Technical University of Moldova
Inginerical department S.A.

Report

№: 4

APA

Subject: Dynamic programming

Author: Terman Emil FAF161
Prof: M. Catruc

Chisinau 2017

1 Objectives

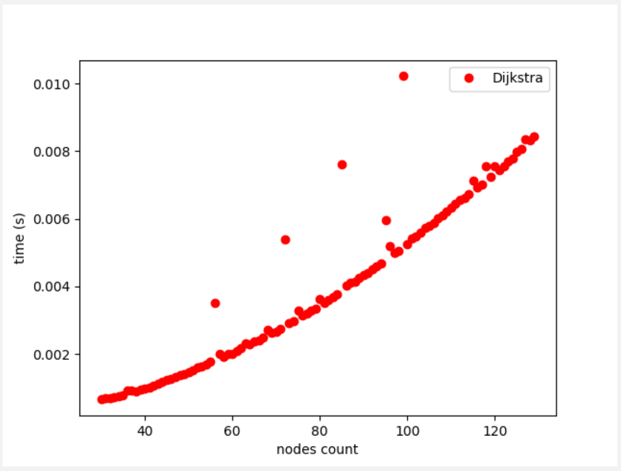
- study of dynamic programming;
- implementation of a few dynamic programming algorithms;
- comparison of greedy and dynamic programming;

2 Definition

In computer science, mathematics, management science, economics and bioinformatics, dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called ”memoization”.

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global solution, but it is often faster to calculate. Some greedy algorithms (such as Kruskal’s or Prim’s for minimum spanning trees) are however proven to lead to the optimal solution.

3 Dijkstra algorithm



Img 1: Dijkstra execution time

Pyhon Code 1: Dijkstra code

```python
import math

# Transforms from [[start_node_1, end_node_1, len], ...] to
# {(start_node_1, end_node_1): len, ...} choosing the smallest length.
def tree_to_dict(tree):
    tree_dict = dict()
    for start, end, length in tree:
        dict_key = (start, end)
        if dict_key in tree_dict:
            if length < tree_dict[dict_key]:
                tree_dict[dict_key] = length
        else:
            tree_dict[dict_key] = length
    return tree_dict

def shortest_path_dijkstra(tree):
    node_names = list(set(i[0] for i in tree))
    heap_map = dict((node_name, math.inf) for node_name in node_names)
    starting_node = tree[0][0]
    heap_map[starting_node] = 0
    tree = tree_to_dict(tree)

    dist_to_all_nodes = {starting_node: 0}
    best_connections = dict((node_name, None) for node_name in node_names)

    while len(heap_map) != 0:
        min_element = min(heap_map, key=heap_map.get)
        dist_to_all_nodes[min_element] = heap_map[min_element]
        dist_to_here = dist_to_all_nodes[min_element]
        del heap_map[min_element]

        for node in heap_map.keys():
            tree_key = tuple(sorted((node, min_element)))
            if tree_key in tree and tree[tree_key] + dist_to_here < heap_map[node]:
                heap_map[node] = dist_to_here + tree[tree_key]
                best_connections[node] = min_element

    return dist_to_all_nodes, best_connections
```

Complexity:

$$Time : O(E\log_2 V)$$

$$Space : O(E+V)$$

Where E are edges and V are vertices.

It works very fast, but it doesn't work with negative weights. Another drawback, is that it does a blind search, which may consume a lot of computation time.

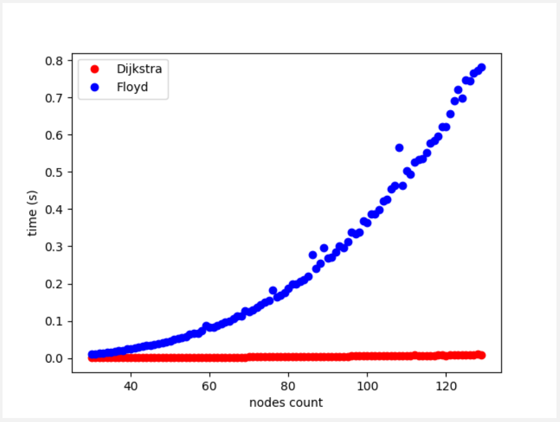
This algorithm is used in:

- Google maps;
- Geographical maps;
- IP routing to find Open shortest Path First;
- the telephone network;

4 Floyd algorithm

Pyhon Code 2: Floyd code

```
def print_floyd_matrix(my_matrix):
    nodes = list(my_matrix.keys())
    for i in nodes:
        for j in nodes:
            print("\t" + str(my_matrix[i][j]), end="")
        print("")

def get_all_nodes(tree):
    return list(set(i[0] for i in tree))

def get_initial_dist_matrix(tree):
    dist_matrix = dict()
    node_names = get_all_nodes(tree)
    for i in node_names:
        dist_matrix[i] = dict()
        for j in node_names:
            dist_matrix[i][j] = 0 if i == j else 100000

    for start, end, length in tree:
        dist_matrix[start][end] = length

    return dist_matrix

def get_initial_path_matrix(tree):
    path_matrix = dict()
    node_names = get_all_nodes(tree)

    for i in node_names:
        path_matrix[i] = dict()
        for j in node_names:
            path_matrix[i][j] = None

    for start, end, length in tree:
        path_matrix[start][end] = start

    return path_matrix

def shortest_path_floyd(tree):
    size = len(get_all_nodes(tree))
    node_names = get_all_nodes(tree)
    dist_matrix = get_initial_dist_matrix(tree)
    path_matrix = get_initial_path_matrix(tree)

    for k in node_names:
        for i in node_names:
            for j in node_names:
                if dist_matrix[i][j] > dist_matrix[i][k] + dist_matrix[k][j]:
                    dist_matrix[i][j] = dist_matrix[i][k] + dist_matrix[k][j]
                    path_matrix[i][j] = path_matrix[k][j]
    return dist_matrix, path_matrix
```



Img 2: Dijkstra and Floyd execution time

Complexity: n^3 where n is the number of nodes.
It works much slower, but:

- supports negative edges;
- supports directioned edges;

Applications:

- shortest paths in directed graphs;
- optimal routing. In this application one is interested in finding the path with the maximum flow between two vertices. This means that, rather than taking minima as in the pseudocode above, one instead takes maxima. The edge weights represent fixed constraints on flow. Path weights represent bottlenecks; so the addition operation above is replaced by the minimum operation;

5 Conclusion

In this laboratory work we studied dymaic programming. We implemented two shortest path algorithms: Dijkstra and Floyd algorithms. The Floyd–Warshall algorithm is a good choice for computing paths between all pairs of vertices in dense graphs, in which most or all pairs of vertices are connected by edges. For sparse graphs with non-negative edge weights, a better choice is to use Dijkstra's algorithm from each possible starting vertex, since the running time of repeated Dijkstra is better than the running time of the Floyd–Warshall algorithm when $|E|$ is significantly smaller.