Implementation of Dijkstra's Algorithm on Directed Graphs for the Optimization of The *Persona Fusion System* in *Persona 3 Reload*

Renuno Yuqa Frinardi - 13524080
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: renunofrinardi@gmail.com , 13524080@std.stei.itb.ac.id

***Abstract*—This paper presents a graph-based approach to optimizing the Persona fusion system in *Persona 3 Reload*. By modeling fusions as a directed, weighted graph and applying Dijkstra's algorithm, the program can get the most cost-efficient fusion path while considering constraints such as player level and Social Link requirements. Implemented in Python with CSV-based input, this solution demonstrates a practical application of discrete mathematics specifically graph theory in game mechanics.**

***Keywords*—Dijkstra's Algorithm; Graph Theory; Directed Graph; Persona 3; Persona 3 Reload;**

I. Introduction

Video games are becoming more prevalent in this digital age. As an interactive form of entertainment, this medium quickly attracts a wide range of audiences. This can be attributed to the immersive experiences that video game has to offer, particularly those within the genre of Role-Playing Games (RPG). Video games that are within the scope of this genre usually have complex storyline with branching paths that player could choose. Allowing players a high degree of freedom and creativity.

A recent release within this genre is *Persona 3 Reload* (2024), a remake of the original *Persona 3* (2006). As the fourth installment in the main persona series, *Persona 3* (2006) played a significant role in expanding the franchise's reach within the global gaming community. Justifying its reintroduction into newer generation of consoles and computers.

Developed by *Atlus*, *Persona* is a prominent RPG series that falls under the subgenre genre of Japanese Role-Playing Games (JRPG). JRPGs is a subgenre of RPG which typically developed by East Asian developers that can be distinguished by certain aspects from other RPGs. A notable aspect from this genre is the gameplay mechanics that mostly focuses on turn-based battle.

In terms of gameplay mechanics, *Persona 3 Reload* uses a calendar system that divides each day into several time segments that players can allocate to doing many different activities, including combat. Consistent with many JRPGs, *Persona 3 Reload* utilize turn-based battle system. Player controls a set of entities known as "Personas," that will help them battle using various skills attacks. There are many different types of Personas, each with a different kind of skill sets, strengths, and weaknesses, giving the player many aspects to take into consideration before engaging in a battle. With different types of Persona, player can *fuse* their owned Persona to gain a different one with new properties. This fusion system resulting in numerous possible combination for a single persona. In which most players tend to seek the most efficient and cost-effective way fusion paths.

This fusion systems from the game can be modeled as a directed graph, where each Persona is a node with edges that have different weights depending on the cost of each fusion. Using this graph-based representation we can find the optimal way to fusion certain Persona with the help of pathfinding algorithm such as Dijkstra's Algorithm.

II. Theoretical Background

A. Graph

Graphs are a commonly used way to represent discrete objects and the relation between them. A graph G can be represented mathematically as $G = (V, E)$, where:

- V, a non-empty set of *nodes* and can be denoted as $V = \{v_1, v_2, \ldots, v_n\}$
- E, a set of edges that can be denoted as $E = \{e_1, e_2, \ldots, e_n\}$. Each edge has either one or two nodes associated with it, called an *endpoint.* The set E can also be empty in a graph in which there can exist a graph without any edges.

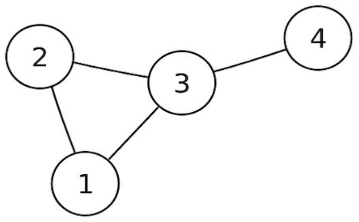


Fig. 1. Example of a graph
Source: https://study.com/academy/lesson/weighted-graphs-implementation-dijkstra-algorithm.html

Graphs can be classified as a many types depending on the structure. Depending on the edges insides of the graph, graphs can be classified into two types. A *simple graph* and a *non-simple graph*. A simple graph is a graph that doesn't have an edge that starts and ends in the same nodes (loops) and has at most one edge between two distinct nodes. A non-simple graph is the opposite of a simple graph.

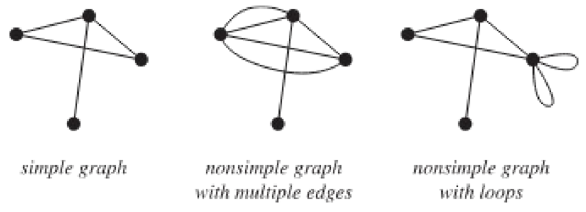

simple graph
nonsimple graph with multiple edges
nonsimple graph with loops

Fig. 2. Difference between simple and nonsimple graph
Source: https://mathworld.wolfram.com/SimpleGraph.html

Graphs can also be classified based on the direction of their edges. Depending on the direction of the edge there are two types of graphs. An *undirected graph* and a *directed graph*. An undirected graph doesn't have any direction in their edges, but a directed graph has.

