Dijkstra Sequence

黄文杰

3210103379

Date：2023-12-7

Chapter 1: Introduction

Dijkstra's algorithm, a well-known and widely applied greedy algorithm, was conceived by computer scientist Edsger W. Dijkstra in 1956 to address the single-source shortest path problem. Its purpose is to determine the shortest paths from a designated source vertex to all other vertices in a given graph.

The algorithm maintains a set that includes vertices forming the shortest path tree. At each step, it selects a vertex not yet included, minimizing the distance from the source, and adds it to the set. This iterative process yields an ordered sequence of vertices known as the "Dijkstra sequence."

This problem confronts us with the challenge of verifying whether a given sequence aligns with the criteria of a Dijkstra sequence, highlighting the need to discern and validate the order of vertices in the context of the algorithm's principles.

Chapter 2: Algorithm Specification

- **Data Structure:**

```
// Two-dimensional edge weight table (INF representing no adjacent edge).
int edge[MAXNUM][MAXNUM];
```

> - **Description :**
>
>   The data structure used in the program is a two-dimensional array `edge`, representing the edge weights between vertices in the given graph. The array is of size `MAXNUM x MAXNUM`, where `MAXNUM` is the maximum number of vertices. The value `INF` is used to denote the absence of an adjacent edge between vertices.

- **Algorithm 1:** `check_dijkstra1(int v_num)`

```
Algorithm check_dijkstra1(v_num):
    Input: Total number of vertices v_num
    Output: 1 if it is a Dijkstra sequence, 0 otherwise

    seq = ReadInputSequence(v_num)
    dist, visited = InitializeArrays(v_num)
    dist[seq[0]] = 0

    for i = 0 to v_num - 1:
        next_v = seq[i]
        min_dist = dist[seq[i]]

        for j = 1 to v_num:
            if not visited[j] and dist[j] < min_dist:
                return 0

        visited[next_v] = 1
```

```
            for j = 1 to v_num:
                if not visited[j] and edge[next_v][j] != INF:
                    if edge[next_v][j] + dist[next_v] < dist[j]:
                        dist[j] = edge[next_v][j] + dist[next_v]

        return 1

```

- **Parameter Description :**
    - Input Parameters: An integer `v_num` representing the total number of vertices.
    - Output: Returns 1 if it is a Dijkstra sequence, 0 otherwise.
    - Function's purpose: Checks whether a given sequence is a Dijkstra sequence.
- **Algorithm Description :**
    1. Read the input sequence of vertices (`seq`).
    2. Initialize distance (`dist`) and visited arrays (`visited`).
    3. Set the distance of the source vertex to 0.
    4. Iterate through vertices in the input sequence.
    5. For each vertex, check if there is any smaller unvisited distance.
    6. If found, it's not a Dijkstra sequence; otherwise, mark the vertex as visited and update distances.

- **Algorithm 2:** `check_dijkstra2(int v_num)`

```
Algorithm check_dijkstra2(v_num):
    Input: Total number of vertices v_num
    Output: 1 if it is a Dijkstra sequence, 0 otherwise

    seq = ReadInputSequence(v_num)
    dist = InitializeArray(v_num)
    dist[seq[0]] = 0

    for i = 0 to v_num - 1:
        u = seq[i]

        for v = 1 to v_num:
            if edge[u][v] != INF:
                if dist[u] + edge[u][v] < dist[v]:
                    dist[v] = dist[u] + edge[u][v]

    for i = 1 to v_num - 1:
        if dist[seq[i]] < dist[seq[i - 1]]:
            return 0

    return 1

```

- **Parameter Description :**
    - Input Parameters: An integer `v_num` representing the total number of vertices.
    - Output: Returns 1 if it is a Dijkstra sequence, 0 otherwise.

> - Function's purpose: Checks whether a given sequence is a Dijkstra sequence.
>
> - **Algorithm Description :**
>
>   1. Read the input sequence of vertices (`seq`).
>   2. Initialize distance array (`dist`).
>   3. Set the distance of the source vertex to 0.
>   4. Iterate through vertices in the input sequence.
>   5. Update distances to adjacent vertices and check if the sequence is non-decreasing.

