ShanghaiTech University

CS101 Algorithms and Data Structures
Fall 2024

Homework 10

Due date: December 11, 2024, at 23:59

1. Please write your solutions in English.
2. Submit your solutions to Gradescope.
3. Set your FULL name to your Chinese name and your STUDENT ID correctly in Gradescope account settings.
4. If you want to submit a handwritten version, scan it clearly. `CamScanner` is recommended.
5. We recommend you to write in LaTeX.
6. When submitting, match your solutions to the problems correctly.
7. No late submission will be accepted.
8. Violations to any of the above may result in zero points.

1. (16 points) Multiple Choices

Each question has **one or more** correct answer(s). Select all the correct answer(s). For each question, you will get 0 points if you select one or more wrong answers, but you will get 1 point if you select a non-empty subset of the correct answers.

Write your answers in the following table.

(a)	(b)	(c)	(d)
BCD	BCD	A	ABC
(e)	(f)	(g)	(h)
ACD	AC	C	BCD

(a) (2') Which of the following statements is/are **TRUE**?

A. Dijkstra algorithm can find the shortest path in any DAG.

B. If we use the Dijkstra algorithm, whether the graph is directed or undirected does not matter.

C. At each iteration of the Dijkstra algorithm, we pop the vertex with the shortest current distance to the start vertex, while in the Prim algorithm, we pop the vertex with the shortest distance to the current minimum spanning tree.

D. In directed graph $G = (V, E)$, if $(s, v_1, v_2, v_3, v_4, t)$, where $s, v_i, t \in V$, is the shortest path from s to t in G, then (v_1, v_2, v_3, v_4) must be the shortest path from v_1 to v_4 in G.

(b) (2') Which of the following statements is/are **TRUE**?

A. Bellman-Ford algorithm can find the shortest path for negative-weighted directed graphs, while the Dijkstra algorithm may fail.

B. The run time of the Bellman-Ford algorithm is $O(|V||E|)$, which is more time-consuming than the Dijkstra algorithm with heap implementation.

C. Topological sort can be extended to determine whether a graph has a cycle while the Bellman-Ford algorithm can be extended to determine whether a graph has a negative cycle.

D. Topological sort can find the critical path in a DAG while the Bellman-Ford algorithm can find the single-source shortest path in a DAG.

Solution:

A. Bellman-Ford works for negative-weighted directed graphs without negative cycles. It fails with negative cycles.

(c) (2') Given an undirected graph $G = (V, E)$ with its minimum spanning tree $T \subset E$, which of the following statements is/are **TRUE**?

A. Both the Prim algorithm and the Dijkstra algorithm can run in time complexity $O(|E|\log|V|)$.

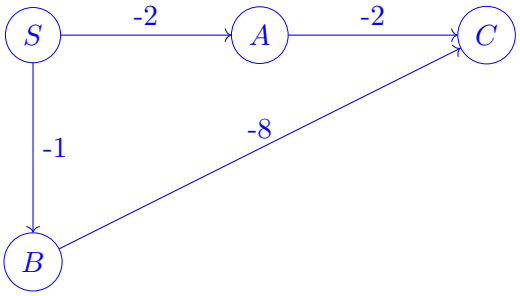
B. After negating all weights of edge in G and getting the new graph G', apply the original Prim algorithm on G' will find the Maximum Spanning Tree of G. Similarly, applying the Dijkstra algorithm will find the longest distance.

C. We prefer Dijkstra's algorithm with binary heap implementation to the naive adjacency matrix implementation in a dense graph where $|E| = \Theta(|V|^2)$

D. For any vertex v in G such that the shortest paths from v to all other vertices in V are the same as the shortest paths only using edges in T.

Solution:

B. The first part is right, and the second is wrong since Dijkstra can't be used to find the longest distance trace. For example, suppose we want to find the shortest distance between S to C in the negative graph below. If we don't change the logic in Dijkstra, it will prefer a $SA \to AC$ path rather than $SB \to BC$. And it will never use node B since both SA and SC are less than SB. However, SBC is the longest path in the original positive graph.



(d) (2') Which of the following statements about shortest path algorithms is/are **TRUE**?

A. If we modify the outer loop of the Bellman-Ford algorithm to execute $|V|$ iterations instead of $|V|-1$ iterations, the algorithm can still find the shortest path on a directed graph with negative-weight edges but no negative cycles.

B. We can modify the Bellman-Ford algorithm to find the longest path in a directed graph with positive-weight edges but no positive cycles.

