Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Computing all Paths between two Locations with a Maximal Time Budget

Christine Lunger

supervised by

Prof. Johann Gamper

October 2015

// Abstract

In my thesis I am focusing on finding all paths in a unimodal spatial network from a source node to a target node, where the total time for a single path is within a given time budget. Cycles are allowed as long as no edge is traversed twice in the same direction.

This problem can be used for tour planning. If all paths are known, they can be intersected with a relation that contains the points of interest to find the best route. Once computed, the set of paths can be used multiple times.

The algorithms I studied to solve this problem are Breadth-First Search (BFS), Depth-First Search (DFS), and A*. Additionally, I developed the BFS* and the DFS* algorithm, where BFS* and DFS* are versions of BFS and DFS that use a heuristic function to reduce the search space. I adapted the first three algorithms and developed the last two to improve the performance. The algorithms were implemented in Java and I made some experiments to compare their runtimes for the pedestrian network of Bozen.

When comparing the runtimes of the five algorithms, BFS, DFS, A*, BFS*, and DFS*, it becomes clear that BFS is the slowest one. The runtime of DFS lies between BFS and the other three algorithms. The performance of A*, BFS* and DFS* is better, because they use some heuristic function, which is the Euclidean distance. The Euclidean distance helps to keep the search space small, but still it becomes clear from these results that there is a need for more efficient algorithms, if large time budgets should be used.

Contents

1 Introduction **3**

1.1 Motivation 3

1.2 Problem Description and Contribution 4

1.3 Organization of the Thesis 4

2 Preliminaries **5**

3 Algorithms **6**

3.1 Breadth-First Search 6

3.2 Depth-First Search 9

3.3 A* Algorithm 11

3.4 BFS* 14

3.5 DFS* 17

4 Experimental Evaluation **19**

4.1 Setup and Data 20

4.2 Runtime Experiments 21

5 Related Work **26**

5.1 Shortest Path Algorithms 26

5.2 Isochrones 27

6 Conclusion and Future Work **28**

List of Figures

1 All possible paths 3
2 Example of a graph 6
3 First steps in BFS 9
4 First steps in DFS 12
5 First steps in A* 14
6 First steps in BFS* 17
7 First steps in DFS* 19
8 Runtimes Q0 21
9 Runtimes Q1 22
10 Runtimes Q2 23
11 Runtimes Q3 24
12 Runtimes Q4 25
13 Runtimes Q5 25

1 Introduction

1.1 Motivation

Finding the right path has always been a problem. Already in the Stone Age travelling and finding the right way had a deep impact on live. Nowadays we have different reasons to travel and a lot more information available, but the problem is still the same: finding the best path.

There are now many tools to help us to find the best path, such as Google Maps or TomTom. These and other companies that work on routing-related projects compute often in addition to the shortest path some alternatives that should be short and significantly different from the shortest path and other alternatives.

Example 1. Consider the case you are in the Computer Science Faculty Building of the Free University of Bolzano-Bozen and you want to make a break of twenty minutes. You just want to go out for a few minutes and return in time. As you do this more often and do not want to take always the same route, you calculate all the paths and before starting you decide which path to take. The result you want to get is something like in Figure 1. The black lines represent the pedestrian network of Bozen, the red lines mark the streets that are part of a path for which at most 20 minutes are needed with an average walking speed of 0.5m/sec. The star is the source and the target node, which are the same in this case.

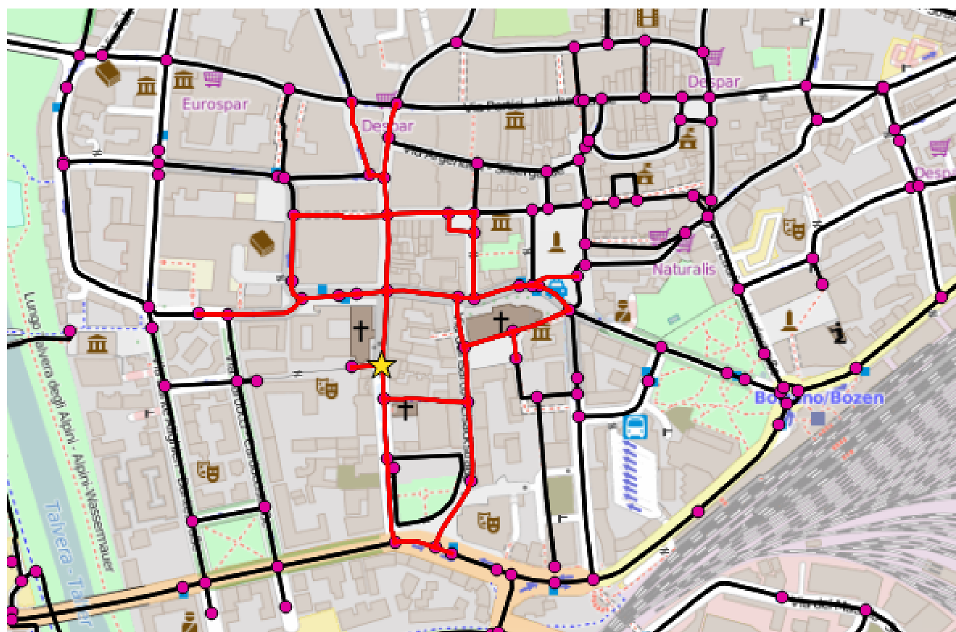

Figure 1: All possible paths

1.2 Problem Description and Contribution

The problem I am focusing on is finding all paths in a static unimodal spatial network from a source node to a target node, where the total time for a single path is less than or equal to the time in the time budget. Source and target do not need to be different. In addition, the walking speed is considered for the computation of the cost, as the cost is the time needed to traverse an edge. As the network is a pedestrian one, both directions between two nodes are possible. It is also possible to have cycles in a path, but an edge can just be traversed once in both directions. In other words, one can traverse the edge in one direction and return from the other one. The target node can be several times on the path, but a path has always to end with the target node, i.e., a path to the target can go beyond the target and then return using the same edges in reverse order and direction or on another path.

The result is the collection of all paths from source to target, and not the set of all edges that are part of a path. These solutions are different, because the first one is a collection of paths and the second one a collection of edges and not all combinations of the edges that are part of a path respect the time constraint and build a path. The network I use has discrete space, continuous time mode, i.e. the user can traverse the edges whenever he wants, but can just start and stop at a node, and not in between.

Two basic algorithms to solve the problem of finding all paths are Depth First Search and Breadth First Search. They both expand the nodes of the underlying network until they run out of nodes or meet some stop condition. The A* Algorithm uses additionally some heuristic function to reduce the search space.

The main contributions are:

- Adaption of Breadth-First Search (BFS), Depth-First Search (DFS), and A* for the problem of computing all paths between two nodes with a given maximal time budget.
- Development of BFS* and DFS* as versions of BFS and DFS that use a heuristic function.
- Experimental evaluation of the algorithms in terms of runtime.

1.3 Organization of the Thesis

This thesis in organized as follows: In Section 2, I will introduce some preliminary concepts, like pedestrian network, path in a spatial network, and Euclidean distance. Then, I will explain the algorithms I found to solve the problem, which are BFS, DFS, A*, BFS*, and DFS*, in Section 3. In Section 4 I will compare the algorithms. For the comparisons of the runtime, I use the pedestrian network of Bozen. In the end I will talk about some related work in Section 5.

2 Preliminaries

In this section, I provide a formal definition of a pedestrian network and the Euclidean distance. As I decided to focus on paths for pedestrian networks, I need to define it here.

Definition 2.1. (Pedestrian Network) A pedestrian network is a directed graph $PN = (V, E, \lambda)$, where V is a set of vertices (nodes) and $E \subseteq V \times V$ is a set of edges. $\lambda : E \mapsto \mathbb{R}^+$ is a function that assigns to each *edge* $(u, v) \in E$ a positive real number that represents the length of the edge.

