HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY



**OBJECT ORIENTED PROGRAMMING**

**MINI PROJECT REPORT**

**SHORSTEST PATH ALGORITHM**

**VISUALIZATION**

**Instructor:** Dr. Bùi Thị Mai Anh

**Group Members**

Trần Hải Đăng 20194738

Nguyễn Hoàng Vũ: 20190100

Đặng Quang Minh: 20194796

**1. Introduction**

Our group wants to create a program where people can see the working principle of different algorithms. The program will allow users to create their customized graph, then select a shortest-path algorithm for the program to visualize, helping the user to get a better understanding of the algorithm.

**2. Project Scope**

The program is created to improve the learning of algorithms through animation. Our application can be used in academic environments such as algorithm classes or tech camps. In cases like this, alongside theory teaching, there will be a need for students to learn by themselves the idea of each algorithm.

**3. Roles**

Tran Hai Dang 20194738 : Assistant programmer, report writer, implement A\* algorithm.

Nguyen Hoang Vu 20190100: Primary programmer, designer, implement Bellman-Ford algorithm.

Dang Quang Minh 20194796: Primary programmer #2, styling, implement Dijkstra’s algorithm.

**4.** [**Glossaries**](https://www.wordhippo.com/what-is/the-meaning-of-the-word/glossaries.html)

**Vertex**: A [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) (plural vertices) is (together with edges) one of the two basic units out of which graphs are constructed. Vertices of graphs are often considered to be atomic objects, with no internal structure.

**Edge**: An edge is (together with vertices) one of the two basic units out of which graphs are constructed. Each edge has two vertices to which it is attached, called its endpoints. Edges may be directed or undirected; undirected edges are also called lines and directed edges are also called arcs or arrows.

**Weight**: A numerical value, assigned as a label to a vertex or edge of a graph.

**Path**: A [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) may either be a walk or a walk without repeated vertices and consequently edges (also called a simple path), depending on the source.

**Shortest Path**: A [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) between two [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) (or nodes) in a [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) such that the sum of the [weights](https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms#weighted_graph) of its constituent edges is minimized.

**S/E**: Start/End; the source and the target of shortest path of the graph.

**5. Algorithms**

* **Dijkstra’s algorithm**

Let's create an array d[] where for each vertex vv we store the current length of the shortest path from ss to vv in d[v]. Initially d[s]=0, and for all other vertices this length equals infinity. In the implementation a sufficiently large number (which is guaranteed to be greater than any possible path length) is chosen as infinity.

d[v] = ∞, v ≠ s

In addition, we maintain a Boolean array u[] which stores for each vertex v whether it's marked. Initially all vertices are unmarked:

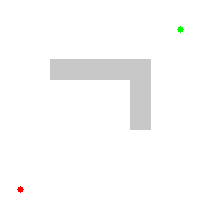
u[v] = false

The Dijkstra's algorithm runs for n iterations. At each iteration, a vertex v is chosen as unmarked vertex which has the least value d[v]. Evidently, in the first iteration the starting vertex s will be selected. The selected vertex v is marked. Next, from vertex v relaxations are performed: all edges of the form (v, to) are considered, and for each vertex to the algorithm tries to improve the value d[to]. If the length of the current edge equals lenlen, the code for relaxation is:

d[to]=min(d[to], d[v]+len)

After all such edges are considered, the current iteration ends. Finally, after n iterations, all vertices will be marked, and the algorithm terminates. We claim that the found values d[v] are the lengths of shortest paths from s to all vertices vv.

Note that if some vertices are unreachable from the starting vertex s, the values d[v] for them will remain infinite. Obviously, the last few iterations of the algorithm will choose those vertices, but no useful work will be done for them. Therefore, the algorithm can be stopped as soon as the selected vertex has infinite distance to it.



*Figure: Dijkstra’s algorithm progress animation*

In sparse graph (graph with relatively low number of edges compare to number of vertices), we can use **Fibonacci Heap** to reduce the algorithm complexity.

Implementation in C++

**const int INF = 1000000000;**

**vector<vector<pair<int, int>>> adj;**

**void dijkstra(int s, vector<int> & d, vector<int> & p) {**

**int n = adj.size();**

**d.assign(n, INF);**

**p.assign(n, -1);**

**d[s] = 0;**

**set<pair<int, int>> q;**

**q.insert({0, s});**

**while (!q.empty()) {**

**int v = q.begin()->second;**

**q.erase(q.begin());**

**for (auto edge : adj[v]) {**

**int to = edge.first;**

**int len = edge.second;**

**if (d[v] + len < d[to]) {**

**q.erase({d[to], to});**

**d[to] = d[v] + len;**

**p[to] = v;**

**q.insert({d[to], to});**

**}**

**}**

**}**

**}**

* **Bellman-Ford algorithm**

Unlike the Dijkstra algorithm, this algorithm can also be applied to graphs containing negative weight edges. However, if the graph contains a negative cycle, then, clearly, the shortest path to some vertices may not exist (due to the fact that the weight of the shortest path must be equal to minus infinity); however, this algorithm can be modified to signal the presence of a cycle of negative weight, or even deduce this cycle.

