Creation of CPU for Hearthstone Chess Minigame Using Breadth First Search

Jaya Mangalo Soegeng Rahardjo - 13520015
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 1320015@std.stei.itb.ac.id

***Abstract*—Hearthstone is a PvP and PvE card game where player summon minions and use spells to try and defeat the opponent. Hearthstone has a chess mini game where both players have the same streamlined cards and therefore need careful moves and luck to defeat the opponent. Using a Breadth First Search Algorithm, we can create a CPU that will find the optimal moves and combination of cards to play each turn.**

Keywords—breadth first search; hearthstone chess; cpu

I. INTRODUCTION

Hearthstone is a digital collectible card game developed by Blizzard Entertainment. The player engages in a card battle against other players in a few game modes or a CPU in some specific PvE game modes or mini games.

Hearthstone gameplay consists of alternating turns between two Players (Heroes). Each turn, the players draw cards from their deck onto their hand and gain an increasing amount of mana, of which the players can use to play said cards from their hand.

The cards mainly consists of *Minions* and *Spells*, alongside a small amount of *Weapon* and *Hero* cards. Minions are units which can be summoned onto the board, after which they can attack enemy minions or the enemy hero itself. Spells are cards which have various effects: draw cards, deal damage, summon minions, buff or de-buff minions, and so much more. Hero and weapon cards are "equippable" cards which can be used to strengthen the player itself.

Another mechanic is the Hero Power, each hero has a special ability which typically cost mana and can be used once each turn. Each "Basic" hero power tend to be weak while some special "Boss" hero power can be very powerful.

One of said mini games is "Chess". Originally released as a pure PvE game mode during the *One Night in Karazhan* Adventure in 2016, the player plays with a chess-themed deck against a CPU opponent with the same deck.

Fig. 1.1 A Hearthstone Chess mini game board state

(Source: https://hearthstone.fandom.com/wiki/Chess)

The gimmick of the game mode is that the player and the CPU only has Minion-type cards, and almost of all the minions will attack the enemy directly in front of it. Due to this, the player placements of cards are quite important.

Another important mechanic is the hero powers each hero has, the player has 2 different hero powers based on the difficulty. On normal difficulty, the player has the ability to choose and create a card. While on heroic difficulty, the player's hero power is quite weak and is only able to move around the position of the minions on the board. This is in stark contrast towards the Boss's hero power. The boss's hero power will destroy your left most minion every turn from the start of turn 6.

In normal difficulty, your power to create a card is stronger than the boss's power to destroy your minions, therefore normal difficulty is quite easy and manageable. However, in heroic difficulty, your power is much weaker than the boss' power. As of such, Heroic difficulty is notorious for being unfair and extremely difficult. You need a combination of both luck and skill to squeeze the narrowest of victory.

II. THEORETICAL FRAMEWORK

A. *Graph Theory*

Graphs are a type of visualizations often used to represent discrete objects and their relations together. Graphs are

structures made of vertices and edges, where a vertices are nodes/objects and edges are the relations that connect these vertices together.

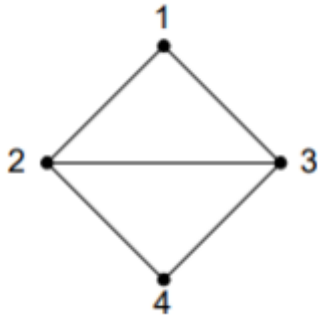

Fig. 2.1 A representation of a graph with 4 vertices and 5 edges

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

A formal definition of a graph would be defined as G = (V, E), with V representing the sets of all vertices in G and E representing the sets of all the edges in G.

B. Breadth First Search

Breadth First Search is an algorithm used to search through a graph using a traversal method. The algorithm alongside Depth First Search (DFS) is used to search when no additional information is known about the graph prior to the search.

BFS searches through a graph by visiting the nodes on the highest level first before moving to the lower levels. Each node of the graph will only be visited once during the search.

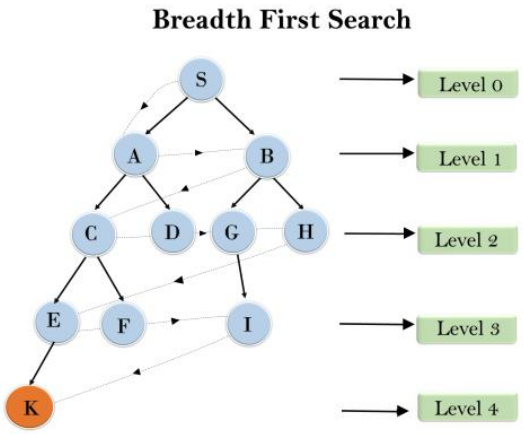

Fig. 2.2 Illustration of the order of searched nodes during a BFS algorithm.

