CSE 317: Design and Analysis of Algorithms
Greedy Algorithms

Shahid Hussain

Fall 2025

SMCS, IBA

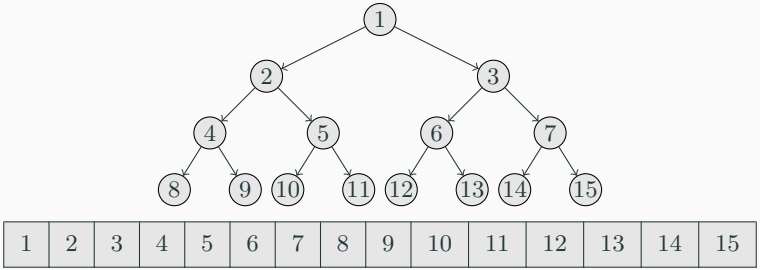
Greedy Algorithms

- Greedy algorithms are algorithms that use the following meta-heuristic: to make the locally optimal choice at each stage with the hope of finding the global optimum
- Often greedy algorithms produce sub-optimal result however the algorithms considered in this section are all optimal
- Running time for algorithms usually depend on the choice of underlying data structure and for two of the greedy algorithms we will discuss choosing a right data structure plays an important role
- Appropriate data structures play important role in efficient greedy algorithms

Binary Heaps

Min Binary Heap

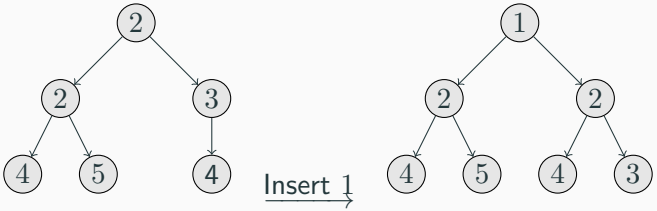
A *min binary heap* is a linear data structure (uses an array) that stores elements from a complete binary tree where key (data value) in each node of the binary tree is less than or equal to the key values of its children.



- We define three operations on heaps: *Insert-key*, *Delete-min*, and *Build-heap*.

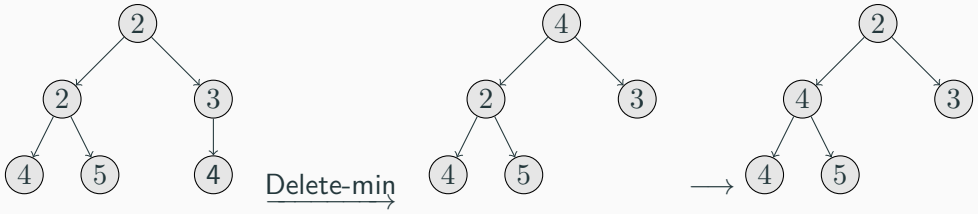
Insert-Key Operation

- To *insert* a new element, we place it at the bottom of the tree (in the first available position)
- If the key of new element is less than the key of its parent, then we should swap these elements, etc.
- The number of swaps is at most the depth of the tree, which is $\lfloor \log_2 n \rfloor$ where n is the number of elements



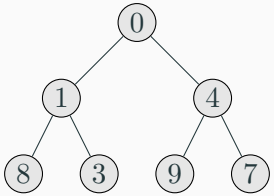
Delete-Min Operation

- To *delete the min element* (which is always the root of the heap) we return the element attached to the root
- Then we remove this element from the heap, take the element attached to the last node in the tree (in the rightmost position in the bottom level) and place it at the root
- If the key of this element is bigger than any of the root's child nodes, we swap these elements, etc. The number of swaps is at most $\lfloor \log_2 n \rfloor$



Build-Heap Operation

- To *build* a heap with n elements we can insert n elements in an empty heap. Each operation taking $O(\log n)$ operations so building a heap takes $O(n \log n)$ operations
- For example, creating a heap for $8, 0, 9, 1, 3, 7, 4$ we will have following:



Minimum Spanning Trees

For a given simple undirected graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}^{\geq 0}$ an *spanning tree* of G is a *tree* $T = (V, E')$ with V as the vertices and E' is a subset of E with cost as the sum: $\sum_{e \in E'} w(e)$.

A spanning tree is called a *minimum spanning tree* if the sum of weights of edges is minimum.

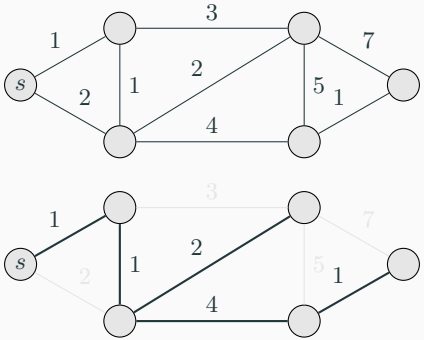
MST: Prim's Algorithm

- Prim's algorithm computes the minimum spanning tree (optimal solution) of a given simple undirected graph with nonnegative weights/costs.
- It works as following:
- We start with a source node s (without the loss of generality)
- We maintain a set $S \subseteq V$ on which a spanning tree is constructed
- Initially, $S = \{s\}$
- During each iteration, we add to S one node $v \in V \setminus S$ which minimizes the value

$$a(v) = \min_{\substack{e=(u,v)\in E \\ u\in S}} w(e),$$

  and include the edge $e = (u, v)$ that achieves this minimum in the constructed spanning tree T
- Algorithms stops when $S = V$.
- The constructed graph is a connected and has $|V| - 1$ edges

