Depth First Search (DFS)

Depth first search (DFS) is useful for

- Find a path from one vertex to another
- Whether or not graph is connected
- Computing a spanning tree of a connected graph.

DFS uses the backtracking technique.

Algorithm Depth First Search

Algorithm starts at a specific vertex S in G, which becomes current vertex. Then algorithm traverse graph by any edge (u, v) incident to the current vertex u. If the edge (u, v) leads to an already visited vertex v, then we backtrack to current vertex u. If, on other hand, edge (u, v) leads to an unvisited vertex v, then we go to v and v becomes our current vertex. We proceed in this manner until we reach to "deadend". At this point we start back tracking. The process terminates when backtracking leads back to the start vertex.

Edges leads to new vertex are called discovery or tree edges and edges lead to already visited are called back edges.

DEPTH FIRST SEARCH (G, *v*)

Input: A graph G and a vertex v.
Output: Edges labeled as discovery and back edges in the connected component.

For all edges e incident on v do
  If edge e is unexplored then
    w ← opposite (v, e) // return the end point of e distant to v
    If vertex w is unexplained then
      - mark e as a discovery edge
      - Recursively call DSF (G, w)
    else
      - mark e as a back edge

DFS algorithm used to solve following problems.

Testing whether graph is connected.

Computing a spanning forest of graph.

Computing a path between two vertices of graph or equivalently reporting that no such path exists.

Computing a cycle in graph or equivalently reporting that no such cycle exists.

Analysis

The running time of DSF is $\Theta(V + E)$.

Breadth First Search (BFS)

Breadth First Search algorithm used in

- Prim's MST algorithm.
- Dijkstra's single source shortest path algorithm.

Like depth first search, BFS traverse a connected component of a given graph and defines a spanning tree.

Algorithm Breadth First Search

BFS starts at a given vertex, which is at level 0. In the first stage, we visit all vertices at level 1. In the second stage, we visit all vertices at second level. These new vertices, which are adjacent to level 1 vertices, and so on. The BFS traversal terminates when every vertex has been visited.

BREADTH FIRST SEARCH (G, S)

Input: A graph G and a vertex.
Output: Edges labeled as discovery and cross edges in the connected component.

Create a Queue Q.
ENQUEUE (Q, S) // Insert S into Q.
While Q is not empty do
  for each vertex *v* in Q do

```
for all edges e incident on v do
    if edge e is unexplored then
        let w be the other endpoint of e.
        if vertex w is unexpected then
            - mark e as a discovery edge
            - insert w into Q
        else
            mark e as a cross edge
```

BFS label each vertex by the length of a shortest path (in terms of number of edges) from the start vertex.

Generic Minimum Spanning Tree

Informally, the MST problem is to find a free tree T of a given graph G that contains all the vertices of G and has the minimum total weight of the edges of G over all such trees.

Problem Formulation

Let G = (V, E, W) be a weighted connected undirected graph. Find a tree T that contains all the vertices in G and minimize the sum of the weights of the edges (u, *v*) of T, that is

$$w(T) = \sum_{(u, v) \in T} w(u, v)$$

Tree that contains every vertex of a connected graph is called spanning tree.

The problem of constructing a minimum spanning tree of MST is computing a spanning tree T with smallest total weight.

Kruskal's Algorithm

Kruskal's Algorithm for computing the minimum spanning tree is directly based on the generic MST algorithm. It builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm consider each edge in turn, order by increasing weight. If an edge (*u*, *v*) connects two different trees, then (*u*, *v*) is added to the set of edges of the MST, and two trees connected by an edge (*u*, *v*) are merged into a single tree on the other hand, if an edge (*u*, *v*) connects two vertices in the same tree, then edge (*u*, *v*) is discarded.

Outline of Kruskal's Algorithm

Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in graph.

I Kruskal's algorithm implemented with disjoint-sets data structure.

Disjoint-Sets Data Structure

- Make_Set (v)
  Create a new set whose only member is pointed to by v. Note that for this operation v must already be in a set.
- FIND_Set
  Returns a pointer to the set containing v.
- UNION (u, v)
  Unites the dynamic sets that contain u and v into a new set that is union of these two sets.