- **Difference between Algorithm1 and Algorithm2**
  - **Main part of algorithm 1 (`check_dijkstra1`):**
    - Description:
      - Maintains an additional array `visited` to mark whether each vertex has been visited.
      - Checks at each step if there is any vertex with a smaller distance that has not been visited(If found, it's not a Dijkstra sequence), ensuring the sequence is non-decreasing.
    - Pseudocode:

```
for i = 0 to v_num - 1:
    next_v = seq[i]
    min_dist = dist[seq[i]]

    for j = 1 to v_num:
        if not visited[j] and dist[j] < min_dist:
            return 0
```

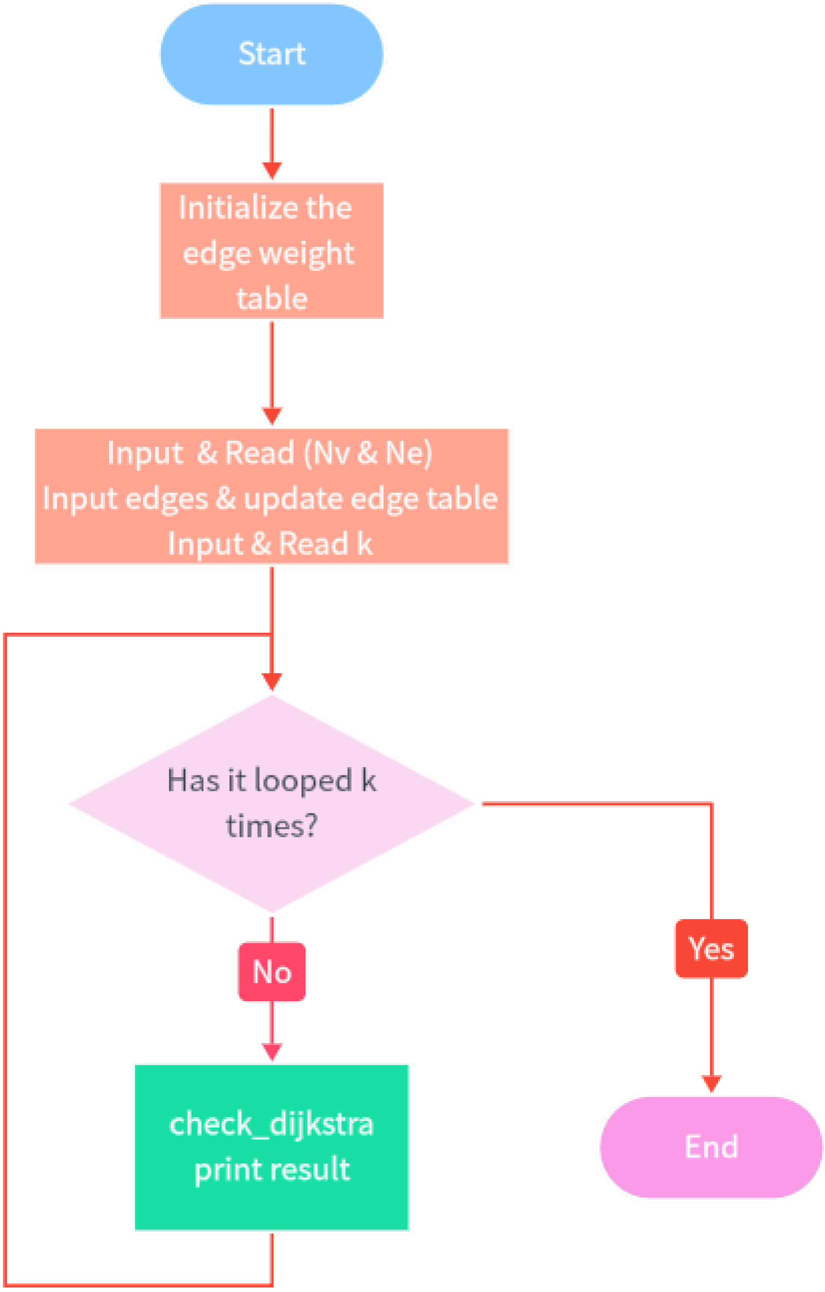
  - **Main part of algorithm 2 (`check_dijkstra2`):**
    - Description:
      - Simplifies the approach by directly checking if the resulting distances are non-decreasing after updating distances to adjacent vertices.
      - Does not use a separate array for marking visited vertices.
    - Pseudocode:

```
for i = 0 to v_num - 1:
        u = seq[i]

        for v = 1 to v_num:
            if edge[u][v] != INF:
                if dist[u] + edge[u][v] < dist[v]:
                    dist[v] = dist[u] + edge[u][v]

for i = 1 to v_num - 1:
    if dist[seq[i]] < dist[seq[i - 1]]:
        return 0
```

