CS 577 - Graph Algorithms Part (A)

Manolis Vlatakis

Department of Computer Sciences
University of Wisconsin – Madison

Fall 2024

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Shortest Path

Finding the Shortest Path

Problem Definition

We have a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$ and a node s that has a path to every other node in V. For each edge e, $\ell_e \geq 0$ is the length of the edge.

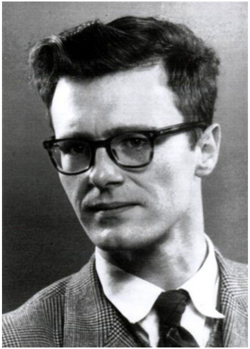
- What is the shortest path from s to each other node?

Finding the Shortest Path

Problem Definition

We have a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$ and a node s that has a path to every other node in V. For each edge e, $\ell_e \geq 0$ is the length of the edge.

- What is the shortest path from s to each other node?



Edsger Dijkstra, 1956
Dijkstra's shortest path fame

Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest

$$d'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

  Append v to S and define $d(v) = d'(v)$.
end

Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
    Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest

$$d'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

    Append v to S and define $d(v) = d'(v)$.
end

How is it greedy?

Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest

$$d'(v) = \min_{e=(u,v):u \in S} \{d(u) + \ell_e\}$$

  Append v to S and define $d(v) = d'(v)$.
end

How is it greedy?
Which technique to prove optimality?

Correctness of Dijkstra's

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s - u$ path.

CORRECTNESS OF DIJKSTRA'S

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s - u$ path.

Proof.

Correctness of Dijkstra's

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s-u$ path.

Proof.

By induction on the size of S.

Correctness of Dijkstra's

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s - u$ path.

Proof.

By induction on the size of S.

- For $|S| = 1$, the claim follows trivially as $S = \{s\}$.

Correctness of Dijkstra's

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s - u$ path.

Proof.

By induction on the size of S.

- For $|S| = 1$, the claim follows trivially as $S = \{s\}$.
- By the induction hypothesis, for $|S| = k$, P_u is the shortest $s - u$ path for all $u \in S$.

Correctness of Dijkstra's

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s - u$ path.

Proof.

By induction on the size of S.

- In step $k + 1$, we add v.
  - By definition, P_v is shortest path connected to S by one edge.

Correctness of Dijkstra's

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s - u$ path.

Proof.

By induction on the size of S.

- In step $k + 1$, we add v.
  - By definition, P_v is shortest path connected to S by one edge.
  - Since P_u is a shortest path to u, P_v is the shortest path to v when considering only the nodes of S.

Correctness of Dijkstra's

Theorem 1

Consider the S at any point in the execution of Dijkstra's. For each $u \in S$, the path P_u is a shortest $s - u$ path.

Proof.

By induction on the size of S.

- In step $k + 1$, we add v.
  - By definition, P_v is shortest path connected to S by one edge.
  - Since P_u is a shortest path to u, P_v is the shortest path to v when considering only the nodes of S.
  - Moreover, there cannot be a shorter path to v passing through another node $y \notin S$ else y that would be added at $k + 1$.

□

DIJKSTRA'S OBSERVATIONS

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
    Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
    Append v to S and define $d(v) = d'(v)$.
end

- Negative edge weights, where does it fail?

Dijkstra's Observations

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
  Append v to S and define $d(v) = d'(v)$.
end

- Negative edge weights, where does it fail?
- 🤔It is graph exploration, what kind of exploration?

Dijkstra's Observations

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
 Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
 Append v to S and define $d(v) = d'(v)$.
end

- Negative edge weights, where does it fail?
- It is graph exploration, what kind of exploration?
  - Weighted (continuous) BFS

Implementation and Run Time of Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
    Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
    Append v to S and define $d(v) = d'(v)$.
end

- 🤔Number of iterations of the loop?

