CS5110 - Final Project

Hermès Henry & Daniel Murrow

Fall 2024

GitHub Repository: https://github.com/MidnightHermes/CS5110-Final-Project/

1 Prim's Algorithm

Prim's Algorithm is an algorithm that defines a method for finding the Minimum Spanning Tree (MST) of an undirected connected graph. The algorithm can be simplified into the following pseudocode at a very high level of abstraction and should output an MST that has V vertices and $V - 1$ edges.

High-Level Pseudocode

1. Initialize the MST based on an arbitrarily chosen vertex in the graph
2. Consider all edges from the graph **not** in the MST that connect to the vertices in the MST
    - ⋆ To avoid cycles, we make sure to check if the edge we are considering connects to a vertex that is not already connected to by another edge; i.e., only connect to a vertex that is in the graph but not in the MST.
3. Select the edge with the lowest weight and add the connected vertex to the MST
4. Repeat Step 2. and Step 3. until the MST contains every vertex from the original graph

Time Complexity & Data Structure

A simple implementation of this would initialize lists of edges and vertices; For each of the vertices, V, we look at each edge, E, connected to it to find the minimum, leading to a time complexity of $O(|V|^2)$. We can improve this by using a priority queue instead of a list to keep track of our edges, E, and vertices, V. With a Priority Queue, keeping in mind that each edge is only ever processed once, we see a total time complexity of $O(|E| \log |E|)$ where $|E|$ represents the cardinality of the set of edges in the input graph. We can further improve the runtime of this algorithm by keeping track of our vertices, V, instead, ordered by the smallest edge-weight connecting them to a vertex already in the MST; this method leads to a total time complexity of $O(|E| \log |V|)$ which asymptotically becomes an improvement as the input graph becomes complete, specifically a graph where $|E| = \frac{|V|(|V|-1)}{2}$.

In order to implement the Priority Queue, we used the Python standard library's `heapq` heap queue algorithm. The important functions are `heapq.heappush(heap, item)` and `heapq.heappop(heap)` which are both $O(\log n)$. This specific heap queue is a binary min-heap; as such, it orders values from smallest to largest (i.e., `heapq.heappop(heap)` returns the smallest value in the heap), as opposed to a max-heap which would order values from largest to smallest. Using a Fibonacci heap instead of a binary heap could further increase the time complexity of this algorithm because the `insert` or `push` operation is $O(1)$ in a Fibonacci heap as opposed to $O(\log n)$ for a binary heap. Therefore, a Fibonacci heap combined with the method of keeping track of vertices instead of edges, as discussed in the previous paragraph, would have a total runtime of $O(|E| + |V| \log |V|)$.

In both cases for speeding up the algorithm, the speed-up is only significant as $|E|$ increases from $|V|$ up to a maximum of $|E| = \frac{|V|(|V|-1)}{2}$. That is, the closer $|E|$ is to $|V|$ the more negligible the speed-up from either or both methods will be.

Discussion

During our presentation of this implementation, we discussed whether it is possible to modify this implementation to detect all possible MSTs. The PriorityQueue data structure used reconciles a tie for minimum value based on which item was added first; that is, if we push items A and B to pq, each with a value of 1, then A would be popped first as it was pushed first, followed by B.

$$pq \leftarrow A \qquad pq \leftarrow B \qquad pq\texttt{.pop()} \leftarrow A$$

It could be said that the necessary modification would be to make the algorithm recursive, such that at each tie in the PriorityQueue, we recursively run the algorithm on n diverging branches where n is the number of edges tied for minimum weight. Each diverging branch would choose one distinct edge from the set of tied edges, then continue. This would explore all possible combinations of MSTs, however it would **severely** impact the runtime of the algorithm for large graphs or graphs with a high concentration of ties.

Low-Level Pseudocode

With all this in mind, we can put together a pseudocode with a much lower level of abstraction that we will call `Prims(`G`)` where G represents the input graph. This final pseudocode should have a worst-case time complexity of $O(|E| \log |E|)$ when implemented in Python.

