MPCS 55001 Algorithms Autumn 2023
Homework 6

Zhiwei Cao zhiweic
Collaborators: Student A, Student B

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.**

Collaboration policy: You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**.

Internet sources: You must include the url of any internet source in your homework submission.

1 Shortest Paths with Revisiting (10 points)

Given a directed weighted graph $G = (V, E, w)$, a source vertex s in V, and a subset $X \subset V$ of the vertices V, find the shortest (min-weight) paths from s to all other vertices in G where you must visit a vertex in X **at least once** every 3 vertices. Assume all edge weights $w(e)$ are nonnegative.

Requirement: Do not modify the algorithm; modify the input data.

Give an efficient algorithm to solve this problem. Your algorithm should run in $O((V + E) \log V)$ time.

(a) (5 points) Write **pseudocode** for your algorithm. Your algorithm should output the predecessors π of each vertex in a shortest path tree and the distance d of each vertex from the source s.

(b) (4 points) Prove that your algorithm is correct. Let Pv be a shortest path from s to v which visits X at least once every 3 vertices, in the original graph. Argue that Pv is represented by a shortest path (with no constraints) in the modified graph G'. Furthermore, argue that your algorithm computes d[v] correctly for each $v \in V$.
1. incorrect shortest path: The algorithm finds a path from s in G that its output as Dijkstra_Three_One(G, s, X), but there exists a shorter one
2. algorithm computes d[v] correctly for each $v \in V$
Proof for 1: Any shortest path Pv' in G' corresponds to path Pv in G that follows the same sequence of original vertices. the weight of the edges in G' also have the same weight in G. Since the algorithm split each vertex into v_0, v_1, and v_2 and store in the G', and they have same attribute as the vertex in G. Let Pv be the path found by the algorithm from s in G that visits a vertex in X at least once every three vertices from s in G. Since the path Pv contains several vertices in G, and there must have vertices v_i in G' correspond to these vertices in G.
Dijkstra correction: The path in G' ensures the p' must be the shortest path with the min-weight in G' starting from s. If p is shorter then G's is shorter (path) and Dijkstra will find it $\rightarrow$ provides the **contradiction**.
Proof for 2: The new graph G' has 3 different vertices corresponding to the vertices in G, where are v_0, v_1, and v_2. At the same time, $d[v_0]$ stands for the shortest path to v in G, where v is in X, $d[v_1]$ stands for the shortest path to v in G where the vertex not in X, but the path just visited a vertex in X for the last one step, and $d[v_2]$ stands for the shortest path to v in G where the vertex not in X, but the path just visited a vertex in X for the last two steps. The minimum value of these three vertices makes sure the shortest path follows the question requirement which will visit the one vertex in X every three steps.

(c) (1 point) Justify the $O((V + E) \log V)$ running time of your algorithm.
New graph G' construction takes $O(v + E)$ and Dijkstra takes $O((V' + E') \log V')$
Conclusion: Overall running time is $O((V + E) \log V)$

Algorithm 1 1 (a): BFS to find the path with minimum weight

Reference: from lecture week 6 lecture example
function Dijkstra_Three_One(G, s, X)
Input: a directed weighted graph $G = (V, E, w)$
Input: a source vertex s in V
Input: a subset $X \subset V$ of the vertices V
Output: return the shortest (min-weight) paths from s to all other vertices in G where you must visit a vertex in X **at least once** every 3 vertices.
build a new graph G', and output the predecessors π of each vertex in a shortest path tree and the distance d of each vertex from the source s.
for each v of the G.V **do**
    for $i \leftarrow 0$ to 2 **do**
        add v_i to G'
    end for
end for
// construct the edge
for each e in G.E **do** // e = (u,v)
    if e.u in X **then**
        // since the v0 is defined as a vertex that is in the X or just visited a v of X
        add the edge e(u0, v0, w(u,v)) to the graph G'
        add the edge e(u1, v0, w(u,v)) to the graph G'
        add the edge e(u2, v0, w(u,v)) to the graph G'
    else
        // inverse version
        add the edge e(u0, v1, w(u,v)) to the graph G'
        add the edge e(u1, v2, w(u,v)) to the graph G'
        if e.v in X **then**
            add the edge e(u2, v0, w(u,v)) to the graph G'
        end if
    end if
end for
// finish graph construction
Dijkstra(G', s)
get Π' d' from Dijkstra(G', s)
$d \leftarrow anarray$ // to store the shortest path base on the G'
for each v in G.V **do**
    $d[v] \leftarrow min(d'[v0], d'[v1], d'[v2])$
end for
return d

