Evaluation of Search Algorithms for the Gaps Game

Artificial Intelligence (3IN1007)

https://quaffel.github.io/gaps/.

Aria David Darmanger[1], Owen Gombas[1,2], and Niklas Radomski[2]

[1] *University of Bern*
[2] *University of Fribourg*
[3] *University of Neuchâtel*
Swiss Joint Master of Science in Computer Science

30.05.2024

Table of contents

1. Abstract

As part of the Artificial Intelligence (3IN1007) Bachelor class taught by Prof. Dr. Christos Dimitrakakis at the University of Neuchâtel, we were tasked with implementing a program using artificial intelligence concepts. In our project, we focused on developing three primary algorithms based on three exploration to find a solution path for a solitaire game called Gaps. The algorithms we implemented are Greedy BFS, A*, which was covered in class, and Monte Carlo Tree Search (MCTS), which we explored independently.

2. Introduction

Gaps, also known as Montana Gaps or Patience Gaps[1], is a solitaire game that is notably time-consuming to solve in its default setting. It belongs to the family of patience games[2], which are generally single-player games where the objective is to arrange cards in a specific order according to certain rules. The game is played with a deck of French-suited playing cards[3].

To date, there is no significant work dedicated to solving this game, and we hope that our research can contribute to this area.

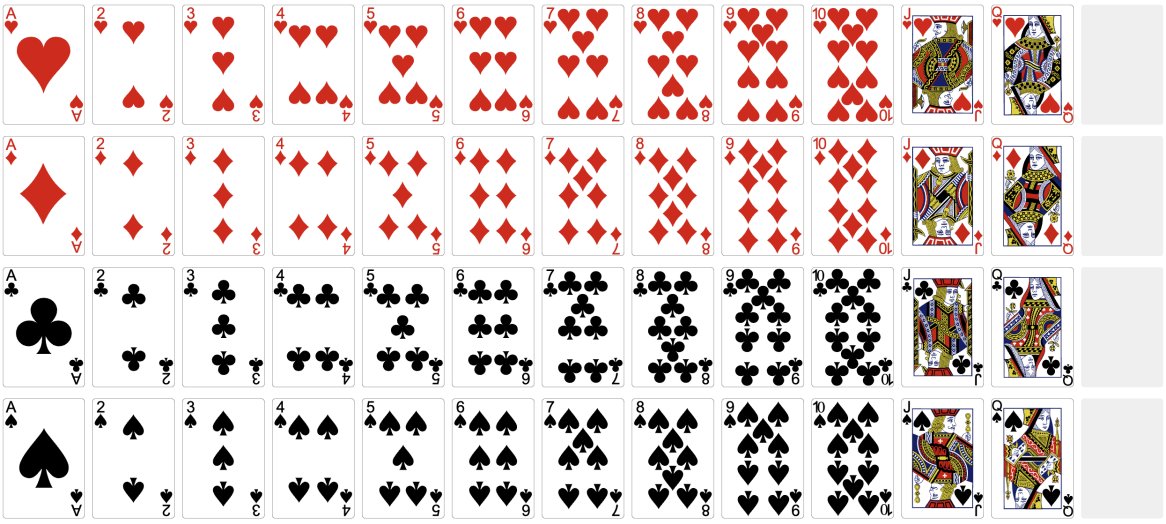

Figure 2.1: The playing cards used to play the Gaps game (in its unshuffled state)

2.1 Vocabulary

The game contains simple rules, and understanding these rules requires a clear definition of the vocabulary we will use.

1. **Gap**: A gap is an empty space in the game where no card is present. It can accept a specific card depending on the preceding card and its position in the row.
2. **Rank**: The rank of a card refers to its number, which ranges from Ace (mapped to ID 0) to King (mapped to ID 12).
3. **Suit**: The suit of a card represents its symbol: Hearts (Red heart symbol, mapped to 0), Diamonds (Red diamond symbol, mapped to 1), Clubs (Black club symbol, mapped to 2), and Spades (Black spade symbol, mapped to 3).
4. **Color**: The color of a card is either black or red.
5. **Card Coordinate**: We define the coordinates in our game as follows: (r, c), where r represents the row and c represents the column, starting from the top left.
6. **Previous Card**: The previous card is the card immediately to the left of the current card. Therefore, the previous card of the card located at (r, c) is located at $(r, c-1)$.
7. **Trap Node / Trap State**: A state in the game where no actions are available, and the game is not solved.

[1] https://en.wikipedia.org/wiki/Gaps
[2] https://en.wikipedia.org/wiki/Patience_(game)
[3] https://en.wikipedia.org/wiki/French-suited_playing_cards

8. **Solve Node / Solved State**: A state in the game where the game is considered solved.
9. **Leaf Node / Leaf State**: A state that can either be a solved state or a trap state.
10. **State Space**: All the possible states that can be reached from an initial game state by performing a series of actions.

2.2 Mathematical symbols

We will refer to the following mathematical representation in this report, here is a summary of what they represent.

G^t	Represent the board at time t, it is a matrix a cards
$G^t_{c,r}$	Represent the card at position (c, r) in the board at time t
C	The number of columns we are working with
R	The number of rows we are working with, it also represent the number of gaps in our game
$X\left(G^t\right)$ and $X_n\left(G^t\right)$	The number of well placed cards, the n denotes its normalized version between 0 and 1
$Y\left(G^t\right)$ and $Y_n\left(G^t\right)$	The number of dead gaps in the game, the n denotes its normalized version between 0 and 1
$Z\left(G^t\right)$ and $Z_n\left(G^t\right)$	The number of repeated gaps in the game, the n denotes its normalized version between 0 and 1
$S\left(W_X, W_Y, W_Z, G^t\right)$	The score of the state G^t with the weights W_X, W_Y, and W_Z
$H\left(G^t, W_p\right)$	The heuristic of the state G^t with a weight penalty W_p
N	The complexity of the shuffle.
W_c	The weight of the exploration term in the UCB formula.
$t(C, N)$	The timeout in seconds for a game of complexity N and C columns.

Table 2.1

2.3 Game rules

From the previous vocabulary we can define our rules

1. A game has a number of rows and columns within the discrete intervals
   $R \in \{1, 2, 3, 4\}$
   $C \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$
2. From all the cards we pick R (number of rows) cards that has to be removed to create the gaps. These ones are often the Kings or the Ace, it doesn't change the difficulty. In our implementation we choosed to remove the Kings.
3. We randomly shuffle the game, moving the gaps into random coordinates
4. We can only move a card in a gap
5. A card can be moved in a gap at location if and only if the previous card is a card with the same suit and with a rank exactly one unit lower than the card being moved. For instance we can move 7♦ in front of 6♦ if there is a gap, but not in front of 6♦ or 5♦.
6. Any card can be moved if the a gap is at the beginning of a row, meaning if the gap position is in the following set of positions: $\{(r, 0) | \forall r \in R\}$.
7. If two or more gaps follow each other the gaps positioned after the first one are considered as a "double-gaps" and they cannot cannot be filled until the preceding one got filled.
8. The game is solved if for every row is populated by one unique suit and the cards are sorted, placing the gap at the last position in the row.

