CSOR W4231 Analysis of Algorithms I - Spring 2023

Homework 2
Tong Wu, tw2906

Problem 1

From the question, two specific nodes s and t has distance that is greater than $2/n$, and the job is to find a node v which can destroys all $s - t$ path if this node is deleted.

The BFS algorithm searches each layer to find the shortest path. In this problem, set the initial point that BFS algorithm start searching to node s, and the layer number, which is equal to the distance, $2/n$.

Pseudo Code

```
BFS_CutNode(G=(V,E), s) {
    discovered[V] = 0
    dist[V] = inf()
    parent[V] = NIL
    queue q
    discovered[s] = 1
    dist[s] = 0
    parent[s] = NIL
    enqueue(q, s)
    // Initialise the number n to the absolute number of vertices V
    n = abs(len(V))
    // Initialise the layer array
    layer = []
    layer[0] = s
    while size(q) > 0:
        u = dequeue(q)
        // Find all connected node to u
        for i=1 in range(len(layer)):
            layer[i] = find_connected(u)
        end for
        for u, v in E:
            if discovered[v] == 0 then
                discovered[v] = 1
                dist[v] = dist[u] + 1
                parent[v] = u
                enqueue(q, v)
            end if
        end for
    end while
30      // Find the node that only contains one connected component
31      for u in range(G):
32          if find_distance(layer[u]) < n/2 && is_destory(layer[u])
33              return u
34              break
35          end if
36      end for
37  }
```

In the pseudo code, set an array to contain all connected nodes for the specific node u in order to determine whether it can met the condition. After the BFS algorithm, use a for loop to find the node that has distance lower than $n/2$ and can be destroyed to met the condition, finally output the node u.

Correctness

Base case:

In layers {1, 2, 3,…, 2/n}, there must has a layer which does not contain the input node s and output node t.

Induction hypothesis:

Assume that layers between node s and node t must has one layer that only contain one node.

Induction:

From the problem can know that, the distance between node s and node t is strictly greater than $n/2$, where n is the number of nodes in this graph. According to the induction hypothesis and the basic graph of a tree, if the hypothesis is false, then the total number of node must exceed V, so the only condition is that the hypothesis is true so then can met the total number is V. Hence, there must has a node which met the condition, then show the correctness.

Running Time

For the BFS algorithm, the running time is $\mathrm{O}(n+E)$, where n is the number of nodes, equivalent to V, and E is the number of edges, since the worst cast for BFS is that the algorithm needs to discover all edges and visit all nodes. Then, in search the cut node, the running time should be $\mathrm{O}(n)$ for the last for loop start from line 31, since the graph has n nodes. Hence, the total running time should be:

$$T(n) = \mathrm{O}(n+E) + \mathrm{O}(n) = \mathrm{O}(n+E)$$

Problem 2

In this problem, the cycle which has odd length need to be found. From the lecture slides can know that, if a graph contains an odd-length cycle, then it is not 2-colourable, which means that two endpoints must not has the same colour. In this case, we can visit all nodes and print colour for each nodes and nodes has edge will not be the same colour. If the parent node and the current node has same colour, then the odd-length cycle found.

Pseudo Code

```
bool isOdd(G(V,E)){
    s = V[0]
    discovered[V] = 0
    colour[V] = 0
    queue q
    enqueue(q, s)
    parent[s] = NIL
    while (q):
        u = dequeue(q)
        // For all nodes connected to u
        for i in range(G[u])
            // If this node is not visited, nor be tagged, and the edge exists
            if (discovered[i] && E[u][i] && colour[i] == 0):
                // Mark this node as visited
                discovered[i] = 1
                // Set parent node
                parent[i] = u
                enqueue(q, i)
                // Colourise this node in a different colour
                colour[i] = 1 - colour[u]
            // If two nodes have the same colour
            else if (colour[i] == colour[u]):
                return True
            end if
        end for
    end while
    return False
}
```

In the pseudo code, a colour array is generated to store each node's colour value. In the main while loop, the condition of the loop is set to all nodes, the algorithm will visit all nodes according to their edge and colour them. Once found that current node has the same colour with the parent node, then return true, otherwise return false.

Correctness

Base case:

if a graph contains an odd-length cycle, then it is not 2-colourable.

Induction hypothesis:

Let graph G is a connected graph, and let $L_1, L_2, \ldots$ be the layers, starting with node s. Then, if the algorithm returns true, then it shows that graph G has an odd-length cycle, otherwise not.

