

CSE 317: Design and Analysis of Algorithms

Shahid Hussain

Week 10: October 21 – October 26, 2024: Fall 2024

Greedy Algorithms

Greedy Algorithms

- Greedy algorithms 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 here 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
- We will discuss *binary heaps* for implementation of two such algorithms: *Prim's algorithm* and *Dijkstra's algorithm*.

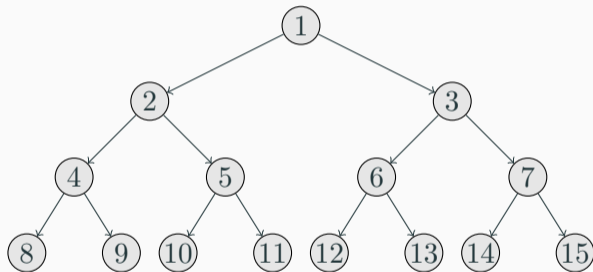
Binary Heaps

- In a *binary heap* elements are stored in a complete binary tree, namely, a binary tree in which each level is filled from the left to the right, and must be full before the next level is started
- In addition, the key value of any node of the tree is less than or equal to key values of its children. In particular, the root of the tree always contains an element with the smallest key

Binary Heaps (cont.)

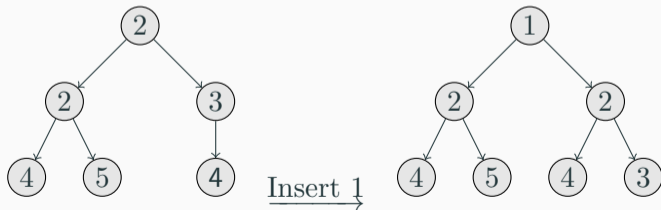
- The binary tree (for binary heaps) can be easily represented using an array
- The nodes of tree have a natural ordering: row by row, starting at the root and moving from the left to the right within each level. The array has positions $1, \dots, n$ corresponding to the considered nodes
- One can show that the node at location j in the array has the parent at location $\lfloor j/2 \rfloor$ has two of its child nodes at locations $2j$ and $2j + 1$

Binary Heaps (cont.)



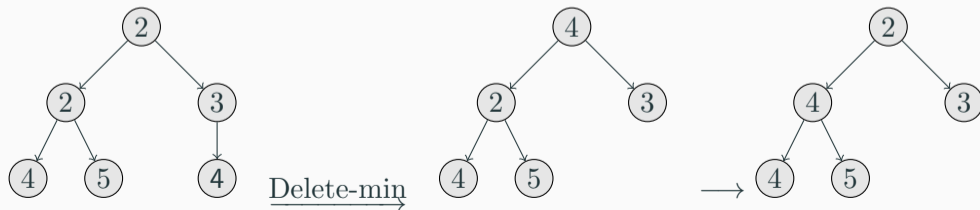
Binary Heaps (cont.): Insert Key

- 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



Binary Heaps (cont.) Delete-Min

- 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 $\lfloor \log_2 n \rfloor$



Binary Heaps (cont.) Build-Heap

- 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
- We can use binary heaps to come up with a sorting algorithm called *heap sort*
- The idea is to build a heap with the elements to be sorted, then delete the min element n times
- Each delete-min operation takes $O(\log n)$ operations, so the total running time is $O(n \log n)$

Single Source Shortest Paths: Dijkstra's Algorithm

Single Source Shortest Paths: Dijkstra's Algorithm

- Let $G = (V, E)$ be a directed graph s.t. each edge $e \in E$ has a length $l_e \geq 0$
- For a path P , the length of P ($l(P)$) is the sum of lengths of all edges in P
- Let s be a node of G (source node)
- We assume that s has a path to every other node in G
- The algorithm constructs a set S of nodes u for which we have determined the length of the shortest path $d(u)$ from s
- Initially, $S = \{s\}$ and $d(s) = 0$
- Now for each node $v \in V \setminus S$, we determine the length $d'(v)$ of a shortest path of the following kind: the considered path passes through nodes of S until some node $u \in S$ and then passes through an edge $(u, v) \in E$
- It is clear that