Implementation and Run Time of Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
  Append v to S and define $d(v) = d'(v)$.
end

- 🤔Number of iterations of the loop? $n - 1$

Implementation and Run Time of Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
    Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
    Append v to S and define $d(v) = d'(v)$.
end

- 🤔Number of iterations of the loop? $n - 1$
- Key Operations: *Finding the min*:

Implementation and Run Time of Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u\in S}\{d(u) + \ell_e\}$
  Append v to S and define $d(v) = d'(v)$.
end

- 🤔Number of iterations of the loop? $n - 1$
- Key Operations: *Finding the min*: Easy in $O(m)$

Implementation and Run Time of Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
  Append v to S and define $d(v) = d'(v)$.
end

- Number of iterations of the loop? $n - 1$
- Key Operations: *Finding the min*: Easy in $O(m)$
- Overall: $O(mn)$

Implementation and Run Time of Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
  Append v to S and define $d(v) = d'(v)$.
end

- Number of iterations of the loop? $n - 1$
- Key Operations: *Finding the min*: Easy in $O(m)$
- Overall: $O(mn)$
- How can we get $O(m \log n)$?

Implementation and Run Time of Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $S \neq V$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest $d'(v) = \min_{e=(u,v):u \in S}\{d(u) + \ell_e\}$
  Append v to S and define $d(v) = d'(v)$.
end

- 🤔Number of iterations of the loop? $n - 1$
- Key Operations: *Finding the min*: Easy in $O(m)$
- Overall: $O(mn)$
- How can we get $O(m \log n)$?
- How can we get $O(n + n \log n)$?

Shortest Path-Negative Side of the Moon

Shortest Path

Going Negative

Problem Definition

We have a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$ and a node s that has a path to every other node in V. For each edge $e = (i, j)$, c_{ij} is the weight of the edge, and the are no cycles with negative weight.

- What is the shortest path from s to each other node?

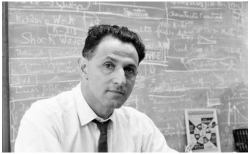
Shortest Path

Going Negative

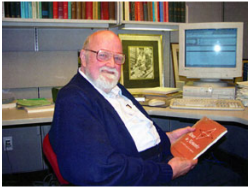
Problem Definition

We have a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$ and a node s that has a path to every other node in V. For each edge $e = (i, j)$, c_{ij} is the weight of the edge, and the are no cycles with negative weight.

- What is the shortest path from s to each other node?



Richard Bellman



L R Ford Jr.

Shortest Path

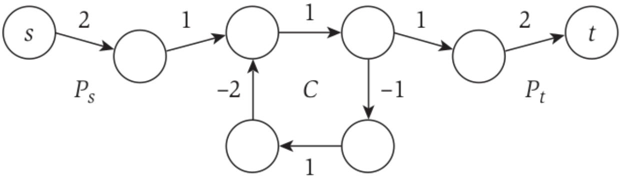
Going Negative

Problem Definition

We have a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$ and a node s that has a path to every other node in V. For each edge $e = (i, j)$, c_{ij} is the weight of the edge, and the are no cycles with negative weight.

- What is the shortest path from s to each other node?

Why no negative cycles?



Dijkstra's

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.
For each $u \in S$, we store a distance value $d(u)$.
Initialize $S = \{s\}$ and $d(s) = 0$
while $\underline{S \neq V}$ **do**
  Choose $v \notin S$ with at least one incoming edge originating from a node in S with the smallest

$$d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$$

  Append v to S and define $d(v) = d'(v)$.
end
return $\underline{S}$

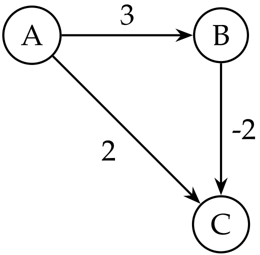
Dijkstra's

Negative Problem

Dijkstra's

Negative Problem

- Lose guarantee that minimum edge between S and $V \setminus S$ is part of minimum path.



Dijkstra's

Negative Problem

- Lose guarantee that minimum edge between S and $V \setminus S$ is part of minimum path.

Why not just boost all edges by max negative value plus a bit (β)?

Dijkstra's

Negative Problem

- Lose guarantee that minimum edge between S and $V \setminus S$ is part of minimum path.

Why not just boost all edges by max negative value plus a bit (β)?

- A path with x edges: Cost increases $x \cdot \beta$.

Dijkstra's

Negative Problem

- Lose guarantee that minimum edge between S and $V \setminus S$ is part of minimum path.

Why not just boost all edges by max negative value plus a bit (β)?

