Use of Dijkstra's Algorithm for NextBot Navigation in Source Engine Games

Sebastian Hung Yansen - 13523070[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]sebastianhung25@gmail.com, 13523070@std.stei.itb.ac.id

***Abstract*— This paper explores the implementation of Dijkstra's algorithm within the NextBot navigation system in Source Engine games such as Garry's Mod. Dijkstra's algorithm, a classic method for solving the shortest path problem in graph theory, plays a crucial role in enabling NextBots—AI entities used in Source Engine games—to compute efficient paths in complex 3D environments. By leveraging the Source Engine's navigation mesh system, NextBots dynamically calculate paths, ensuring responsive and adaptive behaviors in gameplay scenarios. This paper provides an in-depth analysis of Dijkstra's algorithm, its application to pathfinding in video games, and a case study implementation in Garry's Mod, demonstrating its effectiveness and relevance in modern gaming AI.**

***Keywords*—Dijkstra's Algorithm, Source Engine, Shortest Path Problem, Video Game AI**

I. Introduction

Graph theory is a really old subject, a study of mathematical structures used to model relations between objects. As old as it is, it is still being used even today to solve everyday problems including route navigation, network design, and web navigation among other things. One of its everyday uses that is still being used to this day, especially with the rise of 3d graphics, are video games.

Video games started back in the 1950s when computer scientists designed a few simple games and simulations to be used in minicomputers and mainframes. Then in the 1970s, video games spread out into different formats which includes home consoles with the *Magnavox Odyssey* and arcade machines with Pong. Seeing the success of both consumer products, many companies started banking in on the video game market to recreate or even innovate many different games to this day.

Right next to the rise of video game consumerism, video game development also progressed significantly. Graphics evolved from the use of 8-bit sprites up until the use of real time ray tracing to produce hyper-realistic graphics. Gameplay evolved from just the player being able to run and jump in a 2d space to being able to capitalize on physics. Video games also make use of artificial intelligence (AI) and it's used in almost every video game. AI is mostly used for non-playable-characters (NPC) in video games. NPCs are one of the elements used in video games to immerse the player by being the player's sidekick, characters to make a place feel alive, as well as enemies to challenge and threaten the player.

There are many variations of AI systems that are used in different video games. One of which is NextBot, an AI system used in games made using the Source engine such as Team Fortress 2, Left 4 Dead, and Garry's Mod. A NextBot is different to normal NPCs where a NextBot uses an "actor" to run through specific factors when a specific event occurs. A NextBot also uses a navigation mesh, a data structure used by AI to aid in pathfinding through complicated spaces. Even though it uses a different structure, at its core it is still an AI and it uses core AI aspects.

In videogames, especially those that allow the characters to move in a 3d space, creating a good AI is important to immerse the player and there are many aspects in creating good AI, one of which is pathfinding. Pathfinding is the search for the shortest route between two points. In video games, pathfinding is commonly used by the enemy AI to respond to the player's actions. It is commonly used for the AI to roam a certain area or to chase or follow the player. Pathfinding itself is based on Dijkstra's algorithm, an algorithm for finding the shortest paths between two nodes. This paper will discuss and explore how this algorithm is implemented within NextBot AI in source games.

II. Theoretical Basis

A. Graph

A graph at its core is a diagram that represents data in an organized way. A graph shows the relationships between variables or nodes. Graph G is defined as G = (V, E), in which:

- V is a non-empty set of vertices/nodes = $\{v_1, v_2, v_3, \ldots, v_n\}$
- E is a set of edges that connect a pair of nodes = $\{e_1, e_2, e_3, \ldots, e_n\}$

B. Graph Types

Based on the presence of rings or double edges, graphs are classified into two types"

a) Simple graph
A graph that doesn't have a ring or double edges

b) Nonsimple graph
Graf that contains rings or double edges

Unsimple-graphs are further divided into:

a) Multi-graph
   A graph that contains multiple edges that connects the same nodes.

b) Pseudo-graph
   A graph that contains a ring

A visual representation of the graphs can be seen here:

