Shortest paths on very large graphs

1. Definitions

A directed graph $G = (N, E)$ is given. Let $n = |N|$ and $m = |E|$. If the shortest path problem is formulated on a non directed graph it is always possible to turn the graph into a directed one by replacing the non directed arc (i, j) with the anitparallel pair (i, j) and (j, i).

For each arc $(i, j) \in E$ a *length* $d_{ij} \geq 0$ is defined. Two nodes are special and are called *source* and *destination*, respectively denoted as s and t. Given a path its *path length* is the sum of the lengths of the arcs in the path. The *shortest path* between the nodes i and j is the path (non necessarily unique) of minimum length between the given nodes. The *distance* between the node i and the node j is the length of the shortest path $i \to j$.

The problem consists in finding the shortest path from the source to the destination. Let δ be the distance from s to t, δ_i the distance from s to i, δ^i the distance from i to t and δ^i_j the distance between two generic nodes i and j (in the given order - in general $\delta^i_j \neq \delta^j_i$).

The properties of a shortest path imply $\delta^i_j \leq \delta^i_k + \delta^k_j$ for each $i, j, k \in N$ and in particular $\delta \leq \delta_i + \delta^i$ for each $i \in N$. In more detail

$$\begin{aligned} &\delta = \delta_i + \delta^i \qquad \text{if } i \text{ belongs to a shortest path,} \\ &\delta < \delta_i + \delta^i \qquad \text{if } i \text{ does not belong to a shortest path} \end{aligned} \tag{1}$$

2. Computational complexity of Dijkstra's algorithm

Dijkstra's algorithm has complexity $O(n^2)$ with a simple implementation, $O((n+m)\,\log n) = O(m\,\log n)$ if we use binary heaps and $O(m + n\,\log n)$ if we use Fibonacci heaps. The implementation with binary heaps is convenient for planar graphs where the number of arcs grows linearly with the number of nodes (more exactly $m \leq 3\,n - 6$) so that the complexity lowers to $O(n\,\log n)$. Yet, for very large graphs with more than 100.000 nodes, the number of computations is too large if we want the solution quickly. For instance with 100.000 nodes and 300.000 arcs, the number of operations is $(100.000 + 300.000)\,\log_2 100.000 \approx 6.643.860$, whose execution may require several seconds.

However, this is a worst-case evaluation, whereas Dijkstra's algorithm has a much more efficient behavior on average. The most interesting feature is that the algorithms terminates as soon as the destination node is visited.

Let us recall the basic steps of the algorithm. At each iteration there is a subset of nodes whose distance from the source has been computed. Let us call these nodes *visited*. Initially the source is the only visited node (in the forward search, whereas it is the destination in the backward search). The nodes that have not been visited yet and are adjacent to some visited node are said *reached*. For these nodes a temporary evaluation of their distance is available. For all other nodes the distance is unknown. The temporary distances of the reached nodes are stored in a binary heap. At each algorithm iteration the node among

the reached nodes with the minimum temporary distance becomes visited, its distance becomes permanent and it is removed from the heap. The nodes adjacent to this node may be already visited and in this case they are skipped by the algorithm, or they may be already reached and in this case their distance may be updated, or they may be neither visited nor reached and in this case they become reached and are inserted into the heap.

The algorithm terminates when the destination (or the source in the backward formulation) is visited. Hence, if the destination is not very far from the source, only a few operations may be required. Clearly the number of operations depends also on the distance δ between source and destination. Hence it is convenient to take into account this distance in order to better evaluate the number of operations. Let $n(d)$ be the number of nodes at distance less than or equal to d. Then we may express the complexity as $O(n(\delta)\ \log n(\delta))$. Typically for a planar graph $n(d)$ is a quadratic function of d. Roughly speaking one can imagine the set of nodes at distance at most d as a circle of ray d centered in the source.

A measure of the *efficiency* of the algorithm, applied to a particular instance, is given by the ratio between the number of nodes of the shortest path and the number of visited nodes (the nodes that have been reached but have not been visited yet, are not included). An efficiency equal to 1 means that only the nodes of the shortest path are visited one after the other up to the destination.

3. Bidirectional search

A first idea in order to reduce the number of operations consists in starting two simultaneous shortest paths, one from the source to the destination and the other one from the destination to the source. The two paths will meet at about $\delta/2$ distance. Hence the number of nodes that each search has to visit has been reduced to 1/4 (in the hypothesis of a quadratic $n(d)$). In total only half nodes need to be visited and the efficiency has been doubled.