Algorithm 1: `Prims(`G`)`

Input: Undirected Connected Graph G
Output: Minimum Spanning Tree

```
// Step 1.
start ← randomVertex(G)
MST ← new Graph(start)
// Initialize the PriorityQueue
PQ ← new PriorityQueue()
for each edge e in start.edges do
    PQ.push(e)
// Repeat Steps 2 and 3 until the queue is empty
while length(PQ) do
    edge = PQ.pop()
    // Make sure edge is not connected to a vertex alreading in MST
    if edge.link_vertex not in mst.vertices then
        mst.add_vertex(edge.link_vertex)
        mst.add_edge(edge)
        // Add every edge of this new vertex to the priority queue
        for edge_new in edge.link_vertex.edges do
            if edge_new not in mst.vertices then
                PQ.push(edge_new)
// Finally, return the MST once the while loop breaks
return MST
```

Testing

In order to test my implementation of the algorithm, we set a few simple criteria:

1. Does the MST contain the same number of vertices as the graph?
2. Does the MST contain the minimum possible number ($|V| - 1$) of edges?
3. Is the MST a connected graph?
4. Is the sum of all the weights in the MST equal to the sum of all the weights in `networkx`'s built-in MST function?

2 Maximum Clique Approximation

Algorithm 2: `Ramsey(`$\mathcal{G}$`)`

Input: Undirected Graph $\mathcal{G} = (V, E)$
Output: The approximate Max Clique in $\mathcal{G}$
Output: The approximate Max Independent Set in $\mathcal{G}$

if $V = \emptyset$ **then**
  return $\emptyset, \emptyset$
$v \leftarrow$ random member of V
$(C_1, I_1) \leftarrow$ `Ramsey`$(N(v))$
$(C_2, I_2) \leftarrow$ `Ramsey`$(\overline{N}(v))$
return (larger of $(C_1 \cup \{v\}, C_2)$, larger of $(I_1, I_2 \cup \{v\})$)

The maximum clique problem is an NP-Complete problem relevant to the fields of community detection, bioinformatics, and computational chemistry. The problem posed is to find the largest complete subgraph in a given undirected graph. No polynomial-time algorithm has been found to find the true largest clique yet. But there exist many algorithms designed to find an approximation of the largest clique in a graph. One of these is the RAMSEY algorithm seen above [1].

Analysis

The Ramsey algorithm works by stochastically choosing a node v to query on, and then recursing on the subgraphs induced by the neighbors and non-neighbors of v, respectively. Then the clique and independent sets are built by choosing the largest of the sets found so far in that recursion. The choice of query node can greatly alter the outgoing result of the algorithm. For example, if a node is chosen that is adjacent to half the nodes in the largest clique in the graph, then the other half will be non-neighbors and the true maximum clique is partitioned by the algorithm in a way that it will never be found. It should be noted however that if the query node is part of a clique or independent set in the current graph, then those sets *will* be found by the algorithm.

The running of the algorithm can be conceptualized as a binary tree where for each node its left children are adjacent to it in the graph and its right children are non-adjacent to it. This leads to the conclusion that the maximum clique can be found by going down the path in the tree with the most left edges. And likewise, the largest independent set is found on the path with the most right edges. It is conjectured that for graphs without large cliques, a careful implementation of the Ramsey algorithm can run in $O(|V| + |E|)$ time [1].

Runtime Comparison with NetworkX Library

The networkx python library has a function called `ramsey_R2` that is an implementation as the same algorithm described above. We conducted a test on the runtimes of the two algorithms with similar graphs (all had an overall edge density of .37-.40). Though the stochastic nature of this algorithm renders runtime analysis and comparison somewhat obsolete, a test like this does provide a glimmer of insight into which algorithm is faster for most purposes.

	10 Nodes, 2-Clique	100 Nodes, 20-Clique	1000 Nodes, 200-Clique
ramsey	5.8 Seconds	23.8 Seconds	276.4 Seconds
nx.approximation.ramsey_R2	5.7 seconds	16.2 Seconds	127.5 Seconds

Testing

Because of the stochastic nature of the algorithm, there are not many guarantees provided. So the only cases I saw were testing to see that the clique it returned was actually a clique, and that the independent set returned was actually an independent set.

3 Girvan-Newman

The Girvan-Newman algorithm is one for detecting at least two distinct communities or clusters within a given graph.

High-Level Pseudocode

1. Calculate the betweenness of all *edges* in the graph.
2. Identify the maximum value of the set of all betweenness values and remove all edges with that value from the graph.
3. Recalculate the betweenness of all edges and repeat Step 2. until two distinct communities/-clusters are found.

