Computer Science 331

Analysis of Prim's Algorithm

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #34

Outline

Introduction

Objective for Today:

- Proof of the Correctness and Efficiency of Prim's Algorithm (as presented last time)
- **Note:** The specification of requirements for this problem (including a pre-condition and post-condition) and pseudocode for Prim's algorithm were included in the previous set of notes.

Colour Properties

The following properties are proved by inspection of the code:

1. *Colour Properties:*
   - The initial colour of every node $v \in V$ is **white**.
   - The colour of a vertex can change from **white** to **grey**.
   - The colour of a vertex can change from **grey** to **black**.
   - No other changes in colour are possible.
2. *Contents of Priority Queue:* The following properties are part of the *loop invariant* for the **while** loop:
   - If (u, d) is an element of the priority queue then $u \in V$, $colour[u] =$ **grey**, and $d = d[u]$.
   - If a vertex v (and its cost) were included on the queue but have been removed, then $colour[v] =$ **black**.
   - Vertices that have never been in the queue are **white**.

Colour Properties

The following properties are also part of the loop invariant for the **while**-loop.

- For all vertices $v \in V$, if the colour of v is **grey**, then
  - $(v, d[v])$ is an element of the priority queue
  - Either $v = s$ and $d[v] = 0$ or $v \neq s$ and

$$d[v] = \min_{\substack{w \in V \\ colour[w]=\textbf{black} \\ (w,v)\in E}} w((v,w))$$

  - If $v \neq s$ then $\pi(v) \in V$, $colour(\pi(v)) = \textbf{black}$, $(\pi(v), v) \in E$, and $w((\pi(v), v)) = d[v]$.
- The neighbours of any **black** vertex in G are either **grey** or **black**.

Black Subtree

Consider $G_b = (V_b, E_b)$, where

- $V_b = \{v \in V \mid colour(v) = \textbf{black}\}$
- $E_b = \{(\pi(v), v) \mid v \in V_b \text{ and } \pi(v) \neq \text{NIL}\}$

The following properties hold at the end of each execution of the **while** loop (and are part of the loop invariant).

- $V_b \subseteq V$ and $E_b \subseteq E$
- Either $|V_b| = |E_b| = 0$, or
  - For all $u \in V$, if $colour(u) = \textbf{black}$ and $\pi(u) \neq \text{NIL}$ then $colour(\pi(u)) = \textbf{black}$ as well
  - For all $u, v \in V$, if $(u, v) \in E_b$ then $(u, v) \in E$ as well (so that $E_b \subseteq E$)

  Thus $G_b = (V_b, E_b)$ is a *subgraph* of G.

Black Subgraph is a Tree

The following also holds at the end of each execution of the **while** loop and should be part of the loop invariant.

Claim:

If $V_b \neq \emptyset$ then G_b is a **tree**.

Sketch of Proof.

If $V_b \neq \emptyset$ then $|E_b| = |V_b| - 1$ because

- one edge per vertex in V_b (except for s)

If $V_b \neq \emptyset$ then $G_b = (V_b, E_b)$ is a *connected* graph because

- $\pi(v)$ is black (and thus in V_b) for all $v \in V_b$

Conclusion: If $V_b \neq \emptyset$ then G_b is a tree (Lemma 7 of Lecture 30) □

Black Subgraph is a Subgraph of a Spanning Tree

Claim:

If $\widehat{G} = (\widehat{V}, \widehat{E})$ is a subgraph of G and acyclic, then $\widehat{G}$ is also a subgraph of some spanning tree T of G.

Method of Proof: induction on $|\widehat{E}| - (|V| - 1)$ (note that $|\widehat{E}| \leq |V| - 1$, since $\widehat{V} \subseteq V$ and $\widehat{G}$ is acyclic).

Key Idea: If $|\widehat{E}| < |V| - 1$ then it must be possible to include another edge from E without creating a cycle — otherwise the graph G would not be connected!

Application: The following is also part of the loop invariant:

Claim:

G_b is a subgraph of some spanning tree T of G.

Black Subtree is a Subgraph of a MST

The following is also true at the end of each execution of the **while**-loop (and is part of the loop invariant).

Claim:

G_b is a subgraph of a minimum-cost spanning tree of G.

Comments on Proof:

- This is true before and after the first execution of the loop body (when $V_b = \emptyset$ and when $V_b = \{s\}$) because G_b is a subgraph of **every** spanning tree of G in these cases.
- **Complication:** There can be more than one minimum-cost spanning tree of G, and G_b is generally *not* a subgraph of all such spanning trees later on in the computation.

