PS8

221300066 季千🄵

2024 年 6 月 12 日

1 Problem1

(a) 每次循环后的 dist 和 parent 值以及已知区域 R 中的顶点的值：

Iteration	dist(s)	dist(t)	dist(x)	dist(y)	dist(z)	parent(t)	parent(x)	parent(y)	parent(z)	R
0	0	∞	∞	∞	∞	None	None	None	None	{s}
1	0	10	5	3	∞	s	s	s	None	{s, y}
2	0	8	5	3	9	y	s	s	y	{s, y, t}
3	0	8	5	3	7	y	s	s	x	{s, y, t, x}
4	0	8	5	3	6	y	s	s	y	{s, y, t, x, z}

第四次循环后，所有的顶点都在 R 中，算法结束。

(b) 初始化所有顶点的 dist 和 parent 值：

顶点	dist	parent
t	∞	null
x	∞	null
y	∞	null
z	0	null
s	∞	null

第一轮松弛后的 dist 和 parent 值如下：

顶点	dist	parent
t	11	s
x	2	z
y	8	s
z	0	null
s	7	z

第二轮松弛后的 dist 和 parent 值如下：

顶点	dist	parent
t	8	x
x	9	t
y	7	t
z	0	null
s	7	z

第三轮松弛后的 dist 和 parent 值如下：

顶点	dist	parent
t	8	x
x	9	t
y	7	t
z	0	null
s	7	z

第三轮松弛后的 dist 和 parent 值没有发生变化，说明已经达到最优解，无需进行更多的轮数, 算法终止。

(c) 每次循环后的 dist 和 parent 值如下：

Iteration	dist(r)	dist(s)	dist(t)	dist(x)	dist(y)	dist(z)	parent(s)	parent(t)	parent(x)	parent(y)	parent(z)
0	0	∞	∞	∞	∞	∞	None	None	None	None	None
1	0	5	8	∞	7	∞	r	r	None	r	None
2	0	5	8	9	7	∞	r	r	s	r	None
3	0	5	8	9	7	14	r	r	s	r	t
4	0	5	8	9	7	14	r	r	s	r	t

最终得到的从 r 到其他顶点的最短距离和最短路径如下：

顶点	最短距离	最短路径
r	0	r
s	5	$r \to s$
t	8	$r \to t$
x	9	$r \to s \to x$
y	7	$r \to y$
z	14	$r \to t \to z$

2 Problem2

(a) 每轮的矩阵 dist：

k	D^k
0	$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$
1	$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$
2	$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$
3	$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$
4	$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$
5	$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$
6	$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$

(b)h value 如下：

v	$h(v)$
1	-5
2	-3
3	0
4	-1
5	-6
6	-8

w(u,v) 如下：

u	v	$\hat{w}(u,v)$
1	2	NIL
1	3	NIL
1	4	NIL
1	5	0
1	6	NIL
2	1	3
2	3	NIL
2	4	0
2	5	NIL
2	6	NIL
3	1	NIL
3	2	5
3	4	NIL
3	5	NIL
3	6	0
4	1	0
4	2	NIL
4	3	NIL
4	5	8
4	6	NIL
5	1	NIL
5	2	4
5	3	NIL
5	4	NIL
5	6	NIL
6	1	NIL
6	2	0
6	3	18
6	4	NIL
6	5	NIL

所以得到的最短路径矩阵 $D = d_{i,j}$ 如下：

$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

3 Problem3

Modifying the Dijkstra’ s algorithm to solve the problem. My idea is to use the product of reliabilities instead of the sum of weights, because the total reliability of a path is the product of the reliabilities of its edges.
Here is the pseudocode:

```
function MostReliablePath(G, s, t):
    for each vertex v in V[G] do
        r[v] = 0 if v   s else 1
        parent[v] = NIL
    Q = V[G]
    while Q   empty do
        u = ExtractMax(Q, r)
        for each vertex v in Adj[u] do
            if r[v] < r[u] * r(u, v) then
                r[v] = r[u] * r(u, v)
                parent[v] = u
    return r, parent
```

The running time of this algorithm is the same as the Dijkstra’ s algorithm, which is O((V+E) log V) if implemented with a binary heap, where V is the number of vertices and E is the number of edges. Because I perform at most |V| extract-max operations and at most |E| increase-key operations. Each of these operations takes O(log V) time. Therefore, the total time complexity is O((V+E) log V).

4 Problem4

Modify the Bellman-Ford algorithm to solve this problem. My idea is to add a new vertex to the graph and connect it to all other vertices with zero-weight edges. Then run the Bellman-Ford algorithm from this new vertex. The shortest path from the new vertex to any other vertex v will be the minimum distance from any vertex to v.
Here is the pseudocode:

```
function ModifiedBellmanFord(G, w):
    G' = G with a new vertex s added
    for each vertex v in V[G'] do
        add edge (s, v) to E[G'] with w(s, v) = 0
    InitializeSingleSource(G', s)
    for i from 1 to |V[G']| - 1 do
        for each edge (u, v) in E[G'] do
            Relax(u, v, w)
    for each edge (u, v) in E[G'] do
        if d[v] > d[u] + w(u, v) then
            return "Graph contains a negative-weight cycle"
    return d
```

The running time of this algorithm is O(|V| * (|V| + |E|)), which is the same as the Bellman-Ford algorithm. Because I perform |V| - 1 passes over the edges, and each pass takes O(|E|) time. The initialization takes O(|V|) time, and the final check for negative-weight cycles takes O(|E|) time. Therefore the total time complexity is O(|V| * (|V| + |E|)).

