Weighted Graph Algorithms

Presented by Jason Yuan

Slides: Zachary Friggstad

Programming Club Meeting

Weighted Graphs

```
struct Edge {
  int u, v;
  int weight; //can be a double

  Edge (int uu = 0, int vv = 0, int ww = 0)
    : u(uu), v(vv), w(ww) {}
  bool operator<(const Edge& rhs) const {
    return weight < rhs.weight;
  }
};

typedef vector<vector<Edge>> weighted_graph;
weighted_graph g(n); //create a graph with n nodes

//read and add an edge
Edge e;
cin >> e.u >> e.v >> e.weight;
g[e.u].push_back(e);
```

Dijkstra's Algorithm

Find the minimum-weight path from s to every other vertex.

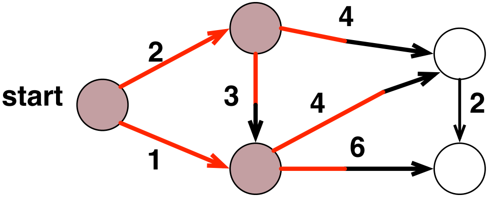
Assumption: $w(u, v) \geq 0$ for all edges uv.

Dijkstra's Algorithm

Find the minimum-weight path from s to every other vertex.

Assumption: $w(u, v) \geq 0$ for all edges uv.

Light a fire at s. The fire takes $w(s, v)$ seconds to reach every neighbour v of s.



When a fire reaches a vertex v, if the vertex has not yet been *burned* then the fire spreads down each edge exiting v.

Use the `Edge` struct to model active fires spreading across edges.
`Edge.weight` is the time when the fire reaches `Edge.v`.

Use the `Edge` struct to model active fires spreading across edges. `Edge.weight` is the time when the fire reaches `Edge.v`.

```
weighted_graph g; //assume already populated

priority_queue pq<Edge>; //active fires
vector<Edge> path(g.size(), Edge(-1, -1, -1));

pq.push(Edge(s, s, 0)); //a fire starts on s a time 0

while (!pq.empty()) {
   Edge curr = pq.top();
   pq.pop();
   if (path[curr.v].u != -1) continue; //already burned
   path[curr.v] = curr;
   for (auto& succ : g[curr.v])
      pq.push(Edge(succ.u, succ.v,
                   succ.weight + curr.weight));
}
```

Use the `Edge` struct to model active fires spreading across edges. `Edge.weight` is the time when the fire reaches `Edge.v`.

```
weighted_graph g; //assume already populated

priority_queue pq<Edge>; //active fires
vector<Edge> path(g.size(), Edge(-1, -1, -1));

pq.push(Edge(s, s, 0)); //a fire starts on s a time 0

while (!pq.empty()) {
   Edge curr = pq.top();
   pq.pop();
   if (path[curr.v].u != -1) continue; //already burned
   path[curr.v] = curr;
   for (auto& succ : g[curr.v])
      pq.push(Edge(succ.u, succ.v,
                   succ.weight + curr.weight));
}
```

Now $path[v].u$ is the vertex prior to v on a min-weight $s - v$ path and $path[v].weight$ is the weight of this path.

Problem

A `c++` `priority_queue` is a max-heap, meaning it will return the largest item in the heap.

I just use change `operator<` to check `weight > rhs.weight`.

Problem

A `c++` `priority_queue` is a max-heap, meaning it will return the largest item in the heap.

I just use change `operator<` to check `weight > rhs.weight`.

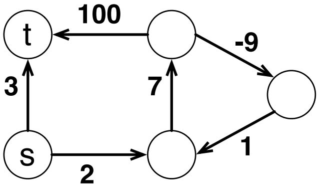
Running Time

$O(m \log m)$ where $m = \#$ edges.

An edge is burned at most once, and it takes $O(\log m)$ time to push and pop.

Handling Negative Weights

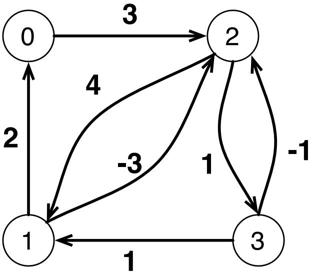
Things are tricker with minimum-weight cycles.



- A walk from s to t could run around a negative weight cycle as long as it wants, so there is no *minimum-weight walk*.
- If we insist on not repeating a vertex, it is NP-hard to find the minimum-weight path.