MST_KRUSKAL (G, w)

1. A ← { } // A will ultimately contains the edges of the MST
2. for each vertex v in V[G]
3. do Make_Set (v)
4. Sort edge of E by nondecreasing weights w
5. for each edge (u, v) in E
6. do if FIND_SET (u) ≠ FIND_SET (v)
7. then A = AU{(u, v)}
8. UNION (u, v)
9. Return A

Analysis

The for-loop in lines 5-8 per forms 'Union' and 'find' operations in $O(|E|)$ time. When the disjoint-set forest implementation with the weighted union and path compression heuristic is used, the $O(|E|)$ 'union' and 'find' operators involved in the for-loop in lines 5-8 will have worst-case time $O(|E| \lg * |E|)$. Kruskal's algorithm requires sorting the edge of E in line 4. We know that the fastest comparison-based sorting algorithm will need $\Theta(|E| \lg |E|)$ time to sort all the edges. The time for sorting all the edges of G in line 4 will dominates the time for the 'union' and 'find' operations in the for-loop in lines 5-8. When comparison-based sorting algorithm is used to perform sorting in line 4. In this case, the running time of Kruskal's algorithm will be $O(|E| \lg |E|)$.

In summary, Kruskal's algorithm as given in the CLR, pp 505 requires $O(E \lg E)$ time

- $O(V)$ time to initialize.
- $O(E \lg E)$ time to sort edges by weight
- $O(E \alpha (E, V)) = O(E \lg E)$ time to process edges.

If all edge weights are integers ranging from 1 to |V|, we can use COUNTING_SORT (instead of a more generally applicable sorting algorithm) to sort the edges in $O(V + E) = O(E)$ time. Note that $V = O(E)$ for connected graph. This speeds up the whole algorithm to take only $O(E \alpha (E, V))$ time; the time to process the edges, not the time to sort them, is now the dominant term. Knowledge about the weights won't help speed up any other part of the sort in line 4 uses the weight values.

If the edges weights are integers ranging from 1 to constant W, we can again use COUNTING_SORT, which again runs in $O(W + E) = O(E)$ time. Note taht $O(W + E) = O(E)$ because W is a constant. Therefore, the asymptotic bound of the Kruskal's algorithm is also $O(E \alpha (E, V))$.

II Kruskal's algorithm implemented with priority queue data structure.

MST_KRUSKAL (G)

1. for each vertex *v* in V[G] do
2. define set S(*v*) ← {*v*}
3. Initialize priority queue Q that contains all edges of G, using the weights as keys.
4. A ← { } // A will ultimately contains the edges of the MST
5. While A has less than n-1 edges do
6. Let set S(*v*) contains *v* and S(*u*) contain *u*
7. IF S(*v*) != S(*u*) then
   Add edge (*u*, *v*) to A
   Merge S(*v*) and S(*u*) into
   one set i.e., union
8. Return A

Analysis

- The edge weight can be compared in constant time.
- Initialization of priority queue takes O(E log E) time by repeated insertion.
- At each iteration of while-loop, minimum edge can be removed in O(log E) time, which is O(log V), since graph is simple.
- The total running time is O((V + E) log V), which is O(E lg V) since graph is simple and connected.

Prim's Algorithm

Like Kruskal's algorithm, Prim's algorithm is based on a generic MST algorithm. The main idea of Prim's algorithm is similar to that of Dijkstra's algorithm for finding shortest path in a given graph. Prim's algorithm has the property that the edges in the set A always form a single tree. We begin with some vertex *v* in a given graph G =(V, E), defining the initial set of vertices A. Then, in each iteration, we choose a minimum-weight edge (*u*, *v*), connecting a vertex v in the set A to the vertex *u* outside of set A. Then vertex *u* is brought in to A. This process is repeated until a spanning tree is formed. Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A. The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edges in set A form a MST.

Outline of Prim's Algorithm

Choose a node and build a tree from there selecting at every stage the shortest available edge that can extend the tree to an additional node.

