Computational Optimization
ISE 407

Lecture 20

Dr. Ted Ralphs

Breadth-first Search

- Processing the vertices in first-in, first-out (FIFO) order results in an algorithm called *breadth-first search* (BFS).
- This corresponds to the policy of choosing a vertex at minimum depth as the next to be processed.
- The implementation is identical to DFS, except that the neighbors of the vertex being processed are inserted into a queue, instead of a stack.
- This creates a very shallow search tree, unlike DFS.

BFS and Shortest Paths

- Consider the problem of finding the *shortest path* from a vertex u to a vertex v.
- A shortest path from u to v is a path containing the fewest intermediate vertices.
- A *shortest paths tree* (SPT) is a rooted tree in which the path from the root vertex to each other vertex in the graph is a shortest such path in the original graph.
- Question: Does such a tree always exist?
- Answer: Yes.
- Question: How do we find it?
- Answer: The search tree created by performing a BFS is an SPT.
- Why is this the case?

Weighted Graphs

- For most practical applications, we will need to consider *weighted graphs*.
- In a weighted graph, each edge has a real number, called its *weight*, associated with it.
- Usually, the weights are specified using a separate weight vector $w \in \mathbb{R}^E$.
- Depending on the application, edge weights can be interpreted in a number of different ways.
  - They are interpreted as lengths or distances in cases where the graph models a physical network, such as a transportation network.
  - They are also frequently interpreted as costs associated with building or operating a network.

Weighted Shortest Paths

- In a weighted graph, the length of a path is the sum of the weights of the edges encountered on the path.
- A shortest path between two vertices in a weighted graph is a path connecting the two vertices that is of minimum length.
- In a transportation network, the edge weights may represent distances between physical locations, such as specific intersections.
- In such a weighted graph, the *shortest path* between two vertices has a natural physical interpretation.
- We are interested in being able to find such a path.
- Actually, we will consider the problem of finding an entire shortest paths tree rooted at a given source vertex.

Weighted Shortest Paths Tree

- An SPT is exactly the same in the weighted case as in the unweighted case.
- Is such a tree still guaranteed to exist?
- Question: Is there always a shortest path between any two vertices in a weighted, undirected graph?
- Answer: Not always.
- If the graph has edges of negative length, there may not exist a shortest path.
- A shortest path exists if and only if there are no *negative length cycles* reachable from either vertex.
- If there are no cycles of negative length, then there is always a shortest path with no cycles.
- This is essentially all we need to show the existence of an SPT.

Properties of Shortest Paths Trees

- Let $G = (V, E)$ be an undirected graph with an associated weight vector $w \in \mathbb{R}^E$.
- Suppose T is an SPT rooted at r and define $\delta(v)$ to be the path length from r to v in the tree.
- By definition, we must have that $\delta(v)$ is the length of a shortest path from r to v.
- For any node u on the path from r to v, the length of a shortest path u to v must be $\delta(v) - \delta(u)$.
- For any edge $e = \{u, v\} \in E$ that is not part of T, we must have that $\delta(v) \leq \delta(u) + w_e$.
- To show that T is an SPT, we need to show that the inequalities above hold for all edges not in T.

Finding the Shortest Paths Tree

- Assuming that all the edge lengths are positive integers, one approach is to subdivide each weight edge into unweighted edges of unit length.
- In other words, we replace an edge of length l with a path consisting of l edges of unit length.
- This essentially converts a weighted graph into an unweighted graph.
- After converting to an unweighted graph, we could simply use breadth-first search to find the (unweighted) SPT.
- This could be converted back to the weighted SPT by contracting the paths that were added back into single edges.
- This is a potentially disastrous algorithm since the running time depends on the number of edges.
- Fortunately, we can modify the algorithm to eliminate this dependence.

Finding Shortest Paths in Acyclic Networks

- In acyclic networks, finding shortest paths is easier than in networks that have cycles.
- If we topologically order the nodes first, then we can compute shortest paths.
- Let $G = (V, E)$ be a given graph and let `order` be a topological ordering of the nodes. Then we can solve the SPP as follows.

```
def acyclic_shortest_paths(G):
    dist = {}
    for j in range(len(G.get_node_list())):
        m = order[j]
        for n in G.get_neighbors(m):
            estimate = dist[m] + G.get_edge_weight(m ,n)
            if dist[n] > estimate:
                dist[n] = estimate
    return dist
```

Proof of Correctness for Algorithm

Claim: When the algorithm examines a node, its distance label is optimal.

Proof

Base Case: When we examine node 1, $d(1) = 0$ is correct. When we examine node 2, $d(2) = d(1) + c_{12}$ is correct because only node 1 can be inbound to node 2 (topological ordering).

Induction: Suppose that the algorithm has examined nodes $1, 2, \cdots, k$ and the distance labels are correct. Show that the distance label for node $k+1$ is correct.

Let the shortest path from s to $k+1$ be $s - i_1 - i_2 - \cdots - i_h - i_k + 1$.

Solving SPP with Non-Negative Arc Lengths

- When there are cycles, the situation is a bit more complex.
- Dijkstra's Algorithm generalizes the previous algorithm for the acyclic case.
- The difference is the order in which the nodes are examined.
- Nodes are divided into two groups
  - temporarily labeled
  - permanently labeled
- In order to produce the shortest paths tree, we keep track of the *predecessor node* each time a label is updated.
- Basic Idea: Fan out from source and permanently label nodes in order of distance from the source.