2.4 Dead gap corollary

According to our rules, gaps located after the highest-ranked cards in the game are unusable for placing any cards. This is because a card can only be placed in a gap if its rank is exactly one unit higher than the rank of the card preceding the gap. However, this rule does not apply to gaps situated in the last column, as they represent the correct position of the gap in the solved state. We refer to these unusable gaps as "dead gaps."

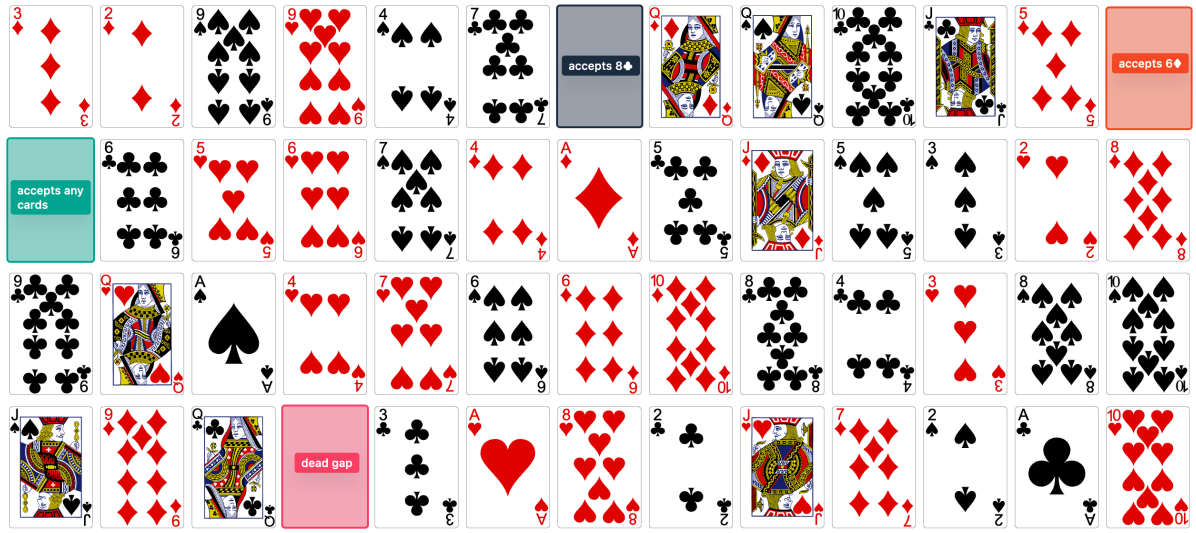

Figure 2.2: The movements rules of the games summarized in one state

2.5 Shuffling

There are multiple ways to shuffle the game to start playing. A naive approach involves selecting two random cards and swapping them, repeating this process N times to shuffle the game. However, this method does not guarantee the solvability of the game, which is typically ensured when a human player shuffles due to its time-consuming nature.

A more reliable method is to start from a solved game, then pick a random card and a random gap, and move the card from its initial position to the gap. This process is repeated N times. Shuffling in this manner from a solved game creates a tree of possible actions with an increasing depth as N increases. The higher the value of N, the more trap nodes are introduced into our state space.

Algorithm 1 Shuffle board

for t **do** from 0 to N
  $(r_c, c_c) \leftarrow$ pick_random_card_position(G^t)
  $(r_g, c_g) \leftarrow$ pick_random_gap(G^t)
  $G^t \leftarrow \text{swap}\left(G^t_{(r_c,c_c)}, G^t_{(r_g,c_g)}\right)$
end for

3. Methodology

3.1 Game scoring

3.1.1 Well-placed cards function

Defining the number of correctly placed cards for a certain game state might seem trivial, but it is actually quite complex. This challenge proved to be one of the most complicated tasks we faced.

Figure 3.1 illustrates the problem. Should the diamonds continue to populate the first row, moving the ones from the second row upward? Or should the diamonds dominate the second row instead, moving the remaining diamonds from the first row to the second? This would mean that even if the spades in the second row are well positioned, we will have to move them.

To address this, we needed to find a way to count the number of well-placed cards. First, we must ensure that this number equals the number of cells ($RC - R$) in the game when it is solved. To compute this number, we determine which suit dominates each column based on the number of correctly positioned cards in each row for every suit. The process is as follows:

1. Distribute points to each suit within the row. A suit receives one point whenever:
   (a) One of its cards is placed in the correct column (e.g., 6♥ is in the 6th column).
   (b) It contains a chain of cards (e.g., 6♥ is directly followed by 7♥).
2. Identify the suit that dominates the current row based on their number of points.
3. Count all the cards that are in the correct column and belong to the dominant suit as well placed.

This scoring method encourages heuristic-based approaches to gather a single suit within a row and sort them in the correct order without initially enforcing a specific suit to fill a particular row. Any row can be filled with any suit.

Retrieving the length of the list of well-placed cards for a certain state $X\left(G^t\right)$ gives us the number of well-placed cards. We normalize the number of well-placed cards $X\left(G^t\right)$ from 0 to 1 by dividing it by the number of cards the game has, as follow:

$$X_n\left(G^t\right) = \frac{X\left(G^t\right)}{RC - R} \tag{3.1}$$

The term "$-R$" accounts for the number of gaps, there is R gaps in our game and the well-placed cards function do not take them into account.

The pseudo-code of this function is given in the algorithm 2.

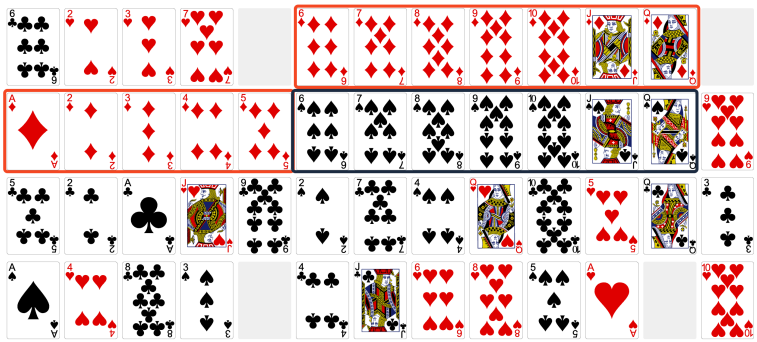

Figure 3.1

Algorithm 2 Find Correctly Placed Cards

```
well_placed_cards ← []
for r from 0 to R do
    best_suits ← array of zeros with length R
    for c from 0 to C do
        card ← G^t_{r,c}
        if is_gap(card) then
            continue
        end if
        if column_correct(card, c) then
            best_suits[i] ← best_suits[i] +1
        end if
        if c > 0 then
            previous_card ← G^t_{r,c−1}
            if is_gap(previous_card) then
                continue
            end if
            if column_correct(previous_card, c − 1) and suit(card) = suit(previous_card) then
                best_suits[i] ← best_suits[i] +1
            end if
        end if
    end for
    best_suit ← argmax(best_suits)
    row_well_placed_cards ← []
    for c from 0 to C do
        card ← G^t_{r,c}
        if is_gap(card) then
            continue
        end if
        if column_correct(card, c) and suit(card) = max_idx then
            append(row_well_placed_cards, (r, c))
        end if
    end for
    append(well_placed_cards, row_well_placed_cards)
end for
return well_placed_cards
```

3.1.2 Dead gaps

We establish a function that returns the list of positions within the game where dead gaps are located. A dead gap can be identified by checking if the preceding card for each gap is the card with the highest possible rank in the game.

