COMP 182: Algorithmic Thinking
Prim and Dijkstra: Efficiency and Correctness

Luay Nakhleh

1 Prim's Algorithm

In class we saw Prim's algorithm for computing a minimum spanning tree (MST) of a weighted, undirected graph g. The pseudo-code is given in Algorithm 1.

Algorithm 1: Prim

Input: Weighted, undirected graph $g = (V, E, w)$.
Output: E_T, the edges of an MST of g.

1 $V_T \leftarrow \{x\}$; // x is an arbitrary node in V
2 $E_T \leftarrow \emptyset$;
3 **for** $i \leftarrow 1$ **to** $|V| - 1$ **do**
4 Let $e = \{u, v\}$ be a minimum-weight edge such that $u \in V_T$ and $v \in (V \setminus V_T)$;
5 $V_T \leftarrow V_T \cup \{v\}$;
6 $E_T \leftarrow E_T \cup \{e\}$;
7 **return** E_T;

1.1 Running Time: Without Using a Min-heap

Let us first assume that g is given by its adjacency matrix (with value ∞ in entry $[i, j]$ if there is no edge between i and j). We show two different running-time analyses. We denote by n and m the numbers of nodes and edges, respectively, in g. Clearly, Lines 1, 2, 5, 6, and 7 are all $O(1)$ operations, and the loop on Line 3 iterates $n - 1 = O(n)$ times. So, the two analyses will differ in terms of how Line 4 is implemented.

- **The most naive implementation of Line 4.** Go through every node $u \in V_T$, and for each such node, go through all neighbors of u, and finally return as v the node that is not in V_T and that has the lightest edge connecting it to one of the u's in V_T. Since the graph is given by its adjacency matrix, going through the neighbors of a node takes $n - 1 = O(n)$ time. Furthermore, since $|V_T| = O(n)$ is the worst case, we have the Line 4 takes $O(n^2)$ time, for a total running time of $O(n^3)$ for the algorithm.

- **A better implementation of Line 4.** Assume the nodes in V as numbered $\{0, 1, 2, \ldots, n - 1\}$. Then, Prim's algorithm could be rewritten as in Algorithm 2. In this algorithm, we maintain a list S of size $|V \setminus V_T$ such an element in S is a pair (j, q, k) where j is a node in $V \setminus V_T$ and q is the smallest weight of an edge that connects node j to a node in V_T, and k is the node in V_T that has an edge $\{k, j\}$ of weight q. Line 5 in Algorithm 2 finds the smallest-weight edge that connects a node $j \in (V \setminus V_T)$ to a node $k \in V_T$. This node and its associated edge are added to V_T and E_T on Lines 6 and 7, respectively. The loop at Line 9 checks for every node $a \in (V \setminus V_T)$ if its smallest-weight connecting to nodes in V_T is affected by the addition of node j to V_T; if it is, then the nodes that a connects to, along with the weight of the corresponding edge, are updated on Line 11. In this new algorithm, Lines 1, 2, 6, 7, 8, 10, 11, and 12 takes $O(1)$ operations each. Lines 3 and 5 take $O(n)$ operations each, in the worst case. The loop on Line 4 iterates $O(n)$ times, and the loop on Line 9 iterates $O(n)$ times in the worst case. Therefore, with this implementation of Prim's algorithm, the algorithm takes $O(n^2)$ time.

Algorithm 2: Prim

Input: Weighted, undirected graph $g = (V, E, w)$ with $V = \{0, 1, \ldots, n-1\}$.
Output: E_T, the edges of an MST of g.

```
1  V_T ← {0};
2  E_T ← ∅;
3  Let S be a list of size n − 1 initialized to have an element (j, w({0, j}), 0) for every j ∈ (V \ V_T);
4  for i ← 1 to |V| − 1 do
5      Let (j, q, k) be an element with the smallest value q of all elements in S;
6      V_T ← V_T ∪ {j};
7      E_T ← E_T ∪ {{j, k}};
8      Remove (j, q, k) from S;
9      foreach (a, b, c) ∈ S do
10         if w({j, a}) < b then
11             (a, b, c) ← (a, w({j, a}), j);
12 return E_T;
```

These two analyses illustrate important aspects of pseudo-code and running-time analysis. While Algorithm 2 gives more implementation details than Algorithm 1, it does not affect one's understanding of **what** the algorithm does. It only changes the details of **how** it does it. So, the pseudo-code gives sufficient details so that one understand **what** it does. When it comes to analyzing the running time, **how** an algorithm performs a certain step could impact the running time. A standard textbook description of Prim's algorithm is what is given in Algorithm 1. The extra details given in Algorithm 2 would be found in the analysis of the running time.

