Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)

Faculty of Informational Technologies and Programming

REPORT

# Discrete optimization algorithms (Sequential Search, Depth-First Search, Depth-First Branch-and-Bound, Iterative Deepening Search, Best-First Search)

Lecturer:

Chunaev Petr

Students:

Anastasiia Belykh, C4132

Mariia Bardina, C4132

Mikhail Sarafanov, C4134

Saint Petersburg, 2019

**Contents**

## Introduction

Formation of the modern information society requires new information technologies. The theory and practice of their development are based on optimization methods, in particular, discrete optimization.

Discrete Optimization aims to make good decisions when we have many possibilities to choose from. Discrete optimization problems are ubiquitous both in industry and theoretical computer science. They appear in a wide range of fields, such as manufacturing, planning, packing, design, and scheduling.

Discrete optimization means searching for an optimal solution in a finite or countably infinite set of potential solutions. This is in contrast to continuous optimization in which the variables are allowed to take on any value within a range of values. Two notable branches of discrete optimization are:

1) combinatorial optimization;

2) integer programming;

Combinatorial optimization is used when exhaustive search is not tractable. It consists of finding an optimal object from a finite set of objects. The shortest paths, flows and spanning trees are some examples of combinatorial optimization problems. Integer programming refers to problems where all of the variables are restricted to be integers [1].

The nature of discrete optimization problems is not simple and unambiguous. The simplest answer is as follows: these problems concern discrete mathematical models. The discrete mathematical models include transportation and simple economic models, a number of problems concerning optimal facility location, the choice of competitive behavior, and so on [2].

In this paper, we consider various optimization algorithms on graphs.

## 1. Theory

In this section, we consider the theoretical aspects of such algorithms as Sequential Search, Depth-First Search, Depth-First Branch-and-Bound, Iterative Deepening Search and Best-First Search.

Sequential search is a method that works with the list and find an element within one. Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. It is often used to test connectivity, search for loops and strongly connected components, for topological sorting and to find the longest path.

One of the most common tasks of discrete optimization is finding the shortest path between two vertices. To find the shortest past, we can use such algorithms as Depth-First Branch-and-Bound, Iterative Deepening Search and Best-First Search. The main ideas and tasks that can be solved with the help of these algorithms, will be considered further.

### 1.1. Sequential Search

The first algorithm we will consider is a linear search. It is also called a sequential search. The linear search is the most basic type of searching algorithm because his implementation is very simple.

Linear search is used to find a particular element in an array. When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection. So, if we run out of items, we have discovered that the item we were searching for was not present [3].

Consider the basic algorithm of sequential search.

Given a list K of n elements with values or records $K_1$ .... $K_n$, and target value T. We assume that $N \geq 1$. To find the index of the target T in K, we use linear search, which includes the following steps:

1) set i to 0;
2) if $K_i = T$, the search terminates successfully, return i;
3) increase i by 1;
4) if $i < n$, go to step 2. Otherwise, the search terminates unsuccessfully [4]

We see that this algorithm can terminate in two different ways, successfully having located the desired key or unsuccessfully having established that the given argument is not present in the table.

The diagram below shows how sequential search works if we look for number "8" in the list. We start at the first item in the list and move from item to item until we find "8". In our example to find the given element we needed 8 steps.


Figure 1. Sequential Search for number “8”

Time Complexity of the linear search is found by number of comparisons made in searching the element. In the best case, the given element is present in the first position of the array, i.e., only one comparison is made. So f(n) = O(1). In the Average case, the given element is found in the half position of the array, then f(n) = O(n/2). But in the worst case the given element is present in the last position of the array, so n comparisons are made. So, f(n) = O(n) [5].

Table 1: Comparisons Used in a Sequential Search of an Unordered List

| Case | Best case | Worst case | Average case |
|---|---|---|---|
| item is present | 1 | n | n/2 |
| item isn’t present | n | n | n |

Earlier the items in our collection had been randomly placed. Assume that the list of items was sorted ascending. If the item we are looking for is present in the list, the chance of it being in any one of the n positions is still the same as before. However, if the item is not present some advantages appear. In this case, the algorithm does not have to continue looking through all of the items if it finds the item which more than the desired value. On this step, the algorithm can stop and report that the item was not found [3].

Table 2: Comparisons Used in a Sequential Search of an Ordered List

| Case | Best case | Worst case | Average case |
|---|---|---|---|
| item is present | 1 | n | n/2 |
| item isn't present | 1 | n | n/2 |

If we have a sorted list it would be better to use binary search instead of linear search. In binary search the entire sorted list is divided into two parts. We first compare the desired element with the mid element of the list. Then we restrict our attention to only the first or second half of the list depending on whether the input item comes left or right of the mid element. In this way, we reduce the length of the list to be searched by half [5].

