31251 – Data Structures and Algorithms

Week 9 - Graphs Part II

Luke Mathieson

Does anyone read these titles?

- Dynamic Programming
  - Example: Dijkstra's Algorithm
- Connectivity in Graphs
  - Connectivity and Connected Components
  - Strongly Connected Components
  - Articulation Points
- Dependencies in Directed Graphs

Dynamic Programming

Dynamic Programming

- Dynamic programming looks a lot like Divide-and-Conquer, but subinstances *overlap*.
- Usually implemented with some sort of table that keeps track of subinstance solutions.
- Normally they work bottom-up (the reverse of D&C):
  - Solve the smallest subinstances, combine them to make bigger subinstances, keep going until you have the whole thing.
- Can be done top-down (recursively) as well.

What makes it work?

- **Optimal Substructure:** The same as greedy algorithms and D&C.
- **Overlapping Subproblems:** As mentioned before.
- That's it actually - it's is very general technique.

Shortest Paths

Single Source Shortest Paths

- In unweighted graphs, the distance between two vertices is easy to calculate:
  - Just do a breadth first search.
  - If you're looking for a particular vertex, stop when you find it.
  - The distance is the same as the number of iterations taken to get to the vertex.
- In weighted graphs, it's not so easy.
  - The lowest weight path is not always the path with fewest edges.

Dijkstra's Algorithm

- The backbone algorithm for most shortest path algorithms.
- Employs a dynamic programming approach.
- It's not too far removed from Prim's algorithm
  - It builds a shortest path tree instead,
  - but a basic greedy approach doesn't work in deciding which edge to add.
- Doesn't handle negative edge weights (in the basic version).

Dijkstra's Algorithm

Given a graph G with non-negative edge weights, with a given starting vertex u:

1. Set the tentative distances for all vertices; 0 for u, infinity for all others.
2. Mark all vertices as unvisited.
3. Set u as the current vertex.
4. While there are unvisited vertices:
    1. For each unvisited neighbour of the current vertex:
        1. Compare the tentative distance to the neighbour with the distance to current plus the edge weight of the edge joining them.
        2. Keep whichever is smaller.
    2. Mark the current vertex as visited.
    3. Select the unvisited vertex with smallest tentative distance and set it as the current vertex.

Dijkstra's Algorithm – Correctness

Let S be the set of marked vertices, u the starting vertex, and $d(v)$ the calculated distance from u to v, then the correctness of Dijkstra's algorithm can be stated as:

Lemma

For every vertex $v \in S$ $d(v)$ is the shortest path from u to v.

Proof. By induction on $|S|$.

- **Base Case.** Easy, $|S| = 1$ implies $S = \{u\}$, and the distance is 0. Alternatively $|S| = 0$ is trivial.
- **Induction Hypothesis.** Assume the theorem is true for $|S| = k \geq 1$.
- **Inductive Step.**
  - Let v be the next vertex added to S, and w be the neighbouring path vertex according to the algorithm.
  - By the assumption, $d(w)$ is minimum.
  - Let P, for contradiction, be the shortest path from u to v.
  - Let xy be the first edge in P that leaves S.
  - Then the length of P is at least $d(x) + w(x, y)$, which is the tenative distance to y.
  - This distance can't be smaller than the tentative distance to v, otherwise we would've chosen it instead of v to add next.
  - Therefore the length of P is at least $d(v)$. □

Dijkstra's Algorithm

- We can keep an array of predecessors that gets updated when we update the distance – this will give us the shortest path tree.
- Where was the dynamic programming?
  - When we select whether to update the distance or keep the existing one.
  - If $dist[v]$ is the distance from u to v, then the core of the algorithm is repeatedly performing $dist[v] = \min\{dist[v], dist[current] + w(current, v)\}$.
- Note there's also a greedy selection of the next vertex to process.
- Where's the problem with negative edges?

Dealing with Negative Edges - Bellman-Ford Algorithm

- Dijkstra's algorithm ran into trouble with negative edges because we never look at marked vertices again.
- The Bellman-Ford algorithm can handle negative edges – it just looks at everything again!
- Doesn't work with negative *cycles* though – but it can detect them, and they pose a semantic problem anyway.
- Historical note: Both algorithms were invented at rougly the same time, Dijkstra's in 1956 (but not published until 1959), Bellman-(Moore)-Ford in '58, '57 and '56 respectively.

