IE801B: Logistics Systems Optimization **Lecturer:** Prof. Changhyun Kwon
Assignment #1 **Author:** Jaewoo Kim
2024.03.19. **Student ID:** 20245084

1 Introduction

This report seeks to validate the statement by Ahuja et al. (1993, p. 143) as referenced in the assignment guide. It details two primary tasks undertaken for this purpose:

- The development of Julia functions to implement two variations of the Bellman-Ford algorithm. The first variation employs a First-In-First-Out (FIFO) queue, while the second uses a double-ended queue (deque) as its data structure.
- The execution of experiments on networks with varying structures to evaluate the performance of these algorithms.

No AI tools were utilized for the development and experimental analysis. However, AI assistance was sought for refining the report's language and expression, specifically employing GPT-4.0 for sentence structure and expression enhancement and Grammarly for grammar checks.

Two Julia scripts are submitted:

- **ShortestPathProblems.jl**: Key functions are defined in this script.
- **main.jl**: This script is the main file performing the experiment by importing functions defined in 'ShortestPathProblems.jl.' It generates instances, solves them, and saves the results. The generated instances are saved in the 'instances' folder as .csv files, and the results are saved in the 'results' folder as .csv files.

The remainder of this report is organized into four sections:

- Section 2 introduces the codes implementing the algorithms, providing a brief overview.
- Section 3 provides the experimental design and the generation of instances briefly, describing the methodology and parameters used for testing the algorithms.
- Section 4 presents the experiments' results, analyzing the algorithms' performance across various testing cases.
- Section 5 concludes the report by summarizing the findings and their implications.

2 Algorithms & codes summary

The function implementing the algorithms is in 'ShortestPathProblems.jl' script:

```julia
function ModifiedBellmanFordAlgorithm(A, s::Int64 = 1, use_dequeue::Bool = true)
    #* preprocessing
    calV = maximum(A[:, 1:2])
    calE = size(A)[1]
    pnt = [findfirst(x->x==v, A[:, 1]) for v in 1:1:(calV+1)]
    pnt[calV+1] = calE+1
    tpv1 = findall(x->isnothing(x), pnt)
    tpv2 = deleteat!(collect(1:1:(calV+1)), tpv1)
    for v in tpv1
        pnt[v] = pnt[tpv2[findfirst(x->x>v, tpv2)]]
    end

    #* initialization
    neg_thr = sum(A[:, 3] .* (A[:, 3] .< 0))
    d = fill(Inf, calV); d[s] = 0
    pred = Vector{Int64}(undef, calV); pred[s] = 0
    if use_dequeue
        LIST = Deque{Int64}(); push!(LIST, s)
        LIST_hist = Queue{Int64}(); enqueue!(LIST_hist, s)
    else
        LIST = Queue{Int64}(); enqueue!(LIST, s)
    end
    flag_nc = false

    #* algorithm
    n_ex = 0    # the number of node examinations
    time::Float64 = 0
    if use_dequeue
        time = @elapsed while !isempty(LIST)
            i = popfirst!(LIST)
            for ind in pnt[i]:1:(pnt[i+1]-1)
                j = A[ind, 2]
                c_ij = A[ind, 3]
                n_ex += 1
                # check optimality condition
                if d[j] > d[i] + c_ij
                    # update label
                    d[j] = d[i] + c_ij
                    # negative cycle detection
                    if d[j] < neg_thr
                        flag_nc = true
                        break
                    end
                    pred[j] = i
                    if !(j in LIST)
                        if j in LIST_hist
                            pushfirst!(LIST, j)
                        else
                            push!(LIST, j)
                            enqueue!(LIST_hist, j)
                        end
                    end
                end
            end
        end
    else
        time = @elapsed while !isempty(LIST)
            i = dequeue!(LIST)
```

```
234             for ind in pnt[i]:1:(pnt[i+1]-1)
235                 j = A[ind, 2]
236                 c_ij = A[ind, 3]
237                 n_ex += 1
238                 # check optimality condition
239                 if d[j] > d[i] + c_ij
240                     # update label
241                     d[j] = d[i] + c_ij
242                     # negative cycle detection
243                     if d[j] < neg_thr
244                         flag_nc = true
245                         break
246                     end
247                     pred[j] = i
248                     if !(j in LIST)
249                         enqueue!(LIST, j)
250                     end
251 
252                 end
253             end
254         end
255     end
256 
257     return d, pred, n_ex, time, flag_nc
258 end
```

The inputs for the algorithms are as follows:

- A: A matrix of integer variables representing the graph. Each row corresponds to an edge, where the first column is the tail node index, the second is the head node index, and the third is the edge's length.
- s: An integer variable representing the source node index from which the paths originate.
- use_dequeue: A boolean variable indicating the choice of data structure. If set to true, the algorithm employs a dequeue; a queue is used if false.

In the preprocessing phase (lines 177 to 186), several supporting variables are initialized before the algorithm execution. Here, the number of vertices (or the cardinality of the vertex set), denoted as calV, and the number of edges (or the cardinality of the edge set), denoted as calE, are defined. Additionally, in line with the star forward representation described by Ahuja et al. (1993, p. 34), the pointer vector, pnt, is also defined.

During the initialization phase (lines 188 to 198), all node labels are set to infinity except for the source node, which is labeled 0. This step also involves preparing containers for predecessors, nodes pending examination, and those already examined at least once when employing a dequeue strategy. A flag for detecting negative cycles is also introduced, set to true if a negative cycle is identified and false otherwise.

In the core algorithm part (lines 200 to 255), the implementation closely follows the pseudocode and explanations related to dequeue utilization as provided by Ahuja et al. (1993, pp. 141-143). The fundamental principle involves sequentially checking the optimality condition and updating any label that violates this condition. The counts of optimality condition checks (or node examinations) are tracked in the variables n_ex. The algorithm terminates if a negative cycle is detected or if every edge meets the optimality condition.