Nodes represent the beginning or the end of a street segment or a crossroad, edges represent street segments, and λ models the length of a street segment. Through walking in a pedestrian network is typically allowed in both directions, it is modelled as a directed graph and represents a street segment as a pair of edges of the same length but opposite directions.

Definition 2.2. (Path, Path Cost) A path from a source node v_s to a target node v_t is defined as a sequence of connected edges $p\,(v_s, v_t) = \langle x_1, \ldots x_k \rangle$, where $x_i = (v_i, v_{i+1})$. The path cost is the sum of the costs of the edges $\gamma(\langle x_1, \ldots x_k \rangle) = \sum_{n=1}^{k} \lambda\,(x_n)$.

Definition 2.3. (Adjacent Edge) An edge $e \in E$ is adjacent to a node $v \in V$, if $e = (v, x)$, where $x \in V$.

Definition 2.4. (Cycle) A cycle is a path where both endpoints coincide. A cycle is thus a sequence of connected edges $c\,(v_s, v_t) = \langle x_1, \ldots x_k \rangle$, where $x_i = (v_i, v_{i+1})$ and $v_s = v_t$.

There is also a need to define the Euclidean distance and the Euclidean Lower-Bound Property, because they are used for the A* Algorithm and I used them to improve the performance of Depth-First Search and Breadth-First Search. The definition of the network distance is needed for the definition of the Euclidean Lower-Bound Property, as it says that the Euclidean distance lower bounds the network distance.

Definition 2.5. (Network distance, Shortest Path) The network distance $d_N\,(v_s, v_t)$ from a source node v_s to a target node v_t is defined as the minimum cost of any path from v_s to v_t, if such a path exists, and ∞ otherwise. The path with the minimum cost is the shortest path.

Definition 2.6. (Euclidean Distance) The Euclidean distance between two points $p = (x_p, y_p)$ and $q = (x_q, y_q)$ in the plane is defined as follows: $d_e\,(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$, where p and q are two points in a rectangular xy Cartesian coordinate system. The distance is also the length of the straight segment, having p and q as its endpoints.

Definition 2.7. (Euclidean Lower-Bound Property) The Euclidean Lower-Bound Property states that for any two nodes v_i and v_j, the Euclidean distance, $d_e(v_i, v_j)$, lower bounds the network distance, $d_N(v_i, v_j)$, i.e. $d_e(v_i, v_j) \leq d_N(v_i, v_j)$.

The Euclidean Lower-Bound Property means that there cannot be a path from v_i to v_j that is shorter than the Euclidean distance, as there cannot be a shorter path than the straight connection of the two nodes.

Definition 2.8. (Set of all Paths between two Locations with a Maximal Time Budget) The set of all paths between two locations v_s and v_t with a maximal time budget t is defined as $AP = \{p | \gamma(p(v_s, v_t)) \leq t\}$

The set of all paths between two locations with a maximal time budget is the set of all path AP such that the cost for the paths that are in the set is less than the time budget.

3 Algorithms

In this section, I am going to explain the algorithms in detail. All the algorithms were implemented in Java. I will use the graph in Figure 2 to illustrate how the different algorithms work. The numbers near the arrows represent the length of the edge, which is in both directions the same. The time budget $t = 11$, the walking speed $ws = 1$, A is the source node and E is the target node.

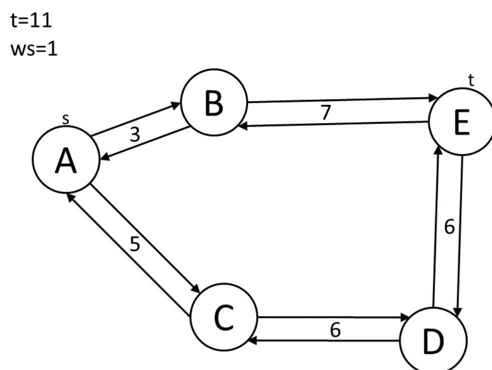

Figure 2: Example of a graph

3.1 Breadth-First Search

Breath-First Search (BFS) visits first all the nodes in a graph that are k edges away from the source and then it goes on with the nodes that are $k+1$

edges away. It continuous to do this until no more nodes are reachable. Breath-First Search uses a queue to store the nodes that are still to be expanded. Therefore, it may need a non-trivial storage space. [6]

Algorithm 1 shows the Breadth-First Search. The algorithm takes the source, the target, the time budged in seconds, and the walking speed in meters per second as parameters and returns a collection of paths.

My implementation of the Breadth-First Search starts by enqueueing all the adjacent edges of the source node. Then it goes through all the elements in the queue. First, it increases the current cost by the length of the edge divided by the walking speed and adds the current edge to the current path. If the end node of the current edge is the target, then a new path was found. If the cost for that path, which is the current cost, is less than or equal to the time budget, then the new path is added to the collection of the found ones. Afterwards all the adjacent edges of the end node of the current edge are enqueued, if the current cost is less than the time budget, which means that there could be enough time to traverse another edge. Another condition for enqueueing an edge is that it is not already contained in the current path. The method returns when there are no more elements in the queue.

Example 2. Figure 3 illustrates a few steps of the BFS algorithm, using the network in Figure 2. A is the start node and E is the target node. The time budget is 11 and the average walking speed is for simplicity 1.

- First, node A is expanded (Fig. 3a). The cost to reach B is 3 and to reach C from A is 5. The costs are shown between the brackets. B and C are enqueued.
- Then, B is expanded (Fig. 3b). The nodes next to B are A and E. To reach A from B the cost is 3, so the total cost for the path $\langle (A,B), (B,A) \rangle$ is 6. To reach E from B the edge of length 7 has to be traversed. Therefore, the cost to reach E from A is 10. A and E are enqueued.
- C is the next node in the queue. It is expanded next (Fig. 3c). The adjacent nodes to C are A and D, and the costs to reach them from A are 10 and 11 respectively. Again, the nodes are enqueued.
- Figure 3d shows the next step. A is the next node that is expanded. B and C are the adjacent edges. As the path to A already contains the edge $e = (A, B)$, node B is not enqueued anymore. C is the only node that is enqueued.
- The next node in the queue is E (Fig. 3e). E is also the target node. Therefore, a new path was found: $p = \langle (A,B), (B,E) \rangle$. The cost for the path is 10. As 10 is still below the time limit, B and D are enqueued.

Algorithm 1 Breadth-First Search

```
1: procedure BFS(source, target, timeBudget, walkingSpeed)
2:     queue ← ∅
3:     foundPaths ← ∅
4:     adjacentEdges ← source.adjacentEdges
5:     for all edge ∈ adjacentEdges do
6:         queue.enqueue (edge, ∅, 0)
7:     end for
8:     while queue ≠ ∅ do
9:         qo := queue.dequeue ()
10:        currentEdge := qo.currentEdge
11:        currentPath := qo.currentPath
12:        cost := qo.cost + currentEdge.length/walkingSpeed
13:        currentPath.add (currentEdge)
14:        if currentEdge.endNode = target then
15:            startNode := currentPath.firstEdge.startNode
16:            endNode := currentEdge.endNode
17:            if cost ≤ timeBudget then
18:                newPath := Path (startNode, endNode, cost, currentPath)
19:                foundPaths.add (newPath)
20:            end if
21:        end if
22:        adjacentEdges := currentEdge.endNode.adjacentEdges
23:        if cost < timeBudget then
24:            for edge ∈ adjacentEdges do
25:                if edge ∉ currentPath then
26:                    queue.enqueue (edge, currentPath, cost)
27:                end if
28:            end for
29:        end if
30:    end while
31:    return foundPaths
32: end procedure
```

- Afterwards the next node is expanded, which is A (Fig. 3f). The edge $e = (A, C)$ is already contained in the current path. Therefore, only B is enqueued.
- The algorithm continues until the queue is empty.