In summary linear search is very simple to implement and is useful when the list has only a few elements, or when performing a single search in an unordered list. When you need to search a lot of values in the same list, it often pays to pre-process the list to use a faster method. For example, one may sort the list and use binary search. However, if you work with a larger array or frequently changed array, it makes sense to use other, faster search method because the initial time to sort the data is comparable to many linear searches.

### 1.2. Depth-First Search

Depth First Search (DFS) is a general technique used in Artificial Intelligence for solving a variety of problems in planning, decision making, theorem proving, expert systems, etc. DFS is algorithm for traversing or searching a tree or graph.

If more details then DFS algorithm is a searching algorithm in which search progresses by expanding the first child node of the root that appears. Search going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring. The algorithm repeats this entire process until it has discovered every vertex. So, DFS is a recursive algorithm [6]. Figure 1 illustrates how DFS algorithm is worked.

The algorithm works in O(|V|+|E|) time where V is the number of vertices and E is the number of edges.

To implement the algorithm, it will be necessary to note which vertices were visited and which were not. We will make a mark in the visited list, where visited[i] equals "True" for visited vertices, and visited[i] equals "False" for unvisited. If we need to find the longest path between vertices, we also can create list with parents to record the parent of vertex i. We have parents[i] = NIL, if and only if vertex i is the root of a depth-first tree.

One application of DFS is finding strongly connected component. Let us be given a graph G = (V, E), where V is a set of vertices or nodes and E is a set of edges. Then strongly connected component (SCC) of G is a maximal set of vertices C subset of V, such that for all u, v in C, both u → v and v → u, that is, both u and v are reachable from each other. In other words, two vertices of directed graph are in the same component if and only if they are reachable from each other [7].

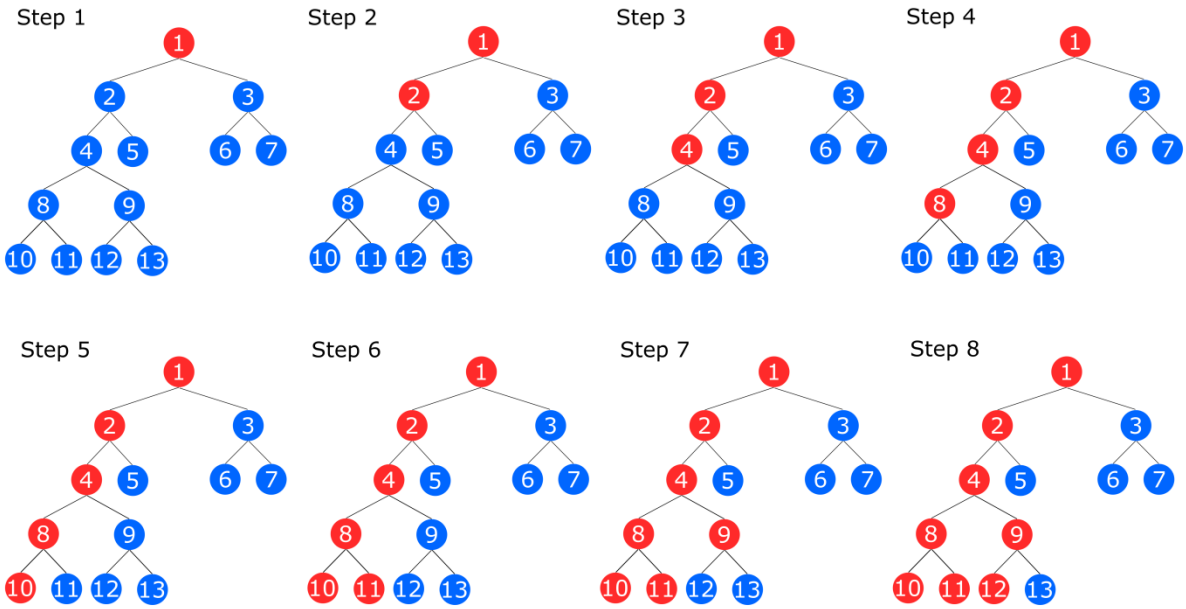


Figure 2. Depth-First Search

Other application of deep search is searching for loops. If a cycle is present in a graph, then a situation may arise when a DFS cannot visit all the vertices. These problems can be solved with the help of the Iterative Deepening Depth First Search algorithm, that will be considered further [7].

To sum up, DFS can be used to find all vertices reachable from a start vertex v, to determine if a graph is connected. There are a large number of applications which use DFS algorithm. The DFS is often used to test connectivity, search for loops and strongly connected components, and for topological sorting. It cannot be used to find the shortest unweighted paths, but longest path problem can be efficiently solved by DFS algorithm.

### 1.3. Depth-First Branch-and-Bound

The Branch-and-Bound was first suggested by Ailsa Land and Alison Doig [8]. It is commonly used for solving NP-hard discrete optimization problems. Although branch-and-bound is often discussed in the context of integer programming, it is actually a general

approach that can also be applied to solve many combinatorial optimization problems even when they are not formulated as integer programs. For example, there are efficient branch-and-bound algorithms specifically designed for job shop scheduling or quadratic assignment problems, which are based on the combinatorial properties of these problems and do not use their integer programming formulations.

