Shortest-paths problem

**Problem.** Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, and destination $t \in V$, find the shortest directed path from $s$ to $t$.

![Diagram of shortest paths problem]

Length of path $= 9 + 4 + 1 + 11 = 25$

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Dijkstra's algorithm

**Greedy approach.** Maintain a set of explored nodes $S$ for which
algorithm has determined the shortest path distance $d(u)$ from $s$ to $u$.

- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes
  $$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,$$
  shortest path to some node $u$ in explored part,
  followed by a single edge $(u,v)$

---

**Dijkstra's algorithm**

**Greedy approach.** Maintain a set of explored nodes $S$ for which
algorithm has determined the shortest path distance $d(u)$ from $s$ to $u$.

- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes
  $$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,$$
  shortest path to some node $u$ in explored part,
  followed by a single edge $(u,v)$
  add $v$ to $S$, and set $d(v) = \pi(v)$.

---

Dijkstra = Special case of $A^*$

$$d(v) + 1$$
Dijkstra’s algorithm: proof of correctness

Invariant. For each node \( u \in S \), \( d(u) \) is the length of the shortest \( s \rightarrow u \) path.

Pf. [by induction on \( |S| \)]

Base case: \( |S| = 1 \) is easy since \( S = \{ s \} \) and \( d(s) = 0 \).

Inductive hypothesis: Assume true for \( |S| = k \geq 1 \).

• Let \( v \) be next node added to \( S \), and let \((u, v)\) be the final edge.
• The shortest \( s \rightarrow u \) path plus \((u, v)\) is an \( s \rightarrow v \) path of length \( \pi(v) \).
• Consider any \( s \rightarrow v \) path \( P \). We show that it is no shorter than \( \pi(v) \).
• Let \((x, y)\) be the first edge in \( P \) that leaves \( S \), and let \( p' \) be the subpath to \( x \).
• \( P \) is already too long as soon as it reaches \( y \).

\[
\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v).
\]

Dijkstra’s algorithm: efficient implementation

Critical optimization 1. For each unexplored node \( v \), explicitly maintain \( \pi(v) \) instead of computing directly from formula:

\[
\pi(v) = \min_{\ell(u, v) \in S} d(u) + \ell(u, v).
\]

• For each \( v \notin S \), \( \pi(v) \) can only decrease (because \( S \) only increases).
• More specifically, suppose \( u \) is added to \( S \) and there is an edge \((u, v)\) leaving \( u \). Then, it suffices to update:

\[
\pi(v) = \min \{ \pi(v), d(u) + \ell(u, v) \}.
\]

Critical optimization 2. Use a priority queue to choose the unexplored node that minimizes \( \pi(v) \).

Over all updates, \# updates is \( O(n) \).
Dijkstra’s algorithm: efficient implementation

Implementation.
• Algorithm stores $d(v)$ for each explored node $v$.
• Priority queue stores $π(v)$ for each unexplored node $v$.
• Recall: $d(u) = π(u)$ when $u$ is deleted from priority queue.

\[
\text{DIJKSTRA}(V, E, s)
\]

Create an empty priority queue.

\[O(1) + O(n) + O(\log n)\]

For each $v ≠ s$:
\[d(v) ← ∞, d(s) ← 0.\]

For each $v ∈ V$:
\[\text{Insert } v \text{ with key } d(v) \text{ into priority queue.}\]

\[O(\log n)\]

While (the priority queue is not empty)
\[u ← \text{delete-min from priority queue.}\]

\[O(1)\]

For each edge $(u, v) ∈ E$ leaving $u$:
\[O(1)\]

If $d(v) > d(u) + ℓ(u, v)$
\[O(1)\]

Decrease-key of $v$ to $d(u) + ℓ(u, v)$ in priority queue.
\[O(1)\]

\[O(1) + O(n) + O(\log n)\]

\[O(\log n)\]

\[≤ \text{overall, } ≤ n \text{ times: } O(n \cdot \log n)\] time

Priority queue data type

A min-oriented priority queue supports the following core operations:
• MAKE-HEAP(): create an empty heap.
• INSERT($H, x$): insert an element $x$ into the heap.
• EXTRACT-MIN($H$): remove and return an element with the smallest key.
• DECREASE-KEY($H, x, k$): decrease the key of element $x$ to $k$.