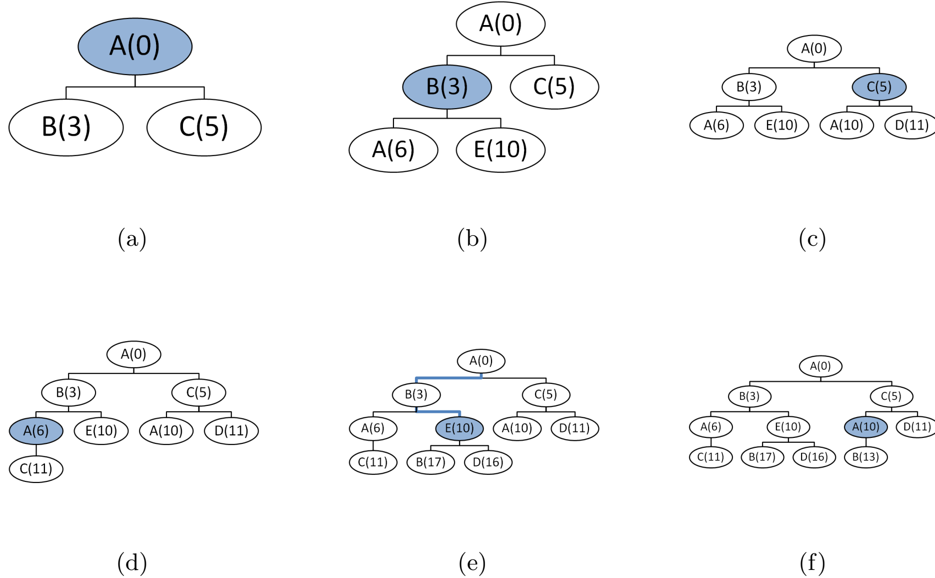

Figure 3: First steps in BFS

3.2 Depth-First Search

Depth-First Search (DFS) follows a different approach from Breadth First Search. It solves the problem by making as much forward progress as possible and by assuming that the target is not far away. That means, randomly select a direction whenever one has to be chosen and go on in that direction, marking where you have come from. Whenever a dead end is reached or the only possible progress is by revisiting a state, go back until a branch is found that is not explored and continue in that direction. [6] Depth-First Search uses a stack to store the information needed.

Algorithm 2 shows the Depth-First Search. The algorithm takes the source, the target, the time budget in seconds, and the walking speed in meters per second as parameters and returns a collection of paths.

My implementation of the Depth-First Search starts by pushing all the adjacent edges of the source node onto the stack, together with the information about the current path, which is empty at the beginning, and the current cost, which is zero. I store also the information about the current path and the current cost, because each node can be visited more than once and so it is not possible to store the information in the node. After the initialization of the stack, it iterates though all the elements on the stack. In each iteration, it adds the current edge to the current path and the cost for traversing the current edge to the current cost. If the end node of the current edge is the target and the cost for the path is less than the given

maximal cost, the time budget, then a new path was found. The new path is added to the collection of the found ones. Then, if the current cost is less than or equal to the given maximal cost, the adjacent edges of the end node of the current edge are pushed onto the stack, if they are not already contained in the current path. The method returns when there are no more elements on the stack.

Algorithm 2 Depth-First Search

```
 1: procedure DFS(source, target, timeBudget, walkingSpeed)
 2:     stack ← ∅
 3:     foundPaths ← ∅
 4:     adjacentEdges ← source.adjacentEdges
 5:     for all edge ∈ adjacentEdges do
 6:         stack.push (edge, ∅, 0)
 7:     end for
 8:     while stack ≠ ∅ do
 9:         so := stack.pop ()
10:         currentEdge := so.currentEdge
11:         currentPath := so.currentPath
12:         cost := so.cost + currentEdge.length/walkingSpeed
13:         currentPath.add (currentEdge)
14:         if currentEdge.endNode = target then
15:             startNode := currentPath.firstEdge.startNode
16:             endNode := currentEdge.endNode
17:             if cost ≤ timeBudget then
18:                 newPath := Path (startNode, endNode, cost, currentPath)
19:                 foundPaths.add (newPath)
20:             end if
21:         end if
22:         adjacentEdges := currentEdge.endNode.adjacentEdges
23:         if cost < timeBudget then
24:             for edge ∈ adjacentEdges do
25:                 if edge ∉ currentPath then
26:                     stack.push (edge, currentPath, cost)
27:                 end if
28:             end for
29:         end if
30:     end while
31:     return foundPaths
32: end procedure
```

Example 3. Figure 4 illustrates a few steps of the DFS algorithm, using the network in Fig. 2. A is the source node and E the target. The time

budget is 11 and the average walking speed is for simplicity 1.

- First, node A is expanded (Fig. 4a), and C and B are pushed onto the stack. The distance to B is 3 and to C is 5, which is shown in the brackets.
- The next step is to expand B (Fig. 4b). The adjacent nodes to B are A and E. A is 3 away from B and E is 7 away, so the total cost to reach A from A through B is 6 and to reach E through B is 10.
- DFS then continues by expanding A (Fig. 4c). The only node to be added to the stack is in this case C, because the edge $e = (A, B)$ is already contained in the path to A.
- The next node that is extracted from the stack is C (Fig. 4d). The cost from C is already 11. Therefore, no path can be found to E that is shorter than 11 continuing the path $p = \langle (A, B), (B, A), (A, C) \rangle$.
- On figure 4e A and C are not shown anymore, because they cannot be part of a path. E is expanded next, as it is the next node on the stack. B and D are the adjacent nodes and have a distance of 17 and 16 to the source respectively. E is also the target node. The new path is $p = \langle (A, B), (B, E) \rangle$ and the cost for it is 10.
- The algorithm continues to look for a path and pops B from the stack (Fig. 4f). The cost for B is greater than the time limit. Therefore, the adjacent nodes are not pushed onto the stack.
- The algorithm continues like this until no more nodes are on the stack.

3.3 A* Algorithm

The A* Algorithm is an iterative, ordered search that maintains a set of states to explore in an attempt to reach the target. At each iteration, A* uses an evaluation function $f^*(n)$ to select a state n from the set of states to explore whose $f^*(n)$ has the smallest value. $f^*(n) = g^*(n) + h^*(n)$, where:

- $g^*(n)$ estimates shortest path from *source* to n
- $h^*(n)$ estimates shortest path from n to *target*
- $f^*(n)$ estimates shortest path from *source* to *target* through n

$g^*(n)$ can be computed on the fly by recording with each state its distance from the start. [6]

Algorithm 3 shows the A* Algorithm. The algorithm takes the source, the target, the time budget in seconds, and the walking speed in meters per second as parameters and returns a collection of paths.

