CS170–Fall 2014 — Solutions to Homework 4

Quoc Thai Nguyen Truong, SID 24547327, cs170-ig

November 21, 2014

Collaborators: Shiv Sundram, Hriday Kemburu, Michael Ross.

1. Compile on a parallel cluster

Main idea.
We will run DFS on G and get topological of a list of vertices in decreasing order. Create 2 hash table, "tableV" is to store the key as vertex, value is the number (1,2,...,n) of order compile, and "tableC" is to store the key as number (1,2,...,n) of order compile, value is the list of vertices. We will use "tableC" to compile modules in parallel later.
Now, we have the topological list of vertices, so we iterate through the list. In the loop, we use "tableV" to store the value of order compile, and use "tableC" to create a list of modules to compile in order of a key value (example: key: 3, value: [C,D,T], would compile C,D,T in parallel at the 3rd time). If the vertex is not the in hash Table "tableV", assign the value of 1 to the key of the vertex in the hash Table. Also, get the list of children of the vertex (from adjacent list) and assign the max value of [(vertex's depth + 1) , (child's depth fro hash Table if exist)] to each children in the hash table,
After the loop, we just use "tableC" to compile the modules as the value in the order as the key.
Iterate through the topological sort list of vertices. If the vertex is not the in hash Table, assign the value of 1 to the key of the vertex in the hash Table. Also, get the list of children of the vertex and assign the max value of [(vertex's depth + 1) , (child's depth fro hash Table if exist)] to each children in the hash table,
Pseudocode.

```
Line 0: Function(G):
Line 1:     sortV = run DFS on G and get topological of a list of vertices in decreasing post order
Line 2:     tableV = is the hash table , key: is vertex, value is either number 1,2,..,or n
Line 3:     tableC = is the has table, key: is number 1,2,..., or n, value is list of vertices
Line 4:     For each vertex v in sortV:
Line 5:         If v is not in the hashtable tableV:
Line 6:             assign the value of 1 to the key v in tableV
Line 7:             append v in the list of vertices to the key 1 in hastable TableC
Line 8:         listV = get the list of vertices of v from adjacency list
Line 9:         valueV = get the value of v in the tableV
Line 10:         for each vertex z in listV:
Line 11:             If z is not in tableV, assign the value of (valueV + 1) to the key z in tableV,
and append z in the list of vertices to the key "(valueV + 1) " in hastable TableC

Line 12:             Else:
```

```
Line 13:                    temp = get the max number between (valueV + 1) and (value from key
z in tableV)
Line 14:                    assign the value of "temp" to the key z in tableV
Line 15:                    append z in the list of vertices to the key "temp" in hastable TableC
Line 16:     For each key k in hash table tableC:
Line 17:            get the list of vertices from key k and complie all of them in parallel.
```

Proof of correctness.
Since we run DFS and get the topological sort list, Therefore, at line 4, we iterate the topological at each vertex v in V, so we have an invariant.
_ Invariant: At end of each loop, the algorithm will guarantee assign the number order of compile for v_i of vertex is higher than or equals to the number order of compile of vertex $v_0,\ v_1,\ \cdots, v_{i-1}$ $(0 < i)$.
_ Base Case: Vertex v is the 1st element in the topological sort list, assign the value 1 to v because there is no key "v" in the hash table. Also assign the value of 1 + 1 = 2 for the children of v. Therefore, the number order of compile of children's v is higher than v. The invariant is true for the base case.
_ Induction step, Assmume that the invariant is true at vertex v_k in the topological list, we prove that it's also true for vertex v_{k+1} topological list.
We know that all the value order of compile of v_k is higher than or equal to other $v_0, v_1, \cdots, v_{k-1}$. At vertex v_{k+1} in the topological list, we know that v_{k+1} can not be ancestor of v_k, so it means that vertex v_{k+1} can be a child of v_k or has the same depth as v_k. Therefore, if v_{k+1} is a child of v_k, the order value compile of v_{k+1} will be higher than v_k. If v_{k+1} has the same depth level as v_k , then their order value compile must be the same. Therefore, the invariant is true for k + 1, so it's true for all k. Therefore the invariant is true

