Cake Sort

Heuristic Search Methods for One-Player Solitaire Games

How it works

- Cake Sort is a single-player game in which the primary goal is to assemble complete cakes by organizing plates with slices of the same type.
- Each complete cake is formed by combining six identical slices, which are then cleared from the board, earning you points or even bonuses. When a plate is empty it will also be cleared from the board.
- After positioning one plate in the board, you cannot change its place, so the player must be careful when placing the plates.
- When the board runs out of space, you lose; if it doesn't, the game will go on indefinitely for as long as there is space.
- New types and combinations of cakes will show up during the game to make it harder.
- This game has no time limits, allowing a more relaxed game play.

State Representation

The board is represented as an array of empty spaces or plate objects with slice objects, the empty spaces are just a None object, and a plate is simply and array of slice objects. The slices in a plate will be represented as an Int for each flavor and the empty space in a plate is represented by a None. The size of the board and other settings in the game can be changed to make it harder.

Initial state

The initial state of the game will depend on the settings defined beforehand, but it will always start with an empty board and at least 3 plates outside the board that must be placed by the player in the game.

Objective Test

The game only ends when the player can't place any more plates on the board because there is no space left. Otherwise, the player can always play indefinitely with the goal to reach a better score. The score is determined by the number of completed cakes and bonuses made throughout the game. In each move, the player must aim to complete more cakes and free the most space possible.

Operators (Actions in Game)

- Place Plate - (Precondition: There is at least one empty slot on the board to place a new plate and there are plates to be placed. Effect: The selected plate from outside the board is placed into the selected empty slot. If possible, the game automatically transfers slices which might lead to one or more complete cakes. If one or more cakes are completed by the move, the game increases the score accordingly. If not even one cake is completed, or a plate is freed of slices the score won't change. Cost: 1 - (w * S) - > The basic cost of placing a cake is 1 but if the score S is increased by the cake completion, then the cost will be lower.)

  For example:

  If no cake is completed or plate freed (S = 0), cost = 1.

  If a complete cake gives S = 20 and weight w = 0.1, then cost = 1 – 2= -1.

  If the move doesn't complete a cake but leaves one slot empty by transferring all the slices of a plate leaving it empty, the score would increase S=10 and then cost = 1-1=0 for a w=0.1.

Heuristic: Estimating Potential Score Gain & A* Formulation

Objective:

Maximize long-term score by choosing plate placements that unlock high-scoring opportunities.

Operator: Place Plate

- Immediate Cost:
  $c = 1 - (w \times S)$
  (Score S reduces cost.)

Heuristic Concept:

- Estimate the maximum additional score available from the current state (S_{pot}).
- Heuristic Function:
  $h(n) = -(w \times S_{pot})$

  (Higher potential score → lower estimated cost.)

A* Formulation:

$f(n) = g(n) + h(n)$

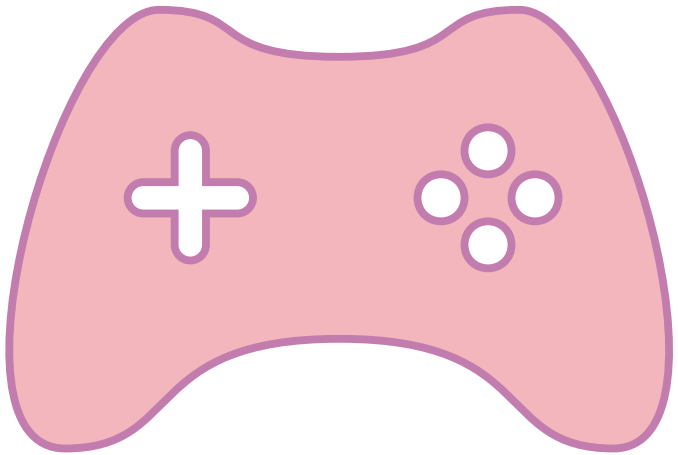
- g(n): Cumulative cost so far (sum of move costs).
- h(n): Estimated future cost based on potential score gain. The heuristic used doesn't calculate the perfect potential score since for that we had to calculate the best moves. Instead, we made this heuristic the most accurate as possible, but the results might be slightly bigger than the actual best score in some scenarios.