The algorithm terminates when there is a node visited by both searches. Let us denote with r this particular node. Let S_s the set of nodes visited from s and S^t the set of nodes visited from t. By definition $S_s \cap S^t = \{r\}$. For each node $i \in S_s$ the distance δ_i is known and for each node $i \in S^t$ the distance δ^i is known.

We claim that the shortest path $s \to t$ must be among the paths $s \to i \to j \to t$ where $i \in S_s$, $j \in S^t$ and the arc (i,j) exists, or $s \to r \to t$. The paths $s \to i \to j \to t$ have length $\delta_i + d_{ij} + \delta^j$ whereas the path $s \to r \to t$ has length $\delta_r + \delta^r$. Therefore the best among these paths has length at most $\delta_r + \delta^r$. Every other path must include a node $k \notin S_s \cup S^t$. By the properties of Dijkstra's algorithm one has $\delta_k \geq \delta_r$ and $\delta^k \geq \delta^r$ so that $\delta_k + \delta^k \geq \delta_r + \delta^r$. Hence the shortest path is $s \to r \to t$ or one of the paths $s \to i \to j \to t$. Actually the path $s \to r \to t$ must have been necessarily evaluated in a previous step as one of the paths $s \to i \to j \to t$.

Hence the algorithm, each time an arc (i,j) is detected with $i \in S_s$ and $j \in S^t$, computes the length of the path $s \to i \to j \to t$ and compares it with the ones already computed.

Example. Let us consider a grid graph ($50 \times 50 = 2500$ nodes and 4900 arcs with lengths uniformly randomly generated on $\{1, \ldots, 5\}$). The node in red is the source and the one in blue the destination. In

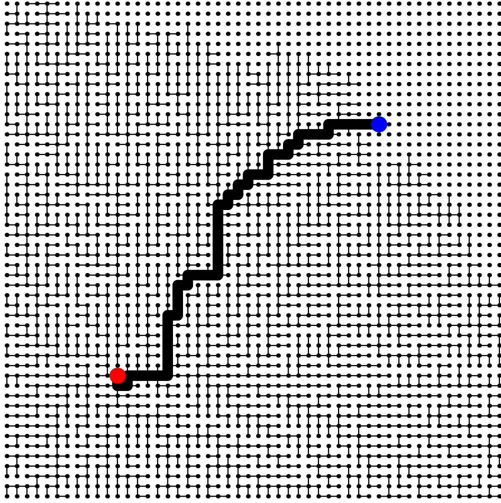


Figura 1(a)

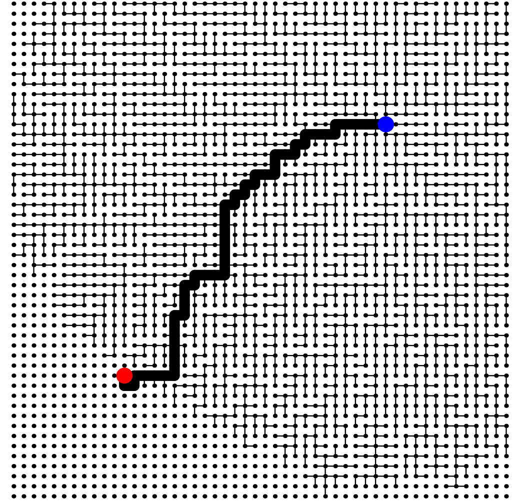


Figura 1(a)

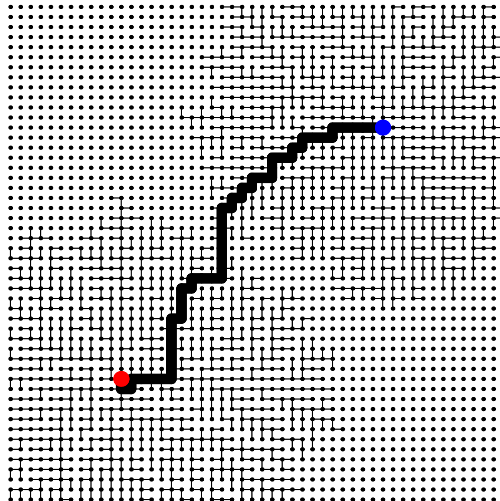


Figura 2

Figure 1(a) we see the tree of the nodes searched starting from the source and in Figure 1(b) the tree from the destination.

