CS 163 & CS 265: Graph Algorithms

Week 3: Shortest paths

Lecture 3c: Changing weights without changing the shortest paths

David Eppstein
University of California, Irvine

Winter Quarter, 2024

Reweighting

Intuition

Shortest path problems have lengths on their edges

What if we also have heights on the vertices?

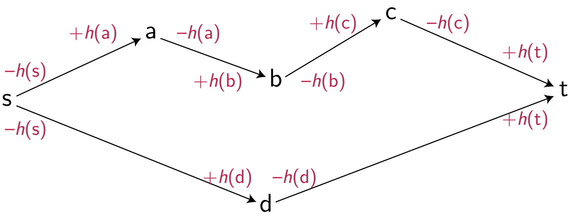
- Easier to travel downhill: makes those edges appear shorter
- Harder to travel uphill: makes those edges appear longer
- Height difference from start to goal doesn't depend on the path you use to get there

opentopomap.org

Reweighting

Suppose we have a directed graph G with edge lengths $\ell(u \to v)$
and vertex heights $h(v)$ (maybe input, maybe calculated)

Define new lengths $\ell_h(u \to v) = \ell(u \to v) - h(u) + h(v)$
Decreases length of downhill edges $\big(h(u) > h(v)\big)$
Increases length of uphill edges $\big(h(u) < h(v)\big)$



Same transformation happens to lengths of all paths!
Subtract h(start of path), add h(end of path)

If it doesn't change shortest paths, what good is it?

Changes how algorithms compute the paths

- Carefully chosen heights can eliminate negative edges
  - Cannot eliminate negative cycles
  - Allows Dijkstra to work, faster than Bellman–Ford
  - Johnson's algorithm, paths between all pairs of vertices
  - Suurballe's algorithm, two non-overlapping paths
- Can change order of computation
  - Combines well with easy optimization in Dijkstra: stop as soon as destination vertex has been found instead of computing distances to all remaining vertices
  - Dijkstra + reweighting is called A*

All-pairs shortest paths

What we already know

All three algorithms from last time compute distances and paths from a single source vertex to all of the other vertices

We can find distances between all pairs of vertices by running the same algorithm repeatedly for all possible starting vertices

Directed acyclic graphs: linear-time per vertex, total $O(mn)$

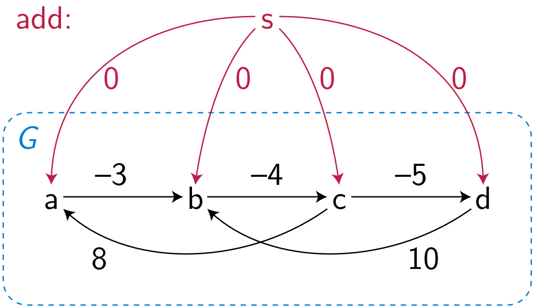
Positive (or non-negative) edge lengths: Dijkstra once per vertex, $O(mn + n^2 \log n)$ for Fibonacci-heap version

Graphs with negative edges but no negative cycles: run Bellman–Ford once per vertex, $O(mn^2)$

Johnson's algorithm, step 1

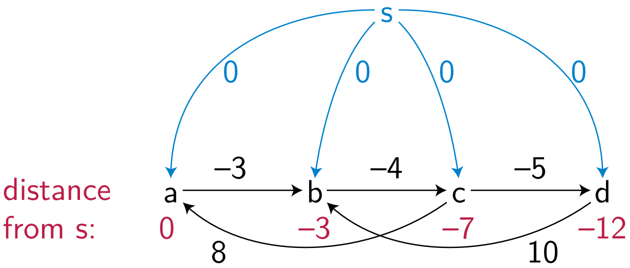
Given an input graph G with negative edges (but no negative cycles) in which we want all-pairs shortest paths

Add a new extra vertex s to G,
with zero-length edges from s to all other vertices



Johnson's algorithm, step 2

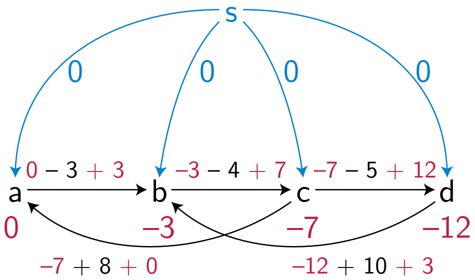
Use Bellman-Ford only once,
to compute shortest paths from s to all other vertices



Distances will all be either zero or negative

Johnson's algorithm, step 3

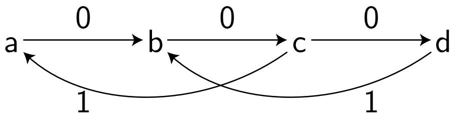
Reweight by absolute value of distance from s (zero or positive)