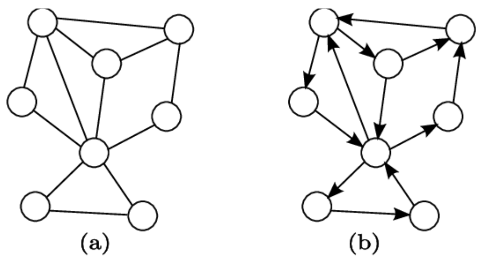


Fig. 3. Difference between directed and undirected graph
Source: https://www.researchgate.net/figure/a-An-example-of-undirected-graph-and-b-an-example-of-directed-graph_fig3_50591619

Graphs can have many different terminologies, some of it are:

1. **Adjacent**
   Two nodes in a graph are adjacent if there is an edge directly connecting them
2. **Incidence**
   A node is called incident with an edge if the edge starts or ends in the node.
3. **Isolated Vertex**
   A node is an isolated vertex, if the node doesn't have any edges connected to it.
4. **Null Graph**
   A null graph is a graph with no edges at all.
5. **Degree**
   The degree of a node is the sum of how many edges that are incident to it. The sum of all node degrees in a graph will always satisfy the *handshaking theorem*, in which the sum of the degrees of a graph equals to twice the number of edges in the graph.
6. **Path**
   An alternating sequence between nodes and edges that is ending in a node from a graph G $\{v_0, e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n\}$. Path shows the way that needs to be taken from v_0 to v_n.

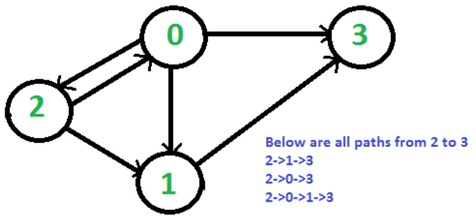


Fig. 4. List of all paths from node 2 to 3
Source: https://www.geeksforgeeks.org/dsa/print-paths-given-source-destination-using-bfs/

7. **Cycle**
   A path that starts and ends at the same node.
8. **Connectivity**
   A graph is a "connected" graph if and only if there exist a path between every pair of its nodes. If it lacks such property the graph is considered a "disconnected graph."

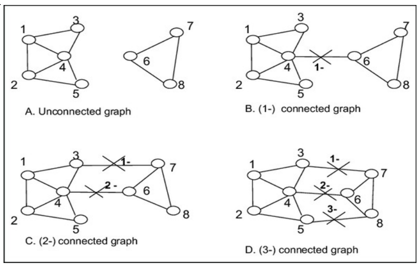


Fig. 5. Connected and disconnected graph
Source: https://www.researchgate.net/figure/Illustration-of-graph-connectivity_fig6_258188158

9. **Sub-graph and Complement of a Sub-graph**
   Sub-graph of a graph $G = (V, E)$ is a graph $G_1 = (V_1, E_1)$ if and only if V_1 is a subset of V and E_1 is a subset of E. The complement of a sub-graph G1 relative to G is $G_2 = (V_2, E_2)$ where $E_2 = E - E_1$ and V_2 is a set of nodes that are incident with the edges in E_2.

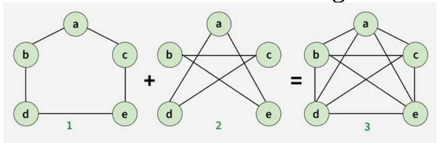


Fig. 6. Subgraph 1 that have a Complement of subgraph, graph 2, that can be combined to create graph 3
Source: https://www.geeksforgeeks.org/complement-of-graph/

10. **Weighted Graph**
    A graph where each edge has an associated numerical weight, representing cost, distance, or other metric.

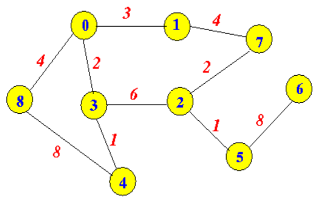
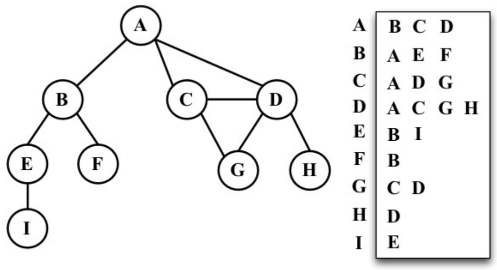


Fig. 7. A weighted graph
Source: https://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/dijkstra1.html

Graphs can be represented in many different ways. Usually there are three common ways to represent a graph:

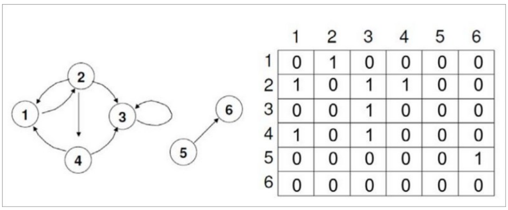
1. **Adjacency Lists**
   A list that specify the nodes that are adjacent to each node insides of the graph.