We will create an array of distances d[0…n−1], which after execution of the algorithm will contain the answer to the problem. In the beginning we fill it as follows: d[v] = 0, and all other elements d[] equal to infinity ∞.

The algorithm consists of several phases. Each phase scans through all edges of the graph, and the algorithm tries produce relaxation along each edge (a, b) having weight c. Relaxation along the edges is an attempt to improve the value d[b] using value d[a] + c. In fact, it means that we are trying to improve the answer for this vertex using edge (a, b) and current response for vertex a.

It is claimed that n – 1 phases of the algorithm are sufficient to correctly calculate the lengths of all shortest paths in the graph (again, we believe that the cycles of negative weight do not exist). For unreachable vertices the distance d[] will remain equal to infinity ∞ .

Implementation in C++

**struct edge**

**{**

**int a, b, cost;**

**};**

**int n, m, v;**

**vector<edge> e;**

**const int INF = 1000000000;**

**void solve()**

**{**

**vector<int> d (n, INF);**

**d[v] = 0;**

**for (int i=0; i<n-1; ++i)**

**for (int j=0; j<m; ++j)**

**if (d[e[j].a] < INF)**

**d[e[j].b]= min(d[e[j].b],d[e[j].a]+e[j].cost);**

**// display d, for example, on the screen**

**}**

* **A\* algorithm**

A\* is a heuristic algorithm, a combination of Dijkstra’s algorithm and Best-First Search. Instead of using only using array d[] in Dijkstra’s algorithm, A\* algorithm uses a modified cost function:

f[]: Cost function of vertices.

d[]: The current distance from source to vertices.

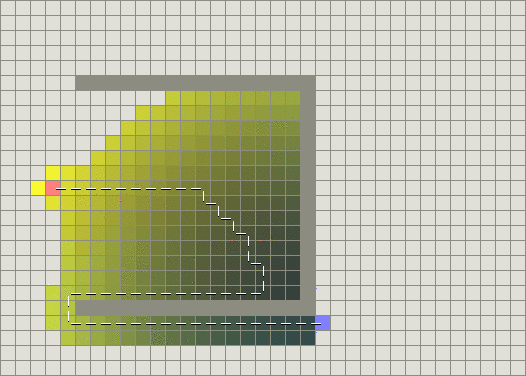
h[]: The estimate distance from vertices to target. (usually Manhattan distance or Euclidean distance).

f[x] = d[x] + h[x]

A\* algorithm while not always provide the best solution, but it significantly improves the running time in many cases. Consider the following example.Chart

Description automatically generated

*Figure: Dijkstra’s algorithm performance*

**

*Figure: Best-First Search performance*

As we can see, Dijkstra’s algorithm works harder but is guaranteed to find a shortest path. Greedy Best-First Search on the other hand does less work but its path is clearly not as good. The trouble is that Greedy Best-First Search is “greedy” and tries to move towards the goal even if it is not the right path. Since it only considers the cost to get to the goal and ignores the cost of the path so far, it keeps going even if the path it is on has become really long. In cases like this, A\* is very effective which is why it is the most popular choice for pathfinding. In the previous example, A\* finds a path as good as what Dijkstra’s Algorithm found with significantly less work.

Chart

Description automatically generated

*Figure: A\* performance*

*A picture containing logo

Description automatically generated*

*Figure: A\* algorithm progress animation*

**6. Functions**

­**Help**: Provide details required for users to correctly uses the program. The details include how to create a graph and how to run the algorithm.

**Create graph:**

* **Add vertex:** Add a vertex on the workspace.
* **Add edge** (required 2 vertices)**:** Create an undirectional edge connecting 2 existing vertices on the workspace. The program will automatically assign a random weight for the newly created edge.
* **Move vertex:** Move an existing vertex on the workspace.
* **Remove vertex:** Remove an existing vertex on the workspace.
* **Remove edge:** Remove an existing edge of the graph on the workspace.
* **Reset:** Remove everything on the workspace.
* **Choose S/E:** Choose starting and ending vertex (or source and target).

**Visualize algorithm:**

* **Dijkstra:** Demonstrate each step of Dijkstra’s algorithm on the graph.
* **Bellman Ford:** Demonstrate each step of Bellman-Ford algorithm on the graph.
* **A\*:** Demonstrate each step of A\* algorithm on the graph.

