Solutions - Problem Set #5

Alice Bob (uni). Collaborators: A. Turing. - uni@columbia.edu

November 13, 2019

1 Problem 1

Since we are given an undirected graph, if there exists a path between (node A, node B) and (node C, node B), then there will exists a path between (node A, node C). Let S_i = set of all nodes reachable from node i.

$$\exists \text{ edge}(i, j) \implies S_i == S_j$$

This implies all nodes, i in a connected component, C of the graph will have the reachable nodes = $C - \{i\}$.

Using above properties we can search for the connected components present in the graph, G, and generate the required lists.

We can traverse the graph in a BFS fashion to generate list of connected components. For each connected component we can generate the output Adjacency List by generating Component - {node}. Following Code outlines the process.

```
def GenerateList(adjList):
    unvisitedNodes = set(1..n)
    traverseQueue = []
    outputAdjList = {}

    node = 1 # starting node

    while(len(unvisitedNodes)):
        connectedSet = set()
        traverseQueue.append(node)
        while(len(traverseQueue)):
            node = traverseQueue.pop()
            unvisitedNodes.remove(node)
            connectedSet.add(node)

            for child in adjList[node]:
```

```
            if child in unvisited:
                traverseQueue.append(child)

    for node in connectedSet:
        outputAdjList[node] = connectedSet
```

Time Complexity

The time complexity of the algorithm is $O(n + e)$.

- Time complexity BFS: $O(n + e)$
- Time complexity generating Lists: $O(n)$

The current implementation has a linear run time in number of edges and number of nodes. To generate the different connected components, we will have to traverse each node and each edge at least once. Since otherwise we'll miss the node from the connected components.

Hence the optimal run time of the above problem: $O(n + e)$.

2 Problem 2

(a) Odd Length Cycle Given that graph, $G = (N, E)$ is bipartite.

- This implies, for each edge $e = (n_1, n_2)$, $\implies n_1 \in N_1, n_2 \in N_2$.
- For any two nodes $n_{1i}, n_{1j} \in N_1$ any path between them should be of even length.

  Since there can't be a direct edge between nodes in N_1. For every edge connecting N_1 to N_2 on the given path, there should exist an edge connecting N_2 to N_1. Hence our path length would always be even.
- This implies any path between $(n_i, n_i), n_i \in N$, i.e. a cycle, should be of even length as well or bipartite graphs can't have odd length cycles.

(b) Algorithm We can detect the presence of odd length cycle by traversing in a DFS fashion.

```
func checkBipartite(n, N1, N2):
    if n.color == 'Black':
        return 0, []

    n.color = Grey

    for child in n.children:
        if (child in N1 and n in N1) or
            (child in N2 and n in N2):
            return 1, [child, n]

        else if child.color != 'Grey':
            cycle, path = checkBipartite(child, N1, N2)

            if cycle != 0: # cycle found
                if cycle == 1:
                    if n == path[0]:
                        cycle = 2 # cycle complete
                    path.append(n)
                return cycle, path

    n.color = Black
    return 0, []

func main():
    G = (N1, N2, E)
    for n in N:
```

```
        if n.color == 'White':
            cycle, path = checkBipartite(n, N1, N2)
            if cycle:
                return path

    return True # Graph is Bipartite
```

Since in a odd length cycle at least one pair of neighbors should belong to same set, we put a special check condition in DFS algorithm. This condition checks if node and its child are in same set and declares a cycle if such condition is true.

Also once such cycle has been discovered we keep appending nodes in the cycle to the path variable. Once we discover the node that started the cycle we can declare our cycle complete and return it without any more additions. If no such cycle is found we can declare our graph bipartite.

Time Complexity

Since the above algorithm modifies the DFS, with adding some $O(1)$ operations. The run time of above algorithm is run time of DFS $= O(|N| + |E|)$.

3 Problem 3

(a) Decide consistency. We will build a weighted DAG $G(V, E, w)$ as follows. V consists of n nodes $x_1, \ldots, x_n$ corresponding to the n variables of the input; and E contains an edge (i, j) if, and only if, at least one of $x_i < x_j$ or $x_i \leq x_j$ belongs to the set C. *(Without loss of generality, we assume that no constraint of the form $x_i \leq x_i$ or $x_i < x_i$ belongs to C; as this can be checked in time $O(|C|) = O(m)$, and inequalities of the first type can be removed from C (as they are automatically satisfied by any assignment) while inequalities of the second type cause immediate rejection (as they cannot be satisfied by any assignment).)*

Furthermore, we define the weight of each edge $(i, j) \in E$ as follows: if $x_i < x_j$ is in C, then $w(i, j) = 1$; and otherwise (if only $x_i \leq x_j$ is in C) we set $w(i, j) = 0$. Clearly, this graph G can be built (in the adjacency list form) or simulated on-the-fly in time $O(n + m)$, and has n nodes and (under the above assumptions) m edges.

Given this, we first run the strongly connected component (SCC) algorithm seen in class (Lecture 16, based on Depth-First Search (DFS)) on G, to compute the list of SCCs in time $O(n + m)$. For each tree of the resulting forest, we then go over all the edges it contains, and check that there is no edge of positive weight (i.e., each tree correspond to a SCC only contains edges of weight 0). If this is the case, we output accept, and otherwise reject. Again, this last step takes time $\sum_{\text{tree T}} O(1+\text{edges}(T)) = O(n + m)$, so overall the algorithm has time complexity $O(n + m)$ as wished.

Correctness. The idea is that the set C is inconsistent if, and only if, C contains a cycle with positive weight.

Assume the algorithm returns reject. Then, there exists a SCC containing an edge with weight 1, and thus a cycle of the form $i_1 \to i_2 \to \cdots \to i_k \to i_1$ such that $w(i_1, i_2) = 1$. By construction, this cycle corresponds to chain of inequalities $x_1 < x_2 \prec \cdots \prec i_k \prec x_1$ (where $\prec$ is either a strict $<$ or weak $\leq$ inequality). In particular, this implies that any assignment should satisfy $x_1 < x_1$, which is impossible: the set C is inconsistent.

Conversely, assume that the algorithm returns accept. This means that every SCC either has no edge, or only 0-weight edges. This in turns means that assigning the same value to every variable in any given SCC will not violate any constraint (as they are all weak inequalities). Doing as in class, we can "shrink" (as a thought experiment) every SCC to a single meta-node and construct the DAG of SCC's: any topological sort of this new DAG implies a valid assignment of variables (starting with 1 for the first meta-node in the topological sort, and 2 for its immediate neighbors, etc.). The resulting assignment does not violate any constraint, and is therefore a witness of the consistency of C.

(b) Find a minimum solution. In this section, we assume the set of inequalities is satisfiable (this can be checked in time $O(n + m)$ using the algorithm of part (a)).

As suggested in the last part of (a), we build the same graph G as before, and in $O(n+e)$ time compute the DAG G_{scc} of SCC's (note that G_{scc} has at most n nodes, and at most m edges). We then compute a topological sort of G_{scc} as seen in class (using DFS, on time $O(n+m)$), but doing so by maintaining another label for each node. Namely, initially each node has a label $a[u] = 1$, and whenever "considered" an unmarked node v (discovered with parent u) is assigned the value $a[v] = \max(a[v], a[u] + w(u,v))$ (so that it has value $\max(a[v], a[u]+1)$ is there is a positive edge (strict inequality) between u and v, and $\max(a[u], a[v])$ if the inequality is weak).

Once this is done, we "map" back to the original graph G by giving to each node v in V the value $a[u]$, where $u \in V_{\text{scc}}$ is the SCC to which u belongs.

The correctness follows immediately from the construction of G_{scc} and properties of the topological sort; and minimality is implied by the fact that the assignment gives to x_i a value greater than to x_j (with differenc $k > 0$) if and only if there is a path with k positive edges between x_i and x_j.

4 Problem 4

(a) We will prove by contradiction. Suppose that the edge weights in the graph are distinct, but there are two different minimum spanning trees A and B. Let e be the minimum weight edge that is in A but not in B. Suppose e connects vertices v_1 and v_2. Consider adding e to B. This creates a cycle, since B is an MST, so there must be some path already from v_1 to v_2. Let f be the maximum weight edge on that cycle. We know that f cannot be part of any MST because of the exchange argument. Now, there are two cases. If $f = e$, then f is in A, which contradicts that f cannot be in any MST. The second case is if $f \neq e$. If we remove f, then we have another spanning tree, C. Now, we can't have $w(f) > w(e)$, because then C is a minimum spanning tree which contradicts A and B being minimum spanning trees. We can't have $w(f) < w(e)$ because then f is not the maximum weight edge on the cycle. So we must have $w(f) = w(e)$, which contradicts the supposition that the edge weights are not unique.

(b) We will prove by contradiction. Suppose that edge $e_1 = (u, v)$ is one of the minimum weight edges across the partition, but there is no MST that contains this edge. Denote this MST as A. Consider adding e_1 to A, which creates a cycle (using a similar argument as part (a)). On the cycle, there must be at least one other edge e_2 that crosses the partition. If $w(e_2) = w(e_1)$, we have that e_2 is also a minimum weight edge across the partition, so replacing e_2 with e_1 and removing e_1 from the cycle gives an MST, contradicting the claim that there is no MST containing e_1. If $w(e_2) \neq w(e_1)$, then we can replace e_2 with e_1 to get an MST of smaller cost, which contradicts the fact that A is an MST. Thus, there is always an MST that contains e_1.

(c) The algorithm doesn't work. Consider a "square" graph with vertices a, b, c, d and edges $w(a, b) = 10, w(c, d) = 10, w(b, c) = 1, w(a, d) = 1$. Then, if we partition into the groups $V_1 = \{a, b\}$ and $V_2 = \{c, d\}$, we get a spanning tree of cost 21. However, the MST is of cost 12.

Problem 5

(a) Let $e = (u, v)$ be the negative cost edge. We can solve this by running Dijkstra's algorithm on a modified graph. Let $G' = (V, E', w)$ where $E' = E/e$ (the same graph as G with the edge e removed). We run Dijkstra's algorithm on G', starting from source s. After running Dijkstra's algorithm, let d_s be the distances from s to all other vertices. The shortest distance to v from s is the minimum between $d_s[v]$ and $d_s[u] + w(e)$. These correspond to the two cases of following some path to v in G' or taking a path to u and following the negative cost edge. The minimum between these choices will always return the shortest path to v, since we assume that Dijkstra's algorithm returns the shortest path to u.

The time complexity of this approach is the time it takes to run Dijkstra's algorithm once, which is $O(m + n \log n)$, where m is the number of edges.

(b) We reduce this problem to that of finding a negative weight cycle in a graph. In particular, we create a graph G with the following properties:

(a) There is a vertex v_i for every currency c_i

(b) There is a directed edge (v_i, v_j) between all pairs of vertices

(c) The weight of edge (v_i, v_j) is $-\log(R[v_i, v_j])$

Lemma:
Let $C = \{v_{i_1}, v_{i_2}, ..., v_{i_k}, v_{i_1}\}$ be a cycle in G. If C has negative weight then

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot ... \cdot R[i_k, i_1] > 1$$

Else, if C has non-negative weight

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot ... \cdot R[i_k, i_1] \leq 1$$

Corollary:
G has a negative weight cycle if and only if there exists a sequence $(c_{i_1}, c_{i_2}, ..., c_{i_k})$ that achieves arbitrage. Moreover, if G does have a negative weight cycle, the vertices of this cycle correspond exatly to the desired sequence of currencies.

Proof. (of lemma)
Because of how we chose edge-weights for G, the weight of C is

$$\begin{aligned} &[-\log(R[v_{i_1}, v_{i_2}])] + [-\log(R[v_{i_2}, v_{i_3}])] + ... + [-\log(R[v_{i_k}, v_{i_1}])] = \\ &-[\log(R[v_{i_1}, v_{i_2}]) + \log(R[v_{i_2}, v_{i_3}]) + \log(R[v_{i_k}, v_{i_1}])] = \\ &-\log(R[v_{i_1}, v_{i_2}] \cdot R[v_{i_2}, v_{i_3}] \cdot ... \cdot R[v_{i_k}, v_{i_1}]) \end{aligned} \tag{1}$$

This completes the proof, as it is clear that this last term is negative (*i.e* the cycle has negative weight) if and only if the value inside of the log is > 1 – *i.e* if $R[v_{i_1}, v_{i_2}] \cdot R[v_{i_2}, v_{i_3}] \cdot ... \cdot R[v_{i_k}, v_{i_1}] > 1$ □

The algorithm:

(a) Create the graph G specified above

(b) Run Bellman-Ford on G, using the variation that returns a negative weight cycle if one exists

(c) If there is no negative weight cycle, return: "No Possible Arbitrage"

(d) If there is a negative weight cycle, return "Arbitrage is Possible" (or, for part b, return the vertices of this cycle)

Correctness stems from the above corollary. Creating the graph takes $O(n^2)$ time, and finding a negative-weight cycle using the procedure below takes $O(n^3)$ time because the graph has $O(n^2)$ edges, $O(n^2)$ vertices. Thus, the total run-time is $O(n^3)$.

Next, we need to show how to find a negative cycle (if it exists), after we run Bellman Ford in part a). To do this, we use the predecessor labels determined by Bellman Ford. If a negative cycle was found, this means that in iteration $n+1$ of Bellman Ford, there is some vertex v such that $v.d$ decreased. This means that there is a negative weight cycle on the path from s to v, so we just follow predecessor vertices back from v, marking every vertex that we see. Since there must be a cycle, we will eventually mark some vertex x twice, which means that x is on some negative weight cycle. We can print this cycle by just starting at x and following predecessor pointers.
Note that it is not enough to just follow predecessor vertices from v until we reach v again, because even though $v.d$ decreased, v might not actually be on a cycle.