The $s - t$ shortest path problem: advanced algorithms

Combinatorial optimization

Giovanni Righini

University of Milan

The $s - t$ shortest path problem

Data:

- a digraph $\mathcal{D} = (\mathcal{N}, \mathcal{A})$;
- a cost function $c : \mathcal{A} \mapsto \Re_+$;
- two nodes s and t (origin and destination).

Problem: find a shortest (minimum cost) path from s to t.

Several applications require computation of $s - t$ shortest paths on very large weighted digraphs (millions of nodes and arcs) in almost real-time (milliseconds).

However the queries concern

- the same digraph
- with different s and t.

The idea is to precompute useful pieces of information that depend on the digraph, but not on s and t.

References

Main references:

- Landmarks: A.V. Goldberg, C. Harrelson, *Computing the Shortest Path: A^* Search Meets Graph Theory*, SODA, 2005;
- Reach: R. Gutman, *Reach-based routing: a new approach to shortest path algorithms optimized for road networks*, Workshop on Algorithm Engineering and Experiments (ALENEX), 2004.
- Reach: A.V. Goldberg, H. Kaplan, R.F. Werneck, *Reach for A^*: efficient point-to-point shortest path algorithms*, Workshop on Algorithm Engineering and Experiments (ALENEX), 2006.
- Highway hierarchies: P. Sanders, D. Schultes, *Highway Hierarchies Hasten Exact Shortest Path Queries*, Annual European Symposium on Algorithms, LNCS 3669 (568-579), 2005.

Potential functions and lower bounds

A potential function $\pi : \mathcal{N} \mapsto \Re$ is used to define the reduced costs:

$$c^{\pi}(i,j) = c(i,j) - \pi(i) + \pi(j).$$

A potential function π is feasible iff

$$c^{\pi}(i,j) \geq 0 \; \forall (i,j) \in \mathcal{A}.$$

Property. A shortest $s - t$ path with respect to c^{π} is a shortest $s - t$ path with respect to c.

If $\pi(t) \leq 0$ and π is feasible, then π is a lower bounding function i.e.

$$\pi(i) \leq dist(i,t) \; \forall i \in \mathcal{N},$$

where $dist(i,t)$ is the shortest path cost from i to t.

Property 1

Property 1. If π is a feasible potential function, then $p = \pi + K$ is also a feasible potential function for any constant K.

Proof.
Reduced costs do not change by adding K: $c^{\pi}(i,j) \geq 0$ implies $c^{p}(i,j) \geq 0 \ \forall (i,j) \in \mathcal{A}$.

Property 2

Property 2. If π_1 and π_2 are feasible potential (lower bounding) functions, then $p = \max\{\pi_1, \pi_2\}$ is a feasible potential (lower bounding) function.

Proof.
(i) Feasibility.
If $(\pi_1(i) \geq \pi_2(i)) \wedge (\pi_1(j) \geq \pi_2(j))$, then $p = \pi_1$ which is feasible.
If $(\pi_1(i) \leq \pi_2(i)) \wedge (\pi_1(j) \leq \pi_2(j))$, then $p = \pi_2$ which is feasible.
If $(\pi_1(i) \geq \pi_2(i)) \wedge (\pi_1(j) \leq \pi_2(j))$, then

$$c^p(i,j) = c(i,j) - \pi_1(i) + \pi_2(j) \geq c(i,j) - \pi_1(i) + \pi_1(j) = c^{\pi_1}(i,j) \geq 0.$$

If $(\pi_1(i) \leq \pi_2(i)) \wedge (\pi_1(j) \geq \pi_2(j))$, then

$$c^p(i,j) = c(i,j) - \pi_2(i) + \pi_1(j) \geq c(i,j) - \pi_2(i) + \pi_2(j) = c^{\pi_2}(i,j) \geq 0.$$

(ii) Lower bounding.
$\pi_1(t) \leq 0$ and $\pi_2(t) \leq 0$ imply $p(t) \leq 0$.

Dijkstra and A^*

Dijkstra algorithm scans the nodes according to the order of their labels $d(i)$, representing an upper bound to $dist(s, i)$.

The A^* algorithm scans the nodes according to the order of their labels $f(i) = d(i) + \pi(i)$, where $\pi(i)$ is a feasible lower bounding function: $\pi(i) \leq dist(i, t)$.

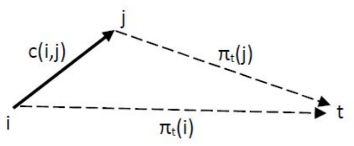
In both cases, when a node is scanned its label is made permanent and $d(i) = dist(s, i)$.

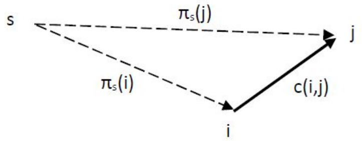
Running A^* is equivalent to running Dijkstra algorithm on the digraph weighted with the reduced costs c^{π}.

The effectiveness of A^* depends on how tight π is to $dist(i, t)$ for all nodes $i \in \mathcal{N}$.

Bi-directional A^*

In bi-directional A^* let π_t and π_s be the two lower bounding functions used in the forward and backward search, respectively.





The two lower bounding functions are consistent iff $\forall(i,j) \in \mathcal{A}$ the forward and backward reduced costs are the same:

$$c^{\pi_t}(i,j) = c(i,j) - \pi_t(i) + \pi_t(j) = c(i,j) - \pi_s(j) + \pi_s(i) = c^{\pi_s}(i,j).$$

This is equivalent to $\pi_t(i) + \pi_s(i) = K$ for some constant K.