A B&B algorithm searches the complete space of solutions for a given problem for the best solution. However, explicit enumeration is normally impossible due to the exponentially increasing number of potential solutions. The use of bounds for the function to be optimized combined with the value of the current best solution enables the algorithm to search parts of the solution space only implicitly [9]. The algorithm is presented on Figure 3.

Depth-First Branch-and-Bound explores the search space in a depth-first manner while keeping track of the current best-known solution cost, denoted $c^b$. It uses an admissible heuristic function h(·), i.e., a function that never overestimates the optimal cost-to-go for every node, and is guided by an evaluation function f(n) = g(n)+h(n), where g(n) is the cost of the path from the root node to n. Since f(n) is an underestimate of the cost of an optimal solution that goes through n, whenever $f(n) \geq c^b$, n is pruned.

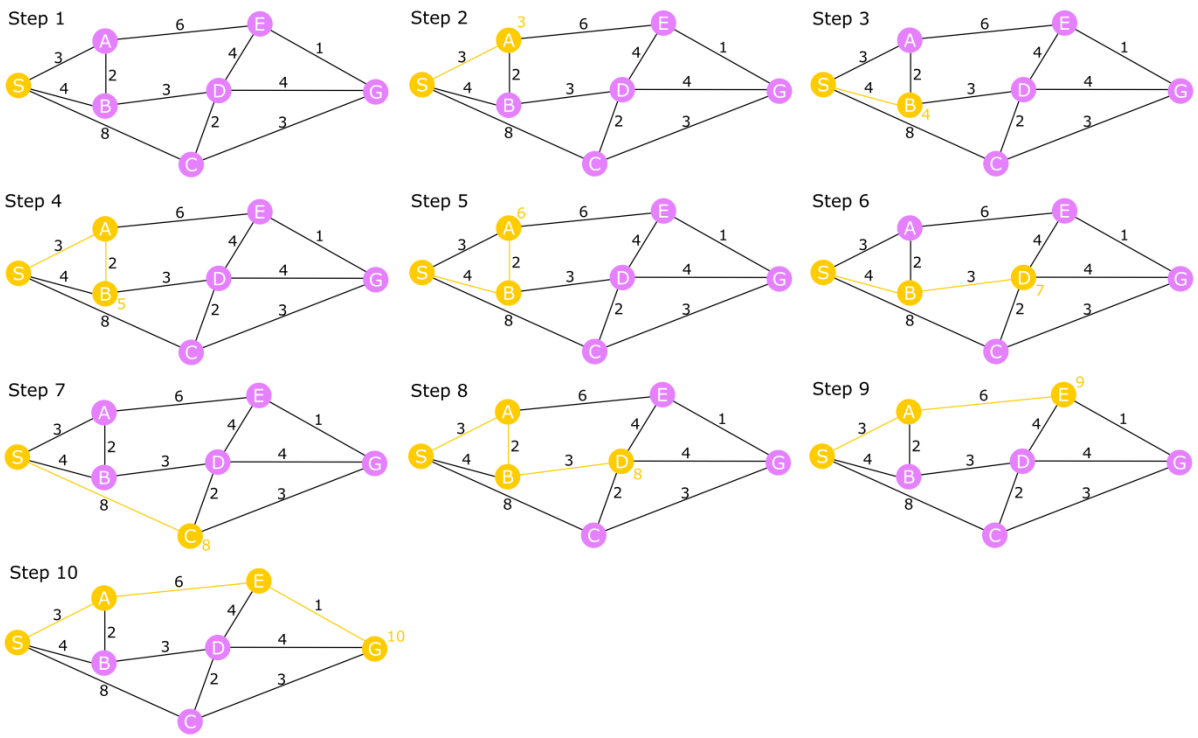


Figure 3. Depth-First Branch-and-Bound

### 1.4. Iterative Deepening Search

The complexities of various search algorithms are considered in terms of time, space, and cost of solution path. It is known that breadth-first search requires too much space and depth-first search can use too much time and doesn't always find a cheapest path. A depth-first iterative-deepening algorithm is shown to be asymptotically optimal in some situations. The algorithm has been used successfully in chess programs, has been effectively combined with bi-directional search, and has been applied to best-first heuristic search as well [10].

It should be noted that Iterative Deepening Search has different refinements. It is used to modify the algorithm A* (IDA*) also [11]. Similar implementations, as stated in [12], are used in path-finding. So, the algorithm is included in network traffic, robot planning, military simulations, and computer games. Typically, a grid is superimposed over a region, and a graph search is used to find the optimal (minimal cost) path. The most common scenario is to use a grid of tiles and to search using A* or IDA*. Application-dependent properties dictate which grid representation and search algorithm will yield the best results.

