31251 – Data Structures and Algorithms

Week 5 - Graphs Part I

Luke Mathieson

Does anyone read these titles?

- Graphs
- Traversing Graphs
- Greedy Algorithms
    - Examples: Prim's Algorithm, Kruskal's Algorithm

Graphs

Graphs as Mathematical Objects

- Graphs are an incredibly useful modelling tool.
- Simple graphs consist of:
  - A set of elements called *vertices*.
  - A set of unordered pairs of distinct vertices called *edges*.
  - There is only one edge between a pair of vertices.
- Other types of graph come from altering these conditions:
  - Ordered pairs gives *directed* graphs.
  - More than one edge per pair gives *multi-graphs*.
  - More than two vertices per edge gives *hypergraphs*.
  - Edges (and vertices) can be weighted, labelled, &c.

Graphs as Mathematical Objects

- The vertices model interesting things:
  - Computers
  - Processes
  - Proteins
  - Production facilities
  - Websites
- The edges model relationships between interesting things:
  - Network links
  - Shared resources
  - Protein interactions
  - Transport links
  - Hyper links
- Used somehow in virtually every part of computer science.

Some Notation and Definitions

- If two vertices have an edge between them, they are *adjacent*.
- If a vertex is one of the pair that forms an edge, it is *incident* to that edge.
- The number of edges incident to a vertex is the *degree* of that vertex.
- Graphs will be denoted with uppercase letters like G, H, &c.
- Vertices will be denoted by lowercase letters like u, v, &c., *or* natural numbers $1, 2, 3, 4, \ldots, n$.
- Edges will be denoted by lowercase letters like e, d, &c. or by their incident vertices:
  - In undirected graphs uv.
  - In directed graphs (u, v) – u is the *tail*, v is the *head*.
  - Directed edges are also called *arcs*.

Graphs as Data Structures

- As an abstract data structure, a graph needs to support a lot of basic operations:
  - `addVertex(Vertex v)`
  - `removeVertex(Vertex v)`
  - `addEdge(Vertex u, Vertex v)`
  - `removeEdge(Vertex u, Vertex v)`
  - `adjacent(Vertex u, Vertex v)`
  - `degree(Vertex u)`
  - return the vertices in the graph
  - return the edges incident to a vertex
  - return the vertices adjacent to a vertex
- As well as the normal things needed for a usable implementation in a given language (constructors, &c.)
- Some operations may depend on the exact implementation.

Graphs as Data Structures – Adjacency Matrix

- Simplest form:
  - Edges are stored as a two-dimensional matrix (e.g. `vector<vector<bool> > edges` or `bool edges[][]`).
  - `edge[i][j] == true` means vertex *i* is adjacent to *j*.
- Some enhancements:
  - Can use a numeric (`int`, `double`, `...`) matrix to give weighted edges.
  - Can use a matrix of `Edges` if edges are more complex – a `null`-type value means no edge.
  - Easily supports directed graphs and undirected graphs.
  - If vertices have associated data, we can store them separately (another array would make matching indices easy).
- Quick access – $O(1)$, not so great space and set-up – $O(n^2)$ (with *n* vertices).

Graphs as Data Structures – Adjacency List

- Each vertex has associated with it a list of its adjacent vertices.
- Could be a size n array of linked lists, or similar.
- Slower to determine adjacency – $O(n)$, but faster to return all adjacent vertices – $O(1)$.
- Most compact space representation $O(m + n)$ where m is the number of edges in the graph – we have to store something for each vertex and edge anyway, so this is the best we can do.
- Easy to modify for more complex edge and vertex data structures.
- Works best for sparse graphs (few edges per vertex).

Graphs as Data Structures – Object Oriented

The extreme version:

- We have classes for `Vertex`, `Edge` and `Graph`.
- `Vertex` contains a list of its `Edges`.
- Each `Edge` knows its endpoints.
- the `Graph` knows about everything.
- *Lots* of references to keep track of.
- Tends to be the slow way to do things, but has a nice match to the conceptual version.