One can use the same lower bounding algorithm (the best available one) to compute π_t and π_s separately. In this case the two lower bounding functions are symmetric.

Symmetric and consistent potentials

Let π_t and π_s be (tight but not consistent) lower bounding functions:

$$\pi_t(i) \leq dist(i,t) \quad \pi_s(i) \leq dist(s,i) \quad \forall i \in \mathcal{N}.$$

Two options for bi-directional A*:

- Symmetric algorithm. Use π_t and π_s independenty in forward search and backward search.
  Pro: one can use the tightest available lower bounds in each direction.
- Consistent algorithm. Combine π_t and π_s to obtain two consistent potentials.
  Pro: optimality is guaranteed as soon as the two searches meet.

Forward and backward labels

Forward labels $d_s(i)$ represent best incumbent distances from s to i.
Backward labels $d_t(i)$ represent best incumbent distances from i to t.

The node set $\mathcal{N}$ is subdivided into subsets

- E^{fw}: nodes with a permanent forward label $d_s(i) = dist(s,i)$;
- E^{bw}: nodes with a permanent backward label $d_t(i) = dist(i,t)$;
- O^{fw}: nodes with a temporary forward label $d_s(i) \geq dist(s,i)$;
- O^{bw}: nodes with a temporary backward label $d_t(i) \geq dist(i,t)$;
- other nodes, not yet reached in either direction.

Only O^{fw} and O^{bw} can intersect.

Labels in O^{fw} are sorted according to $f(i) = d_s(i) + \pi_t(i)$ in a heap F.
Labels in O^{bw} are sorted according to $b(i) = d_t(i) + \pi_s(i)$ in a heap B.

A best incumbent upper bound μ is possibly updated every time a new $s - t$ path is found.

Symmetric bi-directional A*

Pohl (1971), Kwa (1989). Using two independent lower bounds π_t and π_s, run the forward and backward searches, alternating in some way.

Each time a forward search scans an arc (i, j) s.t. $j \in E^{bw}$,

- do not give a forward label to j;
- if $d_s(i) + c(i, j) + d_t(j) < \mu$, then update μ.

Do the same symmetrically during backward search.

Termination. Stop as soon as one of these three conditions hold:

- forward search scans a node $i \in O^{fw}$ with $f(i) \geq \mu$;
- backward search scans a node $i \in O^{bw}$ with $b(i) \geq \mu$;
- one of the two searches has no nodes with temporary labels: $(O^{fw} = \emptyset) \vee (O^{bw} = \emptyset)$.

Consistent bi-directional A*

Ikeda et al. (1994).
Use the following average potential functions:

- $p_t(i) = \frac{\pi_t(i) - \pi_s(i)}{2}$ in forward search,
- $p_s(i) = \frac{\pi_s(i) - \pi_t(i)}{2}$ in backward search.

Consistent bi-directional A*

Observation.
A feasible forward lower bounding function is $p(i) = p_s(t) - p_s(i)$.

Proof.
(i) Feasibility.
$-p_s = p_t$ is a feasible forward lower bounding function. Adding a costant $p_s(t)$ does not affect the reduced costs: $c^{-p_s} \geq 0$ implies $c^p \geq 0$.
(ii) Lower bounding.
$p(t) = p_s(t) - p_s(t) = 0$.

Observation.
A feasible forward lower bounding function is $p_t(i) = \frac{\pi_t(i) - \pi_s(i) + \pi_s(t)}{2}$.
A feasible backward lower bounding function is $p_s(i) = \frac{\pi_s(i) - \pi_t(i) + \pi_t(s)}{2}$.

They are consistent: $p_t(i) + p_s(i) = (\pi_s(t) + \pi_t(s))/2 \; \forall i \in \mathcal{N}$.

Consistent lower bounds: termination

Stop condition for the bi-directional Dijkstra algorithm.

$$Top(F) + Top(B) \geq \mu.$$

Stop condition for the bi-directional consistent A* algorithm.

The same, but using the reduced costs:

$$Top(F) + Top(B) \geq \mu + (\pi_s(t) + \pi_t(s))/2.$$

Landmarks

Landmarks are a technique to compute lower bounds π owing to pre-computed shortest distances.

Consider a landmark L (typically, a node of the digraph) and let $dist(i, L)$ and $dist(L, i)$ be the shortest distances from $i \in \mathcal{N}$ to L and from L to $i \in \mathcal{N}$.

Then, by the triangle inequality,

$$dist(i, L) - dist(j, L) \leq dist(i, j) \quad dist(L, j) - dist(L, i) \leq dist(i, j) \quad \forall i, j \in \mathcal{N}.$$

This holds for any landmark L. Therefore one can select $\pi_t(i) = \max_L\{dist(i, L) - dist(t, L), dist(L, t) - dist(L, i)\}$ (and the same for π_s).

- Pre-compute shortest distances from/to several (e.g. 16) landmarks (independently of s and t).
- Given an (s, t) pair select some landmarks (e.g. 4) providing the largest lower bounds on $dist(s, t)$.