1.2 Running Time: Using a Min-heap

Assume now that the graph g is given by its adjacency list and that the list S in Algorithm 2 is maintained as a min-heap based on the second value; that is, for the elements (a, b, c) in S, the value b is the key that determines the "parental dominance" property of the heap. The pseudo-code, with minor changes, is given in Algorithm 3. With this new implementation, Line 3 takes $O(n)$ time (the time to construct a heap), Line 5 takes $O(\log n)$ (the time to remove the minimum element), and Lines 11 and 12 take $O(\log n)$ each (removing and adding an element to the heap). Lines 4, 9, 10, 11, and 12 take a combined $O(m \log n)$ amount of time (again, think about how many times each edge in the graph g will be visited). Lines 4 and 5 take a combined $O(n \log n)$ amount of time. Therefore, the running time of Algorithm 3 is $O(n + (m+n) \log n) = O((m+n) \log n) = O(m \log n)$ for a connected graph (in a connected graph, $n = O(m)$). The reason I focus on connected graphs is because there is no spanning tree of a disconnected graph. So, in the case of Prim's algorithm, graphs with fewer than $n - 1$ edges are not of interest (they'd be disconnected).

1.3 Correctness

We now prove that Prim's algorithm is correct. That is, we prove that given a weighted undirected graph $g = (V, E, w)$, Prim's algorithm as given in Algorithm 1 returns an MST of g. We prove this by mathematical induction.

Notice that the for-loop on Line 3 in Algorithm 1 iterates $n - 1$ times, where $n = |V|$, each time growing a spanning tree T of g by adding one edge to it. Let us denote by T_0 the spanning tree that the algorithm starts with before entering the for-loop. That is, the tree T_0 consists of only node 0 and no edges. Let us now denote by T_i, for $1 \leq i \leq n-1$ the tree obtained by adding to T_{i-1} the minimum-weight edge e identified on Line 4 in the i-th iteration of the algorithm. Obviously, tree T_{n-1} is the tree that Prim's algorithm returns. We now prove, by induction, that T_i, for $0 \leq i \leq n-1$, is a subtree of an MST of g.

- Base case: T_0 is part of an MST since T_0 consists of only node 0 and no edges. In fact, every MST of g must contain T_0 as a subtree.
- Inductive step: Assume that T_{i-1} is a subtree of an MST T of g (this is the inductive hypothesis) and let us prove that T_i is a subtree of an MST of g. We prove this by contraction. Assume T_{i-1} is a subtree of an MST of g but that T_i is not a subtree of any MST of g. Let $e_i = (u, v)$ be the minimum-weight edge that Prim's

Algorithm 3: Prim

Input: Weighted, undirected graph $g = (V, E, w)$ with $V = \{0, 1, \ldots, n-1\}$.
Output: E_T, the edges of an MST of g.

```
1  V_T ← {0};
2  E_T ← ∅;
3  Let H be a heap of the n − 1 elements (j, w({0, j}), 0) for every j ∈ (V \ V_T);
4  for i ← 1 to |V| − 1 do
5      Let (j, q, k) be an element with the smallest value q of all elements in H;
6      V_T ← V_T ∪ {j};
7      E_T ← E_T ∪ {{j, k}};
8      Remove (j, q, k) from H;
9      foreach (a, b, c) ∈ H do
10         if w({j, a}) < b then
11             Remove (a, b, c) from H;
12             Add (a, w({j, a}), j) to H;
13 return E_T;
```

algorithm identifies on Line 4 in the i-th iteration of the for-loop to add to T_{i-1} (see Fig. 1). Since we assume T_{i-1} is part of an MST of g but T_i is not, this means that edge e_i cannot be part of any MST of g. In other words, the presence of e_i creating a cycle.

Since T_{i-1} is a tree, and since e_i connects a node in T_{i-1} (a blue node in Fig. 1) and a node not in T_{i-1} (a red node in Fig. 1), then for e_i to create a cycle, there must be another edge $e' = (u', v')$ in the MST, with u' being a node in T_{i-1} and v' being a node not in T_{i-1}. If we delete edge e', then we break the cycle and have a spanning tree whose weight is at most the weight of MST T, since $w(e_i) \leq w(e')$ (Line 4 in Prim's Algorithm). Therefore, the spanning tree created by removing edge e' from MST T and replacing it by edge e_i is also an MST, contradicting the assumption that T_i is not a subtree of any MST. Therefore, T_i is part of an MST of g.

By induction, it follows that T_{n-1} is an MST of g.