The shortest path has 54 nodes, 2107 nodes are visited in the forward search and 2100 in the backward search with an efficiency of 0.0256289 and 0.0257143 respectively. As apparent almost all nodes need to be visited (84%) and therefore the possibility of stopping the algorithm as soon as the destination si visited does not produce a real improvement. By using the bidirectional search, the trees at the algorithm termination, when they have in common exactly one node, are shown in Figure 2. In this case 1638 nodes are visited (65.5%) with efficiency 0.032967.

4. Using potentials on the nodes

If we observe the nodes that are visited by the algorithm (Figure 2), even with the bidirectional search, we may see that the algorithm 'wastes time' to visit nodes that are in the opposite direction with respect to the nodes where the shortest path is expected to go. These nodes are of no use in computing the shortest path. Clearly this observation depends on our visual knowledge of the graph, whereas the algorithm has no prior information on the nodes still to be visited.

Hence the idea could be the one of feeding this kind of information to the algorithm. To this purpose, if we run Dijkstra's algorithm in a forward search let us modify the arc lengths as

$$d'_{ij} := d_{ij} - \pi^i + \pi^j \tag{2}$$

where π^i are arbitrary values assigned to the nodes, called *potentials*. Just note that the potentials are defined up to an additive constant. This allows to assign to a particular node an arbitrary value. Since we need non negative distances in order to run Dijkstra algorithm the potentials must satisfy the inequalities

$$\pi^i \leq \pi^j + d_{ij} \qquad (i,j) \in E \tag{3}$$

Moreover, the values π^i that are feasible in (3) with $\pi^t \leq 0$ are lower bounds to δ^i. Indeed, if we sum the inequalities $\pi^i - \pi^j \leq d_{ij}$ along a shortest path $i \to t$, we obtain $\pi^i - \pi^t \leq \delta^i$, that is, $\pi^i \leq \delta^i + \pi^t \leq \delta^i$ (from $\pi^t \leq 0$). Hence the potentials must be an underestimation (possibly also the exact estimation) of the backward shortest path values δ^i from i to t. However, the converse is not necessarily true, i.e., arbitrary lower bound values to optimal paths do not necessarily lead to non negative distance d'.

The length of a path $P : s \to t$ with respect to the new distances is

$$d'(P) = \sum_{(ij)\in P} d'_{ij} = \sum_{(ij)\in P} (d_{ij} - \pi^i + \pi^j) = d(P) + \pi^t - \pi^s$$

The lengths of all paths $s \to t$ have therefore been changed by the constant factor $\pi^t - \pi^s$ and the shortest path $s \to t$ is the same for any choice of the potentials, so that

$$\delta' = \delta + \pi^t - \pi^s$$

Hence the shortest path can be computed with an arbitrary choice of the potentials provided that the new lengths d' are non negative, otherwise we cannot use Dijkstra's algorithm. Similarly, with the new distances d', the length of a path P_k from s to a node k has value

$$d'(P_k) = \sum_{(ij)\in P_k} d'_{ij} = \sum_{(ij)\in P_k} d_{ij} - \pi^i + \pi^j = d(P_k) + \pi^k - \pi^s \tag{4}$$

and we may also say that

$$\delta'_k = \delta_k + \pi^k - \pi^s \tag{5}$$

This implies that the forward shortest path tree rooted in s is the same for any choice of the potentials. The difference is related to the distances on the tree and it is indeed this fact that may speed up the algorithm with an accurate choice of potentials. Just recall that the algorithm (in the forward search) selects the nodes

in order of increasing distance δ'_k, i.e., from (5), according to the values δ_k of the shortest path $s \to k$ with the original lengths plus the value π^k, which is an underestimation of the optimal path k to t (the constant value π^s is irrelevant because it is independent of k).

As an extreme example, if $\pi^k = \delta^k$, i.e., each potential is exactly the distance from k to t and not just an underestimation, for all nodes on the shortest path $s \to t$ we would have $\delta'_k = 0$ (from $\pi^s = \delta$ and from (1)), whereas for the nodes not on the shortest path we would necessarily have $\delta'_k > 0$ (again from $\pi^s = \delta$ and from (1)). This implies that the algorithm will first choose all and only the nodes of the shortest path.