Algorithm

MST_PRIM (G, *w*, *v*)

1. Q ← V[G]
2. for each *u* in Q do
3. key [*u*] ← ∞
4. key [*r*] ← 0
5. π[*r*] ← NIl
6. while queue is not empty do

```
7.    u ← EXTRACT_MIN (Q)
8.    for each v in Adj[u] do
9.       if v is in Q and w(u, v) < key [v]
10.         then π[v] ← w(u, v)
11.            key [v] ← w(u, v)
```

Analysis

The performance of Prim's algorithm depends of how we choose to implement the priority queue Q.

Definitions Sparse graphs are those for which $|E|$ is much less than $|V|^2$ i.e., $|E| << |V|^2$ we preferred the adjacency-list representation of the graph in this case. On the other hand, dense graphs are those for which $|E|$ is graphs are those for which $|E|$ is close to $|V|^2$. In this case, we like to represent graph with adjacency-matrix representation.

a. When a Q is implemented as a binary heap

We can use the BULID_HEAP procedure to perform the initialization in lines 1-4 in $O(|V|)$ time. The while-loop in lines 6-11 is executed $|V|$ times, and since each EXTRACT_MIN operation takes $O(\lg|V|)$ time, the total time for all calls to EXTRACT_MIN operation is $O(|V| \lg|V|)$. The for-loop in lines 8-11 is executed $O(|E|)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the for-loop, the test for membership in Q in line 9 can be implemented in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q. The assignment in line 11 involves an implicit DECREASE_KEY operation on the heap, which takes $O(\lg|V|)$ time with a binary heap. Thus,

the total time for Prim's algorithm using a binary heap is $O(|V| \lg|V| + |E| \lg |V|)$.

b. When Q is implemented as a Fibonacci heap

In this case, the performance of the DECREASE_KEY operation implicitly called in line 11 will be improved. Instead of taking $O(\lg|V|)$ time, with a Fibonacci heap of $|V|$ elements, the DECREASE_KEY operation will take $O(1)$ amortized time. With a Fibonacci heap, an EXTRACT_MIN operation takes $O(\lg |V|)$ amortized time. Therefore, the total time for Prim's algorithm using a Fibonacci heap to implement the priority queue Q is $O(|V| \lg|V| + |E|)$.

c. When graph G = (V, E) is sparse i.e., $|E| = \Theta(|V|)$

From above description, it can be seen that both versions of Prim's algorithm will have the running time $O(|V| \lg|V|)$. Therefore, the Fibonacci-heap implementation will not make Prim's algorithm asymptotically faster for sparse graph.

d. When graph G is dense, i.e., $|E| = \Theta(|V|)$

From above description, we can see that Prim's algorithm with the binary heap will take $|V|^2 \lg|V|$ time whereas Prim's algorithm with Fibonacci-heap will take $|V|^2$. Therefore, the Fibonacci-heap implementation does not make Prim's algorithm asymptotically faster for dense graphs.

e. Comparing the time complexities of the two versions of Prim's algorithms, mentioned in (a) and (b), we can see that $|E|$ is greater than

|V|, the Prim's algorithm with the Fibonacci-heap implementation will be asymptotically faster than the one with the binary-heap implementation.

The time taken by Prim's algorithm is determined by the speed of the queue operations. With the queue implemented as a Fibonacci heap, it takes $O(E + VlgV)$ time. Since the keys in the priority queue are edge weights, it might be possible to implement the queue even more efficiently when there are restrictions on the possible edge weights. If the edge weights are integers ranging from 1 to some constant w, we can speed up the algorithm by implementing the queue as an array $Q[0 \ldots w+1]$ (using the w+1 slot for key = ∞), where each slot holds a doubly linked list of vertices with that weight as their key. . Then EXTRACT_MIN takes only $O(w) = O(1)$ time (just scan for the first nonempty slot), and DECREASE_KEY takes only $O(1)$ time (just remove the vertex from the list its in and insert it at the front of the list indexed by the new key). This gives a total running time of $O(E)$, which is best possible asymptotic time (since $\Omega(E)$ edges must be processed).

