Prim’s Algorithm

• In Prim’s algorithm, the set $A$ maintained by the algorithm forms a single tree.
• The tree starts from an arbitrary root vertex and grows until it spans all the vertices in $V$.
• At each step, a light edge is added to the tree $A$ which connects $A$ to an isolated vertex of $G_A = (V, A)$.
• By our Corollary, this rule adds only safe edges to $A$, so when the algorithm terminates, it will return a MST.

Example Run

Prim’s algorithm run on the example graph, starting with the bottom vertex.
At each stage, thick edges are in $A$, an arrow points along $A$’s safe edge, and dashed edges are useless.
An Implementation

- To implement Prim's algorithm, we keep all edges adjacent to $A$ in a heap.
- When we pull the minimum-weight edge off the heap, we first check to see if both its endpoints are in $A$.
- If not, we add the edge to $A$ and then add the neighboring edges to the heap.
- If we implement Prim's algorithm this way, its running time is $O(|E| \log |E|) = O(|E| \log |V|)$.
- However, we can do better.

Prim's Algorithm

- We can speed things up by noticing that the algorithm visits each vertex only once.
- Rather than keeping the edges in the heap, we will keep a heap of vertices, where the key of each vertex $v$ is the weight of the minimum-weight edge between $v$ and $A$ (or infinity if there is no such edge).
- Each time we add a new edge to $A$, we may need to decrease the key of some neighboring vertices.

Prim's

- We will break up the algorithm into two parts, Prim-Init and Prim-Loop.
- \[ \text{Prim}(V,E,s) \{
\text{Prim-Init}(V,E,s);
\text{Prim-Loop}(V,E,s);
\}\]

Prim-Init

- \[ \text{Prim-Init}(V,E,s)\{\]
  - for each vertex $v$ in $V - \{s\}$
    - if $((v,s)$ is in $E)$
      - $\text{edge}(v) = (v,s)$;
      - $\text{key}(v) = w((v,s))$;
    - else
      - $\text{edge}(v) = \text{NULL}$;
      - $\text{key}(v) = \infty$;
  
- $\text{Heap-Insert}(v)$;
}
Prim-Loop(V,E,s){
    A = {};
    for (i = 1 to |V| - 1){
        v = Heap-ExtractMin();
        add edge(v) to A;
        for (each edge (u,v) in E){
            if (u is not in A AND key(u) > w(u,v)){
                edge(u) = (u,v);
                Heap-DecreaseKey(u,w(u,v));
            }
        }
    }
    return A;
}

Runtime?

- The runtime of Prim’s is dominated by the cost of the heap operations Insert, ExtractMin and DecreaseKey
- Insert and ExtractMin are each called $O(|V|)$ times
- DecreaseKey is called $O(|E|)$ times, at most twice for each edge
- If we use a Fibonacci Heap, the amortized costs of Insert and DecreaseKey is $O(1)$ and the amortized cost of ExtractMin is $O(\log |V|)$
- Thus the overall run time of Prim’s is $O(|E| + |V| \log |V|)$
- This is faster than Kruskal’s unless $E = O(|V|)$

Note

- This analysis assumes that it is fast to find all the edges that are incident to a given vertex
- We have not yet discussed how we can do this
- This brings us to a discussion of how to represent a graph in a computer

Graph Representation

- There are two common data structures used to explicitly represent graphs
  - Adjacency Matrices
  - Adjacency Lists
The adjacency matrix of a graph $G$ is a $|V| \times |V|$ matrix of 0’s and 1’s.

For an adjacency matrix $A$, the entry $A[i,j]$ is 1 if $(i,j) \in E$ and 0 otherwise.

For undirected graphs, the adjacency matrix is always symmetric: $A[i,j] = A[j,i]$. Also the diagonal elements $A[i,i]$ are all zeros.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge.

We can also list all the neighbors of a vertex in $\Theta(|V|)$ time by scanning the row corresponding to that vertex.

This is optimal in the worst case, however if a vertex has few neighbors, we still need to examine every entry in the row to find them all.

Also, adjacency matrices require $\Theta(|V|^2)$ space, regardless of how many edges the graph has, so it is only space efficient for very dense graphs.
Adjacency Lists

- For sparse graphs — graphs with relatively few edges — we’re better off with adjacency lists
- An adjacency list is an array of linked lists, one list per vertex
- Each linked list stores the neighbors of the corresponding vertex

The total space required for an adjacency list is \( O(|V| + |E|) \)
- Listing all the neighbors of a node \( v \) takes \( O(1 + \text{deg}(v)) \) time
- We can determine if \( (u,v) \) is an edge in \( O(1 + \text{deg}(u)) \) time by scanning the neighbor list of \( u \)
- Note that we can speed things up by storing the neighbors of a node not in lists but rather in hash tables
- Then we can determine if an edge is in the graph in expected \( O(1) \) time and still list all the neighbors of a node \( v \) in \( O(1 + \text{deg}(v)) \) time

Take Away

- If we use the right type of heap and the right graph representation, then Prim’s algorithm takes \( O(|E| + |V| \log |V|) \)
- This compares favorably with Kruskal’s algorithm which takes \( O(|E| \log |V|) \)
- Kruskal’s and Prims algorithms are the two main algorithms for finding the minimum spanning tree of a connected graph
- There are many, many other types of problems defined on graphs . . .

