Implementation of Dijkstra's Algorithm in Pathfinding for Artificial Intelligence in Video Games

Addin Munawwar Yusuf - 13521085
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*13521085@std.stei.itb.ac.id*

***Abstract*—This paper discusses about Dijkstra's algorithm and how it is implemented for Artificial Intelligence Pathfinding in video games. Dijkstra's algorithm is based on some concept that contains graph theory, the shortest path problem, greedy algorithm, and priority queue. This paper also discuss about some limitation of Dijkstra's algorithm when implemented in real-time video game environtments. For demonstration, this paper also implements the algorithm in unity using the language C#.**

***Keywords*—Dijkstra's Algorithm, Pathfinding, Video Games, Artificial Intelligence.**

I. Introduction

Video games are digital games that are played on a computer, a console, or a mobile device. The first video game was created in 1958, and the industry has grown into a multi-billion dollar industry since then.

The development of video games has progressed significantly over the years. Especially, in recent years, the use of artificial intelligence (AI) in video games has become increasingly common. One of the main ways that AI is used in video games is to generate the behavior of non-player characters, more commonly known as NPCs, such as enemies in a game. This can involve using AI to create realistic behavior for the enemies that can engage and interact with players in a believable way.

One important aspect of creating a good enemy AI in video games is pathfinding, which can find the shortest path to the player's location or travel from one point to another. This kind of behavior is very common in video games, such as in The Escapist 2 (2017) and Granny (2017), where the game uses pathfinding on the enemy AI to create a believable response to the player's action, such as chasing the player or checking the location which player causes distraction.

Fig. 1. Police AI uses pathfinding to chase the player in the game: The Escapist 2 (2017)

(Source: https://store.steampowered.com/app/641990/The_Escapists_2/)

Dijkstra's algorithm is well-known as one of the algorithms that solve pathfinding to get the shortest path. It has been widely used in a variety of applications, including in pathfinding for game AI. In this paper, the author will explore how this algorithm is implemented for AI in video games. Firstly, the author will discuss about some base theory that underlying this algorithm, which consists of the shortest path problem, graph theory, greedy algorithm, priority queue, and then the Dijkstra's algorithm itself. Later in this paper, the author will also experiment with how this algorithm could be implemented in-game AI. For this experiment, the author will use the game engine unity, which uses C# as its language.

II. Graph Theory

A. Graph Definition

A graph is used to represent discrete objects and its relation between each object. A formal definition of graph G is G = (V, E), in which:

- V is a finite **non-empty** set of vertices (also called as nodes or points) = $\{v_1, v_2, v_3, \ldots, v_n\}$.
- E is a finite set of edges that connects 2 vertices (also called as links or lines) = $\{e_1, e_2, e_3, \ldots, e_n\}$.

B. Types of Graph

Based on the directional orientation of the graph's edges:

a) Undirected graph

An undirected graph is a graph that does not have a direction orientation on its edges.

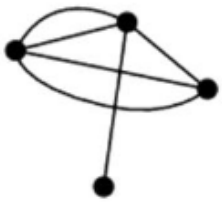

Fig. 2. Example of undirected graph

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf/)

b) Directed Graph

A directed graph is a graph that each of its edges has a directional orientaition.

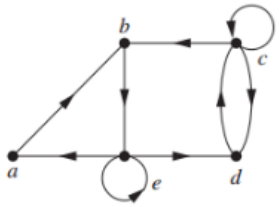

Fig. 3. Example of directed graph

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf/)

Based on the availability of parallel edges or loops:

a) Simple graph

A simple graph is an undirected graph that does not contains parallel edges or loops.

Fig. 4. Example of simple graph

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf/)

b) Not simple graph

A not simple graph is a graph that contains parallel edges and/or loops. The graph that contain parallel edges is called multi-graph.

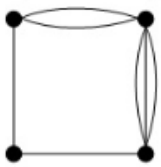

Fig. 5. Example of multi-graph

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf/)

A graph that contain loops is called pseudo-graph.