Changes edge lengths but doesn't change which paths are shortest

Johnson's algorithm, step 4

Use Dijkstra once from each starting vertex in reweighted graph



For every edge $u \to v$ with original length $\ell(u \to v)$,

$$\text{distance}(s,u) + \ell(u \to v) \geq \text{distance}(s,v)$$

(path s to u + edge $u \to v$ is path to v; all paths $\geq$ shortest path)

$$\iff \quad \ell_h(u \to v) = \text{distance}(s,u) + \ell(u \to v) - \text{distance}(s,v) \geq 0$$

(all reweighted edge lengths are non-negative)

Johnson's algorithm analysis

We run Bellman–Ford once, in time $O(mn)$

Reweighting takes linear time

We run Dijkstra n times, total $O(mn + n^2 \log n)$

Grand total $O(mn + n^2 \log n)$

[Johnson 1977]

Two disjoint paths

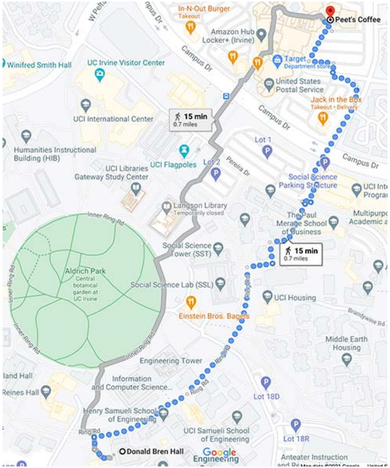
The two-paths problem

Input: A weighted directed graph, starting vertex s, destination vertex t

Goals:

- Find two paths from s to t
- Paths can repeat vertices but cannot share edges

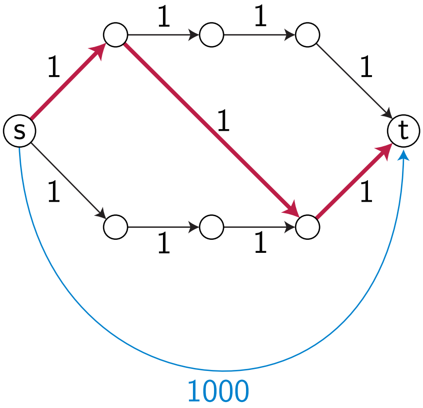
  (Can modify algorithm to forbid shared vertices, more complicated)
- Minimize total length of both, added together



An obvious attempt that fails

Find a shortest path, delete it from the graph, and find a second shortest path

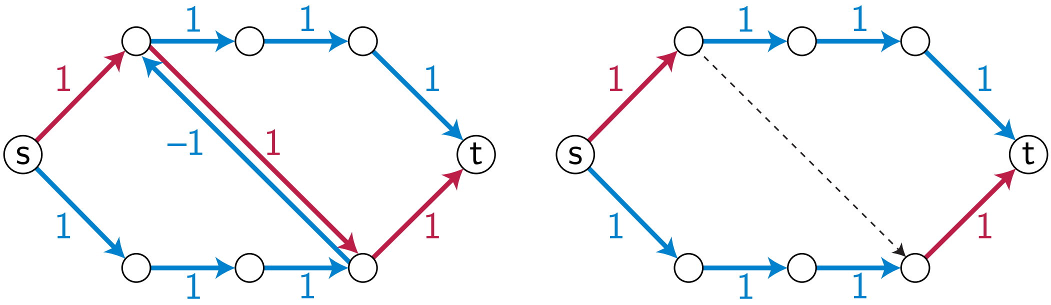
- Second path might not exist even when two paths exist
- If it succeeds, might find non-optimal solution



Allowing the second path to backtrack

After finding a shortest path, allow the second path to follow its edges backwards

Backwards edges cancel forward ones so they have negative cost

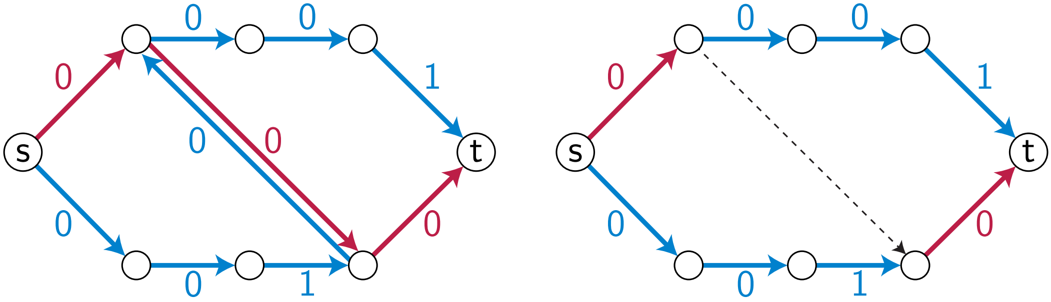


Second idea: avoid negative edges by reweighting before reversing

Suurballe's algorithm

Assuming all input edges are non-negative:

- Use Dijkstra to find single-source paths and distances from s
- Reweight using (negated) distance from s, same as Johnson's algorithm — edges in shortest paths get zero, otherwise > 0
- Find shortest path P from s to t, and reverse its edges
- Find a second shortest path from s to t in the modified graph
- Cancel edges used both ways and reconnect remaining paths



Suurballe analysis

Two runs of Dijkstra

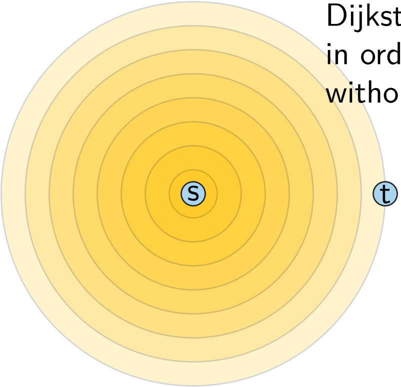
Linear time for reweighting and reversing

Total time $O(m + n \log n)$

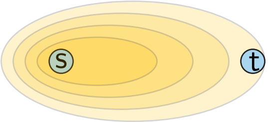
[Suurballe 1974; Suurballe and Tarjan 1984]

A*

Intuition



Dijkstra: explore in all directions
in order by distance from s,
without regard to the destination



Better: prioritize vertices
that appear to lead towards
the eventual destination

How? Reweight to make edges towards t shorter,
and edges away from t longer

A* algorithm

Choose a height function for each vertex v that is easy to calculate and estimates distance(v, t) as accurately as we can

- E.g., straight-line distance instead of road-network distance
- Must be underestimate so paths to t eventually go downhill
- Should not decrease too quickly: for each edge $u \to v$, height at v can be smaller than height at u by at most $\ell(u \to v)$

Reweight by height function

- All edge weights non-negative (by not-too-quick decrease)
- Edges that lead to smaller estimates appear shorter

Run Dijkstra on reweighted graph, stopping when destination found

Alternative description of A*

Instead of changing the edge weights, change the prioritization

$$\begin{aligned}\text{Priority of a vertex} &= \text{current tentative distance}\\ &\quad + \text{estimated distance to goal}\end{aligned}$$

[Hart et al. 1968]

It's the same algorithm!

It chooses the same vertices in exactly the same order

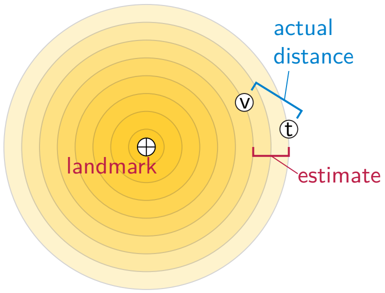
Landmark-based distance estimation

Preprocessing (before handling any distance queries):

- Choose a small set of "landmark" vertices
- Use Dijkstra to compute distances $L_i(v)$ from ith landmark to each vertex v

To compute distance between two vertices s and t:

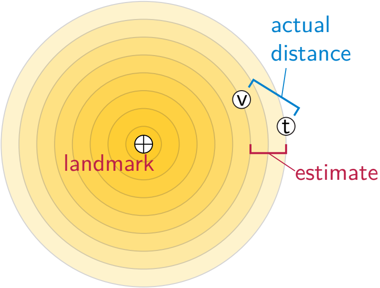
- Estimate distance$(v, t) \approx \max_i L_i(t) - L_i(v)$
- Use this estimate for A*



[Goldberg and Harrelson 2005]

Why does this work?

Estimate: $\mathrm{distance}(v, t) \approx \max_i L_i(t) - L_i(v)$



- Underestimate of true distance (triangle ineq.)
- Max in formula chooses the most accurate landmark (closest to directly behind v)

Morals of the story

Can use vertex heights to adjust edge weights and change algorithm behavior without changing which paths are shortest

Useful for eliminating negative edges in all-pairs computation (Johnson's algorithm), finding two paths (Suurballe's algorithm), guiding Dijkstra to find destination more quickly (A* algorithm)

References I

Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A^* search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 156–165. Society for Industrial and Applied Mathematics, 2005. URL `https://dl.acm.org/citation.cfm?id=1070432.1070455`.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi:`10.1109/TSSC.1968.300136`.

Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977. doi:`10.1145/321992.321993`.

J. W. Suurballe. Disjoint paths in a network. *Networks*, 4(2):125–145, 1974. doi:`10.1002/net.3230040204`.

J. W. Suurballe and Robert E. Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984. doi:`10.1002/net.3230140209`.