A	B C D
B	A E F
C	A D G
D	A C G H
E	B I
F	B
G	C D
H	D
I	E

Fig. 8. Adjacency list of a graph
Source: https://www.oreilly.com/library/view/learning-javascript-data/9781788623872/ef9a9b77-a6d4-480b-a4f4-77336f587b36.xhtml

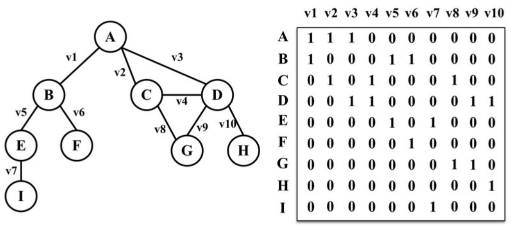
2. **Adjacency Matrices**
   A square matrix *A* with the size of *nxn* in which *n* is the amount of nodes in a graph. The element a_{ij} *in the* matrix will be filled with 0 if there exist no edges between node v_i and v_j and filled with 1 if there exist. If it is a weighted graph, then element a_{ij} in the matrix will be filled with the weight of the edges between v_i and v_j or filled with 0 if there exist no edges between them.



	1	2	3	4	5	6
1	0	1	0	0	0	0
2	1	0	1	1	0	0
3	0	0	1	0	0	0
4	1	0	1	0	0	0
5	0	0	0	0	0	1
6	0	0	0	0	0	0

Fig. 9. Adjacency matrix of a graph
Source: https://www.upgrad.com/tutorials/software-engineering/data-structure/adjacency-matrix/

3. **Incidence Matrices**
   A matrix *A* with the size of *nxm* in which n is the amount of nodes in a graph and m is the amount of edges in a graph. The element a_{ij} *in the* matrix will be filled with 0 if there exist no connection between node v_i and edges e_j and filled with 1 if there exist. If it is a weighted graph, then element a_{ij} in the matrix will be filled with the weight of the edges e_j if v_i is connected to e_j and or filled with 0 if not.



	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
A	1	1	1	0	0	0	0	0	0	0
B	1	0	0	0	1	1	0	0	0	0
C	0	1	0	1	0	0	0	1	0	0
D	0	0	1	1	0	0	0	0	1	1
E	0	0	0	0	1	0	1	0	0	0
F	0	0	0	0	0	1	0	0	0	0
G	0	0	0	0	0	0	0	1	1	0
H	0	0	0	0	0	0	0	0	0	1
I	0	0	0	0	0	0	1	0	0	0

Fig. 10. Incidence matrix of a graph
Source: https://www.oreilly.com/library/view/learning-javascript-data/9781788623872/7e3a2a29-8b6d-471e-9753-7cf3210063ad.xhtml

B. Dijkstra's Algorithm

Dijkstra's algorithm is a pathfinding algorithm that used to determine the fastest path from a starting node A to every other node in a weighted graph, provided that the nodes are reachable from the starting node A. This algorithm will continue to run until it has found the shortest path from the starting nodes to all reachable nodes in the graph, or until it has found the shortest path to a specified target node. For this algorithm to works, all the weights of the edge have to be non-negative. This algorithm works by following a simple set of steps:

1. Set the starting node distance/cost to 0 and all the unvisited node to *infinity* or undefined. Mark the starting node as explored.
2. From the current node, visit all the neighboring node and count the distance/cost from the current node. If the newest distance/cost is smaller than the current recorded distance/cost at the node, update it.
3. From the current node select a node that is unexplored and has the smallest distance/cost value, then go to the node. Mark the node as explored.
4. Repeat steps 2 and 3 until all reachable nodes are explored or the shortest path to a target node is found.

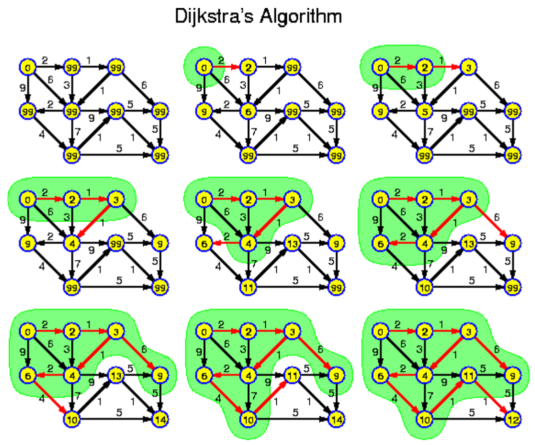


Fig. 11. Step by step of Djikstra's Algorithm (left to right)
Source: https://steemit.com/popularscience/@krishtopa/dijkstra-s-algorithm-of-finding-optimal-paths

Implementing Dijkstra's algorithm efficiently into a programming language requires a suitable data structure to be used. Usually this algorithm uses priority queue. Priority queue is being used to stores all the unexplored nodes along with their current shortest known distance/cost. It allows the algorithm to access the node with the smallest distance first. As the algorithm updates all the distances, the priority queue will maintain the correct order for processing the next node with the smallest distance/cost.

