HW4-solutions

April 8, 2019

0.1 Homework 4

0.1.1 - Amit Makashir

0.1.2 Question 1

Devise a linear time algorithm to solve the following problem: given a directed acyclic graph G, check if G has a directed path that visits every vertex once and only once.

0.1.3 Solution:

Assume we have a graph G=(V,E).

1. Perform a **topological sort** on the directed acyclic graph and store it in a list L.
2. Initialize i=2
3. Repeat the following steps for i <= V
   a. If L[i-1] is not the parent of L[i]; return False
   b. i++
4. If the above steps returned False, we don't have a path that visits every node
5. If the above steps returned nothing, we have a path that visits every node

```
Input: Adjacency list of the graph G=(V,E)

def topologicalSort(graph):
    '''
    Call DFS(s) to compute finishing times for each node/vertex.
    Reverse sort the according to the finishing times and
    return this list
    '''

def isPath(graph):
    sorted_graph = topologicalSort(graph)
    n = len(sorted_graph)

    # As indexing starts from 0 and we want to start from the second node
    for i in range(1,n):
        v = sorted_graph[i] # get the current node
        u = sorted_graph[i-1] # get the previous node in topological sort
```

```
        u_children = graph[u] # get the children of previous node

        if v not in u_children:
            return False

    return True
```

Time analysis: The Topological sort takes $O(V + E)$. After topological sort, we iterate through every node once and search in adjacency list which should be $O(V^2)$ or $O(E)$. So the algorithm is $O(V + E)$ or $O(E)$, which means the algorithm is linear wrt to edges.

0.1.4 Question 2

A feedback edge set of an undirected graph G=(V, E) is a subset of edges E′ belongs E that intersects every cycle of the graph. Thus, removing the edges in E′ will render the graph acyclic. Give an efficient algorithm for the following problem: Input: Undirected graph G=(V, E) with positive edge weights w_e Output: A feedback edge set E belongs to E of minimum total weight

0.1.5 Solution

1. Store all the vertices in a list V and edges in list E
2. For every edge "e" in list E:
    a. say "e" connects vertices u and v
    b. create a temp list of edges without this edge "e" in E_temp (E_temp = E - e)
    c. Use Dijkstra's algorithm to find the shortest path from u to v using E_temp
        i. If such a path exists, let's call it "P", it means that there was a cycle here with edge e. So the minimum edge in this cycle will be: min("e",min(edges in path "P")) We delete this edge from the graph and store it in E_star. ii If there was no path, there was no cycle here.
3. After deleting all the edges from cycle we now have E_star with the edges having minimum weights required to make the graph acyclic

```
Input: Adjacency list of the graph G=(V,E)

def dijsktra(u,v,edges):
    '''
    Dijsktra would return the shortest path from u to v using the input edges
    If there is a path, return the min(edges in the path)
    Else, return -1
    '''

def minCycleEdges(graph,edges):
    min_edges = []
    i = 0
    while True:
        edge = edges[i]
```

```python
        # Removes this edge from the edges and return the remaining edges
        edges_temp = removeEdge(edges,edge)
        u = edge[0]
        v = edge[1]

        min_edge = dijsktra(u,v,edges_temp)

        if min_edge != -1: # There is a path from u to v
            min(min_edge,edge)
            min_edges.append(min_edge)

        i += 1
        if i > len(edges)-1: # If i is more than the edges list, break the loop
            break


    return min_edges
```

0.1.6 Question 3

A bipartite graph is a graph G=(V, E) whose vertices can be partitioned into two non-overlapping sets (V= V1 union V2 and V1 intersects V2 = empty) such that there are no edges between vertices in the same set. Devise a linear-time algorithm to determine whether an undirected graph is bipartite.

0.1.7 Solution

1. Create a "partition" dictionary/hashmap that stores the vertex "v" as the key and it's class (I or II) as the value.
2. Insert s in the "partition" dictionary as {"s":"I"}.
3. We start BFS from a vertex "s".
4. Find all the child/neighbors nodes for "s" from adjacency list:
   i. If this node was present in the partition dict with the same class as it's parent's; return False
   ii. else, put it in the partition dictionary with the second class (different class than parent's). for eg: "u" is a child of "s", then partition = {"s":I,"u":"II"}
5. After inserting in the partition dict, insert these neighbors in a queue "Q" and continue BFS
6. Dequeue from "Q" and repeat step 4 with the dequeued node

Input: Adjacency list of the graph G=(V,E) and initial node "s" Output: True/False

```python
def isBipartite(graph):
    partition = {}

    ''' We will initialize a dict/hashmap that stores the vertex "v" as the key
    and it's class (I or II) as the value    '''
    for "s" in graph:
```

```
        if "s" not in partition:
            partition["s"] = 1
            q = queue.Queue()
            q.put(s) # Push "s" to the queue

            while q.qsize > 0: # queue is not empty
                u = q.get() # extract an element from the queue

                for each v in graph[u]: # exploring neighbors of u
                    if partition[v] exists and partition[v] == partition[u]:
                        # v is present in partition
                        return False
                    else:
                        partition[v] = partition[u]*(-1)
                        # -1*-1=1; 1*-1=-1 (Always give the opposite class)

    return True
```

Time Analysis: The total running time for BFS is $O(V+E)$. The only difference is we are checking in a dictionary if vertex was visited and the class of the vertex which is $O(V)$. Therefore, our algorithm is $O(V+E)$.