- **Key Similarities and Differences:**

> - **Data Structures:**
>   - Both algorithms use the same data structure, a two-dimensional array `edge` representing the edge weights between vertices.
> - **Approach:**
>   - Algorithm 1 fundamentally involves iteratively checking whether the given sequence adheres to the properties of a Dijkstra sequence by progressively evaluating each vertex from the input sequence.
>   - Algorithm 2 simplifies the approach by updating distances and directly checking for a non-decreasing sequence afterward, without maintaining a separate array for visited vertices.
> - **Complexity:**
>   - Algorithm 1 has a slightly more complex structure due to the additional array `visited` and multiple checks during the iteration.
>   - Algorithm 2 is more straightforward, as it directly examines the resulting distances for non-decreasing order.
> - **Conclusion:**
>
>   In summary, both algorithms aim to check whether a given sequence is a Dijkstra sequence, but Algorithm 1 takes a more cautious approach by explicitly checking for smaller distances during the iteration, while Algorithm 2 simplifies the process by directly examining the resulting distances for non-decreasing order after updating them.

- **a sketch of the main program**



- **the main program**

```
int main(void){
    // Initialize the two-dimensional edge weight table.
    for(int i=0;i<MAXNUM;i++){
        for(int j=0;j<MAXNUM;j++){
            if(i==j){
                edge[i][j]=0;
            }else{
                edge[i][j]=INF;
            }
        }
    }

```

```
    int v_num,e_num; // the total numbers of vertices(v_num) and edges(e_num)
    scanf("%d%d",&v_num,&e_num);
    // Read in information about weighted edges and
    // update the two-dimensional edge weight table.
    for(int i=0;i<e_num;i++){
        int x,y,weight;
        scanf("%d%d%d",&x,&y,&weight);
        edge[x][y]=edge[y][x]=weight;
    }

    // the number of queries
    int k;
    scanf("%d",&k);
    for(int i=0;i<k;i++){

        // use algorithm1
        int flag = check_dijkstra1(v_num);
        // use algorithm2
        //int flag = check_dijkstra2(v_num);

        if(flag){
            printf("Yes\n");
        }else{
            printf("No\n");
        }
    }

}
```

Chapter 3: Testing Results

Test Case	Input	Expected Output	Actual Output
Case 1 (comprehensive test)	5 7 1 2 2 1 5 1 2 3 1 2 4 1 2 5 2 3 5 1 3 4 1 4 5 1 3 4 2 5 3 1 2 4 2 3 4 5 1 3 2 1 5 4	Yes Yes Yes No	Yes Yes Yes No
	6 9 1 2 2 1 3 1 1 6 3 2 3 2 2 4 1 2 5 2	No No Yes No	No No Yes No

Case 2 (Comprehensive Test)	3 5 1 4 6 2 5 6 1 4 1 2 3 4 5 6 1 6 5 2 3 4 6 5 4 3 2 1 4 3 2 1 5 6		
Case 3 (Smallest Sizes Test)	2 1 1 2 3 2 1 2 2 1	Yes Yes	Yes Yes
	30 30 1 2 1 2 3 2 3 4 3 4 5 4 5 6 5 6 7 6 7 8 7	No No	No No

Case 4 (Largest Sizes Test)	8 9 8 9 10 9 10 11 10 11 12 11 12 13 12 13 14 13 14 15 14 15 16 15 16 17 16 17 18 17 18 19 18 19 20 19 20 21 20 21 22 21 22 23 22 23 24 23 24 25 24 25 26 25 26 27 26 27 28 27 28 29 28 29 30 29 30 1 **30** 2 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 20 21 22 23 24 25 26 27 28 29 30 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19		
Case 5 (Extreme Cases Test)	1 0 1 1	Yes	Yes
Case 6 (Extreme Cases Test)	1 0 0	null	null

Testing Purpose

- **Case 1**

  > **Purpose:** This test includes multiple paths and nodes, validating the algorithm's ability to handle complex scenarios.