Symmetrically, if we run Dijkstra's algorithm starting from the destination, we may redefine the distances as

$$d''_{ij} := d_{ij} - \pi_j + \pi_i$$

with arbitrary potentials π_i (the difference with (2) should be noted). As before the potentials must satisfy the inequalities

$$\pi_j \le \pi_i + d_{ij} \qquad (i,j) \in E \tag{6}$$

and we may see that the distances d''_{ij} are non negative only if the potentials π_i, with $\pi_s \le 0$, are an underestimation of the forward shortest path values δ_i from s to i.

With the necessary differences the same properties of the forward formulation hold for the backward formulation, namely: the length of any path $P : s \to t$ with the new distances is

$$d''(P) = \sum_{(ij) \in P} (d_{ij} - \pi_j + \pi_i) = d(P) + \pi_s - \pi_t$$

In this case all path lengths $s \to t$ are changed by the constant factor $\pi_s - \pi_t$, from which

$$\delta'' = \delta + \pi_s - \pi_t$$

The length of a path P^k from a generic node k to t is

$$d''(P^k) = \sum_{(ij) \in P^k} (d_{ij} - \pi_j + \pi_i) = d(P^k) + \pi_k - \pi_t$$

so that

$$\delta''^k = \delta^k + \pi_k - \pi_t \tag{7}$$

5. Bidirectional algorithm with potentials

The bidirectional algorithm executes two simultaneous searches, one from the source with lengths d' and another one from the termination with distances d''. The potentials for the two searches are arbitrary and in general they are different. They only have to be feasible in (3) and (6) respectively with the constraints $\pi^t = 0$ and $\pi_s = 0$.

As in the previous bidirectional algorithm, as soon as an arc (i,j) is found with $i \in S_s$ and $j \in S^t$, one computes the true length of the path $s \to i \to j \to t$ and compares it with the value $d(P)$ of the best path found so far, possibly updating it.

The algorithm terminates as soon as a node k has been visited such that either $\delta_k + \pi^k \geq d(P)$ in the forward search or $\delta^k + \pi_k \geq d(P)$ in the backward search. Note that only one of the two conditions needs to be fulfilled to stop the algorithm. The condition $\delta_k + \pi^k \geq d(P)$ may be also expressed as $\delta'_k + \pi^s \geq d(P)$ (from (5)) where δ'_k is directly available from the Dijkstra's algorithm. It has to be remarked that Dijkstra's algorithm, working with lenghts d', visits the nodes in order of increasing values δ'_k, i.e., in order of increasing values $\delta_k + \pi^k$. Analogously the condition $\delta^k + \pi_k \geq d(P)$ may be expressed as $\delta''^k + \pi_t \geq d(P)$ (from (7)).

We claim that at the termination the shortest path has been found. Let P be the best path found so far with length $d(P)$. Note that all paths that have been computed are necessarily made up of visited nodes (either in the forward search or in the backward search). Hence P is the shortest path among those paths that consist of visited nodes. Let Q be a path that includes also non visited nodes and let $d(Q)$ be its length. Let $h \in Q$ be one of the non visited nodes (neither in the forward search nor in backward search). Let Q_h be the part of path Q from s to h and let Q^h be the part of path Q from h to t so that $d(Q) = d(Q_h) + d(Q^h)$. Since the potentials π^i are lower bounds to the shortest path values from i to t, i.e. $\pi^i \leq \delta^i$, we have in particular for the node h

$$\delta_h + \pi^h \leq d(Q_h) + \pi^h \leq d(Q_h) + \delta^h \leq d(Q_h) + d(Q^h) = d(Q)$$

Analogously, we have in the other direction

$$\delta^h + \pi_h \leq d(Q^h) + \pi_h \leq d(Q^h) + \delta_h \leq d(Q^h) + d(Q_h) = d(Q)$$

Let k be the node for which one of the termination conditions is fulfilled. Since h has not been visited and the algorithm visits the nodes in S_s for increasing values $\delta_i + \pi^i$ and in S^t for increasing values $\delta^i + \pi_i$, one has $\delta_k + \pi^k \leq \delta_h + \pi^h$ if $k \in S_s$ or $\delta^k + \pi_k \leq \delta^h + \pi_h$ if $k \in S^t$.

In both cases the termination condition implies $d(P) \leq d(Q)$.