Shortest paths can still be found if no negative weight cycles exist.

Adjacency Matrix Representation



$$A = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & -3 & \infty \\ \infty & 4 & 0 & 1 \\ \infty & 1 & -1 & 0 \end{bmatrix} \quad (\infty \equiv \text{no edge})$$

It might make more sense to use ∞ on the diagonal. It depends on the application.

Floyd-Warshall

Dijkstra's algorithm fails to find shortest paths if some edges have negative weight, even if there are no negative-weight cycles.

Floyd-Warshall

Dijkstra's algorithm fails to find shortest paths if some edges have negative weight, even if there are no negative-weight cycles.

An incredibly simple dynamic programming approach will compute the shortest path between any two vertices in just 4 lines of code.

Floyd-Warshall

Dijkstra's algorithm fails to find shortest paths if some edges have negative weight, even if there are no negative-weight cycles.

An incredibly simple dynamic programming approach will compute the shortest path between any two vertices in just 4 lines of code.

Can handle negative weight edges and detect negative weight cycles.

Floyd-Warshall

Dijkstra's algorithm fails to find shortest paths if some edges have negative weight, even if there are no negative-weight cycles.

An incredibly simple dynamic programming approach will compute the shortest path between any two vertices in just 4 lines of code.

Can handle negative weight edges and detect negative weight cycles.

Let G be represented with an adjacency matrix with 0s on the diagonal.

```
/* Assumes g[u][v] is initially the cost of the edge (u,v),
   INFINITY (i.e. some big number) if no such edge.
   Also need g[v][v] = 0 for all v. */
for (int k = 0; k < n; k++)
   for (int u = 0; u < n; u++)
      for (int v = 0; v < n; v++)
         g[u][v] = min(g[u][v], g[u][k] + g[k][v]);
```

```
/* Assumes g[u][v] is initially the cost of the edge (u,v),
   INFINITY (i.e. some big number) if no such edge.
   Also need g[v][v] = 0 for all v. */
for (int k = 0; k < n; k++)
   for (int u = 0; u < n; u++)
      for (int v = 0; v < n; v++)
         g[u][v] = min(g[u][v], g[u][k] + g[k][v]);
```

If there is a negative weight cycle C, $g[v][v] < 0$ for some $v \in C$.

Otherwise, $g[u][v]$ is the weight of the min-weight $u - v$ path for any $u, v \in V$.

```
/* Assumes g[u][v] is initially the cost of the edge (u,v),
   INFINITY (i.e. some big number) if no such edge.
   Also need g[v][v] = 0 for all v. */
for (int k = 0; k < n; k++)
   for (int u = 0; u < n; u++)
      for (int v = 0; v < n; v++)
         g[u][v] = min(g[u][v], g[u][k] + g[k][v]);
```

If there is a negative weight cycle C, $g[v][v] < 0$ for some $v \in C$.

Otherwise, $g[u][v]$ is the weight of the min-weight $u - v$ path for any $u, v \in V$.

Invariant
After iteration k, $g[u][v]$ is the minimum possible weight of a $u - v$ path using only uses $0, \ldots, k$ as intermediate vertices.

```
/* Assumes g[u][v] is initially the cost of the edge (u,v),
   INFINITY (i.e. some big number) if no such edge.
   Also need g[v][v] = 0 for all v. */
for (int k = 0; k < n; k++)
   for (int u = 0; u < n; u++)
      for (int v = 0; v < n; v++)
         g[u][v] = min(g[u][v], g[u][k] + g[k][v]);
```

If there is a negative weight cycle C, $g[v][v] < 0$ for some $v \in C$.

Otherwise, $g[u][v]$ is the weight of the min-weight $u - v$ path for any $u, v \in V$.

Invariant
After iteration k, $g[u][v]$ is the minimum possible weight of a $u - v$ path using only uses $0, \ldots, k$ as intermediate vertices.

Running Time: $O(n^3)$.

Bellman-Ford

A (potentially) faster way to handle negative-weight edges.

Let $s \in V$, let $bf[s] = 0$ and $bf[v] = \infty$ for all $v \neq s$.

The values $bf[v]$ represent the shortest $s - v$ path “seen so far”.

Bellman-Ford

A (potentially) faster way to handle negative-weight edges.