- **Case 2**

  > **Purpose:** This test includes multiple paths and nodes, validating the algorithm's ability to handle complex scenarios.

- **Case 3**

  > **Purpose:** This test checks if the algorithm correctly handles the smallest input size.

- **Case 4**

  > **Purpose:** This test checks if the algorithm correctly handles the largest input size.

- **Case 5**

  > **Purpose:** This test examines how the algorithm handles extreme cases, such as null input edges with the smallest graph size.

- **Case 6**

  > **Purpose:** This test examines how the algorithm handles extreme cases, such as null input test cases with the smallest graph size.

Chapter 4: Analysis and Comments

- ### Time complexity analysis

  > **Algorithm 1 (`check_dijkstra1`):**
  >
  > - The time complexity of this algorithm primarily depends on the two nested loops.
  > - The outer loop iterates over the vertices in the input sequence, which has a time complexity of $O(N_V)$, where N_V is the number of vertices.
  > - The inner loop, checking for smaller distances among unvisited vertices, contributes $O(N_V)$ in the worst case.
  > - Therefore, the overall time complexity of Algorithm 1 is $O(N_V^2)$.

  > **Algorithm 2 (`check_dijkstra2`):**
  >
  > - This algorithm involves two nested loops.
  >   - The outer loop iterates over the vertices in the input sequence, contributing $O(N_V)$ to the time complexity.
  >   - The inner loop, checking distances to adjacent vertices, also has a time complexity of $O(N_V)$.

> - Therefore, the overall time complexity of Algorithm 2 is $O(N_V^2)$.

- ## Space complexity analysis

> - Both algorithms utilize arrays to store information:
>   - Algorithm 1 (`check_dijkstra1`) uses arrays `seq`, `dist`, and `visited`, each of size $N_V + 1$.
>   - Algorithm 2 (`check_dijkstra2`) uses arrays `seq` and `dist`, each of size $N_V + 1$.
> - Therefore, the space complexity for both algorithms is $O(N_V)$.

> - Additionally, the entire program uses a two-dimensional array `edge` of size `MAXNUM x MAXNUM`, where `MAXNUM` is a constant representing the maximum number of vertices (N_V). Therefore, the overall space complexity of the program is $O(N_V^2)$.

- ## Conclusion

In conclusion, the implemented program utilizes two algorithms, Algorithm 1 (`check_dijkstra1`) and Algorithm 2 (`check_dijkstra2`), to verify whether a given sequence aligns with the criteria of a Dijkstra sequence. The time complexity analysis reveals that Algorithm 1 has a time complexity of $O(N_V^2)$, where N_V is the number of vertices, due to its nested loops. On the other hand, Algorithm 2, despite having a single loop, also has a time complexity of $O(N_V^2)$ because of its nested structure within the loop.

Regarding space complexity, both algorithms use arrays of size $N_V + 1$, contributing to a space complexity of $O(N_V)$. Additionally, the overall space complexity of the program includes the memory used for the two-dimensional edge weight table, resulting in a total space complexity of $O(N_V^2)$.

It's worth noting that Algorithm 1, with its early exit mechanism, might exhibit slightly better practical performance compared to Algorithm 2, which traverses the entire input sequence. The efficiency of the algorithms is influenced by the specific characteristics of the input data and the graph structure. Overall, the choice between Algorithm 1 and Algorithm 2 involves a trade-off between time complexity and practical performance based on the characteristics of the input data.

