ISSN 2616-5775 Vol. 7, Issue 12: 79-84, DOI: 10.25236/AJCIS.2024.071211

Application of Dynamic Programming Algorithm in the Shortest Path Problem

Weiqing Wan[1], Yuanci Wan[2]

[1]*Jiangxi University of Engineering, Xinyu, Jiangxi, 338029, China*
[2]*Xinyu Emerging Industry Engineering School, Xinyu, Jiangxi, 338029, China*

Abstract: *This paper delves into the specific application of dynamic programming algorithms in solving the shortest path problem in multi-stage graphs. Faced with a given multi-stage graph structure, we are committed to finding an optimal path from vertex 0 to vertex 9, with the aim of minimizing the total length of the pipeline laid, which represents a classic optimization problem. To efficiently and accurately address this challenge, we have chosen dynamic programming as our core tool. This algorithm constructs solutions step-by-step and leverages the results of already solved sub problems to avoid redundant calculations, thereby significantly enhancing computational efficiency. The paper not only elaborates on the fundamental principles of dynamic programming and its application in the shortest path problem within multi-stage graphs, but also provides specific algorithm implementation code in C++ programming language. Our goal is to offer readers a comprehensive and in-depth guide, enabling them to understand and apply dynamic programming algorithms to successfully solve similar optimization problems related to the shortest path in multi-stage graphs.*

Keywords: *Dynamic Programming; Shortest Path; Multi-Stage Graph; C++ Implementation; Optimization Problem*

1. Introduction

In the field of graph theory, the shortest path problem has always been a research hotspot and challenge. This problem is widely applied in various fields such as transportation planning, network design, robot path planning, and communication network optimization. Finding the shortest path from the starting point to the destination can not only improve transportation efficiency but also reduce resource consumption and costs[1]. Therefore, studying effective solution methods for the shortest path problem holds significant theoretical and practical importance.

Dynamic programming algorithms, as a classic optimization algorithm, possess robust solving capabilities and can efficiently address the shortest path problem[2]. By decomposing the problem into several subproblems and leveraging the solutions of these subproblems to construct the solution of the original problem, dynamic programming avoids redundant calculations and enhances solving efficiency[3,4]. This paper takes an algorithm training question from the Blue Bridge Cup as an example to explore the application of dynamic programming algorithms in the shortest path problem and provides detailed C++ implementation code.

2. Principles of Dynamic Programming Algorithms

Dynamic programming algorithms are an effective method for solving optimization problems. They decompose complex problems into several subproblems, solve these subproblems one by one, and ultimately obtain the solution to the original problem. The core idea of dynamic programming algorithms is to utilize the solutions of already computed subproblems to construct the solution of the original problem, thereby avoiding redundant calculations and improving solving efficiency[5,6].

The basic steps of dynamic programming algorithms include:

Problem Decomposition: Decompose the original problem into several subproblems, which have certain correlations and repetitions among them[7].

State Definition: Define a state for each subproblem, which represents the solution to the subproblem.

ISSN 2616-5775 Vol. 7, Issue 12: 79-84, DOI: 10.25236/AJCIS.2024.071211

State Transition Equation: Establish state transition equations based on the relationships between subproblems to calculate the solution of the current subproblem[8].

Boundary Conditions: Determine the initial state and terminal state, providing the starting point and endpoint for the dynamic programming algorithm.

Solution: According to the state transition equations, gradually calculate the solution of each subproblem from the initial state until reaching the terminal state, obtaining the solution to the original problem[9].

In the shortest path problem, dynamic programming algorithms gradually calculate the shortest distances from the starting point to each vertex, ultimately obtaining the shortest path from the starting point to the destination. Dynamic programming algorithms can effectively handle shortest path problems with optimal substructure properties, i.e., the shortest path from any node to the starting point only depends on the subpath from the starting point to that node, and is independent of the path after that node[10].

3. Shortest Path Problem and Dynamic Programming

The shortest path problem stands as a cornerstone in graph theory, aiming to identify the path with the minimum cumulative weight from an origin to a destination. In a weighted graph, the weights of edges represent the distances or costs between adjacent vertices. This problem can be articulated as follows: within a weighted graph, locate the path from the origin to the destination that minimizes the sum of edge weights.