C. Persona 3 Reload

Persona 3 Reload is a Japanese Role-Playing Game (JRPG) developed by *Atlus* and a remake of the fourth mainline title in the persona series. The game is set in the fictional Japanese city of Tatsumi Port Island in the year 2009.

The story follows a young transfer student returning to the city where he was born. On the night of his arrival, the protagonist found himself in a strange dormitory during a mysterious time called the *dark hour*, a hidden period of time between the change of day, occurring right after midnight. During this time, most humans are unknowingly asleep and dangerous entity called *Shadows* roam freely. After encountering a shadow, he awakens to a unique inner power, known as *Persona*, enabling him to fight back. He soon joins an extracurricular group called *SEES* (Specialized Extracurricular Execution Squad), which its purpose is to investigate and exterminate the Shadows during the dark hour. To uncover the truth behind the Dark Hour, the protagonist and his allies must explore a mysterious, ever-changing tower that appears only during this time, *Tartarus*. Climbing its floors, they confront Shadows, grow stronger, and unravel the secrets behind the strange phenomenon threatening their world.

Fig. 12. Persona 3 Reload game covers
Source:
https://www.reddit.com/r/PERSoNA/comments/15ywdrw/the_box_art_for_persona_3_reload/

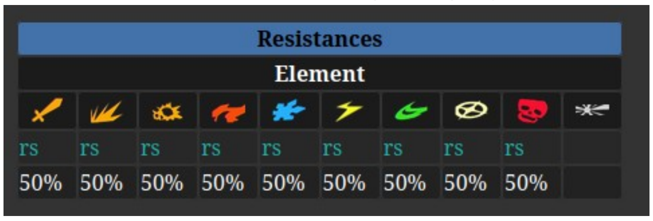
The gameplay in Persona 3 Reload is divided into two main components:

1. **Daily Life**
   This aspect takes place in the city of Tatsumi Port Island. Each in-game day is broken into distinct time segments, typically daytime/after-school and evening in which players can choose how to spend their time to doing different activities. Activities such as hanging out with friends, studying, taking part-time jobs, or preparing for upcoming battles.
2. **Dark Hour**
   Occurring during the Dark Hour, players can enter the mysterious tower or Tartarus. If the player decides to dedicate their time to visit the Dark Hour, they can explore Tartarus, battle Shadows, and advance toward the game's main objectives. Each lunar cycle culminates in a major boss battle, and progressing through Tartarus is essential to reaching the milestone for each month.

All combat in Persona 3 Reload takes during the Dark Hour, either while inside the Tartarus or during a scripted story encounters. The game uses a turn-based battle system, where each party member can perform actions such as attacking, defending, using items, or activating the special abilities of their currently equipped Persona.

There are 173 unique Personas in the base game of Persona 3 Reload. Each skill used by a Persona or Shadow can be categorized into one of ten different elemental types, as a healing, as a passive ability, or a status ailment ability. Every Persona and Shadow has a unique set of strengths and weaknesses to certain elemental attacks. If a Persona or Shadow weak to a specific element it will receive 1.5x more damage when hit by it. If they are strong against an element, the resistance falls into one of following categories:

- Drain, absorbs the attack and restores HP.
- Repel, reflects the attack back to the attacker
- Null, completely negates all the damage
- Resist, reduces the incoming damage by 50%



Resistances

Element

rs	rs	rs	rs	rs	rs	rs	rs	rs	
50%	50%	50%	50%	50%	50%	50%	50%	50%	

Fig. 13. The elemental properties of Orpheus Telos in Persona 3 Reload that resists all elemental damage except Almighty (from left to right: Slash, Strike, Pierce, Fire, Ice, Lightning, Wind, Light, Dark, Almighty)
Source:
https://aqiu384.github.io/megaten-fusion-tool/p3r/personas/Orpheus%20Telos/fissions

There are two primary types of Persona fusion systems in this game:

- *Dyad Fusion*, fusion that combines two Personas
- *Special Fusion*, fusion that involves three or more Personas.

Each Persona belongs to one of 22 *Arcana*, corresponding to the 22 Major Arcana of the tarot cards. A Persona's Arcana plays a key role in determining the outcome of a dyad fusion when combining two Personas of different types, several other factors that influence the fusion results are:

- Persona levels, the base level of the fused Personas can change the resulting Persona. The highest level of all the Personas involved in the fusion will also dictate the minimum level required for the player to be able to do the fusion
- Social links, some Personas are locked until the player reaches a certain bond level with specific characters.
- Skill inheritance, selected skills from the fused Personas can be inherited to the new one.
- Skill change, depending on the moon phase during the fusion, the player may have the option to replace a randomly inherited skill to a randomly generated one.
- Fusion accidents, a small chance in which the fusion result is randomized completely.