- A path with x edges: Cost increases $x \cdot \beta$.
- Solution in new graph is not guaranteed to be optimal in original graph.

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n - 1$ edges.

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n - 1$ edges.

Dynamic Program

- 2D matrix M of # edges in path × vertices.

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n - 1$ edges.

Dynamic Program

- 2D matrix M of # edges in path × vertices.
  - $M[i][v]$ is the shortest path from v to t using $\leq i$ edges.

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n-1$ edges.

Dynamic Program

- 2D matrix M of # edges in path × vertices.
  - $M[i][v]$ is the shortest path from v to t using $\leq i$ edges.
  - 🤔Where is the shortest path from s to t in the solution matrix?

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n - 1$ edges.

Dynamic Program

- 2D matrix M of # edges in path × vertices.
  - $M[i][v]$ is the shortest path from v to t using $\leq i$ edges.
  - Solution: $M[n-1][s]$

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n-1$ edges.

Dynamic Program

- 2D matrix M of # edges in path × vertices.
  - $M[i][v]$ is the shortest path from v to t using $\leq i$ edges.
  - Solution: $M[n-1][s]$
- Dichotomy:

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n-1$ edges.

Dynamic Program

- 2D matrix M of # edges in path $\times$ vertices.
  - $M[i][v]$ is the shortest path from v to t using $\leq i$ edges.
  - Solution: $M[n-1][s]$
- Dichotomy:
  - Use $\leq i-1$ edges.
  - Use $\leq i$ edges.

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n-1$ edges.

Dynamic Program

- 2D matrix M of # edges in path × vertices.
  - $M[i][v]$ is the shortest path from v to t using $\leq i$ edges.
  - Solution: $M[n-1][s]$
- Dichotomy:
  - Use $\leq i-1$ edges.
  - Use $\leq i$ edges.

🤔Build the Bellman equation.

Bellman-Ford

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n-1$ edges.

Dynamic Program

- 2D matrix M of # edges in path × vertices.
  - $M[i][v]$ is the shortest path from v to t using $\le i$ edges.
  - Solution: $M[n-1][s]$
- Dichotomy:
  - Use $\le i-1$ edges.
  - Use $\le i$ edges.

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\},$$

where $c_{vw} = \infty$ if no edge from v to w.

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- 🤔 # of Cells:

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- # of Cells: $O(n^2)$.

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- # of Cells: $O(n^2)$.
- 🤔 Cost per cell:

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- # of Cells: $O(n^2)$.
- Cost per cell: $O(n)$.

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- # of Cells: $O(n^2)$.
- Cost per cell: $O(n)$.
- Overall: $O(n^3)$.

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- # of Cells: $O(n^2)$.
- Cost per cell: $O(n)$.
- Overall: $O(n^3)$.

Worst Case: n nodes, m edges

- For each node v, we only need to consider outgoing edges to w (denoted by η_v).
- For every node v, we need to do this calculation for $0 \leq i \leq n-1$ lengths.

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- # of Cells: $O(n^2)$.
- Cost per cell: $O(n)$.
- Overall: $O(n^3)$.

Worst Case: n nodes, m edges

- For each node v, we only need to consider outgoing edges to w (denoted by η_v).
- For every node v, we need to do this calculation for $0 \leq i \leq n-1$ lengths.
- Overall: $O\left(n \sum_{v \in V} \eta_v\right) = O(mn)$.

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes, m edges

- Overall: $O\left(n \sum_{v \in V} \eta_v\right) = O(mn)$.

Space Saving: $O(n)$.

- To build row i:
  - We only need $i-1$ values for each node.
  - $M[v] = \min\{M[v], \min_{w \in V}\{M[w] + c_{vw}\}\}$ for each i.

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes, m edges

- Overall: $O\left(n \sum_{v \in V} \eta_v\right) = O(mn)$.

Space Saving: $O(n)$.

- To build row i:
  - We only need $i-1$ values for each node.
  - $M[v] = \min\{M[v], \min_{w \in V}\{M[w] + c_{vw}\}\}$ for each i.
- Recovery of actual path:

Bellman-Ford Analysis

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes, m edges

- Overall: $O\left(n \sum_{v \in V} \eta_v\right) = O(mn)$.