For a problem with branching factor b where the first solution is at depth k, the time complexity of iterative deepening is $O(b^k)$, and its space complexity is O(bk). This means that iterative deepening simulates breadth-first search, but with only linear space complexity.

The implementation of the algorithm on an unweighted graph is as follows:

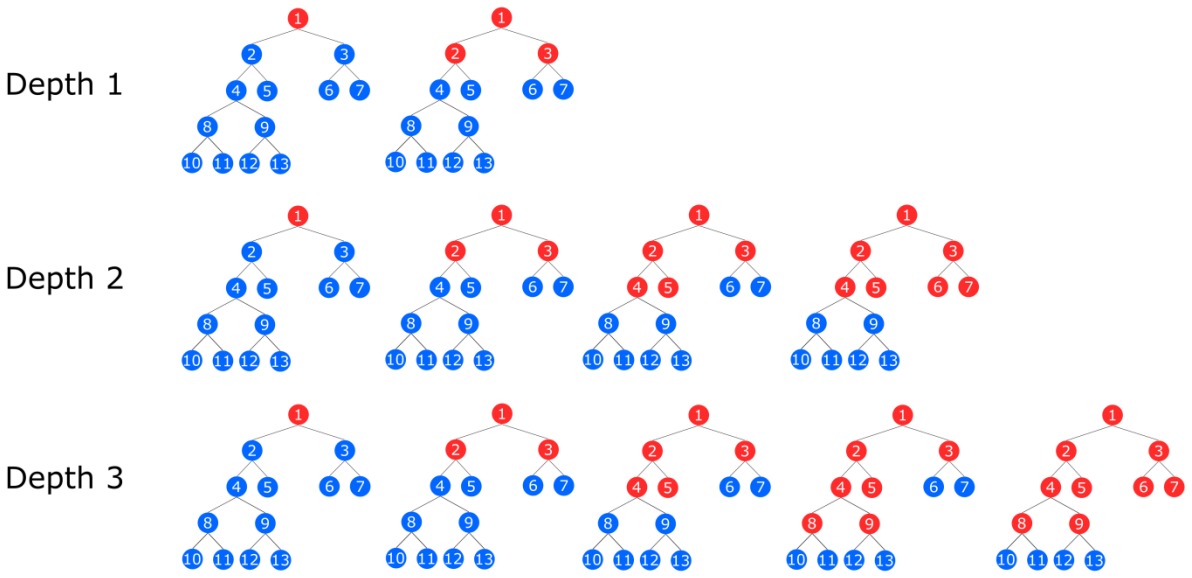


Figure 4. Iterative Deepening Search

The algorithm is being improved for specific tasks. For example, it included in MR-Search (MR-Search is a framework for massively parallel heuristic search), where were

implemented two node expansion strategies: breadth-first frontier search and breadth-first iterative deepening A* [13].

Unfortunately, iterative deepening only performs well when successive cost bounds visit a geometrically increasing number of nodes [14].

### 1.5. Best-First Search

The last algorithm to consider is Best-First search. This algorithm is an instance of graph search algorithm in which a node is selected for expansion based on evaluation function f (n). Usually, the node which is the lowest evaluation is selected for the explanation.

Let's consider a main concept of this algorithm.

Step 1: Traverse the root node.

Step 2: Traverse any neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue.

Step 3: Traverse any neighbour of neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue

Step 4: This process will continue until we are getting the goal node.

Special case of best-first search algorithm is the A* algorithm [15], which uses an evaluation function f(n)= g(n) +h(n), where g(n) is the sum of the cost of the path from initial state to the current node n, and h(n) is an estimated cost from node n to a goal node.

The implementation of the algorithm is presented in the Fig. 5.