Appendix: Source Code (in C)

```c
#include<stdio.h>
#include<stdlib.h>

#define INF 10000 // Define a macro to represent infinity.
#define MAXNUM 1005 // Define the maximum value for the number of vertices

```

```c
// two-dimensional edge weight table (INF representing no adjacent edge).
int edge[MAXNUM][MAXNUM];

// Check whether a given sequence is a Dijkstra sequence and return the result.
// Algorithm1:
int check_dijkstra1(int v_num); // 1:Yes 0:No
// Algorithm2:
int check_dijkstra2(int v_num); // 1:Yes 0:No

int main(void){
    // Initialize the two-dimensional edge weight table.
    for(int i=0;i<MAXNUM;i++){
        for(int j=0;j<MAXNUM;j++){
            if(i==j){
                edge[i][j]=0;
            }else{
                edge[i][j]=INF;
            }
        }
    }

    int v_num,e_num; // the total numbers of vertices(v_num) and edges(e_num)
    scanf("%d%d",&v_num,&e_num);
    // Read in information about weighted edges and
    // update the two-dimensional edge weight table.
    for(int i=0;i<e_num;i++){
        int x,y,weight;
        scanf("%d%d%d",&x,&y,&weight);
        edge[x][y]=edge[y][x]=weight;
    }

    // the number of queries
    int k;
    scanf("%d",&k);
    for(int i=0;i<k;i++){

        // use algorithm1
        int flag = check_dijkstra1(v_num);
        // use algorithm2
        //int flag = check_dijkstra2(v_num);

        if(flag){
            printf("Yes\n");
        }else{
            printf("No\n");
        }
    }

}

int check_dijkstra1(int v_num) {
    int seq[v_num]; // The sequence of vertices
```

```
    int dist[v_num + 1]; // The minimum distance from the source to each vertex
    int visited[v_num + 1]; // Used to mark whether each vertex has been visited (0:NO 1:YES)

    // Read the input sequence of vertices
    for (int i = 0; i < v_num; i++) {
        scanf("%d", &seq[i]);
    }

    // Initialize the dist array and visited array
    for (int i = 0; i < v_num + 1; i++) {
        dist[i] = INF;
        visited[i] = 0;
    }

    dist[seq[0]] = 0;

    // Loop through the vertices in the input sequence
    for (int i = 0; i < v_num; i++) {
        int next_v = seq[i]; // The index of the smallest unknown distance vertex
        int min_dist = dist[seq[i]];

        // Check if there is any vertex with a smaller distance that has not been visited
        for (int j = 1; j <= v_num; j++) {
            if (!visited[j] && dist[j] < min_dist) {
                return 0; // Not a Dijkstra sequence if a smaller distance is found
            }
        }

        visited[next_v] = 1; // Mark the current vertex as visited

        // Update the distances of adjacent vertices
        for (int j = 1; j <= v_num; j++) {
            if (!visited[j] && edge[next_v][j] != INF) {
                if (edge[next_v][j] + dist[next_v] < dist[j]) {
                    dist[j] = edge[next_v][j] + dist[next_v];
                }
            }
        }
    }

    return 1; // It's a Dijkstra sequence
}


int check_dijkstra2(int v_num) {
    int seq[v_num];     // The sequence of vertices
    int dist[v_num + 1]; // The minimum distance from the source to each vertex

    // Read the input sequence of vertices
```

```
    for (int i = 0; i < v_num; i++) {
        scanf("%d", &seq[i]);
    }

    // Initialize the dist array with infinity for all vertices except the source
    for (int i = 0; i < v_num + 1; i++) {
        dist[i] = INF;
    }

    dist[seq[0]] = 0; // Set the distance of the source vertex to 0

    // Loop through the vertices in the input sequence
    for (int i = 0; i < v_num; i++) {
        int u = seq[i]; // Current vertex

        // Update distances to adjacent vertices
        for (int j = 1; j <= v_num; j++) {
            if (edge[u][j] != INF) { // If there is an edge between u and v
                if (dist[u] + edge[u][j] < dist[j]) {
                    // Update the distance if a shorter path is found
                    dist[j] = dist[u] + edge[u][j];
                }
            }
        }
    }

    // Check if the resulting distances are non-decreasing
    for (int i = 1; i < v_num; i++) {
        if (dist[seq[i]] < dist[seq[i - 1]]) {
            // Not a Dijkstra sequence if a smaller distance is found later in the sequence
            return 0;
        }
    }

    return 1; // It's a Dijkstra sequence
}
```