Let $s \in V$, let $bf[s] = 0$ and $bf[v] = \infty$ for all $v \neq s$.

The values $bf[v]$ represent the shortest $s - v$ path "seen so far".

Try to find an edge (u, v) with $bf[u] + weight(u, v) < bf[v]$ to find a shorter path to v.

Iterate until no more changes to $bf[]$.

```
int bf[MAXN]; //MAXN = max # nodes possible
vector<Edge> edges; //list of all edges
...
for (int v = 0; v < n; ++v)
    bf[v] = (v == s ? 0 : INFINITY);

for (int iter = 0; iter < n; ++iter)
    for (auto& e : edges)
        bf[e.v] = min(bf[e.v], bf[e.u] + e.weight);
```

```
int bf[MAXN]; //MAXN = max # nodes possible
vector<Edge> edges; //list of all edges
...
for (int v = 0; v < n; ++v)
  bf[v] = (v == s ? 0 : INFINITY);

for (int iter = 0; iter < n; ++iter)
  for (auto& e : edges)
    bf[e.v] = min(bf[e.v], bf[e.u] + e.weight);
```

If there is still an edge e with $bf[e.u] + e.weight < bf[e.v]$ then e lies on a negative weight cycle.

```
int bf[MAXN]; //MAXN = max # nodes possible
vector<Edge> edges; //list of all edges
...
for (int v = 0; v < n; ++v)
  bf[v] = (v == s ? 0 : INFINITY);

for (int iter = 0; iter < n; ++iter)
  for (auto& e : edges)
    bf[e.v] = min(bf[e.v], bf[e.u] + e.weight);
```

If there is still an edge e with $bf[e.u] + e.weight < bf[e.v]$ then e lies on a negative weight cycle.

Running Time: $O(n \cdot m)$

```
int bf[MAXN]; //MAXN = max # nodes possible
vector<Edge> edges; //list of all edges
...
for (int v = 0; v < n; ++v)
  bf[v] = (v == s ? 0 : INFINITY);

for (int iter = 0; iter < n; ++iter)
  for (auto& e : edges)
    bf[e.v] = min(bf[e.v], bf[e.u] + e.weight);
```

If there is still an edge e with $bf[e.u] + e.weight < bf[e.v]$ then e lies on a negative weight cycle.

Running Time: $O(n \cdot m)$

Loop Invariants:

1. If $bf[v] \neq \infty$, it is the length of some $s - v$ path.
2. After k iterations, $bf[v] \leq$ shortest path that uses $\leq k$ steps.

All-pairs shortest paths with negative-weight edges

Initialize $bf[v] = 0, \forall v \in V$ and run the Bellman-Ford algorithm.

Note: $bf[u] + weight(u, v) < bf[v] < 0$ for some edge uv iff G has a negative-weight cycle.

All-pairs shortest paths with negative-weight edges

Initialize $bf[v] = 0, \forall v \in V$ and run the Bellman-Ford algorithm.

Note: $bf[u] + weight(u, v) < bf[v] < 0$ for some edge uv iff G has a negative-weight cycle.

If there is no negative weight cycle:

$$bf[u] + weight(u, v) - bf[v] \geq 0, \forall\ uv \in E$$

All-pairs shortest paths with negative-weight edges

Initialize $bf[v] = 0, \forall v \in V$ and run the Bellman-Ford algorithm.

Note: $bf[u] + weight(u, v) < bf[v] < 0$ for some edge uv iff G has a negative-weight cycle.

If there is no negative weight cycle:

$$bf[u] + weight(u, v) - bf[v] \geq 0, \forall \ uv \in E$$

Find all-pairs shortest paths by running Dijkstra's from each $v \in V$, but use edge weights $bf[u] + weight(u, v) - bf[v]$.

All-pairs shortest paths with negative-weight edges

Initialize $bf[v] = 0, \forall v \in V$ and run the Bellman-Ford algorithm.

Note: $bf[u] + weight(u, v) < bf[v] < 0$ for some edge uv iff G has a negative-weight cycle.

If there is no negative weight cycle:

$$bf[u] + weight(u, v) - bf[v] \geq 0, \forall\ uv \in E$$

Find all-pairs shortest paths by running Dijkstra's from each $v \in V$, but use edge weights $bf[u] + weight(u, v) - bf[v]$.

