Greedy Algorithm

September 27, 2019
Lecture Overview

- Greedy Algorithm
- Activity Selection Problem
- Minimum Spanning Tree
- Kruskal’s Algorithm
- Prim’s Algorithm
- Shortest Path
Greedy Algorithms

- A greedy algorithm is an algorithm that constructs an object one step at a time, at each step choosing the locally best option.

- In some cases, greedy algorithms construct the globally best object by repeatedly choosing the locally best option.
Greedy Advantages vs Challenges

- Greedy algorithms have several advantages over other algorithmic approaches:
  - **Simplicity**: Greedy algorithms are often easier to describe and code up than other algorithms.
  - **Efficiency**: Greedy algorithms can often be implemented more efficiently than other algorithms.

- Greedy algorithms have several drawbacks:
  - **Hard to design**: Once you have found the right greedy approach, designing greedy algorithms can be easy. However, finding the right approach can be hard.
  - **Hard to verify**: Showing a greedy algorithm is correct often requires a nuanced argument.
Activity Selection Problem

**Instance** $n$ jobs $J = \{1, 2, \ldots, n\}$
For job $j$, start time: $s_j$ and finish time: $f_j$

**Feasible Solution**
Two jobs compatible if they don’t overlap.
Find a subset of mutually compatible jobs.

**Value** Maximize the size of mutually compatible jobs.
Example

The diagram illustrates the concept of compatibility. Green bars represent one set of intervals, and blue bars represent another set. Intervals are considered compatible if they do not overlap. Intervals are considered not compatible if they overlap.
Greedy Rule

**Greedy algorithm:**

Select intervals one after another using some rule.
Rule 1

Select the interval that starts earliest (but not overlapping the already chosen intervals)

**Counterexample**
Rule 2

Select the interval which is shortest (but not overlapping the already chosen intervals)

**Counterexample**

![Diagram showing intervals on a number line]
Rule 3

Select the interval with the fewest conflicts with other remaining intervals (but still not overlapping the already chosen intervals)

**Counterexample**
Rule 4

Select the interval which ends first (but still not overlapping the already chosen intervals)

**VISUALIZE**
Greedy Algorithm

1. Initially $J$ be the set of all requests and let $A$ be an empty.
2. While $J$ is not yet empty
   2.1 Choose a request $j \in J$ that has the smallest finish time
   2.2 Add request $j$ to $A$
   2.3 Delete all requests from $J$ that are not compatible with request $j$
3. EndWhile
4. Return the set $A$ as the set of accepted requests
Correctness

Algorithm gives non-overlapping intervals:

obvious, since we always choose an interval which does not overlap the previously chosen intervals

The solution is optimal:

$A \leftarrow$ set of intervals obtained by the algorithm,
$Opt \leftarrow$ largest set of pairwise non-overlapping intervals.

We show that,

$A$ must be as large as $Opt$. 
Correctness Contd...

\[ A = \{A_1, A_2, \ldots, A_k\} \quad \text{ Opt } = \{O_1, O_2, \ldots, O_m\} \]

Elements in \( A \) and \( \text{ Opt } \) are sorted.

By definition of \( \text{ Opt } \) we have \( k \leq m \).

**Claim:** for every \( i \leq k \), \( A_i \) finishes not later than \( O_i \).

**Proof:** by induction.

For \( i = 1 \) by definition of a step in the algorithm.

Suppose that \( A_{i-1} \) finishes not later than \( O_{i-1} \).

From the definition of a step in the algorithm: \( A_i \) is the first interval that finishes after \( A_{i-1} \) and does not overlap it.

If \( O_i \) finished before \( A_i \) then it would overlap with some of the previous \( A_1, \ldots, A_{i-1} \) and consequently - by the inductive assumption - it would overlap \( O_{i-1} \), which is a contradiction.
**Theorem:** A is the optimal solution.

**Proof:** we show that $k = m$. Suppose to the contrary that $k < m$. We have that $A_k$ finishes not later than $O_k$. Hence we could add $O_{k+1}$ to $A$ and obtain bigger solution by the algorithm - contradiction.