More Comments on the Proof

Structure of Proof:

- Keep track of *some* minimum-cost spanning tree T of G such that G_b is a subgraph of T.
- During an execution of the loop body (for the second and all subsequent executions) a vertex and edge are each added to G_b to produce a larger subgraph G_b'.
- If G_b' is not a subgraph of the spanning tree T then *another* spanning tree T' is constructed such that
  - T' is also a *minimum-cost* spanning tree of G
  - G_b' is a subgraph of T'.

The complete proof of this claim will be provided in a separate handout.

On the Growth of V_b

Here is another part of the loop invariant for the **while**-loop.

Claim:

*If $k \geq 0$ and the body of the **while** loop is executed k or more times then, at the end of the k^{th} execution of the body of the loop,*

$$|V_b| = k.$$

Method of Proof: induction on k

Corollary 1

*The body of the **while** loop is executed at most $|V|$ times.*

Explanation: each vertex is enqueued at most once

On the Growth of V_b

Claim:

*If $0 \leq k < |V|$ then the priority queue is nonempty (and one or more grey vertices exist) immediately after the k^{th} execution of the body of the **while** loop.*

Proof (by contradiction).

- Suppose that $0 \leq k < |V|$ and the priority queue is empty after the k^{th} execution of the body of the **while** loop.
- $|V_b| = k < |V|$ at this point.
- $k \geq 1$ and $s \in V_b$ at this point, so the colour of s is **black**.
- All neighbours of **black** vertices are **black**. *Explanation:* neighbors of black vertices are black or grey, but there are no grey vertices (empty queue)

On the Growth of V_b (continued)

Proof (continued).

- It follows that the only nodes that are *reachable* from **black** nodes are also **black**.
  Explanation: neighbors of black nodes are black or grey, no grey nodes (empty queue)
- However, since s is a **black** node and there is at least one **white** node at this point, it follows that at least one node is *not* reachable from s.
- In other words, G is not connected — but the pre-condition includes the assertion that G *is* a connected graph.

Conclusion: The body of the **while** loop is executed *exactly* $|V|$ times and $V_b = V$ on termination of this loop. □

Partial Correctness

Suppose the **pre-condition** of the algorithm holds initially, that is, $G = (V, E)$ is a **connected** weighted graph.

Properties Established on Termination:

- $G_b = (V_b, E_b)$ is a subgraph of a MST of G.
- G_b is a tree.
- $V_b = V$.
- *Conclusion:* G_b *is* a minimum-cost spanning tree of G.
- Since $V_b = V$, the set of edges

$$\widehat{E} = \{(\pi(v), v) \mid v \in V \text{ and } \pi(v) \neq \text{NIL}\}$$

  is the same as the set of edges E_b included in G_b.

Note: The **post-condition** can be established!

Termination and Efficiency

Claim:

If MST-Prim is executed on a weighted undirected graph $G = (V, E)$ then the algorithm terminates after performing $O((|V| + |E|) \log |V|)$ steps in the worst case.

Proof.

This is virtually identical to the proof of the corresponding result for Dijkstra's algorithm (to compute minimum-cost paths).

- The number of operations on the priority queue, and the number of operations that do not involve this data structure, are each in $O(|V| + |E|)$ in the worst case (by the argument that has been applied to the last three algorithms considered).

Termination and Efficiency (cont.)

Proof (continued).

- Since the size of the priority queue never exceeds $|V|$ and since the only operations on the priority queue used are insertions, decreases of key values, and extractions of the minimum (top priority) element, the cost of each operation on the data structure is in $O(\log |V|)$.
- It follows immediately that the total number of steps is in $O((|V| + |E|) \log |V|)$, as claimed. □

$O(|V| \log |V| + |E|)$ using a Fibonacci heap (amortized)

Additional Comments

On Greedy Algorithms

- Prim's algorithm is an example of a **greedy** algorithm: A "global" *optimization* problem (finding a minimum-cost spanning tree) is solved by making a sequence of "local" *greedy choices* (by extending a tree with edges whose weights are as small as possible).
- Proving correctness of greedy algorithms is often challenging. Indeed, greedy heuristics are often *incorrect*.
- On the other hand, *when they are correct*, greedy algorithms are frequently simpler and more efficient than other algorithms for the same computation.
- See CPSC 413 for more about greedy algorithms!

Additional Comments and References

References

References

- Cormen, Leiserson, Rivest and Stein
  Introduction to Algorithms, Second Edition

  Chapter 23 includes Prim's algorithm along with another greedy algorithm for this problem (Kruskal's algorithm), as well as a more general argument that establishes the correctness of both.