Interpretation:

The algorithm favors moves that yield high future scores, guiding the search toward states with greater scoring potential.

Implementation work already carried out

- The game is being developed in Python using the Pygame library for common game features. There is a main file that initializes and sets up values, and it also contains the main game loop.
- All the elements in the game are implemented as class instances. Additionally, each game element class has an associated renderer class that handles its drawing.
- The Board class maintains a grid (a list of lists) that can hold either None values or Plate objects. Plate objects contain a list that can hold either None values or Slice objects, while Slice objects have only a flavor attribute. Finally, there is a Table object that holds a list of either None or Plate objects.
- For the game logic, we have an input handler that registers interactions between the player and the game—specifically, for selecting and placing plates. Additionally, after placing a plate, the slice_merger module will handle slice transactions if they are possible and appropriate.



Search Algorithms

- Uninformed Search
  - Depth First Search (DFS) and Breadth First Search (BFS) - They are both slow solutions for high depths but become optimal in this scenarios.
- Informed Search
  - Greedy Search - It's the fastest algorithm used in the entire project but also the one that returns the least optimal results.
  - A* Algorithm – this algorithm finds a pretty good balance between the time it runs and the optimality of the results. It also showed similar results to the Monte Carlo algorithm but with a better run time. In the A* algorithm we used an auxiliar heuristic to predict the best score ahead with the most accuracy but without consuming much time. This heuristic had a slight overhead then the actual best possible score.
- Metaheuristic
  - Monte Carlo-this algorithm leverages random simulations to guide decision-making, so its performance depends a bit on that randomness. It showed a good balance between runtime and optimality, but A* was slightly faster.

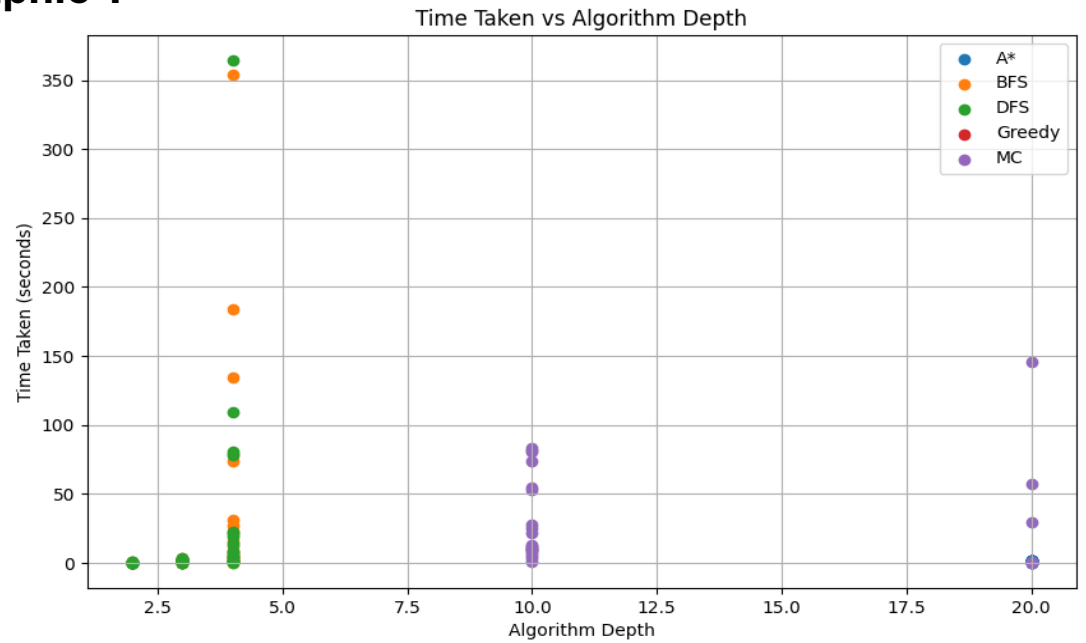
Algorithm Analyses

The graphic 1 shows the relations between the algorithms search depth and time taken to choose a move.
As we can see basic search algorithms like DFS and BFS have their execution time exponential increased with the search depth. For a simple depth of 4 it already was taking upwards of 5 minutes to make the first move, which is not practical.