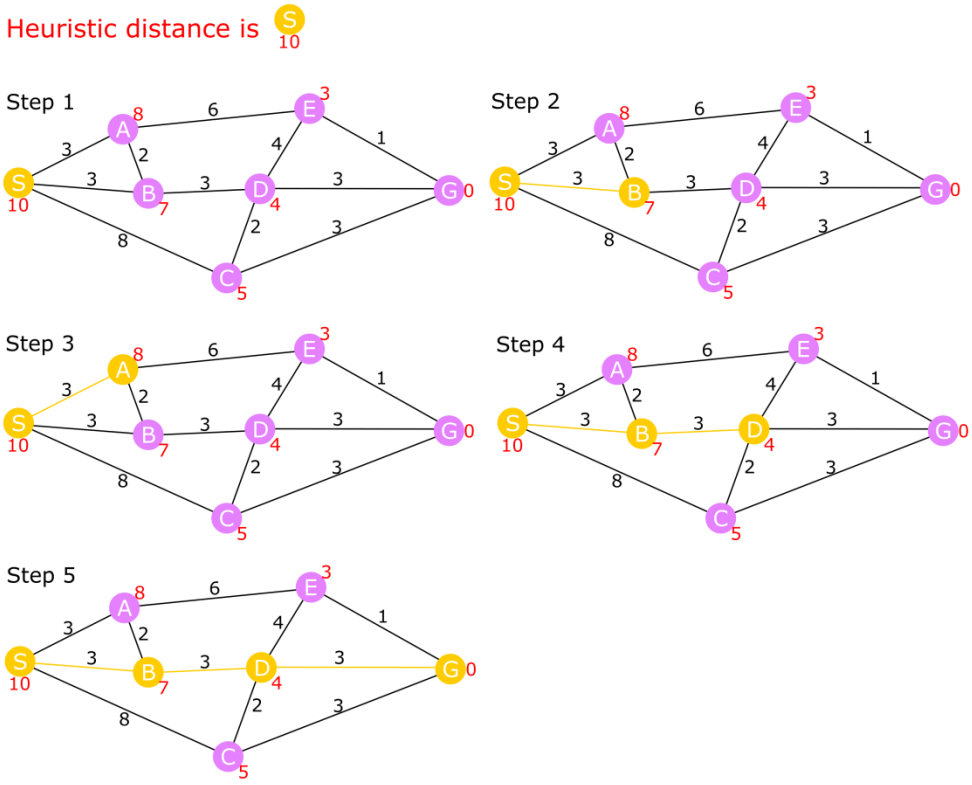


Figure 5. Algorithm A*

A* algorithm find it's application in many fields. It can be used for solving the common pathfinding problems in computer games as well as in robotics. It also can be used for parsing using stochastic grammars in NLP [16] and for Informational search with online learning [17]. Time complexity is O(|E|).

## 2. Practical application

As an example, we took a dataset called “USAir97”. Dataset contains information on the duration of flights between US cities in 1997. We can represent these data as an undirected weighted graph. Consequently, nodes in our graph are cities, edges are routes between cities and weights are flight duration.

Initially, the graph is presented as the edge list with weights. Graph consists of 332 nodes and 2126 edges. Minimum degree of a graph is 1, maximum degree is 139 and the average degree consists of 12 edges.

The graph illustration is presented on figure 6.

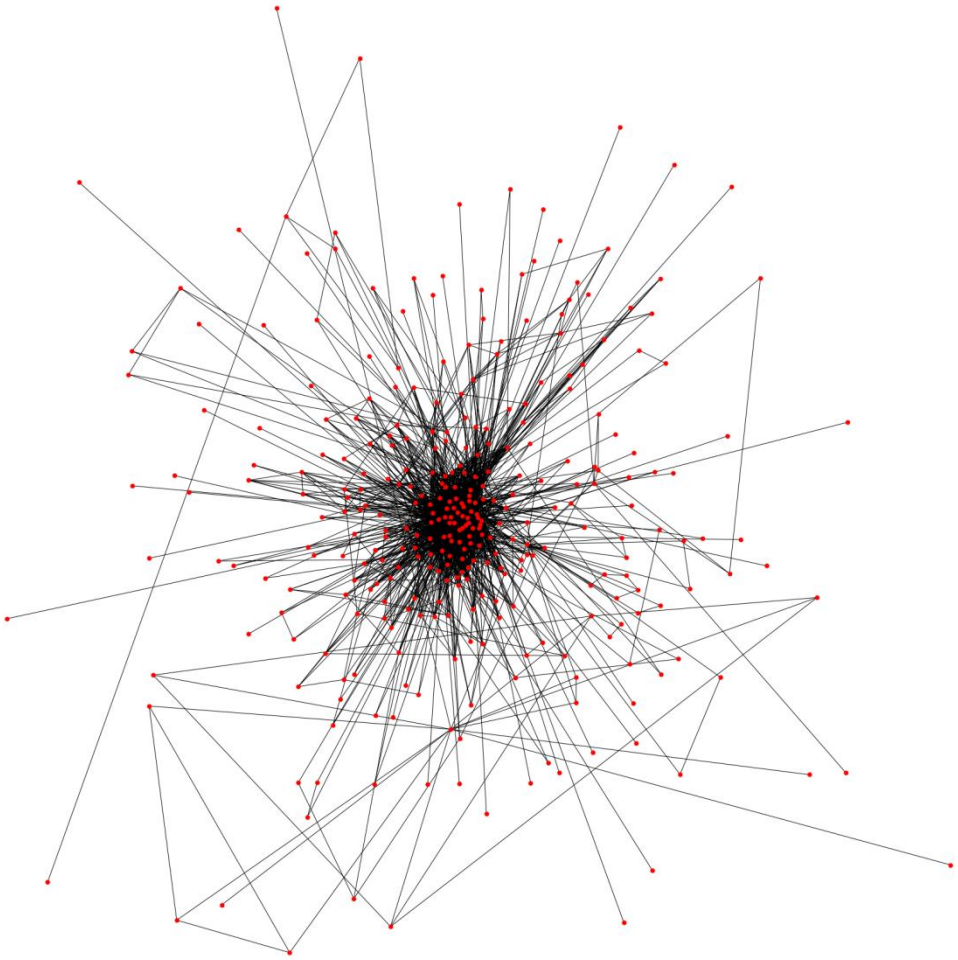

Figure 6. Graph “USAir97"

Representation of the graph as an adjacency matrix you can see in the figure 7.