C. We can modify the Floyd-Warshall algorithm to detect whether there exists a negative cycle or not in a directed graph.

D. If we modify the out-most loop of the Floyd-Warshall algorithm to execute $|V|-1$ iterations instead of $|V|$ iterations, the algorithm can still find all pairs of shortest paths on a directed graph with negative-weight edges but no negative cycles.

Solution:

A. $dist[v]$ will remain the same in the $|V|$-th iteration if the graph has no negative cycles.

B. To find the longest path is to find the shortest path in the graph where all edge weights are negated.

C. There exists a negative cycle if and only if $\exists v, dist[v][v] < 0$ after $|V|$ iterations for the out-most loop.

D. If only $|V| - 1$ iterations are executed, then every pair of shortest paths that must contain the $|V|$-th vertex is not computed correctly.

(e) (2') Which of the following is/are **TRUE**?

A. The A* graph search algorithm with a consistent heuristic will always return an optimal solution if it exists.

B. The A* graph search algorithm with an admissible heuristic will always return an optimal solution if it exists.

C. The A* tree search algorithm with a consistent heuristic will always return an optimal solution if it exists.

D. The A* tree search algorithm with an admissible heuristic will always return an optimal solution if it exists.

Solution:
If $h(n)$ is admissible, then the tree search is optimal.
If $h(n)$ is consistent, then the graph search is optimal.
If $h(n)$ is consistent, then it must be admissible. But if $h(n)$ is admissible, then it may not be consistent.

(f) (2') Which of the following is/are **TRUE**?

A. Dijkstra algorithm can be viewed as a special case of the A* Graph Search algorithm where the heuristic function from any vertex u to the terminal z is $h(u, z) = 0$.

B. Floyd-Warshall algorithm can always give the correct shortest distance between any two vertices in directed graphs with negative weights.

C. In A* graph search algorithm with a consistent heuristic function, if vertex u is marked visited before v, then $d(u)+h(u) \leq d(v)+h(v)$, where $d(u)$ is the distance from the start vertex to u.

D. Bellman-Ford algorithm can work correctly on any graphs with negative edges.

Solution:
B, D. Floyd-Warshall algorithm fails when a negative cycle exists, but it could be used to detect negative cycles.
C. The correction of A* is based on this. The proof is similar to Dijkstra.

(g) (2') We run the Floyd-Warshall algorithm on a simple graph without negative cycles with $3n$ vertices $v_1, \ldots, v_n, \ldots, v_{2n}, \ldots, v_{3n}$. Suppose all three loops (k, i, j) are iterated from 1 to $3n$. After running at least how many iterations of the out-most loop k, it is ensured to find the shortest path between v_{2n} and v_{3n}?

A. $2n - 1$
B. $2n$
C. $3n - 1$
D. $3n$

(h) (2') To solve the N Puzzle problems with the A* search, which of the following heuristics is/are admissible?

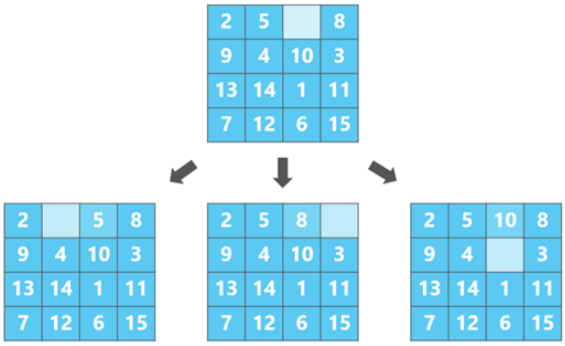


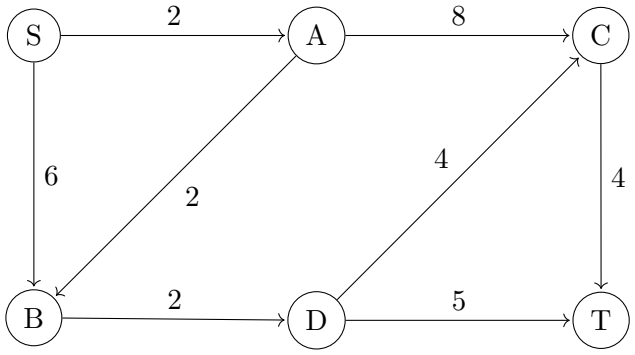
Figure 1: A kind of 15 puzzle

A. $h =$ if the state is the goal state, it is 1; otherwise, it is 0
B. $h =$ **number of misplaced tiles**
C. $h =$ **the sum of the minimum number of moves required to put each tile in its correct location**
D. $h =$ **actual minimum number of moves required to put all tiles in their correct location.**

2. (10 points) A star

Logan is doing an A* **graph search** from S to T on the graph below. Edges are labeled with weights.