Retrieving the size of this list gives us the number of dead gaps for a certain state $Y\left(G^t\right)$. We can normalize this value from 0 to 1 by dividing it by R since its maximum value is the number of gaps R:

$$Y_n\left(G^t\right) = \frac{Y\left(G^t\right)}{R} \tag{3.2}$$

Algorithm 3 Get dead gaps

```
max_rank ← C − 1
dead_gaps_positions ← []
for r from 0 to R do
    for c from 1 to C do
        current_card ← G^t_{r,c}
        preceding_card ← G^t_{r,c−1}
        if is_gap(current_card) and rank(preceding_card) = max_rank then
            append(dead_gaps_positions, (r, c))
        end if
    end for
end forreturn dead_gaps_positions
```

3.1.3 Repeated gaps

The method is similar to the previous one but instead retrieves the list of positions of gaps that have another gap preceding them.

As with the dead gaps function, retrieving the size of this list gives us the number of repeated gaps for a certain state $Z\left(G^t\right)$. We can normalize this value from 0 to 1 by dividing it by $R-1$ since the maximum number of repeated gaps is $R-1$ (the initial gap is not considered as part of a chain of repeated gaps):

$$Z_n\left(G^t\right) = \frac{Z\left(G^t\right)}{R-1} \tag{3.3}$$

Algorithm 4 Get repeated gaps

```
repeated_gaps_positions ← []
for r from 0 to R do
    for c from 1 to C do
        current_card ← B_{c,r}
        preceding_card ← B_{r,c−1}
        if is_gap(current_card) and is_gap(preceding_card) then
            append(repeated_gaps_positions, (r, c))
        end if
    end for
end forreturn repeated_gaps_positions
```

3.2 Normalized score

We designed a scoring function to measure how "good" a state is. This score ranges from 0 to 1[1] and is composed of the number of well-placed cards, the number of dead gaps, and the number of repeated gaps. To achieve this, we perform a weighted average of these values. We experimentally chose our weights, with further research required to determine the optimal weights.

This scoring function will be used by our algorithms to progress towards a solved state.

$$S\left(W_X, W_Y, W_Z, G^t\right) = \begin{cases} \infty, & \text{if solved} \\ \frac{W_X X_n\left(G^t\right) + W_Y Y_n\left(G^t\right) + W_Z Z_n\left(G^t\right)}{W_X + W_Y + W_Z}, & \text{otherwise} \end{cases} \tag{3.4}$$

3.3 Estimate distance to goal

To provide an estimate of the distance to the goal H. We start with the number of cards in our game to which we substract the number well-placed cards, providing the number of misplaced cards. Additionally, we add a penalty term to this estimate, which accounts for the number of dead gaps and repeated gaps. We gathered them in a single term because it makes it easier to interpret the value, this way W_p indicates how many misplaced cards the penalty terms should account for. In other words, it represents how much further from the goal the value of the penalty term places us.

$$H\left(W_p, G^t\right) = X\left(G^t\right) + W_p \frac{X_n\left(G^t\right) + Y_n\left(G^t\right)}{2} \tag{3.5}$$

$H\left(0, G^t\right)$ an admissible heuristic, this means that $H\left(0, G^t\right)$ will never overestimate the distance to the goal because from n misplaced cards in the game, we can reach the goal in at least n moves but it usually takes more moves to reach the goal.

3.4 Selected algorithms

3.4.1 Greedy BFS

Solving our game involves progressing from a root state to possible leaf states. A leaf state can be either a trap state, where no action is available, or a solved state. Thus, our algorithm aims to reach the leaf states as quickly as possible, making Depth-First Search (DFS) an appropriate approach for this problem. To guide our search, we rely on the score S, which helps the search converge to a solution leaf instead of a terminal one. However, it is still prone to ending up in non-solution terminal nodes since it lacks knowledge of the deeper node states.

3.4.2 A*

Unlike the greedy search, A* explores the state space more methodically and comprehensively. While DFS attempts to go as deep as possible in a straightforward manner, A* maintains a balance between depth and breadth. This careful exploration makes A* less likely to end up in trap states compared to the greedy algorithm. However, A* may take more time due to its broader search frontiers. Despite this, A* is designed to find the shortest path to the solution, making it more efficient to find a solution with the least possible actions.

[1] This range facilitates determining an exploration parameter W_c for MCTS

3.4.3 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that uses random sampling to explore the state space. MCTS is particularly useful for problems with large and complex state spaces, like our solitaire game. It consists of four main steps: selection, expansion, simulation, and backpropagation.

- **Selection:** Starting from the root node, the algorithm selects child nodes based on a policy that balances exploration and exploitation, typically using the Upper Confidence Bounds for Trees (UCT) formula.
- **Expansion**: When a leaf node is reached, if it is not terminal, one or more child nodes are added to the tree.
- **Simulation**: From the newly added node, a simulation (or playout) is run to the end of the game by making random moves.
- **Backpropagation** The result of the simulation is propagated back up the tree, updating the nodes' values.

MCTS has several advantages for our problem. It doesn't require a heuristic function for every state and can handle the large branching factor efficiently through random sampling. The random simulations help MCTS to escape local optima, making it less prone to getting stuck in trap states compared to Greedy BFS or A*. However, MCTS can be computationally intensive due to the number of simulations required to achieve reliable results. Despite this, its ability to balance exploration and exploitation makes it a robust choice for navigating the vast state space of our game.

4. Application of the MCTS

A way to evaluate the possible cards moves, is to use Stochastic optimization method. Stochastic optimization introduce a random element into the solution-finding process, making probabilistic updates at each iteration. This randomness often helps them avoid getting stuck in local minima, improving performance for certain tasks. For example, Monte Carlo Tree Search (MCTS) uses random simulations to estimate the expected score of each option.

These methods efficiently generate solutions with good performance across many problem types, making probabilistic decisions. In our case, we chose the Monte Carlo method due to its effectiveness and relevance to our course.

4.1 General concept

Monte Carlo Tree Search (MCTS) [Browne et al., 2012] is a technique used mainly for decision-making, the main idea is to explore possible options in order to make the best decision.