Traversing Graphs

Moving About in Graphs

- Many algorithms rely on being able to explore the graph.
- We need to be able to keep track of which vertices we've been to.
- We also need a way of picking which neighbour to move to next:
    - We can use an inherent order on the vertices (by label, number, &c.), or pick an arbitrary order.

Depth First Traversal

- Pick a starting vertex.
- Recursively pick an unvisited neighbour and visit it.
- Can be implemented recursively, or iteratively using a stack.

Depth First Traversal – Recursive

```
dft(Vertex v){
  mark v as visited;
  visit(v);
  for each neighbour u of v{
    if (u is unmarked)
      dft(u);
  }
}
```

Depth First Traversal – Iterative

```
dft(){
  pick starting vertex v;
  Stack unprocessed = new Stack();
  unprocessed.push(v);

  while (!unprocessed.isEmpty()){
    Vertex u = unprocessed.pop();
    if (u is unmarked){
      visit(u);
      mark u;
      for each neighbour w of u{
        unprocessed.push(w);
      }
    }
  }
}
```

Breadth First Traversal

- Pick a starting vertex and put it in a queue.
- Iteratively take a vertex from the queue, visit it and place all its neighbours in the queue.
- It's inherently iterative, it's not impossible to implement recursively, just silly.

Breadth First Traversal – Iterative

```
bft(){
  pick starting vertex v;
  Queue unprocessed = new Queue();
  unprocessed.offer(v);

  while (!unprocessed.isEmpty()){
    Vertex u = unprocessed.poll();
    if (u is unmarked){
      visit(u);
      mark u;
      for each neighbour w of u{
        unprocessed.offer(w);
      }
    }
  }
}
```

Breadth First vs. Depth First

- Some graphs produce the same traversal order for both.
- Which one to use will depend upon the application.
- Notice the iterative versions of both are actually identical – just swap the stack and the queue.
- Both $O(n + m)$. (Why?)

Greedy Algorithms

Greedy is ~~good~~cheap

- Greedy Algorithms are based on picking what looks good now.
- One of the simplest algorithmic paradigms.
- Usually easy to implement.
- Only really works for certain types of problems.
- For some problems, a greedy approach may produce the worst solution!

When do Greedy Algorithms work?

- Greedy Algorithms usually work when the problem satisfies two properties:
    - **Optimal Substructure:** Optimal solutions contain optimal subsolutions to subproblems (you can split solutions up and they're still good).
    - **Greedy Choice Property:** Any decisions depend only on what you've already seen (you don't have to come back and fix things).

Spanning Trees of Graphs

A Motivating Problem

Consider the following problem:

- A company has to connect cities with fibre optic cable such that each city has a (possibly multi-hop) link to every other city. The company knows the cost of linking each pair of cities, and wants to accomplish its task with a minimum cost.
- We can use a weighted graph to model this problem, but what are we looking for?
  - A set of edges that connects all the vertices.
  - No unneeded edges.
  - It's a thing called a tree! (See next slide)
- So we want a tree that includes all the vertices, and has the minimum total edge weight.

Spanning Trees

- A *subgraph* is a subset of the vertices and edges of a graph that form a graph.
- A subgraph is *spanning* if it includes all the vertices of the original graph.
- A spanning subgraph is a *spanning tree* if it contains no cycles.
- A spanning tree is a *minimum spanning tree* if it has minimum total (edge) weight over all possible spanning trees of that graph (is it unique?).

Spanning Trees

- In unweighted graphs (or a graph where all edge weights are the same), *any* spanning tree is a minimum spanning tree.
- We can compute one from a depth-first or breadth-first traversal.

```
df_spanning_tree(Vertex v, Tree t){
  mark v as visited;
  for each neighbour u of v{
    if (u is unmarked)
      add edge vu to t;
      dft(u,t);
  }
}
```

Weighted Graphs

- If we have different weights on the edges, a simple traversal is not enough.
- There are two main algorithms:
  - Prim's Algorithm
  - Kruskal's Algorithm
- If we have time, we'll look at Borůvka's (Sollin's) Algorithm.
- These are all greedy algorithms, with similar but slightly different approaches.