(Source: https://static.javatpoint.com/tutorial/ai/images/breadth-first-search.png)

To implement this, BFS typically uses a queue data structure, where every time a node is expanded or searched, the nodes connected to said node will be added onto the queue. To make sure a node is only visited once, a boolean table is often used to represent that status if a node has been expanded or not.

Here are the steps the BFS algorithm takes during a search for a **static** graph:

1. Create an empty queue
2. Add the root node into the queue
3. Repeat:
   3.1 Expand the first node in the queue.
   3.2 Check the connected nodes if it has been expanded before.
   3.3 If not, add the nodes into the queue.
4. Repeat until end goal is reached or queue is empty

```
procedure BFS(input v:integer)
{ Traversal graf dengan algoritma pencarian BFS.
  Masukan: v adalah simpul awal kunjungan
  Keluaran: semua simpul yang dikunjungi dicetak ke layar
}
Deklarasi
   w : integer
   q : antrian;

   procedure BuatAntrian(input/output q : antrian)
   { membuat antrian kosong, kepala(q) diisi 0 }

   procedure MasukAntrian(input/output q:antrian, input v:integer)
   { memasukkan v ke dalam antrian q pada posisi belakang }

   procedure HapusAntrian(input/output q:antrian,output v:integer)
   { menghapus v dari kepala antrian q }

   function AntrianKosong(input q:antrian) → boolean
   { true jika antrian q kosong, false jika sebaliknya }

Algoritma:
   BuatAntrian(q)          { buat antrian kosong }

   write(v)                { cetak simpul awal yang dikunjungi }
   dikunjungi[v]←true      { simpul v telah dikunjungi, tandai dengan
                             true}
   MasukAntrian(q,v)       { masukkan simpul awal kunjungan ke dalam
                             antrian}

  { kunjungi semua simpul graf selama antrian belum kosong }
  while not AntrianKosong(q) do
        HapusAntrian(q,v)   { simpul v telah dikunjungi, hapus dari
                              antrian }
        for tiap simpul w yang bertetangga dengan simpul v do
            if not dikunjungi[w] then
               write(w)     {cetak simpul yang dikunjungi}
               MasukAntrian(q,w)
               dikunjungi[w]←true
            endif
     endfor
  endwhile
  { AntrianKosong(q) }
```

Fig. 2.3 A typical BFS Algorithm written in Indonesian using the Algorithmic Notation.

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf)

BFS can also be used for dynamic graphs. Unlike static graphs where the graph has already been created and all the algorithm does is traverse through it, dynamic graphs are graphs which the program create and search through during run-time.

The dynamic algorithm works the same as the static version, the only difference is that during **expansion** process, the program will **create** and then enqueue it into the queue.

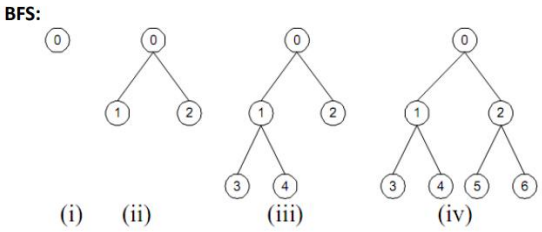

Fig. 2.4 An example of a dynamic BFS algorithm steps.

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf)

C. Hearthstone Chess

A. Player

A Hearthstone Chess game starts with the player and the CPU draws 3 and 4 cards from their deck respectively. The game alternates between the player's and the CPU's turns with the player going first. Each turn, each player will draw 1 card from their deck and they will gain mana each turn to play the cards in their hand. Each player has 20 HP and the game will end when one player's HP reaches 0 or lower.

B. Mana

Mana is a resource used to play cards, each card will have a mana cost with weaker cards costing less mana. Each player starts with 1 mana and at the start of every turn, the player will gain an increasing amount of mana until a maximum of 10 mana per turn.

C. Cards

In this game mode, only minion cards exist so other type of cards don't have to be considered. Cards have mana cost which are shown by the number in the blue crystal at the top left of the card. All cards except the knight have 6 maximum HP, depicted by the number in the red blood symbol at the bottom right of the card. Each minion also has an attack value which depicts the amount of damage it will deal to the enemy, the value is shown by the number in the yellow sword symbol at the bottom left of the card.