6. Choice of potentials

Various potential choices have been proposed. One of the most effective consists in the choice of a subset of nodes, that we may call *landmarks*, and in computing the potentials from the distances between the landmarks. More exactly let $L \subset N$ be a subset sufficiently small to make it possible to compute and store all distances between any node in N and any node in L but not too small to make the potentials useless. The triangular inequality implies

$$\begin{array}{llll}
\delta_\ell \leq \delta_i + \delta^i_\ell, & \text{i.e.,} & \delta_\ell - \delta^i_\ell \leq \delta_i & \ell \in L, i \in N \\
\delta^\ell_i \leq \delta^\ell_s + \delta_i, & \text{i.e.,} & \delta^\ell_i - \delta^\ell_s \leq \delta_i & \ell \in L, i \in N \\
\delta^\ell \leq \delta^\ell_i + \delta^i, & \text{i.e.,} & \delta^\ell - \delta^\ell_i \leq \delta^i & \ell \in L, i \in N \\
\delta^i_\ell \leq \delta^i + \delta^t_\ell, & \text{i.e.,} & \delta^i_\ell - \delta^t_\ell \leq \delta^i & \ell \in L, i \in N
\end{array}$$

Therefore, from the stored values δ^i_ℓ and δ^ℓ_i, a lower bound of δ_i is given by $\max \left\{\delta_\ell - \delta^i_\ell, \delta^\ell_i - \delta^\ell_s\right\}$, and a lower bound of δ^i is given by $\max \left\{\delta^\ell - \delta^\ell_i, \delta^i_\ell - \delta^t_\ell\right\}$. A stronger lower bound can be obtained by taking the maximum value, i.e.,

$$\pi_i = \max_{\ell \in L} \max \left\{\delta_\ell - \delta^i_\ell, \delta^\ell_i - \delta^\ell_s\right\}, \qquad \pi^i = \max_{\ell \in L} \max \left\{\delta^\ell - \delta^\ell_i, \delta^i_\ell - \delta^t_\ell\right\} \tag{8}$$

We have to check that these values satisfy $\pi_j - \pi_i \le d_{ij}$ and $\pi^i - \pi^j \le d_{ij}$. We show just a few cases. For instance, suppose we have

$$\pi^i = \delta^\ell - \delta^\ell_i, \qquad \pi^j = \delta^\ell - \delta^\ell_j$$

where ℓ is the same landmark for both cases. Then

$$\pi^i - \pi^j = \delta^\ell - \delta^\ell_i - (\delta^\ell - \delta^\ell_j) = \delta^\ell_j - \delta^\ell_i \le d_{ij}$$

where the triangular inequality derives from the property of shortest path $\ell \to j$. As another case let

$$\pi^i = \delta^i_\ell - \delta^t_\ell, \qquad \pi^j = \delta^j_\ell - \delta^t_\ell$$

Then

$$\pi^i - \pi^j = \delta^i_\ell - \delta^t_\ell - (\delta^j_\ell - \delta^t_\ell) = \delta^i_\ell - \delta^j_\ell \le d_{ij}$$

where the triangular inequality derives from the property of shortest path $i \to \ell$. Let us now consider the case of maximum obtained with different landmarks

$$\pi^i = \delta^i_{\ell_1} - \delta^t_{\ell_1}, \qquad \pi^j = \delta^j_{\ell_2} - \delta^t_{\ell_2} > \delta^j_{\ell_1} - \delta^t_{\ell_1}$$

Then

$$\pi^i - \pi^j = \delta^i_{\ell_1} - \delta^t_{\ell_1} - (\delta^j_{\ell_2} - \delta^t_{\ell_2}) < \delta^i_{\ell_1} - \delta^t_{\ell_1} - (\delta^j_{\ell_1} - \delta^t_{\ell_1}) = \delta^i_{\ell_1} - \delta^j_{\ell_1} \le d_{ij}$$

The other cases can be proved in a similar way.

If the node i is on the shortest path between s and the landmark ℓ, then π_i is exactly the shortest path value $s \to i$ and, analogously, π^i is the shortest path value $i \to t$ if i is on the shortest path between the landmark ℓ and t.

The computation of δ^ℓ_i is carried out with $|L|$ forward executions of the Dijkstra's algorithm, each one of them starting from a landmark up to visiting all nodes. Analogously the computation of δ^i_ℓ is carried out with $|L|$ backward executions of the Dijkstra's algorithm, each one of them starting from a landmark up to visiting all nodes. This is clearly a time consuming computation, but is done off-line only once and its data are then retrieved in constant time by each subsequent execution.