Prim's Algorithm

- Given a *connected* graph G

```
1. Pick a starting vertex $v$ (however you want), add $v$ to the
   partially complete tree $T$.
2. While $|T| < n$
    1. Let $E'$ be the set of edges $uv$ where $u \in T$ and $v \in G \setminus T$.
    2. Let $uv$ be the edge of smallest weight in $E'$.
    3. Add $uv$ to the edges of $T$ and $v$ to the vertices of $T$.
3. Return $T$.
```

Prim's Algorithm

- In other words:
- Keep track of which edges and vertices are in the tree.
- Pick the next smallest edge that extends the tree, add it.
- Keep going until the whole graph is spanned.

Prim's Algorithm Complexity

- If we use an adjacency matrix, and search for the edges: $O(n^2)$.
- Putting the edges into a binary heap, with the graph stored as an adjacent list: $O((n+m)\log n) = O(m \log n)$.
- Using Fibonacci heap (don't worry about what this is) and adjacency lists: $O(m + n\log n)$.

Kruskal's Algorithm

- Prim's algorithm grows the spanning tree by greedily picking the best next edge.
- Kruskal's algorithm approaches the problem more globally - start with a lot of trees (a *forest*), and pick the best edge to connect two components.

Kruskal's Algorithm

- Given a *connected* graph G, start with n trees $\{T_i\}$, each with one vertex.
  1. While there is more than one tree
     1. Pick the smallest edge uv such that u is in one tree T_i, and v is in another T_i.
     2. Merge T_i and T_j by adding uv.
  2. Return the final tree T.

Kruskal's Algorithm Complexity

- By labelling vertices with which component they're in, we can get $O(n \cdot m)$ – not great.
- If we sorted the edges, and employ an efficient disjoint set data structure (haven't seen one in the course): $O(m \log m) = O(m \log n)$ – about the same as the normal Prim implementation.
- If the edges can be sorted efficiently by counting sort or radix sort or similar, we can get $O(m \cdot \alpha(n))$, where $\alpha(n)$ is the inverse of the single valued Ackermann function (look it up some time).

Extra Stuff on Spanning Trees

Borůvka's Algorithm

- Invented in 1926 by Otakar Borůvka – see computers aren't necessary for *computer science*.
- Reinvented three more times, lastly by Sollin in 1965.
- Works like a cross between Prim's and Kruskal's algorithms

Borůvka's Algorithm

- Given a graph G, initialise n tree $\{T_i\}$, each containing one vertex.

1. While there is more than one tree
    1. For each component tree
        1. Pick the smallest outgoing edge (connecting this component to another)
        2. Add this edge to the trees, merging them.
2. Return the single remaining tree T

Borůvka's Algorithm – Complexity

- The outer loop only needs to execute $O(\log n)$ times – we halve the number of components at each step.
- Along with search for the edges at each iteration, we get $O(m \log n)$ without too much fiddling.
- A similar approach as used for Kruskal's can be used to get $O(m \cdot \alpha(n))$.
- A randomised version exists with $O(m)$ expected running time – remember this is linear in the size of the graph, about as fast as possible.

Correctness of Prim's Algorithm I

Lemma

Given a connected, weighted graph G, Prim's algorithm produces a minimum spanning tree of G.

Proof:

- As G is connected, it is (or at least should be) clear that Prim's algorithm produces a tree that spans the graph. Thus we need only argue that it is a *minimum* spanning tree.
- Let T_P be the tree produced by Prim's algorithm.
- Assume for contradiction that there exists a minimum spanning tree T_M og G and that the weight of T_M is less than the weight of T_P.
- Let e be the first edge added to T_P that is not in T_M, and let $S \subset V$ be the vertices in the partial tree at the point e is added.
- Note that e has one endpoint in S and the other in $V \setminus S$.