The exploration order follows in their lexicographical order. (For example, $S \to X \to A$ would be expanded before $S \to X \to B$, and $S \to A \to Z$ would be expanded before $S \to B \to A$.



Node	S	A	B	C	D	T
Heuristic	11	9	12	4	5	0

Table 1: The initial heuristic value

Node	S	A	B	C	D	T
Heuristic	10	6	4	2	3	0

Table 2: The new heuristic value

(a) (3') He first uses the heuristic value in the Table 1 but finds it does not work well. That's because the given heuristic values are:

A. Admissible but not consistent

B. Consistent but not admissible

C. Neither admissible nor consistent

(b) Logan decides to modify the heuristic.

i. (3') He first gives out a new heuristic as in Table 2 above, please check whether the heuristic meets both admissibility and consistency (Write 'Yes' or 'No' for this).
If so, write down the path from S to T returned by A^* graph search (in the form of nodes, e.g. $S \to A \to C \to T$ should be written as $SACT$).
Otherwise, give out a contradiction that will lead admissibility or consistency to fail (in the form of 'Admissibility/Consistency' and its corresponding inequality.
e.g. If consistency fails on $A \to C$, write 'Consistency, $h(C) + 8 < h(A)$', here 8 corresponds to $w(A, C)$.).

Solution: No. Consistency, $h(A) + 2 < h(S)$

ii. (4') He finds that he can modify the heuristic of **only one node** to meet admissibility and consistency from the initial heuristic values in Table 1. Find the node and corresponding heuristic value for the chosen node. Justify your answer by stating the equalities obtained from admissibility and consistency.

Solution: Choose B with new value 7.
Admissibility: $h(B) \leq 7$. Consistency: $h(B) \geq 7.(A \to B \to D \to E)$.

3. (10 points) The shortest path with vertex weights

Given a directed graph $G = (V, E)$, you need to find the path from s to t with minimum cost. The edges are not weighted i.e. you can regard them as 1. However, the vertices in V have their weights and those are required to be considered in the path.

The graph $G = (V, E)$ is

- $V = \{1, 2, \ldots, n\}$, where each vertex i is associated with a positive vertex weight $a_i > 0$,
- $E = \{(u_i, v_i) : i = 1, 2, \ldots, m\}$, where each edge (u_i, v_i) is associated with a positive edge weight 1.

And $F : V \to \mathbb{R}^+$ represents the weight assigning function. (e.g. weight of vertex u is $F(u)$.)

A path is defined as a series of vertices $v_0 \to v_1 \to \cdots \to v_k$, which involves k edges and satifies that $\forall\ 0 \le i < k, ((v_i, v_{i+1}) \in E)$ (which implies $\forall\ 0 \le i \le k, v_i \in V$.).

The cost $C(P)$ of path $P = v_0 \to v_1 \to \cdots \to v_k$ is as follows:

$$C(P) = \sum_{i=0}^{k} F(u_i) + k$$

Now given a vertex $s \in V$, you need to find path $P = s \to (v_1 \to \cdots) \to t$ with minimum cost for all $t \neq s$. You are supposed to give out the $|V| - 1$ minimum cost of the paths. Guaranteed that s can reach all other vertices.

You should give out the steps of your algorithm (**as efficient as possible**) together with the time complexity (tight). You don't need to prove the correctness of your algorithm.

Hint: Try to create a new graph instead of modifying the algorithm known.

Solution:

- Construct a new weighted graph $G' = (V', E')$, where $V' = \{1, \cdots, 2n\}$ and $E' = \{(v + kn, u + kn, 1) \mid k \in \{0, 1\}, (v, u) \in E\} \cup \{(k, k + n, F(k) \mid k \in V\}$.
- Run Dijkstra on the new graph, the answer of vertex v of the original graph is the minimum cost of the path from s to $v + n$.
- The time complexity is $O(|E'| \log |V'|) = O((|E| + |V|) \log |V|)$. ($|E'| = |V| + 2|E|, |V'| = 2|V|$.)

4. (10 points) Negative Cycle Detection

The **inf(∞)** mentioned in this problem could be regarded as a pre-defined large enough constant. The graph mentioned in this problem is a simple directed graph. Please write down your codes in **standard C++**.