Landmarks selection

Landmark selection techniques to select k landmarks in a given digraph:

- Random: select k nodes at random in $\mathcal{N}$.
- Farthest: iteratively select the node that maximizes the minimum distance from/to all selected landmarks. Variant: consider the number of arcs, instead of the distance (run BFS instead of Dijkstra).
- Planar (for road networks): find a node c closest to the median of the graph. Partition the region into k sectors centered at c, each containing approximately the same number of nodes. For each sector, pick a node farthest away from c (in the sense of the number of arcs).
- Optimized farthest: repeatedly remove a landmark and replace it with the farthest one from the remaining set of landmarks.
- Optimized planar: repeatedly remove a landmark and replace it by the best landmark in a set of candidates. To rate the candidates, compute a score for each one, based on lower bounds tightness for some randomly chosen pairs of nodes.

- Avoid. Given a set S of already selected landmarks, compute a shortest-path tree T_r rooted at some node r.
  Then, for each $v \in \mathcal{N}$ compute its weight, defined as the difference between $dist(r,v)$ and the lower bound for $dist(r,v)$ given by S.
  For each $v \in \mathcal{N}$ compute its size $s(v)$, which depends on T_v, the subtree of T_r rooted at v.
  If T_v contains a landmark, then $s(v) = 0$; otherwise, $s(v)$ is the sum of the weights of all vertices in T_v.
  Let w be the vertex of maximum size. Traverse T_w, starting from w and always following the child with the largest size, until a leaf is reached.
  Make this leaf a new landmark.
  A natural way of selecting r is uniformly at random. Better results are obtained by selecting r with higher probability from the nodes that are far from S.

- Max cover. Define $\overline{c}^L(i,j) = c(i,j) - d(L,j) + d(L,i)$.
  If $\overline{c}^L(i,j) = 0$, then L *covers* (i,j).
  Define $Cost(S) = |\{(i,j) \in \mathcal{A} : \min_{L \in S}\{\overline{c}^L(i,j)\} > 0\}|$.
  Initialize a set C of k *candidate landmarks* by *Avoid*.
  Iteratively remove each landmark from C with probability $1/2$ and generate more landmarks (using *Avoid*) until they are k again.
  Add all newly generated landmarks to C.
  Repeat until either $|C| = 4k$ or *Avoid* is executed $5k$ times.
  Interpreting each landmark as the set of arcs that it covers, solve an instance of the maximum cover problem (NP-hard).
  Multistart heuristic: each iteration starts with a random subset S of C with k landmarks and runs a local search procedure.
  Return the best solution found after $\lfloor \log_2 k + 1 \rfloor$ iterations.
  Local search: iteratively replace a candidate landmark $u \in S$ with $v \in C \setminus S$. Among swaps with positive profit $Cost(S) - Cost(S \setminus \{u\} \cup \{v\})$, pick one at random with probability proportional to the profit. Stop when no improving swaps exist. Each local search iteration takes $O(km)$ time.

Active landmarks

Static landmarks. Select h landmarks providing the best lower bounds of the $s - t$ distance.

Trade-off between:

- number of labelled nodes,
- number of landmarks to be examined for each label extension.

Active landmarks

Dynamic landmarks. Initially select 2 landmarks L_1 and L_2, providing the best lower bounds of the $s - t$ distance to L_1 and from L_2.
The search reaches a *checkpoint* when the lower bound for completing the $s - t$ path from the current node is 90%, 80%, 70% and so on of the initial $s - t$ lower bound and at least 100 nodes have been labelled since the last checkpoint.
At a checkpoint at a node v, all landmarks are considered to test whether some of the inactive landmarks provide a lower bound from node v that is larger than $1 + \epsilon$ times the current lower bound (e.g. $\epsilon = 0.01$).
If this is the case, the new landmark is made active (at most 6 active landmarks are accepted) and the potentials are updated.

When p_t and p_s are updated because the active landmarks have been updated, the keys of all labeled vertices are updated and the heaps are updated. This takes $O(|F| + |B|)$ time.

Bounding

Consider a forward iteration in which A^* scans a permanently labelled node i (the same holds symmetrically for backward iterations). Consider one of the outgoing arcs, (i,j). The algorithm should check whether $d_s(i) + c(i,j) < d_s(j)$. If so, $d_s(j)$ is updated in the forward priority queue.

Using lower bounds, the algorithm also checks if $d_s(i) + c(i,j) + \pi_t(j) < \mu$, where π_t is a feasible forward lower bounding function. When the test fails, the shortest $s - t$ path through (i,j) cannot improve upon the current shortest path. Therefore, there is no need to store an updated value of $d_s(j)$.

The lower bound functions π_t and π_s used for bounding in either direction do not need to be consistent.

Reach

Given a shortest path $P^*(u,v)$ from $u \in \mathcal{N}$ to $v \in \mathcal{N}$ and given a node $i \in P^*(u,v)$,

$$r(i, P^*(u,v)) = \min\{dist(u,i), dist(i,v)\},$$

where *dist* indicates the shortest path distance.

On the whole graph

$$r(i) = \max_{u \in \mathcal{N}, v \in N : u \neq v} \{r(i, P^*(u,v))\}.$$

Intuitively, the reach r of a node is a measure of how likely the node is to belong to long shortest paths.

The use of reach

Let $\overline{r}(i)$ be an upper bound: $\overline{r}(i) \geq r(i)$.

Let $\underline{d}(i,j)$ be a lower bound: $\underline{d}(i,j) \leq dist(i,j)$.