The following operations are also useful:
• IS-EMPTY($H$): is the heap empty?
• FIND-MIN($H$): return an element with smallest key.
• DELETE($H, x$): delete element $x$ from the heap.
• MELD($H_1, H_2$): replace heaps $H_1$ and $H_2$ with their union.

Note. Each element contains a key (duplicate keys are permitted) from a totally-ordered universe.
Priority queue applications

Applications.
- A* search.
- Heapsort.
- Online median.
- Huffman encoding.
- Prim’s MST algorithm.
- Discrete event-driven simulation.
- Network bandwidth management.
- Dijkstra’s shortest-paths algorithm.
- ...

Complete binary tree

Binary tree. Empty or node with links to two disjoint binary trees.

Complete tree. Perfectly balanced, except for bottom level.

Property. Height of complete binary tree with \( n \) nodes is \( \lceil \log_2 n \rceil \).

Pf. Height increases (by 1) only when \( n \) is a power of 2. •
**Binary heap**

**Binary heap.** Heap-ordered complete binary tree.

**Heap-ordered tree.** For each child, the key in child $\geq$ key in parent.

![Binary heap diagram](image)

**Explicit binary heap**

**Pointer representation.** Each node has a pointer to parent and two children.
- Maintain number of elements $n$.
- Maintain pointer to root node.
- Can find pointer to last node or next node in $O(\log n)$ time.

![Explicit binary heap diagram](image)
Implicit binary heap

**Array representation.** Indices start at 1.
- Take nodes in **level** order.
- Parent of node at $k$ is at $\lfloor k / 2 \rfloor$.
- Children of node at $k$ are at $2k$ and $2k + 1$.

```
  1
 / \
2   3
/   /
4   5
\   \
12  10
\  /  \
9  18
\//
21 17
```

---

Binary heap: insert

**Insert.** Add element in new node at end; repeatedly exchange new element with element in its parent until heap order is restored.
Binary heap: extract the minimum

**Extract min.** Exchange element in root node with last node; repeatedly exchange element in root with its smaller child until heap order is restored.

Binary heap: decrease key

**Decrease key.** Given a handle to node, repeatedly exchange element with its parent until heap order is restored.

decrease key of node x to 11
Binary heap: analysis

**Theorem.** In an *implicit* binary heap, any sequence of \( m \) **insert**, **extract-min**, and **decrease-key** operations with \( n \) **insert** operations takes \( O(m \log n) \) time.

**Pf.**
- Each heap op touches nodes only on a path from the root to a leaf; the height of the tree is at most \( \log n \).
- The total cost of expanding and contracting the arrays is \( O(n) \).

**Theorem.** In an *explicit* binary heap with \( n \) nodes, the operations **insert**, **decrease-key**, and **extract-min** take \( O(\log n) \) time in the worst case.

Binary heap: find-min

**Find the minimum.** Return element in the root node.
Binary heap: delete

**Delete.** Given a handle to a node, exchange element in node with last node; either swim down or sink up the node until heap order is restored.

delete node x or y

![Binary heap diagram]

---

Priority queues performance cost summary

<table>
<thead>
<tr>
<th>operation</th>
<th>linked list</th>
<th>binary heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>IS-EMPTY</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>MELOD</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>FIND-MIN</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

---

Min Spanning Tree (MST)
Spanning tree properties

**Proposition.** Let $T = (V, F)$ be a subgraph of $G = (V, E)$. TFAE:
- $T$ is a spanning tree of $G$.
- $T$ is acyclic and connected.
- $T$ is connected and has $n - 1$ edges.
- $T$ is acyclic and has $n - 1$ edges.
- $T$ is minimally connected: removal of any edge disconnects it.
- $T$ is maximally acyclic: addition of any edge creates a cycle.
- $T$ has a unique simple path between every pair of nodes.