Aside from fusion, players can also obtain their Personas from the *Personas Compendium*, a database of all previously acquired Personas. Players can re-summon these Personas at any time by using the in-game currency (yen). Every persona has an associated summoning cost based on its level. The higher the level, the more expensive it is to summon.

III. Problem Analysis

To determine the most efficient fusion path for a Persona, the entire fusion system must first be modeled as a graph. Given that there are 173 different Personas, it is practical to begin with a simplified version of the miniature version of the fusion database. The scaled down model allows for easier development. Later, the program can be expanded to support the full fusion dataset dynamically loaded from a CSV file.

However, before constructing the graph, several special considerations must be addressed. Unlike traditional applications of graph theory, such as geographic maps, modelling a Persona fusion system involves a non-standard transformations. The things that need to be taken when converting the dataset into a graph are:

- Each node in the graph will represent one of the base Personas used in a fusion.
- The name of the edges will represent the name of the second required Personas.
- The weight of the edges will represent the cost (in yen) of summoning the second Persona from the compendium.
- The graph that must be directed, showing the direction of the fusion, from base Personas to the resulting Persona.
- Fusion accident will be excluded from the system due to their low probability and unpredictability

Knowing the considerations above, therefore, it can be determined a step-by-step on how to convert the dataset and represents it into a graph.

A. *Filtering Unnecessary Persona Attributes*

Persona has a wide range of attributes, but not all of them are relevant for the fusion pathfinding algorithm. To improve both memory efficiency and source code clarity, only a select number of key properties will be retained:

- First Persona's name, as the Persona's name will be used as the name of the graph node.
- Resulting Persona, as the resulting persona will determine the adjacency.
- Cost, as the cost will represent the amount of yen required to summon the second Persona from compendium.
- Second Persona's name, as the Persona's name will be used to name all the edges as well as the required component of the fusion.
- Required social link, ensures locked Personas are only accessible if the corresponding bond is completed.
- Highest level among three Personas involved in the fusion (two required Personas and one resulting Persona), sets the minimum level required for the player to perform the fusion.

All other attributes, such as elemental affinities, strengths, and weaknesses, will be excluded, as they are not necessary for the fusion program optimization.

B. *Choosing the Graph Representation*

Once the essential data fields have been identified, the next step is to find the suitable graph representation. Given that each fusion involves multiple pieces of metadata, such as social link requirements and level restriction, an *adjacency list* is the most appropriate choice. This representation allows for easy association with descriptive edge information while maintaining efficiency in both memory and traversal operations.

IV. Algorithm Implementation

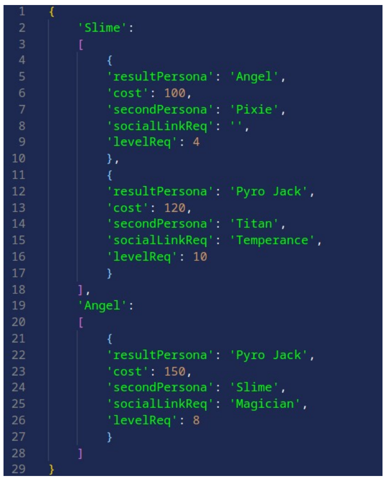
To calculate the most efficient fusion path for a Persona using the graph, it is crucial to determine the suitable pathfinding algorithm. Given that this graph different from the traditional geographical maps, the fusion graph presents unique challenges, such as multiple directed edges between the same pair of nodes, each with potentially vastly different costs depending on the targeted fusion node. These cost disparities make it difficult to apply heuristic-based algorithms (an approach used to find solutions to complex problems by providing near-optimal solutions in a faster and more efficient way compared to traditional methods) like *A-Star* effectively, as estimating an admissible and consistent heuristic becomes unreliable.

Given these complexities, a more suitable approach is Dijkstra's algorithm, which can find the shortest path in a weighted graph without relying on heuristics. Dijkstra's algorithm explores all possible path from the starting node, always expanding the node with the lowest accumulated cost. This makes it well suited for scenarios where edge weights vary unpredictably. Additionally, Dijkstra's algorithm can be easily modified to account for specific restriction such as level restriction and social link requirements.

Taking all prior factors into account, a step-by-step process can be defined for implementing the algorithm within the graph. For simplicity and flexibility, this algorithm will be implemented using *Python* programming language.