- **Expansion**: The process starts by representing the current state in the game with a node. The algorithm generates new nodes (child nodes) for any actions that haven't been explored yet, representing possible next moves.
- **Selection**: The algorithm chooses a promising path through the tree. It selects actions based on the scores from past simulations. If an action hasn't been explored yet, it stops expanding here and proceeds with simulation.
- **Simulation/Playout**: From the selected node, the algorithm simulates what might happen next by playing out a series of random moves. The simulation continues until reaching the end of the game or a predefined depth.
- **Return** : Once the simulation ends, the algorithm scores the resulting game state and propagates this score back up the path, updating all nodes involved. Each node's scores are updated to reflect the results of the new simulation.
- **Reiteration/Backpropagation**: The process repeats many times, progressively improving the decision tree's information (quality of choices). Each iteration provides more data, refining the choices and increasing the probability of selecting the best move.

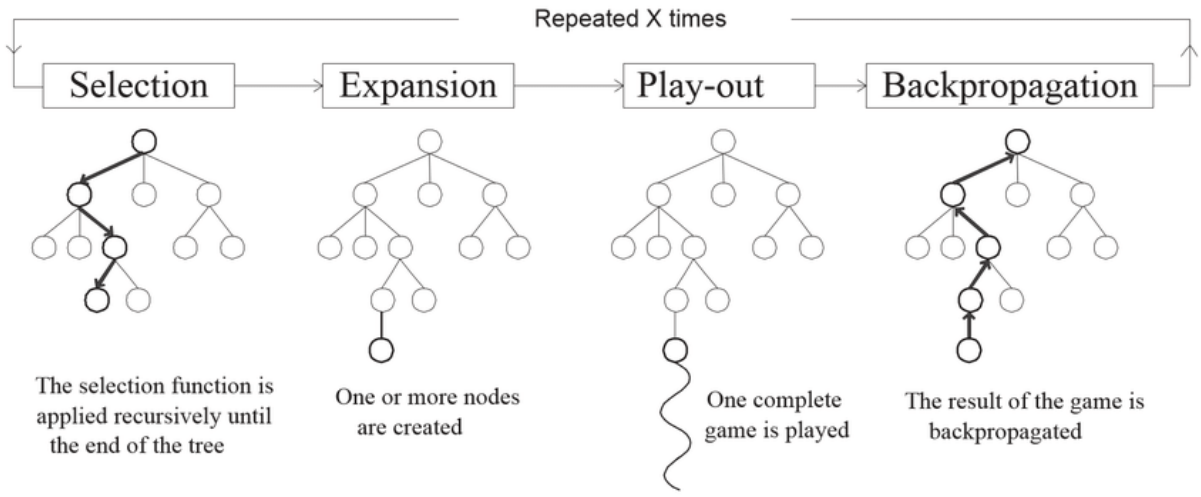

Figure 4.1: Monte-Carlo Tree Search scheme from [Mańdziuk, 2018]

4.2 Application to the Gaps game

Our solver for the Gaps card game uses the standard four phases of MCTS: Selection, Expansion, Simulation, and Backpropagation.

Here is how these steps are implemented:

4.2.1 Selection

We start from the root node and select the child node with the highest score, based on the Upper Confidence Bound (UCB) formula. This formula balances the exploration of new nodes with the exploitation of known nodes. The UCB formula is defined as:

$$\text{UCB}(G^t, W_c) = \frac{w_i}{n_i} + W_c\sqrt{\frac{\ln N}{n_i}}$$

From this we add an additional term to the UCB formula to include a heuristic score S with a certain weight w:

$$\text{UCB}(G^t, W_c, W_S) = \frac{w_i}{n_i} + W_c\sqrt{\frac{\ln N}{n_i}} + W_S S(G^t)$$

Having a higher UCB value means that the node is more promising and should be selected for further exploration, therefore adding the heuristic score to the UCB formula helps to guide the search towards more promising nodes. However we still want to explore new nodes, so we keep the exploration term in the formula and set a relatively low weight for the heuristic score.

Recall that $S\left(G^t\right) = \infty$ for solved states, this means that the algorithm will always select a node if promise to provide a solution since $\infty >$ UCB for any node

We used S function with MCTS with the parameters $W_X = 10$, $W_Y = 2$, and $W_Z = 1$. These values were chosen experimentally and can be adjusted to improve the performance of the algorithm. We decide to use S and not H because it is easier to determine a good value for the W_c parameter, which is used to balance exploration and exploitation. Aswell as the weight of the heuristic W_S. If we used the function H as the heuristic, it would be harder to determine a good value for W_c and W_S since the two terms are not on the same scale.

4.2.2 Expansion

From the selected node, the algorithm expands the tree by generating child nodes for all possible actions. This step is crucial for exploring new paths and increasing the tree's depth.

To select the next move, the algorithm select the 4 first best child node based on their respective value given by S and sample one of them randomly with a linearly decreasing probability. This allows the algorithm to explore new paths while still focusing on the most promising ones. If one of these most promising nodes present a score of ∞, the algorithm will select it without any randomness.

4.2.3 Simulation

From the newly expanded node, the algorithm simulates the game's progress using random moves until we either:

1. Reach a solved state
2. Reach a trap state
3. Reach the computational depth limits (set to 100 in our experiments)

The exploration is done similarly to the expansion phase, we select the 4 best child nodes based on their respective score S and sample one of them randomly with a linearly decreasing probability. If one of these most promising nodes present a score of ∞, the algorithm will select it without any randomness.

4.2.4 Backpropagation

After the simulation ends, the algorithm updates the scores of all nodes involved in the simulation. The score is updated based on the simulation result, with the score of the solved state being ∞ this will propagate back to the current node being explored, indicating that the node will lead to a solved state.

4.2.5 Path finding

The algorithm repeats these steps for a predefined number of iterations, in our case 1,000. After the iterations are completed, the algorithm selects the child node with the highest score and returns the path from the root node to this node. We can reconstruct the path because through the expansion phase we created the children we wanted to explore, and we can keep track of the parent of each node.

5. A* Algorithm

Another approach for evaluating and choosing moves in the Gaps game is to use a deterministic method. Unlike stochastic methods, this approach will always produce the same output given a specific input.

A* (A-star) is a pathfinding and graph-searching algorithm known for efficiently finding the shortest path between nodes using a combination of cost-based metrics. It maintains a record of the lowest known cost to reach each node from a starting point (G-cost), while estimating the remaining distance to the goal (H-cost). The combination of these values (F-cost) helps prioritize exploration paths, balancing between the cost so far and the estimated cost to reach the goal. This consistency is beneficial for solving the Gaps game effectively and reproducibly.

Unlike the two other methods we have discussed, A* aims to provide the shortest path to the solution, likely yielding a smaller number of moves to solve the game. However, this strength comes with a cost. Since it blends depth-first and breadth-first exploration, and given that in our game the goal is located at the end of the tree, it might take more time to find the solution compared to the other methods.