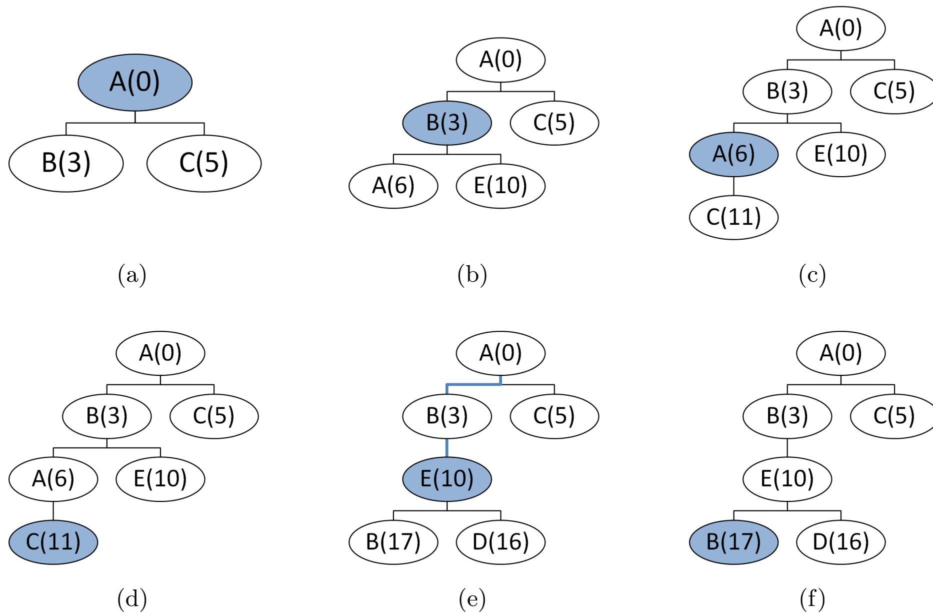

Figure 4: First steps in DFS

My implementation of A* starts by checking if the Euclidean distance from source to target is less than the time budget. If that is the case, it enqueues all the adjacent edges of the source node, together with the information about the path so far, which is empty, and the cost so far, which is zero as no edge was traversed so far. Then it goes one by iterating through the priority queue. In each iteration, it dequeues an element and adds the cost for traversing the current edge to the cost so far and the current edge to the current path. Afterwards it checks if the end node of the current edge is the target and the cost is within the time budget. If that is the case, the newly found path is added to the found paths, otherwise it just continues by checking if it can still go on, which means that the cost so far plus the Euclidean distance to the target is less than the time in the time budget. In that case, the adjacent edges to the end node of the current edge are inserted into the priority queue, if they are not already contained in the current path, together with the cost so far and the current path. The method returns when there are no more elements in the queue.

Example 4. Figure 5 illustrates a few steps of the A* algorithm, using the network in Figure 2. A is the source node and E is the target. The average walking speed is for simplicity 1 and the time budget is 11. The first number in the brackets of the path trees in Figure 5 is the value for $g^*(n)$, the cost so far, and the second number is the value for $h^*(n)$, the estimate for the cost to reach the target from n on.

Algorithm 3 A*

```
 1: procedure A*(source, target, timeBudget, walkingSpeed)
 2:     adjacentEdges ← ∅
 3:     priorityQueue ← ∅
 4:     foundPaths ← ∅
 5:     if distanceToTarget <= timeBudget then
 6:         adjacentEdges := source.adjacentEdges
 7:         for all edge ∈ adjacentEdges do
 8:             priorityQueue.enqueue (edge, ∅, 0)
 9:         end for
10:         while priorityQueue ≠ ∅ do
11:             qo := priorityQueue.dequeue ()
12:             currentEdge := qo.currentEdge
13:             currentPath := qo.currentPath
14:             cost := qo.cost + currentEdge.length/walkingSpeed
15:             currentPath.add (currentEdge)
16:             if currentEdge.endNode = target then
17:                 startNode := currentPath.firstEdge.startNode
18:                 endNode := currentEdge.endNode
19:                 if cost ≤ timeBudget then
20:                     newPath := Path (startNode, endNode, cost, currentPath)
21:                     foundPaths.add (newPath)
22:                 end if
23:             end if
24:             adjacentEdges := currentEdge.endNode.adjacentEdges
25:             if cost + distanceToTarget <= timeBudget then
26:                 for edge ∈ adjacentEdges do
27:                     if edge ∉ currentPath then
28:                         priorityQueue.enqueue (edge, currentPath, cost)
29:                     end if
30:                 end for
31:             end if
32:         end while
33:     end if
34:     return foundPaths
35: end procedure
```

- First, A is expanded (Fig. 5a). The value for $f^*(n)$, the estimated cost from source to target through A, is 9. B and C are enqueued.

- The next node that is expanded is B, because $f^*(B) = 10$ and $f^*(C) = 15$, and the node with the smallest value for $f^*(n)$ is chosen (Fig. 5b). A and E are the adjacent nodes to B.

- E is dequeued and expanded next (Fig. 5c). As E is the target, a new path was found: $p = \langle (A, B), (B, E) \rangle$. The algorithm enqueues the adjacent nodes B and D, because $f^*(E) = 10$ is less than the time budget 11.
- Now the node to dequeue is A. The value for $f^*(A)$ is greater than the time limit. Therefore, there cannot be found a path to E from this A on (Fig. 5d).
- The same is for C (Fig. 5e). The estimated cost is too high and therefore no adjacent node is enqueued.
- The only nodes in the queue are B and D. The value for $f^*(B) = 24$ and $f^*(D) = 22$. D has the smaller value and is considered next (Fig. 5f). Its value is higher than the allowed one, so no node is enqueued.
- The algorithm continues until the queue is empty.

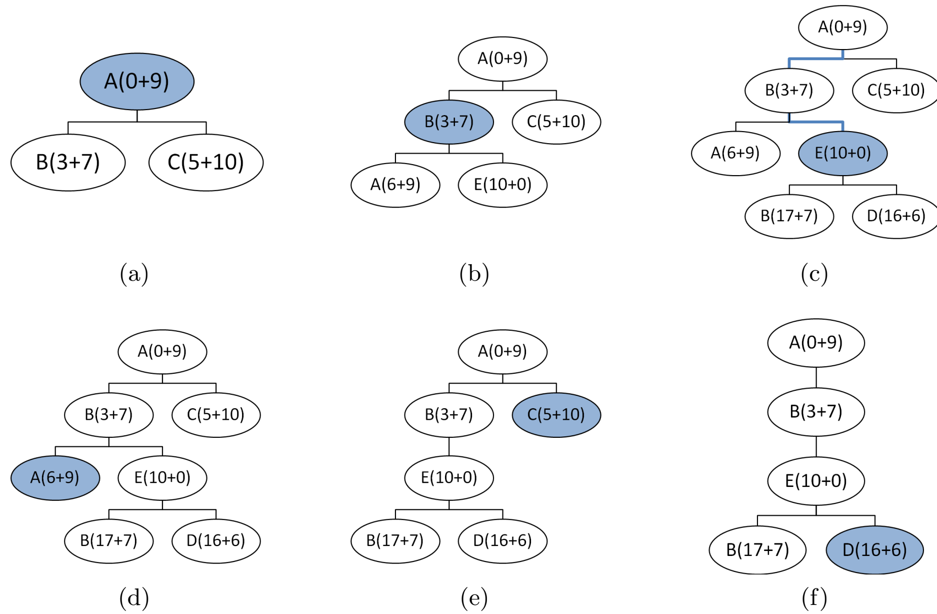

Figure 5: First steps in A*

3.4 BFS*

To optimize the Breadth-First Search I changed the condition for continuing the search. Only if the sum of the current cost and the Euclidean distance from the end node of the current edge to the target is less than or equal to

the time in the time budget, the adjacent edges are pushed onto the stack together with the information about the current path and the current cost. As of the Euclidean Lower-Bound Property, we can exclude that there can be found a path to the target, if the sum of the cost so far and the Euclidean distance to the target are greater than the time budget. I used the planar geometry, because the network spans only over a small area and the mistake is therefore minimal. The asterisk (*) next to the name of the algorithm indicates that some heuristic function is used to restrict the search space.

Algorithm 4 shows the BFS* Algorithm. The algorithm takes the source, the target, the time budget in seconds, and the walking speed in meters per second as parameters and returns a collection of paths.

The implementation of the BFS* is almost the same as the implementation of the BFS. The only difference lies in the condition for enqueueing an edge. In BFS the condition is that the cost so far has to be less than the time budget. BFS* includes also the estimated distance, i.e. the sum of the cost so far and the Euclidean distance to the target has to be within the time budget. This reduces the search space, because it forces the algorithm to stop going in one direction, when it gets out of range of the target.

Example 5. Figure 6 illustrates a few steps of the BFS* algorithm, using the network in Figure 2. A is the source node and E is the target. The average walking speed is for simplicity 1 and the time budget is 11. The first number in the brackets of the path trees in Figure 6 is the the cost so far, and the second number is the estimate for the cost to reach the target from the current node on.