Time Complexity & Data Structure

Unlike Prim's Algorithm, the Girvan-Newman algorithm does not require any special data structures (excluding the graph itself) to perform. The least trivial aspect of this algorithm is calculating the betweenness of each edge, which can be done using the built-in networkx `edge_betweenness_centrality` function, which follows Brandes' algorithm [2] at a time complexity of $O(|V| \cdot |E|)$. Afterwards, finding the edge with the highest betweenness is $O(|E|)$ because we must iterate through all edges. Finally, we must repeat until we identify a cluster, which, in the worst-case scenario, will be done E times. All this together gives us an expected runtime of $O(|E| \cdot (|V| \cdot |E|))$ aka $O(|V| \cdot |E|^2)$.

Discussion

During our presentation of this implementation, we discussed how the concept of edge betweenness relates to bridges in graphs. These two concepts are nearly identical in that they are both critical to identifying communities or clusters within a graph. Specifically, a bridge is any edge in a graph that, when removed, creates two unconnected components. Equivocally, in the Girvan-Newman algorithm, we remove the edge with the highest betweenness with the goal of creating two unconnected components. That is, these two are nearly synonymous in that a bridge is the edge in a graph with the highest betweenness. Additionally, both are found in a similar way, via some graph traversal algorithm, albeit not the same algorithm in every case. An important clarification here is that this similarity is not symmetric; a bridge will always have high betweenness, but not every edge with the highest betweenness will disconnect two components and therefore not a bridge.

$$\text{bridge} \rightarrow \text{high betweenness} \qquad \text{high betweenness} \not\rightarrow \text{bridge}$$

Additionally, the original presented implementation of this algorithm relied on the networkx edge betweenness function. Since then, we have implemented Ulrik Brandes' algorithm for edge betweenness as described in his paper "*On variants of shortest-path betweenness centrality and their generic computation*" [2]. The pseudocode for said algorithm is available below.

Low-Level Pseudocode

Algorithm 3: `Girvan-Newman(`G`)`

Input: A graph G
Output: A tuple of communities / clusters

```
COMPONENTS ← list(nx.connected_components(G))
// While there is only one distinct community/cluster
while len(COMPONENTS) = 1 do
    // Get the map of edges to betweenness values using networkx
    BTWNSS ← Brandes(G)
    // Store the highest betweenness value
    MAX_BTWNSS ← max(BTWNSS.values())
    // Identify and remove the edge(s) mapped to the highest betweenness
       value from the graph
    foreach edge e, value v in BTWNSS do
        if v = MAX_BTWNSS then
            G.remove_edge(e)
    // Finally, update COMPONENTS to see if we have removed enough edges to
       identify more than one distinct communities/clusters
    COMPONENTS ← list(nx.connected_components(G))
return tuple(COMPONENTS)
```

Algorithm 4: `Brandes(`G`)`

Input: A graph G
Output: A dictionary c_B mapping edges to their betweenness centrality values

```
INITIALIZE c_B[(v, w)] ← 0 for all (v, w) ∈ E
foreach s ∈ V do
    // Single-source shortest-paths problem
    INITIALIZE Pred[w] ← [], dist[t] ← ∞, σ[t] ← 0 for all w, t ∈ V
    dist[s] ← 0, σ[s] ← 1
    Q ← deque(), S ← list()
    Enqueue s → Q
    while Q not empty do
        Dequeue v ← Q
        Push v → S
        foreach w such that (v, w) ∈ E do
            // Path discovery
            if dist[w] = ∞ then
                dist[w] ← dist[v] + 1 Enqueue w → Q
            // Path counting
            if dist[w] = dist[v] + 1 then
                σ[w] ← σ[w] + σ[v] Append v → Pred[w]
    // Accumulation phase
    INITIALIZE δ[v] ← 0 for all v ∈ V while Snotempty do
        Pop w ← S foreach v ∈ Pred[w] do
            c ← (σ[v]/σ[w]) · (1 + δ[w]) if (v, w) ∈ c_B then
                c_B[(w, v)] ← c_B[(w, v)] + c
            else
                c_B[(v, w)] ← c_B[(v, w)] + c
            δ[v] ← δ[v] + c
return c_B
```