A. *Creating the Graph Representation*

The dataset's graph will be represented using an adjacency list, where each node is going to be mapped to its adjacent node. Since there are additional information needs to be stored such as incident edge name, incident edge cost, required level, as well as the required social link, the adjacency list will store a set of properties from the adjacent node. In Python this can be represented as a dictionary, where each key corresponds to a Persona, and the value is a list of dictionaries containing the details of each fusion, as such:



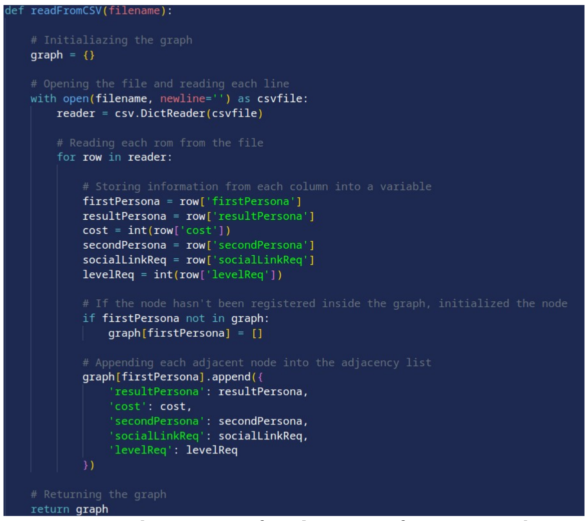
```
{
    'Slime':
    [
        {
        'resultPersona': 'Angel',
        'cost': 100,
        'secondPersona': 'Pixie',
        'socialLinkReq': '',
        'levelReq': 4
        },
        {
        'resultPersona': 'Pyro Jack',
        'cost': 120,
        'secondPersona': 'Titan',
        'socialLinkReq': 'Temperance',
        'levelReq': 10
        }
    ],
    'Angel':
    [
        {
        'resultPersona': 'Pyro Jack',
        'cost': 150,
        'secondPersona': 'Slime',
        'socialLinkReq': 'Magician',
        'levelReq': 8
        }
    ]
}
```

Fig. 14. Representation of graph in adjacency list in Python
Source: author

From the image it can be determined that the *Slime* is adjacent to *Angel* and *Pyro Jack*, and *Angel* is adjacent to *Pyro Jack*. All the weight as well as the properties of the edge and the requirements that are connecting the fusion are also listed in the dictionary.

B. *Creating the Dictionary from a CSV File*

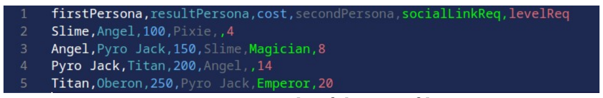
To easily modified the adjacency list, the program can be made to read the input directly from a CSV file, therefore a function to read the CSV must be created. An implementation such as a `readFromCSV` function could be created with the built-in library from Python, named *csv*. This results in a function such as:



```
def readFromCSV(filename):

    # Initialiazing the graph
    graph = {}

    # Opening the file and reading each line
    with open(filename, newline='') as csvfile:
        reader = csv.DictReader(csvfile)

        # Reading each row from the file
        for row in reader:

            # Storing information from each column into a variable
            firstPersona = row['firstPersona']
            resultPersona = row['resultPersona']
            cost = int(row['cost'])
            secondPersona = row['secondPersona']
            socialLinkReq = row['socialLinkReq']
            levelReq = int(row['levelReq'])

            # If the node hasn't been registered inside the graph, initialized the node
            if firstPersona not in graph:
                graph[firstPersona] = []

            # Appending each adjacent node into the adjacency list
            graph[firstPersona].append({
                'resultPersona': resultPersona,
                'cost': cost,
                'secondPersona': secondPersona,
                'socialLinkReq': socialLinkReq,
                'levelReq': levelReq
            })

    # Returning the graph
    return graph
```

Fig. 15. Implementation of readFromCSV function in Python
Source: author

Reading from a CSV file with the structure of:



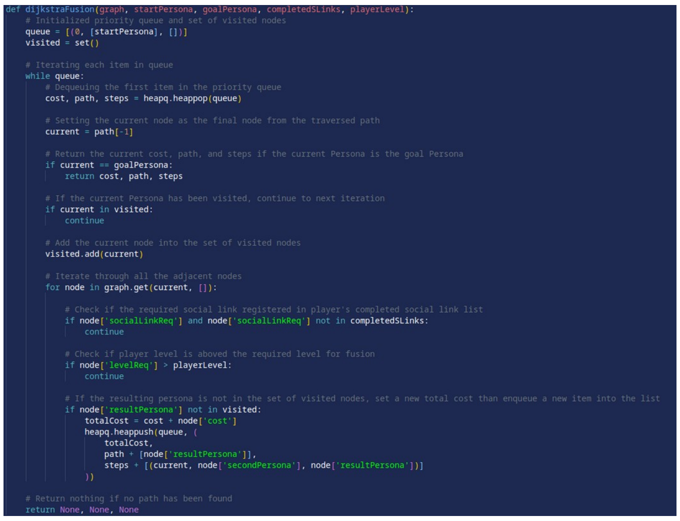
```
firstPersona,resultPersona,cost,secondPersona,socialLinkReq,levelReq
Slime,Angel,100,Pixie,,4
Angel,Pyro Jack,150,Slime,Magician,8
Pyro Jack,Titan,200,Angel,,14
Titan,Oberon,250,Pyro Jack,Emperor,20
```

Fig. 16. Example of the CSV file
Source: author

The function first initialized an empty adjacency list. It then prepares to read the specified CSV file from the input. For each row inside the CSV file the function is going to extract the data from each column and stores it into appropriate variables. The function then going to check if the node has already been initialized into the graph or not, if it isn't, it will create the entry for current node. Then, it appends the properties of adjacent node into the list associated with the current node. Once all the row has been processed, it will return the adjacency list that has been created.

