Ministry of Science and Higher Education of the Russian

Federation National Research University ITMO

Evolutionary Computation

Spring

2024

Laboratory work No. 1

COMPLEXITY OF ALGORITHMS AND THEIR OPTIMIZATION

1. Implement the algorithm in any language according to the assignment option.

- The algorithm option:

Bellman Ford algorithm.

- The optimization method applied:

Early stopping.

- The definitions needed (explanation is in comments between code lines):

A) Defining graphs:

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])
```

B) Defining the method that implements the normal algorithm:

```
    def bellman_ford(self, src):
        # the initial values for the path between source
        # and other vertices is inf
        # and zero with the source himself
        dist = [float("Inf")] * self.V
        dist[src] = 0

        start_time = time.time()
        start_cpu_time = time.process_time()

        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        for u, v, w in self.graph:
          #maximum number of the updating loops is v-1, so if there
          #are still updates, this means that there are
          #negative loops
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return None, None

        end_time = time.time()
        end_cpu_time = time.process_time()

        return end_time - start_time, end_cpu_time - start_cpu_time
```

C) Defining the method that implements the optimized version of the algorithm:

```
41      def optimized_bellman_ford(self, src):
42          dist = [float("Inf")] * self.V
43          dist[src] = 0
44
45          start_time = time.time()
46          start_cpu_time = time.process_time()
47
48          for _ in range(self.V - 1):
49              changed = False
50              for u, v, w in self.graph:
51                  if dist[u] != float("Inf") and dist[u] + w < dist[v]:
52                      dist[v] = dist[u] + w
53                      changed = True
54              #if in some loop no changes happened, this means that
55              #there won't be changes anymore in further loops.
56              if not changed:
57                  break
58
59          for u, v, w in self.graph:
60              if dist[u] != float("Inf") and dist[u] + w < dist[v]:
61                  print("Graph contains negative weight cycle")
62                  return None, None
63
64          end_time = time.time()
65          end_cpu_time = time.process_time()
66
67          return end_time - start_time, end_cpu_time - start_cpu_time
```

D) Defining a function for generating a random graph with specific number of vertices and edges:

```
1 def generate_random_graph(vertices, edges):
2     g = Graph(vertices)
3     for _ in range(edges):
4         u = random.randint(0, vertices - 1)
5         v = random.randint(0, vertices - 1)
6         w = random.randint(-10, 10)  # Random weight between -10 and 10 for having realistic results
7         g.add_edge(u, v, w)
8     return g
```

E) Defining function for making statistics about the algorithm with different dimensions, with multiple runs (here it's 3 runs):

```
 1 def average_times(func, src, num_vertices, num_edges, num_runs=3):
 2     total_wall_clock_time = 0
 3     total_cpu_time = 0
 4     valid_runs = 0
 5     while valid_runs < num_runs:
 6         g = generate_random_graph(num_vertices, num_edges)
 7         wall_clock_time, cpu_time = func(g, src)
 8         if wall_clock_time is not None and cpu_time is not None:
 9             total_wall_clock_time += wall_clock_time
10             total_cpu_time += cpu_time
11             valid_runs += 1
12     return total_wall_clock_time / num_runs, total_cpu_time / num_runs if valid_runs > 0 else 0
```

F) Code for running statistics, taking into consideration not exceeding maximum number of possible edges (v*(v-1)):

```
results = []

for i in range(100, 1001, 100):
    print(f"Number of vertices: {i}")
    num_edges = min(2 * i, i * (i - 1))  # Ensure edges don't exceed maximum for directed graphs
    bf_wall_clock_time, bf_cpu_time = average_times(Graph.bellman_ford, 0, i, num_edges)
    obf_wall_clock_time, obf_cpu_time = average_times(Graph.optimized_bellman_ford, 0, i, num_edges)
    results.append([i, num_edges, bf_wall_clock_time, bf_cpu_time, obf_wall_clock_time, obf_cpu_time])