If the graph is symmetrical, one of the two executions is enough because the forward execution produces the same results of the backward execution. The road graphs are 'almost' symmetrical and both executions need to be done. However, the values δ^ℓ_i and δ^i_ℓ are very close and this fact is exploited for their storage. Indeed most bits of the two numbers are equal and so they are stored only once, whereas the last bits are different and are stored separately. This allows for a data compression of almost 50%. If the graph has one million nodes and 16 landmarks (the maximum obtainable with the current technology if the node are one million) this compression allows storing the data on a flash card and reading quickly the data.

The potentials computed according to (8) depend on s and t. Therefore they must be recomputed each time a shortest path between two new nodes has to be computed. However, their computation can be done within the Dijkstra's algorithm as soon as a node is reached, thus providing a dramatic reduction of the computing time.

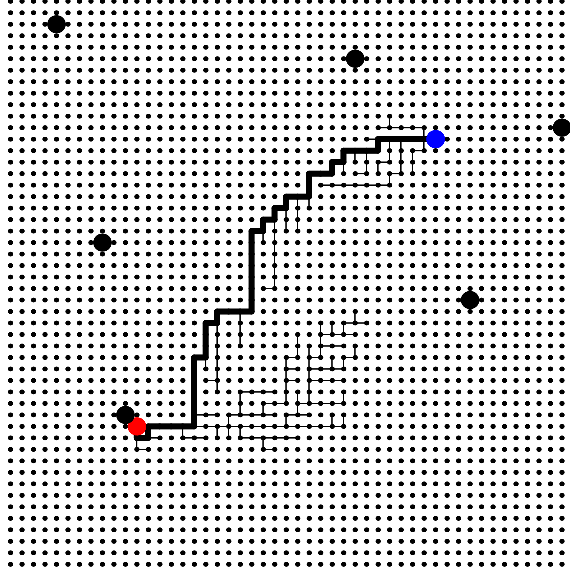


Figura 3 – Random landmarks

We show two alternative choices of landmarks. The first one is random. First the number p of landmarks is decided and then p nodes are uniformly randomly chosen in N.

In Figure 3 we see a random choice of 6 landmarks out of 2500 nodes together with the visited nodes and the shortest path. It can be observed that where landmarks are 'missing' the algorithm has to visit many nodes. The visited nodes are 206 with an efficiency 0.262136.

In Figure 4 the results are shown for nine random choices of the source and the destination with the same set of landmarks. The three numbers under each figure are the number of nodes of the shortest path, the number of visited nodes and the efficiency.

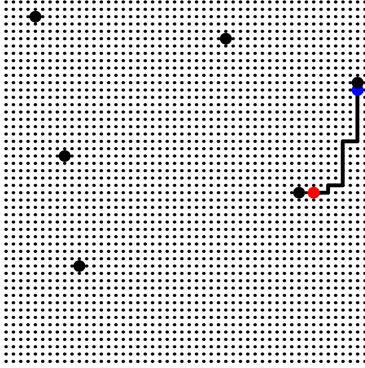
The second choice of landmarks is also random but tries to distribute them as uniformly as possible. Initially a node is chosen randomly, then the second node is the most distant from the first, the third one is the most distant from the generated set and so on until all landmarks are generated. One may use either the real distances or the distances taken as the number of arcs. If the real distance are used, one may compute in this phase also the values δ^i_ℓ. In order to find the most distant node from the generated landmarks one can use a heap structure that contains all nodes of the graph and the distances from the set of landmarks. The heap root is the most distant node. When the root value is removed (and the heap is updated) one computed the distances of this landmark to all other nodes, and the heap is consequently updated.

In Figure 5, six landmarks are shown that have been generated in this way (they have been generated according to this order: first the random one on the top-left, then bottom right, then top right, bottom left, in the center and bottom in the middle) and the visited nodes for the same source-destination pair. The visited nodes are 168 (versus 54 of the shortest path) with an efficieny 0.321429.

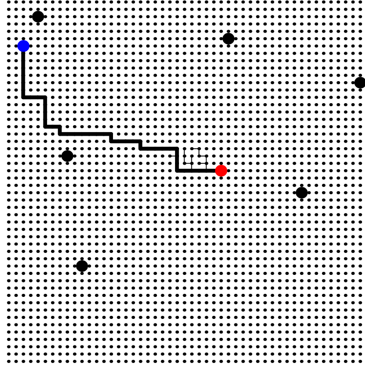
In Figure 6 we show the results for the same random choices of source and destination.