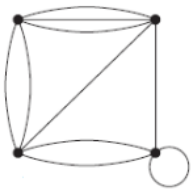

Fig. 6. Example of pseudo-graph

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf/)

Based on the availibility of edges weights:

a) Weighted graph

A graph that has weight on each edges is called weighted graph.

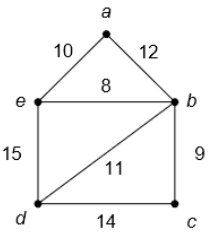

Fig. 7. Example of weighted-graph

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf/)

b) Non-weighted graph

A graph that does not has weight on each edges is called weighted graph.

C. Graph Terminology

In a graph, there are some terminology that can be used:

1. Adjacency
   Two vertices is called adjacent if there is at least one edge that directly connect these vertices.
2. Incidency
   For a random edge e = (v_1, v_2), e is called incident with vertex v_1 and e incident with vertex v_2, vice versa.
3. Degree
   Degree if a vertex is the number of edges that incident with that vertex.
4. Path
   Path that has a length of n from vertex v_0 to vertex v_n in graph G is a sequence of alternating vertices and edges such that each successuve vertex is connected by the edge. The length n also represent the number of edges passed from v_0 to v_n.
5. Circuit (Cycle)
   A circuit or a cycle ia a path that starts and ends in the same vertex.
6. Connectedness
   Vertex v_0 to vertex v_n is called to be connected if there is a path in graph G from vertex v_0 to vertex v_n. A connected graph is graph that each of pair v_i and v_j in graph G, vertex v_i and v_j is connected.

III. THE SHORTEST PATH PROBLEM

The shortest path problem is a well-known problem in computer science and mathematics. It involves finding a path between two vertices (or nodes) in a graph, such that the sum of the weights of all edges passed is minimized.

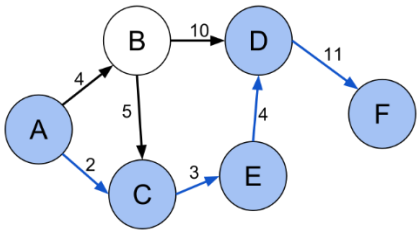

Fig. 8. Shortest path (A, C, E, D, F) between vertices A and F

in the weighted directed graph.

(Source: https://en.wikipedia.org/wiki/File:Shortest_path_with_direct_weights.svg)

The problem is often formulated as follows: suppose a graph G with vertices V and edges E, and two nodes s and t, find the the shortest path from s to t. This problem is also called as **single-pair shortest path problem** to distinguish it from other variations, such as:

- The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
- The **single-destination shortest path problem**, in which we have to find the shortest paths from all vertices in the directed graph to a single destination vertex v.
- The **all-pairs shortest path problem**, in which we have to find the shortest paths between every pair of vertices v and v' in the graph.

Some examples of the application of this shortest path problem are for finding route in road networks, logistics, communications, electronic design, etc. It also includes the pathfinding for game AI which the author will discuss it about it more in this paper.

IV. Greedy Algorithm

Greedy algorithm is a paradigm or an approach that builds up a solution piece by piece, making the locally optimal choice at each stage, with the hope of finding a global optimum. In the other words, this algorithm always makes the choice that seems to be the best at the given moment. It never reconsider the previous decision, and only considers the available options before making a decision.

That's why in many cases, a greedy strategy does not produce the optimal solution to a problem, as they may get trapped in a suboptimal solution. However, it can yield the approximation of a globally optimal solution for such a complex problem in a reasonable amount of time.

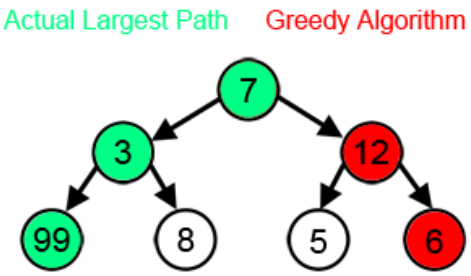

Fig. 9. Greedy algorithm failure in searching for the largest path.