- First, A is expanded (Fig. 6a). The estimated cost from A to E is 9. That is inside the time budget, so the adjacent nodes B and C are enqueued.
- Like BFS, BFS* follows the first in first out principle (FIFO), i.e. it selects first the nodes that are k edges away from the source and then the nodes that are $k+1$ edges away, because the nodes on the k-th level were enqueued first. Therefore, the node that is expanded next is B (Fig. 6b), followed by C (Fig. 6c). While the estimated cost to the target through B is still inside the time budget, it is too high to enqueue the adjacent nodes for C.
- The value for A, the next node in the queue, is like the value for C, too high (Fig. 6d). Thus, the adjacent nodes to A are not enqueued.
- Figure 6e shows the next step. The node E is expanded and a new path was found (Fig. 6e). The new path is $p = \langle (A, B), (B, E) \rangle$ and the cost for it is 10. The adjacent nodes to E are B and D.
- B is enqueued before D and is therefore the next node to be considered (Fig. 6f). The estimated cost for the path through B is 24. That is

Algorithm 4 BFS*

```
 1: procedure BFS*(source, target, timeBudget, walkingSpeed)
 2:     adjacentEdges ← ∅
 3:     queue ← ∅
 4:     foundPaths ← ∅
 5:     if distanceToTarget <= timeBudget then
 6:         adjacentEdges := source.adjacentEdges
 7:         for all edge ∈ adjacentEdges do
 8:             queue.enqueue (edge, ∅, 0)
 9:         end for
10:         while queue ≠ ∅ do
11:             qo := queue.dequeue ()
12:             currentEdge := qo.currentEdge
13:             currentPath := qo.currentPath
14:             cost := qo.cost + currentEdge.length/walkingSpeed
15:             currentPath.add (currentEdge)
16:             if currentEdge.endNode = target then
17:                 startNode := currentPath.firstEdge.startNode
18:                 endNode := currentEdge.endNode
19:                 if cost ≤ timeBudget then
20:                     newPath := Path (startNode, endNode, cost, currentPath)
21:                     foundPaths.add (newPath)
22:                 end if
23:             end if
24:             adjacentEdges := currentEdge.endNode.adjacentEdges
25:             if cost + distanceToTarget <= timeBudget then
26:                 for edge ∈ adjacentEdges do
27:                     if edge ∉ currentPath then
28:                         queue.enqueue (edge, currentPath, cost)
29:                     end if
30:                 end for
31:             end if
32:         end while
33:     end if
34:     return foundPaths
35: end procedure
```

not in the time budget anymore. The adjacent nodes to B are not enqueued.

- The algorithm continues until the queue is empty.

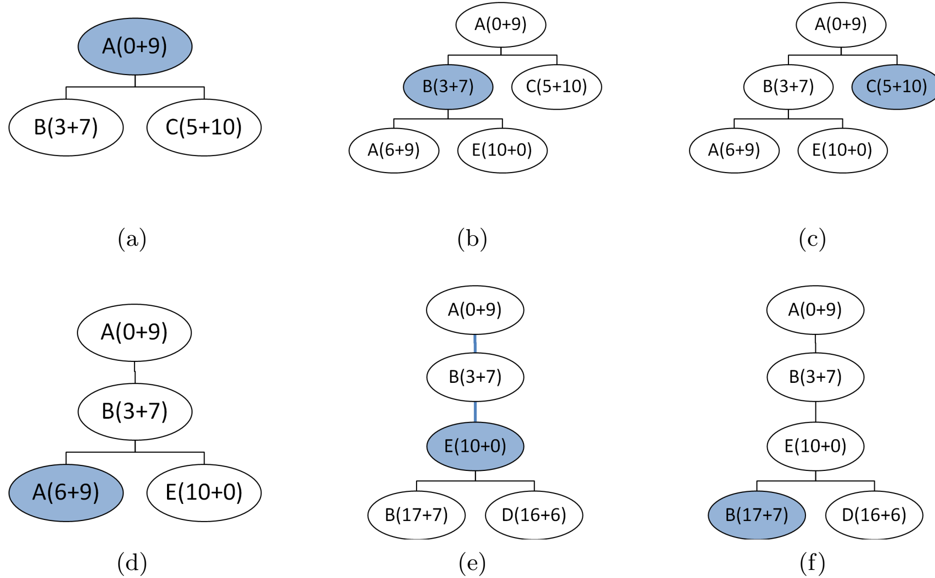

Figure 6: First steps in BFS*

3.5 DFS*

To reduce the runtime of the Depth-First Search I used again the Euclidean distance. It restricts the number of edges that are enqueued. As already explained previously, the Euclidean distance is a lower bound to the network distance. Therefore, if the sum of the cost so far and the Euclidean distance is lower than the time budge, there could exists a path that reaches the target in time.

Algorithm 5 shows the DFS* Algorithm. The algorithm takes the source, the target, the time budget in seconds, and the walking speed in meters per second as parameters and returns a collection of paths.

This algorithm is very similar to DFS. The only difference lies in the condition for pushing an edge onto the stack. While DFS just considers the cost so far, DFS* considers the Euclidean distance to the target too.

Example 6. Figure 7a illustrates a few steps of the DFS* algorithm, using the network in Figure 2. A is the start node and E is the target. The time budget is 11 and the average walking speed is for simplicity 1. The first number in the brackets of the path trees in Figure 7 is the the cost so far, and the second number is the estimate for the cost to reach the target from the current node on.

- First, A is considered (Fig. 7a). As the estimated cost from A to E is 9, the adjacent nodes C and B are pushed onto the stack.

Algorithm 5 DFS*

```
1: procedure DFS*(source, target, timeBudget, walkingSpeed)
2:     adjacentEdges ← ∅
3:     stack ← ∅
4:     foundPaths ← ∅
5:     if distanceToTarget <= timeBudget then
6:         adjacentEdges := source.adjacentEdges
7:         for all edge ∈ adjacentEdges do
8:             stack.push (edge, ∅, 0)
9:         end for
10:        while stack ≠ ∅ do
11:            so := stack.pop ()
12:            currentEdge := so.currentEdge
13:            currentPath := so.currentPath
14:            cost := so.cost + currentEdge.length/walkingSpeed
15:            currentPath.add (currentEdge)
16:            if currentEdge.endNode = target then
17:                startNode := currentPath.firstEdge.startNode
18:                endNode := currentEdge.endNode
19:                if cost ≤ timeBudget then
20:                    newPath := Path (startNode, endNode, cost, currentPath)
21:                    foundPaths.add (newPath)
22:                end if
23:            end if
24:            adjacentEdges := currentEdge.endNode.adjacentEdges
25:            if cost + distanceToTarget <= timeBudget then
26:                for edge ∈ adjacentEdges do
27:                    if edge ∉ currentPath then
28:                        stack.push (edge, currentPath, cost)
29:                    end if
30:                end for
31:            end if
32:        end while
33:    end if
34:    return foundPaths
35: end procedure
```

- Like DFS, DFS* follows the last in first out principle (LIFO), i.e. the last node that is pushed onto the stack is popped out first. B is on top of the stack and is extracted and expanded next (Fig. 7b). Its estimated cost is 10. The adjacent nodes are A and E. Both are added to the stack.

- Then A is on top of the stack (Fig. 7c). The cost so far (6) plus the Euclidean distance from A to the target (9) is greater than the time in the time budget. The adjacent nodes to A are not pushed onto the stack.
- Figure 7d show the next step, the expansion of E, which is the target, and the newly found path $p = \langle (A, B), (B, E) \rangle$.
- The estimated cost for the path through B, the next node on the stack, is 24 and thus above the time limit (Fig. 7e).
- The same is valid for D (Fig. 7f). Both times, no node is added to the stack.
- The algorithm finishes when no more elements are on the stack.