By definition

$$i \in P^*(s,t) \Longrightarrow r(i) \geq r(i, P^*(s,t)) = \min\{dist(s,i), dist(i,t)\}.$$

Therefore

$$\overline{r}(i) < \min\{\underline{d}(s,i), \underline{d}(i,t)\} \Longrightarrow i \notin P^*(s,t).$$

This allows to neglect many nodes (with small reach value) while running Dijkstra algorithm or A^*.

The use of reach

When we consider j as a successor of i in a labeling algorithm (Dijkstra, A^*), we already know $dist(s,i)$.

The following test is done before possibly updating the label of node j (*early pruning*):

$$\bar{r}(j) < \min\{dist(s,i) + c(i,j), \underline{d}(j,t)\} \text{ implies } (i,j) \notin P^*(s,t).$$

A lower bound $\underline{d}(j,t)$ can be provided

- by the Euclidean distance between j and t, if the nodes are embedded in a plane;
- by the largest permanent label in the reverse direction, if bi-directional search is used.

Bi-directional bounding

In bi-directional search, let γ^{bw} the minimum cost of non-permanent backward labels (labels in O^{bw}).

Consider the iteration in which node $i \in O^{fw}$ is selected for being permanently labelled. If

$$\bar{r}(i) < \min\{dist(s,i), \gamma^{bw}\},$$

then we can prune the search at i (*bi-directional bound*).

Self-bounding

Alternatively (*self-bounding*) we can prune node i checking whether

$$\bar{r}(i) < dist(s, i)$$

and we stop the search in a direction when

- O in that direction is empty,
- or the minimum distance label in O is at least half of μ,

where μ is an upper bound (best incumbent $s - t$ path).

It is advisable to scan the minimum label among the forward and the backward candidates.

Each node i can be inserted in E^{fw} (E^{bw}) only if $dist(s, i) \leq (\geq) dist(i, t)$.

Arc sorting

Arc sorting: sort the outstars (in-stars) by non-increasing value of (estimated) *reach* of the head (tail) node.

If $\bar{r}(j) < \min\{dist(s,i), \gamma\}$, all the arcs after (i,j) in the out-star of i can be safely skipped.

Hence, arc sorting may allow to neglect some arcs.

Computing *reach* exactly

To compute the *reach* values exactly:

- Set all reaches to ∞.
- Compute all-pairs shortest paths.
- For each $s - t$ shortest path:
  - Compute the reach of all nodes along the path.
  - Possibly update the reach of each node with the new value, if it is smaller.

Complexity: $O(nm)$, impractical for large graphs (even if sparse).

Computing *reach* approximately

Three main ideas are combined:

- partial trees
- iterative node deletion
- shortcuts

Preprocessing works in two phases:

- Main phase:
  - shortcut arcs are added;
  - partial trees are grown and low reach nodes are deleted;
- Refinement phase: upper bounds on reaches are re-evaulated and possibly strengthened.

Main phase

Main phase:

- Add shortcuts
- For each iteration *k*
  - Select a threshold value ϵ_k
  - Grow partial trees depending on ϵ_k
  - Eliminate nodes with reach less than ϵ_k
  - Add shortcuts

The threshold values are computed as $\epsilon_k = \alpha\epsilon_{k-1}$ for some $\alpha > 1$.

Canonical paths

Gutman (2004) observed that if more than one shortest path exists from s to t, only one is included in the partial trees. Therefore all nodes along alternative shortest paths may not appear in the partial tree. Therefore they can be misclassified as “low reach nodes” even if they are “high reach nodes” and they can receive an incorrect upper bound $\overline{r}(i)$.

However, this incorrect upper bounding does not prevent a shortest path algorithm like Dijkstra or A^* to find *at least one* shortest $s-t$ path.

Canonical paths

Goldberg et al. (2006) introduced the notion of *canonical path*, i.e. a shortest path with the additional property of being unique for each $s - t$ pair.
A small random perturbation is computed for each arc cost.
The perturbation of a path cost is the sum of the perturbations of its arcs.
When two or more shortest paths exist between s and t, the canonical one is the path with minimum perturbed cost.

Partial trees

For each node i compute a partial shortest path arborescence $T^{\epsilon}(i)$ rooted at i (with Dijkstra algorithm).
At a generic iteration the arborescence T of the labelled nodes includes an arborescence $\overline{T}$ of nodes with permanent label.

Stop criterion: for all leaves j of $\overline{T}$

- either j is a leaf of T,
- or $dist(i', j) \geq 2\epsilon$,

where i' is the node next to i in the (i, j) shortest path.

Let $T^{\epsilon}(i)$ be the partial tree $\overline{T}$ when the algorithm stops.
Nodes with reach larger than ϵ in $T^{\epsilon}(i)$ are marked as “high reach nodes”.
Repeating this procedure for all roots $i \in \mathcal{N}$ allows to partition nodes with reach larger than ϵ from nodes with reach smaller than ϵ.

Proof

Thesis 1. Nodes with reach less than ϵ cannot be marked from any root i.
Proof. Their reach in $T^{\epsilon}(i)$ cannot be larger than their actual reach in the digraph.

Thesis 2. All nodes k with $r(k) \geq \epsilon$ are guaranteed to be identified as "high reach nodes" in at least one partial arborescence $T^{\epsilon}(i)$ for some $i \in \mathcal{N}$.
Proof. If $r(k) \geq \epsilon$, then $\exists$ a path P in which k has reach at least ϵ.
Then, $\exists$ a *minimal* canonical path P' in P, in which k has reach at least ϵ.
Let x and y be the first and the last node of P'.