![Diagram showing timeline of tasks and operations]

- $A_{k-1}$
- $O_{k-1}$
- $A_k$
- $O_k$
- $O_{k+1}$
Implementation and Time Complexity

**Explanation 1**

1. Reorder jobs so that they are in increasing order of finish time.
2. Store starting time of jobs in an array $S$.
3. Always select first interval. Let finish time be $j$.
4. Iterate over $S$ to find the first index $i$ such that $S[i] \geq j$.

**Explanation 2**

1. Sorting intervals according to the right-most ends
2. For every consecutive interval:
   2.1 If the left-most end is after the right-most end of the last selected interval then we select this interval
   2.2 Otherwise we skip it and go to the next interval

**Time Complexity:**

Running time is $O(n \log n)$, dominated by sorting. Actually, $O(n \log n + n)$
Minimum Spanning Tree

Greedy Algorithms for Minimum Spanning Trees

**INSTANCE** Connected graph $G = (V, E)$ with edge costs

**Feasible Solution** Find $T \subseteq E$ such that $(V, T)$ is connected

**Value** total cost of all edges in $T$ is smallest

$T$ is the *minimum spanning tree (MST)* of $G$
**Minimum Spanning Tree**

**Greedy Algorithms for Minimum Spanning Trees**

**Instance** Connected graph $G = (V, E)$ with edge costs

**Feasible Solution** Find $T \subseteq E$ such that $(V, T)$ is connected

**Value** total cost of all edges in $T$ is smallest

$T$ is the *minimum spanning tree (MST)* of $G$
Applications

- **Network Design**
  - Designing networks with minimum cost but maximum connectivity
- **Approximation algorithms**
  - Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.
- **Cluster Analysis**
Greedy Template

Initially \( E \) is the set of all edges in \( G \)

\( T \) is empty (* \( T \) will store edges of a MST *)

while \( E \) is not empty do

\hspace{1cm} choose \( e \in E \)

\hspace{1cm} if (\( e \) satisfies condition)

\hspace{1.5cm} add \( e \) to \( T \)

return the set \( T \)

**Main Task:** In what order should edges be processed?
When should we add edge to spanning tree?
Kruskal’s Algorithm

Process edges of $G$ in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Graph $G$  

MST $T$ of $G$
Kruskal’s Algorithm

Process edges of $G$ in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Graph $G$

MST $T$ of $G$
Kruskal’s Algorithm

Process edges of $G$ in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Graph $G$

MST $T$ of $G$
Kruskal’s Algorithm

Process edges of $G$ in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Graph $G$

MST $T$ of $G$
Kruskal’s Algorithm

Process edges of $G$ in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Graph $G$

MST $T$ of $G$
Kruskal’s Algorithm

Process edges of $G$ in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Graph $G$

MST $T$ of $G$
Kruskal’s Algorithm

Process edges of \( G \) in the order of their costs (starting from the least) and add edges to \( T \) as long as they don’t form a cycle.

Graph \( G \)

MST \( T \) of \( G \)
**Prim’s Algorithm**

$T$ maintained by algorithm will be a tree.
Start with a node in $T$.
In each iteration, pick edge with least attachment cost to $T$.

---

**Graph $G$**

- $v_1$ connected to $v_2$: cost 20
- $v_1$ connected to $v_6$: cost 23
- $v_1$ connected to $v_7$: cost 1
- $v_2$ connected to $v_3$: cost 15
- $v_2$ connected to $v_4$: cost 9
- $v_2$ connected to $v_5$: cost 3
- $v_6$ connected to $v_7$: cost 4
- $v_6$ connected to $v_5$: cost 36
- $v_7$ connected to $v_3$: cost 16
- $v_7$ connected to $v_4$: cost 25
- $v_7$ connected to $v_5$: cost 28
- $v_7$ connected to $v_6$: cost 17

**MST $T$ of $G$**

- $v_1$ and $v_2$
- $v_6$ and $v_7$
- $v_3$ and $v_4$
- $v_5$ and $v_6$

---

21/48
Prim’s Algorithm

$T$ maintained by algorithm will be a tree.
Start with a node in $T$.
In each iteration, pick edge with least attachment cost to $T$.
Prim’s Algorithm

$T$ maintained by algorithm will be a tree.
Start with a node in $T$.
In each iteration, pick edge with least attachment cost to $T$.
Prim’s Algorithm

\( T \) maintained by algorithm will be a tree.
Start with a node in \( T \).
In each iteration, pick edge with least attachment cost to \( T \).

