

Developing NPC Behavior in Grid-Based Tactical RPGs: An Implementation of Decision Trees and Dijkstra's Algorithm

Lutfi Hakim Yusra and 13523084¹

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

13523084@itb.ac.id, luthfihakimjusra@gmail.com

Abstract—This paper presents the implementation of Discrete Math in the development of NPC behavior in a tactical RPG, mainly the logic behind the enemies' actions. By setting the expected behavior through a predetermined decision tree, alongside developing a proper movement system through Dijkstra's algorithm, we can create a generic model to be used in tactical RPGs that utilize a graph system for the game board.

Keywords— Decision Tree, Dijkstra's Algorithm, Graph-Based Movement, NPC Behaviour

I. INTRODUCTION

The game development market is currently in a state of rapid growth, with both developers and players constantly evolving their capabilities and needs over the years. This is especially true for the Tactical Role-Playing Game (RPG) genre, where the design of levels and the intelligence of both players and Non-Playable Characters (NPCs) play a crucial role in delivering a positive gaming experience for the average player. Tactical RPGs often involve intricate scenarios that challenge players to make strategic decisions while navigating grid-based environments. In such games, NPC behavior is pivotal, as it significantly influences the game's difficulty, pacing, and overall enjoyment.

A typical Tactical RPG features numerous NPCs within a single level, each requiring a robust behavioral system to act intelligently and align with the game's objectives. The design of these systems can follow a variety of rules and constraints, depending on the overarching vision of the game being developed. This paper focuses on a structured approach to developing NPC behavior by implementing a generic model that leverages both optimal pathfinding techniques and decision-making frameworks. The proposed model aims to create a flexible foundation for designing diverse NPC behaviors, supporting the creation of compelling gameplay experiences tailored to the Tactical RPG genre.

A well-designed NPC behavioral system typically comprises two essential components: an efficient pathfinding method to navigate towards desired targets

and a decision-making system that determines the actions an NPC should take in any given situation. Pathfinding ensures that NPCs can move strategically across grid-based maps, avoiding obstacles and adapting to changing game states. On the other hand, the decision-making process enables NPCs to evaluate their environment, assess potential threats or opportunities, and select actions that align with their roles within the game. By integrating these components, developers can create NPCs that exhibit consistent and believable behavior, enhancing the game's immersion and strategic depth.

The primary objective of this paper is to outline a generic model for NPC behavior that can accommodate a wide range of scenarios within Tactical RPGs. This model is designed to be adaptable, enabling developers to create simple, straightforward behaviors for standard enemy units while also supporting more complex and intelligent behaviors for boss characters or key NPCs. For example, a standard enemy NPC might prioritize moving towards the nearest player character and attacking, whereas a boss NPC could evaluate multiple factors, such as positioning, resource management, and potential escape routes, to execute more sophisticated strategies. By utilizing a single generic framework, developers can maintain consistency across NPC behaviors, streamlining the development process and improving the overall gameplay experience.

One of the key challenges in designing NPC behavior systems is achieving a balance between complexity and accessibility. While advanced AI techniques, such as machine learning, can generate highly intelligent and adaptive behaviors, they often require significant computational resources and are less transparent to developers. Instead, this paper adopts a pragmatic approach by leveraging concepts from discrete mathematics, such as graph theory and decision trees, to address pathfinding and decision-making problems. These methods provide efficient, interpretable solutions that align with the computational constraints of grid-based Tactical RPGs.

Pathfinding is addressed through the implementation of Dijkstra's algorithm, a well-established method for finding the shortest path in weighted graphs. This algorithm ensures that NPCs can navigate complex grid-based environments efficiently, accounting for obstacles, terrain types, and other constraints. Decision-making, on the other hand, is modeled using decision trees, which enable NPCs to evaluate multiple factors and select optimal actions based on predefined rules. Together, these techniques form the foundation of the proposed behavioral system, offering a balance of flexibility, efficiency, and transparency.

The development of a generic NPC behavioral system has implications beyond individual games. For developers, a consistent and modular approach simplifies the process of creating and refining NPC behaviors, reducing development time and costs. For players, a well-designed system enhances the gameplay experience by ensuring that NPCs act in ways that are both challenging and coherent with the game's narrative and mechanics. Moreover, a flexible framework allows developers to fine-tune the difficulty and complexity of NPC behaviors, ensuring that the game remains engaging for its intended audience.