Proof

Consider $T^{\epsilon}(x)$, which contains $\overline{T}^{\epsilon}(x)$.
Owing to the uniqueness of (perturbed) shortest paths, two cases can occur:

- Case 1: P' is completely contained in $\overline{T}^{\epsilon}(x)$;
- Case 2: P' is partially contained in $\overline{T}^{\epsilon}(x)$.

Case 1.
In this case k is identified as a “high reach node” in $\overline{T}^{\epsilon}(x)$.

Proof

Case 2.
$\overline{T}^{\epsilon}(x)$ contains a subpath of P', starting at x and ending at a leaf z.

By definition of reach, $r(k) \geq \epsilon \Longrightarrow dist(x,k) \geq \epsilon$.

Let x' be the node next to x along P'.

Since P' is minimal, $dist(x',k) < \epsilon$.

Node z cannot be a leaf of $T^{\epsilon}(x)$, because

- it belongs to $\overline{T}^{\epsilon}(x)$ (it has a permanent label) and
- it has got at least one successor (the next node along P').

Hence z being a leaf of $\overline{T}^{\epsilon}(x)$ implies $dist(x',z) \geq 2\epsilon$.

$$dist(k,z) = dist(x',z) - dist(x',k) > 2\epsilon - \epsilon = \epsilon.$$

Therefore $\min\{dist(x,k), dist(k,z)\} \geq \epsilon$ and k is marked as a "high reach node" in $\overline{T}^{\epsilon}(x)$.

Long arcs

Assume all arc costs are integer.

Consider the case when an arc (x, y) adjacent to the root x has a cost equal to $M\epsilon$ for some large M.

Then $\overline{T}^{\epsilon}(x)$ will extend up to the successors of y, at a distance at least 2ϵ from y, i.e. at a distance at least $(M+2)\epsilon$ from x.

Therefore the algorithm cannot stop until all nodes within a distance $(M+2)\epsilon$ have been permanently labelled.

Solution: smaller trees are built, with the drawback that some low reach nodes can be misclassified as high reach nodes.

This produces weaker upper bounds, but does not affect the correctness of the $s-t$ shortest path algorithm.

Smaller trees

Let

- x be the root of the shortest path arborescence $T^{\epsilon}(x)$;
- $k \neq x$ a node in $T^{\epsilon}(x)$;
- $f(k)$ the successor of x along the shortest path from x to k.

The set of *inner nodes* of $T^{\epsilon}(x)$ is

$$I^{\epsilon}(x) = \{x\} \cup \{k \in T^{\epsilon}(x) : (k \neq x) \wedge (dist(f(k), k) \leq \epsilon)\}.$$

The set of *outer nodes* of $T^{\epsilon}(x)$ is its complement.

$$O^{\epsilon}(x) = T^{\epsilon}(x) \setminus I^{\epsilon}(x).$$

Smaller trees

For any outer node $w \in O^\epsilon(x)$, its distance from $I^\epsilon(x)$ is defined as

$$\min_{v \in I^\epsilon(x)} \{dist(v, w)\}.$$

The algorithm stops growing the shortest paths arborescence when

- all nodes with non-permanent labels are outer nodes, and
- they have distance at least ϵ from $I^\epsilon(x)$.

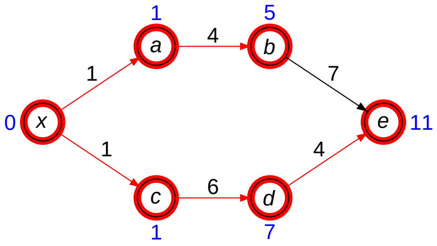


Figure: Shortest path tree T^ϵ for $\epsilon = 5$. Red: permanent labels. $r(b) = 0$.

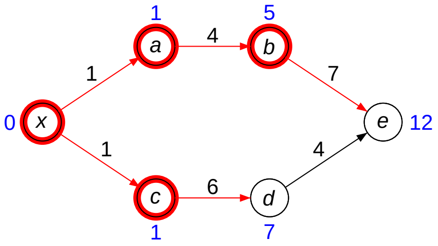


Figure: Smaller tree T^ϵ for $\epsilon = 5$. Red: permanent labels. $r(b) = 5$.

Arc reaches

Let (u, v) be an arc along an $s - t$ shortest path $P^*(s, t)$.

Then $r(u, v, P^*(s, t)) = \min\{dist(s, v), dist(u, t)\}$.

On the whole graph $r(u, v) = \max_{s \in \mathcal{N}, t \in N}\{r(u, v, P^*(s, t))\}$.

Node reaches can be computed from arc reaches:
$r(i) = \max\{\max_{(i,j) \in \mathcal{A}}\{r(i, j)\}, \max_{(j,i) \in \mathcal{A}}\{r(j, i)\}\}$.

The reach of an arc can be smaller than the reaches of its endpoints.

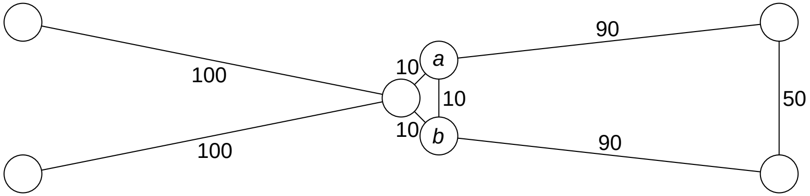


Figure: $r(a) = r(b) = 90$. $r(a, b) = 10$.