Bellman-Ford Algorithm

1. Set things up as with Dijkstra's.
2. For 1 to the number of vertices
    1. For each edge uv
        1. If using uv improves the distances, use it and update as needed.
3. Check for negative cycles:
    1. For each edge uv
        1. If uv improves the distances, there's a negative cycle.

Bellman-Ford Algorithm - Correctness

Formally, we need a slightly more precise statement:

Lemma

After i iterations of the outer for loop:

- *If $d(v) \neq \infty$, there is a (known) path from the start vertex u to vertex v with distance $d(v)$.*
- *If there is a path from u to v with at most i edges, $d(v)$ is the length of the shortest path with at most i edges.*

Bellman-Ford Algorithm - Correctness

Proof. By induction on i:

- **Base Case:** $i = 0$, nothing has happened yet, trivial!
- **Inductive Assumption:** Assume that at step i we have the shortest paths with at most i edges.
- **Inductive Step:**
  - (First part) When $d(v)$ is updated to $d(r) + w(r, v)$, there is a path from u to r to v with weight $d(v)$.
  - (Second part) Before updating, we have by assumption the distances of all the shortests paths with at most i edges computed (if they exist).
  - Then at each possible update, we compare the shortest path to a neighbour plus an edge, with the shortest path on at most i vertices, and keep the minimum.
  - So we have a path on $i + 1$ edges that is shorter than the i edge path (and we pick the smallest of all these), or we keep the i edge path – thus the path must be the shortest path on $i + 1$ edges – there are no other possible ways to get it. □

Dijkstra's vs. Bellman-Ford – Complexity

- Dijkstra's:
  - $O(n^2)$ time.
  - $O(m + n \log n)$ time if we use a priority queue to help pick the next vertex.
  - $O(n)$ space.
- Bellman-Ford:
  - $O(nm)$ time – can't really get around this.
  - $O(n)$ space.

Connectivity

Connectivity of Graphs

- When using graphs to model things, we usually want to know if they're all in one piece.
- We've used the word "connected" before.
- A graph is *connected* if there's a way to get from any vertex to any other vertex via the edges.
- A pretty useful property if you're running a computer network, or transport network...

Testing Connectivity

- Testing this property is actually easy.
- Just do a traversal!
- If we visit all the vertices, the graph must be connected. If we don't, it's not.

What if it's not connected?

- If it turns out that the graph is not connected, we might be interested in finding out what bits (subgraphs) are.
- These are called *connected components*.
- Finding them is almost as easy as testing connectivity:
  1. While not all vertices have been visited:
     1. Pick an unvisited vertex.
     2. Start a traversal there.
     3. All the vertices you can reach from there form a component, record this and mark them as visisted.

Strongly Connected Components

In directed graphs, things get more complicated:

- In an undirected graph, if a is adjacent to b then b is also adjacent to a.
- Not necessarily true in directed graphs.
- In undirected graphs, every vertex in a connected component is reachable from every other vertex in the same component.
- Not necessarily true in directed graphs.

Strongly Connected Components

The directed graph analogy to this reachability based connectivity is *strong connectivity*.

- Two vertices are *strongly connected* if there is a path from one to the other, and a path back again.
- This gives a *strongly connected component*: a subgraph where every pair of vertices is strongly connected.

How do we find these?

Strongly Connected Components

We can just run a bunch of reachability queries.

- Parallelises well.
- $O(n^2(n+m))$ without clever optimisations (keeping track of all the vertices visited in the query).
- Can get it to $O((n+m)\log n)$ expected sequential time if we make the approach complicated enough (uses divide and conquer).

Strongly Connected Components

We can do it faster overall (but without the boost from parallelism) using *Tarjan's Algorithm*.

- Depth first traversal based.
- Each vertex is assigned an *index* and a *low* value.
  - *index* keeps track of where it is in the traversal.
  - *low* tracks the lowest index reachable from this vertex.
- If *low* is less than *index*, there's a path back to something earlier in the traversal (and of course one forward - the traversal itself), so all those vertices must be in a strongly connected component together.