5.1 Application to the Gaps game

Our A* implementation does not deviate from the standard A* algorithm. As a cost for an action (arc), we chose to use the number of moves made to reach the current state. This cost is incremented by one for each move made. The heuristic is simply our estimate $H(G^t, 0)$ that we defined in section 3.3.

The open and closed sets are managed using a hash value that represents the state of the game. This hash value is calculated based on the current state and is used to check if a state is already in either the open or closed list. This mechanism helps prevent revisiting equivalent states and avoids loops.

For instance, consider a state G^t. If we apply an action a^t to G^t, resulting in state G^{t+1}, it is possible that in G^{t+1}, the optimal action a^{t+1} might lead us back to a state equivalent to G^t (i.e., $G^{t+2} \equiv G^t$). Using hash values to track states ensures that once a state has been explored, it won't be revisited, thus maintaining the efficiency of the algorithm by avoiding loops and redundant calculations.

5.1.1 Initialization

1. We start by adding the initial state to the open list.
2. Create an empty closed list.

5.1.2 Node evaluation

1. Select the node with the lowest F-cost from the open list (F-cost = G-cost + H-cost).
2. Move this node to the closed list.

5.1.3 Goal check

1. Check if the current node is the goal state, if so, return the path from the initial state to the current node.

5.1.4 Neighbor exploration

1. For each neighbor of the current node:
   (a) Calculate the G-cost, which is the cost to reach the this node from the initial state (G-cost of the current node + 1).
   (b) Calculate the H-cost, which is the heuristic estimate of the remaining cost to reach the goal (using the $H(G^t, 0)$ heuristic).
   (c) Calculate the F-cost, which is the sum of the G-cost and H-cost.
   (d) If the neighbor is not in the open list or the closed list, add it to the open list.
   (e) If it is in the open list with a higher F-cost, update the node with the new F-cost and set the parent to be the current node.

5.1.5 Reiteration

1. Repeat the process until the open list is empty or the goal state is reached.

Algorithm 5 A* for Gaps Game

```
open_list ← priority queue containing the start node
closed_list ← empty set
while open_list is not empty do
    current_node ← node from open_list with lowest heuristic value
    if current_node is the goal then
        return the path to current_node
    end if
    remove current_node from open_list
    add current_node to closed_list
    for each neighbor in neighbors of current_node do
        if neighbor is in closed_list then
            continue
        end if
        neighbor.g ← current_node.g + 1          ▷ Increment G-cost by 1
        neighbor.h ← H(G^neighbor, 0)            ▷ Calculate heuristic
        neighbor.f ← neighbor.g + neighbor.h     ▷ Calculate F-cost
        if neighbor is not in open_list then
            add neighbor to open_list
        end if
    end for
end while
return failure
```

5.2 Heuristic Function

The heuristic function is a critical component of the A* algorithm, guiding the search towards the goal. For the Gaps game, we chose the function $H\left(G^t, 0\right)$ 3.3 as being our heuristic, which we consider to be admissible. This means that the $H\left(G^t, 0\right)$ will never overestimate the number of moves needed to reach the goal. If this heuristic is found to not be admissible, the algorithm may not find the optimal solution or may take longer to find it.

6. Greedy BFS Algorithm

Another deterministic approach for evaluating and choosing moves in the Gaps game is to use a greedy algorithm. Greedy algorithms are known for making the best choice at each step, with the hope that this will lead to the best possible solution. In the context of the Gaps game, this means selecting the move that will maximize the number of well-placed cards while being penalized for dead gaps and repeated gaps.

This algorithm is well suited for the Gaps game because we aim to reach a leaf node as quickly as possible. The algorithm is guided by the score S, which helps the search converge to a solution leaf instead of a terminal one. However, it is still prone to ending up in non-solution terminal nodes since it lacks knowledge of the deeper node states.

6.1 Algorithm

Essentially, in our implementation, we used the code for A* as a base for our Greedy BFS algorithm. The main difference between the two algorithms is the G-cost calculation.

In A*, the G-cost is the number of moves made to reach the current state, while in Greedy BFS, we set the G-cost to 0 for all nodes, which makes the algorithm behave like a best-first search.

To enable a more careful selection at each step than A*, we chose to use $H(G^t, 2)$ as the heuristic. This heuristic takes into account the number of dead gaps and repeated gaps, and will account for at most 2 misplaced cards.

Therefore we can use the same algorithm as A* with the following modifications:

- Set the G-cost to 0 for all nodes
- Use $H(G^t, 2)$ as the heuristic

This way at each step we will select the node that will minimize the number of misplaced cards in the game and avoid dead-gaps and repeated gaps.

Algorithm 6 Greedy Best-First Search for Gaps Game

```
open_list ← priority queue containing the start node
closed_list ← empty set
while open_list is not empty do
    current_node ← node from open_list with lowest heuristic value
    if current_node is the goal then
        return the path to current_node
    end if
    remove current_node from open_list
    add current_node to closed_list
    for each neighbor in neighbors of current_node do
        if neighbor is in closed_list then
            continue
        end if
        neighbor.g ← 0                                  ▷ Set G-cost to 0 for all nodes
        neighbor.h ← H(G^neighbor, 2)                   ▷ Calculate heuristic
        neighbor.f ← neighbor.g + neighbor.h            ▷ Calculate F-cost (here F-cost = H-cost)
        if neighbor is not in open_list then
            add neighbor to open_list
        end if
    end for
end while
return failure
```

7. Experiments and results

To evaluate the performance of the algorithms, we conducted experiments on different board sizes ranging from 4×5 to 4×13, with only the number of columns varying. For each board size, we considered games of increasing complexity: 20, 25, 30, and 35. For each level of complexity, we generated 5 games and ran each algorithm on each game. This resulted in a total of:

$$(13 - 5 + 1) \cdot 4 \cdot 5 = 180 \text{ games}$$

With 3 algorithms to test (Greedy BFS, A*, and MCTS), we had a total of:

$$180 \cdot 3 = 540 \text{ game instances to solve}$$

For each run, we measured the following metrics:

1. **Time taken to solve the game**: This metric helps in understanding the computational efficiency of each algorithm.
2. **Path length to a possible solution**: This provides insight into the optimality of the solution found by each algorithm.
3. **Success rate**: This indicates the robustness and reliability of each algorithm in finding a solution.

The experimental results will help us compare the performance of deterministic algorithms (Greedy BFS and A*) with the stochastic MCTS algorithm, particularly in terms of reproducibility, efficiency, and solution quality.

For MCTS we chose to perform 1,000 iterations for each game, as this value was found to be a good trade-off between the time taken to solve the game and the quality of the solution found. We also experimentally chose to set the exploration parameter W_c to 0.3 and the weight of the heuristic to 0.2, these provided good results be can still be adjusted to improve the performance of the algorithm.