(a) Consider the following implementation of the Floyd-Warshall algorithm. Suppose W is the adjacency matrix of the graph, and assume that $W_{ij} = \infty$ where there is no edge between vertex i and vertex j, and assume $W_{ii} = 0$ for every vertex i. And other W_{ij} are the weights of the edge between vertex i and vertex j. The array 'graph' passed into the function is the adjacency matrix W.

```
1  bool DetectNegCycle_Floyd(const int graph[][V])
2  {
3      int dist[V][V];
4
5      for (int i = 0; i < V; ++i)
6          for (int j = 0; j < V; ++j)
7              dist[i][j] = graph[i][j];
8
9      for (int k = 0; k < V; ++k)
10         for (int i = 0; i < V; ++i)
11             for (int j = 0; j < V; ++j)
12                 if (dist[i][j] > dist[i][k] + dist[k][j])
13                         dist[i][j] = dist[i][k] + dist[k][j];
14
15     ____________________________________
16     ____________________________________
17     ____________________________________
18     ____________________________________
19
20     return false;
21 }
```

i. (2') Consider the three loop lines in the Floyd-Warshall algorithm: lines 9, 10, and 11. Which pair(s) of these lines can be swapped without affecting the correctness of the algorithm? List all possible pairs of lines that can be swapped.

Solution: 10, 11.

ii. (4') Add some codes in the blank lines to detect whether there are negative cycles in the graph. (You may not use all blank lines, or you can add more lines.)

Solution:

```
    for (int i = 0; i < V; ++i)__________
        if (dist[i][i] < 0)______________
            return true;_________________
    ____________________________________
```

(b) (4') Consider the following implementation of the Bellman-Ford algorithm. The 'edge' structure is used to represent the edges of the graph, with its elements u, v, and w denoting an edge from node u to node v and its corresponding weight w.

```
struct edge
{
    int u, v, w;
};

bool detectNegCycle_BellmanFord(const std::vector<edge>& Edge, int s)
{
    int dist[V];

    for (int i = 0; i < V; ++i)
        dist[i] = inf;
    dist[s] = 0;

    for(i = 1; i <= V - 1; ++i)
    {
        for(const auto& e : Edge)
            if (dist[e.v] > dist[e.u] + e.w)
                dist[e.v] = dist[e.u] + e.w;
    }

    ____________________________________
    ____________________________________
    ____________________________________
    ____________________________________

    return false;
}
```

Add some codes in the blank lines to detect whether there are negative cycles in the graph. (You may not use all blank lines, or you can add more lines.)

Solution:

```
1     for(const auto& e : Edge)____________
2         if (dist[e.v] > dist[e.u] + e.w)_
3             return true;_________________
4     ____________________________________
```

5. (10 points) Arbitrage

Preface: Shortest-path algorithms can be applied to many real-life domains. One of the most challenging aspects is constructing the graph in a complex environment. Consider the scenario below:

Recently, our beloved TA, Flash, was hired by an anonymous financial firm to assist with financial oversight. At the firm, Flash has access to a set of n currencies $C = c_1, c_2, \ldots, c_n$, including US dollars, Euros, Bitcoin, Dogecoin, and others. For every pair of currencies c_i and c_j, Flash knows the exchange rate $r_{i,j}$ which indicates how many units of currency c_j you can receive in exchange for one unit of currency c_i. Assume that $r_{i,i} = 1$ for all currencies and that $r_{i,j} > 0, r_{i,j} = \dfrac{1}{r_{j,i}}$ for all pairs i, j.

Since the exchange rates between different currencies can be volatile and unpredictable, there may be times when you can start with one unit of arbitrary currency c_i and end up with more than one unit of c_i through exchanging. Such a situation is known as **arbitrage**.

More formally, arbitrage occurs if there exists a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that:

$$r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k} \cdot r_{ik,i_1} > 1$$

A system of exchange rates is considered **arbitrage-free** if there is no such sequence of exchanges that results in a profit, i.e. it is impossible to gain more than one unit of any currency by performing a series of exchanges.

Due to his heavy TA workload, Flash needs your help to design algorithms that can efficiently perform the tasks assigned by the company.

(a) (5') Flash has access to a set of exchange rates of $(r_{i,j})_{i,j \in 1,2,\ldots,n}$, his first task is to find the most profitable sequence between currencies a and b. In other words, given a fixed amount of currency a, he can get the most amount of currency b through this sequence of exchanges. **In this part, suppose the set of exchange rates is arbitrage-free.**

To assist Flash, please provide your algorithm and analyze the time-cost of it. You don't need to prove the correctness of the algorithm. Your time complexity should not exceed the complexity of the algorithms taught in the current course.