Fig. 3.1 White Pawn minion with 1 mana cost, 1 damage, and 6 HP.

(Source: https://hearthstone.fandom.com/wiki/Chess)

Three of those cards are standard attack cards. These cards will attack enemies at end of the player's turn. The Pawn, The Rook, and The Queen will attack enemies for 1, 2, 4 amounts of damage respectively.

Fig. 3.2 White Pawn, White Rook, and White Queen

(Source: https://hearthstone.fandom.com/wiki/Chess)

Another type of card is The Bishop. Instead of dealing damage to enemies, the bishop will heal adjacent friendly minions for an amount of 2 HP at the end of each turn.

Fig. 3.3 White Bishop

(Source: https://hearthstone.fandom.com/wiki/Chess)

The last type of card is The Knight, the knight is a special and extremely powerful card as it is able to attack any card directly akin to normal hearthstone minions. However, the complexity of choosing and attacking an enemy is quite tricky and therefore will be ignored for this case study.

D. Hero Power

Hero powers are abilities that each player have. On heroic difficulty, the player has the "Castle" Hero Power, the player can spend 1 mana to move a position of a minion in his board to the left. This hero power can be used multiple times each turn.

The CPU has the "Cheat" Hero Power, starting from turn 6 and onwards, the CPU will always spend 2 mana to destroy the left-most minion the player owns.

Fig. 3.4 Castle and Cheat Hero Power

(Source: https://hearthstone.fandom.com/wiki/Chess)

E. Board Mechanics

A player can summon up to a maximum of 7 minions unto the board. When attempting to summon a minion, a player can place them anywhere in the board and then the card alongside the rest of the board will shift to the center.

Figure 3.5 Example Board State A

Figure 3.6 Player attempting to summon a bishop.

Figure 3.7 After the summon, the board shifts with the center/median being the bishop and a pawn.

F. Battle Mechanics

When minions are placed on the board, minions will attack the enemy minions directly in front of them, reducing the enemies' HP or killing it entirely. If there is no enemy minion, they will attack the enemy hero instead.

Fig. 3.8 An example board state with the same parity

(Source: https://hearthstone.fandom.com/wiki/Chess)

In this example, the queen and the rook will attack the enemy hero because there are no enemy minions in front of them. The bishop will attempt to heal the queen and the pawn adjacent to it. While the pawn will attack and kill the enemy black pawn.

If the player's and the enemy's board has different parities (e.g. the player is even and the enemy is odd), the player's minion will attack both of the minions in front of it.

Fig. 3.9 An example board state with different parities

(Source: https://hearthstone.fandom.com/wiki/Chess)

In this example, both players' boards have different parities.

The pawn on the left and the rook on the right will attack the enemy hero directly the same way. The Bishop will still heal adjacent allies the same way.

However, the second pawn from the left and the queen on the right will only attack the enemy in front of them. While the third pawn in the middle will attack both enemies in front of it.

Board states with different parities can deal up to twice the amount of damage as boards with the same parities. Due to this, different parities boards are often much more desirable.

III. Implementation of Breadth First Search to Create a CPU

A. BFS Structure and Concept

In order to create a CPU using breadth first search, we need to define how the BFS algorithm will find the "solution" of the problem. In this case, we create a BFS algorithm to dynamically create and search nodes. The algorithm then will try to find the node with the highest weight (priority). Said node will contain the steps which will result in the best move for the CPU.

After identifying the design of the BFS program we will use, we must then define how to calculate the weight of each node. A way to do this is to "simulate" the battle of each board state and then calculate the result of each simulation. For example, dealing 1 damage to a pawn will result in a score of 1 while dealing 1 damage to a queen will result in a score of 4. These scores are then summed and becomes the weight of each node.