_ Prove correctness property:
After finished the loop, the hash Table "tableC" in the algorithm will have all the order value of compile as the key and the list of vertices to compile as the value. The algorithm will just iterate through the hash table, and compile the modules of the increasing order 1,2, $\cdots$

Running time.

$$\boxed{T(n, E) = \Theta(n + E)}$$

Justification of running time. Let T(n,E) be the run the run time of the algorithm, and n is the of modules. E is the number of edges At line 1: run DFS on G and get topological of vertices take $\Theta(n + E)$
At line 2,3: create hash Table take $\Theta(1)$
At line 4 through 16: iterate through the topological of vertices , doing the checking take and assign the value, key in "tableV" and "tableC" (go to adjacent list) take $\Theta(n + E + 1)$
At 17 and 18: iterate through the key and compile the value as the list of module inn "tableC" will take $\Theta(n)$
Therefore,

$$T(n, E) = \Theta(n + E) + \Theta(1) + \Theta(n + E + 1) + \Theta(n)$$

$$\boxed{T(n, E) = \Theta(n + E)}$$

2. Peace for Grudgeville

Main idea. The idea is to create a graph/adjacency list on m pairs. Using 2 colors: Blue and Red to color each vertex. Run DFS and each vertex and also color the nodes with difference color than their parent (the enemy). If there is a cycle or the node has been visited, then check for the color of that node with the current color that intent to color for that node. If the node has the same color as the current color, then return False which we can't divide into 2 group. Otherwise, continue to explore. It only return True, if all the nodes has been explore.

Pseudocode.

```
Line 0: Peace([$p_0, p_1, \cdots, p_m$]):
Line 1:     G(V,E) = create a graph or adjacency list on ([$p_0, p_1, \cdots, p_m$])
Line 2:     current = Blue
Line 3:     colors = the hash table, key: vertex (person), value = either blue or red
Line 4:     person = get a source node from G
Line 5:     colors[person] = current
Line 6:     visited = hash set of vertex/person, add person to visited set
Line 7:     return Explore(person, current)

Line 8:     Explore(node, current):
Line 9:         For each person/vertex(neighbor) from Vertex node in G:
Line 10:            If person is in visited:
Line 11:                If colors[person] = current: return False
Line 12:            Else:
Line 13:                add person to HashSet color
Line 14:                If current is Red: current = Blue, Else: current = Red
Line 15:                colors[person] = current
Line 16:                Expore(person,current)
Line 17:        Return True
```

Proof of correctness.
Direct proof:
After create a graph/adjacency list on m pair and assigned the color for a person (source node in the graph), we call explore on that person, with the current color of that person. The explore function is basically doing the DFS, and we assigned the color for the child (enemy) different color than the current color. We know that DFS guarantee us that if there is current node is the node that has been visited, then we check for the color of that node with the current color.If they're the same, it means that we need to have at least 3 colors which is divided into 3 group of people. Hence, in the case they're the same color, we can't divided in two group of people, so the algorithm, will return False. If running DFS, and there are no cycle or cycles that doesn't have any adjacent nodes that have same color for the whole graph. It means, the algorithm will just return True after the for loop.
Therefore, the algorithm is true for finding whether can be divided in two peace group.

Running time.

$$\boxed{T(n,m) = \Theta(n+m)}$$

Justification of running time. Let T(n,m) be the run time of the algorithm, n people, and m pairs of people that hate each other.

Line 1: create the graph/adjacency list from m pair which take $\Theta(m)$
Running DFS on a source node (line 8, Explore function) will take $\Theta(m+n)$
Doing some checking, computation,... which take $\Theta(1)$ Therefore,

$$T(n,m) = \Theta(m) + Theta(m+n) + \Theta(1)$$

$$\boxed{T(n,m) = \Theta(n+m)}$$

3. Model checking

Main idea. We have n different states, and 16 inputs.
In order to test the program, we need to call update on every single n state with 16 inputs. Therefore, using DFS, at every state, there will be a loop with 16 input, calling update on each of these inputs, and get the new state and "o" color of light in each direction. Return True if doubleGreen(o) to see it's exist green light at 2 perpendicular directions. The new state will be call recursive on explore if it hasn't been visited and return a result True of False of that new state, if it's True and the program will return True which is the System is fail. Therefore, If the system is good, the program will return false, and it means it has to check every single n different state which every 16 inputs, which is total of 16n edges.