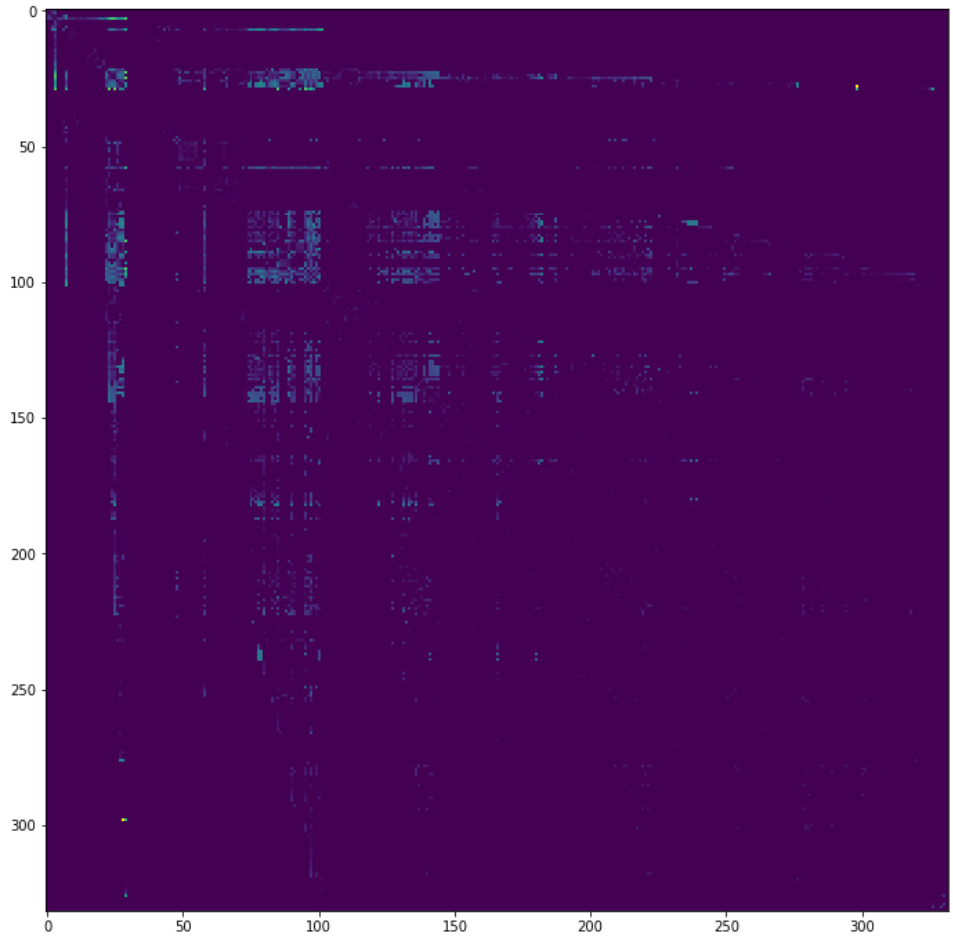


Figure 7. Adjacency matrix of graph “USAir97"

Based on the figures, we can conclude that there are vertices with many adjacent vertices and those with only a few.

In this section, we will find the shortest path between two cities. We will solve this task using algorithms discussed in the previous section. We will present the implementation such algorithms as Depth-First Branch-and-Bound, Iterative Deepening Search and Best-First Search and compare their effectiveness.

### 2.1. Depth-First Search

Before you find the shortest path, you need to find out whether our graph is connected. To do this, find the number of connected components using the depth-first search.

Below is a DFS implementation in Python.

```
def dfs(visited, graph, node):
    visited[node] = True
    for w in graph[node]:
        if not visited[w]:
            dfs(visited, graph,w)
```

Figure 8. Depth-First Search

Next we use the function dfs to find the number of connected components. If after one step of dfs we have unvisited nodes, we increase number of connected components by 1. Else we visited all nodes and number of connected components of the graph equals 1.

```
def connect_comp(graph):
    n = len(graph)
    # List of visited nodes
    visited = [False] * n
    num_components = 0
    for v in range(n):
        if not visited[v]:
            dfs(visited, graph,v)
            num_components += 1
    print('Number of the connected components of the graph - %i' % num_components)
```

Figure 9. Finding the number of connected components

We give the number of connected components of the graph equals 1. So, our graph is connected, because it has exactly one connected component. It means that there is a path between every pair of vertices.