(Source: https://en.wikipedia.org/wiki/File:Greedy-search-path-example.gif)

For example, suppose we want to represent 36 cents using only coins with values {1, 5, 10, 20}. The coin with the highest value that is less than the remaining change owed, is the local optimum. Then, the greedy algorithm will look like the Fig. 10 below.

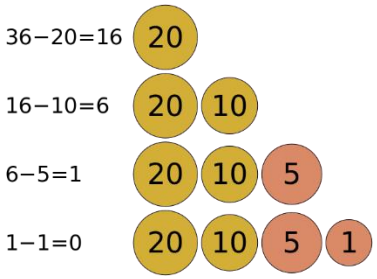

Fig. 10. Greedy algorithm in change-making problem.

(Source: https://en.wikipedia.org/wiki/File:Greedy_algorithm_36_cents.svg)

Greedy algorithm does not produce good solutions on every problems, but there are some properties in which the algorithm will work:

1) *Greedy choice property*
   An optimal solution to the problem can be found by choosing the best choice at each step, without any reconsideration to the previous choice.

2) *Optimal substructure*
   An optimal solution to the problem correspond to the optimal solution to its subproblems.

3) *Matroid*
   A matroid is a mathematical structure that generalizes the notion of linear independence from vector spaces to arbitrary sets. If a problem has the structure of a matroid, then the greedy approach can solve the problem optimally.

One example of greedy algorithm is dijkstra's algorithm, which the author will talk into more detail later in section VI.

V. Priority Queue

A priority queue is an abstract-data type that is similar to regular queue or stack, which holds a set of elements, with a difference of the addition of priority on its elements. In a priority queue, an element with high priority will be served before an element with low priority, making this a useful data structure for certain types of algorithms.

A priority queue has this properties:

- Every item has a priority associated with it.
- An element with high priority will be dequeued befor an element with lower priority.
- If two elements have the same priority, then the rules from regular queue apply (First In, First Out).

Priority queues are often implemented using binary heap, which is a special kind of binary tree that satisfies heap property: the value of each node is greater than or equal of its parent node, called as min heap, or complemently — each node is less than or equal of its parent node — is called max heap. This ensure the element with the highest priority to always stored in the root of the binary heap.

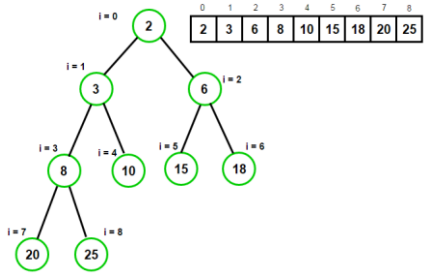

Fig. 11. Priority queue implementation using min heap.

(Source: https://www.techiedelight.com/wp-content/uploads/2016/11/Min-Heap.png)

This data structure will be beneficial when we talk about Dijkstra's algorithm.

VI. Dijkstra's Algorithm

Dijkstra's algorithm is a popular algorithm for finding the shortest path between two nodes in a graph. It was first invented by a Dutch computer scientist, Edsger Dijkstra in 1956. It was originally used to find the shortest path to travel from city to city. Now, it has been widely use in a variety of applications.

The defined problem in dijkstra's algorithm is a weighted graphs, and we had to find the shortest path to travel from vertex A to vertex B. More generally, this is what the author discuss at section III, the shortest path problem.

The basic idea of this algorithm is to sequantially explore the graph, starting from the source node, and choose the next node based on the lowest distance from the source node. This means in each step, the algorithm selects the closest node from the source node and updates distance of all adjacent nodes based on the distance to the selected node. We repeats this process until all nodes has been explored, at which point it will have found the shortest part from the source node to all other nodes in the graph. This follows the greedy algorithm principle, as discussed in section IV.