By using similar techniques (16 landmarks) it is possible to compute in time less than 200 ms shortest paths for road graphs with more than 6 millions nodes and 15 millions arcs (corresponding to part of the United States).

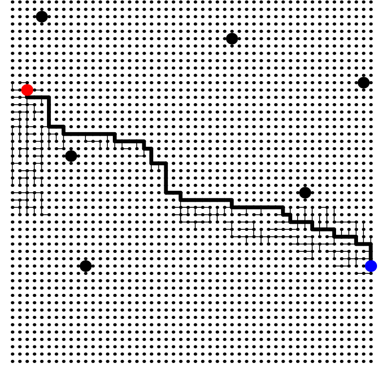
Fig. 5 – Uniform landmarks, shortest paths



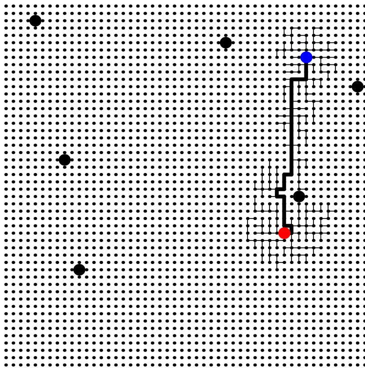
21, 22, 0.954545



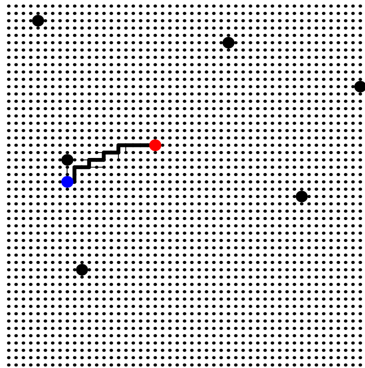
45, 58, 0.775862



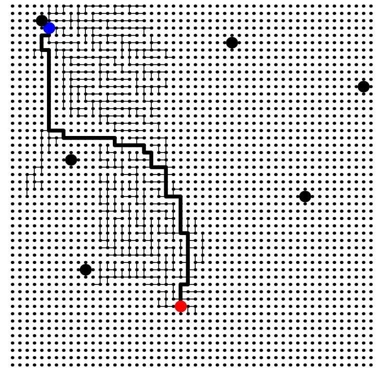
72, 318, 0.226415



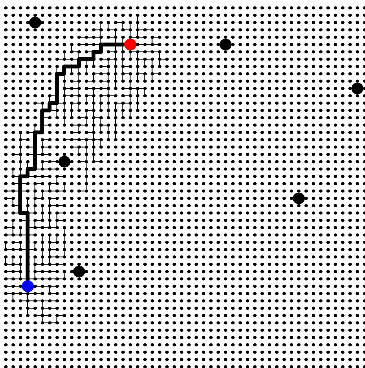
32, 236, 0.135593



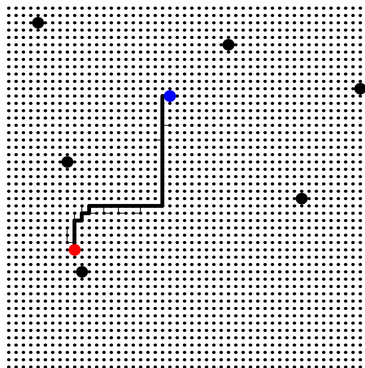
18, 32, 0.5625



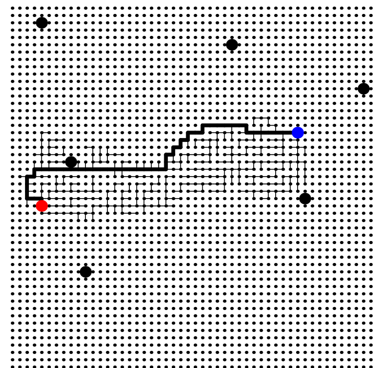
