CS3230 Chapter 7 - Graph Algorithms

Based on lectures by Chang Ee-Chien
Notes taken by Andrew Tan
AY18/19 Semester 1

1 Graph representations

There are several ways that we can represent graphs within a program:

Representation 1 - **Adjacency matrix**: each entry within the matrix is a 0 or a 1, and an edge from a to b is represented by filling in the entry at row a and column b with 1.
Representation 2 - **Adjacency list**: we use an array to deonte our list of vertices, and each entry x in the array points to a linked list that contains the vertices such that there exists an edge from x to that vertex.

If $|V| = n$ and $|E| = m$ where V is our number of vertices and E our number of edges, then our adjacency matrix will always have a space complexity of $O(n^2)$, while our adjacency list will have a space complexity of $O(m + n)$.

1.1 Graph definitions

We introduce a few graph definitions:

- An **Undirected and weighted graph** is a graph where each edge is associated with a weight. Formally, each edge is of the form $(\{u, v\}, w)$ where $u, v \in V$ and w is a real number. We can easily express this graph with an incidence matrix by simply modifying our adjacency matrix to have entries corresponding to the weights.
- An **Euclidean graph** is a weighted undirected graph where each vertex is a point in $\mathbb{R}^2$. The weight of the edge $\{v, w\}$ is the distance between v and w, i.e. $||v - w||_2$. This is essentially a complete graph, and an Euclidean graph has a space complexity of $O(n)$ as all the edges are implicitly defined.
- A **subgraph** $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- A **spanning tree** $T = (V', E')$ of $G = (V, E)$ is where T is a subgraph of G, and
  - T is a tree
  - $V' = V$
- A **minimum spanning tree** T of G is where T is the spanning tree of G with minimum weight. If the edges of a graph are unique, then the minimum spanning tree is also unique.
- A **shortest path** from v to w is the path with minimum weight.

2 Depth-First Search (DFS)

Depth-first search, along with Breadth-first search, is one of the simplest algorithms for searching a graph. With DFS, we simply search "deeper" in the graph whenever possible.

Whenever DFS discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute $v.\pi$ to u. We define the **predecessor subgraph** as such: we let $G_\pi = (V, E_\pi)$, where $E_\pi = \{(v,\pi,v) : v \in V \text{ and } v.\pi \neq NIL\}$. The predecessor subgraph of a DFS forms a depth-first forest comprising of several depth-first trees. The edges in E_π are tree edges.

DFS colors vertices during the search to inicate their state. Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely.

```
DFS($G$):
for each vertex $u \in G.V$:
    $u.color = WHITE$
    $u.\pi = NIL$
time = 0
for each vertex $u \in G.V$:
    if $u.color == WHITE$:
        DFS-VISIT($G, u$)
```

```
DFS-Visit($G, u$):
$u.color = GRAY$
for each $v \in G.Adj[u]$:
    if $v.color == WHITE$:
        $v.\pi = u$
        DFS-Visit($G, u$)
$u.color = BLACK$
```

The running time of the loops in DFS is in $\Theta(V)$ (exclusive of the time to execute calls to DFS-Visit). During an execution of DFS-Visit(G, v), the loop in DFS-Visit executes $|Adj[v]|$ times. Since $\sum\limits_{v \in V} |Adj[v]| = \Theta(E)$, the total code of executing DFS-Visit is $\Theta(E)$. Thus, the running time of DFS is therefore $\Theta(V + E)$.

3 Minimum spanning trees

There are two key greedy algorithms for finding minimum spanning trees (MST): Kruskal's algorithm, and Prim's algorithm. We shall explore both of them, while also examining how the different graph representations affect their running time.

3.1 Kruskal's Algorithm

We shall sketch Kruskal's algorithm as such:

1. Let T be the empty set
2. Repeat
   (a) Choose the minimum weighted edge e such that $T + \{e\}$ does not contain a cycle
   (b) Insert e into T.