Hint: $\log(xy) = \log(x) + \log(y)$.

Solution: We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from a to b corresponds to finding the path with the largest product of exchange rates. To turn this into a shortest path problem, we weight the edges with the negative log of each exchange rate, i.e. set $w_{ij} = -\log r_{ij}$. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

Since it's just an original Bellman-Ford algorithm, it takes $O(|E||V|)$ time, which is also $O(|V|^3) = O(n^3)$ in this problem.

(b) (5') The second task of Flash is to detect whether arbitrage exists given an exchange rate set at any specific time, which is, obviously, too hard to do manually.

For this part, please give your algorithm, analyze its time complexity, and briefly explain why it can work. Your time complexity should not exceed the complexity of the algorithms taught in the current course.

Solution: Just iterate the updating procedure once more after $|V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exit, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{j-1} r_{i_j,i_{j+1}} > 1$ as required.

The runtime is the same as $O(n^3)$.

6. (6 points) k-layer Shortest Path

Consider a weighted directed graph $G = (V, E)$ and source vertex $s \in V$. It is given that for each vertex $v \in V$, there exists a shortest path from s to v that uses at most k edges. Devise an algorithm to determine the shortest path weight from s to every vertex $v \in V$.

Hints: *Consider to rebuild a graph with $O(k|V|)$ number of node.*

(a) (3') Design an algorithm to solve this problem using the shortest-path algorithm. Which algorithm will you choose Explain why.

Solution:

Consider a naive algorithm in which, we duplicate the graph G in k times. For such original edge (u, v), we connect (u_i, v_{i+1}) in the new graph that u_i are the vertex u in layer i. Then the answer for vertex v could be minimized through v_i for $i \in [1, k]$. However, such an algorithm will expand $|E|'$ to $k|E|$ then shall get bad time complexity.

The best algorithm in the original graph we chose is the Bellman-Ford algorithm since there may be a negative cycle in the graph. But somehow it's unnecessary since we reconstruct the graph to a DAG.

(b) (3') As the shortest path only uses at most k edges, optimize your algorithm to $O(|V| + k|E|)$ time.

Solution:

Rather than creating the duplicated graph G all at once and then running DAG Relaxation, we begin with G_0 only containing the first layer (v_0 for $v \in V$) and δ initialized with first-layer distances $\delta(s_0, v_0) = \infty$ for $v \in V$, except for $\delta(s_0, s_0) = 0$.

Then for i starting at 0, assume for induction that G_0 contains layers 0 to i and $\delta(s_0, v_j)$ have been computed for all $v \in V$ and $j \in \{0, \ldots, i\}$. Append layer $i + 1$ by adding vertices v_{i+1} for $v \in V$ and associated edges from layer i. This new layer cannot affect distances in previous layers (since the new layer comes later in the topological order). So, continue DAG Relaxation by relaxing the new edges out of the vertices in layer i, which correctly computes $\delta(s_0, v_{i+1})$ for all $v \in V$.

Since a shortest path from s to each vertex uses at most k edges, the distance $\delta(s, v)$ equals the k_0-edge distance $\delta_{k_0}(s, v) = \delta(s_0, v_{k_0})$ for every $k_0 \geq k$. Thus, as soon as $\delta(s_0, v_{k_0}) = \delta(s_0, v_{k_0+1})$ for all $v \in V$, then $k_0 = k$ and we can terminate by outputting $\delta(s, v) = \delta(s_0, v_k)$ for each $v \in V$.

Since every vertex is reachable from s, $|V| = O(|E|)$, and since we only construct $k+1$ layers of G_0 before finding two equal layers, and each layer takes at most $O(|E|)$ time to process, this algorithm runs in $O(k|E|)$ time (which is also $O(|V| + k|E|)$). The $|V|$ was originally provided for this problem to allow the input to include disconnected graphs, where an initial single-source reachability algorithm could restrict to the subset

of the graph reachable from s; but this term is no longer relevant for connected input graphs.

Another method: Use the DP method(Bellman-Ford without rolling array). Let $dp[x][k]$ represent the shortest path from s to x with less than or equal to k edges. Then the Bellman equation is $dp[x][k] = \min_{(u,x)\in E} dp[u][k-1] + w(u,x)$, where $w(u,x)$ is the weight of the edge from u to x. And the final solution is $dp[x][k]$.