C. *Finding the Shortest Path*

To traverse the graph, the program will implement a function that uses Dijkstra's Algorithm, such as `dijkstraFusion`. Before creating the algorithm, the program needs to use another built-in library from Python, named *heapq*. This library will serve as a generator for the priority queue. The function will receive five different parameters, such as the graph, starting Persona, target Persona, completed social links, and player level. This results in a function such as:

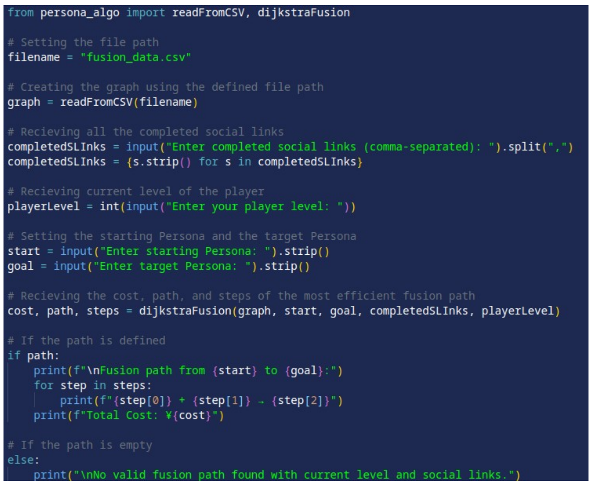


```
def dijkstraFusion(graph, startPersona, goalPersona, completedSLinks, playerLevel):
    # Initialized priority queue and set of visited nodes
    queue = [(0, [startPersona], [])]
    visited = set()

    # Iterating each item in queue
    while queue:
        # Dequeuing the first item in the priority queue
        cost, path, steps = heapq.heappop(queue)

        # Setting the current node as the final node from the traversed path
        current = path[-1]

        # Return the current cost, path, and steps if the current Persona is the goal Persona
        if current == goalPersona:
            return cost, path, steps

        # If the current Persona has been visited, continue to next iteration
        if current in visited:
            continue

        # Add the current node into the set of visited nodes
        visited.add(current)

        # Iterate through all the adjacent nodes
        for node in graph.get(current, []):

            # Check if the required social link registered in player's completed social link list
            if node['socialLinkReq'] and node['socialLinkReq'] not in completedSLinks:
                continue

            # Check if player level is aboved the required level for fusion
            if node['levelReq'] > playerLevel:
                continue

            # If the resulting persona is not in the set of visited nodes, set a new total cost than enqueue a new item into the list
            if node['resultPersona'] not in visited:
                totalCost = cost + node['cost']
                heapq.heappush(queue, (
                    totalCost,
                    path + [node['resultPersona']],
                    steps + [(current, node['secondPersona'], node['resultPersona'])]
                ))

    # Return nothing if no path has been found
    return None, None, None
```

Fig. 17. Implementation of the Dijkstra's Algorithm
Source: author

D. *Showing The Results*

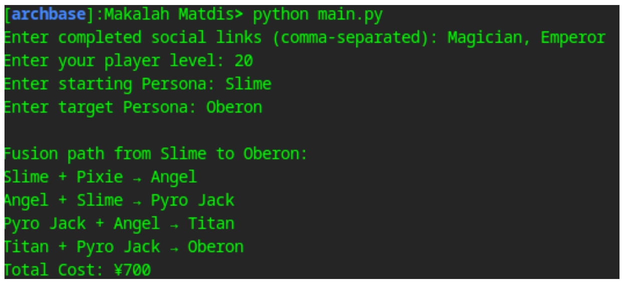
After the shortest path has been found, the results will be shown. If the path has been found it will print the fusion path from the starting Persona to the target Persona, if it hasn't been found, it will print a message that the path hasn't been found with the current level and social links. This section of the program will result in:



```
from persona_algo import readFromCSV, dijkstraFusion

# Setting the file path
filename = "fusion_data.csv"

# Creating the graph using the defined file path
graph = readFromCSV(filename)

# Recieving all the completed social links
completedSLInks = input("Enter completed social links (comma-separated): ").split(",")
completedSLInks = {s.strip() for s in completedSLInks}

# Recieving current level of the player
playerLevel = int(input("Enter your player level: "))

# Setting the starting Persona and the target Persona
start = input("Enter starting Persona: ").strip()
goal = input("Enter target Persona: ").strip()

# Recieving the cost, path, and steps of the most efficient fusion path
cost, path, steps = dijkstraFusion(graph, start, goal, completedSLInks, playerLevel)

# If the path is defined
if path:
    print(f"\nFusion path from {start} to {goal}:")
    for step in steps:
        print(f"{step[0]} + {step[1]} → {step[2]}")
    print(f"Total Cost: ¥{cost}")

# If the path is empty
else:
    print("\nNo valid fusion path found with current level and social links.")
```