   Until T is a spanning tree.

Let $G = (V, E)$ be a connected, weighted, and undirected graph, let T be any promising set of edges, and let e be the minimum weighted edge such that $T + \{e\}$ does not contain a cycle.

For simplicity, we'll assume the weights of edges are distinct. The proof can easily be extended towards the general case. Let U be the minimum spanning tree of G such that $T \subset U$. If $e \in U$, then there is nothing to prove. Otherwise, $U + \{e\}$ will contain exactly one cycle. In this cycle, there is another edge e' in U that is not in T. Consider the subgraph $U + \{e\} - \{e'\}$. This subgraph U' is a spanning tree. Since $T \subset U$, $T + \{e'\}$ does not contain a cycle.

By definition, U is a MST and thus the weight of U' must be larger or equal to that of U. This implies that the weight of e' is smaller or equal to the weight of e. However, since the weights are distinct, the weight of e' must be smaller than the weight of e. This contradicts the fact that e is the minimum weighted edge that doesn't create a cycle.

Thus, we must have $e \in U$, and thus $T + \{e\}$ is promising.

3.1.1 Implementations

DFS:
We can use a DFS to check if $\{e\} + T$ contains a cycle, before inserting it into T.

1. Let T be the empty set
2. Sort the set of edges in increasing order and store them in an array E.
3. Initialize *count* to 0
4. Repeat
    (a) $count = count + 1$
    (b) $e = E[count]$
    (c) Using DFS, check if $\{e\} + T$ contains a cycle. If there is no cycle, insert e into T.

   Until $|T| == n - 1$ where n is the number of vertices in the graph.

Sorting will simply take $|E|log|E|$. Here, the running time of DFS is in $\Theta(|V|)$, since the number of edges in a tree can't be more than $|V| - 1$. In the worst case, the number of DFS calls is in $\Theta(|E|)$, so the running time is in $\Theta(|E||V|)$.

Disjoint Set:
A disjoint set data structure maintains a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$. Each set $S_i = \{a_1, a_2, \ldots, a_m\}$ is identified by a representative which can be any element in the set.

The data structure supports three operations:

1. Create_set(x): Creates a set $\{x\}$.
2. Find_set(x): Finds the set containing x and returns the representative of this set.
3. Union(x, y): Merges the two sets which contain x and y respectively and returns the representative of this combined set.

We can implement this with a data-structure known as union by rank with path compression.

Back to Kruskal's algorithm, we can use our disjoint set to implement this algorithm:

1. Let T be the empty set
2. Sort the set of edges in increasing order and store them in an array E.
3. For each $v \in V$, create a set $\{v\}$.
4. Initialize *count* to 0
5. Repeat
    (a) $count = count + 1$
    (b) $\{v, u\} = E[count]$
    (c) Find the set containing v, and the set containing u. If v and u are not in the same set, then insert $\{v, u\}$ into T, and merge the two sets containing v and u.

   Until $|T| == n$ where n is the number of vertices in the graph.

With our disjoint set, the running time of m operations on n elements is in $O(m\alpha(n))$, where n is the number of elements in the data-structure, and $\alpha(n)$ is an extremely slow-growing function ($\alpha(n) \approx log^*n$).

Using this, our running time is improved accordingly, with sorting taking $\Theta(|E|log|E|)$, and our number of set operations are less than $3|E|$. Since $3|E|\alpha(|V|) \in O(|E|log|E|)$, our runnning time of Kruskal's algorithm is now $\Theta(|E|log|E|)$.

3.2 Prim's Algorithm

We shall sketch Prim's algorithm as such:

1. Let T be the empty set
2. Let B be the set v_0 where v_0 can be any vertex.
3. Repeat
    (a) Choose the minimum weighted edge $e = \{u, v\}$ such that $u \in B$ and $\{e\} + T$ is a tree.
    (b) Insert e into T.
    (c) Insert v into B.