Graph \( G \)

MST \( T \) of \( G \)
Prim’s Algorithm

$T$ maintained by algorithm will be a tree. Start with a node in $T$. In each iteration, pick edge with least attachment cost to $T$.

Graph $G$

MST $T$ of $G$
Prim’s Algorithm

$T$ maintained by algorithm will be a tree.
Start with a node in $T$.
In each iteration, pick edge with least attachment cost to $T$.

Graph $G$

MST $T$ of $G$
Prim’s Algorithm

$T$ maintained by algorithm will be a tree. Start with a node in $T$. In each iteration, pick edge with least attachment cost to $T$.

Graph $G$

MST $T$ of $G$
Reverse Delete Algorithm

Initially $E$ is the set of all edges in $G$
$T$ is $E$ (* $T$ will store edges of a MST *)
while $E$ is not empty do
    choose $e \in E$ of largest cost
    if removing $e$ does not disconnect $T$ then
        remove $e$ from $T$
return the set $T$

Returns a minimum spanning tree.
Correctness of MST Algorithms

- Many different MST algorithms
- All of them rely on some basic properties of MSTs, in particular the Cut Property to be seen shortly.
Assumption

Edge costs are distinct, that is no two edge costs are equal.
Given a graph $G = (V, E)$, a cut is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.

Edges having an end point on both sides are the edges of the cut.

A cut edge is crossing the cut.
**Safe and Unsafe Edges**

**Safe Edge**
An edge $e = (u, v)$ is a **safe** edge if there is some partition of $V$ into $S$ and $V \setminus S$ and $e$ is the unique minimum cost edge crossing $S$ (one end in $S$ and the other in $V \setminus S$).

**Unsafe Edge**
An edge $e = (u, v)$ is an **unsafe** edge if there is some cycle $C$ such that $e$ is the unique maximum cost edge in $C$.

**Proposition**
If edge costs are distinct then every edge is either safe or unsafe.

WHY??
Safe Edge

Every cut identify one safe edge.

...the cheapest edge in the cut.
Unsafe Edge

Every cycle identify one unsafe edge.

...the most expensive edge in the cycle.
**Cut Property**

Let $S$ be any subset of nodes and let $e = (v, w)$ be the minimum cost edge with one end in $S$ and the other in $V \setminus S$. Then every minimum spanning tree contains the edge.

**Proof (by contradiction)**

Suppose $e = (v, w)$ is not in MST $T$ and $e$ is min weight edge in cut $(S, V \setminus S)$. Wlog $v \in S$.

$T$ is spanning tree: there is a unique path $P$ from $v$ to $w$ in $T$.

Let $w' \in V \setminus S$ be the first vertex on $P$ in $V \setminus S$; let $v' \in V \setminus S$ be the vertex just before it on $P$, let $e' = (v', w')$.

$T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?)
Cut Property and it’s Proof

**Cut Property**
Let $S$ be any subset of nodes and let $e = (v, w)$ be the minimum cost edge with one end in $S$ and the other in $V \setminus S$. Then every minimum spanning tree contains the edge.

**Proof** (by contradiction)

- Suppose $e = (v, w)$ is not in MST $T$ and $e$ is minimum weight edge in cut $(S, V \setminus S)$. Wlog $v \in S$.
- $T$ is spanning tree: there is a unique path $P$ from $v$ to $w$ in $T$.
- Let $w'$ be the first vertex on $P$ in $V \setminus S$; let $v'$ be the vertex just before it on $P$, let $e' = (v', w')$.
- $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?)
**Claim**

\[ T' = (T \setminus \{e'\}) \cup \{e\} \]

is spanning tree

**Proof**

- **\( T' \) is connected**

- **\( T' \) is a tree**
**Claim**

\[ T' = (T \setminus \{e'\}) \cup \{e\} \] is spanning tree

**Proof**

\[ T' \] is connected

Removed \( e' = (v', w') \) from \( T \) but \( v' \) and \( w' \) are connected by the path \( P - e' + e \).