Testing

Much testing was done with this algorithm by providing it a graph not specifically generated to have clusters, that is, a normal connected graph with a random number of nodes and edges. Additionally, some specifically clustered graphs were also introduced by generating two separate aforementioned random graphs and connecting them, which we found to be too obvious for the algorithm. However, no error checking was introduced into the implementation, so it would fail to handle a graph with zero nodes or edges. Inversely, the implementation was tested on large graphs (with many nodes and edges) and did not suffer performance-wise; the expected runtime was $O(|V| \cdot |E|^2)$ but seemed to operate in near constant time when averaged over 10 trials on graphs of varying sizes within the range of 10 to 1000 in steps of 50. This moderately outperformed the built-in networkx method, solely because of how little error checking was involved; this implementation unfortunately has to make assumptions as a sacrifice to gain runtime efficiency which ended up being both a limitation and a time-saver.

4 Bellman-Ford

Algorithm 5: Bellman-Ford($\mathcal{G}, w, s$)

Input: $\mathcal{G} = (V, E)$
Input: $w(e)$ is defined for every $e \in E$
Input: $s \in V$
Output: Mapping DIST of the shortest distance from each node to s
Output: Map of predecessors PRED of every node on a shortest path from s in $\mathcal{G}$
Output: NULL if a negative-weight cycle exists in $\mathcal{G}$

```
foreach v ∈ V do
    DIST[v] ← ∞
    PRED[v] ← NULL
DIST[s] ← 0
for i ← 1 to |V| − 1 do
    foreach edge (u, v) ∈ E do
        if DIST[u] + w(u, v) < DIST[v] then
            DIST[v] ← DIST[u] + w(u, v)
            PRED[v] ← u
foreach edge (u, v) ∈ E do
    if DIST[v] > DIST[u] + w(u, v) then
        CYCLE ← {v}
        while u ≠ v do
            CYCLE = CYCLE ∪ {u}
            u ← PRED[u]
        return ERROR "Negative cycle found", CYCLE
return DIST, PRED
```

The Bellman-Ford algorithm is an algorithm used to compute shortest paths in a directed graph with negative edge weights (but no negative cycles). It can also be used for the express purpose of detecting negative cycles in a weighted directed graph. The algorithm runs in $O(mn)$ time, where m is the number of edges in the directed graph, and n is the number of vertices. Most versions of the algorithm will terminate if a negative-weight cycle is detected, although the version we implemented can return the negative cycle found in the final loop of the algorithm.

Time Complexity and Analysis

The initialization step is $O(|V|)$. The main loop iterates through every edge in E $|V| - 1$ times, thus it is $O(|V||E|)$. The negative cycle detection step is $O(|E|)$. Considering this, we can see that the Bellman-Ford algorithm runs in $O(|V||E|)$ time.

In the provided Python implementation, DIST and PRED are implemented as dictionaries in case the vertices in the graph are not represented as integers. Since lookup in a python dictionary is $O(1)$, the above time complexity is justified. I couldn't find any resources on the space complexity of the dictionary implementation Python uses, but it is a hash table which can potentially use more than $O(n)$ space. However, if the nodes are known to only be integers then DIST and PRED can be implemented as arrays indexed on the node number, meaning space for Bellman-Ford can be as low as $O(n)$.

Correctness Proof

Proof based on [3].

for the proof, we will first alter the algorithm such that a new distance array is calculated for each iteration in the main loop of the algorithm. We will call the distance array at iteration k d_k. We define and $d_k[u] = \text{inf}$ for all k and vertices u. Then we set $d_0[s] = 0$. For each iteration i on edge (u, v) replace the edge relaxation and if statement with

$$d_i[v] \leftarrow \min(d_{i-1}[v], d_{i-1}[u] + w(u, v), d_i[v])$$

Bellman-Ford detects negative cycles

If a negative cycle exists and is accessible from the source node, then after the final iteration of Bellman-Ford (with array d_{n-1}) there would be an edge (u, v) such that $d_{n-1}[v] > d_{n-1}[u]+w(u, v)$. Also if such a cycle existed, it would be of the form $v_0 \to v_1 \to \cdots \to v_k$ with $v_0 = v_k$ and

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

.