### 2.2. Depth-First Branch-and-Bound

In the project, it was decided to compare the operating time of the algorithms for finding shortest path using the USAir97 graph as an example. To present our graph NetworkX library was used. However not all search algorithms were presented in the library as well. For the implementation of the Depth-First Branch-and-Bound algorithm third party implementation was used. The function that helps to bring NetworkX's graph format to that implementation format and then calculate the path is presented on Figure 10.

```
def DFBnB(G,start,goal):
    d = {}
    adj_matrix = nx.adjacency_matrix(G)
    adj_matrix = adj_matrix.toarray()
    adj_list={}
    for i in range(G.number_of_nodes()):
        res=[i for i, e in enumerate(adj_matrix[i]) if e != 0]
        adj_list.update({i:res})
    s='\',\''
    adj=[str(i) for i in G.nodes]
    adjString=str(G.nodes)
    adjString=s.join(adj)
    exString = 'problem = Search_problem_from_explicit_graph({\''+adjString[:-1]+'\'}, ['
    for edge in G.edges.data('weight', default=1):
        exString+='Arc(\''+str(edge[0])+'\',\''+ str(edge[1])+ '\',' +str(float(edge[2]))+ r'),'
    exString = exString[:-1] + '], start = \''+str(start)+'\', goals = {\''+str(goal)+'\'})'
    exec(exString, globals())
    B.dfbnb(problem)
```

Figure 10. DFBnB function

### 2.3. Iterative Deepening Search

The following function has been prepared for the Iterative Deepening Search algorithm:

```
# Iterative Deepening Search
def IDS(G, start, goal):
    visited = []

    def bfs(G, start, goal = goal):
        visited.append(start)
        # From the selected vertex, we look for all neighboring
        bfs_result = list(nx.bfs_successors(G, source = start, depth_limit = 1))
        bfs_result = np.array(bfs_result[0][1]) # Список всех найденных вершин

        # Make a list of vertices that the algorithm has not yet visited
        unvisited_bfs_result = []
        for vertex in bfs_result:
            for i in visited:
                if vertex == i:
                    pass
                else:
                    unvisited_bfs_result.append(vertex)
        unvisited_bfs_result = np.array(unvisited_bfs_result)

        # Check if the vertex of interest to us is found
        if any(goal == vertex for vertex in unvisited_bfs_result):
            return('The goal was found!')
        else:
            return(unvisited_bfs_result)

    result_1 = bfs(G, start = start, goal = goal)
    if result_1 == 'The goal was found!':
        return([start,goal])

    for vertex in result_1:
        result = bfs(G, start = vertex, goal = goal)
        if result == 'The goal was found!':
            return([start,vertex,goal])

        # Add the nodes that we may check in the future
        potential = []
        for i in result[1:]:
            potential.append(i)

        result = bfs(G, start = result[0], goal = goal)
        if result == 'The goal was found!':
            return([start,vertex,result[0],goal])
        else:
            for i in result[1:]:
                potential.append(i)
            for j in potential:
                result = bfs(G, start = j, goal = goal)
                if result == 'The goal was found!':
                    return([start,vertex,goal])
    print('No solution found')
```

Figure 11. IDS function

The algorithm is not developed to search a path on a weighted graph, so it was believed that the algorithm found the "shortest path", if it found a solution at all.

### 2.4. Best-First Search

For Best-First Search, we used the implementation of the A* algorithm from the NetworkX library. The function call is presented on the Figure 12.

```
nx.astar_path(G, start, goal)
```

Figure 12. A* function from NetworkX library

All functions presented were unified, as they take graph, start node and goal node as arguments and return the path as result.

**Conclusion**

The presented algorithms were used to find the shortest path in the graph. To compare the algorithms by the operating time, 1 starting point and several vertices were chosen, to which the algorithm should find the shortest path. They are presented on Figure 13.

```
start = 2
goals = [5, 6, 7, 8, 40, 3, 10, 20]
```

Figure 13. Initial conditions

Each algorithm was run 10 times. During each run, the operating time was detected. The results of the experiments are presented in Figures 14, 15.