Hence \( T' \) is connected if \( T \) is.

\[ T' \] is a tree

\( T' \) is connected and has \( n - 1 \) edges (since \( T \) has \( n - 1 \) edges) and hence \( T' \) is a tree.
Prim's Algorithm

Prım_CompûteMST

$E$ is the set of all edges in $G$

$S = \{1\}$ (* take any vertex *)

$T$ is empty (* $T$ will store edges of a MST *)

while $S \neq V$ do

pick $e = (v, w) \in E$ such that

$v \in S$ and $w \in V \setminus S$

$e$ has minimum cost

$T = T \cup e$

$S = S \cup w$

return the set $T$

Running Time:

- $n$: number of vertices, $m$: number of edges.
- Number of iterations $= O(n)$
- Picking $e$ is $O(m)$
- Total time $O(nm)$
Implementation: Kruskal’s Algorithm

Kruskal_ComputeMST

$E$ is the set of all edges in $G$
$T$ is empty (* $T$ will store edges of a MST *)

while $E$ is not empty do
    choose $e \in E$ of minimum cost
    if ($T \cup \{e\}$ does not have cycle)
        add $e$ to $T$

return the set $T$

Running Time:

- $n$: number of vertices, $m$: number of edges.
- Presort edges based on cost.
- Choosing minimum can be done in $O(1)$ time.
- Do BFS/DFS on $T \cup \{e\}$. Takes $O(n)$ time.
- Total time $O(m \log m) + O(nm) = O(mn)$
Data Structures for MST

- Priority Queue
- Union-Find

Prim’s Algorithm
- Using standard Heaps: $O((m + n) \log n)$
- Using Fibonacci Heaps: $O(n \log n + m)$.

Kruskal’s Algorithm
- $O(m \log m)$ time.
**Input: Shortest Path problems**

A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $w(e) = w(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.
3. Find shortest paths for all pairs of nodes.

Many applications ...
Consider a digraph $G = (V, E)$ with edge-weight function $w : E \rightarrow \mathbb{R}$. The weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is defined to be $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$.

**Example:**

![Diagram of a digraph with weighted edges](image)

$w(p) = 13$
Some terminologies contd...

A shortest path from \( u \) to \( v \) is a path of minimum weight from \( u \) to \( v \).

The shortest-path weight from \( u \) to \( v \) is defined as
\[
\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.
\]

**Note:** \( \delta(u, v) = \infty \) if no path from \( u \) to \( v \) exists.
Single-Source Shortest Path Problems

**Input:** Single-Source Shortest Path problems
A (undirected or directed) graph \( G = (V, E) \) with non-negative edge lengths (or costs). For edge \( e = (u, v) \), \( w(e) = w(u, v) \) is its length.

1. Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
2. Given node \( s \) find shortest paths from \( s \) to all other nodes.

▶ Restrict attention to directed graphs
▶ Undirected graph problem can be reduced to directed graph problem - how?
Single-Source Shortest Path Problems

**Input:** Single-Source Shortest Path problems

A (undirected or directed) graph $G = (V, E)$ with non-negative edge lengths (or costs). For edge $e = (u, v)$, $w(e) = w(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest paths from $s$ to all other nodes.

- Restrict attention to directed graphs
- Undirected graph problem can be reduced to directed graph problem - how?
  - Given undirected graph $G$, create a new directed graph $G'$ by replacing each edge $\{u, v\}$ in $G$ by $(u, v)$ and $(v, u)$ in $G'$
  - set $w(u, v) = w(v, u) = w(\{u, v\})$
  - Exercise: show reduction works
Special case: All edge lengths are 1
**Special case:** All edge lengths are 1

- Run \( \text{BFS}(s) \) to get shortest path distances from \( s \) to all other nodes.
- \( O(m + n) \) time algorithm.

**Special case:** Suppose \( w(e) \) is an integer for all \( e \).
Special case: All edge lengths are 1

- Run BFS(s) to get shortest path distances from s to all other nodes.
- $O(m + n)$ time algorithm.

Special case: Suppose $w(e)$ is an integer for all $e$
Can we use BFS?
Special case: All edge lengths are 1

- Run BFS($s$) to get shortest path distances from $s$ to all other nodes.
- $O(m + n)$ time algorithm.