Penalties

When an arc is identified as a “low reach arc” and its reach is bounded above by the current ϵ, in the next iteration the arc is deleted and replaced by a penalty, representing upper bounds on the effect of the deleted arc on the reach of its endpoints.

Let A_k be the set of arcs remaining (not yet upper bounded) at iteration k.

In-penalties π^- and out-penalties π^+ are defined as follows for all nodes that are endpoints of deleted (low-reach) arcs:

$$\pi^-(i) = \max_{(j,i)\in\mathcal{A}^+:(j,i)\notin A_k} \{\overline{r}(j,i)\}$$

$$\pi^+(i) = \max_{(i,j)\in\mathcal{A}^+:(i,j)\notin A_k} \{\overline{r}(i,j)\},$$

where $\mathcal{A}^+$ includes both $\mathcal{A}$ and the shortcut arcs added in previous iterations. The definition of reach is generalized as follows:

$$r(u,v,P^*(s,t)) = \min\{dist(s,v) + \pi^-(v), dist(u,t) + \pi^+(u)\}.$$

Penalties

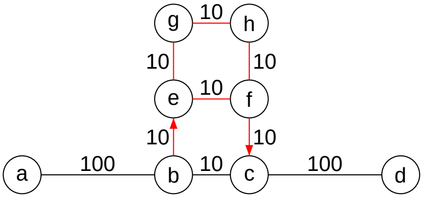


Figure: Red arcs have (low) reach $\leq$ 30.

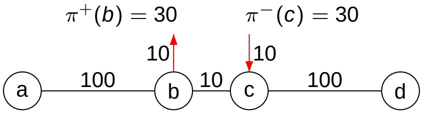


Figure: Low reach arcs replaced by penalties when growing partial trees.

Shortcuts

A node j is *by-passable* if

- it has only one incoming arc (i, j) and one outgoing arc (j, k) (one-way by-passable), or
- it has only two incoming arcs (i, j) and (k, j) and only two outgoing arcs (j, i) and (j, k) (two-ways by-passable).

A *line* is a path of at least three nodes, where all nodes different from the endpoints are by-passable.

Lines can be one-way or two-ways.

A by-pass is an arc directly connecting the endpoints of a line.

By-passes can be one-way or two-ways.

Cost and perturbation of shortcut arcs are given by the sum of costs and perturbation of the by-passed arcs.

Shortcuts

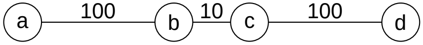
By-passed nodes are no longer visited in shortest paths containing their by-passed line.
Therefore shortcuts reduce the reaches of by-passed nodes.

If a line (s, t) has more than two arcs,

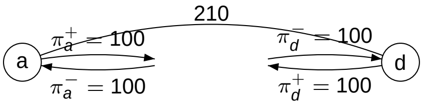
- find the node k in it, that minimizes $|dist(s, k) - dist(k, t)|$ (median node);
- add a shortcut (s, t) (if it is not in the current arc set $\mathcal{A}^+$);
- recursively do the same on each subpath (s, k) and (k, t).

Shortcuts

When a node is by-passed, it is deleted and replaced by a penalty assigned to its neighbors.



A two-ways line with three arcs.



The by-passed line replaced by π.

To avoid long shortcuts that would imply large partial trees, the maximum length of shortcuts is limited to $\frac{\epsilon_{k+1}}{2}$ at each iteration k.

Shortcuts

Given a one-way line (u, v, w), when a shortcut (u, w) is added, arc (u, v) will never be used on any shortest path that goes through u and w anymore.

Any shortest path traversing (u, v) will end either in v or in some low-reach area neighboring v.

Therefore, a valid upper bound for the reach of (u, v) is $\bar{r}(u, v) = c_{uv} + \pi^+(v)$ (and the same holds for (v, w) symmetrically).

Owing to these upper bounds, one can immediately remove v, (u, v) and (v, w) from the graph and update the appropriate penalties.

A similar procedure can be adopted for two-way lines.

Refinement phase

The use of penalties makes the upper bounds looser and looser as the algorithm progresses.

This is more evident on nodes with larger reaches.

Therefore the reaches are re-computed in a more accurate way for the δ nodes with highest reaches, where $\delta = \lceil 10\sqrt{n} \rceil$.

Let V_δ the set of such nodes and G_δ the subgraph induced by V_δ.

A complete shortest path arborescence is computed from each node in G_δ, using penalties to account for missing nodes.

Parameter tuning

Select $k = \min\{500, \lfloor \lceil \sqrt{n} \rceil / 3 \rfloor\}$ nodes at random.

Grow a partial shortest path arborescence until $\lfloor n/k \rfloor$ nodes are permanently labeled.

For each root consider the radius, i.e. the distance of the last label.

Set ϵ_1 to twice the minimum among the k radii.

Parameter tuning

We also have to choose a multiplier α to compute $\epsilon_i = \alpha^{i-1}\epsilon_1$ at each iteration.

- **Running time:** the smaller α is, the more iterations will be done; but if α is large, iterations will take longer (since vertices are eliminated less frequently).
- **Number of shortcuts:** if α is relatively small, the algorithm has a better chance of shortcutting vertices before they are eliminated.
- **Upper bounds:** the error in an arc reach estimate at iteration *i* depends on the penalties, which in turn depend on the maximum reaches of arcs eliminated in previous iterations; the larger α is, the smaller the sum $\sum_{j<i} \epsilon_j$ compared to ϵ_i.