There are 2 approaches in this algorithm, using Prim's Algorithm in $O(V^2)$, or using heap (or priority queue) in $O(^{E}\log V)$. For this paper, we will only talk about the second option, which is using a priority queue (for the next part, heap/ priority queue will just be referred to as prio queue). To implement this, we first store all the unexplored nodes in the prio queue. The priority of each node is determined by the distance from the source node. This allows us to quickly retrieve the node that is closest to the source node and update the distance of all adjacent nodes accordingly.

The step-by-step for Dijkstra's algorithm would be as follows:

1. Create a priority queue that contain all nodes. For initialization, set the source node to 0 and other nodes to infinity. This is the tentative distance value which will be updated as the algorithm runs.
2. Set the initial node as current.
3. For the current node, consider all the unvisited neighbors and calculate their distance from the current node. Compare the newly calculated distance with the one that has been assigned before and assign the smaller one.
4. After done considering all neighbours, pop the current node from the priority queue, and the next node will be the top of the priority queue.
5. Repeat step 3-4 until the priority queue is empty.

Limitation

While Dijkstra's algorithm is quite simple, it also comes with limitations. One of the main challenges is its computational complexity. This can be a problem in real-time video game environments, where the algorithm needs to be executed quickly in order to keep up with the fast-paced gameplay.

Another limitation is that Dijkstra's algorithm only handles static environments. This means that Dijkstra's algorithm assumes that the graph data is static, which means it cannot be changed

VII. Dijkstra's Algorithm Experiment in Unity

For this experiment, the author will use the map from the game "The Escapist 2". The reason why the author choose The Escapist 2 as the game reference is that this game implements a lot of pathfinding for NPC, such as for the police, the medic team, the guard dogs, etc, so it would be very natural to experiment there. Fig. 12 is a layout of one of the maps, Center Perks 2.0.

Fig. 12. The map layout of Center Perks 2.0 from the game "The Escapist 2"

(Source: Author Documentation)

1) Graph Modelling

First of all, let's model this map into a graph model first, because the algorithm needs the defined problem to be in form of a graph. The idea of converting this map into a graph is every room will represent a vertex, and the door between each room will be the edge. The location point of each vertex will be located in the center of the room, and in front of each door if there is a branch. The weights of each edge can be approximated by the length of the edge, which can be obtained from unity, the game engine that the author used for this simulation. Here is what the graph model would look like in unity.

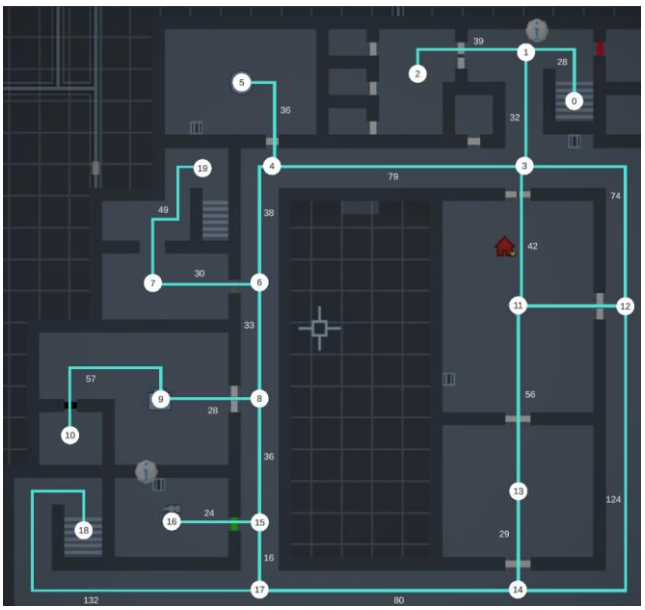

Fig. 13. The modeled graph from the map in unity

(Source: Author Documentation)

Converting this into a simpler graph, the result of the graph would look like Fig. 14.