Special case: Suppose $w(e)$ is an integer for all $e$
Can we use BFS?

Reduce to unit edge-length problem by placing $w(e) - 1$ dummy nodes on $e$
**Special case:** All edge lengths are 1

- Run $\text{BFS}(s)$ to get shortest path distances from $s$ to all other nodes.
- $O(m + n)$ time algorithm.

**Special case:** Suppose $w(e)$ is an integer for all $e$

Can we use BFS?

Reduce to unit edge-length problem by placing $w(e) - 1$ dummy nodes on $e$

Let $L = \max_e w(e)$. New graph has $O(mL)$ edges and $O(mL + n)$ nodes. BFS takes $O(mL + n)$ time. Not efficient if $L$ is large.
Towards an Algorithm

Why does BFS works?

BFS(\(s\)) explores nodes in increasing distance from

**Lemma:**

Let \(G\) be a directed graph with non-negative edge lengths. Let \(\text{dist}(s, v)\) denote the shortest path length from \(s\) to \(v\). If \(s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k\) is a shortest path from \(s\) to \(v_k\) then for \(1 \leq i < k\):

1. \(s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i\) is a shortest path from \(s\) to \(v_i\)

2. \(\text{dist}(s, v_i) \leq \text{dist}(s, v_k)\)

**Proof:** Suppose not. Then for some \(i < k\) there is a path \(P'\) from \(s\) to \(v_i\) of length strictly less than that of \(s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_i\). Then \(P'\) concatenated with \(v_i \rightarrow v_{i+1} \rightarrow \ldots \rightarrow v_k\) contains a strictly shorter path to \(v_k\) than \(s = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k\).
Theorem. A subpath of a shortest path is a shortest path.
THEOREM. A subpath of a shortest path is a shortest path.
Proof by Picture

Theorem. A subpath of a shortest path is a shortest path.
**Problem.** From a given source vertex $s \in V$, find the shortest-path weights $\delta(s, v)$ for all $v \in V$.

If all edge weights $w(u, v)$ are non-negative, all shortest-path weights must exist.

**IDEA:** Greedy.
Single-source shortest paths

**Problem.** From a given source vertex $s \in V$, find the shortest-path weights $\delta(s, v)$ for all $v \in V$.

If all edge weights $w(u, v)$ are non-negative, all shortest-path weights must exist.

**IDEA:** Greedy.

1. Maintain a set $S$ of vertices whose shortest-path distances from $s$ are known.
2. At each step add to $S$ the vertex $v \in V \setminus S$ whose distance estimate from $s$ is minimal.
3. Update the distance estimates of vertices adjacent to $v$. 
Dijkstra’s algorithm

DIJKSTRA\((G, w, s)\)

\(d[s] \leftarrow 0\)

for each \(v \in V \setminus \{s\}\)

\(\text{do } d[v] \leftarrow \infty\) \(\pi[v] = NIL\)

\(S \leftarrow \emptyset\)

\(Q \leftarrow V\) \(\triangleright Q\) is a priority queue maintaining \(V \setminus \{s\}\)

while \(Q \neq \emptyset\)

\(\text{do } u \leftarrow \text{EXTRACT-MIN}(Q)\)

\(S \leftarrow S \cup \{u\}\)

for each \(v \in Adj[u]\)

\(\text{do if } d[v] > d[u] + w(u, v)\)

\(\text{then } d[v] \leftarrow d[u] + w(u, v)\) \(\triangleright\) DECREASE-KEY

\(\pi[v] \leftarrow u\)
Example of Dijkstra’s algorithm
Example of Dijkstra’s algorithm

Initialize:

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>π</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>NIL</td>
</tr>
<tr>
<td>a</td>
<td>∞</td>
<td>NIL</td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td>NIL</td>
</tr>
<tr>
<td>c</td>
<td>∞</td>
<td>NIL</td>
</tr>
<tr>
<td>d</td>
<td>∞</td>
<td>NIL</td>
</tr>
</tbody>
</table>

\[ S = \{ \} \]
Example of Dijkstra’s algorithm

```
Example of Dijkstra's algorithm

EXTRACT-MIN(Q):

<table>
<thead>
<tr>
<th>Q</th>
<th>d</th>
<th>π</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>NIL</td>
</tr>
<tr>
<td>a</td>
<td>∞</td>
<td>NIL</td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td>NIL</td>
</tr>
<tr>
<td>c</td>
<td>∞</td>
<td>NIL</td>
</tr>
<tr>
<td>d</td>
<td>∞</td>
<td>NIL</td>
</tr>
</tbody>
</table>

S = {}"
```
Example of Dijkstra’s algorithm