Tarjan's SSC Algorithm 1

```
set<set<vertex>> tarjan(directed_graph<vertex> d) {
  int index = 0;
  stack s;
  for (each vertex v in d) {
    if (index[v] is undefined) {
      components.add(strongconnect(v));
    }
  }
}
```

Tarjan's SSC Algorithm 2

```
set<vertex> strongconnect(vertex v) {
    index[v] = index; low[v] = index; index = index + 1;
    s.push(v); on_stack[v] = true;

    for (each edge (v, w) in d) {
      if (index[w] is undefined) {
        strongconnect(w);
        low[v]  = min(low[v], low[w])
      } else if (w.onStack){
        low[v]  = min(low[v], index[w])
      }
    }

    if (low[v] = index[v]) {
      set<vertex> new_component;
      do {
        w = s.pop(); on_stack[w] = false;
        new_component.add(w);
      } while (w != v)
      return new_component;
    }
}
```

Complexity

- Time: $O(n + m)$.
- Space: $O(n)$ (what extra stuff was kept track of in the algorithm?).

Finding Articulation Points

Back in undirected land, a similar algorithm can be used to find the equivalent of strongly connected components.

- Which vertices (or edges) can we remove that will increase the number of connected components?
- These vertices are called *articulation* points (the edges are called *bridges*.
- A connected component that has no articulation points is called *biconnected*.

Finding Articulation Points

- We can find articulation points using an augmented depth first traversal.
  - We need to keep track of the depth of a vertex in the traversal, and
  - the lowest depths of the neighbours of each vertex and all its descendants in the DFS tree (the "lowpoint").
- If a vertex's depth is less than or equal to the lowpoint of any of its neighbours, it's an articulation point!
- The lowpoint is actually just a way of tracking whether there's at least two ways to get to a vertex – vertices in biconnected components all have the same lowpoint.
- The root vertex is a special case – if it has at least two children, it's an articulation point.

```
FindArticulationPoints(i, d)
  visited[i] = true, depth[i] = d
  low[i] = d, child_count = 0
  is_articulation = false
  for (each neighbour n of i)
      if (not visited[n])
        parent[n] = i
        FindArticulationPoints(n, d+1)
        child_count = child_count + 1
        if (low[n] >= depth[i])
            is_articulation = true
        low[i] = min{low[i], low[n]}
      else if (n != parent[i])
        low[i] = min{low[i], depth[n]}
  if ((i has a parent and is_articulation) or
     (i has no parent and child_count > 1))
    i is an articulation point
```

Complexity

- Time: $O(n + m)$ – it's just a depth-first traversal.
- Space: $O(n)$
- Where n and m are the number of vertices and edges respectively.
- Can be modified to retrieve the biconnected components with the same complexity.
- Can be done in parallel in $O(\log n)$ time! (But with $O(n + m)$ processors...)

Dependencies in Directed Graphs

DAGs

If a directed graph has no directed cycles, we called it a *Directed Acyclic Graph*, or DAG.

How many strongly connected components does this have?

DAGs

DAGs are like the trees of the directed graph world, and as such have many applications:

- Anything where dependencies are important (anyone running a *NIX distro will understand this one).
- Task scheduling.
- Compilation.
- Computing information in a spreadsheet.
- Modelling multi-stage processes.

For all of these, it's pretty clear we need to know what order to do things in.

This ordering is called a *topological ordering*. It may not be unique. (What if it is?)

To find a topological ordering, we perform a *topological sort*.

Kahn's Algorithm

```
list<vertex> topo_order;
set<vertex> s = {all vertices with no incoming edges};
while (!s.empty()) {
    v = s.remove_something_from_a_set();
    topo_order.append(v);
    for (each neighbour u of v) {
        remove (v,u) from graph;
        if (u has no incoming edges) add u to s;
}
if (any edges left in graph) return error;
else return topo_order;
```

Depth First Topological Sort

```
list<vertex> topo_order;

while (not all vertices permanently marked) {
    v = next unmarked vertex;
    visit(v)
}

visit(vertex v) {
    if (v is permanently marked) return;
    if (v is temporarily marked) return error;
    mark v temporarily;
    for (each (v,u) in edges) {
        visit(u);
    }
    clear temporary mark from v;
    permanently mark v;
    topo_order.append_front(v);
}
```

- Both linear time algorithms – $O(n + m)$.
- Both use $O(n)$ extra space (can do some tricks here to get $O(1)$ for Kahn's algorithm at the cost of increasing time to $O((n + m) \log n)$.
- Can find a topological (a different way) in $O(\log^2 n)$ time ... if we have a polynomial number of processors.