0.1.8 Question 4

Suppose a CS curriculum consists of n courses, all of which are mandatory. The prerequisite graph G has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w. Devise an algorithm that works directly on this graph representation to compute the minimum number of semester necessary to complete the curriculum (assuming that a student can take any number of courses in a semester). The running time of your algorithm should be linear.

0.1.9 Solution:

Assume we have a graph G=(V,E).

1. Perform a **topological sort** on the directed acyclic graph and store it in a list L.
2. Find all the nodes with in-degree 0 and store in a list start_nodes
3. Start BFS for each node in start_nodes
4. Store the max-distance/level for each of these BFS in list no_of_semesters
5. max(no_of_semesters) is the minimum number of semesters necessary to complete the curriculum

```
Input: Adjacency list of the graph G=(V,E)

def createDict(graph):
    '''
    Iterate through all the courses and create a dictionary/hashmap
    with them and initialize the courses to 1
    Initially we are thinking of scheduling all courses in sem 1
```

```
    for eg: sem_to_enroll = {"A":1,"B":1,"C":1}
    return the dict courses
    '''

def noOfSem(graph):
    sem_to_enroll = createDict(graph)

    sorted_graph = topologicalSort(graph)
    n = len(sorted_graph)

    # As indexing starts from 0 and we want to start from the second node
    for i in range(n):
        u = sorted_graph[i] # get the current node
        u_children = graph[u] # get the children of previous node

        for v in u_children:
            sem_to_enroll[v] += 1

    # sem_to_enroll will have courses and the semester they should be enrolled in
    # Semester number = 1 + (no. of in-degree to this node)
    # Find the maximum value in sem_to_enroll which should take linear time
    no_of_sems = max(sem_to_enroll.values)
    return no_of_sems
```

Time analysis: Topological sort is $O(V+E)$. For every node in topological sort we are looping through all its children and incrementing. This is $O(V^2)$ or $O(E)$. Therefore, our algorithm is $O(V+E)$

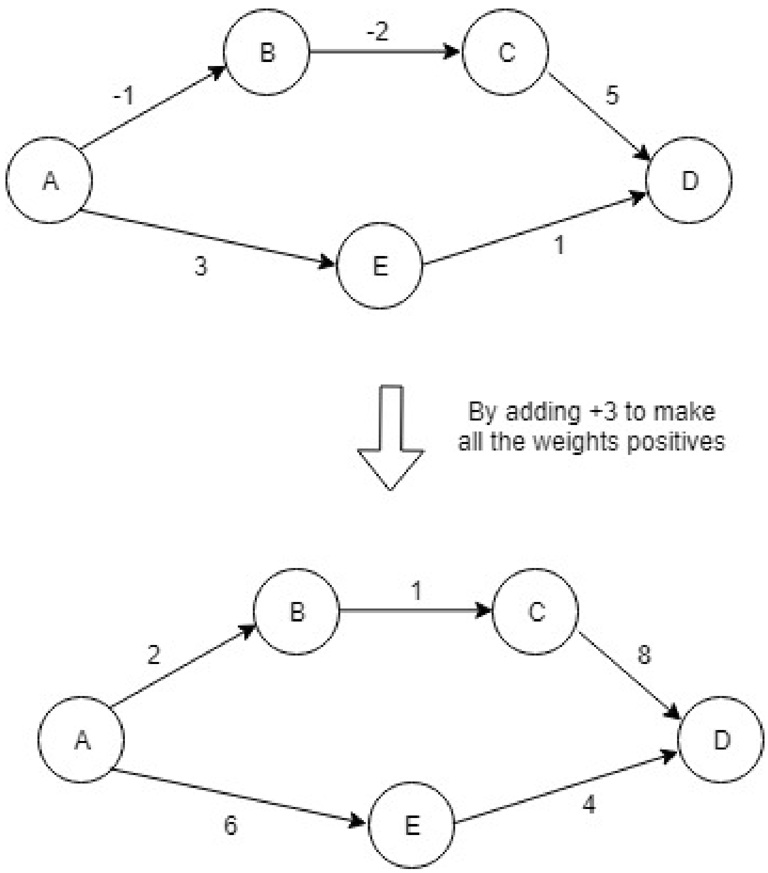
0.1.10 Question 5

Professor Luke suggests the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node s, and return the shortest path found to node t. Is this a valid algorithm? If yes, prove it is correct. Otherwise, give a counterexample.

0.1.11 Solution:

The algorithm is not valid. Following is a counter example for this algorithm:

In the example, the shortest path from A to D should should be A-B-C-D. But after adding large constant we A-E-D as the shortest path which is clearly wrong.