For every game dimension and complexity level, we set an increasing timeout in second to avoid the algorithm to run indefinitely, this timeout is defined from the following formula:

$$t(C, N) = 8(C - 4) + 0.7(N - 20)$$

This ranges from 8 seconds for the simplest game to 82.5 seconds for the most complex game.

7.1 Results by size

Size	Method	Path length	Time elapsed (s)	Success rate	Timeout rate
20	A*	14.59	0.83	0.85	0.15
	Greedy	20.25	0.07	1.00	0.00
	MCTS	27.35	3.27	1.00	0.00
24	A*	17.43	4.30	0.70	0.30
	Greedy	22.00	0.06	0.80	0.20
	MCTS	30.35	5.89	0.85	0.05
28	A*	18.09	0.71	0.55	0.40
	Greedy	29.56	0.32	0.80	0.15
	MCTS	36.31	8.39	0.65	0.05
32	A*	20.83	4.24	0.60	0.40
	Greedy	23.25	0.43	0.60	0.40
	MCTS	35.64	12.21	0.70	0.00
36	A*	17.50	0.86	0.40	0.40
	Greedy	29.14	0.54	0.70	0.10
	MCTS	31.40	12.21	0.50	0.10
40	A*	22.22	2.88	0.45	0.45
	Greedy	30.07	0.31	0.70	0.20
	MCTS	45.70	21.87	0.50	0.10
44	A*	20.55	1.52	0.55	0.40
	Greedy	26.31	1.06	0.80	0.15
	MCTS	38.93	22.69	0.75	0.00
48	A*	20.89	7.44	0.45	0.50
	Greedy	27.75	0.11	0.60	0.35
	MCTS	36.30	86.44	0.50	0.30
52	A*	19.78	3.49	0.45	0.45
	Greedy	28.42	1.98	0.60	0.30
	MCTS	35.75	27.47	0.40	0.20

Table 7.1: "Size" is the number of cell in the board ($R \times C$), the columns "Path length" and "Time elapsed (s)" display their average values for the games that could have been solved. Not doing so would average the timeout values and a path length of 1 for the games that could not be solved, which would not be representative of the actual performance of the algorithm.

7.2 Overall results

method	Time elapsed (s)	Path length	Success rate	Timeout rate
A*	2.81	18.73	0.56	0.38
Greedy	0.51	26.01	0.73	0.21
MCTS	18.90	34.51	0.65	0.09

Table 7.2: "Path length" and "Time elapsed (s)" display their average values for the games that could have been solved. Not doing so would average the timeout values and a path length of 1 for the games that could not be solved, which would not be representative of the actual performance of the algorithm.