Relax all edges leaving $s$

<table>
<thead>
<tr>
<th></th>
<th>$Q$</th>
<th>$d$</th>
<th>$\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>0</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>$a$</td>
<td>10</td>
<td>$s$</td>
<td>NIL</td>
</tr>
<tr>
<td>$b$</td>
<td>5</td>
<td>$s$</td>
<td>NIL</td>
</tr>
<tr>
<td>$c$</td>
<td>$\infty$</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>$d$</td>
<td>$\infty$</td>
<td>NIL</td>
<td>NIL</td>
</tr>
</tbody>
</table>

$S = \{s\}$
Example of Dijkstra’s algorithm

\[
\begin{array}{c|c|c}
Q & d & \pi \\
\hline
s & 0 & NIL \\
a & 10 & s \\
b & 5 & s \\
c & \infty & NIL \\
d & \infty & NIL \\
\end{array}
\]

\[
S = \{s\}
\]
Example of Dijkstra’s algorithm

Relax all edges leaving $b$

$S = \{s, b\}$
Example of Dijkstra’s algorithm

“d” ← EXTRACT-MIN(Q):

\[
\begin{array}{c|c|c}
Q & d & \pi \\
\hline
s & 0 & NIL \\
a & 10 & \times b \\
b & 5 & s \\
c & 14 & b \\
d & 7 & b \\
\end{array}
\]

\[ S = \{s, b\} \]
Example of Dijkstra’s algorithm

Relax all edges leaving $d$

<table>
<thead>
<tr>
<th>Q</th>
<th>$d$</th>
<th>$\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>0</td>
<td>NIL</td>
</tr>
<tr>
<td>$a$</td>
<td>10</td>
<td>$b$</td>
</tr>
<tr>
<td>$b$</td>
<td>5</td>
<td>$s$</td>
</tr>
<tr>
<td>$c$</td>
<td>14</td>
<td>$d$</td>
</tr>
<tr>
<td>$d$</td>
<td>7</td>
<td>$b$</td>
</tr>
</tbody>
</table>

$S = \{s, b, d\}$
Example of Dijkstra’s algorithm

“a” ←
EXTRACT-MIN(Q):

<table>
<thead>
<tr>
<th>Q</th>
<th>d</th>
<th>π</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>NIL</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>s</td>
</tr>
<tr>
<td>c</td>
<td>14</td>
<td>b d</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td>b</td>
</tr>
</tbody>
</table>

S = {s}
Example of Dijkstra’s algorithm

Relax all edges leaving $a$

<table>
<thead>
<tr>
<th></th>
<th>$d$</th>
<th>$\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>0</td>
<td>NIL</td>
</tr>
<tr>
<td>$a$</td>
<td>10 8</td>
<td>$s$, $b$</td>
</tr>
<tr>
<td>$b$</td>
<td>5</td>
<td>$s$</td>
</tr>
<tr>
<td>$c$</td>
<td>14 13 9</td>
<td>$b$, $d$, $a$</td>
</tr>
<tr>
<td>$d$</td>
<td>7</td>
<td>$b$</td>
</tr>
</tbody>
</table>

$S = \{s, b, d, a\}$
Example of Dijkstra’s algorithm

“c” ← EXTRACT-MIN(Q):

<table>
<thead>
<tr>
<th>Q</th>
<th>d</th>
<th>π</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>NIL</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
<td>s</td>
</tr>
<tr>
<td>c</td>
<td>14</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td>b</td>
</tr>
</tbody>
</table>

S = \{s, b, d, a\}
Example of Dijkstra’s algorithm

Relax all edges leaving \( c \)

\[
\begin{array}{c|c|c}
Q & d & \pi \\
\hline
s & 0 & NIL \\
a & 10 & 8 \\
b & 5 & s \\
c & 14 & 13 \quad 9 \\
d & 7 & b \\
\end{array}
\]

\[ S = \{s, b, d, a, c\} \]
**Lemma:**
Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all $v \in V \setminus \{s\}$ establishes $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