Induction:

Assume no edge in graph G joins two nodes of the same layer of the BFS tree, then all edges in G not belonging to the BFS tree are edges between nodes in the same layer or between nodes in adjacent layers. It implies that all edges of G not appearing in the BFS tree are between nodes in adjacent layers. In the pseudo code, different colour will be marked to nodes, so the graph will be 2-colourable, then it is bipartite.

On the other hand, assume there is an edge between two nodes in the same layer, then G will not be 2-colourable since in the pseudo code, these two nodes will be marked as the same colour, which means G is not bipartite. Since two nodes in the same layer has edge, then the cycle must be odd-length, hence the correctness is proofed.

Running Time

The outer while loop will run $\mathrm{O}(V)$ times since the queue has V of nodes. The inner for loop will also run $\mathrm{O}(V)$ times for the same reason. So the total running time should be:

$$T(n) = \mathrm{O}(V) \times \mathrm{O}(V) = \mathrm{O}(V^2)$$

Problem 3

From the problem can know that, an algorithm needs to be implemented to found the minimum steps getting a bottle with A litres water from two bottles with capacity {X, Y} and initial water {x, y}. There are three different actions, FILLUP, EMPTY and POUR, each actions can targeted to two different bottles, so there is total 6 distinct actions. In this case, a graph can be imagined which has a root node with value {x, y}, then each node has 6 sub-nodes. The job is to find the node contains value A, it could be {A, y} or {x, A}. In order to find the minimum steps, the step can be seen as the number of layers in the graph, since one action can be made in each layer. So the problem can be converted to BFS algorithm. Since there is no existing graph to use BFS algorithm, then creating nodes and edges is needed in the algorithm.

Pseudo Code

```
// Define each operation and update bottles' state
def Ope(X, Y, x, y, operation) {
    switch(operation)
    {
        case "FILLUP(1)":
            return(X, y)
        case "FILLUP(2)":
            return(x, Y)
        case "EMPTY(1)":
            return(0, y)
        case "EMPTY(2)":
            return(x,0)
        case "POUR(1, 2)":
            vol = min(x, Y-y)
            return(x-vol, y+vol)
        case "POUR(2, 1)":
            vol = min(X-x, y)
            return(x+vol, y-vol)
    }
}

def WaterPouringBFS(X, Y, x, y, A) {
    // Determine the base condition
    // If initial litre met the condition
    if (x==A || y==A):
        return 0
    end if
    // If the condition is larger than the top
    if (A>X || A>Y):
        return "Impossible"
    end if

    queue q
    // 0 for the initial operation number
    q = [(x, y, 0)]
    discovered[(x,y)] = False
    // Create nodes and edges for BFS
    while q:
        // Pop the set from queue
        // n for operation number
        x, y, n = dequeue(q)
        discovered[] = False
        // Compare conditions
        if (x==A || y==A):
            return n
            break
        end if
        // Iterate through all six possible behaviors to generate the node
        //// FILLUP
        node = create_node(Ope("FILLUP(1)"))
        if (!discovered(node)):
            enqueue(Ope("FILLUP(1)"), n+1)
            discovered[node]
        end if
        node = create_node(Ope("FILLUP(2)"))
        if (!discovered(node)):
            enqueue(Ope("FILLUP(2)"), n+1)
            discovered[node]
        end if
        //// EMPTY
        node = create_node(Ope("EMPTY(1)"))
        if (!discovered(node)):
            enqueue(Ope("EMPTY(1)"), n+1)
            discovered[node]
        end if
        node = create_node(Ope("EMPTY(2)"))
        if (!discovered(node)):
            enqueue(Ope("EMPTY(2)"), n+1)
            discovered[node]
        end if
        //// POUR
        node = create_node(Ope("POUR(1, 2)"))
        if (!discovered(node)):
            enqueue(Ope("POUR(1, 2)"), n+1)
            discovered[node]
        end if
        node = create_node(Ope("POUR(2, 1)"))
        if (!discovered(node)):
            enqueue(Ope("POUR(2, 1)"), n+1)
            discovered[node]
        end if
    end while

    return "Impossible"
}
```

In the algorithm, first define a function `Ope` to clarify each action and their return values, which will be used in the main function. In the main function `WaterPouringBFS` , first will compare the variables with the base condition:

1. If the target litre is larger than the capacity, then return `Impossible`
2. If the initial litre of one of the bottle is equal to target litre, then return `0` since it is at the root layer

Then, add set to the queue and enter the while loop to create nodes and edges, then compare the current litre. Since each layer will have 6 actions, so each nodes will be created by codes. The algorithm will found whether the nodes created in last loop met the condition, if yes, will return the level as the step number. If all nodes cannot reach the condition, then return `Impossible` . In order to prevent the infinite while loop in the algorithm, use `discovered` array to determine the current litre is distinct.