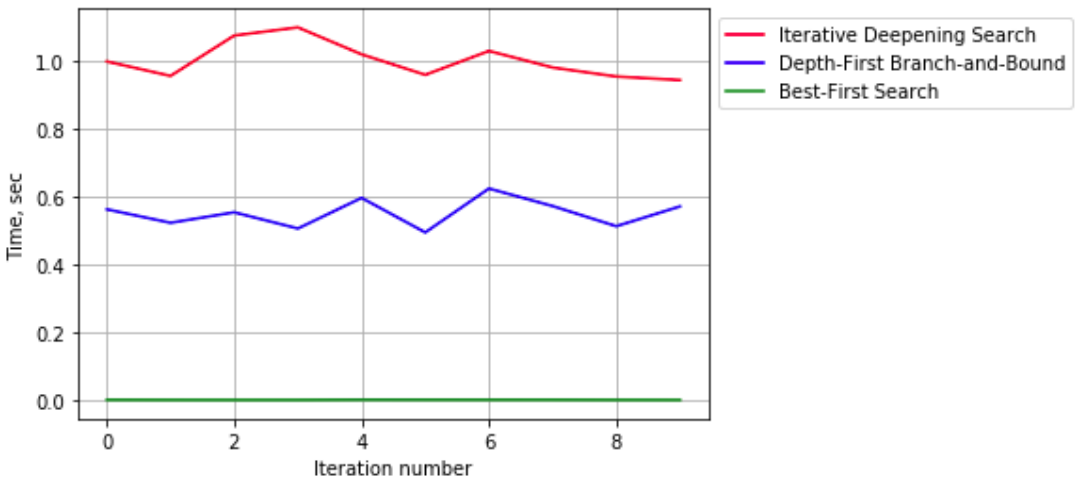


Figure 14. Execution time during iterations

For clarity, boxplot is presented, where it is clearly seen that the Best-First Search algorithm is the best in terms of operating time.

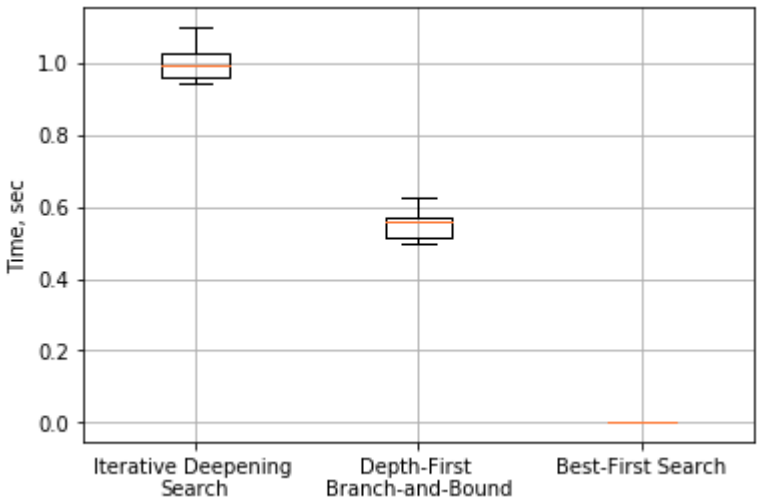


Figure 15. Boxplot presentation of the results

Thus, for the problem of finding the best route from starting point to various endpoints on the USAir97 graph the Best-First Search algorithm is proved to be most effective in terms of runtime. Though it is important to note, that the implementations of algorithms came from different sources therefore conditions for comparison are not ideal.

**List of references**


1. Michael Junger, Gerhard Reinelt, Optimization and operations research – Vol. II - Combinatorial Optimization and Integer Programming. Chapter in book Encyclopedia of Life Support Systems EOLSS, 2004.
2. Leont'ev V, Discrete optimization. Computational Mathematics and Mathematical Physics, Vol. 47, No. 2, pp. 318–352. 2007. DOI: 10.1134/S0965542507020157
3. Miller B, Ranum D, Problem Solving with Algorithms and Data Structures using Python. 2013.
4. Knuth, Donald, Sorting and Searching. The Art of Computer Programming. 1998.
5. Debadrita Roy, Arnab Kundu, A Comparative Analysis of Three Different Types of Searching Algorithms in Data Structure. International Journal of Advanced Research in Computer and Communication EngineeringVol. 3, Issue 5, 2014.
6. Navneet kaur, Deepak Garg. Analysis Of The Depth First Search. Algorithms. International Journal of Data Mining Knowledge Engineering, 2012.
7. Gaurav Rathi, Dr. Shivani Goel, Applications of Depth First Search: A Survey, International Journal of Engineering Research & Technology (IJERT), Vol. 2 Issue 7, p. 1341-1347, 2013.
8. A. H. Land and A. G. Doig, An automatic method of solving discrete programming problems. Econometrica. 28 (3). pp. 497–520, 1960.
9. Jens Clausen, Branch and Bound Algorithms - Principles and Examples. 1999
10. Richard E. Korf , Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence 27(1) 97-109, 1985.
11. Nie X, Plaisted D, Refinements to depth-first iterative-deepening search in automatic theorem proving. Artificial Intelligence, 41(2) 223-235, 1989.
12. Yap P, Grid-based path-finding. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2338 44-55, 2002.
13. Schütt T, Reinefeld A, Maier R, MR-search: Massively parallel heuristic search. Concurrency Computation Practice and Experience, 25(1) 40-54, 2013.
14. Burns E, Ruml W, Iterative-deepening search with on-line tree size prediction. Annals of Mathematics and Artificial Intelligence, 69(2) 183-205, 2013.

15. P. E.; Nilsson, N. J.; Raphael, B, A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics. 4 (2): 100–107, 1968.
16. Klein, Dan and Christopher D. Manning, A* Parsing: Fast Exact Viterbi Parse Selection, HLT-NAACL, 2003.
17. Kagan E. and Ben-Gal I, A Group-Testing Algorithm with Online Informational Learning (PDF). IIE Transactions, 46:2, 164-184, 2014.