whereas all that is needed for this problem is to find the shortest path between a pair of nodes in the network. Since it does seem that Dijkstra's algorithm performs extra work while carrying out this task, the A* algorithm was tested to see whether it could speed up the process by using heuristics capable of leading straight towards the goal node. No theoretical or empirical evidence was found that A* performs better than Dijkstra's algorithm. In fact, the latter performed slightly better in practice, although the results are inconclusive as to which algorithm has the least running time.

Both of these algorithms made use of a priority queue, usually implemented with a binary heap. In order to speed up the algorithms, the authors of this paper looked into a more advanced data structure which can be used to implement a priority queue, the Fibonacci heap. Fibonacci heaps are more complex than binary heaps, but have better theoretical time bounds. Two versions of each algorithm were tested, one using a binary heap, and the other using a Fibonacci heap. The results of the running time of both versions were surprising, given that Fibonacci heaps were expected to perform better than binary heaps. Instead, binary heaps were able to outperform Fibonacci heaps in the implementations of both algorithms[3].

Having reflected upon possible solutions to the problem of finding an optimal path in an intermodal transportation network, the next step would be to implement a real-world user-oriented application, so as to investigate how the algorithms would behave with real public transportation networks of large cities. The authors also hope that this paper will lead to further research on subjects related to large transportation networks, such as the problem of route assignment and other fundamental issues which smart cities' mobility agenda nowadays must address.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[2] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *NUMERISCHE MATHEMATIK* 1.1 (1959), pp. 269–271.

[3]Note that this result could depend on the implementations of binary and Fibonacci heaps which are being tested. As said before, the binary heap implementation of the C++ STL, and the Fibonacci heap implementation of the Boost library were used.

[3] Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, eds. *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Berlin, Heidelberg: Springer-Verlag, 2009. ISBN: 978-3-642-02093-3.

[4] N. J. Nilsson P. E. Hart and B. Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Transactions on Systems, Science, and Cybernetics* SSC-4.2 (1968), pp. 100–107.

[5] Yosef Sheffi. *Urban Transportation Networks: Equilibrium Analysis With Mathematical Programming Methods*. Prentice Hall, 1985. ISBN: 0139397299.

[6] Mark A. Weiss. *Data Structures & Algorithm Analysis in C++*. 4th. Pearson Education, 2014. ISBN: 013284737X, 9780132847377.