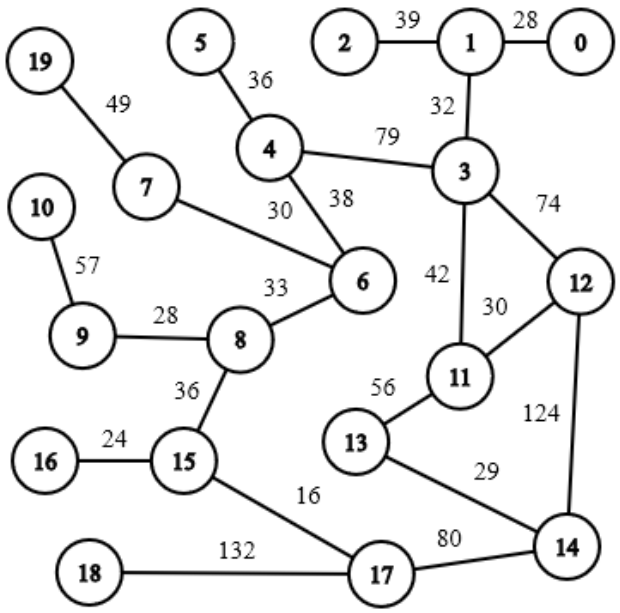

Fig. 14. The simpler graph of map layout from Fig.13.

(Source: Author Documentation)

2) Defined Problem

Now let's defined the problem. The author will take this example from the game "The Escapist 2" again. Suppose there is a player and an enemy. The enemy's objective is to catch the player, but he has no idea where the player could be. The only way for the enemy to find the player is by looking through a security camera, which is planted in some of the rooms. Suppose the player is in room 16, and there is a security camera there. The enemy itself is in room 1. What is the shortest path the enemy could take to go from room 1 to room 16?

This is exactly what Dijkstra's algorithm solves, the shortest path problem. Now, let's head to unity.

3) Implementation

For the simulation, the author uses a third-party library, Quikgraph. Quikgraph is a library for the C# programming language that provides data structures and algorithms for working with graphs. The Quikgraph library provides a number of classes and methods that can be used to create and manipulate graph data structures. For this, the author will use the `UndirectedGraph<T>` class from the library

First, let's create a Vertex and Edge class in unity. The Vertex class will have an ID, while the Edge class will have 2 vertices and edge weight. Here is the implementation.

```
using UnityEngine;

public class Vertex : MonoBehaviour
{
    public int ID;
}
```

Snapshot. 1. The Vertex Class in Unity

(Source: Author)

```
using UnityEngine;
using QuikGraph;

public class Edge : MonoBehaviour,
IEdge<Vertex>
{
    public Vertex Source { get; set; }
    public Vertex Target { get; set; }
    public float Weight { get; set; }
}
```

Snapshot. 2. The Edge Class in Unity

(Source: Author)

The next step is to make a list of vertices and a list of edges. Let's create a script called `GraphManager.cs` to handle all of the problems related to graphs. The class `GraphManager` will inherit from `MonoBehaviour`, which is a built-in class in unity that can be used for the geneobjectsject. It has a method `Awake()`, which will be called once the script is initialized. In this method, we can initialize the list of vertices and the list of edges.

```
using System.Collections.Generic;
using UnityEngine;
```

```
using QuikGraph;
using QuikGraph.Algorithms;

public class GraphManager :
MonoBehaviour
{
    public List<Vertex> vertices = new
List<Vertex>();
    public List<Edge> edges = new
List<Edge>();
    private void Awake() {
        // Initialize list of vertices
        // Initialize list of edges
    }
}
```

Snapshot. 3. Initialization of List of Vertices and List of Edges

(Source: Author)