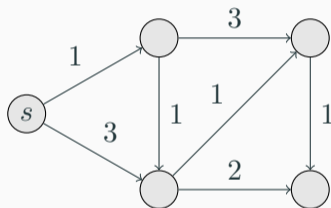
$$d'(v) = \min_{e=(u,v) \in E, u \in S} d(u) + l_e.$$

- If there is no $(u, v) \in E$ such that $u \in S$, then $d'(v) = +\infty$

Single Source Shortest Paths: Dijkstra's Algorithm (cont.)

- We choose $v \in V \setminus S$ for which $d'(v) = \min\{d'(v') : v' \in V \setminus S\}$ and add v to S . We set $d(v) = d'(v)$ and $predecessor(v) = u$, where $u \in S$ and there exists an edge $e = (u, v) \in E$ such that $d'(v) = d(u) + l_e$
- The value $predecessor(v)$ for each $v \in V$, $v \neq s$, will allow us to restore the shortest paths from s to all other nodes in G
- The algorithm stops when $S = V$

Single Source Shortest Paths: Dijkstra's Algorithm (cont.)



Single Source Shortest Paths: Dijkstra's Algorithm (cont.)

- During each step the algorithm adds new node v to the set S , computes the value $d(v)$ for this node 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
- From here it follows that $d(v)$ is the minimum length of a path from s to v
- Let $|S| = 1$, in this case $S = \{s\}$ and $d(s) = 0$, the statement holds in this case
- Assume that the statement holds when $|S| = k$ for some value of $k \geq 1$

Single Source Shortest Paths: Dijkstra's Algorithm (cont.)

- So during the step number $k + 1$ the algorithm chooses the node v , and $e = (u, v)$ be the final edge in the path P_v
- Let us consider an arbitrary path P from s to v
- Since $v \notin S$, there is an edge $e' = (x, y)$ in this path such that $x \in S$ and $y \notin S$
- By the inductive hypothesis, the length of any path from s to x is at least $d(x)$
- Therefore $l(P) \geq d(x) + l_{e'} \geq d(u) + l_e = l(P_v)$. The inequality $d(x) + l_{e'} \geq d(u) + l_e$ follows from the description of the algorithm
- Therefore $l(P) \geq l(P_v)$, and P_v is the shortest path from s to v

Single Source Shortest Paths: Dijkstra's Algorithm (cont.)

- To optimize, we use binary heap to store values $d'(v)$ for nodes $v \in V \setminus S$
- Initially, we have $d'(s) = 0$ and $d'(v) = +\infty$ for each $v \in V \setminus \{s\}$
- We use *make-queue* operation to build a binary heap with values $d'(v)$ as keys
- We modify the heap to have current set of nodes $V \setminus S$ and current values $d'(v)$
- After the k -th step the heap has all node v from $V \setminus S$ and their current values
- During the step number $k + 1$, by operation *delete-min*, we choose v with the minimum value $d'(v)$ and delete v from the heap, and set $d(v) = d'(v)$
- Now we modify the values of $d'(w)$ for each node $w \in V \setminus (S \cup \{v\})$
- If (v, w) is not an edge then we keep $d'(w)$ unchanged
- Let $e' = (v, w)$ 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 it is necessary to set $predecessor(w) = v$ and to use the *decrease-key* operation to decrease the key of node w up to $d(v) + l_{e'}$

Single Source Shortest Paths: Dijkstra's Algorithm (cont.)