MST: Example



Correctness of Prim's Algorithm

- The correctness proof is by induction on $|S|$ and partial spanning tree: $T_{|S|}$
- **Basis step:** If $|S| = 1$, the statement holds and T_1 is empty
- **Induction hypothesis:** Statement hold for some $k \geq 1$
- **Induction step:** In step $(k+1)$ an edge $e = (u, v)$ is chosen s.t., $u \in S$, $v \in V \setminus S$ and $w(e)$ is minimum
- If $e \in T$, the statement holds
- If $e \notin T$, we can add e to T. It creates a cycle
- This cycle must contain another edge e' which joins S and $V \setminus S$
- If we remove e', we obtain a connected graph with $|V|$ nodes and $|V| - 1$ edges so $T' = (T \setminus \{e'\}) \cup \{e\}$ is a spanning tree
- According to the choice of e, the total cost of T' is at most the total cost of T.
- Thus, T' is a minimum spanning tree
- It is clear, that $T_k \subseteq T'$ and $e \in T'$. Therefore the considered statement holds

Analysis of the Algorithm

- For the graph $G = (V, E)$, let $|V| = n$ and $|E| = m$
- The Prim's algorithm keeps the vertices in a priority queues with $a(v)$ as the keys
- We build the binary heap with a *make-queue* operation
- We select a vertex with a *delete-min* operation and update the attachment cost using *decrease-key* operations
- We perform one *make-queue* operation
- There are $n - 1$ iterations and we perform *delete-min* operations at most once for each edge
- Therefore the total cost is $O((n + m) \log n)$

Single Source Shortest Path Problem

Single Source Shortest Path Problem (SSSPP)

For a given directed graph $G = (V, E)$ with the length function $l : E \to \mathbb{R}^{\geq 0}$ and a source vertex $s \in V$. The *single source shortest path problem* asks to compute the paths between s and every vertex $v \in S$ such that sum of lengths (weights) along the paths is minimum.

[We assume there exists a path between the source vertex and every other vertex of the graph.]

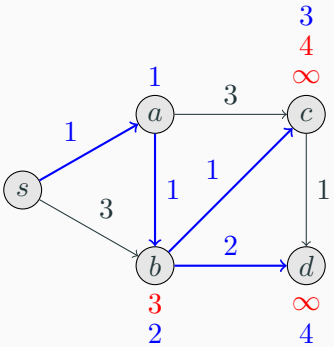
SSSPP: Dijkstra's Algorithm

- Dijkstra's algorithm computes the *shortest paths* for a source vertex s to every other vertex v of the input graph
- It constructs a set S and a value $d(v)$ for each vertex $v \in V$
- Initially, $S = \{s\}$ and $d(s) = 0$.
- Now, for each vertex $v \in V \setminus S$, it calculates $d'(v)$ (length of a potential shortest path from s to v, local optimal), we say:

$$d'(v) = \min_{\substack{e=(u,v)\in E \\ u\in S}} \{d(u) + l(e)\}$$

- If no such e exists then $d'(v) = +\infty$
- The algorithm now chooses $v \in V \setminus S$ for which $d'(v) = \min\{d'(v') : v' \in V \setminus S\}$
- It sets $d(v) = d'(v)$, $predecessor(v) = u$, and pushes v to S
- Algorithm stops when $S = V$

Example



Proof of Correctness

- During the each step, the algorithm adds new vertex v to the set S, computes the value $d(v)$ for this vertex and forms the path P_v from s to v with the help of values $predecessor(\cdot)$
- Let us show by induction on $|S|$ that P_v is the shortest path from s to v and that $d(v)$ is the minimum length of a path from s to v

Proof of Correctness

- **Basis step:** When $|S| = 1$, $S = \{s\}$ and $d(s) = 0$, the statement holds
- **Induction hypothesis:** Assume the statement holds for $|S| = k$, $k \geq 1$
- **Induction step:** During the step no. $k + 1$, algo. chooses a vertex v and an edge $e = (u, v)$ in the path P_v
- Let us consider an arbitrary path P from s to v
- Since $v \notin S$, there is an edge $e' = (x, y)$ s.t. $x \in S$ and $y \notin S$
- By induction hypothesis length of the any path to x is at least $d(x)$,

$$l(P) \geq d(x) + l(e') \geq d(u) + l(e) = l(P_v)$$

- So, $d(x) + l(e') \geq d(u) + l(e)$ therefore $l(P) \geq l(P_v)$ and P_v is the shortest path from s to v
- This completes the proof

Analysis of the Algorithm

- The straight-forward implementation of Dijkstra's algorithm requires $O(|V|^2)$ time
- We can improve it using binary heaps
- We store the values $d'(v)$ for vertices in $V \in V \setminus S$ in a binary heap
- Initially we have $d'(s) = 0$ and $d'(v) = +\infty$ for each vertex $v \in V \setminus S$
- We use *make-queue* operation to build the heap for vertices in V
- We will modify the heap step-by-step to have current set of vertices $V \setminus S$ and current values $d'(v)$
- After k steps we have some vertices from $V \setminus S$ and their key values
- During the step $k + 1$ we choose the vertex v with the minimum value $d'(v)$
- We set $d(v) = d'(v)$
- Now we modify values of $d'(w)$ for each vertex $w \in V \setminus (S \cup \{v\})$ as:
- If (v, w) is not an edge then we keep $d'(w)$ unchanged
- Else, $e' = (v, w)$ be an edge then the new value of the key for w is $\min\{d'(w) : d(v) + l(e')\}$
- If $d'(w) > d(v) + l(e')$ then we set $predecessor(w) = v$

Analysis of the Algorithm

- For the graph $G = (V, E)$, let $|V| = n$ and $|E| = m$
- The algorithm makes one operation of *make-queue* in time $O(n \log n)$
- At most n operations of *delete-min* in time $O(n \log n)$, and
- At most m operations of *decrease-key* in time $O(m \log n)$
- Thus, the overall time for the implementation is

$$O((m + n) \log n)$$