For sake of contradiction, assume that Bellman-Ford cannot detect negative cycles and as such, $d_{n-1}[v_i] \leq d_{n-1}[v_{i-1}] + w(v_{i-1}, v_i)$ for all $i = 1, ..., k$. If we sum up these values for all possible values of i we get

$$\sum_{i=1}^{k} d_{n-1}[v_i] \leq \sum_{i=1}^{k} d_{n-1}[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{1}$$

We now make the following derivation:

$$\sum_{i=1}^{k} d_{n-1}[v_{i-1}] = \sum_{i=0}^{k-1} d_{n-1}[v_i] = d_{n-1}[v_0] + \sum_{i=1}^{k-1} d_{n-1}[v_i] = d_{n-1}[v_k] + \sum_{i=1}^{k-1} d_{n-1}[v_i] = \sum_{i=1}^{k} d_{n-1}[v_i]$$

Combining this result with equation 1 yields:

$$\sum_{i=1}^{k} d_{n-1}[v_i] \leq \sum_{i=1}^{k} d_{n-1}[v_i] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

This contradicts the invariant that the sum of all weights in a negative cycle is negative. A contradiction. Therefore it must be true that the Bellman-Ford algorithm detects cycles with negative weight.

Bellman-Ford finds shortest paths

As we did for the DIST array, let us transform the PRED array such that a for each iteration k in the main loop of the algorithm, we have a new array p_k that stores for each nodes its predecessor in the shortest path from s containing at most k edges. And p_0 is defined as NULL for all vertices. For each edge (u, v) in an iteration i of the main loop we define

$$p_i[v] = u \text{ if } d_{i-1}[u] + w(u, v) < d_{i-1}[v] \text{ and } d_{i-1}[u] + w(u, v) < d_i[v] \text{ else } p_{i-1}[v]$$

We will prove that the Bellman-Ford algorithms finds the shortest paths via mathematical induction.

Base Case. In the case that $k = 0$, (i.e., before the main loop starts). Then the only vertex in d_0 with a finite value is $d_0[s] = 0$, which implies that the only path from s to any other vertex with 0 edges is one from s to itself with a weight of 0, which is correct. We also know that all values in p_0 are NULL, implying that there is no shortest path from s to any other node, which is also correct. Thus the base case holds.

Inductive Hypothesis. Suppose that for all vertices v, the value held by $d_{k-1}[v]$ is the weight of the shortest path from s to v with at most $k-1$ edges. Also assume that the value $p_{k-1}[v]$ is the predecessor of v in a shortest path from s to v.

Inductive Step. In iteration k. After iterating through all edges, for every vertex v either $d_k[v] = d_{k-1}[v]$ or $d_k[v] = d_{k-1}[u'] + w(u', v)$ for some u' such that $\forall v' \in V \setminus \{u', v\}$, $d_{k-1}[u'] + w(u', v) \leq d_{k-1}[v'] + w(v', v)$. From this, we can see that any value of $d_k[v]$ that differs from $d_{k-1}[v]$ has a smaller weight, and thus represents a path from s to v with at most k edges that is a shorter path than the one from s to v with at most $k-1$ edges. We know that $d_k[v]$ contains the weight of the *shortest* path from s to v with at most k vertices because the algorithm chose the lowest weight obtained from adding an edge (u, v) onto the shortest path from s to u with at most $k-1$ edges.

$p_k[v]$ contains the predecessor of v in the shortest path from s to v with at most k edges, and this is verified because $p_k[v]$ is the value of u such that $d_{k-1}[u] + w(u, v)$ is smaller than $p_{k-1}[v]$ and any value already in $p_k[v]$, which we showed above is the shortest path from s to v.

Testing

The Bellman-Ford algorithm effectively has two modes: the normal mode where it simply finds the single-source shortest path of a directed graph, and the mode where a negative cycle is found. For the first case I merely tested on a few known graphs containing negative edge weights that my implementation of the algorithm finds the shortest path.

The negative cycle case was somewhat more involved:

- Can the algorithm detect negative cycles when the source is part of it?
- Can the algorithm detect negative cycles that don't include the source?
- Will the algorithm complete successfully when all negative cycles are unreachable from the source?

References

[1] Ravi B. Boppana and Magnús M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. In *BIT. Numerical Mathematics*, 2006.

[2] Ulrik Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, May 2008.

[3] Jessica Su. Cs 161 lecture 14 – amortized analysis.