Heuristic rule: keep $\alpha = 3$ while the number of nodes remains larger than δ. Then reduce it to $\alpha = 1.5$.

Combining reaches with A^*

In A^* each node i has a label $k(i) = d(i) + \pi(i)$, where $d(i)$ is the distance from s and $\pi(i)$ is a lower bound on the distance to t.

In bi-directional A^* each node has two labels, $f(i) = d^s(i) + \pi^t(i)$ (forward) and $b(i) = d^t(i) + \pi^s(i)$ (backward).

When A^* is about to make the forward label of node i permanent, it checks the reach of i: if

$$r(i) < \min\{d_s(i), \pi^t(i)\},$$

then i is pruned.

The stop criterion (lower bound = upper bound) is not affected.

Highway hierarchies: definitions

In a graph, let the s-rank of a vertex i, $r_s(i)$ the position of vertex i in the list of the vertices permanently labeled by Dijkstra algorithm running from s ($r_s(s) = 0$).
A suitable tie-breaking rule must be defined to ensure that the s-rank of each vertex is unique from each s (canonical shortest paths).

For any given vertex s, the distance of the H-closest vertex from s is denoted by $d_H(s)$: $d_H(s) = dist(s, v)$, where $r_s(v) = H$.

The H-neighbourhood $N_H(s)$ of s is
$N_(s) = \{i \in \mathcal{N} : dist(s, i) \leq d_H(s)\}$.

On digraphs, symmetrical definitions (distances to t) apply.

Highway hierarchies: definitions

Definition (highway network). For a given value of the parameter H, the highway network $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ of a graph $\mathcal{G}$ is defined as the set of edges $[u, v] \in \mathcal{E}$ that appear in at least one canonical $(s - t)$ shortest path $(s, \ldots, u, v, \ldots, t)$ with the property that $v \notin N_H(s)$ and $u \notin N_H(t)$.

The set $\mathcal{V}_1$ is the maximal subset of $\mathcal{V}$ such that $\mathcal{G}_1$ contains no isolated vertices.

Definition (2-core). The 2-core of a graph is the maximal subgraph with minimum degree two. A graph consists of its 2-core and attached trees, i.e., trees such that their roots belong to the 2-core, but all other nodes do not.

Definition (line). A line in a graph is a path $(u_0, u_1, \ldots, u_k)$, where the inner vertices have degree two.

Highway hierarchies: definitions

Definition (Contracted highway network). From the highway network $\mathcal{G}_1$ of a graph $\mathcal{G}$, the contracted highway network $\mathcal{G}'_1$ of $\mathcal{G}$ is obtained by taking the 2-core of $\mathcal{G}_1$, removing the inner vertices of all lines and replacing each line by a single edge between its endpoints.

Thus, the highway network $\mathcal{G}_1$ consists of the contracted highway network $\mathcal{G}'_1$ and some additional components (trees or lines).

The highway network can be contracted in time $O(m + n)$.

Definition (highway hierarchy). The highway hierarchy of a graph $\mathcal{G}$ is obtained by applying this contraction iteratively.

The algorithm

Pre-processing.
For each vertex $s \in \mathcal{V}$, $d_H(s)$ is computed by growing a shortest path tree from s with Dijkstra algorithm and stopping it as soon as H nodes have received a permanent label.

Construction.
We start with an empty set of highway edges E_1.

For each vertex s, two operations are executed:

- forward construction of a partial shortest path tree B;
- backward evaluation of B.

Forward construction

A Dijkstra search from s is executed.

During the search, a reached vertex is either active or passive.

The source node s is active.

Each vertex that is reached for the first time (*Insert*) and each reached vertex that is updated (*DecreaseKey*) adopts the same activation state from its (tentative) predecessor in the shortest path tree B.

When a vertex p is made permanent using the path $(s, u_1, \ldots, u_k, p)$, then the state of p is set to passive if $|N_H(u_1) \cap N_H(p)| \leq 1$ (i.e. p is "far enough" from u_1).

When no active vertex is left in the priority queue (set of "open" vertices), the growth of B stops.

Backward evaluation: selection of highway edges

All edges $[u, v]$ are added to $\mathcal{E}_1$ if they lie on paths $(s, \ldots, u, v, \ldots, t)$ in B with the property that $v \notin N_H(s)$ and $u \notin N_H(t)$, where t is a leaf of B.

This can be done in time $O(|B|)$ for each source node s.

To speed up the construction, an active vertex v is declared to be a maverick if $dist(s, v) > f\ d_H(s)$, where f is a parameter.

When all active vertices are mavericks, the search from passive vertices is no longer continued.

The pre-processing is accelerated, $\mathcal{E}_1$ becomes a superset of the highway network, queries will be slower, but still exact.

The maverick factor f allows to adjust the trade-off between pre-processing time and query time.

Hierarchy

The highway hierarchy of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of the graphs $\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_L$, arranged in $L + 1$ levels.

For each node $v \in \mathcal{V}$ and each level ℓ such that $v \in \mathcal{V}_\ell$, there is a copy v_ℓ of vertex v in level ℓ.

In the same way, there are several copies of an edge $[u, v]$ when both u and v belong to more than one common level.

Edges between vertices in the same level are horizontal edges.

