CS 163 & CS 265: Graph Algorithms

Week 3: Shortest paths

Lecture 3b: Relaxation algorithms

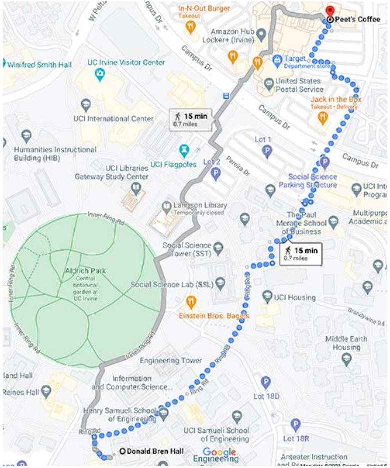
David Eppstein
University of California, Irvine

Winter Quarter, 2024

Typical application of shortest paths

Routing in street networks

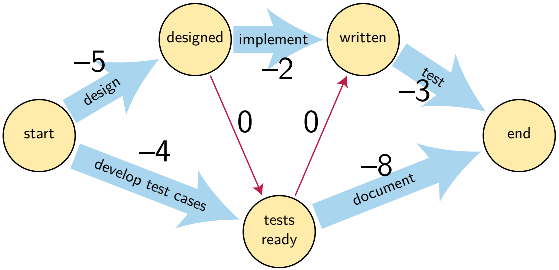
- Vertices: Points where multiple paths meet (e.g. street intersections)
- Edges: Possible routes between these points (segments of streets)
- Weights (length): physical distance or travel time
  All positive numbers!



Goal: Find a path from start vertex to destination vertex with minimum total weight

Critical path planning as a shortest path problem

Negate all the edge lengths!



Longest (critical) path in original scheduling graph =
shortest (most negative) path in the graph with negated weights

Another application with negative weights

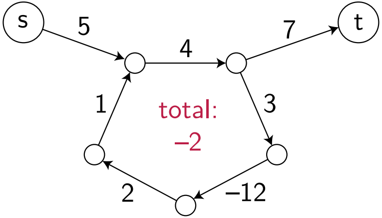
"Tramp steamer" (cargo ship) route planning

- Vertices = ports the ship could travel between
- Edges = trips from one port to another (directed)
- Weight of an edge = expenses − profit
  (positive: net loss, negative: net profit)

Goal: Find a cycle (path from any vertex back to itself) with negative total weight

Shortest walk might not exist

Walk: Like a path but allowing repeated edges and/or vertices



Length of s—t walks:

- Avoid loop: $5 + 4 + 7 = 16$
- Once around cycle: 14
- Twice around cycle: 12
- ...

Problem: Cycle with negative total length

(Exactly what we want to find in the tramp steamer problem)

If some path from s to t touches a negative cycle

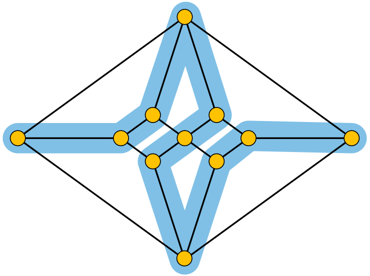
then going many times around the cycle gives arbitrarily short walks

Shortest path might be hard to find

Paths do not allow repetitions, so there are only finitely many paths
(at most $\sum_{i=1}^{n} \binom{n}{i} i!$ of them)

Therefore, shortest path is well-defined and always exists

But when all weights are -1, shortest (most negative) paths use all vertices, when this is possible: "Hamiltonian path". NP-complete to find these, so efficient algorithms are believed not to exist.



Overview of algorithms

All our algorithms for shortest paths require that the input does not have any negative cycle

For these inputs, shortest path = shortest walk

When the input is a directed acyclic graph:
$O(m)$ time using topological ordering (last time)

When all edge lengths are ≥ 0:
Dijkstra's algorithm, near-linear time

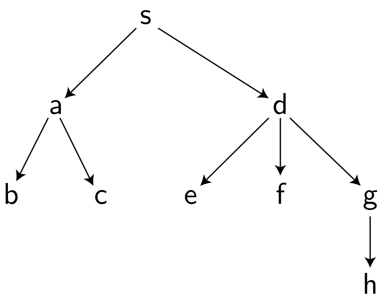
With negative edges but no negative cycles:
Bellman–Ford algorithm, $O(mn)$
can also find negative cycles when one exists

Shortest path trees

In graphs without negative cycles, paths from a single source vertex s to all other vertices form a tree

Parent of x is the second-to-last vertex y on the shortest path from s to x

Shortest path from s to x must use the shortest path to y, because if not then shortest path to y plus edge $y \to x$ would be a better path