The shortest path problem embodies the optimal substructure property, which stipulates that the shortest path from any node to the origin solely depends on the sub-paths from the origin to that node, irrespective of the paths subsequent to that node. This characteristic renders dynamic programming algorithms particularly well-suited for solving the shortest path problem. By employing dynamic programming, we can incrementally compute the shortest distances from the origin to each vertex, ultimately yielding the shortest path from the origin to the destination[11].

The application of dynamic programming algorithms in the shortest path problem manifests primarily in several aspects:

State Definition: The shortest path problem is decomposed into subproblems that involve calculating the shortest distances from the origin to each vertex. Here, the state is defined as the shortest distance from the origin to the current vertex. This definition allows us to break down the complex problem into manageable parts, each focusing on a specific vertex.

State Transition Equation: Based on the adjacency matrix or adjacency list of the graph, a state transition equation is established to compute the shortest distance from the origin to the current vertex. This equation encapsulates the relationships between vertices and their connected edges, guiding the algorithm in determining the optimal path.

Boundary Conditions: The initial state sets the distance from the origin to itself as 0, while the distances from other vertices to the origin are considered infinite (indicating unreachability). The terminal state corresponds to the shortest distance from the origin to the destination. These conditions frame the problem, ensuring that the algorithm starts and ends in a well-defined manner.

Solution Process: Starting from the initial state, the algorithm iteratively computes the shortest distances to each vertex, progressively refining the estimates until it reaches the terminal state. This iterative refinement ensures that the algorithm considers all possible paths and selects the optimal one based on the cumulative edge weights.

In summary, dynamic programming algorithms leverage the optimal substructure property and state transition equations to systematically solve the shortest path problem, transforming a complex, multi-step decision-making process into a series of manageable computations[12].

4. Problem Presentation

The Blue Bridge Cup competition is a nationwide algorithmic contest aimed at assessing participants' algorithmic design and programming skills. Among the training questions for the Blue Bridge Cup competition, there is a problem related to the shortest path that requires finding the shortest

ISSN 2616-5775 Vol. 7, Issue 12: 79-84, DOI: 10.25236/AJCIS.2024.071211

path from vertex 0 to vertex 9 in a multi-stage graph.

A multi-stage graph is a special type of graph where vertices are divided into several stages, and each stage contains a group of vertices. The weight of an edge represents the distance between adjacent vertices. In this problem, we need to find the shortest path from vertex 0 to vertex 9 in a multi-stage graph G=(V, E), where the numbers on the edges indicate the distances between the corresponding two vertices, and the graph is represented by an adjacency matrix A.

The specific problem description is as follows:

In a multi-stage graph G=(V, E), there is a reservoir at vertex 0. We need to lay a pipeline from vertex 0 to vertex 9, and the numbers on the edges indicate the distances between the corresponding two vertices. The goal is to find a route from vertex 0 to vertex 9 such that the length of the laid pipeline is the shortest, it is shown in Figure 1. The code implementation is required to be in C++.