61, 582, 0.104811



50, 436, 0.114679

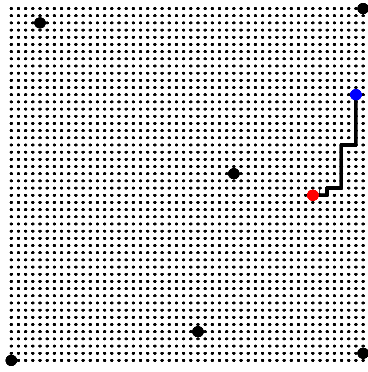


35, 50, 0.7

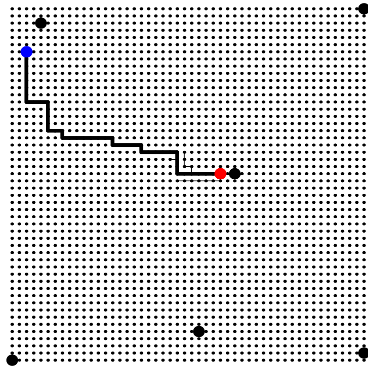


52, 360, 0.144444

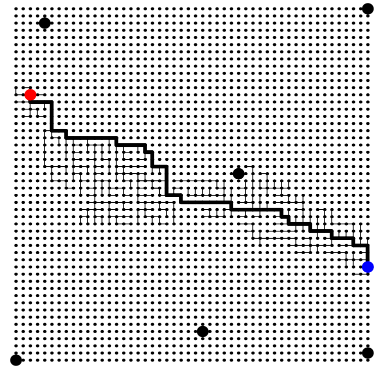
Fig. 4 – Random landmarks



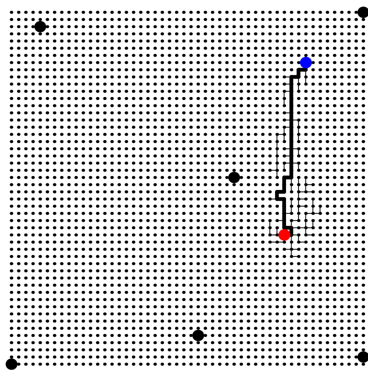
21, 24, 0.875



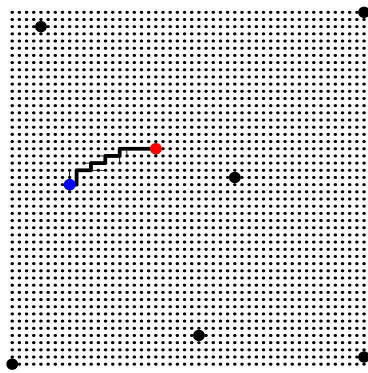
45, 58, 0.775862



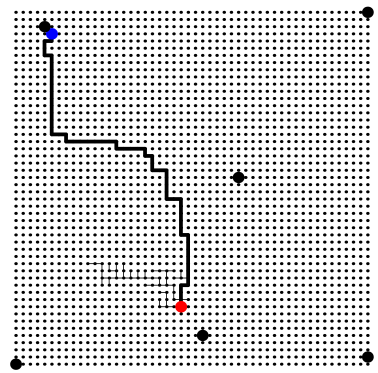
72, 424, 0.169811



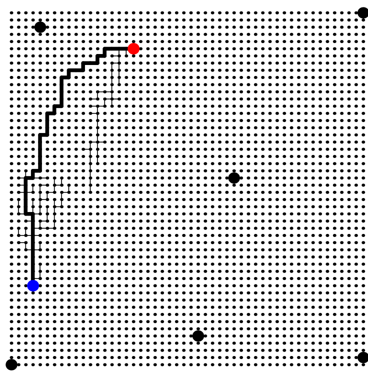
32, 120, 0.266667



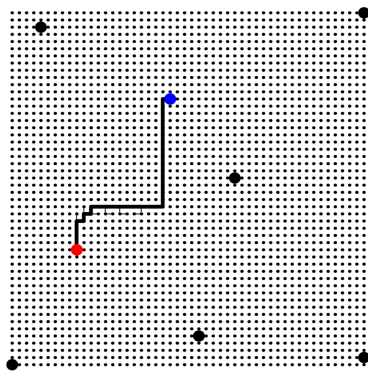
18, 32, 0.5625



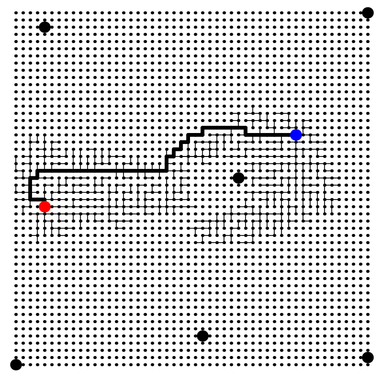
61, 108, 0.564815



50, 148, 0.337838



35, 44, 0.795455



52, 566, 0.0918728

Fig. 6 – Uniform landmarks