Space Saving: $O(n)$.

- To build row i:
  - We only need $i-1$ values for each node.
  - $M[v] = \min\{M[v], \min_{w \in V}\{M[w] + c_{vw}\}\}$ for each i.
- Recovery of actual path: An additional array $\mathit{first}[v]$ that maintains the first hop from v to t.

Negative Cycles

Observation 2

If there is a negative cycle along the path from s to t, then the shortest path is $-\infty$.

Negative Cycles

Observation 2

If there is a negative cycle along the path from s to t, then the shortest path is $-\infty$.

Observation 3

$M[i][v] = M[n-1][v]$ *for all* $i > n-1$ *and all nodes* v *if there are no negative cycles on the paths to* t.

Negative Cycles

Observation 2

If there is a negative cycle along the path from s to t, then the shortest path is $-\infty$.

Observation 3

$M[i][v] = M[n-1][v]$ *for all* $i > n-1$ *and all nodes* v *if there are no negative cycles on the paths to* t.

Augmented Graph for Negative Cycle Finding

- Add a node t with an incoming edge from all other nodes with cost 0.
- Run Bellman-Ford from any node s to t until number of edges n.
- If, for some v, $M[n][v] \neq M[n-1][v]$, then there is a negative cycle.

MST

Minimum Spanning Tree Problem

MST Problem

Let $G = (V, E)$ be a connected graph, where $|V| = n$ and $|E| = m$. For each edge e, $c_e > 0$ is the cost of the edge.

- Find an edge set $F \subseteq E$ with minimum cost that keeps the graph connected. That is, F should minimize $\sum_{e \in F} c_e$.

Minimum Spanning Tree Problem

MST Problem

Let $G = (V, E)$ be a connected graph, where $|V| = n$ and $|E| = m$. For each edge e, $c_e > 0$ is the cost of the edge.

- Find an edge set $F \subseteq E$ with minimum cost that keeps the graph connected. That is, F should minimize $\sum_{e \in F} c_e$.

Observation 4

Let $T = (V, F)$ be a minimum-cost solution to the problem described above. Then, T is a tree.

Minimum Spanning Tree Problem

MST Problem

Let $G = (V, E)$ be a connected graph, where $|V| = n$ and $|E| = m$. For each edge e, $c_e > 0$ is the cost of the edge.

- Find an edge set $F \subseteq E$ with minimum cost that keeps the graph connected. That is, F should minimize $\sum_{e \in F} c_e$.

Observation 4

Let $T = (V, F)$ be a minimum-cost solution to the problem described above. Then, T is a tree.

Proof.

Minimum Spanning Tree Problem

MST Problem

Let $G = (V, E)$ be a connected graph, where $|V| = n$ and $|E| = m$. For each edge e, $c_e > 0$ is the cost of the edge.

- Find an edge set $F \subseteq E$ with minimum cost that keeps the graph connected. That is, F should minimize $\sum_{e \in F} c_e$.

Observation 4

Let $T = (V, F)$ be a minimum-cost solution to the problem described above. Then, T is a tree.

Proof.

- By the definition of the problem, T must be connected.
- By way of contradiction, assume that T has a cycle C. Remove any edge from C resulting in a graph T'. T' is still connect and has a cost less than T.

□

Algorithm Design

🤔What greedy heuristic might work?

Algorithm Design

🤔What greedy heuristic might work?

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Algorithm Design

🤔What greedy heuristic might work?

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Jarník's (1929), Kruskal's (1956), Prim's (1957), Loberman and Weinberger (1957), Dijkstra's (1958) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Algorithm Design

🤔What greedy heuristic might work?

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Algorithm Design

🤔What greedy heuristic might work?

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Kruskal's (1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

Algorithm Design

What greedy heuristic might work?

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Reverse-Delete (Kruskal's 1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

Assume Distinct Weights

WLOG (without loss of generality)

Theorem 2

(HW Q2) If all edge weights in a connected graph are distinct, then G has a unique MST.

Assume Distinct Weights

WLOG (without loss of generality)

Theorem 2

(HW Q2) If all edge weights in a connected graph are distinct, then G has a unique MST.