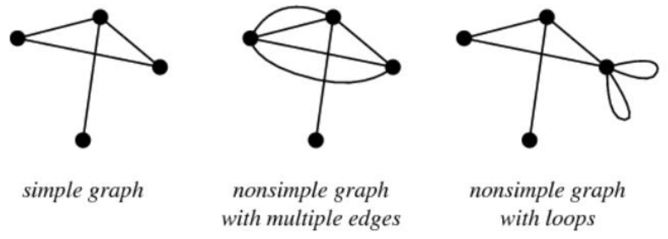

simple graph
nonsimple graph with multiple edges
nonsimple graph with loops

Fig. 2.1. Example of simple and nonsimple graphs (Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

Based on the direction, graphs can be divided into two types:

a) Undirected graph
   A graph whose edges have no direction.
b) Directed graph
   A graph whose edges have direction symbolled with an arrow that points towards a single direction.

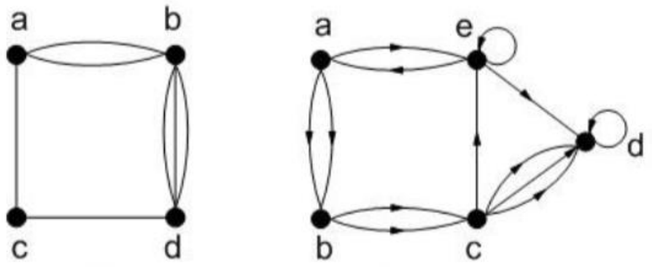


Fig. 2.2. Example of undirected and directed graph (Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

C. Graph Terminology

a) Adjacent
   Two nodes that are both directly connected.

b) Incidency
   A node is incident with an edge if the node is one of the two nodes the edge connects

c) Isolated Vertex
   A node that has no sides.

d) Null graph
   A graph whose set of edges is empty.

e) Degree
   The number of edges adjacent to the node.

f) Path
   A set of edges from one node to another that forms a continuous path.

g) Cycle or Circuit
   A set of edges from one node to another just like a path with the only difference being that the path starts from and ends at the same node

h) Connected
   A connected graph is a graph where every node is connected

i) Subgraph
   A graph that is a subset of another graph sharing some or all of its nodes and edges

j) Spanning Subgraph
   A graph that contains all of the nodes from a parent graph but may not contain all of the edges

k) Cut-Set
   A cut-set is a set of edges, when removed from a graph, renders the graph disconnected.

l) Weighted Graph
   A graph in which each edge is assigned a value (weight)

III. The Shortest Path Problem

The shortest path problem is a problem that involves finding the shortest path between two vertices or nodes within a graph. The solution for the problem is the minimized sum of the weights of all edges passed. For example, let's say there is a small city that can be represented by a map. The vertices or nodes represent the different places and branching roads whilst the road is represented by the edges with weights that represent the travel time or the distance. Many different algorithms are used to find solutions such as the Floyd-Warshall algorithm and Dijkstra's Algorithm. For this paper, the author will touch more on Dijkstra's algorithm.

Dijkstra's algorithm is one of the algorithms used to find solve the shortest path problem. Dijkstra's algorithm work by picking the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. For each step, it is necessary to record the information of the shortest distance between 2 nodes and whether or not the node has been visited.

There are two approaches in this algorithm: using Prim's Algorithm or using a priority queue. Prim's Algorithm is used to find a Minimum Spanning Tree (MST) of a graph that connects all nodes with each other, with the minimum total edge weight, and without forming circuits. Dijkstra's Algorithm with a priority queue is used to find the shortest path from a single source vertex to all other vertices in a graph, and it optimizes for distances, not tree structure.

A priority queue is typically implemented using a heap, a specialized tree-based data structure that allows efficient insertion of elements with priorities and retrieval of the element with the highest (or lowest) priority, usually in O(log n) time per operation. In Dijkstra’s Algorithm, the priority aspect is the shortest known distance from the source, making the heap an essential tool for efficiently managing the vertices to process.