- To decrease the key for w we should find w in the heap (in the corresponding array)
- To this end, we will assume that all n nodes in G are numbered by numbers $1, \dots, n$ and we have an array in which in the t -th position we have the current value of the position of t -th node in the heap
- We can store in this array values $d(u)$ and $predecessor(u)$
- The *decrease-key* operation can be used at most once per edge, when the initial node of the edge is added to S
- 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 $O((m + n) \log n)$

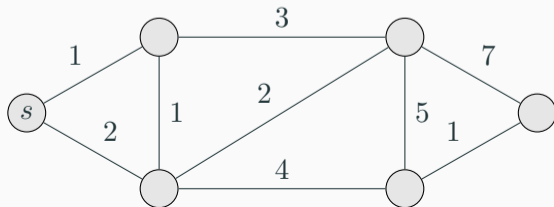
Minimum Spanning Trees

Minimum Spanning Trees: Prim's Algorithm

- Let $G = (V, E)$ be a connected undirected graph such that each edge e in E is labeled with a cost $c_e \geq 0$
- We call a subset $T \subseteq E$ a *spanning tree* of G if (V, T) is a tree
- A spanning tree for which the total cost $\sum_{e \in T} c_e$ is as small as possible is called *minimum spanning tree*
- We need to construct a minimum spanning tree for G
- We use Prim's algorithm to construct such an MST

Minimum Spanning Trees: Prim's Algorithm

- We start with a root node s , 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 $a(v) = \min_{e=(u,v) \in E, u \in S} c_e$ and include the edge $e = (u, v)$ that achieves this minimum in the constructed spanning tree T . Algorithm stops when $S = V$
- It is clear that the constructed graph is connected and has $|V| - 1$ edges. So this graph is a spanning tree.



Minimum Spanning Trees: Prim's Algorithm

- Let us prove by induction on $|S|$ that partial spanning tree constructed during the step number $|S|$ is a part of minimum spanning tree
- If $|S| = 1$ then the partial spanning tree is empty, and the considered statement holds. Let for some $k \geq 1$ this statement hold: $|S| = k$ and the constructed partial spanning tree T_k is a part of some minimum spanning tree T

Minimum Spanning Trees: Prim's Algorithm

- During the step number $k + 1$ we add to S an edge $e = (u, v)$ such that $u \in S, v \in V \setminus S$ and e has the minimum cost c_e among all edges that join S and $V \setminus S$
- If $e \in T$ then the statement holds.
- Now suppose $e \notin T$. In this case we add e to T , as a result we obtain a graph K with a cycle
- This cycle must contain another edge e' which joins S and $V \setminus S$

Minimum Spanning Trees: Prim's Algorithm

- If we remove e' from K , we obtain a connected graph with $|V|$ nodes and $|V| - 1$ edges (since T is a spanning tree, $|T| = |V| - 1$)
- Therefore, we obtain a tree. 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. From this statement it follows that Prim's algorithm constructs a minimum spanning tree

Minimum Spanning Trees: Prim's Algorithm

- e can implement Prim's algorithm almost in the same way as Dijkstra's algorithm
- By analogy with Dijkstra's algorithm we should be able to decide which node $v \in V \setminus S$ will be added to S . To this end we will consider the attachment cost $a(v) = \min_{e=(u,v) \in E, u \in S} c_e$ for each node $v \in V \setminus S$.
- As before, we keep the nodes in a priority queue with $a(v)$ as the key
- We build a binary heap with a *make-queue* operation
- We select a node with a *delete-min* operation, and update the attachment cost using *decrease-key* operation

Minimum Spanning Trees: Prim's Algorithm

- We perform *make-queue* operation one time
- There are $n - 1$ iterations in which we perform *delete-min* operation, and we perform *decrease-key* operation at most once for each edge
- So the overall running time is $O((n + m) \log n)$ where $n = |V|$ and $m = |E|$
- At the beginning of the algorithm we have $S = \{s\}$. We form values of $a(v)$ for $v \in V \setminus \{s\}$ in the following way
- If $(s, v) \in E$ then $a(v) = c(s, v)$. Otherwise, $a(v) = +\infty$
- Let during some step we add to S a new node v . Then for each node $w \in V \setminus (S \cup \{v\})$ for which $(v, w) \in E$, instead of old value $a(w)$ we should use the new value which is equal to $\min\{a(w), c(v, w)\}$