**Proof:**
Suppose not. Let $v$ be the first vertex for which $d[v] < \delta(s, v)$, and let $u$ be the vertex that caused $d[v]$ to change: $d[v] = d[u] + w(u, v)$. Then,
\[
\begin{align*}
    d[v] &< \delta(s, v) \\
    &\leq \delta(s, u) + \delta(u, v) \quad \text{supposition} \\
    &\leq \delta(s, u) + w(u, v) \quad \text{triangle inequality} \\
    &\leq d[u] + w(u, v) \quad \text{shortest path} \leq \text{specific path} \\
    &\leq d[u] + w(u, v) \quad v \text{ is first violation}
\end{align*}
\]
Contradiction.
**Lemma:**
Let $u$ be $v$’s predecessor on a shortest path from $s$ to $v$. Then, if $d[u] = \delta(s, u)$ and edge $(u, v)$ is relaxed, we have $d[v] = \delta(s, v)$ after the relaxation.

**Proof:**
Observe that $\delta(s, v) = \delta(s, u) + w(u, v)$. Suppose that $d[v] > \delta(s, v)$ before the relaxation. (Otherwise, we’re done.) Then, the test $d[v] > d[u] + w(u, v)$ succeeds, because

$$d[v] > \delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v),$$

and the algorithm sets $d[v] = d[u] + w(u, v) = \delta(s, v)$. 
**Theorem:** Dijkstra’s algorithm terminates with $d[v] = \delta(s, v)$ for all $v \in V$.

**Proof:** It suffices to show that $d[v] = \delta(s, v)$ for every $v \in V$ when $v$ is added to $S$. Suppose $u$ is the first vertex added to $S$ for which $d[u] > \delta(s, u)$. Let $y$ be the first vertex in $V \setminus S$ along a shortest path from $s$ to $u$, and let $x$ be its predecessor:

Since $u$ is the first vertex violating the claimed invariant, we have $d[x] = \delta(s, x)$. When $x$ was added to $S$, the edge $(x, y)$ was relaxed, which implies that $d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$. But, $d[u] \leq d[y]$ by our choice of $u$. Contradiction.
Dijkstra’s algorithm: complexity

\[ \text{DIJKSTRA}(G, w, s) \]

d\[s\] \leftarrow 0
for each \( v \in V \setminus \{s\} \) do \( d[v] \leftarrow \infty \)
\( \pi[v] = \text{NIL} \)
\( S \leftarrow \emptyset \)
\( Q \leftarrow V \quad \triangleright Q \text{ is a priority queue maintaining } V \setminus \{s\} \)

while \( Q \neq \emptyset \)
do \( u \leftarrow \text{EXTRACT-MIN}(Q) \)
\( S \leftarrow S \cup \{u\} \)
for each \( v \in \text{Adj}[u] \)
do if \( d[v] > d[u] + w(u, v) \)
then \( d[v] \leftarrow d[u] + w(u, v) \quad \triangleright \text{DECREASE-KEY} \)
\( \pi[v] \leftarrow u \)

While loop: \( O(V) \)
For loop: \( \sum_v \text{deg}(v) = O(E) \)
Total: \( O(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{EXTRACT-MIN}}) \)
Dijkstra’s algorithm: complexity Contd...

Total: \( O(V \cdot T_{\text{EXTRACT-MIN}} + E \cdot T_{\text{EXTRACT-MIN}}) \)

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( T_{\text{EXTRACT-MIN}} )</th>
<th>( T_{\text{DECREASE-KEY}} )</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>( O(V) )</td>
<td>( O(1) )</td>
<td>( O(V^2) )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( O(\log V) )</td>
<td>( O(\log V) )</td>
<td>( O(E \log V) )</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>( O(\log V) )</td>
<td>( O(1) )</td>
<td>( O(E + V \log V) )</td>
</tr>
<tr>
<td></td>
<td>amortized</td>
<td>amortized</td>
<td>worst case</td>
</tr>
</tbody>
</table>