CS161 - Minimum Spanning Trees and Single Source Shortest Paths

David Kauchak

Single Source Shortest Paths

- Given a graph G and two vertices s, t what is the shortest path from s to t?

  For an unweighted graph, BFS gives us a solution to this problem.

  For weighted graphs, as it turns out, we can calculate the shortest distance from s to all vertices $t \in V$ in worst case the same amount of time for any particular t, so we'll look at this problem, which is the single source shortest paths.

- Shortest path property

  If the path $v_1, v_2, v_3, ..., v_k$ where $v_i \in V$ is the shortest path from v_1 to v_k then for all $1 \leq i \leq j \leq k$, $v_i, v_{i+1}, ..., v_j$ is the shortest path from v_i to v_j

  Proof: Consider that a shorter path exists between v_i and v_j, then we could use this path instead of the path $v_i, v_{i+1}, ..., v_j$ in the path from v_1 to v_k, resulting in a shorter path from v_1 to v_k, but this is a contradiction.

- General idea for all the algorithms

  mark each vertex with an upper bound on the distance from the source to that node. Decrease that value until it is correct.

- Dijkstra's algorithm

  Assume that all of the weights are positive

Like BFS, exept our frontier that we expand is based on the weights of the edges not the number of edges

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Example

Why doesn't this hold with negative weights?

Consider the graph:

$A \to B : 1, C : 10$
$B \to D : 1$
$C \to D : -10$
$D \to E : 5$

What is the shortest path from A to E?

- Is it correct?

  Invariant: For every vertex that has been visited/removed from the heap, $dist[v]$ is the actual shortest distance from s to v

  The only time a vertex u gets visited is when the distance from s to that vertex is smaller than any remaining vertex. In addition, because we enforce positive weights, there cannot be any other path to u that hasn't been visited already that would result in a shorter path, since all paths visited in the future are longer.

– Runtime

  Depends on the heap implementation

  1 call to MakeHeap

  $|V|$ calls to ExtractMax

  $|E|$ calls to DecreaseKey

  1. array
     $V + V * V + E = O(V^2)$
  2. binary heap
     $V + V \log V + E \log V = O((V + E) \log V) = O(E \log V)$

     if $E < V^2/\log V$ then this is an improvement
  3. fibonacci heap
     $V + V \log V + E = O(V \log V + E)$

- negative cycles

  positive cycles - if a positive cycle exists can a path through that cycle be the shortest path?

  negative cycles - What happens when a negative cycles exists along the path to a negative cycle?

- Bellman-Ford algorithm (general case)

Bellman-Ford(G, s)

```
1   for all v ∈ V
2           dist[v] ← ∞
3           prev[v] ← null
4   dist[s] ← 0
5   for i ← 1 to |V| − 1
6           for all edges (u, v) ∈ E
7                   if dist[v] > dist[u] + w(u, v)
8                           dist[v] ← dist[u] + w(u, v)
9                           prev[v] ← u
10  for all edges (u, v) ∈ E
11          if dist[v] > dist[u] + w(u, v)
12                  return false
```

Example

- Is it correct?
  Assuming no negative cycles (along the paths from s),

  Invariant: After any iteration i, all i edge paths from the source s to any vertex are the shortest possible path of i edges or less.

  For $i = 1$ this is true, since we're only traversing one edge, so the distance for any vertex v, 1 edge away from s will be $w(s, v)$, which is the shortest path.

  Consider the difference between paths of length $i - 1$ and paths of length i. There are two options:

  * Adding another edge decreases the length of a particular path In this case, the comparison at line 6 will notice this difference (since it iterates over all edges) and the new distance for v will be updated accordingly
  * Adding another edge doesn't decrease the length of a particular path In this case, the comparison at line 6 will not be true and no changes will be made, so the invariant still holds

  Does it identify negative cycles?

  The check in lines 9-11, see if we can continue to decrease the shortest path to a node. The only time this can happen, is if there is a negative cycle exists since all paths of length $V-1$ should already have the correct values. Any path longer than this must contain a cycle. A positive cycle would not decrease the value, so it must be a negative cycle.

  - Running time

    $V-1$ loops and each loop iterates over all edges, $O(VE)$

    Is there any way we can speed this up slightly? What happens if at a given iteration we don't update any distances?