E.g. shortest path from s to e is s$\longrightarrow d \longrightarrow e$

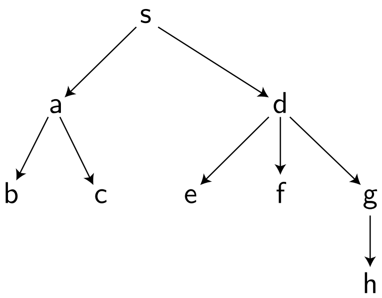
parent(e) $=$ second to last vertex, d

Single source shortest path problem

Input: graph with edge lengths
(can be directed or undirected)
plus starting vertex s

Outputs

- Tree of shortest paths from s to all other reachable vertices
- Distances (lengths of paths) to all vertices ($+\infty$ if unreachable)



Represent output by two decorations for each vertex x:

$$P[x] = \text{parent vertex of } x$$

$$D[x] = \text{distance from start vertex to } x$$

Relaxation algorithms

Maintain two decorations $P[x]$ and $D[x]$ for each vertex x

They will not always be the correct values
(correct: $P =$ parent in shortest path tree,
$D =$ length of shortest path)

Invariants:

- $D[x]$ is the length of some path to x
  (therefore, it is always $\geq$ the correct value)
- $P[x]$ is the second-to-last vertex on a path of length $\leq D$

Gradually find shorter paths and decrease $D[x]$ until everything becomes correct

Relaxation algorithms (more detail)

Initialize: $P[x] = \text{None}$; $D[x] = 0$ if $x = s$, $+\infty$ otherwise

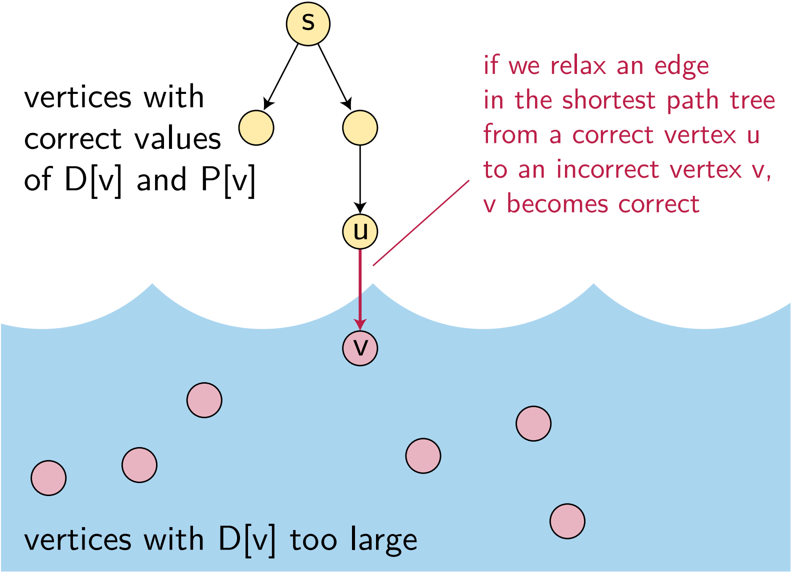
"Relax" edge uv: test whether path to u + edge uv gives a better path to v, and if so update the decorations for v

```
def relax(u,v):
    if D[u] + length(edge uv) < D[v]:
        D[v] = D[u] + length(edge uv)
        P[v] = u
```

Key insights:

- Initialization gives s the correct decorations (its distance and parent in the actual shortest path tree)
- If shortest path to v goes through edge uv and u already has correct decorations, then relax(uv) gives v correct decorations
- Other calls to relax are harmless
  (maintain invariant that $D[v] \geq$ actual distance)

Intuitive picture of a relaxation algorithm



Shortest paths in DAGs (from last time)

Two versions, both equally good:

```
initialize D, P
for v in topological order:
    for incoming edges uv:
        relax(u,v)
```

```
initialize D, P
for v in topological order:
    for outgoing edges vw:
        relax(v,w)
```

By induction on topological ordering, whenever we relax edge xy, its first vertex x will already have the correct values of D and P

So if we relax an edge in the shortest path tree, correct part grows

Total time is $O(m)$

Bellman–Ford algorithm

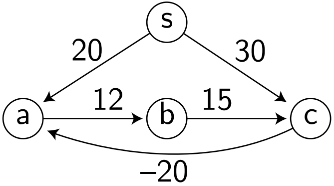
```
initialize D, P
repeat n-1 times:
    for each edge uv in the whole graph:
        relax(u,v)
```

Each time through the outer loop relaxes at least one shortest-path-tree edge from a correct vertex to an incorrect vertex

Total time is $O(mn)$

[Ford 1956; Bellman 1958; Moore 1959]