print(tabulate(results, headers=["Vertices", "Edges", "Bellman-Ford Wall Clock Time", "Bellman-Ford CP
```

G) Results of running statistics (Memory usage was not calculated because it was neglected and it was always being zero:

Vertices	Edges	Bellman-Ford Wall Clock Time	Bellman-Ford CPU Time	Optimized Bellman-Ford Wall Clock Time	Optimized Bellman-Ford CPU Time
100	200	0.00355069	0.00357921	0.000105063	9.17837e-05
200	400	0.0150342	0.014889	0.00031511	0.000319925
300	600	0.0355708	0.0335433	0.000227928	0.000225215
400	800	0.0547238	0.054504	0.000352144	0.000349116
500	1000	0.0935667	0.0913511	0.000741323	0.000738841
600	1200	0.120631	0.120561	0.000616312	0.00061337
700	1400	0.167796	0.167636	0.000954707	0.000984893
800	1600	0.303891	0.283095	0.00164167	0.0017103
900	1800	0.36115	0.350252	0.000780185	0.000778262
1000	2000	0.34738	0.344957	0.00113829	0.00113617

In the previous results, we can see that time consumed is proportional with V*E, for example, in the first two liens, we can see that:

$$\frac{V_2 * E_2}{V_1 * E_1} = \frac{400 * 200}{200 * 100} = 4 \approx \frac{clock\_T_2}{clock\_T_1} = 4.3$$

And in the second two lines

$$\frac{V_4 * E_4}{V_3 * E_3} = \frac{800 * 400}{600 * 300} = 0.15 \approx \frac{clock\_T_4}{clock\_T_3} = 1.53$$

2. Calculate the complexity of the algorithm, provide calculations, results of load tests, measuring the time and resources spent.

The complexity of the Bellman-Ford algorithm is as follows:

Time Complexity:

a) The main loop runs V-1 times, where V is the number of vertices.

b) Inside this loop, there is another loop iterating over all edges, which takes O(E) time, where E is the number of edges.

So, the overall time complexity is **O(V * E)**.

Space Complexity:

a) The space complexity is O(V), where V is the number of vertices, for storing the distance array.

3. Perform both algorithmic optimization, if possible, with the removal of an invariant, for example, and program methods of the selected language.

The code of the optimized algorithm is included in the previous code, and results of it are printed along with the normal version of the algorithm.

4. Calculate the complexity of the optimized algorithm, provide calculations, results of load tests measuring the time and resources spent.

To calculate the complexity of the optimized Bellman-Ford algorithm, we need to analyze its time complexity and space complexity.

Time Complexity:

a) In the worst case scenario, where there are no negative cycles, the optimized Bellman-Ford algorithm performs a single pass through all edges for each vertex. Since there are V vertices and E edges, the time complexity is O(V * E).

b) However, the optimized version has the potential to terminate early if no updates are made during a pass through all edges. This can reduce the number of iterations compared to the standard Bellman-Ford algorithm, leading to improved performance in practice, where best case scenario is O(V) where it needs at least to do one loop to make sure that there won't be any updates, and terminate.

Space Complexity:

- The space complexity of the optimized Bellman-Ford algorithm is the same as the normal one, which is O(V), where V is the number of vertices. This space is used to store the distances from the source vertex to all other vertices.
- Additionally, the space complexity of the graph representation is O(E), where E is the number of edges.

5. Describe the difference in complexity values, results, give justification.

a) Wall Clock Time Comparison:

- For both the Bellman-Ford and Optimized Bellman-Ford algorithms, the wall clock time increases as the number of vertices and edges in the graph increases. This is expected, as larger graphs require more computation time to find the shortest paths.
- However, the increase in wall clock time for the Bellman-Ford algorithm is much more significant compared to the Optimized Bellman-Ford algorithm. The optimized version demonstrates significantly lower wall clock times across all graph sizes.

b) CPU Time Comparison:

- Similar to wall clock time, CPU time also increases with the size of the graph for both algorithms. This is because larger graphs require more computational effort to process.
- The CPU time for the Bellman-Ford algorithm is consistently higher than that of the Optimized Bellman-Ford algorithm for all graph sizes. This suggests that the optimized version is more efficient in utilizing CPU resources.

c) Complexity Difference:

- The standard Bellman-Ford algorithm has a time complexity of O(V * E), where V is the number of vertices and E is the number of edges. This complexity arises from the need to iterate over all edges for each vertex.
- The optimized Bellman-Ford algorithm has a similar time complexity of O(V * E) in the worst case. However, it often terminates early if no updates are made during an iteration, leading to improved performance in practice.
- The significant reduction in wall clock and CPU times for the optimized algorithm compared to the standard algorithm suggests that the optimization effectively reduces the actual computational workload, despite having the same worst-case time complexity.

d) Justification:

- The results validate the effectiveness of the optimization in reducing computational time and CPU usage. By terminating early when no updates are made, the optimized algorithm avoids unnecessary iterations, leading to faster execution.
- The complexity analysis aligns with the observed results, as both algorithms exhibit similar time complexities. However, the practical performance of the optimized algorithm outperforms the standard algorithm due to its early termination condition.
- Overall, the optimization demonstrates improved efficiency in solving the shortest path problem, making it a preferable choice for real-world applications where performance is critical.

6. Formulate conclusions.

The optimized Bellman-Ford algorithm demonstrates superior practical performance compared to the standard algorithm (especially when the graph doesn't contain negative cycles), achieving lower wall clock times and CPU usage across various graph sizes. Despite having the same worst-case time complexity, the optimization significantly reduces computational workload by terminating early when no updates are made.