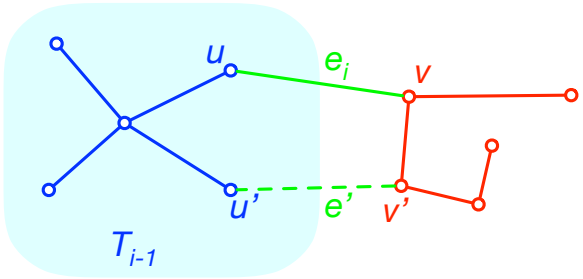

Figure 1: **Correctness of Prim's Algorithm.** T_{i-1} is the tree that Prim's Algorithm has built by the end of iteration $(i-1)$ of the for-loop on Line 3 in Algorithm 1. Edge e_i is the minimum-weight edge it identifies on Line 4 to add to T_{i-1} in order to create tree T_i.

2 Dijkstra's Algorithm

In class we saw Dijkstra's algorithm for computing shortest paths and their lengths from a source node to all other nodes in a weighted directed graph, where all edge weights are nonnegative. The pseudo-code is given in Algorithm 4.

Algorithm 4: Dijkstra.

Input: Undirected, weighted graph $g = (V, E, w)$ where are weights are nonnegative; source node i.
Output: Shortest paths, as well as their lengths, from i to every other node in g.

```
1  X ← ∅;
2  foreach j ∈ V do
3      d_j ← ∞ ;                          // d_j is the geodesic distance between i and j
4      p_j ← null ;                       // p_j carries the node label of j's parent
5      X ← X ∪ {j} ;                      // Set X contains all nodes not visited yet
6  d_i ← 0;
7  while X ≠ ∅ do
8      Let k be a node with the minimum value of d_k in the set X;
9      if d_k = ∞ then
10         break;
11     X ← X \ {k};
12     foreach neighbor h of k in the set X do
13         if d_k + w((k, h)) < d_h then
14             d_h ← d_k + w((k, h));
15             p_h ← k;
16 return p, d;
```

2.1 Running Time: Using a Min-heap

Let us assume that input graph g is represented by its adjacency list, and let n and m be the numbers of nodes and edges, respectively, of g.

Lines 1 and 6 take $O(1)$ time, and Lines 2–5 take $O(n)$ time to initialize d and p for all nodes.

Let us assume that the d_k values are stored in a min-heap so that finding a node k with the minimum d_k on Line 8 takes $O(1)$ time and that removing that value from the heap and updating it takes $O(\log n)$ time. Analyzing Lines 7–15 carefully reveals two main components that determine the running time:

- Finding node k on Line 8 and removing it from the min-heap once k is removed from X on Line 11. This takes $O(\log n)$ in the worst case, and it is executed for every node (until X is empty) for a total of $O(n \log n)$.
- Line 14 results in an update to d_h. Such an update might require updating the min-heap with respect to value d_h, which costs $O(\log n)$ in the worst case. Similar to our analysis of **BFS**, the update on Line 14 happens $O(m)$ times since the loop on Line 12 iterates for a total of m times. Therefore, in the worst case, Lines 7–15 take $O(m \log n)$ time.

Therefore, the running time of Dijkstra's algorithm when g is represented by its adjacency list and a min-heap is used to store d, is $O((m + n) \log n)$. Since the algorithm is usually run on connected graphs (where $m \geq n$), the running time is $O(m \log n)$.

2.2 Correctness

Notice that Dijkstra's algorithm builds a tree of paths, where the nodes of the tree are the nodes of the graph (assuming all nodes are reachable from source node i) and the edges are given by the p values.

Just as in proving the correctness of Prim's Algorithm, let us denote by T_j, $1 \leq j \leq n$, the tree with the first j nodes that Dijkstra's algorithm has finished processing (computing d and p thereof) after iteration j of the loop on Line 7. We demonstrate the correctness Dijkstra's algorithm by proving that T_j contains j closest nodes to source node i (including node i itself). We prove this result by induction on the number of nodes in T_j.

- Base case: For $j = 1$, the only node in T_1 is the source nodes i, and it is closest node to the source node, that is, itself.
- Inductive step: Assume T_j contains j closest nodes to source node i (the inductive hypothesis). Let T_{j+1} be the tree formed by adding node v_{j+1} to tree T_j. All the nodes that fall on shortest paths from i to v_{j+1} must be

already in T_j since they are closer to i than v_{j+1} is. Hence, the $(j+1)$-st closest node to i is a node h that is not in T_j and that minimizes $w((q,h)) + d_q$ over all nodes q in T_j. But this is precisely what Dijkstra's algorithm does.

By induction, tree T_n contains n closest nodes to source node i.