2 Pay to Win (13 points)

Your friend has brought a new board game to game night. Pay to Win consists of a rectangular $n \times m$ grid and each square contains a positive number of gold coins. When you land in a square, you take the number of gold coins in it. You start from the top left with the gold coins in that square. In each round you choose a square that is further down and to the right to move to. For that move you need to pay gold coins equal to the square of the distance you traveled. For example, moving from (x_1, y_1) to (x_2, y_2) requires $g = (x_2 - x_1)^2 + (y_2 - y_1)^2$ gold coins. Obviously you cannot pay more gold coins than the ones you have. Your goal is to get to the bottom right in as few rounds as possible.

Table 1 shows two optimal paths of length four highlighted in red.

1	1	2	5
1	3	1	8
5	3	4	1
5	3	8	2
1	4	2	1

1	1	2	5
1	3	1	8
5	3	4	1
5	3	8	2
1	4	2	1

Table 1: Example problem on a 4×5 grid. There might exist multiple shortest paths.

Write an efficient **dynamic programming** algorithm that determines the fewest moves necessary to get to the bottom right square. Maximum points for most efficient correct algorithms.

(a) (2 points) Define the subproblem(s). Define each variable you introduce.
Define a 2 dimensions table $DP[x][y]$ to store the minimum number of steps to reach the current cell $grid[x][y]$ by the direction to the bottom right square from top-left cell $grid[0][0]$ and ensure the coins are enough to pay the move. x stands for the index of the row in the grid, x' stands for the index of the row of the cell that is the last cell visited in the grid, y stands for the index of the column in the grid, y' stands for the index of the column of the cell that is the last cell visited in the grid

(b) (2 points) Give a recurrence which expresses the solution to each subproblem in terms of smaller subproblems. State any base case(s). Justify your recurrence.
$DP[x][y] = min_{0 \le x' \le x, 0 \le y' \le y}(DP[x'][y'] + 1)$ where $grid[x'][y'] \ge (x - x')^2 + (y - y')^2$
Basis Case: DP[0][0] = 1, start from top-left it needs 1 step to reach.
The recurrence relationship is based on the previous cells (x', y') that can reach the current cell (x, y), at the same time, it has to ensure that the player has enough coins to make this movement.

(c) (2 points) Write **pseudocode** for your algorithm.

(d) (1 point) State and justify the time complexity of your algorithm.
Time complexity is $O(n^2m^2)$, since DP takes $O(nm)$ time, and for each cell of DP table, it generally takes $O(nm)$ time. In conclusion, the time complexity is $O(n^2m^2)$.

(e) (1 point) Describe how to modify your algorithm to recover this shortest path.
Define a new list P to store the path, when the $DP[x][y]$ is updated because of the recurrence relationship, which means the algorithm finds the cell that could update the DP table, append the current value or the location of $grid[x][y]$ into the P. When the algorithm is finished, it will also find the shortest path by P

Algorithm 2 2 (c): DP Pay to Win

function DPPayToWin($grid$)
Input: a rectangular $n \times m$ grid and each square contains a positive number value.
Output: return the fewest moves necessary to get to the bottom right square
create a 2D table DP with the size $n * m$
create a 2D table $coinSaved$ with the size $n * m$
$DP[0][0] \leftarrow 1$
$coinSaved[0][0] \leftarrow grid[0][0]$
for $x \leftarrow 0$ to $n-1$ **do**
    for $y \leftarrow 0$ to $m-1$ **do**
        if $(x, y) == (0, 0)$ **then**
            continue // skip this step, since the DP[0][0] already defined
        end if
        for $x' \leftarrow 0$ to x **do**
            for $y' \leftarrow 0$ to y **do**
                $stepCost \leftarrow (x - x')^2 + (y - y')^2$
                if $grid[x'][y'] \geq stepCost$ **then** // make sure player at least can pay for current step
                    if $DP[x][y] - DP[x'][y'] > 1$ or $(DP[x'][y'] + 1 == DP[x][y] and coinSaved[x'][y'] + grid[x][y] - stepCost > coinSaved[x][y]$ **then**
                        // if DP[x][y] - DP[x'][y'] > 1 means it find a better path; if $== 1$, need to check if the money saved enough for this step
                        $DP[x][y] \leftarrow DP[x'][y'] + 1$
                        $coinSaved[x][y] \leftarrow coinSaved[x'][y'] + grid[x][y] - stepCost$
                    end if
                end if
            end for
        end for
    end for
end for
return $DP[n-1][m-1]$

This problem can also be solved using a graph algorithm.