7.3 Performance by size

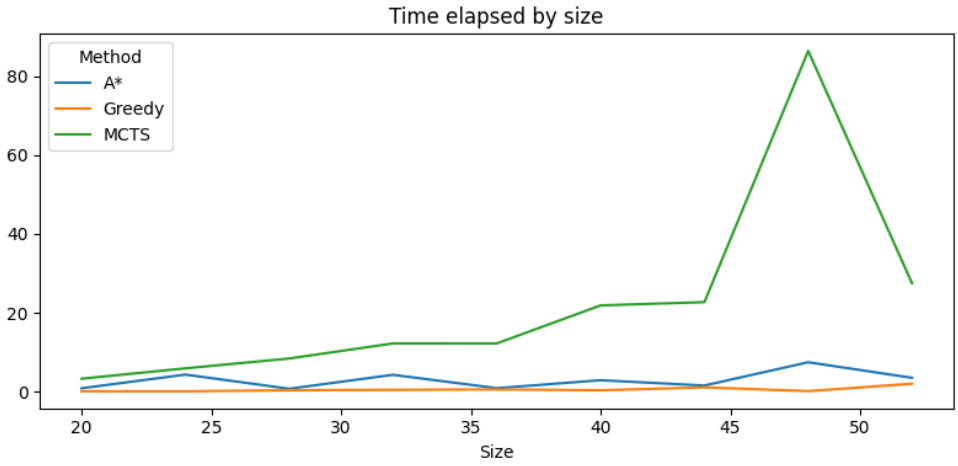

Figure 7.1

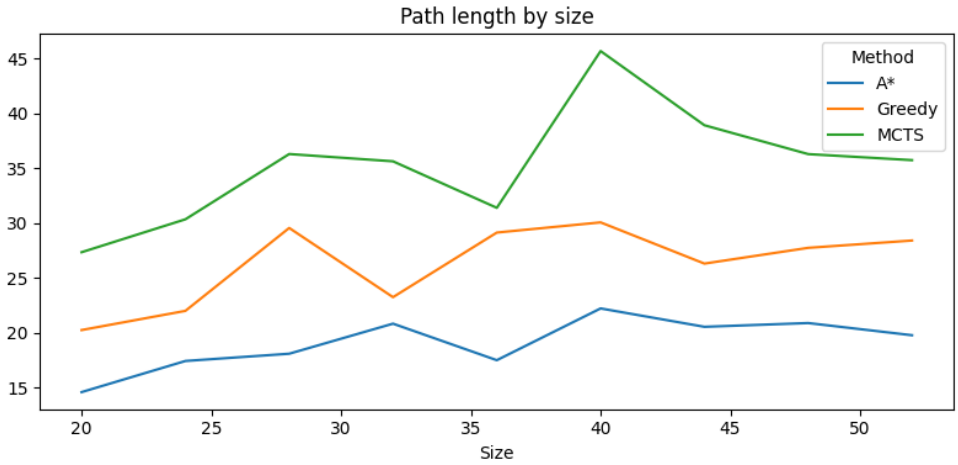

Figure 7.2

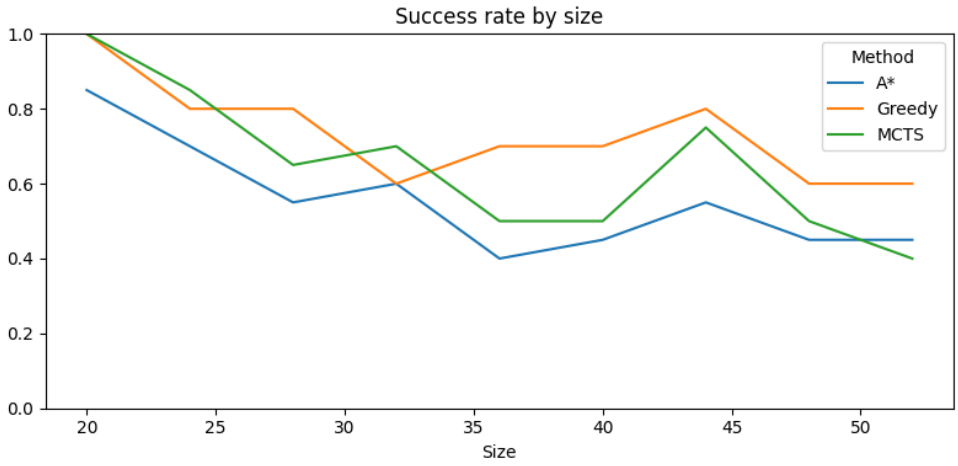

Figure 7.3

7.3.1 Overall performance

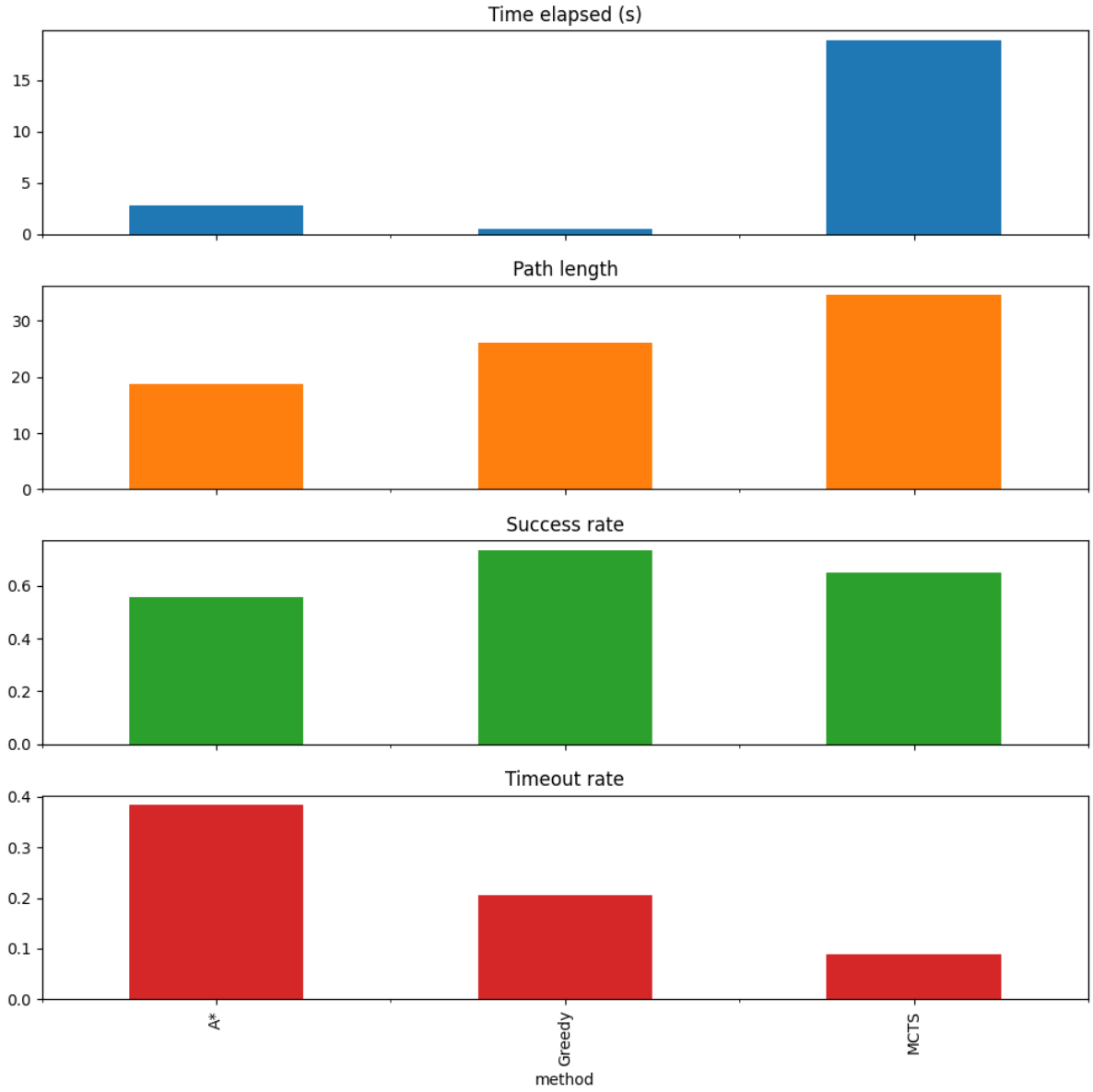

Figure 7.4: Overall performance

7.4 Reproducibility

To ensure reproducibility, we implemented a simple linear congruential generator (LCG) to generate random numbers given a seed. This seed initializes the random number generator, ensuring that the same sequence of random numbers is generated each time the algorithm is run with the same seed. This approach allows us to reproduce the same results consistently. The LCG is used whenever random numbers are needed in our algorithms, such as in the shuffling of the game and in the MCTS algorithm.

Each game we generate has an associated seed, enabling the reproduction of the same game configuration. An identical board can be generated by using the same number of rows, columns, and seed.

Greedy BFS and A* are deterministic algorithms, meaning they will always find the same solution if given the same initial state. On the other hand, MCTS is a stochastic algorithm, involving some randomness in the decision-making process. To allow for reproducibility in MCTS, we used

our LCG with a seed that depends on a default value plus the current iteration number. This approach ensures that different actions are taken in each iteration, while still allowing us to reproduce the same results for MCTS.

The code which contains the seed and all the parameters used to generate the game is available in the `src/headless.ts` file. The script can be run with the following command: `npm run headless`.

7.5 Justification of the weight choice

The weights and scoring methods for evaluating game states were chosen after thorough experimentation and analysis. We first identified the crucial factors influencing the game's solvability, such as the number of well-placed cards and the presence of problematic gaps. Each factor was assigned a weight based on its importance in achieving the game's objective, with higher weights given to criteria that directly contribute to successful solutions.

The weights were then fine-tuned by simulating different game scenarios and observing how various scoring combinations impacted the MCTS and A* algorithms' performance. With repeated adjustments, we developed a balanced scoring system that prioritizes well-placed cards while minimizing the effect of challenging gaps. This setup enables the algorithm to consistently select good moves for solving the Gaps card game.

8. Discussions

8.1 Path length analysis

In figure 7.4 we can see that the method yielding to the shortest path length is, as expected, A*. This is due to the fact that A* is designed to find the shortest path to the solution. MCTS provides the longest path length, this is likely due to the stochastic nature of the algorithm, which can lead to suboptimal moves; it also does not naturaly prevent loops in the path unlike the two other methods. Greedy BFS, is in between the two other methods, this is because it does not aim to find the shortest path but rather to reach a leaf node as quickly as possible, in general if the heuristic is well designed it should provide descent path lengths.

8.2 Time complexity analysis

The most time-consuming algorithm to solve the Gaps game is MCTS, this is because it requires a large number of iterations to find a good solution. In contrast the Greedy BFS algorithm simply requires to jump from one node to another by selecting the best move, this is why it is the fastest algorithm, as soon as it reaches a leaf node it returns it, no matter if it is a solution or not. A* is in between the two other methods, it is slower than Greedy BFS but faster than MCTS, this is because it explores the state space with a balance between depth and breadth.