This paper aims to contribute to the field of Tactical RPG development by providing a comprehensive framework for NPC behavior design. Through the integration of decision trees and Dijkstra's algorithm, this study demonstrates how discrete mathematics can be applied to create intelligent, adaptable NPCs that enhance both the strategic depth and overall enjoyment of grid-based Tactical RPGs.

II. THEORETICAL BASIS

A. Graph Theory

Graph theory is a branch of discrete mathematics that studies graphs, and the algorithms that involve them. Graphs are structures that mainly consists of nodes and edges. A node is a point or an object in the graph, representing a piece of information depending on how the graph is used. An edge is used to connect two vertices with each other. In the context of map representation, each vertex can represent a specific location or point of interest, while edges represent the connections or paths between these locations.

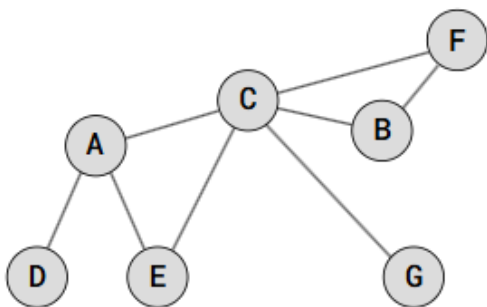


Figure 2.1 Graph

https://www.w3schools.com/dsa/dsa_theory_graphs.php

Another algorithm that will be important in the implementation of grid-based map in graph form is the pathfinding and search algorithm. This paper will use Dijkstra's algorithm, one of the most widely used methods for finding the shortest path in a weighted graph. It guarantees the shortest path from a source vertex to all other vertices in the graph make it ideal. Dijkstra's algorithm works as follows:

1. Initialize the distance to the source vertex as 0 and to all other vertices as infinity.
2. Create a priority queue and add the source vertex with a priority equal to its distance (initially 0).
3. While the priority queue is not empty:
 - o Dequeue the vertex with the smallest distance.
 - o For each of its neighbors, calculate the tentative distance through the current vertex.
 - o If the tentative distance is smaller than the previously recorded distance, update it and add the neighbor to the priority queue.

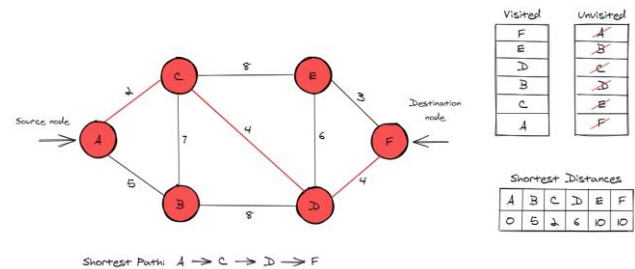


Figure 2.2 Dijkstra Algorithm

<https://algodaily.com/lessons/an-illustrated-guide-to-dijkstras-algorithm>

In this paper, grid-based maps in tactical RPGs are represented in graph form in order to apply the algorithms that are exclusive to graphs. A grid-based map can be modelled as a graph where each grid cell is a node, and edges represent movement between adjacent cells. The weight between the nodes represent the 'cost' of movement (e.g., forest tiles or obstacles could compromise the ability to normally traverse).

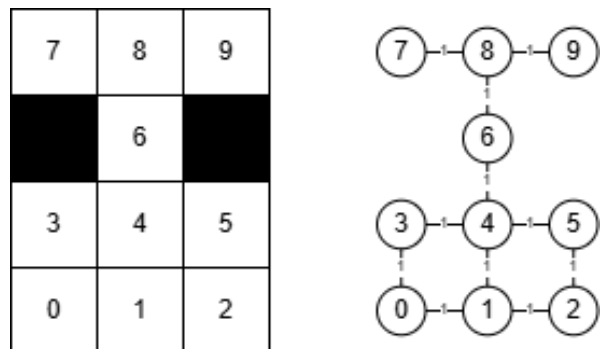


Figure 2.3 Graph Representation of Grid Map
Private Documentation

B. Decision Tree

A decision tree is a flowchart-like structure used for decision-making and classification tasks. It is composed of nodes, edges, and leaves that represent decisions, conditions, and outcomes, respectively. Decision trees work by recursively splitting data or choices based on specific criteria, resulting in a clear pathway to a decision or classification.