However, if the edge-weights are integers ranging from 1 to |V|, the array implementation does not help. The operation DECREASE_KEY would still be reduced to constant time, but operation EXTRACT_MIN would now use $O(V)$ time, for the total running time of $O(E + V^2)$. Therefore, we are better off sticking with the Fibonacci-heap implementation, which takes $O(E + VlgV)$ time. To get any advantage out of the integer weights, we would have to use data structure we have not studied in the CLR, such as

- The VanEmde Boas data structure mentioned in the introduction to part V: upper bound $O(E + VlglgV)$ for prim's algorithm.
- Redistribution heap (not in the CLR): upper bound $O(E + Vsqrt\{lgV\})$ for prim's algorithm.

Prim's Algorithm with Adjacency Matrix

Now we describe the Prim's algorithm when the graph G =(V, E) is represented as an adjacency matrix.

Instead of heap structure, we use an array to store the key of each node

1. A ← V[g] // Array
2. For each vertex u∈ Q do
3. key [u]∈ ∞
4. key [r] ← 0
5. π[r] ← NIL
6. while Array A empty do
7. scan over A find the node u with smallest key, and remove it from array A
8. for each vertex *v*∈ Adj[u]
9. if v∈ A and w[u, *v*] < key[*v*] then
10. π[*v*] ← u
11. key[*v*] ← w[u, *v*]

Analysis

For-loop (line 8-11) takes $O(\deg[u])$ since line 10 and 11 take constant time. Due to scanning whole array A, line 7 takes $O(V)$ timelines 1-5, clearly $\Theta(V)$.

Therefore, the while-loop (lines 6-11) needs

$$O(\sum_u(V+dg[u])) = O(V^2+E)$$
$$= O(V^2)$$

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)

1. INITIALIZE SINGLE-SOURCE (G, s)
2. S ← { } // S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q ← V[G]
4. while priority queue Q is not empty do
5. *u* ← EXTRACT_MIN(Q) // Pull out new vertex
6. S ← S ∪ {*u*}
    // Perform relaxation for each vertex *v* adjacent to u
7. for each vertex v in Adj[u] do
8. Relax (*u*, *v*, *w*)

Analysis

Like Prim's algorithm, Dijkstra's algorithm runs in $O(|E|\lg|V|)$ time.

Topological Sort

A topological sort of a directed acylic graph (DAG) G is an ordering of the vertices of G such that for every edge (e_i, e_j) of G we have $i < j$. That is, a topological sort is a linear ordering of all its vertices such that if DAG G contains an edge (e_i, e_j), then e_i appears before e_j in the ordering. DAG is cyclic then no linear ordering is possible.

In simple words, a topological ordering is an ordering such that any directed path in DAG G traverses vertices in increasing order.

It is important to note that if the graph is not acyclic, then no linear ordering is possible. That is, we must not have circularities in the directed graph. For example, in order to get a job you need to have work experience, but in order to get work experience you need to have a job (sounds familiar?).

Theorem *A directed graph has a topological ordering if and only if it is acyclic.*

Proof:

Part 1. G has a topological ordering if is G acyclic.

Let G is topological order.
Let G has a cycle (Contradiction).
Because we have topological ordering. We must have $i_0, < i, < \ldots < i_{k-1} < i_0$, which is clearly impossible.
Therefore, G must be acyclic.

Part 2. G is acyclic if has a topological ordering.

Let is G acyclic.
Since is G acyclic, must have a vertex with no incoming edges. Let v_1 be such a vertex. If we remove v_1 from graph, together with its outgoing edges, the resulting digraph is still acyclic. Hence resulting digraph also has a vertex *

Algorithm Topological Sort

TOPOLOGICAL_SORT(G)

1. For each vertex find the finish time by calling DFS(G).
2. Insert each finished vertex into the front of a linked list.
3. Return the linked list.

Given graph G; start node u with no incoming edges, and we let v_2 be such a vertex. By repeating this process until digraph G becomes empty, we obtain an ordering $v_1 < v_2 < , \ldots , v_n$ of vertices of digraph G. Because of the construction, if (v_i, v_j) is an edge of digraph G, then vi must be detected before v_j can be deleted, and thus $i < j$. Thus, $v_1, \ldots , v_n$ is a topological sorting.

Total running time of topological sort is $\Theta(V + E)$. Since DFS(G) search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.