CS350 ABET Objective 10

Part 3 – Greedy Algorithms

Program greedy algorithms.

Textbook Section 9.1-9.3 and 6.4, ~10%.

A greedy algorithm constructs a solution to an optimization problem through a sequence of steps. Each step extends a partial solution with a feasible, locally optimal, and irrevocable choice until a complete solution is computed.

Prim 1

Minimum spanning tree of a graph, stepwise. Start MST with any node; a step attaches the closest node not yet in the MST.

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Performance depends on efficiency of loop body which depends on data structures used.

Prim 2

Time efficiency of naive approach is $O(|V||E|)$, since $|V|$ times it accesses one among $|E|$ edges.

$|E|$ can range from $\approx|V|$ to $\approx|V|^2/2$

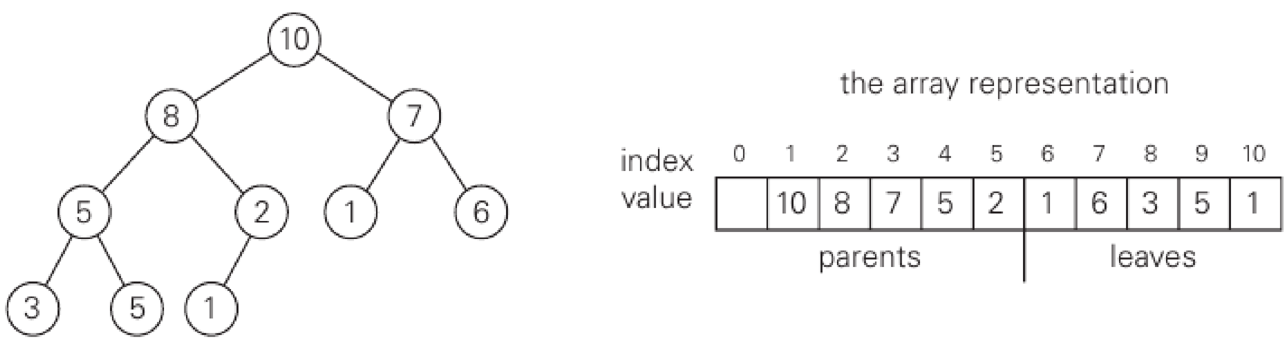
Performance $O(|E|\log|V|)$ is achieved with a binary heap.

Heap

Binary tree with two properties:

1. *complete*: full levels left-to-right except last
2. *dominance*: each node greater than children

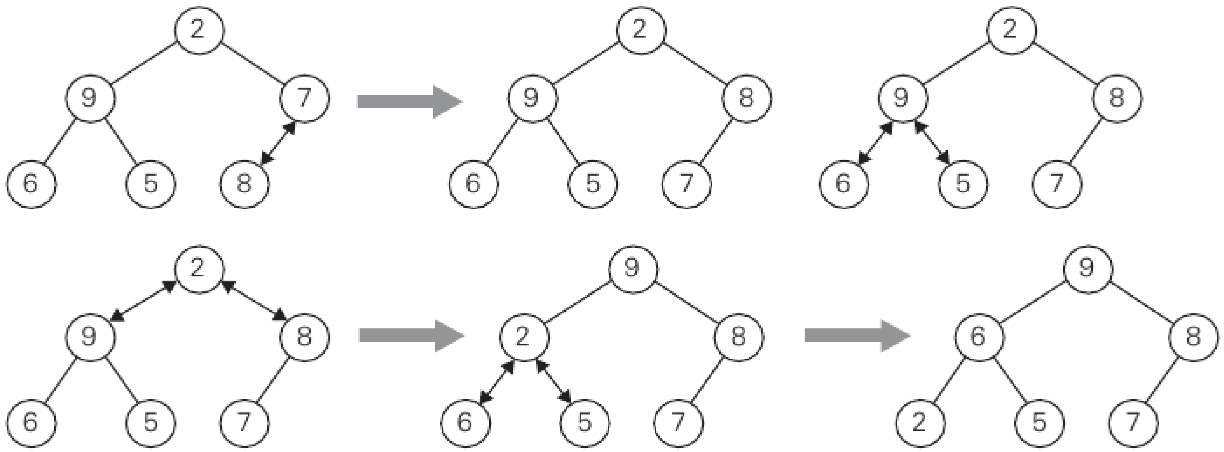
Represented as array:



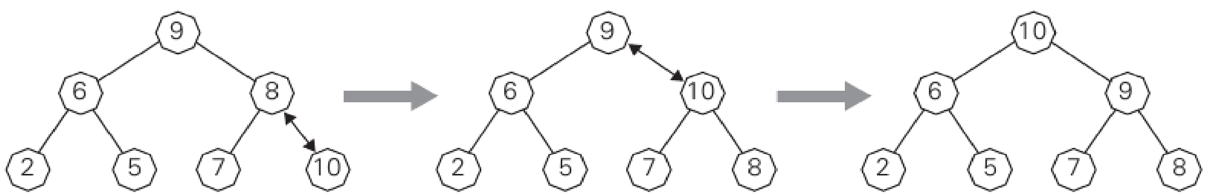
Efficient operations

1. finding an item with the highest priority
2. deleting an item with the highest priority
3. adding a new item to the multiset