The correctness of this algorithm follows from the correctness of the Bellman-Ford algorithm. By adding a new vertex s and zero-weight edges from s to all other vertices, I can ensure that there is a path from s to every vertex. Since all these new edges have weight zero, they do not change the minimum distance from any vertex to any other vertex. Therefore, the shortest path from s to any vertex v in the modified graph is the minimum distance from any vertex to v in the original graph.

5 Problem5

(a) Compute the total number of paths in G:

```
function TotalPaths(G, s, t):
    Initialize pathCount[] as 0 for all vertices
    pathCount[s] = 1
    for each vertex u in topological order do
        for each vertex v adjacent to u do
            pathCount[v] += pathCount[u]
    return pathCount[t]
```

My algorithm first computes a topological ordering of the vertices, then iterates over them in this order. For each vertex, it adds the number of paths to it from its predecessors, so the time complexity is O(|V| + |E|).

(b) Compute the earliest starting time of each job:

```
function EarliestStartTimes(G, s, w):
    Initialize dist[] as $\infty$ for all vertices except s
    dist[s] = 0
    for each vertex u in topological order do
        for each vertex v adjacent to u do
            if dist[v] < dist[u] + w(u) then
                dist[v] = dist[u] + w(u)
    return dist
```

The algorithm is similar to the longest path algorithm in a DAG. It computes a topological ordering of the vertices, then iterates over them in this order. For each vertex, it updates the earliest start time if a longer path is found, so the time complexity is O(|V| + |E|).

(c) Compute the latest starting time of each job without affecting the project' s total duration:

```
function LatestStartTimes(G, s, t, w, dist):
    Initialize slack[] as $\infty$ for all vertices except t
    slack[t] = dist[t]
    for each vertex u in reverse topological order do
```

```
        for each vertex v adjacent to u do
            if slack[u] > slack[v] − w(u) then
                slack[u] = slack[v] − w(u)
    return slack
```

The algorithm is similar to the earliest start times algorithm but iterates in reverse topological order. It computes the latest start time such that the total duration of the project is not affected, so the time complexity is O(|V| + |E|).

6 Problem6

Using a greedy algorithm, my idea is to sort the intervals by their right endpoints, then iteratively select the interval with the smallest right endpoint that intersects with the current interval.
Here is the pseudocode:

```
function SmallestCover(L, R):
    n = length(L)
    sort intervals by R
    Y = empty set
    i = 1
    while i    n do
        [l, r] = [L[i], R[i]]
        Y = Y    {[l, r]}
        i = i + 1
        while i    n and L[i]    r do
            i = i + 1
    return Y
```

The running time of this algorithm is O(n log n), which is dominated by the sorting of the intervals. Because I perform a single pass over the sorted intervals, which takes O(n) time, and each operation inside the loop takes constant time. Therefore the total time complexity is O(n log n).

The correctness of this algorithm follows from the greedy choice property. At each step, I make the locally optimal choice of selecting the interval with the smallest right endpoint that intersects with the current interval. This ensures that I leave as much room as possible for the remaining intervals, which leads to the smallest cover. This greedy strategy yields a global optimum, which is the smallest cover of all intervals.

The proof of the greedy choice property is as follows:
Suppose that there is an optimal solution that includes an interval with a larger right endpoint. We can replace this interval with the interval with the smallest right endpoint without affecting the coverage, because the interval with the smallest right endpoint also covers all intervals that intersect with the current interval. This gives us another solution with the same coverage but a smaller size, which contradicts the assumption that the original solution is optimal. Therefore, the greedy choice property holds, and the greedy algorithm is correct.

The proof of the optimality of the greedy algorithm is as follows:
Suppose that there is an optimal solution that differs from the greedy solution. We can transform this solution into the greedy solution by repeatedly applying the greedy choice property, without increasing the size of the solution. This shows that the greedy solution is at least as good as any other solution, and therefore it is optimal. Therefore, the greedy algorithm is optimal, and it computes the smallest cover of all intervals.

7 Problem7

(a) A counterexample to the greedy algorithm can be constructed with a coin system where the denominations do not have a common divisor other than 1. For instance, consider a coin system with denominations 1, 3, and 4. If I want to make change for 6, the greedy algorithm would first take a 4 coin, then two 1 coins for a total of three coins. However, the optimal solution is to use two 3 coins.

(b) For a currency system where the coin denominations are consecutive powers of some integer $b \geq 2$, the greedy algorithm does indeed make optimal change. Because any smaller coin denomination is a divisor of all larger coin denominations. Therefore, using a larger coin will never prevent us from being able to make exact change later.

Proof: Suppose I have an amount A to make change for, and c[i] is the largest coin that is less than or equal to A. If c[i] is not part of the optimal solution, then the optimal solution consists of coins all smaller than c[i]. But since $c[i] = b \times c[i-1]$, we could replace b coins of denomination c[i-1] with a single coin of denomination c[i], contradicting the optimality of the original solution. Therefore, the greedy algorithm is optimal for this currency system.

The time complexity of the greedy algorithm is O(n)