Traversing a Graph

- Suppose we want to visit every node in a connected graph (represented either explicitly or implicitly)
- The simplest way to do this is an algorithm called depth-first search
- We can write this algorithm recursively or iteratively - it’s the same both ways, the iterative version just makes the stack explicit
- Both versions of the algorithm are initially passed a source vertex \( v \)
Recursive DFS:

```java
RecursiveDFS(v) {
    if (v is unmarked) {
        mark v;
        for each edge (v, w) {
            RecursiveDFS(w);
        }
    }
}
```

Iterative DFS:

```java
IterativeDFS(s) {
    Push(s);
    while (stack not empty) {
        v = Pop();
        if (v is unmarked) {
            mark v;
            for each edge (v, w) {
                Push(w);
            }
        }
    }
}
```

Generic Traverse:

```java
Traverse(s) {
    put (nil, s) in bag;
    while (the bag is not empty) {
        take some edge (p, v) from the bag;
        if (v is unmarked) {
            mark v;
            parent(v) = p;
            for each edge (v, w) {
                put (v, w) into the bag;
            }
        }
    }
}
```

- DFS is one instance of a general family of graph traversal algorithms.
- This generic graph traversal algorithm stores a set of candidate edges in a data structure we'll call a "bag".
- A "bag" is just something we can put stuff into and later take stuff out of - stacks, queues and heaps are all examples of bags.

**Generic Traverse**

Generic Traverse is one instance of a general family of graph traversal algorithms. This generic graph traversal algorithm stores a set of candidate edges in a data structure we’ll call a “bag.” A “bag” is just something we can put stuff into and later take stuff out of - stacks, queues and heaps are all examples of bags.
**Analysis**

- Notice that we’re keeping edges in the bag instead of vertices.
- This is because we want to remember when we visit vertex $v$ for the first time, which previously-visited vertex $p$ put $v$ into the bag.
- This vertex $p$ is called the *parent* of $v$.

**Proof**

- It’s obvious that no node is marked more than once.
- We next show that each vertex is marked at least once.
- Let $v \neq s$ be a vertex and let $s \to \cdots \to u \to v$ be the path from $s$ to $v$ with the minimum number of edges. (Since the graph is connected such a path always exists.)
- If the algorithm marks $u$, then it must put $(u, v)$ in the bag, so it must later take $(u, v)$ out of the bag, at which point $v$ must be marked.
- Thus by induction on the shortest-path distance from $s$, the algorithm marks every vertex in the graph.

**Lemma**

- Traverse(s) marks each vertex in a connected graph exactly once, and the set of edges $(v, \text{parent}(v))$, with parent(v) not nil, form a spanning tree of the graph.

**Proof**

- Call an edge $(v, \text{parent}(v))$ with parent($v$) $\neq$ nil a **parent edge**.
- It now remains to be shown that the parent edges form a spanning tree of the graph.
- For any node $v$, the path of parent edges $v \to \text{parent}(v) \to \text{parent}(\text{parent}(v)) \to \cdots$ eventually leads back to $s$, so the set of parent edges form a connected graph.
- Since every node except $s$ has a unique parent edge, the total number of parent edges is exactly one less than the total number of vertices.
- Thus the parent edges form a spanning tree (we’ll show this in the in-class exercise).
DFS and BFS

- If we implement the “bag” by using a stack, we have Depth First Search
- If we implement the “bag” by using a queue, we have Breadth First Search

DFS vs BFS

- Note that DFS trees tend to be long and skinny while BFS trees are short and fat
- In addition, the BFS tree contains shortest paths from the start vertex $s$ to every other vertex in its connected component. (here we define the length of a path to be the number of edges in the path)

Analysis

- Note that if we use adjacency lists for the graph, the overhead for the “for” loop is only a constant per edge (no matter how we implement the bag)
- If we implement the bag using either stacks or queues, each operation on the bag takes constant time
- Hence the overall runtime is $O(|V| + |E|) = O(|E|)$

Final Note

- Now assume the edges are weighted
- If we implement the “bag” using a priority queue, always extracting the minimum weight edge from the bag, then we have a version of Prim’s algorithm
- Each extraction from the “bag” now takes $O(|E|)$ time so the total running time is $O(|V| + |E| \log |E|)$
Example

A depth-first spanning tree and a breadth-first spanning tree of one component of the example graph, with start vertex \( a \).

In Class Exercise

- Consider a connected graph that has \( n \) vertices and \( n - 1 \) edges. Prove by induction on \( n \) that such a graph is a tree.
- Q: What is the base case?
- Q: What is the inductive hypothesis?
- Q: What is the inductive step?