There are other scenarios that are relevant as well, such is bonus points for a killing blow towards an enemy minion, damage to the enemy hero, or a bonus for summoning a minion.

```
PAWN_SUMMON_WEIGHT = 1
BISHOP_SUMMON_WEIGHT = 2
ROOK_SUMMON_WEIGHT = 2
QUEEN_SUMMON_WEIGHT = 5

PAWN_DAMAGE_WEIGHT = 1
BISHOP_DAMAGE_WEIGHT = 2
ROOK_DAMAGE_WEIGHT = 2
QUEEN_DAMAGE_WEIGHT = 4

PAWN_HEAL_WEIGHT = 1
BISHOP_HEAL_WEIGHT = 2
ROOK_HEAL_WEIGHT = 2
QUEEN_HEALTH_WEIGHT = 4

PAWN_KILL_WEIGHT = 2
BISHOP_KILL_WEIGHT = 4
ROOK_KILL_WEIGHT = 4
QUEEN_KILL_WEIGHT = 8

PLAYER_DAMAGE_WEIGHT = 1
```

Fig. 3.1 Configuration of relevant scenarios and their respective points.

These numbers can also be tweaked around to give the CPU a characteristic. Such as an aggressive CPU would have high point configuration for dealing damage to the player. A defensive CPU would have high points for healing its own minions.

B. BFS Specification: Node Calculations

When expanding a node for a certain card (e.g. Pawn), the program will find the best placement for that Pawn in the board. Depending on the board state, the pawn could be placed into 1 to 7 different positions where each position might have a different weight than the others. Our program then will find the position for that pawn with the biggest weight and it will be recorded into the node alongside with the weight after the pawn placement.

Say for example, say the current board state was this:

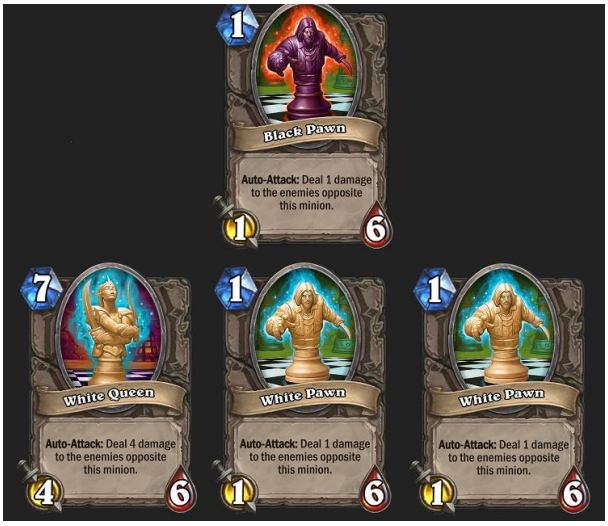

Fig. 3.2 Example board state B.

Currently (Black King/CPU's turn), the black pawn will attack the white pawn in front of it. Now, our program is tasked to find the best placement for a Black Rook. There are two options: the program can place the rook on either the left or right side of the pawn.

Fig. 3.3 Example board state B after a black rook is placed on the right side of the black pawn.

If placed on the right side of the pawn, the board will shift and the black pawn is in front of a white queen and a white pawn, while the black rook is in front of 2 white pawns.

In our simulation, the black pawn will deal 1 damage to the white queen (4 points) and the middle white pawn (1 points), while the black pawn will deal 2 damage to the middle and right white pawn (2 and 2 damage respectively). Our placement of the black rook on the left resulted in a total weight/points of 9 points.

Fig. 3.4 Example board state B after a black rook is placed on the left side of the black pawn.

If placed on the left side of the pawn, the board will shift and the black rook is in front of a white queen and a white pawn, while the black pawn is in front of 2 white pawns.

In our simulation, the Black Rook will deal 2 damage to the white queen (8 points) and the middle white pawn (2 points), while the black pawn will deal 1 damage to the middle and right white pawn (1 and 1 damage respectively). Our placement of the black rook on the left resulted in a total weight/points of 12 points.

After calculating all possible positions, the program will find the position with the highest weight. After which the position of the Rook, the weight of the board state will be noted down.

C. BFS Specification: Node Creation and Enqueueing

After calculating and saving the weight of a node, we need to create and enqueue any connected nodes from said node. For each node, the program will try to enqueue 4 nodes that will each attempt to play a Pawn, Bishop, Rook, and Queen card.

For Example, a node with the steps [Pawn, Pawn] (meaning that to reach that node, the CPU must play 2 pawn minions), when the node is expanded, it will try to enqueue 4 nodes with the steps [Pawn, Pawn, Pawn], [Pawn, Pawn, Bishop], [Pawn, Pawn, Rook], [Pawn, Pawn, Queen].

To make sure the program doesn't infinitely loop and to slightly increase the efficiency of the BFS, some heuristic approaches can be implemented.