(f) (4 points) Write **pseudocode** for an efficient algorithm to construct a graph from the input grid and return a path using the fewest moves needed to get to the bottom right square.

Algorithm 3 2 (f): Graph Pay to Win

```
function GraphPayToWin(grid)
Input: a rectangular n × m grid and each square contains a positive number value.
Output: return a path using the fewest moves needed to get to the bottom right square.
Reference from: https://chat.openai.com/share/3c638917-d19f-48c6-9fc7-e3a02cd8a369
Q ← ∅
Black ← anemptyset // store the cells that already visited
Π ←
ENQUEUE(Q, ((0, 0), [(0, 0)]) // store the start location, and the path
P ← [] // return value
while Q ≠ ∅ do
    girdLoc, path = DEQUEUE(Q)
    add girdLoc to Black
    if gridLoc == (n − 1, m − 1) then // reach the end, return condition
        P ← path
    end if
    adjacentCells ← []
    for i ← 0 to n do
        for j ← 0 to m do
            if (i > girlLoc[0] or j > grilLoc[1]) and grid[girlLoc[0]][grilLoc[1]] ≥ (i − girlLoc[0])^2 + (j − girlLoc[1])^2 then // reference to 2(b): grid[x'][y'] ≥ (x − x')^2 + (y − y')^2
                adjacentCells.append((i, j)) // store the reachable cells based on current cells
                // construe the adjacent list format
            end if
        end for
    end for
    for each in v in adjacentCells do
        if v not in Black then
            ENQUEUE(Q, (v, (path + [v])))
            add v to Black
        end if
    end for
end while
return P
```

(g) (1 point) State and justify the time complexity of your algorithm.// It is a BFS approach, then the time complexity is $O(V + E)$, V is the number of vertices, and E is the edges. Since we have n * m cells, so the $|V|$ = (nm). The edges of the graph are dependent and much less than the number of cells, because the algorithm always goes to the bottom right.
In conclusion: Time complexity is $O(V + E)$

3 Project Scheduling (14 points)

Scheduling is an interesting area which is the source of many classic algorithms problems. Suppose we want to build a schedule for a project given the following information:

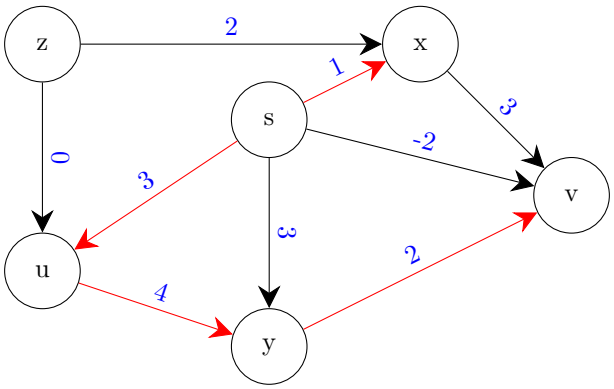
- Tasks and their estimated durations
- Dependencies between the tasks

For example, on a construction project, an electrician may need to pull cables and install electrical boxes before a carpenter can hang drywall.

3.1 DAG Longest Paths

Consider the problem of finding the single-source **longest** paths in a **weighted directed acyclic graph**.

For example, a weighted DAG is shown below. Longest paths from the node s to all other vertices are highlighted in **red**.



Note that some vertices may not be reachable from the source vertex. In the example above, we would consider z to have distance ∞ from s.

(a) (4 points) Write **pseudocode** for `DAGLongestPaths`, an algorithm which solves this problem. It should take as input a weighted DAG $G = (V, E, w)$ and a source vertex s and return the longest path distances d and parents π representing the longest path tree for all vertices in V. Your algorithm should run in $O(V + E)$ time.