Observation 5

All we need is a consistent tie-breaker when $c_{e_1} = c_{e_2}$ for some pair of edges. I.e. based on the labels of the vertices of $e_1 \cup e_2$.

Assume Distinct Weights

WLOG (without loss of generality)

Theorem 2

(HW Q2) If all edge weights in a connected graph are distinct, then G has a unique MST.

Observation 5

All we need is a consistent tie-breaker when $c_{e_1} = c_{e_2}$ for some pair of edges. I.e. based on the labels of the vertices of $e_1 \cup e_2$.

Assumption: all edge weights are distinct.

ANALYZING MST HEURISTICS

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Analyzing MST Heuristics

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Proof.

Analyzing MST Heuristics

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Proof.

By exchange argument:

Analyzing MST Heuristics

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Proof.

By exchange argument:

- Let T be a spanning tree that does not contain e.

Analyzing MST Heuristics

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Proof.

By exchange argument:

- Let T be a spanning tree that does not contain e.
- Let $e' = (v', w')$, where e' **is in** $P_{v,w} \in T$**,** $v' \in S$**, and** $w' \in V \setminus S$.

ANALYZING MST HEURISTICS

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Proof.

By exchange argument:

- Let T be a spanning tree that does not contain e.
- Let $e' = (v', w')$, where e' **is in** $P_{v,w} \in T$**,** $v' \in S$**, and** $w' \in V \setminus S$.
- Let $T' = T \setminus e' \cup e$.

Analyzing MST Heuristics

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Proof.

By exchange argument:

- Let T be a spanning tree that does not contain e.
- Let $e' = (v', w')$, where e' **is in** $P_{v,w} \in T$**,** $v' \in S$**, and** $w' \in V \setminus S$.
- Let $T' = T \setminus e' \cup e$.
- T' is connected as e is a $P_{v,w} \in T'$.

Analyzing MST Heuristics

Lemma 3

Let $S \subset V$ be an non-empty proper subset of the nodes, and let $e = (v, w)$ be the minimum cost edge connecting S and $V \setminus S$. Then, every MST contains e.

Proof.

By exchange argument:

- Let T be a spanning tree that does not contain e.
- Let $e' = (v', w')$, where e' **is in** $P_{v,w} \in T$**,** $v' \in S$**, and** $w' \in V \setminus S$.
- Let $T' = T \setminus e' \cup e$.
- T' is connected as e is a $P_{v,w} \in T'$.
- Since $c_e < c_{e'}$, cost of T' is less than T.

□

Kruskal's Algorithm is Optimal

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Theorem 4

Kruskal's Algorithm produces an MST.

Kruskal's Algorithm is Optimal

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Theorem 4

Kruskal's Algorithm produces an MST.

Proof.

Kruskal's Algorithm is Optimal

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Theorem 4

Kruskal's Algorithm produces an MST.

Proof.

- Let $e = (v, w)$ be the edge added at any step i.

Kruskal's Algorithm is Optimal

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Theorem 4

Kruskal's Algorithm produces an MST.

Proof.

- Let $e = (v, w)$ be the edge added at any step i.
- Since e does not create a cycle, $v \in S$ and $w \notin S$ (WLOG).

Kruskal's Algorithm is Optimal

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Theorem 4

Kruskal's Algorithm produces an MST.

Proof.

- Let $e = (v, w)$ be the edge added at any step i.
- Since e does not create a cycle, $v \in S$ and $w \notin S$ (WLOG).
- As c_e is the minimum cost edge, the claim follows from Lemma 3.

□

Prim's Algorithm is Optimal

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Theorem 5

Prim's Algorithm produces an MST.

Prim's Algorithm is Optimal

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Theorem 5

Prim's Algorithm produces an MST.

Proof.

Prim's Algorithm is Optimal

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Theorem 5

Prim's Algorithm produces an MST.

Proof.

- Immediate from Lemma 3.

Prim's Algorithm is Optimal

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Theorem 5

Prim's Algorithm produces an MST.

Proof.

- Immediate from Lemma 3.
- That is, Prim's algorithm does exactly what Lemma 3 describes.

□

Reverse-Delete is Optimal

Reverse-Delete (Kruskal's 1956) Algorithm

- Sort edges by cost from highest to lowest.
- Remove edges unless graph would become disconnected.