IV. IMPLEMENTATION

The author will use the game Garry’s Mod for implementing Dijkstra’s algorithm in source games. The author chose Garry’s Mod as the game reference because the game is a physics sandbox game that allows the player to do anything they want with no set objective. The player can also upload and download custom user created content through mods. Through the custom user created content, the author is able to download mods required for implementation as well as view and edit the node graph within the game’s map.

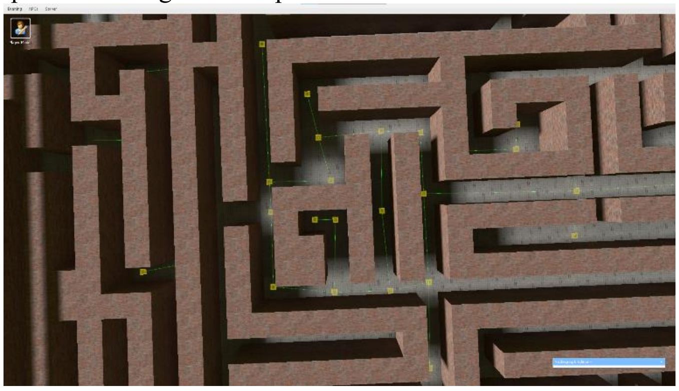


Fig. 4.1. Node graph within the map in Garry’s Mod (Source: Author Documentation)

The map’s node graph has been automatically generated from the game itself. The weights of each edge can be approximated using a ruler within the game. After converting the part of the map into a graph, the modeled graph is as follows:

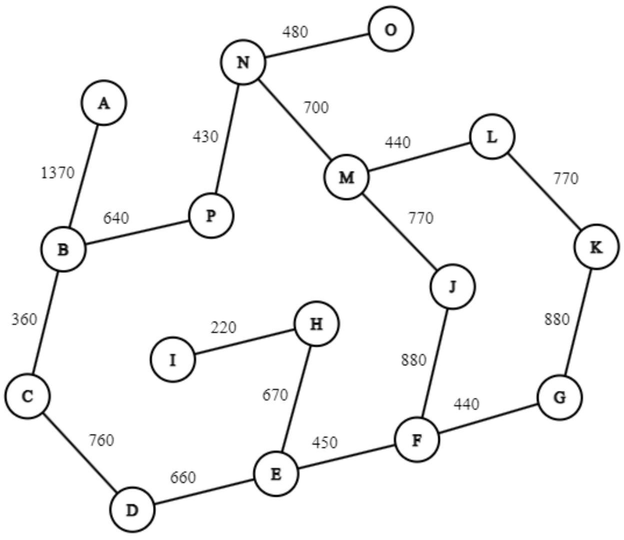


Fig. 4.2. Modeled Node graph based on the map (Source: Author Documentation)

Defining the problem is quite simple. Suppose there is an enemy and a player. The enemy’s objective is to chase the player and eliminate them. The enemy automatically targets every player and/or NPC within their distance. Suppose the location of the enemy is at node I and the location of the player is at node O. What would be the shortest path the enemy could take?

The author uses Lua for implementation. Lua is used because Garry’s Mod uses Lua as the primary scripting language. It is also used for its simplicity, speed, and flexibility. Not only that, it used in many games for modding and allows developers to write custom scripts to create custom gameplay elements, including NPCs, entities, and user interfaces.

The author uses the Garry’s Mod NextBot API. It uses the Source engine's navigation mesh system. The NextBot API abstracts the low-level pathfinding mechanics to assist the programmer. The programmer only needs to interact with high-level functions and have the Source engine internally handle the computation. The implementation of Lua is as follows:

```
function ENT:RecomputeTargetPath()
     if CurTime() - self.LastPathingInfraction <
PATH_INFRACTION_TIMEOUT then
          return
     end

     local targetPos = self.CurrentTarget:GetPos()

     trace.start = targetPos
     trace.endpos = targetPos - VECTOR_HIGH
     trace.filter = self.CurrentTarget
     local tr = util.TraceEntity(trace, self.CurrentTarget)

     if tr.Hit and util.IsInWorld(tr.HitPos) then
          targetPos = tr.HitPos
     end

     local rTime = SysTime()
     self.MovePath:Compute(self, targetPos)

end
```

The pathfinding itself is specifically called with self.MovePath:Compute(self, targetPos). The function instructs the bot to calculate a path from its current position to the target's position (targetPos) using the navmesh. Once the path has been computed, self.MovePath:Update(self) moves the NPC along the calculated path. The bot would dynamically follow the path while continuously checking for obstacles or changes in the environment. Not only that, the bot would dynamically calculate the shortest path in case the player tries to run away through branching paths.