Upon completion, the function returns the labels and predecessors for each node, the total number of node examinations and label corrections, the computation time, and whether a negative cycle is present or not.

3 Experiment design and instances generation

The experiments were conducted across diverse network structures, categorized into 19 distinct groups based on three primary characteristics: the number of nodes, the number of arcs, and the presence (and the quantity) of edges with negative lengths. These groupings are detailed in Table 1. The first nine groups encompass networks where all edges possess non-negative lengths, while groups 10 through 19 include networks with varying edges with negative lengths. Despite the diversity of the network structures, all the nodes of each instance are reachable from the source node (with index 1).

Table 1: Network Structure of Each Instances Group

Inst. Group No.	# Nodes	# Edges	# Neg. Edges
1	10	18	0
2	10	45	0
3	10	72	0
4	50	490	0
5	50	1225	0
6	50	1960	0
7	100	1980	0
8	100	4950	0
9	100	7920	0
10	100	1980	20
11	100	1980	40
12	100	1980	99
13	100	1980	158
14	100	1980	198
15	100	4950	50
16	100	4950	99
17	100	4950	248
18	100	4950	396
19	100	4950	495

Groups 1 to 9 focus on networks that only include edges with non-negative lengths. These groups are subdivided based on the number of nodes, with 10 nodes in Groups 1 to 3, 50 in Groups 4 to 6, and 100 in Groups 7 to 9. Within each node-count category, the groups are ordered by an ascending number of edges. The edge lengths for these networks range from 0 to 50.

Groups 10 to 19 shift the focus towards networks with 100 nodes each but are unique in their incorporation of edges with negative lengths. These groups are segmented into two tiers based on edge density: Groups 10 to 14 are less dense, whereas Groups 15 to 19 feature more edges. Within both tiers, the number of edges with negative lengths increases. For these groups, the range of edge lengths extends from -5 to 50.

To statistically analyze the algorithms' performance, each group comprised 1000 randomly generated instances.

4 Experiment results

The experiment is performed in my personal computer located in my laboratory, N7-2 3332. The specifications of the computer, language, and used packages are in Table 2

Table 2: Experimental Environment

Resource	Specification
CPU	13th Gen Intel(R) Core(TM) i9-13900K
RAM	128GB
OS	Windows 11
Language	Julia 1.10.0
Used packages	Random, Base, Statistics StatsBase (v0.33.21) DataStructures (v0.18.18) DelimitedFiles (v1.9.1) ProgressBars (v1.5.1)

The experiment results are in Table 3. The averages of time and the number of node examinations of each group with different data structures are provided. The values in the relative ratio are the dequeue's values relative to the queue's, given in percentage. These relative ratios provide a direct comparison: a negative sign indicates that the queue's performance metrics are higher (worse) than the dequeue's, and a positive sign indicates the opposite, where the dequeue's metrics are higher (worse) than those of the queue.

Analyzing the results reveals two observations. First, employing a queue consistently yields better computation times in every scenario involving exclusively non-negative arc lengths. However, for Groups 1, 2, and 4, which feature relatively simpler network structures, utilizing a queue surpasses the queue in terms of the number of basic operations.

Furthermore, in scenarios involving networks with negative arc lengths, the greater the number of negative arcs, the better the performance of using dequeue.

5 Conclusions

To summarize the findings:

- Using dequeue performs better on a moderate-size or spare network with non-negative arc lengths.
- Using dequeue performs better on a network containing many arcs with negative lengths.

Table 3: Experiments Results, Average of N=1000 Instances for Each Group (Rel. Ratio = (dequeue's-queue's)/queue's)

	# Node Exam.			Time (Sec)		
Group No.	Dequeue	Queue	Rel. Ratio	Dequeue	Queue	Rel. Ratio
1	18.6	18.8	-1.1	5.9e-7	4.8e-7	24.7
2	53.1	54.5	-2.6	8.2e-7	6.7e-7	23.1
3	96.0	94.7	1.4	9.1e-7	7.9e-7	14.9
4	804.0	814.1	-1.2	8.0e-6	6.3e-6	26.7
5	2576.4	2399.2	7.4	1.3e-5	1.0e-5	26.0
6	4918.1	4182.2	17.6	1.6e-5	1.3e-5	27.1
7	4401.9	4141.4	6.3	3.1e-5	2.3e-5	36.4
8	13542.8	11989.8	13.0	5.4e-5	4.0e-5	35.5
9	24551.8	21030.7	16.7	7.1e-5	5.2e-5	36.4
10	48694.1	61792.6	-21.2	2.4e-4	2.4e-4	0.1
11	71142.3	80907.6	-12.1	3.8e-4	3.6e-4	5.9
12	85284.5	111829.7	-23.7	5.8e-4	5.8e-4	0.2
13	88553.8	142770.2	-38.0	7.2e-4	8.1e-4	-10.1
14	86043.9	160159.9	-46.3	7.8e-4	9.6e-4	-19.0
15	134393.7	185703.3	-27.6	5.1e-4	5.2e-4	-1.7
16	174228.8	268568.6	-35.1	7.8e-4	8.1e-4	-4.3
17	214607.4	419130.9	-48.8	1.4e-3	1.5e-3	-10.3
18	243736.8	483383.8	-49.6	1.9e-3	2.0e-3	-6.8
19	230921.0	499116.2	-53.7	2.0e-3	2.3e-3	-10.6

These observations lend cautious support to the statements made by Ahuja et al. (1993, p.143), particularly concerning networks with negative arc lengths.

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Pearson.