Since the Topological sort is based in DFS, so the time complexity is $O(V + E)$, and for Lines 13-21, it checks all vertices from SortedList which takes $|V|$, and all adjacent vertices which take $|E|$. It takes $O(|V| + |E|)$ in sum.
In conclusion: The time complexity is $O(V + E)$

Algorithm 4 3 (a): DAGLongestPaths

```
1: function DAGLongestPaths(G, s)
2: Input: a weighted DAG G = (V, E, w)
3: Input: a source vertex s
4: Output: return the longest path distances d and parents π representing the longest path tree for all vertices in V
5: SortedList ← sorttheDAGbythetopologicalSort
6: // Reference: https://www.geeksforgeeks.org/topological-sorting/
7: initialize-single-source(G,s) // Reference from week 6 lecture note
8: for each vertex v in V do
9:     d[v] = −∞ // since we are long for the longest path
10:    Π[v] = NIL
11: end for
12: d[s] ← 0
13: longestD ← 0
14: for each vertex u in SortedList do
15:    for each vertex v in Adj[u] do
16:        // RELAX(u,v,w) // Reference from week 6 lecture note
17:        if d[v] < d[u] + w(u, v) then // since we are long for the longest path
18:            d[v] ← d[u] + w[w(u, v)]
19:            longestD ← max(longestD, d[v])
20:            Π[v] ← u
21:        end if
22:    end for
23: end for
24: return Π, d
```

3.2 Critical Path Method

In project management, the critical path is defined as the longest sequence of tasks which must be completed in order, since each task cannot be started until the previous task is completed. This is important because any delay to a task on the critical path will delay the entire project. This problem can be modeled using a weighted DAG.

Note that the critical path is not necessarily unique. For example, consider the following schedule:

Task	Duration (days)	Dependencies
A	3	-
B	4	-
C	3	A, B
D	1	A
E	3	A
F	3	D

In this example, there are two critical paths: A, D, F and B, C. Both take 7 days. Task E is not on any critical path.

(b) (4 points) Write **pseudocode** for `CriticalPath`, a function which computes the critical path for a given set of tasks.

It should take two inputs:

- An array $T[1 \ldots n]$ of tasks as (task name, duration) pairs, where duration is an integer number of days.
- A array $D[1 \ldots m]$ of dependencies as (task1, task2) pairs, where task1 must be completed before task2.

In the above example, the pairs of dependencies are $(A, C), (B, C), (A, D), (A, E), (D, F)$. You may assume that $D[1 \ldots m]$ has no dependency cycles (so it always possible to complete all tasks).

The output should contain two items: the length of a critical path (in days), as well as the sequence of tasks along some critical path. Your algorithm should run in $O(n+m)$ time, where n is the number of tasks and m is the number of dependencies. You may call your `DAGLongestPaths` algorithm from part (a).

Reference: https://www.workamajig.com/blog/critical-path-method

Algorithm 5 3 (a): CriticalPath

function CriticalPath(T, D)
Input: An array $T[1 \dots n]$ of tasks as (task name, duration) pairs, where duration is an integer number of days.
Input: A array $D[1 \dots m]$ of dependencies as (task1, task2) pairs, where task1 must be completed before task2.
Output: returns the length of a critical path (in days), as well as the sequence of tasks along some critical path.
Construct the graph
$edgeWeight \leftarrow dict()$
for each t,d in T **do**
    $edgeWeight[t] \leftarrow d$
end for
$inDegree \leftarrow [0] * T.size$
$G \leftarrow$ with the structure "$task : [dependencies]$"
for each (task1, task2) in D **do**
    $G[Task1].append(Task2)$ // add edges, adjacent list format
    $G.w(Task1, Task2) \leftarrow edgeWeight[Task2]$
    $inDegree[Task2] \leftarrow inDrgree[Task2] + 1$
end for
for each Task,d in T **do**
    if $inDegree[t] == 0$ **then**
        $G['dummyNode'].append(Task)$
        $G.w('dummyNode', Task) \leftarrow edgeWeight[Task]$
    end if
end for
d, $\Pi \leftarrow$ DAGLongestPath(G, 'dummynode')
$length \leftarrow$ max distance value of d
$lastTask \leftarrow$ the task with $length$ value
$workPath \leftarrow dict()$
call Print-Path(G, $'dummyNode'$, $lastTask$) // reference: week 5 lecture note, need to set a condition that doesn't append 'dummyNode' into the $workPath$
$workPath \leftarrow Print - Path$(G,'dummyNode',lastTask)
return $length$, $workPath$