While the implementation of Dijkstra's algorithm is not seen directly, it is possible to implement the algorithm and not use the NextBot API. Although the calculation and computation may be more complex, take more time, and can even be heavier, it allows us to see the process directly. The code is as follows:

```
local function dijkstra(graph, startNode, targetNode)
    local distances = {}
    local previous = {}
    local unvisited = {}

    -- Initialize distances and unvisited set
    for node, _ in pairs(graph) do
        distances[node] = math.huge
        unvisited[node] = true
    end
    distances[startNode] = 0

    while next(unvisited) do
        -- Find the unvisited node with the smallest distance
        local currentNode = nil
        local shortestDistance = math.huge
        for node in pairs(unvisited) do
            if distances[node] < shortestDistance then
                currentNode = node
                shortestDistance = distances[node]
            end
        end

        unvisited[currentNode] = nil

        for neighbor, cost in pairs(graph[currentNode]) do
            if unvisited[neighbor] then
                local newDistance = distances[currentNode] + cost
                if newDistance < distances[neighbor] then
                    distances[neighbor] = newDistance
                    previous[neighbor] = currentNode
                end
            end
        end
    end

    local path = {}
    local current = targetNode
    while current do
        table.insert(path, 1, current)
        current = previous[current]
    end

    -- Return the path and the total cost
    return path, distances[targetNode]
end
```

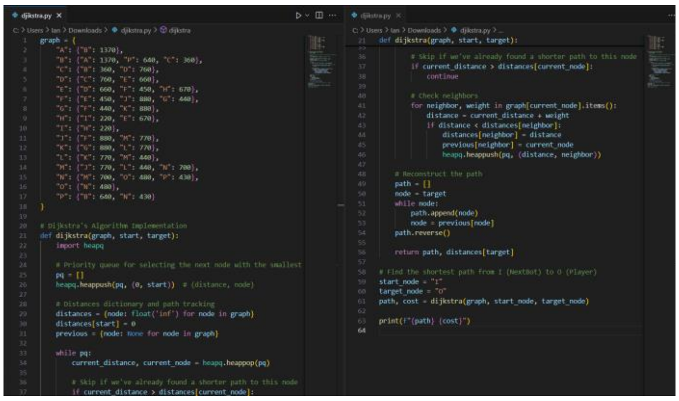
Based on the node graph and the problem that has been defined, we can define the graph as an adjacency list as follows:

```
local graph = {
    A = {B = 1370},
    B = {A = 1370, P = 640, C = 360},
    C = {B = 360, D = 760},
    D = {C = 760, E = 660},
    E = {D = 660, F = 450, H = 670},
    F = {E = 450, J = 880, G = 440},
    G = {F = 440, K = 880},
    H = {I = 220, M = 670},
    I = {H = 220},
    J = {F = 880, M = 770},
    K = {G = 880, L = 770},
    L = {K = 770, M = 440},
    M = {J = 770, L = 440, N = 700},
    N = {M = 700, O = 480, P = 430},
    O = {N = 480},
    P = {B = 640, N = 430}
}
```

With the adjacency list combined with Dijkstra's algorithm, we can find the shortest path and answer the problem.

```
local path, cost = dijkstra(graph, “I”, “O”)
```