Correctness of Prim's Algorithm II

- As T_M is a spanning tree, there must be a path in the tree between the endpoints of e.
- On this path there must be some edge f in T_M with one endpoint in S and the other not.
- Then at the point of adding e, f must've been a condidate edge, that the algorithm didn't pick, hence the weight of f is at least the weight of e.
- If the weight of f is strictly greater than that of e, we can construct a new tree $T_{M'} = T_M - f + e$ with smaller weight that T_M, contradicting the assumption that T_M is minimum.
- If the weight of f is the same as that of e, we can construct $T_{M'}$, then repeat the argument with $T_{M'}$ in place of T_M – either we get a contradiction as before, or we progressively modify T_M to be T_P and therefore T_P must also be minimum.

Correctness of Kruskal's Algorithm I

Lemma

Given a connected, weighted graph G, Kruskal's algorithm produces a minimum spanning tree of G.

Proof:

- This time we use an inductive proof.
- What we will show is that if F is the set of edges chosen at any point in the algorithm, then there is some minimum spanning tree that contains F.
  1. **Base Case:**
     - $F = \emptyset$. The proposition is trivially true in this case as $\emptyset$ is a subset of any set.
  2. **Inductive Assumption:**
     - Assume the algorithm (to this point) has produced edge set F', and F' can be extended to some MST T.
  3. **Inductive Step:**

Correctness of Kruskal's Algorithm II

  - If the next chosen edge e is also in T, then our proposition holds for $F + e$.
  - If it is not, then $T + e$ contains a cycle, and there is some edge f that is in the cycle, but not in F.
  - The weight of f must be at least the weight of e – otherwise the algorithm would choose f at this point.
  - Then $T - f + e$ is a minimum spanning tree that contains $F + e$, and we're done.
- Then by induction the final set of edges can be 'extended' to an MST – as it spans the graph, this extension is 'do nothing', so the algorithm produces an MST.

Some Properties of MSTs

Lemma

If e is the unique smallest weight edge in a connected, weighted graph G, then e is in every MST of G.

Proof: Assume for contradiction there is some MST T that does not contain e, then the graph $T + e$ contains a cycle, we can pick any edge from this cycle (other than e) and remove it to obtain a new spanning tree. As e has weight less than every other edge, this new tree must have smaller weight that T, therefore T was not an MST.

What happens if there's more than one minimum weight edge – are they all in every MST?

Some Properties of MSTs

This can be inductively extended:

Lemma

If G is a connected, weighted graph where all edge weights are distinct, G has a unique minimum spanning tree.

And similar proofs give (assume G is connected):

Lemma

Let C be a cycle in G, and e be the largest weight edge in C. e is not in any MST of G.

Lemma

Let D be any cut in G, and e be the smallest weight edge in D. e is in every MST of G.

What about disconnected graphs?

- If the graph has several disconnected components, we can't get a spanning tree (trees have to be connected).
- We can get a spanning forest.
- The algorithms we have seen so far won't work (why not? can they be fixed?).

Reverse-Delete – Kruskal's Other Algorithm

- Appears in the same paper as Kruskal's algorithm.
- Sort of like a backwards Kruskal's.
- We remove edges, instead of adding them, and see what we're left with at the end.

Reverse-Delete – Kruskal's Other Algorithm

Given a weighted graph G:

1. Sort the edges by decreasing weight.
2. While edges remain to be processed
    1. Take the next biggest edge e.
    2. Check if deleting e will create more components than you already have.
    3. If not, delete it, otherwise keep it.
3. Return the remaining graph.

Reverse-Delete – Complexity

- We can sort the edges in $O(m \log m)$.
- We can check the connectivity in $O(\log n(\log \log n)^3)$.
- So in total we get $O(m \log n(\log \log n)^3)$ time.