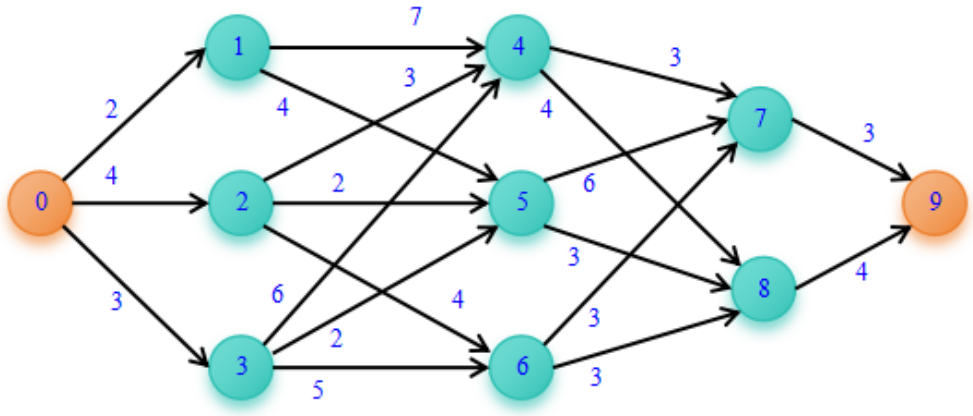

Figure 1: Seeking the Shortest Path for Pipeline Laying

To address this issue, we need to employ a dynamic programming algorithm to calculate the shortest path from vertex 0 to vertex 9. By defining states, establishing state transition equations, determining boundary conditions, and solving step-by-step, we can obtain the shortest path length from vertex 0 to vertex 9.

5. Steps of Solving the Shortest Path Problem Using Dynamic Programming

For the shortest path problem in a multi-stage graph, we can use a dynamic programming algorithm to find the solution. Here are the specific steps of solving the shortest path problem using dynamic programming:

Initialization: Define an array dist[i] to store the shortest distance from the starting point to the ith vertex. Set dist[0] to 0, indicating that the distance from the starting point to itself is 0; set other elements to infinity (indicating unreachability).

State Transition: For each vertex i (from 1 to n-1), traverse all its adjacent vertices j. If vertex i can be reached through edge (i-1, j) and dist[i-1] + A[i-1][j] is less than dist[i], then update dist[i] to dist[i-1] + A[i-1][j]. This step is the core of the dynamic programming algorithm, gradually calculating the shortest distances from the starting point to each vertex and constructing the shortest path from the starting point to the destination.

Solution: Repeat step 2 until all vertices are traversed. Ultimately, dist[n-1] will store the shortest distance from the starting point to the destination.

However, the above steps are mainly applicable to linear or nearly linear graphs. For multi-stage graphs, we may need to adopt more complex dynamic programming algorithms, such as the Floyd-Warshall algorithm. The Floyd-Warshall algorithm is a dynamic programming algorithm suitable for computing the shortest paths between all pairs of vertices, with a time complexity of O(n^3), where n is the number of nodes.

The basic idea of the Floyd-Warshall algorithm is to update the shortest distances between any two vertices by gradually introducing intermediate vertices. The specific steps are as follows:

ISSN 2616-5775 Vol. 7, Issue 12: 79-84, DOI: 10.25236/AJCIS.2024.071211

Initialization: Create a two-dimensional array dist to store the shortest distances between any two vertices. Initialize dist[i][j] to the value of the adjacency matrix A[i][j], representing the direct distance from vertex i to vertex j. If there is no direct path between vertex i and vertex j, set dist[i][j] to infinity.

State Transition: For each vertex k (from 0 to n-1), traverse all vertex pairs (i, j) (from 0 to n-1). If the distance from vertex i to vertex j through vertex k is smaller than the current value of dist[i][j], then update dist[i][j] to dist[i][k] + dist[k][j]. This step is the core of the Floyd-Warshall algorithm, continuously updating the shortest distances between any two vertices by gradually introducing intermediate vertices k.

Solution: Repeat step 2 until all vertices k are traversed. Ultimately, dist[0][9] will store the shortest distance from vertex 0 to vertex 9.

6. C++ Implementation Code of Algorithm

To efficiently find the shortest path from vertex 0 to vertex 9, we can opt for the classic algorithm in the field of dynamic programming, the Floyd-Warshall algorithm. This algorithm performs particularly well when dealing with multi-segment graphs that contain weights and may have complex structures. It is not only capable of calculating the shortest path between all pairs of vertices in the graph but also possesses high versatility, allowing it to adapt to various graph structures.

Although the design intention of the Floyd-Warshall algorithm is to solve the global shortest path problem, with a time complexity of O(n^3), where n represents the total number of nodes in the graph, it can be time-consuming when dealing with large graph structures. However, for the current specific problem, we can certainly optimize and adjust the algorithm. By streamlining the calculation process, we can focus the algorithm's attention solely on solving the shortest path between the specific pair of vertices from the starting vertex 0 to the ending vertex 9. In this way, we can maintain the algorithm's effectiveness while significantly enhancing the specificity and efficiency of the computation. Such optimization and adjustment make the Floyd-Warshall algorithm more suitable for solving the current specific problem, thereby providing us with an efficient and reliable solution.