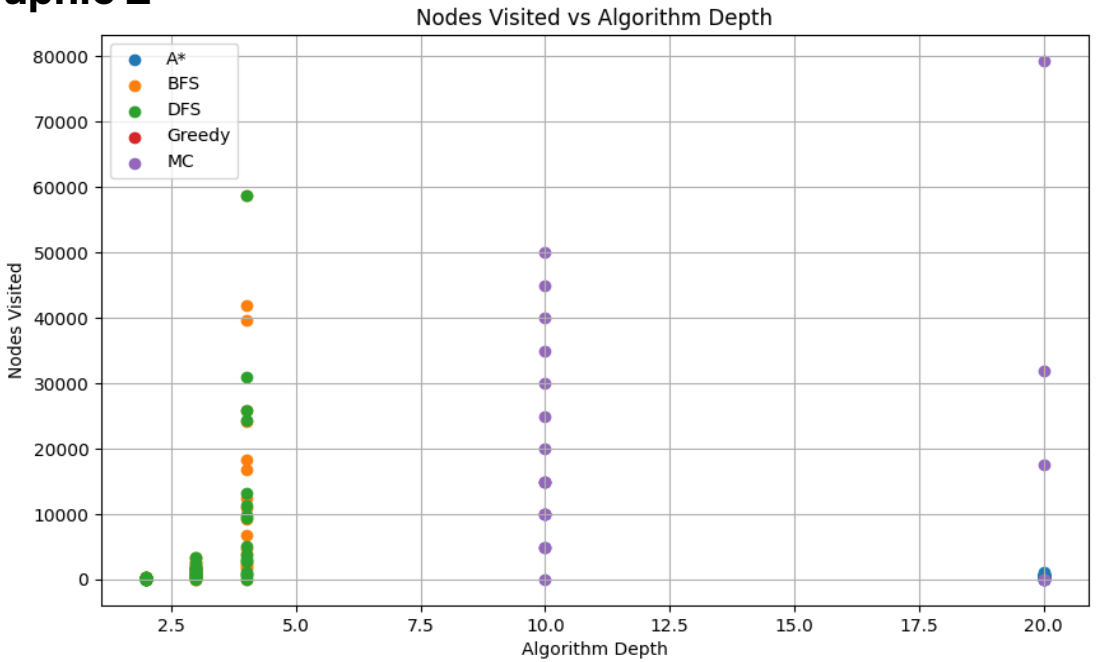
On the other hand, more complex algorithms like Monte Carlo and A* can search way bigger depths (in the graphic we can see 10 and 20) and obtain time results way more bearable. The greedy algorithm has a fixed depth of 1 and is always extremely fast. We can also see that the time taken is proportional to the number of nodes explored, as shown in the graphic 2.

But what about the effects on the score?

