Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department: **Software Engineering and Automation**

CC - Laboratory Work Nr 7

Dynamic Programming

Ciuş Iurie FAF-203

Chişinău 2022

Contents

1 Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

1.1 Introduction

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Let's take the example of the Fibonacci numbers. As we all know, Fibonacci numbers are a series of numbers in which each number is the sum of the two preceding numbers. The first few Fibonacci numbers are 0, 1, 1, 2, 3, 5, and 8, and they continue on from there.

If we are asked to calculate the nth Fibonacci number, we can do that with the following equation,

$$Fib(n) = Fib(n-1) + Fib(n-2), for n > 1$$

As we can clearly see here, to solve the overall problem (i.e. Fib(n)), we broke it down into two smaller subproblems (which are Fib(n-1) and Fib(n-2)). This shows that we can use DP to solve this problem.

1.2 Control questions:

- Describe the dynamic programming method.
- Why is this method called dynamic programming?
- What is the difference between the divide et impera method and the dynamic programming?
- What is the classification of dynamic programming problems?
- What happens in the case when the costs of graph arcs processed with dijkstra and floyd algorithms are negative?

1.3 Theoretical Notes

A problem solvable by dynamic programming method it must first be brought to a discreet form in time. Decisions to be taken take to get a result must be able to take it step by step. Of also very important is the order in which they are taken. Dynamic programming is (and you don't take these rows as a definition) essentially a decision-making process in several stages: in the initial state of the problem we make the first decision, which determines a the new state of the problem in which we make a decision. The dynamic term is relates to this very thing: the problem is solved in stages time-dependent. Variables, or functions that describe each the stage must be so defined as to completely describe a process, so for this we will have to answer two questions:

- what is the initial stage (in which case we are dealing with a top-down decision-making) or what is the final stage (in which case we are dealing with an upward decision-making process)?
- what is the rule by which we go from one stage to another? Of usually this rule is expressed by a recurrence.

Because, we are dealing with a problem that is solved in May many stages, all we have to do is see how we make the decisions from one stage to another.

For example, the problem of calculating Fibonaci numbers is falls into the category of dynamic programming because:

- it is a phased process;
- each step k corresponds to the calculation of the k-th number Fibonacci;
- there is only one decision to move to a higher stage;

In the following, by strategy we mean a series of decisions. According to Bellman's principle, called the principle of optimality we have:

A strategy has the property that whatever its original state and the initial decision, the remaining decisions must constitute an optimal strategy regarding the state of the previous decision.

1.4 Description of The Algorithms

Dijkstra's shortest path algorithm

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3. While sptSet doesn't include all vertices

    - Pick a vertex u which is not there in sptSet and has a minimum distance value.
    - Include u to sptSet.
    - Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices 0, 1, 2, .. k-1 as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

- k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

- k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] ¿ dist[i][k] + dist[k][j]

2 Code

2.1 Implementation

Djikstra Algorithm

```
def dijkstra(graph_dict, start, end):
    if not start in graph_dict:
        raise ValueError('missing {0}'.format(start))
    if not end in graph_dict:
```

```python
        raise ValueError('missing {0}'.format(end))

    nodes = graph_dict.keys()

    f = float('inf')
    dist_from_start = {n: f for n in nodes}
    dist_from_start[start] = 0
    predecessors = {n: None for n in nodes}

    while len(nodes) > 0:
        candidates = {n: dist_from_start[n] for n in nodes}
        closest = min(candidates, key=candidates.get)  # ref
    footnote 2

        for n in graph_dict[closest]:
            if not n in dist_from_start:
                msg = 'missing node {0} (neighbor of {1})'.
    format(n, closest)
                raise ValueError(msg)
            dist_to_n = graph_dict[closest][n]
            if dist_to_n < 0:
                msg = 'negative distance from {0} to {1}'.format
    (closest, n)
                raise ValueError(msg)
            d = dist_from_start[closest] + dist_to_n
            if dist_from_start[n] > d:
                dist_from_start[n] = d
                predecessors[n] = closest

        nodes.remove(closest)

    return (dist_from_start, predecessors)


def shortestPath(graph_dict, start, end):
    distances, predecessors = dijkstra(graph_dict, start, end)

    if predecessors[end] is None and start != end:
        return [], distances[end]

    path = [end]
    while path[-1] != start:
        path.append(predecessors[path[-1]])
    path.reverse()

    return path, distances[end]


if __name__ == '__main__':
    from test import test_fixtures
    G = test_fixtures.test_fixtures['multinode']
    print(G)
    path, total_distance = shortestPath(G, 'a', 'e')
    msg = 'Shortest distance from a to e is {0}, total distance
    {1}.'
    print(msg.format(' -> '.join(path), total_distance))
```