    Until T is a spanning tree.

Let $G = (V, E)$ be a connected, weighted, and undirected graph, let T be any promising set of edges and T is connected, and let e be the shortest edge such that $T + \{e\}$ is a connected tree.

We'll continue to assume that the weights of edges are distinct. The proof can be extended easily to the general case. Let U be the MST of G. Thus we have $T \subset U$. Let B be the vertices that are in T. Suppose an edge has one vertex in B, we say that the edge touches B. If $e \in U$, then there is nothing to prove. Otherwise, $U + \{e\}$ will contain exactly one cycle. In this cycle, there is another edge e' that is not in T, and e' touches B. Since e' touches B, the subgraph $T + \{e'\}$ is a tree. Consider the subgraph $U + \{e\} - \{e'\}$. This subgraph U' is a spanning tree.

Since U is the MST, the weight of U' must be larger or equal to that of U. This implies that the weight of e' is smaller than the weight of e. This contradicts the fact that e is the smallest edge that touches B.

Thus, we must have $e \in U$, and thus $T + \{e\}$ is promising.

3.2.1 Implementation

Storing edges in a heap:
A sketch of the algorithm is as follows:

1. Let T be the empty set
2. Let H be an empty heap
3. Let B be the set v, where v is any vertex in V
4. For each u adjacent to v, insert the weight of v, u into the heap (the directed edge $(v, u\ )$ is stored as the data)
5. Repeat
    (a) Extract the minimum weight from the heap. Let (p, q) be the edge associated with the minimum weight (Note that p is guaranteed to be in the set B)
    (b) If $q \notin B$,
    then insert q into B, insert $\{p, q\}$ into T, and for each u which is adjacent to q and $u \notin B$, insert the weight of $\{q, u\}$ into the heap. The associated data is the directed edge (q, u)

    Until $|B| == |V|$

There are at most $|E|$ calls of extract_min and insertion, hence our running time is $O(|V| + |E|log|E|)$.

Storing vertices in a heap:
We can achieve a smaller time complexity by storing the vertices rather than the edges in the heap. We will need an additional operation with the heap: reduce_key(A, k, x), where A is the array storing the heap, k is the indice of a vertex in the array, and x is the value we wish to reduce $A[k]$ to. The reduced value must be smaller or equal to the original value, as we shall reheapify our heap with a call to SiftUp(A, k).

Below is a sketch of the algorithm:

```
1. Let $T$ be the empty set
2. Let $H$ be an empty heap
3. For each $v \in V$, let $k[v] = \infty$, and let $n[v] = \{v, v\}$
    (a) Find the vertex $p$ with the minimum key
    (b) Insert $p$ into $B$.
    (c) Insert $n[p]$ into $T$.
    (d) For each $u$ such that $\{p, u\} \in E$, if the weight of $p, u$ is less than $k[u]$, then reduce $k[u]$
        to the weight of $\{p, u\}$, and let $n[u] = \{p, u\}$.
   Until $|B| == |V|$
```

We call extract_min a total of $|V|$ times, and reduce_key at most $|E|$ times. Thus, our running time is in $O(|V|log|V| + |E|log|V|) = O(|E|log|V|)$, which is the same as our disjoint set implementation of Kruskal's algorithm (since the graph is connected, $log|E| \in \Theta(log|V|)$).

We can in fact improve our vertices implementation of Prim's algorithm using Fibonacci Heaps. A Fibbonacci heap supports the following operations:

- extract_min: $O(log(n))$ time
- insert: $\Theta(1)$ amortized time
- reduce_key: $\Theta(1)$ amortized time

The discussion on amortized time is in a later chapter, but for now it can be thought of as roughly the average time of each operation in a sequence of operations.

Using the Fibonacci heaps, our vertices implementation for Prim's algorithm runs in $O(|V|log|V| + |E|)$ time.