![Graph](image)

Minimum spanning tree

Given a connected graph $G = (V, E)$ with edge costs $c_e$, an MST is a subset of the edges $T \subseteq E$ such that $T$ is a spanning tree whose sum of edge costs is minimized.

![Graph](image)

MST cost = $50 = 4 + 6 + 8 + 5 + 11 + 9 + 7$

**Cayley’s theorem.** There are $n^{n-2}$ spanning trees of $K_n$.
Applications

MST is fundamental problem with diverse applications.
- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).

Fundamental cycle

**Fundamental cycle.**
- Adding any non-tree edge $e$ to a spanning tree $T$ forms unique cycle $C$.
- Deleting any edge $f \in C$ from $T \cup \{ e \}$ results in new spanning tree.

![Diagram of a graph showing a fundamental cycle](image)

**Observation.** If $e_i < e_j$, then $T$ is not an MST.

Kruskal's Algorithm: $G=(V,E)$, edge costs $cost(e_i)$

sort $e_1, e_2, ..., e_m$ so that $cost(e_1) \leq cost(e_2) \leq ... \leq cost(e_m)$

$T := \emptyset$

for $i = 1$ to $m$
  if $T \cup \{ e_i \}$ has no cycle then $T := T \cup \{ e_i \}$
endif
endfor

return $T$
Example:

\[ T_{opt} \]

\[ \text{cost} = 11 \]

Correctness of Kruskal's algo:

\[ T_0 = \emptyset \]

\[ T_i = T \text{ after iteration } i \]

Need to prove: \( T_m \) is MST.

Proof strategy:

Argue that each \( T_i \) can be
Argue that each $T_i$ can be extended to some MST, using some of the edges $e_{i+1}, e_{i+2}, \ldots, e_m$.

**Defn:** $T_i$ is promising if there exists some MST $T_{opt}$ s.t.

$$T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, \ldots, e_m\}.$$

**Claim:** $\forall i$, $T_i$ is promising.

$$\Rightarrow T_m \text{ is promising}$$

$$\Rightarrow \exists \text{ MST } T_{opt}: T_m \subseteq T_{opt} \subseteq T_m \cup \emptyset$$

$$\Rightarrow T_m \text{ is MST. } T_m = T_{opt}$$

**Pf of Claim:** (by induction on $i$)

- **Base case:** $i = 0$. $T_0 = \emptyset$ is promising:

  $$T_0 \subseteq T_{opt} \subseteq T_0 \cup \{e_{i+1}, \ldots, e_m\} \text{ for some MST}$$
Induction step: Assume \( T_i \) is promising \((i \geq 0)\). Prove \( T_{i+1} \) is promising.

Case 1: \( e_{i+1} \notin T_{i+1} \). \( e_{i+1} \notin T_{opt} \) \( \Rightarrow T_{i+1} \leq T_{opt} \leq T_i \cup \{e_{i+1}\} \)

Case 2: \( e_{i+1} \in T_{i+1} \)

Case 2.1: \( e_{i+1} \in T_{opt} \).

Case 2.2: \( e_{i+1} \notin T_{opt} \).

\( T_i \leq T_{opt} \Rightarrow T_{i} \cup \{e_{i+1}\} \) has a cycle

\( T_{opt} \cup \{e_{i+1}\} \) has a cycle

**Fundamental cycle**

**Fundamental cycle.**

- Adding any non-tree edge \( e \) to a spanning tree \( T \) forms unique cycle \( C \).
- Deleting any edge \( f \in C \) from \( T \cup \{e\} \) results in new spanning tree.

**Observation.** If \( c_e < c_j \), then \( T \) is not an MST.
Define $T'_\text{opt} = T_{\text{opt}} \cup \{e_{i+1} j\} - \{e_j\}$.