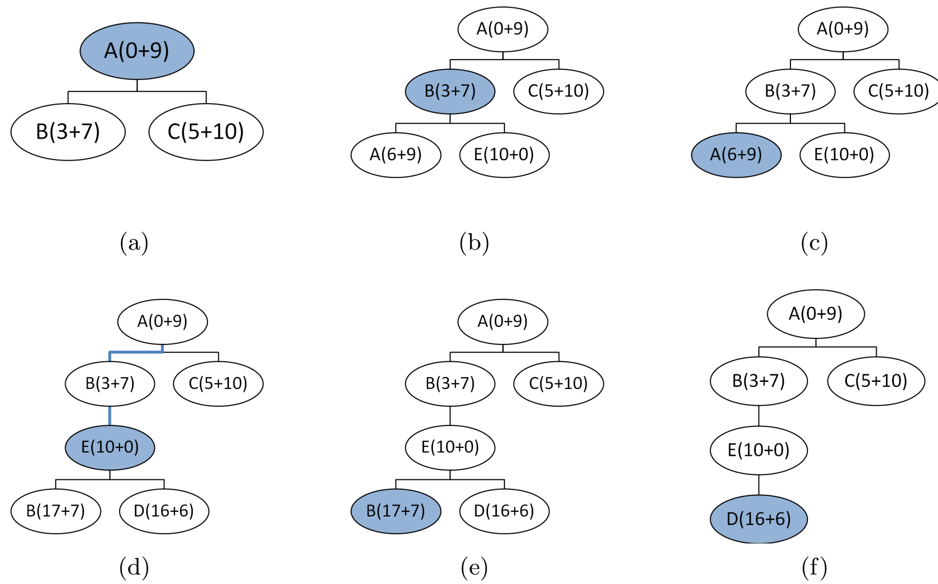

Figure 7: First steps in DFS*

4 Experimental Evaluation

In this section, I am going to compare the algorithms of the previous section in terms of runtime. The runtime includes also the time for reading all the nodes and edges from the database, which is on average 122 milliseconds.

4.1 Setup and Data

The database and the JVM run on the same computer, a Lenovo B570e 2.2GHz Intel Premium B960 with 4GB of RAM. The database server consists of a PostgreSQL server version 9.3.9. For the evaluation the pedestrian network of Bozen is used. The relation *bz_edges* contains 7287 entries, from which 6484 are edges of the pedestrian network. Additionally, the relation *bz_nodes* is used, which has 3245 entries and contains the nodes of both pedestrian and public transport system network. The length of the edges is between 0.72 and 5213.4 meters, and on average 144.4 meters. The average walking speed is in all experiments 0.5m/sec. The runtime is given in milliseconds.

There are six classes of queries:

- Q0: Source and target are the same node
- Q1: The network distance between source and target node is below 240 meters.
- Q2: The network distance between source and target node is between 240 and 480 meters.
- Q3: The network distance between source and target node is between 480 and 720 meters.
- Q4: The network distance between source and target node is between 720 and 960 meters.
- Q5: The network distance between source and target node is between 960 and 1200 meters.

According to Nielsen [7] the basic advice regarding response time is the following:

- **0.1 second** is about the limit for giving the user the impression that the system is reacting instantaneously. No special feedback should be necessary, except for displaying the result.
- **1.0 second** is about the limit where the user will notice the delay, but his flow of though stays uninterrupted. The user loses the feeling of operating directly on the data, but no special feedback is necessary.
- **10 seconds** is about the limit for keeping the attention of the user focused. For longer delays, special feedback is needed, as the user will want to perform other tasks while waiting for the computer to finish. There should be a feedback indicating how much longer the computer will take.

4.2 Runtime Experiments

For each query ten examples are taken. For the computation of the runtime, the algorithms run five times per algorithm and per time budget. The time budgets lie between 1 and 35 minutes. Then the arithmetic mean is taken from the runtimes for each query, algorithm, and time budget. The figures 8, 9, 10, 11, 12, 13 show the results for the different queries. The y-axis is in logarithmic scale, so that the differences for small time budgets are better visible.

Q0 is the set of queries, where source and target are the same node. Every path contains at least one cycle, but usually there are more. The simplest path for this queries is to traverse one adjacent edge and return on the same edge. Figure 8 shows the results until the algorithm reaches an average runtime around one second. If the time budget is very small or the edges are long, it can happen that no paths are found.

BFS is the first algorithm to reach the limit of one second, which is the limit where the user notices the delay, according to Nielsen. The average runtime is 1.561 seconds with a time budget of 25 minutes. DFS is the next one to reach the limit. For a time budget of 26 minutes it needs on average 1.142 seconds. The runtime of A*, DFS*, and BFS* is at a time budget of half an hour around one second. For A* it is 1.125 seconds, for BFS* it is 944 milliseconds, and for DFS* it is 958 milliseconds. These results show that BFS is the worst one, followed by DFS. A*, BFS*, and DFS* have comparable results and are clearly better than the other two, but DFS* is the best one for the queries of Q0.

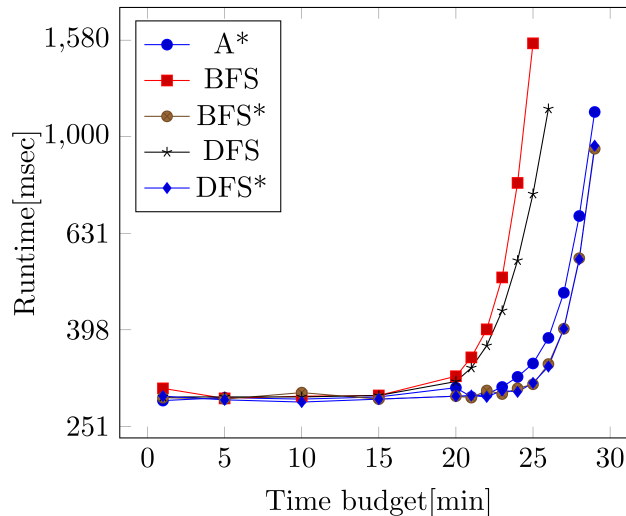

Figure 8: Runtimes Q0

Figure 9 shows the results for the queries in Q1. The length of the

shortest path from source to target for the queries in Q1 is less than 240 meters, but source and target are different. The figure shows the results until the algorithm reaches an average runtime around one second.

BFS is again the first algorithm to reach the limit of one second, which is the limit where the user notices a delay. At a time budget of 25 minutes its average runtime is 1.673 seconds. Again DFS reaches a runtime of over one second before DFS*, BFS*, and A* reach it. Its average runtime is 1.136 seconds with a time budget of 18 minutes. At a time budget of 25 minutes DFS*, BFS*, and A* have an average runtime over one second. DFS* needs 1.150, BFS* 1.437, and A* 1.673 seconds. Again BFS is the slowest one and DFS* the fastest one.

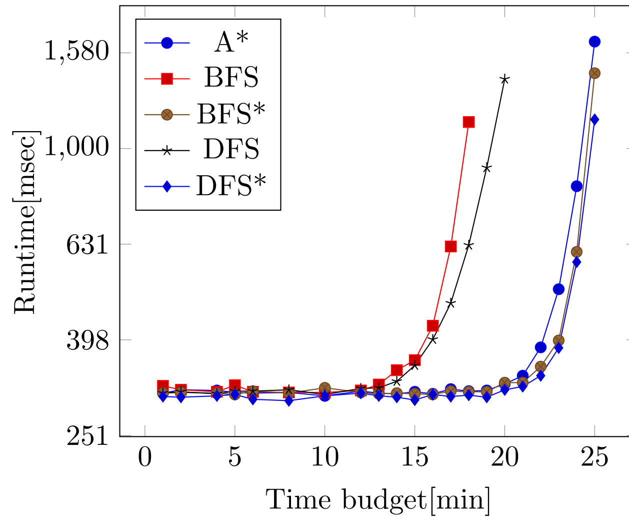

Figure 9: Runtimes Q1

In Figure 10 the runtimes for the queries of Q2 are shown. The nodes of source and target in Q2 are at least 240 meters apart and at most 480. If the walking speed is 0.5m/sec, like it is the case in these experiments, a time budget of at least 480 seconds, or 8 minutes is needed. While BFS and DFS look for a path, DFS*, BFS*, and A* stop immediately, if the time budget is too low and the Euclidean distance is not too far from the actual cost of the shortest path. The figure shows the results until the algorithm reaches an average runtime around one second.