Additionally, the hierarchy contains an edge $[v_\ell, v_{\ell+1}]$ for each pair of vertices $v_\ell \in \mathcal{V}_\ell$ and $v_{\ell+1} \in \mathcal{V}_{\ell+1}$ that are copies of the same vertex v in consecutive levels.

These additional edges are called vertical edges and have zero cost.

Hierarchy

For each vertex v, the radius $d_H^\ell(v)$ is computed in all levels: for each level $\ell < L$, $d_H^\ell(v)$ is the distance from v to the H-closest vertex in $\mathcal{G}'_\ell$.

If v does not belong to $\mathcal{G}'_\ell$ then $d_H^\ell(v)$ is set to ∞.

In the last level, $d_H^L(v)$ is set to ∞ for all vertices.

Neighborhoods $N_H^\ell(v)$ are also computed for all vertices and levels: $N_H^\ell(v) = \{v' \in \mathcal{V}'_\ell : dist(v, v') \leq d_H^\ell(v)\}$ is the neighbourhood of v in $\mathcal{G}'_\ell$.

Remark. The neighourhood of a vertex belonging to a component (tree or line) contains all the vertices of the corresponding level. The same holds for to $N_H^L(v)$, for any v.

Query

The multi-level query algorithm is a slight modification of bi-directional Dijkstra algorithm on the hierarchy graph.

The endpoints s and t are s_0 and t_0 at level 0.

Restriction 1. In each level ℓ, no horizontal edge is used that would leave the neighbourhood $N^\ell(v^*)$ of the corresponding entrance point v^*.

Entrance points in level ℓ:

- vertices s_ℓ and t_ℓ;
- any vertex of the core, permanently labeled from a horizontal predecessor out of the core;
- any vertex permanently labeled from a vertical predecessor.

A corresponding entrance point for a permanently labeled vertex v is the last entrance point along the path to v.

Query

Restriction 2. Components (trees and lines) are never entered using a horizontal edge.

An edge $[u, v]$ enters a component if either u belongs to the core and v does not or u belongs to a line and v to an attached tree.
Any edge from an attached tree to a line leaves the attached tree and therefore it does not rank among the edges that enter a component.

Remark. The endpoint(s) of a component do not belong to the component but to the core (or to the line in case of the root of a tree that is attached to a line).

When restriction 1 applies, the search continues on the next level.
Horizontal edges that cannot be used in one direction owing to restriction 2, can be used in the opposite direction.

Collapsing the hierarchy in a single level

It is not needed to explicitly represent all levels of the hierarchy.

It is sufficient that at most one copy of each vertex is reached horizontally: the copy with the smallest label and, in case of ties, the one on the lowest level.

Therefore it is enough to store the original graph with some additional pieces of information:

- each edge $[u, v]$ is assigned a maximum level $\lambda(u, v)$, i.e. it belongs to $\mathcal{G}_\ell \ \forall \ell = 0, \dots, \lambda(u, v)$;
- each vertex v is assigned to at most one component $c(v)$;
- each component belongs to the level ℓ of its inner edges;
- the value $d_H^\ell(v)$ is stored only if $v \in \mathcal{G}'_\ell$.

Stop criterion

The algorithm cannot stop when a vertex v is permanently labelled forward and backward: there is no guarantee that all vertices within a given distance have been already labeled (as in bi-directional Dijkstra algorithm).

Let $\mathcal{E}_s$ and $\mathcal{E}_t$ be the sets of horizontal edges that have been skipped during the search from s and t respectively.

When both search scopes meet, the algorithm can stop as soon as the search from t has finished searching level $\hat{\ell}_s = \max_{e \in \mathcal{E}_s} \{\lambda(e)\}$ and the search from s has finished searching level $\hat{\ell}_t = \max_{e \in \mathcal{E}_t} \{\lambda(e)\}$.

A level ℓ is finished when there are no open vertices in it or below. If the level ℓ is finished, edges e in levels lower than ℓ cannot be used any longer. Hence, when the search from t has finished searching level $\hat{\ell}_s$, it is guaranteed that no edge e in level $\ell \leq \hat{\ell}_s$ would be used even if the algorithm could go on.

Reach and Highway hierarchies

Define c-reach (cardinality reach) of a vertex.

Given v along a shortest $s-t$ path, $P^*(s,t)$, grow equal cardinality balls around s and t until v is included in one of them.
Let $c_{P^*(s,t)}(v)$ the cardinality of the two balls at that point.

$$c(v) = \max_{(s,t):v\in P^*(s,t)} \{c_{P^*(s,t)}(v)\}.$$

For a vertex v and a non-negative integer k, let $\rho(v,k)$ be the radius of the smallest ball centered at v that contains k vertices.

When searching for a shortest $s-t$ path we do not need to scan v if

$$\rho(s,c(v)) < dist(s,v) \wedge \rho(t,c(v)) < dist(v,t).$$

This would require keeping $n-1$ values of ρ for each vertex.

Reach and Highway hierarchies

The partial tree algorithm is used for c-reaches instead of reaches.

Given a threshold H, the algorithm identifies vertices with c-reach below H (local vertices).

Consider a bi-directional search. During the search from s, once the search radius advances past $\rho(s, H)$, one can prune local vertices (and the same backward).

This idea is applied recursively to the graph with low c-reach vertices deleted.

This gives a hierarchy of vertices, in which each vertex needs to store a ρ value for each level of the hierarchy it belongs to.