3.3 PERT

The ideas behind the critical path method can be extended in several ways. For example, you might like to know:

- **Early start/finish (ES/EF)**: What is the earliest possible time a task can start/finish, given its dependencies?
- **Late start/finish (LS/LF)**: What is the latest possible time a task can start/finish without delaying the entire project?
- **Slack time (S)**: What is the most a task can be delayed without delaying the entire project? This is computed as $S = LS - ES = LF - EF$.

Collectively, these ideas are referred to as the program evaluation and review technique (PERT).

Here are the values for the example above:

Task	Duration	Early Start	Early Finish	Late Start	Late Finish	Slack
A	3	0	3	0	3	0
B	4	0	4	0	4	0
C	3	4	7	4	7	0
D	1	3	4	3	4	0
E	3	3	6	4	7	1
F	3	4	7	4	7	0

(c) (2 points) Prove: A task is located along some critical path **if and only if** it has a slack time of 0.

If S is not 0 it means that there is some kind of delay no mater is a start delay or a finish delay. In other words, this task caused the project delayed. As a result, it implies that the task is not in the critical path. On the contrary, if a task is in the critical path, then the task doesn't allow any delay, otherwise, it will cause the whole project delay, then the S value should be 0.

(d) (4 points) Write **pseudocode** for `ComputeSlackTimes`, an algorithm which computes the slack times for every task. The inputs are in the same format as before. The output should be a hashtable which maps tasks to their slack times. Your algorithm should run in $O(n + m)$ time, where n is the number of tasks and m is the number of dependencies. You may call your algorithms from parts (a) and (b).

Algorithm 6 3 (d): ComputeSlackTimes

function ComputeSlackTimes(T, D)
Input: An array $T[1 \dots n]$ of tasks as (task name, duration) pairs, where duration is an integer number of days.
Input: A array $D[1 \dots m]$ of dependencies as (task1, task2) pairs, where task1 must be completed before task2.
Output: return a hashtable which maps tasks to their slack times.
$slackTime \leftarrow dict()$ // returned hash table
G have the same construction steps as 3(b) criticalPath
d, $\Pi \leftarrow DAGLonestPats(G, 'dummyNode')$
$pathLen$, $path \leftarrow CriticalPath$(T,D)
// d = 'dummyNode': 0, 'A':3, 'B':4, 'C':7, 'D':4, 'E':6, 'F':7
$ES \leftarrow dict()$ // Early Start
$EF \leftarrow dict()$ // Early Finish
$LS \leftarrow dict()$ // Late Start
$LF \leftarrow dict()$ // Late Finish
for each task,time of d **do**
    if $task \neq' dummyNode'$ **then**
        $EF[task] \leftarrow time$
        $ES[task] \leftarrow time - edgeWeight[task]$ // edgeWeight is same as 3(b) store the duration of the task
        if $G[task] == NIL$ **then** // leaves node, the task that doesn't have dependencies
            $LS[task] \leftarrow pathLen - edgeWeight[task]$
            $LF[task] \leftarrow pathLen$
        else
            $LS[task] \leftarrow time - edgeWeight[task]$
            $LF[task] \leftarrow time$
        end if
    end if
end for
// transpose the graph G, takes O(V+E)
build a new graph G_T
for each u in G.V **do**
    add u to $G_T.V$
end for
for each u in G.V **do**
    for each v in G.Adj[u] **do**
        add u to $G_T.Adj[v]$
    end for
end for
//My original idea is to use the transposed graph to construct the DAG from the leaves node to find the LF LS, but it will over the required time complexity. That is the only idea I have.
for each task in G **do**
    $slackTime[task] \leftarrow LF[task] - EF[task]$ // = LS[task]-ES[task] as well
end for
return $slackTime$

4 Programming: Dial's Algorithm (20 points)

Follow this GitHub Classroom link to accept the assignment. Your code should be pushed to GitHub; you do not need to include it here.