1. $T'_\text{opt}$ is a spanning tree.
2. $\text{cost}(T'_\text{opt}) = \text{cost}(T_{\text{opt}}) + c(e_{i+1}) - c(e_j) \leq 0$ by the edge ordering.

Finally, observe $T_{i+1} \subseteq T'_\text{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$. So $T_{i+1}$ is promising. ☑

\underline{Prim's Algorithm}: $G = (V, E)$
- $S = \{s\}$, $T = \emptyset$
- repeat $|V| - 1$ steps
  - find $e = (u, v) \in E$ with $u \in S$, $v \notin S$
find $e = (u, v) \in E$ with $v \in \text{min cost of } S$ & add $v$ to $S$; add $e$ to $T$ 

do repeat

end repeat

return $T$

Ex:

Correctness of Prim’s Algo

Using promising

$S_i, T_i = S, T$ after iteration $i$
iteration \( i \)

Set: \((S_i, T_i)\) is promising

If it can be extended to some \(\text{MST}\), using edges with at most one endpoint in \(S_i\)

Claim: \(\forall i, (S_i, T_i)\) is promising

Cor: \((S_{n-1}, T_{n-1})\) is \(\text{MST}\)

Proof of Claim: By induction on \(i\).

Base case: \(i = 0\)

\((S_0, T_0)\) is promising

Ind. Step: Assume \((S_i, T_i)\) is promising

Prove \((S_{i+1}, T_{i+1})\) is promising

\(\text{MST}\) \(T_{opt}\) extending \(T_i\)

Edge \(e\) is added to \(T_i\) to form \(T_{i+1}\).

Case 1: \(e \in T_{opt}\)

\(\checkmark\)
Case 2: \( e \notin T_{opt} \)

**Fundamental cycle**

- Adding any non-tree edge \( e \) to a spanning tree \( T \) forms unique cycle \( C \).
- Deleting any edge \( f \in C \) from \( T \cup \{ e \} \) results in a new spanning tree.

![Diagram showing the fundamental cycle](image)

**Observation.** If \( c_e < c_f \), then \( T \) is not an MST.

---

**Prüfer’s algorithm: implementation**

**Theorem.** Prüfer’s algorithm can be implemented in \( O(m \log n) \) time.

**Pf.** Implementation almost identical to Dijkstra’s algorithm.

\[ d(v) = \text{weight of cheapest known edge between } v \text{ and } S \]

<table>
<thead>
<tr>
<th>( \text{PRÜFER} (V, E, c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_1 \text{ create an empty priority queue.} )</td>
</tr>
<tr>
<td>any node in ( V ).</td>
</tr>
<tr>
<td>FOR EACH ( v \neq s : d(v) \leftarrow \infty ); ( d(s) \leftarrow 0 ).</td>
</tr>
<tr>
<td>FOR EACH ( v ) : insert ( v ) with key ( d(v) ) into priority queue.</td>
</tr>
<tr>
<td>WHILE (the priority queue is not empty)</td>
</tr>
<tr>
<td>( u \leftarrow \text{delete-min} ) from priority queue.</td>
</tr>
<tr>
<td>FOR EACH edge ( (u, v) \in E ) incident to ( u ):</td>
</tr>
<tr>
<td>IF ( d(v) &gt; c(u, v) )</td>
</tr>
<tr>
<td>decrease-key of ( v ) to ( c(u, v) ) in priority queue.</td>
</tr>
<tr>
<td>( d(v) \leftarrow c(u, v) ).</td>
</tr>
</tbody>
</table>

Can assume: \( m > n - 1 \)
Kruskal’s algorithm: implementation

Theorem. Kruskal’s algorithm can be implemented in $O(n \log m)$ time.

- Sort edges by weight.
- Use union-find data structure to dynamically maintain connected components.