Pseudocode.

```
Line 0: Function($s_0$):
Line 1:     visited = is the HashSet, add $s_0$ to visited
Line 2:     return Explore($s_0$)

Line 3: Explore(s):
Line 4:     for i:= 0 to 15 (16 inputs):
Line 5:         t,o = update(s,i)
Line 6:         If doubleGreen(o) is True:
Line 7:             print(s)
Line 8:             return True
Line 9:         If t is not in visited:
Line 10:            add t to visited set
Line 11:            rc = explore(t, visited)
Line 12:            If rc is True:
Line 13:                print(s)
Line 14:                return True
Line 15:    return False
```

Proof of correctness.
Direct proof:
We basically doing the DFS. We know that there are n different states , and 16 inputs. Running the DFS, we know for sure that it will reach out all of the states and $16n$ edges, and at each state we check for the doubleGreen(o) (o from the output of update) to make sure whether double green line at two perpendicular way. Therefore, if any of the state/vertex has the result of doublGreen is true, the algorithm will just terminate the problem and return True. Otherwise, it will reach out all possible n states and return False if none of these doubleGreen result of n states return True. Therefore the algorithm is true for finding whether the program is fail at double green light at two perpendicular ways.

Running time.

$$\boxed{T(n) = \Theta(n)}$$

Justification of running time.
Let T(n) be the run time for the algorithm, and n is the number of different possible states.

We know that there are n different states or vertices, and 16 input for each of the state or edges. Doing DFS, we reach all the vertices and edges. Therefore, the total run time would be $\Theta(16n+n) = \Theta(17n) \Rightarrow \boxed{T(n) = \Theta(n)}$

4. Disrupt the terrorists

(a) Running DFS will give us pairs of (pre,post) order for each vertex in graph G. We will get the topological list of vertices in decreasing post order. Because vertex r is the root, it must be the 1st element in the topological list, so we just remove it from the topological list.
Now, we have a topological list without vertex r. Create a "curr" which store the pair (pre,post) of a vertex, set the pair of the 1st vertex in the topological list to "curr". Create the "counter" which count the number of connected, set it to 0.
Iterate through the list (go through every vertex in topological list), get the pair (pre, post) from a vertex. If the pair is not a sub-interval of "curr", then increase "counter" to 1, and set "curr" = to the pair(pre,post) that we get from the vertex.
When the loop is finished, we just return "counter" which will be the numner of connected components after remove vertex r.

(b) There are 2 cases when we remove v:
_case 1: the descendants of v does connect to v or doesn't connect to v. When we remove v, we can just call $f_a(v)$(function from part(a)) to compute the connected components on that sub-graph.
_case 2: when we remove v, all the ancestor nodes of v will be a new sub-graph, so we just count it as a strongly connect component graph on the sub-graph which is 1.
Therefore, $\boxed{f(v) = 1 + f_a(v)}$, f_a is the function from part a

(c) the idea to find the depth for each vertex in V is similar to problem 1 of this homework.
We will get the topological of a list of vertices in decreasing post order by running DFS.
Create a hash table with key is vertex, and value is the depth level for the vertex.
Iterate through the topological sort list of vertices. If the vertex is not the in hash Table, assign the value of 1 to the key of the vertex in the hash Table. Also, get the list of children of the vertex and assign the max value of [(vertex's depth + 1) , (child's depth fro hash Table if exist)] to each children in the hash table,