Now we will also need to initialize the graph. For this, we will use the `UndirectedGraph<T>` from the Quikgraph library. We can initialize this in another built-in method from MonoBehaviour, Start() method. Similar to Awake(), Start() method will also called once right on the frame when the simulation started. Start() is called after Awake(), so the list of vertices and the list of edges will be initialized already. Here is how it is implemented.

```
public class GraphManager : MonoBehaviour
{
// Initialization of the list of vertices and
edges..
public UndirectedGraph<Vertex, Edge> graph =
new UndirectedGraph<Vertex, Edge>();

private void Start() {
        foreach (Vertex vertex in vertices) {
            graph.AddVertex(vertex);
        }
        foreach (Edge edge in edges) {
            graph.AddEdge(edge);
        }
    }
}
```

Snapshot. 4. Initialization of Graph

(Source: Author)

After all of the requirements have been satisfied, we can run Dijkstra's algorithm now. To do this, we will use the method that is provided by Quickgraph, `ShortestPathsDijkstra`. We will then update the path color in the game to red to show the shortest path.

```
// Will be called from a button
public void RunDjikstra(){
    ResetColor(); // Reset previous path
      color
    var tryGetPath =
      graph.ShortestPathsDijkstra(e =>
      e.Weight, vertices.Find(x =>
      x.ID.ToString() == startingVertex));
    IEnumerable<Edge> path;
    if (tryGetPath(vertices.Find(x =>
      x.ID.ToString() == endingVertex), out
      path)) {
        Debug.Log("Path found: " + path);
        foreach (Edge edge in path) {
            foreach (Transform child in
      edge.transform) {
                // Change path color to red
                child.GetComponent<SpriteRend
      erer>().color = Color.red;
            }
        }
    }
}
```

Snapshot. 5. Dijkstra's algorithm using Quickgraph library

(Source: Author)

4) *Results*

Now the algorithm is done, we can use this for our AI to find the shortest path between 2 points that we can implement in our own video game. The source code for this simulation can be obtained from:

https://github.com/moonawar/DijkstraAlgoUnity

a) Shortest path from room 1 to room 16

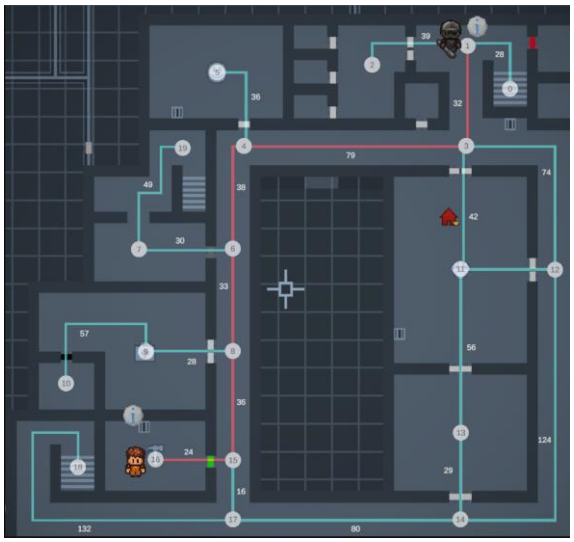

Fig. 15. The shortest path generated by Dijkstra's algorithm from room 1 to room 16

(Source: Author Documentation)

b) Shortest path from room 5 to room 14

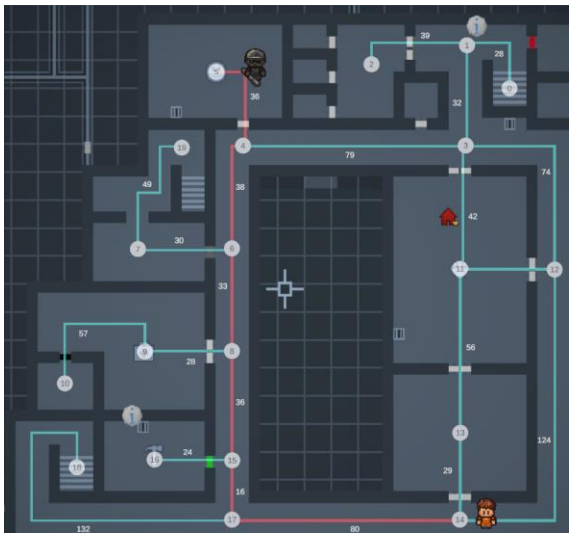

Fig. 16. The shortest path generated by Dijkstra's algorithm from room 5 to room 14