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
int main() {
    int numVertices = 10; // Number of vertices
    //The adjacency matrix A represents the weights of the multi segment graph G
    vector<vector<int>> A = {
            {0, 4, 0, 0, 0, 0, 0, 0, 0, 0},
            {4, 0, 8, 0, 0, 0, 0, 2, 0, 0},
            {0, 8, 0, 7, 0, 4, 0, 0, 0, 0},
            {0, 0, 7, 0, 9, 0, 0, 0, 0, 0},
            {0, 0, 0, 9, 0, 10, 0, 0, 0, 0},
            {0, 0, 4, 0, 10, 0, 1, 0, 0, 0},
            {0, 0, 0, 0, 0, 1, 0, 6, 0, 0},
            {0, 2, 0, 0, 0, 0, 6, 0, 7, 0},
            {0, 0, 0, 0, 0, 0, 0, 7, 0, 2},
            {0, 0, 0, 0, 0, 0, 0, 0, 2, 0}
    };

    //The dist array is used to store the shortest distance from vertex 0 to other vertices
    vector<vector<int>> dist(numVertices, vector<int>(numVertices, INT_MAX));
    // Initialize the dist array and copy the values from A to dist
    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
            dist[i][j] = A[i][j];
        }
    }

    // Initialize the shortest path from vertex 0 to vertex 9 to the value in A
```

ISSN 2616-5775 Vol. 7, Issue 12: 79-84, DOI: 10.25236/AJCIS.2024.071211

```
    for (int j = 0; j < numVertices; ++j) {
        dist[0][j] = A[0][j];
    }