The main components of a decision tree include:

- **Root Node:** The topmost node representing the initial decision point.
- **Decision Nodes:** Intermediate nodes that represent conditions or splits based on specific features or attributes.
- **Leaf Nodes:** Terminal nodes that represent the final decision or classification outcome.
- **Edges:** Connections between nodes that indicate the outcomes of decisions or conditions.

Decision trees are used to act as the behaviour logic of NPCs, deciding how to interact with players based on situational context.

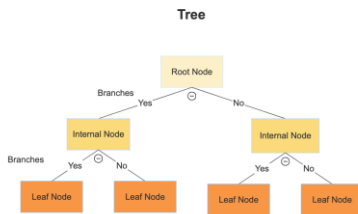


Figure 2.4 Decision Tree

<https://medium.com/@jainvidip/understanding-decision-trees-1ba0ef5f6bb4>

C. Finite State Machine

A finite-state machine (FSM), or simply a state machine, is a computational model used to design systems that transition between a finite number of states based on inputs or events. The main components of an FSM include:

1. **States:** Represent different conditions or modes of the system (e.g., idle, moving, attacking).
2. **Transitions:** Define the conditions or events that cause the system to move from one state to another.
3. **Events/Inputs:** External factors or triggers that initiate transitions (e.g., player proximity, health level).

In this paper, the state will allow NPC behavior to have different decision trees based on the context and NPC type. This allows for flexibility and modularity in designing NPC behavior, leading to a more complex and engaging behavioral system.

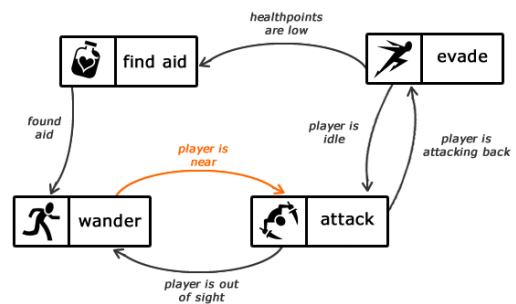


Figure 2.5 Finite State Machine

<https://code.tutsplus.com/finite-state-machines-theory-and-implementation--gamedev-11867t>

III. CODE IMPLEMENTATION

In this paper, the implementation of the code will be done in Python. Though Python is not the most optimal game developing software, the aim of this paper is the generic model implementation rather than a full-on implementation inside a functioning game. To follow with the common practices of developing NPC in game development, this paper also applies the bare minimum in state management with a basic Finite State Machine, alongside simple representation of states. To avoid losing focus, these states will only hold very simple events.

For the graph representation in code, this paper will use an adjacency list to represent an undirected weighted graph. An adjacency list represents the graph by storing all the nodes in an array, with each node storing all the connected edges in an array of itself. This approach is chosen due to its simplicity in observing all the possible paths from a single tile. In Python, instead of an array, we use a dictionary to hold all the nodes, and for every edge is appended upon both the connected nodes. The following figures show the common adjacency list diagram, alongside its code implementation.

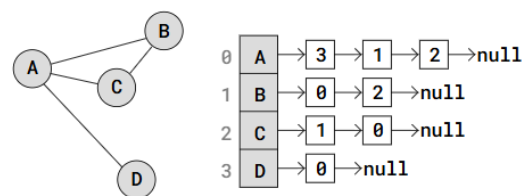


Figure 3.1 Adjacency List

https://www.w3schools.com/dsa/dsa_theory_graphs.php

```

class Graph:
    def __init__(self):
        self.nodes = {}

    def add_node(self, node):
        if node not in self.nodes:
            self.nodes[node] = []

    def add_edge(self, node1, node2, weight=1):
        self.add_node(node1)
        self.add_node(node2)
        self.nodes[node1].append((node2, weight))
        self.nodes[node2].append((node1, weight))

    def neighbors(self, node):
        return self.nodes.get(node, [])

```

Figure 3.2 Graph Implementation in Python
Private Documentation

The pathfinding method for NPC is Dijkstra's algorithm, as mentioned before. In order to utilize Dijkstra's algorithm, priority queue is also implemented, alongside the code. A priority queue is essentially a regular queue, with each element in the queue associated with a priority value, dequeuing based on priority rather than the original order. The implementation of Dijkstra's algorithm follows the steps stated in the previous section.

```

def shortest_path(self, start, goal):
    queue = PriorityQueue()
    queue.push((0, start, []))
    visited = set()

    while not queue.is_empty():
        cost, current, path = queue.pop()

        if current in visited:
            continue

        visited.add(current)
        path = path + [current]

        if current == goal:
            return path

        for neighbor, weight in self.neighbors(current):
            if neighbor not in visited:
                queue.push((cost + weight, neighbor, path))