BFS and DFS are the first to reach the limit, like before. BFS reaches the limit of one second at 26 minutes of time budget. The average runtime of DFS is at 939 milliseconds with a time budget of 27 minutes. DFS*, BFS*, and A* are faster and reach the limit at a time budget of 29 minutes. Here the average runtime of A* is a bit faster than the runtime for DFS*. For A* it is 1.246 seconds, BFS* needs 1.272 seconds, and DFS* 1.266 seconds.

The queries of Q3 use examples, where source and target are between

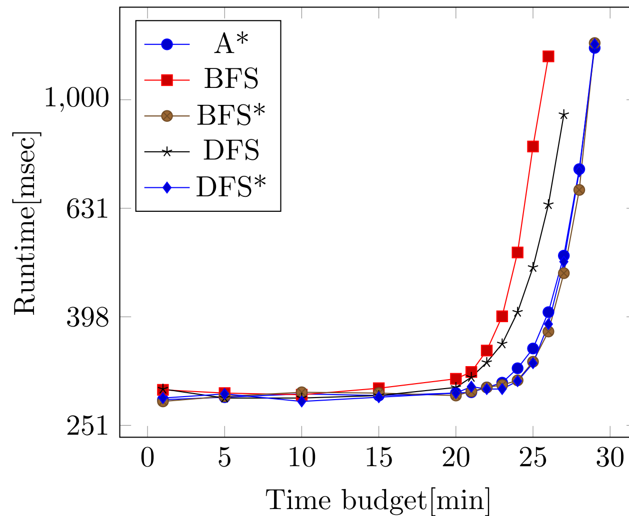

Figure 10: Runtimes Q2

480 and 720 meters apart. Figure 11 shows the results. A time budget of at least 16 minutes, or more is needed to reach the target, if the we assume an average walking speed of 0.5m/sec. For lower time budgets no path can be found. BFS and DFS try to compute a path and expand the nodes as long as they have enough time. A*, DFS*, and BFS* do not expand the nodes, as long as the time budget is not high enough to traverse a distance of the Euclidean distance from source to target. Therefore, BFS and DFS need some time to compute, also for low time budgets, while the other three do not. The figure shows the results until the algorithm reaches an average runtime around one second.

The average runtime for BFS reaches the limit of one second at a time budget of 20 minutes. Its runtime is 1.012 seconds. DFS is again faster than BFS. It reaches the limit at 23 minutes of time budget and needs for that time budget on average 1.189 seconds. For the queries of Q3 A*, BFS*, and DFS* reach the limit of one second not at the same time budget. A* is the first of the three to come near the limit. It has a runtime of 929 millisecond at a time budget of 29 minutes. With one minute more, so at 30 minutes, BFS* has an average runtime of 926 milliseconds. Again with one minute more, DFS* reaches a runtime of 909 milliseconds on average. For the queries of Q3 the order in which the algorithms come near the limit is: BFS, DFS, A*, BFS*, DFS*.

Figure 12 shows the results for Q4. The shortest path between source and target is between 720 and 980 meters long. That means that a time budget of more than 24 minutes is needed. BFS and DFS search also for lower budgets a path, but cannot find one. DFS*, BFS*, and A* need a time budget that is high enough to traverse a path of the length of the Euclidean

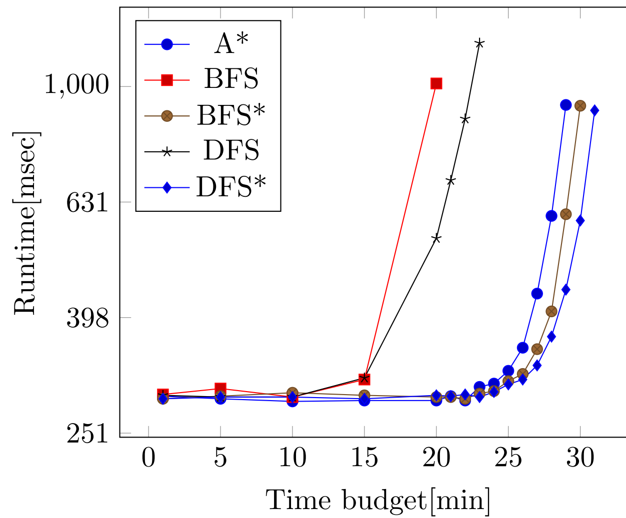

Figure 11: Runtimes Q3

distance before they start looking. The figure shows the results for BFS and DFS until the algorithm reaches an average runtime around one second and for A*, BFS*, and DFS* until a time budget of half an hour.

Like before, BFS is the first to cross the limit of one second. It reaches 2.036 seconds of runtime at 23 minutes of time budget. DFS has a runtime of 1.375 seconds at 24 minutes. This is the point where the average runtime is above the limit of one second. The runtime for A*, DFS*, and BFS* is still around 0.3 seconds for a time budget of 30 minutes. Because of the high distance there are not many possible paths and the direction in which to search is therefore relatively clear defined.

Figure 13 shows the results for the queries of Q5. The shortest path between source and target has at least a length of 960 and at most a length of 1200 meters. With a walking speed of 0.5m/sec that means that for the shortest path between 32 and 40 minutes are needed. The figure shows the results for BFS and DFS until the algorithm reaches an average runtime around one second and for A*, BFS*, and DFS* until a time budget of half an hour.

The first algorithm to reach the one-second-limit is again BFS. Its runtime is 1.107 seconds at a time budget of 12 minutes. The runtime of DFS is at 14 minutes of time budget above the limit of one second. It needs there 1.509 seconds on average. That means that BFS and DFS cross the limit of one second before even getting enough time to reach the target. As the runtimes are only computed up to a time budget of 30 minutes, the runtimes for A*, DFS*, and BFS* are low, i.e. their runtime is still below 0.3 seconds. They just need the time to read the database and check that the target is too far away from the source to reach it.

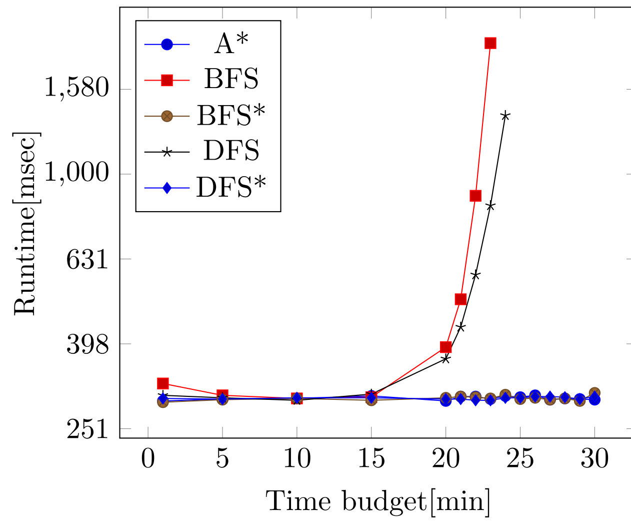

Figure 12: Runtimes Q4

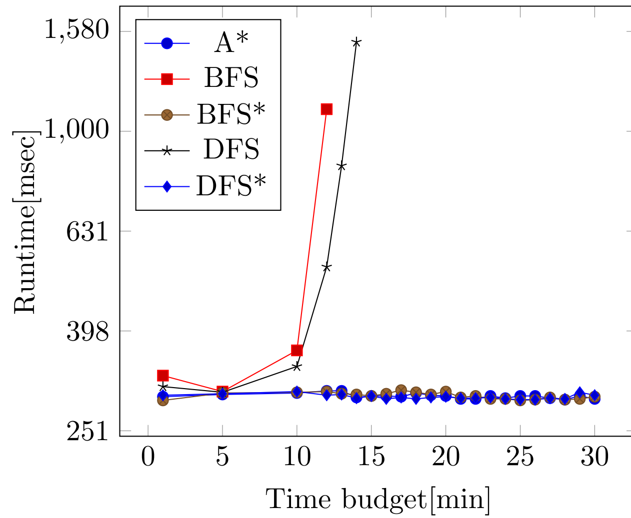

Figure 13: Runtimes Q5

All in all, one could say that DFS* is almost always the best one for the cases considered here. BFS is the worst one. Also DFS is not very useful for low time budgets and high distances. They start the search for paths although there is not enough time to reach the target. DFS*, A*, and BFS* are doing better and stop immediately when the time budget is too low for the distance between source node and target node.