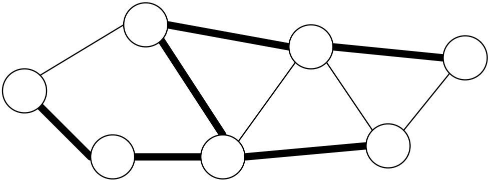
Running time: $O(nm \log m)$. Better than $O(n^3)$ in sparse graphs.

Minimum Spanning Tree

Let $G = (V, E)$ be an undirected, connected, and weighted graph.

Spanning Tree

A subset of edges $T \subseteq E$ forming a connected tree.



Find the spanning tree with minimum total edge weight.

Kruskal's Algorithm

- Sort the edges $e_1, \ldots, e_m$ by weight.
- $T = \emptyset$
- For each $e_i = (u, v)$ in this order, if u is not connected to v in T then add e_i to T.

Kruskal's Algorithm

- Sort the edges $e_1, \ldots, e_m$ by weight.
- $T = \emptyset$
- For each $e_i = (u, v)$ in this order, if u is not connected to v in T then add e_i to T.

Exercise

- The final T will always be connected (assuming G is connected).
- There are no cycles in T.

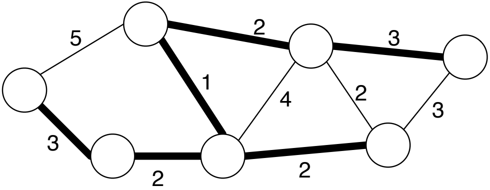
So, T is a spanning tree.

Greedy algorithms like this often fail to find optimum solutions. Why does it work in this case?

Greedy algorithms like this often fail to find optimum solutions. Why does it work in this case?

Lemma (good coffee shop math)

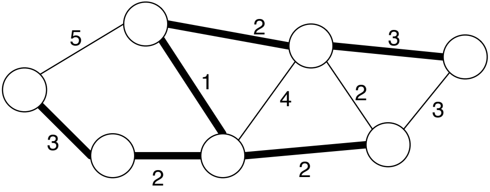
A spanning tree T is a minimum-weight spanning tree if and only if for every cycle C, some heaviest edge of C is not in T.



Greedy algorithms like this often fail to find optimum solutions. Why does it work in this case?

Lemma (good coffee shop math)

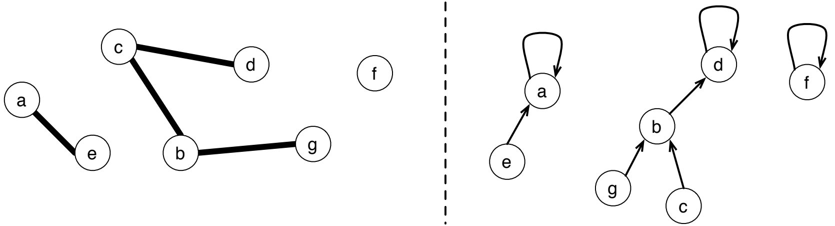
A spanning tree T is a minimum-weight spanning tree if and only if for every cycle C, some heaviest edge of C is not in T.



So, when the algorithm considers the heaviest edge e_i of some cycle, its endpoints are already connected so e_i will not be added!

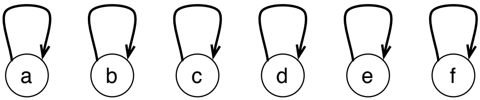
To **efficiently** detect if T connects the endpoints of an edge e_i, we use the union-find data structure.

It represents a connected component in T by a directed tree, the root is the *representative* of the tree.

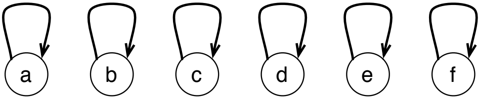


The endpoints of e_i have the same representative if and only if they are already connected.