How should we prove that it produces an MST?

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Proof.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Proof.

- Let T be a spanning tree that does contain e.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Proof.

- Let T be a spanning tree that does contain e.
- Let S and $V \setminus S$ be the nodes of the connected components after removing e from T.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Proof.

- Let T be a spanning tree that does contain e.
- Let S and $V \setminus S$ be the nodes of the connected components after removing e from T.
- Let e' be an edge in C that connects S and $V \setminus S$.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Proof.

- Let T be a spanning tree that does contain e.
- Let S and $V \setminus S$ be the nodes of the connected components after removing e from T.
- Let e' be an edge in C that connects S and $V \setminus S$.
- Let $T' = T \setminus e \cup e'$.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Proof.

- Let T be a spanning tree that does contain e.
- Let S and $V \setminus S$ be the nodes of the connected components after removing e from T.
- Let e' be an edge in C that connects S and $V \setminus S$.
- Let $T' = T \setminus e \cup e'$.
- T' is connected as e' reconnects S and $V \setminus S$.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Proof.

- Let T be a spanning tree that does contain e.
- Let S and $V \setminus S$ be the nodes of the connected components after removing e from T.
- Let e' be an edge in C that connects S and $V \setminus S$.
- Let $T' = T \setminus e \cup e'$.
- T' is connected as e' reconnects S and $V \setminus S$.
- Since $c_e > c_{e'}$, cost of T' is less than T.

□

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Theorem 7

Reverse-Delete Algorithm produces an MST.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Theorem 7

Reverse-Delete Algorithm produces an MST.

Proof.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Theorem 7

Reverse-Delete Algorithm produces an MST.

Proof.

- Let $e = (v, w)$ be an edge removed at any step i.

REVERSE-DELETE IS OPTIMAL

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Theorem 7

Reverse-Delete Algorithm produces an MST.

Proof.

- Let $e = (v, w)$ be an edge removed at any step i.
- By definition e, belongs to a cycle C.

Reverse-Delete is Optimal

Lemma 6

Let C be any cycle in G, and let e be the most expensive edge of C. Then, e is not in any MST of G.

Theorem 7

Reverse-Delete Algorithm produces an MST.

Proof.

- Let $e = (v, w)$ be an edge removed at any step i.
- By definition e, belongs to a cycle C.
- As c_e is the maximum cost edge of C, the claim follows from Lemma 6.

□

Implementing Prim's Algorithm

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Key Operations

Implementing Prim's Algorithm

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Key Operations

- Retrieve the minimum valued edge between S and $V \setminus S$.

Implementing Prim's Algorithm

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Key Operations

- Retrieve the minimum valued edge between S and $V \setminus S$.
- Prim's and Dijkstra's have nearly identical implementations (but different minimizers)!

Implementing Prim's Algorithm

Prim's (1957) Algorithm

- Initialize a node set S with an arbitrary node s.
- Keep the least expensive edge as long as it does not create a cycle.

Key Operations

- Retrieve the minimum valued edge between S and $V \setminus S$.
- Prim's and Dijkstra's have nearly identical implementations (but different minimizers)!

Priority Queue (min-heap)

- `ExtractMin` $(O(1))$: $n-1$ times.
- `ChangeKey` $(O(\log(n)))$: m times.

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

Implementing Kruskal's Algorithm

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Key Operations

Implementing Kruskal's Algorithm

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Key Operations

- Sorting the edges: $(O(m \log m)$ and, since $m \leq n^2$, $O(m \log n))$.

Implementing Kruskal's Algorithm

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Key Operations

- Sorting the edges: $(O(m \log m)$ and, since $m \leq n^2$, $O(m \log n))$.
- Maintain sets of connected components that we merge.

Implementing Kruskal's Algorithm

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Key Operations

- Sorting the edges: $(O(m \log m)$ and, since $m \leq n^2$, $O(m \log n))$.
- Maintain sets of connected components that we merge.
- Initialize one set per node: $O(n)$.

Implementing Kruskal's Algorithm

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Key Operations

- Sorting the edges: ($O(m \log m)$ and, since $m \leq n^2$, $O(m \log n)$).
- Maintain sets of connected components that we merge.
- Initialize one set per node: $O(n)$.