5 Related Work

5.1 Shortest Path Algorithms

The research field of graph algorithms lies at the intersection of graph theory and computer science. Problems that are of great interest are the shortest path problems. As Demetrescu, Goldberg, and Johnson [1] put it:

> *Shortest path problems are among the most fundamental combinatorial optimization problems with many applications, both direct and as subroutines in other combinatorial optimization algorithms. Algorithms for these problems have been studied since the 1950's and still remain an active area of research.*

Whenever we want to send something or someone through a network between two points or want to traverse a network in a fast or cheap way, solving the shortest path problem is likely to provide the optimal solution.

Shortest path algorithms can be classified according to the problem type. There are single source shortest path algorithms, single pair shortest path algorithms, and all pairs shortest path algorithms. The single source shortest path algorithms find a shortest path from a given source node to all other nodes of the graph. Single pair shortest path algorithms find the shortest path between two given nodes. All pair shortest path algorithms find for every pair of nodes the shortest path between them.

The most famous algorithm for solving single source shortest path problems is Dijkstra's algorithm [2]. It starts at the source node and explores the whole graph in all directions until all the shortest paths and the distance to all the other nodes is known. The algorithm maintains two sets of nodes:

- A: Set of nodes for which the distance to the source is known
- B: Set of remaining nodes, for which the exact distance is not know yet

At the beginning the distance $d(s, s) = 0$ and all the other nodes have labels with the distance set to ∞. At each step the algorithm selects the node u with the shortest tentative distance to the source from the set B. The tentative distance is the distance to the node only through the nodes in A. When a node is selected, its exact distance from the source is known. Then, the algorithm updates the labels of all the neighbors of u in the set B if necessary. That is, if the tentative distance of v is larger than $d_N(s, u) + \lambda(u, v)$, then its label is decreased.

Dijkstra's algorithm can also be used to solve single pair shortest path problems. It just needs to stop, when the target node is extracted from set B.

For road networks in addition to the graph the coordinates of the nodes are know. Heuristics based on the geometry help to guide the search towards the target [3].

The A* algorithm uses this heuristics. In the implementation of Dijkstra's algorithm that uses a priority queue, at each iteration the node with the shortest distance to the source is dequeued. The A* algorithm orders the nodes by their distance from the source plus the potential distance from the target. A good potential function alters the order in which the nodes are removed from the queue, so that the nodes that lie on a shortest path are removed first. In road networks the Euclidean distance provides a good lower bound on the graph distance and thus a good potential function, if the coordinates are known [3].

5.2 Isochrones

Isochrones is a similar concept to our problem. 'An isochrone in a spatial network is the minimal, possibly disconnected subgraph that covers all the locations from where a query point is reachable within a given time span and by a given arrival time' [4]. That means, an isochrone is a subgraph that can be disconnected and it contains all space points with a shortest path to the query point that is smaller than the given time span. Given an objects relation, all objects within an isochrone can be determined without computing the distance to the individual objects. That can be done by intersecting the area of the isochrone with the relation [5].

I have tried to use isochrones to solve the problem of computing all paths between two locations with a maximal time budget. For this approach two isochrones are used to restrict the search space. An isochrone in a spatial network cannot only be the minimal, possibly disconnected subgraph that covers all the locations from where a query point is reachable within a given time span and by a given arrival time, but it can also be the minimal, possibly disconnected subgraph that covers all the locations that can be reached from a query point within a given time span.

For this approach two isochrones are computed, one with the source location as a query point and one with the target location. The first one contains all the points that are reachable from the query point, i.e. all the nodes are on outgoing edges from the source node, and the second one contains all the points from where the query point is reachable, i.e. all the nodes are on ingoing edges to the target node. As time span the given time budge is used. The arrival time can be ignored, because a pedestrian network is used and there is no schedule for that one.

Lemma 1. All the nodes that are part of a path of the solution have to lie in the intersection of the two isochrones.

Proof. Let us assume by contradiction that $\exists v_x \in V$ s.t. $v_x \in I_s \wedge v_x \notin$

$I_t \wedge v_x \in AP$, where I_s is the isochrone for the start node and I_t for the target node.

If $v_x \in AP \Rightarrow \exists p \in AP$ s.t. $v_x \in p \Rightarrow \exists q\,(v_s, v_x) \in p \wedge \exists r\,(v_x, v_t) \in p$ s.t. $\gamma(q) + \gamma(r) \leq t \Rightarrow \gamma(r) \leq t - \gamma(q)$, where p is a path, q and r are subpaths of p, and t is the time in the time budget. Then $\gamma(r) \leq t$, which means that v_t is reachable from v_x in less than or equal time to the time in the time budget. As I_t contains all the nodes from where v_t is reachable, it has to contain also v_x, which contradicts our assumption that $\exists v_x \in V$ s.t. $v_x \in I_s \wedge v_x \notin I_t \wedge v_x \in AP$. □

As I have proven that all the nodes that are part of a path lie in the intersection of the isochrones, I can reduce the search to just those nodes. After this reduction another algorithm, like BFS* or DFS* has to be used to find all the paths. The computation of the isochrones is very expensive and time consuming.

6 Conclusion and Future Work

In this thesis I looked for solutions for the problem of finding all paths between two locations with a given time budget. I presented five algorithms that could be uses, which are Depth-First Search, Breadth-First Search, DFS*, BFS*, and A*. Depth-First Search makes as much forward progress as possible, while Breadth-First Search looks first at all nodes that are k edges away from the source and then continues with the nodes at a distance of $k + 1$ edges. A* uses a heuristic function and a priority queue to decide where to continue the search. DFS* and BFS* use similar approaches to DFS and BFS respectively. They use a heuristic function to stop them from searching too long in the wrong direction.

As one can see, these approaches are useful for relatively small time budgets and low walking speeds. Breadth-First Search is in all cases the slowest one, followed by Depth-First Search. BFS*, DFS*, and A* have low runtimes compared to them. They are also more efficient, as they do not try to compute the set of paths, if the time budget is too low. DFS* is the best of the last three in the experiments.

For my implementations of the algorithms I used the network with discrete space mode. In reality a pedestrian network is usually continuous, i.e. it can be accessed and left wherever one wants. Therefore, the algorithms need to be extended, so that the paths can also contain edge segments. I did not do that here.

References

[1] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, "Implementation challenge for shortest paths," in *Encyclopedia of Algorithms* (M.-Y. Kao, ed.), pp. 1–99, Springer US, 2008.

[2] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] R. Sedgewick and J. Vitter, "Shortest paths in euclidean graphs," *Algorithmica*, vol. 1, no. 1-4, pp. 31–48, 1986.

[4] J. Gamper, M. Böhlen, W. Cometti, and M. Innerebner, "Defining isochrones in multimodal spatial networks," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, (New York, NY, USA), pp. 2381–2384, ACM, 2011.

[5] J. Gamper, M. Böhlen, and M. Innerebner, "Scalable computation of isochrones with network expiration," in *Scientific and Statistical Database Management* (A. Ailamaki and S. Bowers, eds.), vol. 7338 of *Lecture Notes in Computer Science*, pp. 526–543, Springer Berlin Heidelberg, 2012.

[6] G. T. Heineman, G. Pollice, and S. Selkow, *Algorithms in a Nutshell*. Sebastopol: O'Reilly Media, 2008.

[7] J. Nielsen, "Response times: The 3 important limits." `http://www.nngroup.com/articles/response-times-3-important-limits/`, 1993. Accessed August 3, 2015.