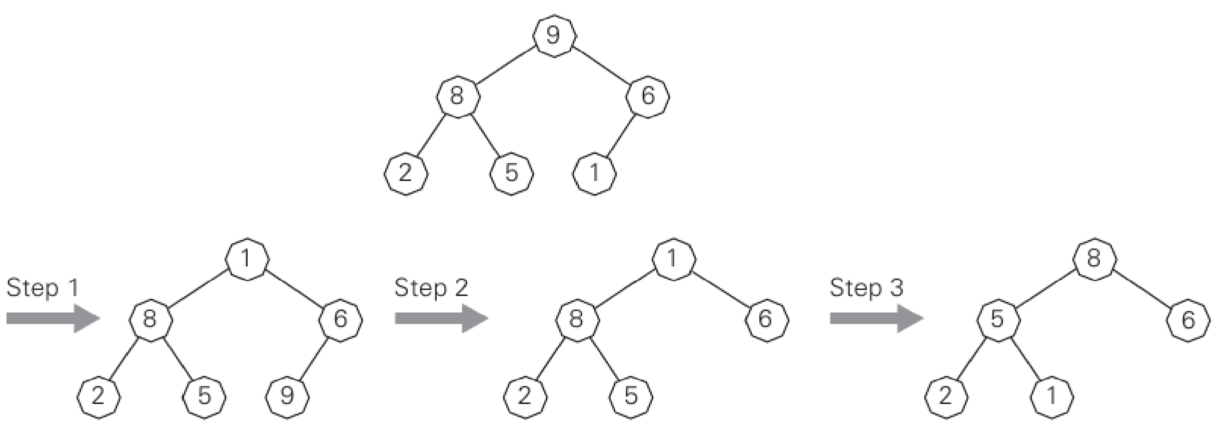
Construct heap bottom up, example:



Insert item in heap:



Delete item from heap, root because of priority:



Kruskal

Minimum spanning tree of a graph, stepwise. (1) sort all edges; (2) a step adds the next smallest edge that does not create a cycle.

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $\quad ecounter \leftarrow 0$ $\qquad$ //initialize the set of tree edges and its size
$k \leftarrow 0$ $\qquad$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
$\quad k \leftarrow k + 1$
$\quad$ **if** $E_T \cup \{e_{i_k}\}$ is acyclic
$\qquad E_T \leftarrow E_T \cup \{e_{i_k}\}$; $\quad ecounter \leftarrow ecounter + 1$
return E_T

Time efficiency dominated by sorting edges:
$O(|E|\log|E|) = O(|E|\log|V|)$

Note $O(\log|E|) = O(\log|V|)$ since $|E| \leqslant |V|^2$ and $O(\log|V|^2) = O(\log|V|)$.

Dijkstra 1

Single-source shortest-path in a graph: For a given source node in the graph, the algorithm finds the path with lowest cost between that vertex and every other vertex.

Algorithm is stepwise. (1) start with source as current, rest as unvisited nodes with infinite distance; (2) at each step, current is closest unvisited node to source; update distance of each unvisited node adjacent to current; mark current as visited.

Use a priority queue for unvisited nodes. Efficiency is $O(|E| \log |V|)$.

Dijkstra 2

A graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree.

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

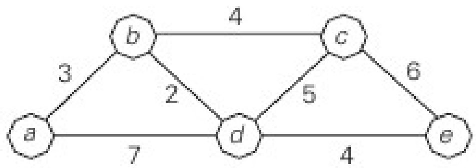
1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes except the initial node.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. For example, if the current node A is marked with a

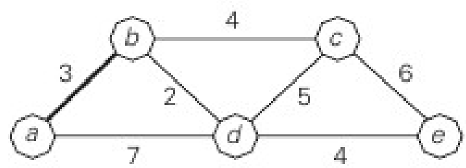
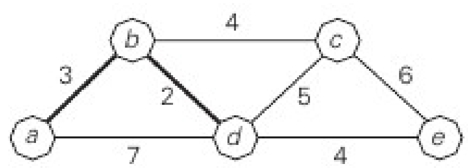
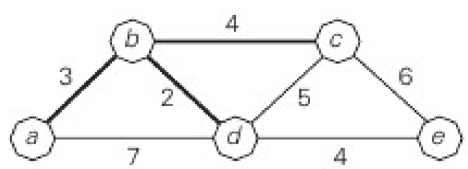
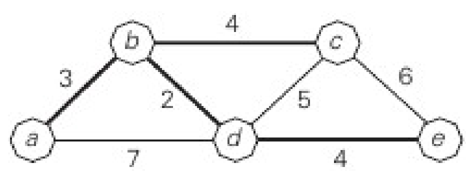
tentative distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6+2=8. If this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the *unvisited set*.

4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again; its distance recorded now is final and minimal.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal), then stop. The algorithm has finished.
6. Set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3.

Dijkstra 3

Execution:



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

References

Textbook Section 9.1-9.3 (for greedy)
Textbook Section 6.4 (for heap)

Web:

http://en.wikipedia.org/wiki/Prim%27s_algorithm

http://en.wikipedia.org/wiki/Kruskal%27s_algorithm

http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm