CSOR4231 Algorithm
HW4

Jialin Zhao — jz2862

April 2, 2018

1 Problem I

Give an efficient algorithm to find an odd-length cycle in a directed graph.

Description:

First break the graph into strongly connected components (SCCs). In Each SCC, run BFS from an arbitrary node s to color the levels of the tree as red and blue to represent two parities.
If there is no edge found between levels of the same color then G is bipartite, which means the graph contains no odd cycle.
Otherwise, if there is an edge (u,v) between levels of the same color, run BFS to get a path from v to s. If the length of the path (s to u + edge(u,v) + v to s) found is odd, an odd cycly is composed of path s to u to v to s. If the length of this path is even then an odd cycle is s to v to s.

Pseudocode:

Algorithm 1 Function odd-cycle(G)

```
Using DFS to find all SCCs in G
for every C in SCCs do
    Run BFS from an arbitrary node s in C and color the levels of BFS tree using two colors representing
    different parts, represent the path from s to u as P_su
    if When running BFS, the edge (u,v) considered in this step is between nodes of same color then
        Run BFS from v in C to get a path P_vs from v to s
        if The length of (path P_su from s to u + P_vs) is even then
            return The cycle P_su+ edge(u,v) + P_vs
        else
            return The cycle P_sv + P_vs
        end if
    end if
end for
```

Correctness:

Because a cycle can only be found in a SCC, the problem is equal to finding an odd length cycle in one of the SCCs in graph G. We only need to prove that the algorithm can find odd-length cycle in a SCC after breaking the Graph into SCCs.
If there is a odd length circle, then it cannot be two-colored so that an edge between the nodes in parity will be found by our algorithm.
If an edge(u,v) between nodes in same parity is found, then the length of path P_{su} from s to u and the path P_{vs} from s to v are of same parity, the length of cycle P_{su} + edge(u,v) + P_{vs} and the length of cycle P_{sv} + P_{vs} are of different parity, i.e. either cycle P_{su} + edge(u,v) + P_{vs} or the cycle P_{sv} + P_{vs} are odd-cycle we want.
So, the algorithm return a odd cycle if exist.

Complexity:
Running Time: Finding SCCs takes O(n+m) time; Two-coloring takes O(n+m) time; Finding P_{vs} takes O(n+m) time; So the total time complexity is O(n+m).

2 Problem II

Given an undirected graph G = (V, E) and a specific edge $e \in E$, give an efficient algorithm that determines whether G has a cycle that contains e.

Description:

Let e = (u,v). Using DFS in (V,E-e) to check if start node u can get to v. If so, G has a cycle containing e; otherwise there is no cycle containing e in G.

Pseudocode:

Algorithm 2 Function cyclecontain(G, e)

deleting e(u,v) from E
Start from node u and run DFS until find a path to v and return YES
return NO

Correctness:

The graph has a cycle containing e(u,v) if and only if (v are reachable from u through e) whcih is already known and u are reachable from v which can be correctly checked by the algorithm after removing v and dding a DFS on this graph from v.

Complexity:

Running DFS takes $O(|V| + |E|)$ time.

3 Problem III

design an algorithm that fills in the entire cost array.
(a) Give a linear-time algorithm that works for directed acyclic graphs.

Description:

Sort DAG topologically. For each node $u \in V$, in reverse topological order, compute the minimum cost of u by min(the cost of u itself, the cost of every node v in edge $(u, v) \in E$)

Pseudocode:

Algorithm 3 Function MiniCostDAG(G)

Use DFS to sort the DAG in topological order.
Array C $\Longleftarrow$ the array storing cost for each node, initialized with each nodes' price
In reverse topological order, compute C[u] = min(C[u],C[v] for every edge $(u, v) \in E$)

Correctness:

Lemma: C[u] is corrected for nodes computed in reverse topological order. **Base case**: For the first node u in reverse topological order, C[u] = the price of u is correct because there is no nodes reachable from u.
Induction: Suppose the set of nodes traversed in reverse topological order yet are computed with correct C[u].
For the next node v in reverse topological order, C[v] = min(C[u],C[v] for every edge $(u, v) \in E$) should be correct if all C[v] for reachable nodes v from u is computed correctly. Since we traversed nodes in reverse topological order, all v reachable from u is already computed correctly. So the cost for u, i.e. C[u] is correctly computed.
Conclusion: So the cost of every node is correctly computed in reverse topological order.

Complexity:

Sorting takes $O(|V| + |E|)$ time; Iterating over all edges in E and compute the minicost takes $O(|E|)$ time; The total running time is $O(|V| + |E|)$;

(b) Extend this to a linear-time algorithm that works for all directed graphs

Description:

Find all SCCs in G. Set the cost of each SCC as the minimum cost of its nodes. View the graph as DAG C(A,B) composed of SCCs(A) and edges B between SCCs, sort it and compute the minimumcost by min(the cost of SCC a itself, the cost of every nodes b in edge(a,b) $\in B$) in reverse topological order. Assign the SCC cost to its node members as node cost.

Pseudocode:

Algorithm 4 Function MiniCost(G)

Use DFS to find all SCCs in G and get DAG G_{SCC}
for each SCC C **do**
  $P_{SCC}[C] \Longleftarrow$ *the minimum price of all nodes in C*
  assign $P_{SCC}[C]$ to SCC C as its price
end for
Run MiniCostDAG(G_{SCC}) to find the minimum cost for every SCC
Assign the minimum cost of every SCC to each nodes in it

Correctness:

Because nodes in a same SCC are reachable from each other, so the cost of them is the same. If there is no edge out of the SCC, the cost for the nodes in this SCC should be the minimum price of nodes in SCC. The problem is equal to (a) if we consider every SCC as a component of DAG.
Lemma: C[u] is corrected for SCCs computed in reverse topological order. **Base case**: For the first SCC u in reverse topological order, C[u] = the price of u is correct because there is no SCCs reachable from u.
Induction: Suppose the set of SCCs traversed in reverse topological order yet are computed with correct

C[u].
For the next SCC v in reverse topological order, C[v] = min(C[u],C[v] for every edge $(u, v) \in E$) should be correct if all C[v] for reachable SCCs v from u is computed correctly. Since we traversed SCCs in reverse topological order, all v reachable from u is already computed correctly. So the cost for u, i.e. C[u] is correctly computed.
Conclusion: So the cost of every SCC is correctly computed in reverse topological order.

Complexity:

Finding SCCs takes $O(|V| + |E|)$ time; Sorting takes $O(|V| + |E|)$ time; Iterating over all edges in E and compute the minicost takes $O(|E|)$ time;
The total running time is $O(|V| + |E|)$;

4 Problem IV

Design and analyze a dynamic programming algorithm for this problem that runs in time polynomial in n and $\sum_{i=1}^{n} a_i$

Description:

Using a modified version of Dijkstra's Algorithm.
Inplement a little modification to Update function to raise the priority of the shortest distance with less edges in this path.
Use a priority queue implemented as a binary min-heap: store vertex u with key dist[u] attached with attribute best[u].

Pseudocode:

Algorithm 5 Function DijkstraBest(G, e)

```
Priority queue Q with key dist[v] and attribute best[v]
Initialize dist[s] = 0, and other value of dist as ∞
Initialize best[s] = 0, and other value of best as ∞
Maintain a set S of nodes for which the distance from s has been determined
Initialize Q = {V;dist;best} , S = []
for Q is not empty do
  u = ExtractMin(A)
  //Extract the node with minimum dist[u] from Q,
  //if there is more than one minimum, choose one with the smallest best[u]
  S = S ∪ {u}
  for all edge (u, v) ∈ E do
    Update(u,v)
  end for
end for
```

Algorithm 6 Function Update(u, v)

```
if dist[v] > dist[u] + w(u, v) then
  dist[v] = dist[u]+w(u,v)
  best[v] = best[u]+1
else if dist[v] == dist[u] + w(u, v) then
  if best[v] > best[u] + 1 then
    best[v] = best[u]+1
  end if
end if
```

Correctness:

Lemma: dist[u] of nodes in S is the shortest distance of path from s;
best[u] of nodes in S is the smallest number of edges for shortest path.
Base case: S = {s}, since s is the start node, dist[u] = 0, best[u] = 0 represent the shortest path
Induction: Suppose S satisfy the Lemma;
For case $S' = S + u$, where u is the last node added into S by the algorithm:
Since the algorithm picks u based on smallest dist and best value via (path in S) + (edge from node in S to u), u is nearer to s than other nodes outside S. Suppose for a contradiction that the shortest path from s-to-u is Q whose length a is less than dist[u] or its number of edges b is less than best[u];
Since the path for dist[u],best[u] is the minimum value of distance to s and number of edges for each best paths from s to a node v in S + edge(v,u), $a < dist[u]$ or $b < best[u]$ means the shortest path Q can only get to u from some node m outside S.
However, since dist[u] and best[u] is already the smallest for nodes not in S, a = dist[m] + edge(m,u) must

$\geq$ dist[u] or b = best[m]+1 $\geq$ best[u] which means the Q can only be from inside S and the dist and best value for u is correct.
Conclusion: The lemma also holds for S', which can also hold for S = V

Complexity:

Time complexity: same as normal Dijkstra: $O(|E|log|V|)$

5 Problem V

Consider a network of roads G = (V, E) connecting a set of cities V . Each road in E has an associated length l_e. There is a proposal to add one new road to this network, and there is a list E' of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length.
As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network G will result in the maximum decrease in the driving distance between two fixed cities s and t in the network. Give an efficient algorithm for solving this problem.

Description:

View the problem as adding a pair of edges(u,v) edge(v,u) between two nodes in an directed graph to minimize the distance for path from s to t. Use Dijstra algorithm twice to get the shortest distance from s to any other nodes and from t to any other nodes. Then compute the shortest distance for every possible edge in $|E'|$ with the equation (the shortest path from s to u)+edge(u,v)+(the shortest path from v to t). Choose the edge with the shortest distance.

Pseudocode:

Algorithm 7 Function BestPath(G, E')

For every pair (u,v) in E', replace it with edge(u,v) and edge(v,u)
Use Dijkstra algorithm on G to get all Ps[u], Pt[u]
Ps[u] $\Longleftarrow$ the shortest distance from s to other node u
Pt[u] $\Longleftarrow$ the shortest distance from t to other node u
for $e \in E'$ **do**
  compute the shortest distance dist[e] = $Ps[u] + l_e + Pt[u]$
end for
return the e with the smallest dist[e]

Correctness:

Since the algorithm goes through all possible city pairs that can add new edge, we only need to essure that the distance from s to t computed for the best edge is smaller than any other edges so that it can be chosen. For edges that are not the best edge, the shortest distance P_{su}+$edge(u, v)$+P_{vt} computed by running Dijkstra algorithm before adding edge(u,v) can only be equal or larger than the real distance because P_{su} *and* P_{vt} can only be equal or larger than the shortest path whens considering in edge(u,v).
For the best edge(u,v) to get shortest path, the shortest path from s to t must be equal to P_{su}+$edge(u, v)$+P_{vt} computed by running Dijkstra algorithm before adding edge(u,v), since the shortest path from s to u does not contain v because:
If the shortest path from s to u contains v, then at least path from s to v to t is better than this case which is contradictory to the shortest path assumption. Similarly the shortest path from v to t does not contain u. So the distance is computed correctly in this case.
CONCLUSION: The best edge is chosen correctly because the path distance is the shortest in every case.

Complexity:

Runnning Dijsktra algorithm takes $O(|E| \log |V|)$.
Iterating on every possible edges takes $O(|E'|)$.
The total time complexity is $O(|E| \log |V| + |E'|)$

6 Problem VI

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy 1.0486 dollars, thus turning a profit of 4.86 percent.
Suppose that we are given n currencies $c_1, c_2, ..., c_n$ and an $n * n$ table R of exchange rates, such that one unit of currency c_i buys R[i,j] units of currency c_j.

a. Give an efficient algorithm to determine whether or not there exists a sequence of currencies $c_{i_1}, c_{i_2}, ..., c_{i_k}$ such that

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

Analyze the running time of your algorithm.

Description:

Convert the R[i,j] to $-\log R[i, j]$ then use Bellman Ford algorithm to find whether there exists a negative cycle in G(V,E), in which V is composed of nodes representing currencies and E is composed of edges representing $R[i, j]$.

Pseudocode:

Algorithm 8 Function Arbitrage(G)

for every edges(i,j) in E **do**
  $R[i, j] \Longleftarrow -\log R[i.j]$
end for
Adding an vertex v_0 to V, Adding 0-weighted edges between v_0 and any other vertex in G to E
Run Bellman-Ford algorithm starting from vertex v_0 and determine whether there is a negative cycle in G
if There is a negative cycle in G **then**
  return YES
end if
return FALSE

Correctness:

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

is equal to

$$-\log R[i_1, i_2] - \log R[i_2, i_3]... - \log R[i_k - 1, i_k] - \log R[i_k, i_1] < 0$$

the second equation is equal to a negetive cycle in G.
So if and only if Bellman-Ford finds a negtive length cycle, the currency buying chains in this cycle satisfy the equation above.

Complexity:

Time: Construction of graph G(V,E) costs $O(n + n^2)$ time because the number of edges is n^2;
Running Bellman-Ford method costs $O(nm) = O(n * n^2) = O(n^3)$ time.
So the total time complexity is $O(n^3)$.