8.3 Success rate analysis

The algorithm with the highest success rate is, surprisingly, the simpliest one, the Greedy BFS algorithm. This is likely because unlike A^*, the Greedy BFS algorithm does not aim to find the shortest path to the solution, it just follows the best path at each step. MCTS is not far behind, and unlike the two other methods, we can improve its success rate by increasing the number of iterations. A^* has the lowest success rate, this is due to the fact it spends more time descending the tree to find the shortest path which leads more often reach the timeout before finding a solution, this is demonstrated in the last plot of figure 7.4.

Having a solution which takes a long time to find is not useful in practice, this is why we do not consider setting up a longer timeout for further experiments.

8.4 A*

The A* algorithm is also effective for solving the Gaps game because it narrows down the search efficiently by focusing on promising paths. However, if the goal is not to find a short path to the solution, this makes A* the least promising method for solving this game, as it is the slowest (reaches the timeout more often) and has the lowest success rate.

8.5 Greedy BFS

The Greedy BFS algorithm is the fastest and most efficient algorithm to solve the Gaps game. It is well suited for this game because it does not aim to find the shortest path to the solution, but rather to reach a leaf node as quickly as possible. This is why it has the highest success rate. However, it does not guarantee a short path to the solution.

This shows that the having a good way of counting the number of misplaced cards can already provide a good heuristic for finding a solution to the Gaps game. And as mentioned in the

section 3.1.1 finding a good way of counting the number of well-placed is not trivial, we believe that better solutions can be found by improving this fun

8.6 MCTS

MCTS is well suited for the Gaps game because it allows for the exploration of a large state space in a depth-focused manner. Its backpropagation process enables the algorithm to efficiently and progressively converge towards a solution. However, this comes at the cost of high computational resources due to the significant number of iterations required, resulting in longer times to find a solution.

Better results can potentially be achieved by fine-tuning the number of iterations, the exploration constant W_c, and the weight constant W_S in the UCB formula. Additionally, improving the selection and expansion policies can further enhance the performance of MCTS in solving the Gaps game.

9. Code and final product

We chose to implement our algorithms in TypeScript[1] using React[2]. This is because we wanted to showcase our algorithms in a user-friendly way. This way one can play the game, see the algorithms in action, and understand how they work simply by visiting our website at the following URL: `https://quaffel.github.io/gaps/`.

9.1 Algorithms

We implemented three algorithms: A*, MCTS, and Greedy BFS. The A* and MCTS algorithms in a separated logic file, there are located in folder `src/logic/solver`. The Greedy BFS algorithm is implemented in the same file as the A* algorithm, but it is called with different parameters.

[1] https://www.typescriptlang.org/
[2] https://react.dev/

10. Conclusion

In this work, we explored and evaluated different algorithms for solving the Gaps game, focusing on Greedy BFS, A*, and MCTS. Each algorithm demonstrated unique strengths and weaknesses, revealing insights into their suitability for this specific problem domain.

Greedy BFS proved to be the fastest and most efficient algorithm, achieving the highest success rate due to its strategy of quickly reaching a leaf node without seeking the shortest path. This efficiency demonstrates the effectiveness of a straightforward, heuristic-based approach for solving the Gaps game.

The A* algorithm, while efficient in narrowing down searches to promising paths, showed limitations in this context. Its focus on finding the shortest path often led to longer computation times and a higher likelihood of reaching the timeout before finding a solution, resulting in the lowest success rate among the three methods.

MCTS, on the other hand, demonstrated the ability to explore large state spaces in a depth-focused manner, gradually converging towards a solution through its backpropagation process. Despite requiring significant computational resources and longer times to find solutions, MCTS showed potential for improvement through parameter tuning and policy enhancements.

Our analysis highlights the importance of selecting the appropriate algorithm based on the specific objectives and constraints of the problem. For the Gaps game, where quick solutions are preferable, Greedy BFS emerged as the most practical choice. However, for scenarios where path optimality is crucial, enhancements to A* and MCTS could offer better performance.

Future work could explore hybrid approaches, combining elements of both MCTS and A*, to leverage their strengths and mitigate their weaknesses. Additionally, automated tuning of parameters and dynamic heuristic adjustments present promising avenues for optimizing algorithm performance. Exploring alternative methods, such as modeling the Gaps game as a Constraint Satisfaction Problem (CSP), could also lead to more effective heuristic functions and improved overall performance.

In conclusion, while each algorithm has its own advantages, the choice of method should align with the specific goals and constraints of the task. Our findings contribute to a deeper understanding of algorithmic strategies for solving the Gaps game and open up new possibilities for further research and optimization in this area.

11. Further works

11.1 Hybrid approach

Combining elements of both MCTS and A* could create a hybrid algorithm that leverages the strengths of each. For instance, the initial exploration phase could use MCTS to explore a broader set of possibilities and identify promising paths. Once the solution space is narrowed down, A* could then be employed to find an optimal path efficiently. This hybrid method could ensure robust exploration while also benefiting from the deterministic precision of A*.

11.2 Automated tuning of parameters

Automated methods like grid search or Bayesian optimization could be used to fine-tune the algorithms' hyperparameters. This includes determining the ideal exploration-exploitation balance for MCTS or finding the most appropriate heuristic weight combinations for A*. By automating parameter optimization, future implementations can ensure more optimal performance in solving the Gaps game.

11.3 Dynamic heuristic adjustments

The current implementations of MCTS and A* rely on static weights for evaluating game states. An improvement could involve dynamic adjustments to these weights based on game progression. For example, as the game approach its completion, the weight given to double gaps might be decreased while emphasizing well-placed cards. Adaptive weight adjustments could improve each algorithm's ability to make more contextually relevant decisions, further improving their pathfinding efficacy.

11.4 Using Constraint Satisfaction Problems (CSP) to model the Gaps game

As mentioned in the discussion, finding a good way of counting the number of well-placed cards is not trivial. A better solution could be found by improving this function. A possible approach could involve using Constraint Satisfaction Problems (CSP) to model the Gaps game and find a more efficient way of counting well-placed cards. By exploring alternative methods for evaluating game states, we can potentially enhance the performance of our algorithms in solving the Gaps game since they all heavily rely on the heuristic function.

Bibliography

[Browne et al., 2012] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

[Figure, 2012] Figure, S. (2012). *Monte Carlo Tree Search for the Hide-and-Seek Game Scotland Yard*. ResearchGate.

[Mańdziuk, 2018] Mańdziuk, J. (2018). *MCTS/UCT in solving real-life problems*, pages 277–292.

[Russell and Norvig, 2020] Russell, S. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.

[Świechowski et al., 2023] Świechowski, M., Godlewski, K., Sawicki, B., and Mańdziuk, J. (2023). Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562.