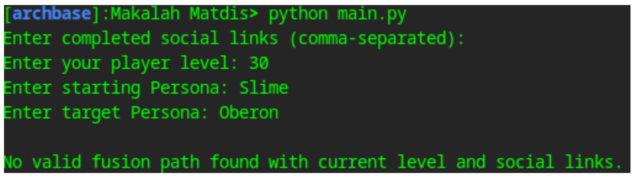
Fig. 18. Main section of the program
Source: author

This program will output two different result depending on the return value of the path. Using the CSV dataset from section B, here are the result:



```
[archbase]:Makalah Matdis> python main.py
Enter completed social links (comma-separated): Magician, Emperor
Enter your player level: 20
Enter starting Persona: Slime
Enter target Persona: Oberon

Fusion path from Slime to Oberon:
Slime + Pixie → Angel
Angel + Slime → Pyro Jack
Pyro Jack + Angel → Titan
Titan + Pyro Jack → Oberon
Total Cost: ¥700
```

Fig. 19. Output from the program for the most efficient fusion path from Slime to Oberon (found)
Source: author



```
[archbase]:Makalah Matdis> python main.py
Enter completed social links (comma-separated):
Enter your player level: 30
Enter starting Persona: Slime
Enter target Persona: Oberon

No valid fusion path found with current level and social links.
```

Fig. 20. Output from the program for the most efficient fusion path from Slime to Oberon (not found)
Source: author

V. CONCLUSION

In conclusion, this short analysis has presented a structured approached to modeling an in-game *fusion system* from *Persona 3 Reload* using graph theory and a pathfinding algorithm, specifically Dijkstra's Algorithm, in Python. By representing the fusion dataset as a graph and using pathfinding algorithm, the program can determine the most cost-efficient fusion path while accounting for certain restriction from the game, such as level and required social links. This implementation shows the versatility of graph theory on tackling problem that we can found on our daily activities.

VIDEO LINK AT YOUTUBE & GITHUB REPOSITORY

Youtube: https://youtu.be/jObilZD3duI

Github: https://github.com/renuno-frinardi/program-makalah-matdis.git

ACKNOWLEDGMENT

First and foremost, I sincerely thank Allah, whose guidance has enabled me to complete this paper. I would also like to express my heartfelt appreciation to my Discrete Mathematics lecturer, Mr. Arrival Dwi Sentosa, M.T, as well as to my family and friends for their unwavering support, valuable insights, and continuous encouragement throughout the development of this work. Lastly, I apologize for any mistakes or shortcomings that may be present in this paper.

REFERENCES

[1] K. H. Rosen, Discrete Mathematics and Its Application, 7th Ed. New York:McGraw-Hill, 2012, pp. 641–678.

[2] R. Munir, "Graf (Bag.1)." Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. Accessed: June 16th 2025.

[3] M. Sigid, "Decision Tree Application to Find the Optimal Way of Spending Daily Life Activities in Persona 4." Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Makalah/Makalah-Matdis-2020 (149).pdf. Accessed: June 16th 2025.

[4] G. Simon, "Tree-Based Phylogenetic Analysis: A Tool for Understanding Pathogen Dynamics in Indonesia." Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/Makalah/Makalah-IF1220-Matdis-2024%20(162).pdf. Accessed: June 16th 2025.

[5] C. Norris, "Persona 3 Reload – Review." Available: https://turnbasedlovers.com/review/persona-3-reload-review. Accessed June 16th 2025.

[6] A. Warih, "Penggunaan Algoritma Dijkstra untuk Mengoptimalisasi Rute Pengumpulan *Starconch* dalam Permainan Genshin Impact." Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/Makalah2023/Makalah-Matdis-2023%20(155).pdf. Accessed: June 17th 2025.

[7] N. N. Srinidhi, "Network optimizations in the Internet of Things: A review." Available: https://www.sciencedirect.com/science/article/pii/S2215098618303379. Accessed: June 17th 2025.

[8] V. Joshi, "Finding The Shortest Path, With A Little Help From Dijkstra." Available: https://medium.com/basecs/finding-the-shortest-path-with-a-little-help-from-dijkstra-613149fbdc8e. Accessed June 17th 2025.

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, 18 Juni 2025

[illegible]

Renuno Yuqa Frinardi
13524080