Kruskal($V, E, c$)

1. $S \leftarrow \emptyset$
2. Foreach $v \in V$: MAKESET($v$).
3. For $i \leftarrow 1$ To $m$
   a. $(u, v) \leftarrow e_i$
   b. If FINDSET($u$) $\neq$ FINDSET($v$)
      i. $S \leftarrow S \cup \{e_i\}$
      ii. UNION($u, v$).
4. RETURN $S$

Disjoint-sets data structure

Representation. Represent each set as a tree of elements.

- Each element has a parent pointer in the tree.
- The root serves as the canonical element.
- FIND($x$). Find the root of the tree containing $x$.
- UNION($x, y$). Make the root of one tree point to root of other tree.

Note. For brevity, we suppress arrows and self loops in figures.
Link-by-size

Link-by-size. Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

union(7, 3)

size = 4

4
3 8 9

size = 6

6
0 2 5
1 7

Link-by-size

Link-by-size. Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

union(7, 3)

size = 10

6
0 2 5
1 7
Link-by-size

**Link-by-size.** Maintain a subtree count for each node, initially 1.
Link root of smaller tree to root of larger tree (breaking ties arbitrarily).

### Make-Set (x)

- `parent(x) ← x.`
- `size(x) ← 1.`

### Find (x)

- **While** `(x ≠ parent(x))`
  - `x ← parent(x).`
- **Return** `x`.

### Union-by-Size (x, y)

- `r ← Find(x).`
- `s ← Find(y).`
- **If** `(r = s)` **Return**.
- **Else If** `(size(r) > size(s))`
  - `parent(s) ← r.`
  - `size(r) ← size(r) + size(s).`
- **Else**
  - `parent(r) ← s.`
  - `size(s) ← size(r) + size(s).`

---

**Link-by-size: analysis**

**Property.** Using link-by-size, for every root node `r`:
- `size(r) ≥ 2^{\text{height}(r)}`.

**Proof.** (by induction on number of links)

- **Base case.** Singleton tree has size 1 and height 0.
- **Inductive hypothesis.** Assume true after first `i` links.
- **Tree rooted at r changes only when a smaller tree rooted at s is linked into r.**

- **Case 1.** `[\text{height}(r) > \text{height}(s)]
  - `size'(r) ≥ size(r)``
  - `size'(r) ≥ 2^{\text{height}(r)}`
  - `= 2^{\text{height}(r)},` inductive hypothesis

---

**Note:**

- `height(r) ≤ \log_{\log_{\log_{\log_{\log_{\log}}}}(\text{size}(r))}`
Link-by-size: analysis

**Property.** Using link-by-size, for every root node \( r \), \( \text{size}(r) \geq 2^{\text{height}(r)} \).

**Pf.** [by induction on number of links]

- Base case: singleton tree has size 1 and height 0.
- Inductive hypothesis: assume true after first \( i \) links.
- Tree rooted at \( r \) changes only when a smaller tree rooted at \( s \) is linked into \( r \).

**Case 2.** [\( \text{height}(r) \leq \text{height}(s) \)]

\[
\text{size}'(r) = \text{size}(r) + \text{size}(s) \\
\geq 2 \cdot \text{size}(s) \\
\geq 2 \cdot 2^{\text{height}(s)} \\
= 2^{\text{height}(s) + 1} \\
= 2^{\text{height}(r)}. \]

**Theorem.** Using link-by-size, any UNION or FIND operations takes \( O(\log n) \) time in the worst case, where \( n \) is the number of elements.

**Pf.**

- The running time of each operation is bounded by the tree height.
- By the previous property, the height is \( \leq \lfloor \log_2 n \rfloor \).

\[
\log n = \log_2 n \\
\text{size}(r) \geq 2^{\text{height}(r)} \\
\log \text{size}(r) \geq \text{height}(r). \]
A matching lower bound

**Theorem.** Using link-by-size, a tree with $n$ nodes can have height $\leq \lg n$.

**Pf.**
- Arrange $2^k - 1$ calls to UNION to form a binomial tree of order $k$.
- An order-$k$ binomial tree has $2^k$ nodes and height $k$. •