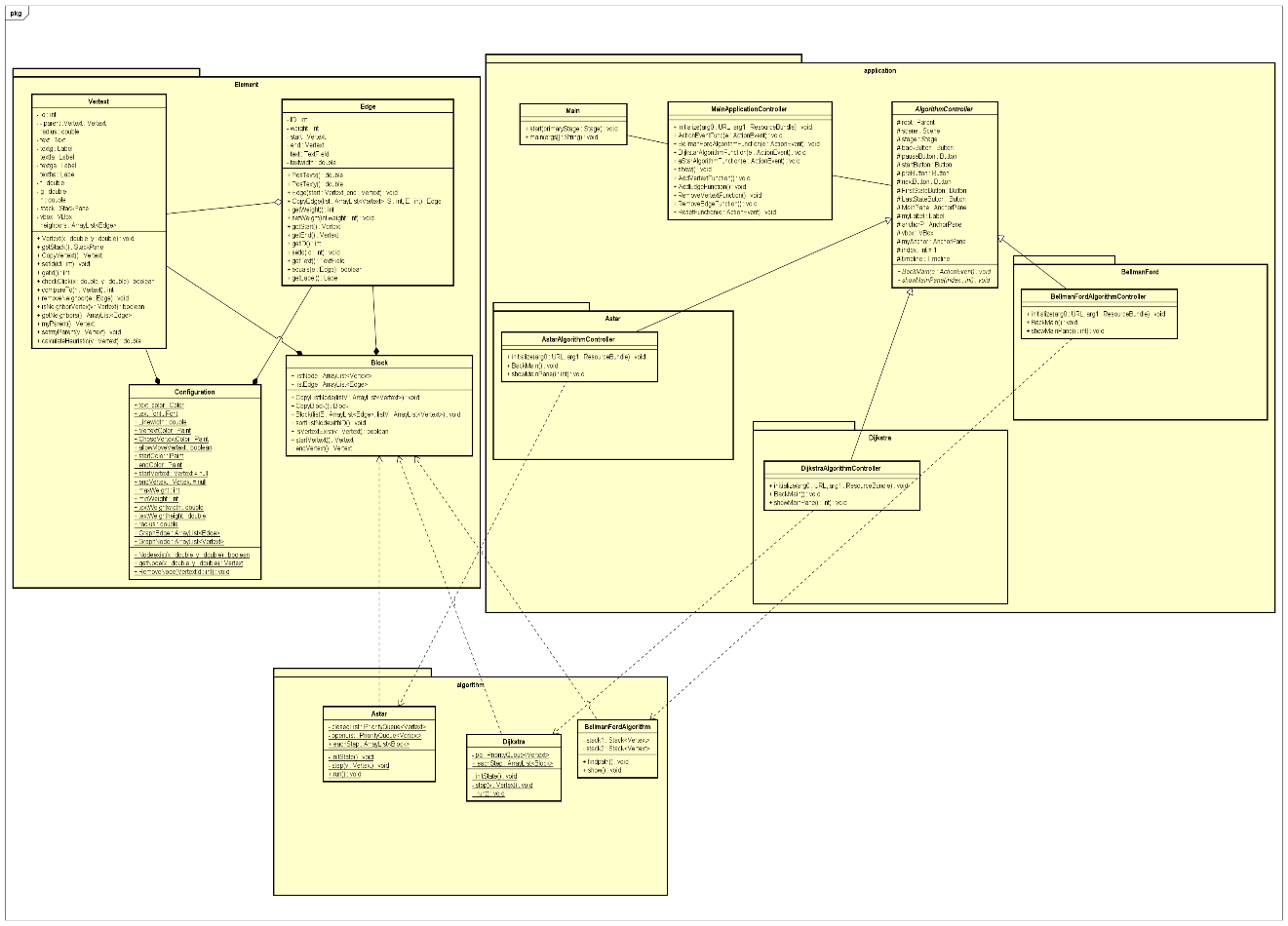
**7. How to use**

Steps to run the runnable jar file on Window 10

1. Open cmd, then change directory to the directory of the jar file.
2. Enter the following line “java --module-path [lib\_path\_javafx] --add-modules javafx.controls,javafx.fxml - jar file.jar” with [lib\_path\_javafx] as the directory of Javafx lib folder

Example: java --module-path D:\Study\edit\javafx-sdk-16\lib --add-modules javafx.controls,javafx.fxml - jar file.jar

**7. Class Diagram**

****

|  |  |
| --- | --- |
| Package “Element” | |
| Class | Description |
| Vertext | Represent vertices in the graph with basic properties like label, neighbors list … and value for each algorithm. |
| Edge | Represent edges (undirectional) in the graph with basic properties like weight, start and end vertex… |
| Block | Represent the graph with List<Vertext> and List<Edge> |
| Configuration | Methods and properties to represent graph in the program. |

|  |  |
| --- | --- |
| Package “Algorithm” | |
| Class | Description |
| Astar | The implementation of A\* algorithm. |
| Dijkstra | The implementation of Dijkstra’s algorithm. |
| BellmanFord | The implementation of Bellman-Ford algorithm. |

|  |  |
| --- | --- |
| Package “Application” | |
| Include the main menu controller, algorithm controller, how-to-use help menu, and CSS. Create the GUI of the program. | |
| Sub-packages | Description |
| Astar | Include A\* controller, help menu and methods to represent the algorithm in the program. |
| Dijkstra | Include Dijkstra controller, help menu and methods to represent the algorithm in the program. |
| BellmanFordAlgorithm | Include Dijkstra controller, help menu and methods to represent the algorithm in the program. |

**8. Conclusion**

Throughout the development of our project, we learned that it is quite hard to use JavaFX because it is quite a new language for us. However, we had overcome it successfully to complete the project and we had learned many things by experience during the development. We had learned on how to create objects, classes, GUI (Graphical User Interface), polymorphism and how to produce a program using Java language itself.