Graphic 1



Graphic 2



Algorithm Analyses

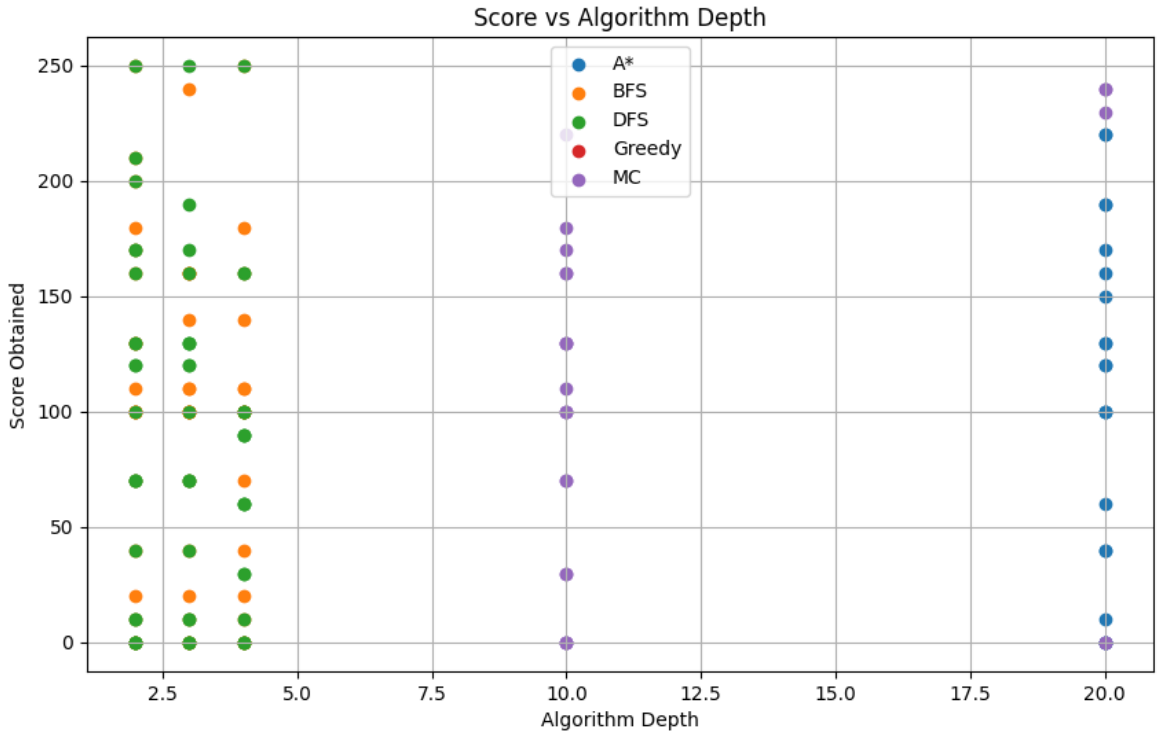
The graphic 3 shows the relation between the different algorithms, their depth and their score.

The greedy algorithm, in the controlled test ambient (20 cakes on deterministic order) obtained the worst result, with only 130 points. That was expected because this algorithm nature, where it can miss the solution path by choosing others that fail but are more promising earlier.

On the other hand, BFS and DFS obtained the best results (250), which was also expected because they exhaust all possible paths and are sure to find the best one. Even though they are limited by a low depth, the small case test and the nature of the game makes this low depth not that impacting.

Finally, algorithms like A* and Monte Carlo obtained high results (210 to 240) in more reasonable times and even though they are not perfect they find a really good balance in good result and good execution time.

Graphic 3



Conclusions

In conclusion, our project demonstrates clear trade-offs between the different search and optimization approaches:

- **Uninformed Searches (DFS & BFS):**
  They guarantee optimal solutions in smaller search spaces because they explore systematically. However, as the depth increases, they become prohibitively slow and resource-intensive.

- **Greedy Search:**
  Although it is the fastest—because it only looks one move ahead—it often returns the least optimal results. Its myopic nature means it can miss better long-term strategies, making it the worst choice when solution quality is critical.

- **A* Algorithm:**
  A* strikes a good balance between run time and solution quality. Its use of a heuristic allows it to efficiently guide the search, often yielding near-optimal results much faster than uninformed searches.

- **Monte Carlo (Metaheuristic):**
  Monte Carlo leverages random simulations to statistically estimate the value of moves. It tends to produce results comparable to A* in terms of solution quality, but it generally requires more computation time due to the large number of iterations needed for reliable estimates.

Additional Information

References:

- https://youtu.be/EBOI59_39ow?si=kDjLzr57Vs4PYEzG
- https://play.google.com/store/apps/details?hl=en_US&id=com.falcon.dm.water.cake.sort.puzzle&utm_source=chatgpt.com

Materials Used

- Pygame
- VSCode
- Pandas for the graphics and data
- CSV to store results