The first is to make sure that the program will not create or process any invalid nodes. For example the program will not queue a *Queen* Node if the black king does not have enough mana to play it or if the black king does not have the *Queen* card in his hand.

The second approach is to give rules of which nodes can be created from each node. For example, a Pawn node can create *Pawn*, *Bishop*, *Rook*, and *Queen* Nodes. But a *Bishop* node cannot create Pawn nodes and instead can only create *Bishop*, *Rook*, and *Queen* Nodes.

This approach will prevent any [Pawn, Bishop] and [Bishop, Pawn] duplications from happening.

The same goes for *Rook* Nodes being able to create *Rook* and *Queen* Nodes. And while a *Queen* Node can theoretically create another queen node, it will be rejected due to the cost of 2 queens (14 mana) being higher than the maximum possible mana (10 mana).

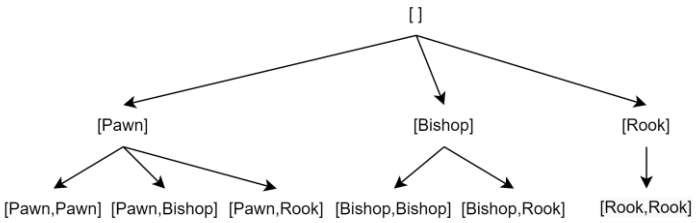

Fig. 3.5 Illustration of Possible Node Expansion during BFS not including the queen

D. BFS Search Results

Using the Node Calculations and Node Expansion, the BFS will loop until the queue is empty. During the BFS search finished, the highest weight node will be recorded. After which the solution of said node will be played by the CPU and the CPU will then attack and end their turn.

IV. Visualizer and Testing

Using the Python Django framework, a bare-boned visualizer of the mini game can be constructed. The player can summon minions and use the castle hero power using a form or move to the next phase with a button.

A few random test cases can be taken from our program:

1. Basic Move

Fig. 4.1 Visualizer of Basic Test Case before CPU Move

In this scenario, the CPU chooses to play only one pawn instead of two to achieve the double attacks onto the player units.

Fig. 4.2 Visualizer of Basic Move Test Case after CPU Move

2. Multiple Moves

Fig. 4.3 Visualizer of Multiple Move Test Case before CPU Move

In this scenario, the CPU chooses to play two cards: a rook and a pawn, still to achieve double attacks

Fig. 4.4 Visualizer of Multiple Move Test Case after CPU Move

3. Positioning Move

Fig. 4.5 Visualizer of Positioning Move Test Case before CPU Move

In this scenario, the CPU chooses to play a rook positioned in front of the white bishop, the following attack would lead to the death of the bishop.

Fig. 4.6 Visualizer of Positioning Move Test Case after CPU Move

Overall, the CPU works very well. In fact, in tests done spanning an entire match, the CPU is almost impossible to defeat. This is partly due to the brutally difficult nature of the game and a very important card in the game, the Knight, is removed to make the programming process easier.

V. Conclusion

In conclusion, Breadth First Search is a powerful tool that is able to search through multiple combinations of a certain problem. Though Breadth First Search might not be the most efficient algorithm, it proves to still be useful and certainly easier to design then it's more powerful counterparts.

BFS is more widely used not as a CPU, but as a search tool which can be used in mapping, File system searching, graphing, and many more.

Despite this, the BFS, even though it is not the most obvious choice to make a CPU, is able to function as a one by looking at all possible moves and combination and then choosing the best possible move.

Completion wise, the current program still has room for improvements. For example, the *Knight* card is extremely powerful and mandatory to defeat the CPU in a fair match. It is certainly difficult to implement as its behavior is very different from the other cards.

Video Link at Youtube

https://bit.ly/VideoMakalahStimaJaya

ViSualizer and game Repository

https://github.com/JayaMangalo/HearthstoneChessCPU

Acknowledgment

The Author would like to appreciate and thank all the lecturers and lab assistants for the course IF221 Algorithm Strategy who has given the necessary knowledge in computer science in general and in Breadth First Search algorithm used in this research.

REFERENCES

[1] Munir, Rinald informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf, accessed on May 22, 2021

[2] Munir, Rinaldi, informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf, accessed on May 22, 2021

[3] Munir, Rinaldi, informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf, accessed on May 22, 2021

[4] https://hearthstone.fandom.com/wiki/Chess accessed on May 22, 2021

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, 21 Mei 2022

Jaya Mangalo Soegeng Rahardjo 13520015