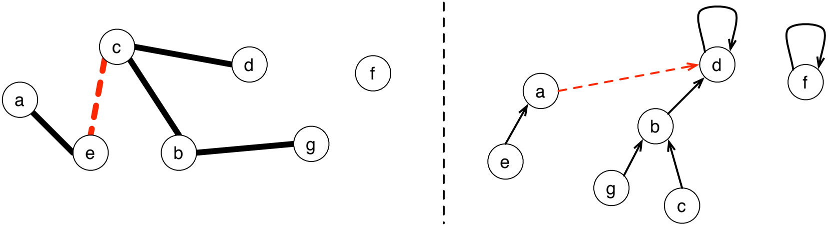
Before adding any edges (i.e. $T = \emptyset$):



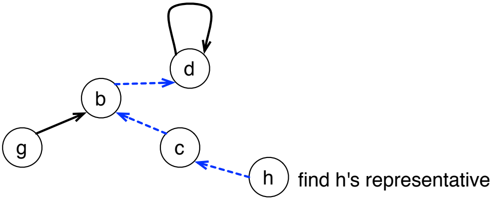
Before adding any edges (i.e. $T = \emptyset$):



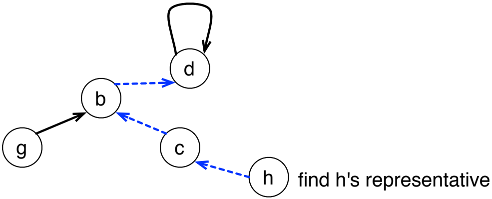
To merge two components, just point one representative at the other.



Crawl up the tree to find representatives.

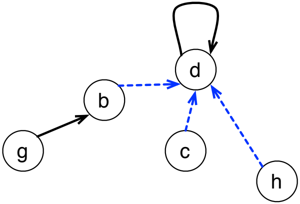


Crawl up the tree to find representatives.



Path Compression

To speed up future calculations, also point all nodes seen to the rep.



```
int uf[MAXN]; //union find pointers
int find(int u) {
  if (uf[u] != u) uf[u] = find(uf[u]);
  return uf[u];
}
bool merge(int u, int v) {
  u = find(u); v = find(v);
  uf[u] = v;
  return u == v; //true iff this merged different components
}
...
for (int v = 0; v < n; ++i) uf[v] = v; //initialization

//suppose edges are stored in vector<Edge> edges;
sort(edges.begin(), edges.end());

int mst = 0;
for (auto& e : edges)
  if (merge(e.u, e.v))
    mst += e.weight;
```

Running time:

- $O(m \log m)$ to sort the edges
- m union-find “merges” has **total** running time $O(m \log m)$

Overall: $O(m \log m)$.

Running time:

- $O(m \log m)$ to sort the edges
- m union-find "merges" has **total** running time $O(m \log m)$

Overall: $O(m \log m)$.

Union Find by Rank (**in textbook**)
k merge calls has running time $O(\alpha(k) \cdot k)$ where $\alpha(k)$ is a crazy slow function.

$\alpha(k) \leq 5$ for $k \leq 100^{100^{100}}$

Though, $\alpha(k) \rightarrow \infty$ so it's technically not a constant.

Open Kattis - `borg`

`https://open.kattis.com/problems/borg`

Open Kattis - `borg`

`https://open.kattis.com/problems/borg`
The path of edges taken looks like a tree and the nodes are either `S` or `A` nodes.

Compute the shortest path distance between any two of these characters: $O(a \cdot x \cdot y)$ time using a BFS from each of the a aliens.

Then compute the minimum spanning tree of the shortest-path graph in $O(a \log a)$ time.

Overall: $O(a \cdot x \cdot y)$ time

Open Kattis - allpairspath

https://open.kattis.com/problems/allpairspath

Open Kattis - `allpairspath`

`https://open.kattis.com/problems/allpairspath`

Recall the running times. What was the fastest one for this task?

Not Bellman-Ford, not Floyd-Warshall, not Dijkstra's.

The Dijkstra + Bellman-Ford hybrid!

Precomputes all-pairs shortest path distances in $O(nm \log m)$ time, each query can be answered in $O(1)$ time.

Open Kattis - `redbluetree`

https://open.kattis.com/problems/redbluetree

Open Kattis - `redbluetree`

`https://open.kattis.com/problems/redbluetree`

Compute any spanning tree T, first by processing the red edges. Return `no` if no spanning tree or if $> k$ blue edges are used.

Re-run Kruskal's algorithm by first adding blue edges of T, then the remaining edges until exactly k are added, then adding red.

Correctness Idea: The first pass identifies the minimum number of blue edges required.

Let T^* be a feasible solution. For any blue $e \in T - T^*$, there is some blue $f \in T^* - T$ with $T^* - f + e$, $T - e + f$ being spanning trees. So $T^* - f + e$ is another feasible solution agreeing more with T on blue edges. Iterating shows B_T can be extended to a feasible solution.