Bellman–Ford example



Initialize: P[all] = None, D[s] = 0, D[a]=D[b]=D[c]=∞

Outer loop #1

- relax ab: no change
- relax bc: no change
- relax ca: no change
- relax sa:
  D[a]=20 P[a]=s
- relax sc:
  D[c]=30 P[c]=s

Outer loop #2

- relax ab:
  D[b]=32 P[b]=a
- relax bc: no change
- relax ca:
  D[a]=10 P[a]=c
- relax sa: no change
- relax sc: no change

Outer loop #3

- relax ab:
  D[b]=22 P[b]=a
- relax bc: no change
- relax ca: no change
- relax sa: no change
- relax sc: no change

Bellman–Ford variations

Better in practice but all lead to same O-notation:

- Stop outer loop early if no relax step changes anything
- Only relax edges from changed vertices
- Better order of edges in inner loop $\Rightarrow$ fewer outer loops
  - Yen 1970: Split graph edges into two DAGs and topologically order them, reduce outer loop to $n/2$ times
  - Bannister & E. 2012: Choose the split randomly, reduce outer loop to $\approx n/3$ times
- If still changing after n outer loops, report negative cycle

Dijkstra's algorithm intuition

- Bellman–Ford is too slow because it relaxes edges many times; DAG algorithm is fast because it relaxes each edge only once
- DAG algorithm doesn't need to topologically sort the whole graph, only the shortest-path tree

  Shortest-path tree is always acyclic, even when the whole graph isn't
- If all edge weights are positive, then sorting vertices by distance from s is topologically sorts the shortest path tree

  For shortest path edge $u \to v$, $D[v] = D[u] + \text{positive} > D[u]$,
  so u will be earlier than v in the sorted order by distance
- We can't sort before we start (because we don't know the distances yet) but we can use a priority queue to sort as we go

Dijkstra's algorithm

```
initialize D, P
make priority queue Q of vertices, prioritized by D[v]
while Q is non-empty:
    find and remove minimum-priority vertex v in Q
    for each edge vw:
        relax(vw)
```

Time analysis:

- $\leq n$ find-and-remove operations in priority queue
- $\leq m$ decrease-priority operations
  (when relax changes D, that's a queue operation!)
- $O(m)$ other stuff such as looping through adjacency lists

- Binary heap: $O(\log n)$ per operation, $O(m \log n)$ total
- Fibonacci heap: $O(\log n)$ per find-and-remove,
  $O(1)$ per decrease-priority, $O(m + n \log n)$ total

Breaking news!

Bellman–Ford is optimal

Any randomized or deterministic relaxation-based algorithm that makes each decision without regard to the outcome of earlier relaxations uses time $\Omega(mn)$ on some graphs

[Eppstein 2023]

Bellman–Ford can be improved

A new un-reviewed preprint claims randomized expected time $\tilde{O}(mn^{8/9})$

The $\tilde{O}$ notation means we ignore logarithmic factors

Main idea: reweight and use Dijkstra (see Friday's lecture)

[Fineman 2023]

The morals of the story

Path length can be measured in many ways (road distance, travel time, profit) some of which allow negative lengths

Relaxation algorithms provide a unifying framework for several shortest path algorithms

Different input types have different choices of the best algorithm:

acyclic $\Rightarrow$ the DAG algorithm

has cycles but all edge lengths are positive $\Rightarrow$ Dijkstra

otherwise $\Rightarrow$ Bellman–Ford

References I

Michael J. Bannister and David Eppstein. Randomized speedup of the Bellman–Ford algorithm. In *Proc. Analytic Algorithmics and Combinatorics (ANALCO12)*, pages 41–47, 2012. doi:10.1137/1.9781611973020.6.

Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958. doi:10.1090/qam/102435.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. doi:10.1007/bf01386390.

David Eppstein. Lower bounds for non-adaptive shortest path relaxation. In Pat Morin and Subhash Suri, editors, *Proc. 18th Algorithms and Data Structures Symposium (WADS 2023)*, volume 14079 of *Lecture Notes in Computer Science*, pages 416–429. Springer-Verlag, 2023. doi:10.1007/978-3-031-38906-1_27.

Jeremy T. Fineman. Single-source shortest paths with negative real weights in $\tilde{O}(mn^{8/9})$ time. Electronic preprint arxiv:2311.02520, November 2023.

References II

Lester R. Jr. Ford. Network Flow Theory. RAND Papers P-923, RAND Corporation, Santa Monica, California, August 14 1956. URL https://www.rand.org/pubs/papers/P923.html.

Edward F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching, 1957*, volume 2, pages 285–292, Cambridge, Massachusetts, 1959. Harvard University Press.

Jin Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27(4):526–530, 1970. doi:10.1090/qam/253822.