(Source: Author Documentation)

c) Shortest path from room 10 to room 0

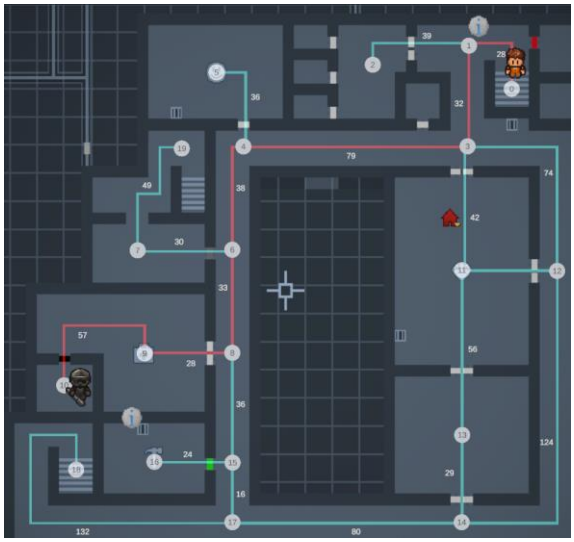

Fig. 17. The shortest path generated by Dijkstra's algorithm from room 10 to room 0

(Source: Author Documentation)

Based on the 3 experiments above, the algorithm shows a pretty accurate result. These results demonstrate the effectiveness of Dijkstra's algorithm for pathfinding AI agents in video games. For a simple game like The Escapist 2, Dijkstra's algorithm for pathfinding is more than enough. For the more complex cases, Dijkstra's algorithm may not be as effective and we might need a more complex algorithm, such as A* pathfinding or Theta* pathfinding. etc.

VIII. Conclusion

AI has been very common in recent times in game development. One of it applications is to create Non-Player Characters, known as NPCs. One of the common problems in game development is pathfinding for NPC.

There are many algorithms that can generate pathfinding for AI. Dijkstra's algorithm is one of them. There are some concepts and theories that underlie Dijkstra's algorithm. Those include graph theory, the shortest path problem, greedy algorithm, and priority queue.

Dijkstra's algorithm is quite effective to handle pathfinding, specifically in real-time video game environments. Its limitation it is that Dijkstra's algorithm can only handle static data graphs. Thus, for the more complex game that requires dynamic graphs that keeps changing rapidly, Dijkstra's algorithm is not suitable anymore, and that's where more complex algorithm such as A* or Theta* can play its role.

References

[1] Lists, Decisions and Graphs: With an Introduction to Probability. Edward A. Bender, S. Gill Williamson

[2] Graf Bagian 1 [Online]. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf/. Diakses pada 8 Desember 2022.

[3] *"A note on two problems in connexion with graphs". Numerische Mathematik. **1**. Dijkstra, E. W. (1959).*

[4] Greedy Algorithms [Online]. https://www.geeksforgeeks.org/greedy-algorithms/. Diakses pada 10 Desember 2022.

[5] What is Priority Queue | Introduction to Priority Queue [Online]. https://www.geeksforgeeks.org/priority-queue-set-1-introduction/. Diakses pada 10 Desember 2022.

[6] Dijkstra's Shortest Path Algorithm | Greedy Algo-7 [Online]. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. Diakses pada 10 Desember 2022.

[7] *"Section 24.3: Dijkstra's algorithm". Introduction to Algorithms (Second ed.)*. Cormen et al. (2001).

[8] DjikstraAlgoUnity, Addin Munawwar Yusuf (2022). https://github.com/moonawar/DijkstraAlgoUnity.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Desember 2020

Addin Munawwar Yusuf - 13521085