Union-Find Data Structure

- `Find(x)`: Finds the set containing x. ($O(\log n)$ can be $O(\alpha(n))$)
- `Union(x,y)`: Joins two sets x and y. ($O(1)$)

Union-Find / Disjoint-Set

Key Operations

- `Find(x)`: Finds the set containing x. ($O(\log n)$ can be $O(\alpha(n))$)
- `Union(x,y)`: Joins two sets x and y. ($O(1)$)

Union-Find / Disjoint-Set

Key Operations

- `Find(x)`: Finds the set containing x. ($O(\log n)$ can be $O(\alpha(n))$)
- `Union(x,y)`: Joins two sets x and y. ($O(1)$)

Basic Container

`node`	`rank`	`parent`

Union-Find / Disjoint-Set

Key Operations

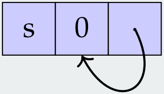
- `Find(x)`: Finds the set containing x. ($O(\log n)$ can be $O(\alpha(n))$)
- `Union(x,y)`: Joins two sets x and y. ($O(1)$)

Basic Container

node	rank	parent

Initializing Data Structure for Kruskal's

For each node s, create a singleton set. That is each container has rank 0 and points to itself.



Union-Find Operations

Find(x): $O(\log n)$

Union-Find Operations

`Find`(x): $O(\log n)$

- If x`.parent` points to x, return x.
- Else `Find`(x`.parent`)

Union-Find Operations

`Find`(x): $O(\log n)$

- If x`.parent` points to x, return x.
- Else `Find`(x`.parent`)
- $O(\log n)$ requires balanced trees.

Union-Find Operations

`Find(`x`)`: $O(\log n)$

- If x`.parent` points to x, return x.
- Else `Find(`x`.parent)`
- $O(\log n)$ requires balanced trees.
- $O(\alpha(n))$ with path compression.

Union-Find Operations

`Find`(x): $O(\log n)$

- If x`.parent` points to x, return x.
- Else `Find`(x`.parent`)
- $O(\log n)$ requires balanced trees.
- $O(\alpha(n))$ with path compression.

`Union`(x,y): $O(1)$

Union-Find Operations

`Find`(x): $O(\log n)$

- If x`.parent` points to x, return x.
- Else `Find`(x`.parent`)
- $O(\log n)$ requires balanced trees.
- $O(\alpha(n))$ with path compression.

`Union`(x,y): $O(1)$

- (WLOG) x`.rank` $\geq y$`.rank`:
  y`.parent` $= x$

Union-Find Operations

`Find`(x): $O(\log n)$

- If x`.parent` points to x, return x.
- Else `Find`(x`.parent`)
- $O(\log n)$ requires balanced trees.
- $O(\alpha(n))$ with path compression.

`Union`(x,y): $O(1)$

- (WLOG) x`.rank` $\geq y$`.rank`:
  y`.parent` $= x$
- If x`.rank` $= y$`.rank`:
  x`.rank` $:= x$`.rank + 1`

Union-Find Operations

Find(x): $O(\log n)$

- If x.parent points to x, return x.
- Else Find(x.parent)
- $O(\log n)$ requires balanced trees.
- $O(\alpha(n))$ with path compression.

Union(x,y): $O(1)$

- (WLOG) x.rank $\geq y$.rank:
  y.parent $= x$
- If x.rank $= y$.rank:
  x.rank $:= x$.rank + 1
- By using rank, we maintain balanced sets if we start with balanced sets.

Implementing Kruskal's Algorithm

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Key Operations

- Sorting the edges: ($O(m \log m)$ and, since $m \leq n^2$, $O(m \log n)$).
- Maintain sets of connected components that we merge.
- Initialize one set per node: $O(n)$.

Union-Find Data Structure
TH: How many Find and Unions?

- `Find(x)`: Finds the set containing x.
- `Union(x,y)`: Joins two sets x and y.

Implementing Kruskal's Algorithm

Kruskal's (1956) Algorithm

- Sort edges by cost from lowest to highest.
- Insert edges unless insertion would create a cycle.