- Dags

  Adds the constraint that there are no cycles

Minimum Spanning Trees

- what is the problem?

  What is the lowest weight set of edges that connects all vertices of an undirected graph with positive weights?

  Can there be cycles?

  Example

- what are the applications

  - Network connectivity
  - Wiring connectivity

- Cut property

  What is a cut?

  Let S be a subset of the vertices and let edge $e=(u,v)$ be the minimum cost edge with $u \in S$ and $v \in V-S$. Every minimum spanning tree contains the edge e.

  Proof: Consider a minimum spanning tree T that does not contain e. There must be come cycle in the graph that contains an edge

$e' = (u', v')$ with $u' \in S$ and $v' \in V - S$ with a higher weight (otherwise e would be the only option for creating an spanning tree).

If we remove e' from the spanning tree and include e, we will still have a spanning tree since we still connect sets S and $V - S$. However, this new tree will have a lower weight since the weight of e is less than the weight of e', so T is not a minimum spanning tree.

We'll use this property to prove the correctness of the MST algorithms.

- Kruskal's algorithm

  Add the lowest weight edge to the tree as long as that edge does not connect two vertices that are already connected via some other path.

  ```
  Kruskal(G)
  for all v ∈ V
          MakeSet(v)
  T ← {}
  sort the edges of E by weight
  for all edges (u, v) ∈ E in increasing order of weight
          if Find-Set(u) ≠ Find-Set(v)
                  add edge to T
                  Union(Find-Set(u),Find-Set(v))
  ```

  Example

  1. Is it correct?

     Let S be the set FIND-SET(u). The edge (u, v) is the minimum edge from S to $V - S$ since we're visiting edge in increasing order and if S were connected to $S - V$ then FIND-SET(u) $\neq$ FIND-SET(v) would not be true. Therefore, by the cut propery, e must be part of the MST.

  2. Running time

     V calls to MAKESET

     Sort the edges: $O(E \log E)$

$2E$ calls to Find-Set

$V - 1$ calls to Union

Depends on the implementation of the sets

- Linked lists
  $V + E \log E + E * V + V = O(E * V)$
- Linked lists + heuristics (see section 21.3 of [1])

$$\begin{aligned} V + E \log E + E \log V + V &= O(E \log E + V \log V) \\ &= O(E \log V + V \log V) \\ &= O((E + V) \log V \\ &= O(E \log V) \end{aligned}$$

- Prim's algorithm

  Start at some root node and build out the MST by adding the lowest weighted edge at the frontier.

  ```
  Prim(G, r)
  for all v ∈ V
          key[v] ← ∞
          prev[v] ← null
  key[r] ← 0
  H ← MakeHeap(key)
  while !Empty(H)
          u ← Extract-Min(H)
          visited[u] ← true
          for each edge (u, v) ∈ E
                  if !visited[v] and w(u, v) < key(v)
                          Decrease-Key(v, w(u, v))
                          prev[v] ← u
  ```

  Example

  - Is it correct?
    Let S be the set of vertices visited so far (i.e. $v : visited[v] = true$). The only time a new edge is added to the MST is when it

is the lowest weight edge from S to $V - S$ because we use a heap and we only add edges from nodes in S. Therefore, by the cut property, this added edge is part of the MST.

- Runtime
  V initialization operation of $\Theta(1)$

  1 call to MakeHeap

  V calls to Extract-Min

  E calls to Decrease-Key

  1. Binary heap
     $V + E + V \log V + E \log V = O((V + E) \log V) = O(E \log V)$
  2. Fibonacci heap
     $V + E + V \log V + E = O(V \log V)$

These notes are adapted from material found in chapters 22,23 of [1], chapter 4 of [2] and chapters 4,5 of [3]

References
[1] Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest and Clifford Stein. 2007. Introduction to Algorithms, 2nd ed. MIT Press.
[2] Jon Kleinberg and Eva Tardos. 2006. Algorithm Design. Pearson Education, Inc.
[3] Sanjoy Dasgupta, Christos Papadimitiou and Umesh Vazirani. 2008. Algorithms. McGraw-Hill Companies, Inc.