Dijkstra's Algorithm

- We will assume for now that the edge lengths are all positive.
- The idea of Dijkstra's Algorithm is to perform a graph search, updating the shortest known path to each encountered vertex as the search evolves.
- Throughout the algorithm, we maintain a quantity $d(v)$ for each node v, which represents the length of the shortest path found so far.
- We will call $d(v)$ the current *estimate* for node v.
- We start by assigning $d(r) = 0$ for the source node and $d(v) = \infty$ for all other nodes v.
- The node v to be processed next is the unprocessed node for which $d(v)$ is minimized.
- The processing step consists of updating the estimates for all the neighbors of v.

Algorithm Summary

- We are given a graph $G = (V, E)$ and a source node node r from which we want to find shortest paths to all other nodes.
- Algorithm
  - Initialize by assigning $d(r) = 0$ for the source node and $d(v) = \infty$ for all other nodes $v \in V \setminus \{r\}$.
  - Place r on the list L of unprocessed nodes.
  - While L is not empty
    - Choose $v \in L$ such that $d(v) = \min_{u \in L} d(u)$.
    - For each neighbor x of v, set $d(x) = \min\{d(x), d(v) + w_{\{v,x\}}\}$.
- When the algorithm is completed, we will have $d(v) = \delta(v)$ for all $v \in V$ and the search tree will be a shortest paths tree.
- Why is this algorithm correct?

Proof of Correctness

Claim: At the end of any iteration the following inductive hypotheses hold:

1. The distance label $d(i)$ is optimal for any node i in the set S.
2. The distance label $d(j)$ for any node $j \in \bar{S}$ is the length of the shortest path from the source to j such that all internal path nodes are in S.

Proof Strategy

- Show that statements 1 and 2 are true after the first iteration.
- Assume that they are true after iteration $i-1$ and prove that they hold after iteration i.
- (Assume iteration i moves node i from $\bar{S}$ to S.)

Implementing the Algorithm with Priority Queues

- To implement the algorithm, we need to maintain the node list L as a *priority queue*.
- Recall that a priority queue is a data structure for maintaining a list of items that have associated *priorities*.
- The usual operations are
  - construct a queue from a list of items.
  - find the item with the highest priority.
  - insert an item.
  - delete an item.
  - change the priority of an item.
- A "naive" implementation is to simply maintain a vector of the estimates for each node and then scan through it each time to find the minimum.
- We may be able to do better using a heap.

Minimum Weight Spanning Trees

- Consider an undirected graph $G = (V, E)$ with weight vector $w \in \mathbb{R}^E$.
- If $T \subseteq E$ is a spanning tree of G, the *weight* of T is

$$\sum_{e \in T} w_e$$

- The *minimum weight spanning tree* (MST) problem is that of finding, among all spanning trees of G, one that has minimum weight.
- How many spanning tree are there?
- What about simply enumerating all of them?

Finding a Minimum Weight Spanning Tree

- Although finding an MST may seem to be a difficult problem, it can be solved efficiently.
- The first algorithm we'll consider uses another variant of graph search to build a search tree that is guaranteed to be an MST.
- We build the tree up, adding one vertex at a time.
- At each iteration, we have a partially completed tree that spans the vertices that have been processed so far.
- The vertex that is processed next is the one that is "closest" to the partially completed tree.
- The algorithm is almost identical to Dijkstra's Algorithm.

Algorithm Summary: Prim's Algorithm

- We are given a connected undirected weighted graph $G = (V, E)$ and we want to find an MST of G.
- Prim's Algorithm
  - Arbitrarily choose a source node r.
  - Initialize by assigning $d(r) = 0$ for the source node and $d(v) = \infty$ for all other nodes $v \in V \setminus \{r\}$.
  - Place r on the list L of unprocessed nodes.
  - While L is not empty
    - Choose $v \in L$ such that $d(v) = \min_{u \in L} d(u)$.
    - For each neighbor x of v, set $d(x) = \min\{d(x), w_{\{v,x\}}\}$.
- When we're finished, the search tree will be an MST.
- Why is this algorithm correct?
- How do we implement it?
- What is the running time?

Another View of Prim's Algorithm

- Prim's Algorithm can be viewed as a special case of graph search.
- The algorithm can also be viewed as a special case of another general class of algorithms called *greedy algorithms*.
- A *greedy algorithm* is one that makes the choice at each step that looks the best "at the moment" and doesn't reconsider that choice later.
- We can view the construction of an MST as a greedy algorithm, but first we must define some terminology.
- Given an undirected graph $G = (V, E)$, a *cut* is a set $S \subset V$ that defines a partition of V into two nonempty subsets, S and $V \setminus S$.
- An edge is said to *cross the cut* if it connects a node in S to a node in $V \setminus S$.
- Our goal is to build a spanning tree by adding one edge at a time to a set T in a "greedy" fashion.
- Basically, we just need to somehow guarantee ourselves that at each step, the current set can be "extended" to an MST.
- How do we do that?

Safe Edges

- Let's assume that our current set of edges T already satisfies the property that T can be extended to an MST.
- Question: What edges can we add to T to maintain the property?
- Answer: Any edge that is a minimum edge crossing some cut S.
- Rationale: In any connected graph, there must be an edge crossing each cut in the graph (why?)
- We will call such edge a *safe edge* if it also doesn't create a cycle when added to T.
- How do we find such an edge?
  - Prim's Algorithm simply considers the cut S consisting of nodes that have already been processed.
  - At each step, we add the minimum edge crossing that cut.
  - There are other possibilities, however.

Generic Greedy Algorithm for Building an MST

- Generic greedy algorithm for constructing a spanning tree.
  - Set $T = \emptyset$.
  - Select a safe edge and add it to T.
  - Repeat until T is a spanning tree.
- This is guaranteed to work, no matter how the safe edges are selected.