Key Operations

- Sorting the edges: $(O(m \log m)$ and, since $m \leq n^2$, $O(m \log n))$.
- Maintain sets of connected components that we merge.
- Initialize one set per node: $O(n)$.

Union-Find Data Structure

- `Find(x)`: $2m$ times $O(\log n)$ (can be $O(\alpha(n))$).
- `Union(x,y)`: $n - 1$ times $O(1)$.

Graph Exploration Overview

BFS and DFS

- Traverses a graph G starting from some node s.
- Builds a tree T.
- No guarantee on any distance measure.

Graph Exploration Overview

BFS and DFS

- Traverses a graph G starting from some node s.
- Builds a tree T.
- No guarantee on any distance measure.

Dijktra's

- Traverses a graph starting from some node s.
- Builds a tree T.
- All s to u paths in T are the shortest such path in G.

Graph Exploration Overview

BFS and DFS

- Traverses a graph G starting from some node s.
- Builds a tree T.
- No guarantee on any distance measure.

Dijktra's

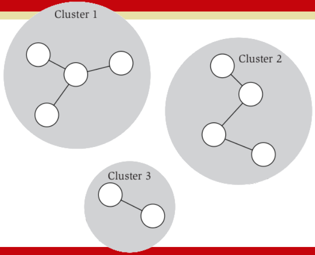
- Traverses a graph starting from some node s.
- Builds a tree T.
- All s to u paths in T are the shortest such path in G.

MST Algorithms

- Explores a graph G edges.
- Builds a tree T.
- T is minimum cost to connect all nodes in G.

Clustering

k-Clustering



Maximizing Spacing Problem

- A universe $\mathcal{U} := \{p_1, \ldots, p_n\}$ of n objects.
- Distance function $d : \mathcal{U} \times \mathcal{U} \to \mathbb{R}$ such that, for all $p_i, p_j \in \mathcal{U}$:
  - $d(p_i, p_i) = 0$
  - $d(p_i, p_j) > 0$
  - $d(p_i, p_j) = d(p_j, p_i)$
- Objective: Partition $\mathcal{U}$ into k non-empty groups $\mathcal{C} := C_1, \ldots, C_k$ with maximum spacing:

$$\text{maximize} \min_{C_i, C_j \in \mathcal{C}} \min_{u \in C_i, v \in C_j} d(u, v)$$

Algorithm Design

🤔 What greedy approach might work?

Algorithm Design

Algorithm

- Build an MST.
- Remove $k - 1$ largest edges.

Algorithm Design

Algorithm

- Build an MST.
- Remove $k-1$ largest edges.

k-Clusters at max spacing?

- Start with a tree, remove $k-1$ edges: We get a forest of k trees.
- By definition largest edges are removed so max spacing.

Algorithm Design

Algorithm

- Build an MST.
- Remove $k-1$ largest edges.

k-Clusters at max spacing?

- Start with a tree, remove $k-1$ edges: We get a forest of k trees.
- By definition largest edges are removed so max spacing.

🤔Which MST algorithm?

Algorithm Design

Algorithm

- Build an MST.
- Remove $k-1$ largest edges.

k-Clusters at max spacing?

- Start with a tree, remove $k-1$ edges: We get a forest of k trees.
- By definition largest edges are removed so max spacing.

Which MST algorithm?

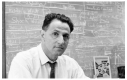
Kruskal's ($O(m \log n)$ which is $O(n^2 \log n)$ for clustering):

- Merge sets from lowest to most expensive edges.
- Stop when we have k sets.

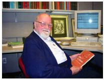
Appendix

References

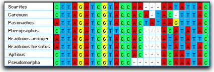
Image Sources I



https://medium.com/neurosapiens/2-dynamic-programming-9177012dcdd



https://angelberh7.wordpress.com/2014/10/08/biografia-de-lester-randolph-ford-jr/



http://www.sequence-alignment.com/

https://medium.com/koderunners/genetic-algorithm-part-3-knapsack-problem-b59035

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

https://brand.wisc.edu/web/logos/

Image Sources II

https://www.pngfind.com/mpng/mTJmbx_
spongebob-squarepants-png-image-spongebob-cartoo

https://www.pngfind.com/mpng/xhJRmT_
cheshire-cat-vintage-drawing-alice-in-wonderland