(d) There are 3 cases when we remove v:
_ case 1: If no descendants of v connect to any ancestor of v, in this case $up(w) \geqslant d(v)$. Therefore removing v will just create a new sub-tree, so it becomes a strongly connected component.
_ case 2: If any descendant of v connect to an ancestor of v, means that y is the ancestor of v. Therefore $up(w) < d(v)$
, so w is strongly connect component with ancestor of v.
_ case 3: If w doesn't have any connected to graph of w, it means that $up(w) = \infty$. Therefore, it's count as a new strongly component.
$\Rightarrow \boxed{f(v) = Size\ Of\ Set(\{up(w_i), (i = 1, \cdots, k) \mid up(w_i) \geqslant d(v)\ or\ up(w_i) = \infty\})}$

(e) **Main idea.** The idea is to have 2 function:
Explore function which doing DFS, calculate the depth for each vertex, and and the list of list of vertices that has cycles by append the explore node with all nodes that has been pre-visited and not post-visited . Example: [(A,B,C), (X, Y,Z)] $\rightarrow$ 2 cycles (A,B,C) and (X,Y,Z).
Up function, iterate through each vertices in the graph, and check if the vertex is in the list of list of cycles. If so, then the up value of that vertex would be the minimum value of the depth

of vertex in that list. Otherwise, the up value of the vertex would be max of the vertex's depth and 0.

Pseudocode.

```
Line 0: depths = hash Table , key: vertex, value: depth level
Line 1: listOfCycles = [], visited = hashSet
Line 2: Explore(v):
Line 3:     Using part c to calculate the depth of v and update to depths hash Table
Line 4:     For neighbor of v:
Line 5:         If neighbor in visited:
Line 6:             listOfCycles.append( neighbor + [all nodes that has been pre-visited and not post-visited])
Line 7:         Else: Explore(neighbor)

Line 8: Up(v):
Line 9:     For all v in V:
Line 10:        If v is in listOfCycles:
Line 11:            upValue = minimum depth $d(v_k)$ of a vertex in the list of list of listOfCycles that has v.
Line 12:        Else: upValue = max(d(v) - 1, 0)
Line 13:        print(v , depthValue)
```

Proof of correctness.
Direct proof:
In the function Explore (line 2), we basically doing DFS, get he depth of each vertex in graph, and get the list of list of vertices that has a cycle. We know for sure that DFS will guarantee us that the run time will be linear. If there is a cycle, it means that the node that we explore is already been visited, so we will append the node with the list of all nodes that been pre-visited and not post-visited (because they have not been complete explore all their children).
In the function Up (line 8), we iterate through the list of vertices in graph. If the vertex v is in the list of list of vertices that has a cycle, its up value will be the value of depth of vertex that has minimum depth in the list of vertices that has a cycle, because that vertex is close to the root of so it must be a minimum depth compare to other vertices's depths in the list of vertices that has a cycle. If vertex v is not in the list of list of vertices that has a cycle, it mean that its up value will be its parent's depth vertex which is depth of v - 1 or $d(v) - 1$. If v happend to be the child of the root of root, then up value of v would be 0. Hence, if v is not in the list of list of vertices that has a cycle, then its up value must be max of $d(v) - 1$ and 0.
Finally, we print all the vertex and it up value of the end of each iteration.
Therefore, the algorithm is true for finding up value of all vertices in the graph.

Running time.

$$\boxed{T(V, E) = \Theta(V + E)}$$

Justification of running time. let T(V,E) be the run time of the algorithm, with V vertices, and E edges
Running function Explore (Line 2) is basically doing DFS which take $\Theta(V + E)$
Running function Up (Line 8) is basically iterate through all vertices which take $\Theta(V)$.
Doing other checking, adding, hashing will take $\Theta(1)$

Therefore,

$$T(V,E) = \Theta(V+E) + \Theta(V) + \Theta(1)$$

$$\boxed{T(V,E) = \Theta(V+E)}$$

(f) From part (e), we can compute d(v) and f(v) in linear time, which also the inputs of f(v). Hence, using part (d) , we can find f(v) in linear time

5. Optional bonus problem: More traffic

Main idea. YOUR ANSWER GOES HERE
Pseudocode. YOUR ANSWER GOES HERE
Proof of correctness. YOUR ANSWER GOES HERE
Running time. YOUR ANSWER GOES HERE
Justification of running time. YOUR ANSWER GOES HERE