Correctness

Induction hypothesis:

Assume that the algorithm can determine whether the initial condition can form a A litre bottle of water

Induction:

From the problem can know that, the goal is to found a possible set of bottles contains a specific litre of water, with six specific actions. The BFS algorithm has been proofed its correctness in previous problem and in lecture slides. Note that the algorithm explores all possible states in the state space in a breadth-first manner, which search (generate) all nodes in one layer before continues to the next layer. This means that it guarantees to find a solution if one exists, because it searches all states at a given distance from the starting state before moving on to states that are farther away. If the goal state is reachable from the starting state, BFS will eventually find it. Otherwise, if it is not possible to find the goal state, the algorithm will return impossible to the terminal, which shows the algorithm is complete. Since the correctness has been proofed.

Running Time

At the function `Ope` and the initialise phase of function `WaterPouringBFS` , the running time is $\mathrm{O}(1)$. Then, the algorithm needs to iterate all states and generate new states according to the different actions. For each of the six possible operations, a new state is generated and checked if it has been visited before, which takes $\mathrm{O}(1)$ time for each operation. If the new state has not been visited before, it is added to the queue, which takes $\mathrm{O}(1)$ time. Therefore, the total running time should be:

$$T(n) = \mathrm{O}(1) + \mathrm{O}(n^2) = \mathrm{O}(n^2)$$

Problem 4

Proof: ϕ is satisfiable if and only if no strongly connected component of $G\phi$ contains both a variable x and its negation $\neg x$

1. If G_ϕ has a strongly connected component containing both a variable x and its negation $\neg x$, then ϕ is not satisfiable

Assume that there is a graph G_ϕ that has a strongly connected component containing both a variable x and its negation $\neg x$. Then, according to the definition of SCC, that is, there must have a path from x to $\neg x$ and from $\neg x$ to x. In a directed graph, the SCC is a subnet of nodes so there must have a path between every pair of nodes in the subnet. In G_ϕ, if a SCC containing x and $\neg x$, then these two are implied by each other, which means that there must has a clause that contains both of them. Hence, ϕ cannot be satisfiable since the truth value of x and $\neg x$ cannot be the same, while satisfiable needs these two has same truth value, which is a contradiction. Therefore, ϕ is not satisfiable.

2. If no strongly connected component of G_ϕ contains both a variable x and $\neg x$, then ϕ is satisfiable.

Assume there is no SCC in G_ϕ contains both a variable x and $\neg x$, then it suffices to exhibit a satisfying truth assignment for ϕ. In order to construct the assignment, assign the truth value 1 to all literals y, which has a path from x to y, since according to the implication clause $x \implies y$, the truth value will always be 1 if y is assigned to 1, whatever the value of x is. Hence, the satisfiable of ϕ can be proofed since there is no negation can be reached.

Algorithm of 2SAT

In order to design an algorithm for 2SAT, by using the proved statement above, first found all SCCs by calling the function `SCC` mentioned in lecture slides. Then, find if there has the specific node and its negation in a same SCC, if yes, then return unsatisfiable. If not found, then store all truth values into an array with node is 1 and negation node is 0.

Pseudo Code

```
def two_sat(G_phi) {
    // Call SCC function mentioned in lecture slides "SCCS" to find all SCC
    sccs = SCC(G_phi)
    // If the SCC contains x and its negation causes contradition, return not satisfiable
    for i in range(sccs):
        if (sccs.find(x) && sccs.find(x.neg())):
            return "no"
        end if
    end for

    // Initialise array for truth assignment
    truth_assign = {}
    // Found every truth value and store them into the array, node with 1,
negation node with 0
    for scc in range(sccs):
        for i in range(scc):
            truth_assign[i] = 1
            truth_assign[i.neg()] = 0
        end for
    end for
    return truth_assign
}
```

Correctness

The correctness is proofed by proofing two statements above. The algorithm is designed according to the statements.

Running Time

For the function `SCC`, the running time is $\mathrm{O}(n+m)$, where assuming that there has n number of variables and m number of clauses. Then, the first for loop costs running time $\mathrm{O}(n)$, the second for loop costs $\mathrm{O}(n)$. Hence, the total running time should be:

$$T(n) = \mathrm{O}(n+m) + \mathrm{O}(n) + \mathrm{O}(n) = \mathrm{O}(n+m)$$