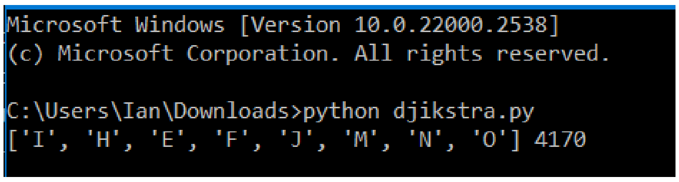
To be able to view the results based on the nodes and distance, the code can be implemented in python to see the output.



```
graph = {
    "A": {"B": 1370},
    "B": {"A": 1370, "P": 640, "C": 360},
    "C": {"B": 360, "D": 760},
    "D": {"C": 760, "E": 660},
    "E": {"D": 660, "F": 450, "H": 670},
    "F": {"E": 450, "J": 880, "G": 440},
    "G": {"F": 440, "K": 880},
    "H": {"I": 220, "E": 670},
    "I": {"H": 220},
    "J": {"F": 880, "M": 770},
    "K": {"G": 880, "L": 770},
    "L": {"K": 770, "M": 440},
    "M": {"J": 770, "L": 440, "N": 700},
    "N": {"M": 700, "O": 480, "P": 430},
    "O": {"N": 480},
    "P": {"B": 640, "N": 430}
}

# Dijkstra's Algorithm Implementation
def dijkstra(graph, start, target):
    import heapq

    # Priority queue for selecting the next node with the smallest
    pq = []
    heapq.heappush(pq, (0, start))  # (distance, node)

    # Distances dictionary and path tracking
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    previous = {node: None for node in graph}

    while pq:
        current_distance, current_node = heapq.heappop(pq)

        # Skip if we've already found a shorter path to this node
        if current_distance > distances[current_node]:
```

```
def dijkstra(graph, start, target):
        # Skip if we've already found a shorter path to this node
        if current_distance > distances[current_node]:
            continue

        # Check neighbors
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous[neighbor] = current_node
                heapq.heappush(pq, (distance, neighbor))

    # Reconstruct the path
    path = []
    node = target
    while node:
        path.append(node)
        node = previous[node]
    path.reverse()

    return path, distances[target]

# Find the shortest path from I (NextBot) to O (Player)
start_node = "I"
target_node = "O"
path, cost = dijkstra(graph, start_node, target_node)

print(f"{path} {cost}")
```

Fig. 4.3. Implementation in Python
(Source: Author Documentation)



```
Microsoft Windows [Version 10.0.22000.2538]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Ian\Downloads>python djikstra.py
['I', 'H', 'E', 'F', 'J', 'M', 'N', 'O'] 4170
```

Fig. 4.4. Implementation output
(Source: Author Documentation)

Based on the output, we can answer that the shortest path the enemy could take to reach the player is through nodes I, H, E, F, J, M, N, O with 4170 being the total sum of the weight.

V. Conclusion

Based on the analysis and the implementation that has been done, it can be concluded that Dijkstra's algorithm is an effective and reliable algorithm for finding the shortest route therefore solving the shortest path problem. Its systematic approach to exploring nodes and updating distances ensures accuracy and optimality in results, making it a foundational tool in both theoretical graph studies and practical applications.

NextBot AI dynamically calculates paths and finds the shortest path every few milliseconds to prepare in case the player runs away through many branching paths.

VI. Acknowledgment

The writer of this paper would like to thank the Lord for His guidance and mercy for giving me the ability and strength to finish this paper. The writer would also like to thank their parents, friends, and teacher for their support in knowledge and material. The writer is grateful to everyone that they've met and those who've helped the writer go through tough parts of their life.

References

[1] Munir, R., *Matematika Diskrit*, 4th ed. Bandung: Informatika Bandung, 2010, p. 356

[2] Valve Developer Community, "NextBot," Valve Developer Community, Oct. 8, 2023. [Online]. Available: https://developer.valvesoftware.com/wiki/NextBot

[3] Booth, M., "The AI Systems of Left 4 Dead," 2009. [Online]. Available: https://steamcdn-a.akamaihd.net/apps/valve/2009/ai_systems_of_l4d_mike_booth.pdf

[4] Javaid, A., "Understanding Dijkstra's algorithm," SSRN 2340905, 2013. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2340905.

[5] Schrijver, A., "On the history of the shortest path problem," Documenta Mathematica, vol. 17, no. 1, pp. 155–167, 2012.

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,8 Januari 2025

[illegible]

Sebastian Hung Yansen/13523070