    // Floyd Marshall Algorithm Subject
    for (int k = 0; k < numVertices; ++k) {
        for (int i = 0; i < numVertices; ++i) {
            for (int j = 0; j < numVertices; ++j) {
                // If the path through vertex k is shorter, update dist [i] [j]
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] <
dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Output the shortest path length from vertex 0 to vertex 9
    cout << "The shortest distance from vertex 0 to vertex 9 is: " << dist[0][9] << endl;
    return 0;
}
```

Upon deeply analyzing the provided code implementation, we first notice a crucial initialization step: the creation and initialization of an array named dist. This step serves as the foundation for the entire algorithm execution process. The dist array is responsible for recording the shortest distances from every vertex in the graph to all other vertices. The way it is initialized directly determines whether the subsequent algorithm can run correctly and efficiently. Specifically, each element dist[i][j] in the dist array is initialized to the direct path length from vertex i to vertex j (if there is a direct connection between i and j), or to a value representing infinity (if there is no direct connection between i and j). This initialization strategy provides the basic data for gradually updating the shortest path lengths through the Floyd-Warshall algorithm.

Subsequently, the code cleverly designs a triple-nested loop structure to implement the core logic of the Floyd-Warshall algorithm. The core idea of this algorithm is to continuously iterate and update the values in the dist array to gradually explore and determine the shortest paths between all possible vertex pairs in the graph. Each iteration is based on an intermediate vertex k, attempting to update the shortest path length from vertex i to vertex j by using k as a transit point. If the path through k is shorter than the currently recorded shortest path, then the value of dist[i][j] is updated. This process not only embodies the core concept of gradually constructing solutions in dynamic programming algorithms, but also demonstrates the powerful capabilities and flexibility of the Floyd-Warshall algorithm in dealing with complex graph structure problems.

However, it is worth noting that the code is designed and implemented based on an important premise: there are no negatively weighted edges in the graph. This is because the Floyd-Warshall algorithm may encounter problems in correctly calculating the shortest paths when dealing with graphs containing negative weight cycles. Specifically, if there is a negative weight cycle, then starting from any point on the cycle, walking around the cycle any number of times can result in increasingly shorter paths, which can cause the algorithm to fail to converge to a determined shortest path length. Therefore, this premise ensures the correctness and reliability of the algorithm, allowing it to accurately calculate the shortest paths in graphs without negatively weighted edges.

Ultimately, after a series of complex iterations and updates, the code successfully outputs the length of the shortest path from vertex 0 to vertex 9, meeting the basic requirements of the problem. However, it is worth mentioning that the current code implementation mainly focuses on calculating the length of the shortest path and does not include logic for outputting the actual path. If, in practical applications, besides knowing the length of the shortest path, it is also necessary to find and display specific path information, then additional steps can be added during the execution of the Floyd-Warshall algorithm to record path information. Specifically, an additional two-dimensional array can be introduced to record the predecessor vertex for each vertex reached, so that after the algorithm ends, the shortest path from the starting point to the endpoint can be constructed through backtracking. In this way, the functionality of the code will be more complete and better able to meet diverse application requirements, such as path planning, network optimization, and more.

ISSN 2616-5775 Vol. 7, Issue 12: 79-84, DOI: 10.25236/AJCIS.2024.071211

7. Conclusion and Outlook

Dynamic programming algorithms have demonstrated remarkable application value in solving shortest path problems. This algorithmic framework, through clever strategy selection and adaptive application, can provide effective solutions for shortest path problems with different characteristics. Whether dealing with simple linear paths or facing complex network structures, dynamic programming can leverage its unique advantages to ensure the accuracy and efficiency of path solving.

Looking ahead, with the booming development in computer science and artificial intelligence, the application prospects of dynamic programming algorithms in the shortest path problem will become even broader. As the scale of data continues to expand and complexity continues to increase, the requirements for algorithm performance and flexibility are also increasing. Therefore, we have reason to believe that dynamic programming algorithms will play a more crucial role in this field, pushing the solution technology for the shortest path problem to new heights.

At the same time, we eagerly anticipate the innovation and development of dynamic programming algorithms in the future. We hope to see the emergence of more efficient, flexible, and adaptable new algorithms to better address the challenges of shortest path problems and other related fields. The emergence of these new algorithms will not only enhance the efficiency and quality of problem solving but also inject new vitality and momentum into the development of computer science and artificial intelligence.

References

[1] Sedgewick R, Wayne K. Algorithms [M]. Translated by Xie Luyun. 4th Edition. Beijing: People's Posts and Telecommunications Press, 2012
[2] Cormen T H, et al. Introduction to Algorithms [M]. Translated by Pan Jingui, et al. Beijing: China Machine Press, 2009
[3] Levitin A. Fundamentals of Algorithm Design and Analysis [M]. Translated by Pan Yan. Beijing: Tsinghua University Press, 2015
[4] Goodrich M T, et al. Algorithm Design and Applications [M]. Translated by Qiao Haiyan, et al. 3rd Edition. Beijing: China Machine Press, 2018
[5] Alsuwaiyel M H. Algorithmic Design Techniques and Analysis [M]. Translated by Wu Weichang, et al. Beijing: Publishing House of Electronics Industry, 2004
[6] Zhang Defu. Algorithm Design and Analysis [M]. Beijing: National Defense Industry Press, 2009
[7] Qu Wanling, Liu Tian, Zhang Liang, et al. Algorithm Design and Analysis [M]. 2nd Edition. Beijing: Tsinghua University Press, 2016
[8] Qu Wanling, Liu Tian, Zhang Liang, et al. Solutions and Learning Guidance for Algorithms Design and Analysis Exercises [M]. 2nd Edition. Beijing: Tsinghua University Press, 2016
[9] Wang Xiaodong. Computer Algorithm Design and Analysis [M]. 4th Edition. Beijing: Tsinghua University Press, 2018
[10] Li Chunbao. Algorithm Design and Analysis [M]. 2nd Edition. Beijing: Tsinghua University Press, 2018
[11] Li Chunbao, Li Xiaochi. In-depth Analysis of Algorithm Design for Programmer Interviews and Written Tests [M]. Beijing: Tsinghua University Press, 2018
[12] Li Chunbao, Data Structure Tutorial [M]. 5th Edition. Beijing: Tsinghua University Press, 2018