djikstra.py

Floyd Warshall Algorithm

```
def floydwarshall(graph):
    distance = {}
    predesessor = {}
    for u in graph:
        distance[u] = {}
        predesessor[u] = {}
        for v in graph:
            distance[u][v] = 1000
            predesessor[u][v] = -1
        distance[u][u] = 0
        for neighbor in graph[u]:
            distance[u][neighbor] = graph[u][neighbor]
            predesessor[u][neighbor] = u

    for t in graph:
        for u in graph:
            for v in graph:
                newdistance = distance[u][t] + distance[t][v]
                if newdistance < distance[u][v]:
                    distance[u][v] = newdistance
                    # make a new route through t
                    predesessor[u][v] = predesessor[t][v]

    return distance, predesessor


graph = {0: {1: 6, 2: 8},
         1: {4: 11},
         2: {3: 9},
         3: {},
         4: {5: 3},
         5: {2: 7, 3: 4}}

distance, predesessor = floydwarshall(graph)
for v in predesessor:
    print("%s: %s" % (v, predesessor[v]))
for v in distance:
        print("%s: %s" % (v, distance[v]))
```

floyd.py

2.2 Complexity of the Algorithms

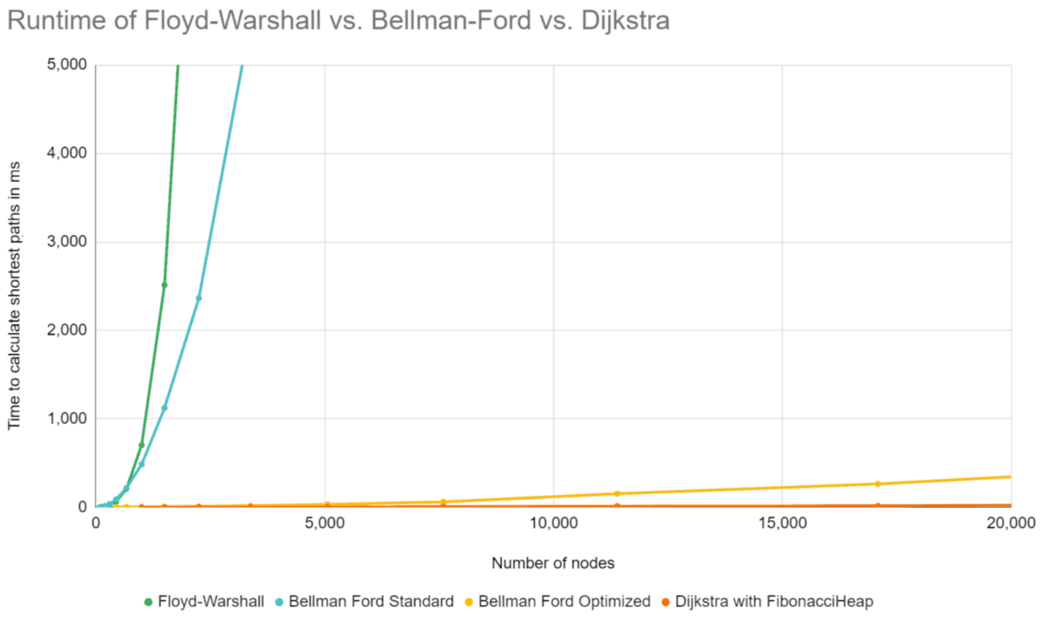
Djikstra

Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to $O(V + ElogV)$.

Floyd Warshall

The Floyd-Warshall algorithm is a graph-analysis algorithm that calculates shortest paths between all pairs of nodes in a graph. It is a dynamic programming algorithm with $O(|V|^3)$ time complexity and $O(|V|^2)$ space complexity.

2.3 Graphs



3 Conclusion

To conclude, the current piece of work represents my personal implementation of 2 algorithms - Djikstra and Floyd. The programming language

I chose is Python. The results show the time performance in mili-seconds based on the N. As you can see in the Figure 1, the Djikstra algorithms performed the best with a complexity of O(1), while the second one - $O(2^n)$.

3.1 References

- https://github.com/IuraCPersonal