0.1.12 Question 6

Devise an algorithm that takes as input a direct graph with positive edge lengths, and returns the length of the shortest cycle in the graph (if the graph is acyclic, it should say "no cycle". Your algorithm should take time at most $O(|V|^3)$.

0.1.13 Solution

1. Store all the vertices in a list V and edges in list E
2. For every edge "e" in list E:

    a. say "e" connects vertices u and v
    b. create a temp list of edges without this edge "e" in E_temp (E_temp = E - e)
    c. Use Dijkstra's algorithm to find the shortest path from u to v using E_temp

        i. If such a path exists, let's call it "P", it means that there was a cycle here with edge e. Calculate length of this cycle and store in list cycle_lengths
        ii If there was no path, there was no cycle here.

3. if len(cycle_lengths) > 0 ; find the min(cycle_lengths) and return it else; return "no cycle"

```
Input: Adjacency list of the graph G=(V,E), edges is list of tuples of (from_node,to_node)

def dijsktra(u,v,edges):
    '''
```

```
    Dijsktra would return the shortest path from u to v using the input edges
    If there is a path, return the sum(edges in the path)
    Else, return -1
    '''

def shortestCycle(graph,edges):
    cycle_length = []

    for each edge in edges:
        edges_temp = removeEdge(edges,edge)
        u = edge[0]
        v = edge[1]

        shortest_path = dijsktra(u,v,edges_temp)

        if shortest_path != -1: # There is a path from u to v
            cycle_length.append(shortest_path)

    return min(cycle_length)
```

Time analysis: Dijsktra's has running time of $O(Vlog(V) + E)$ by implementing the min-priority queue with a Fibonacci heap. As we are iterating through every node and using Diksktra's our running time is $O(V.E)$ or $O(V^3)$

0.1.14 Question 7

You are given a set of cities, along with the pattern of highways between them, in the form of an undirected graph G=(V, E). Each stretch of highway eÎE connects two of the cities, and you know its length in miles, Le. You want to get from city s to city t. There is one problem: your car can only hold enough gas to cover L miles. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length Le č L. a) Given the limitation on your car's fuel tank capacity, show how to determine in linear time whether there is a feasible route from s to t. b) You are now planning to buy a new car, and you want to know the minimum fuel tank capacity (in terms of the miles in coverage without re-fueling) that is needed to travel from s to t. Devise an O((|V| + |E|) log |V|) algorithm to determine this. (Hint: you may modify Dijkstra's algorithm).

0.1.15 Solution:

a)

1. Start BFS from the nodes "s"
2. Only put nodes whose edge weights w<=L in the Queue for this BFS
3. After Dequeuing, first check if this node was "t" and then continue with the BFS

```
Input: Adjacency list of the graph G=(V,E), edge weights "E", initial node "s", final node "t"
    and L fuel tank capacity
Output: True/False
```

```python
def isFeasible(graph,E,s,t,L):
    q = queue.Queue()
    q.put(s) # Push "s" to the queue

    visited = {}

    while q.qsize > 0: # queue is not empty
        u = q.get() # extract an element from the queue

        visited[u] = 1 # Mark this node as visited

        if u == t: # Goal state reached
            return True

        for each v in graph[u]: # exploring neighbors of u
            # If the distance between cities is less than fuel tank capacity
            if E(u,v) <= L:
                if visited[v] does not exists:
                    q.put(v) # Push "v" to the queue

    return False
```

Time analysis: As we are using BFS with just one additional condition our running time is $O(V+E)$ or in other words linear wrt edges.

b)

1. We can use Dijkstra's algorithm by modifying the parameter to be optimized
2. Instead of optimizing the total distance from source "s" to a node, we are only interested in optimizing individual edge weights/segments of the path

```
Input: Adjacency list of the graph G=(V,E), edge weights "E", initial node "s", final node "t"
Output: min. Fuel tank capacity required to go from "s" to "t"
```

```python
def minFuelTank(graph,E,s,t):
    '''
    elements in priority-queue => (priority,node,max_edge)
    priority => edge weight of the current node. for eg: if the current node is "v"
                and it's parent is "u" then priority is equal to E(u,v)
    node => node is the current node
    max_edge => say the current node was explored via the path u-x-y-w-v.
                This parameter is the max. edge encountered in this path
    '''
    q = queue.PriorityQueue()
    q.put((0,s,0)) # Push (edge-weight/priority,node) to the priority-queue

    while q.qsize > 0: # queue is not empty
        u = q.get() # extract an element from the priority-queue
```

```
        if u == t: # Goal state reached
            return u[2] # The max edge weight would be the minimum fuel tank capacity needed

        for each v in graph[u]: # exploring neighbors of u
            dist_uv = E(u,v)
            max_edge = max(dist_uv,u[2])

            # We search in the priority queue for this node
            # Only insert in the queue if the current max edge is lesser
            # than previously inserted max edge
            if q[v][2] > max_edge:
                q.put((dist_uv,v,max_edge)) # Push (edge-weight/priority,node) to the priority

    return False
```

Time analysis: As we using Dijsktra's with just one additional condition, our running time is $O((V+E)log(V))$.