```

Figure 3.3 Dijkstra's Algorithm in Python
Private Documentation

The following will be implementations of decision tree, states, and the finite state machine. Although more complex decision-making can be implemented in this test run, the aim of this paper is to make a generic model. With that in mind, the usage of the mentioned models are very simple in this example model.

```

class DecisionTree:
    def __init__(self, question, yes_branch=None, no_branch=None):
        self.question = question # A function that returns True or False
        self.yes_branch = yes_branch # DecisionTree or Action
        self.no_branch = no_branch # DecisionTree or Action

    def decide(self, context):
        if self.question(context):
            if isinstance(self.yes_branch, DecisionTree):
                return self.yes_branch.decide(context)
            return self.yes_branch
        else:
            if isinstance(self.no_branch, DecisionTree):
                return self.no_branch.decide(context)
            return self.no_branch

```

Figure 3.4 Decision Tree Implementation in Python
Private Documentation

```

class State:
    def __init__(self, name, on_enter=None, on_exit=None):
        self.name = name
        self.on_enter = on_enter
        self.on_exit = on_exit

    def enter(self):
        if self.on_enter:
            self.on_enter()

    def exit(self):
        if self.on_exit:
            self.on_exit()

class FiniteStateMachine:
    def __init__(self, initial_state):
        self.states = {}
        self.current_state = None
        self.add_state(initial_state)
        self.set_state(initial_state.name)

    def add_state(self, state):
        self.states[state.name] = state

    def set_state(self, state_name):
        if self.current_state:
            self.current_state.exit()
        self.current_state = self.states[state_name]
        self.current_state.enter()

```

Figure 3.5 States and Finite State Machine
Implementation in Python
Private Documentation

IV. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the developed model, this paper made a test case where an NPC 'X' is hostile towards a stationary NPC 'O' and approaches it to attack. The grid-map would be a 5x5 board with obstacles blocking the path directly, 'O' would be placed in the top left part of the graph, and 'X' is placed in the bottom right part of the graph. The *enemy_steps* variable represent the total distance 'cost' that can be afforded in each turn by the 'X' NPC.

```

grid_graph = Graph()
for x in range(5):
    for y in range(5):
        grid_graph.add_node((x, y))

for x in range(5):
    for y in range(5):
        if x + 1 < 5:
            grid_graph.add_edge((x, y), (x + 1, y), weight=2)
        if y + 1 < 5:
            grid_graph.add_edge((x, y), (x, y + 1), weight=2)

obstacle_positions = [(2, 2), (1, 1), (3, 3), (1,4)]
for obstacle_position in obstacle_positions:
    if obstacle_position in grid_graph.nodes:
        grid_graph.nodes.pop(obstacle_position)

enemy_steps = 3
enemy_position = (4, 4)
player_position = (0, 0)

```

Figure 4.1 Setting Up the Test Map
Private Documentation

VII. ACKNOWLEDGMENT

The author wishes to express heartfelt gratitude to several parties for their support in the creation of this paper. First, gratitude is extended to God for His guidance throughout the process of learning and writing. The author also acknowledges the invaluable teachings and support of Mr. Ir.Rila Mandala, the lecturer of ITB's Matematika Diskrit IF2120 course, whose guidance greatly enriched the learning experience. Finally, the author thanks their family and friends for their unwavering support throughout the semester.


REFERENCES

- [1] C. J. T. and M. A. M. "A Survey of Decision Tree Algorithms," in *Journal of Computer Science and Technology*, vol. 30, no. 4, pp. 1-15, 2015.
- [2] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269-271, 1959.
- [3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.
- [4] J. R. Quinlan, "C4.5: Programs for Machine Learning," San Mateo, CA, USA: Morgan Kaufmann, 1993.
- [5] M. A. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, 6th ed. Hoboken, NJ, USA: Wiley, 2014.
- [6] https://www.w3schools.com/dsa/dsa_theory_graphs.php

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, 1 Januari 2025



Lutfi Hakim Yusra 13523084