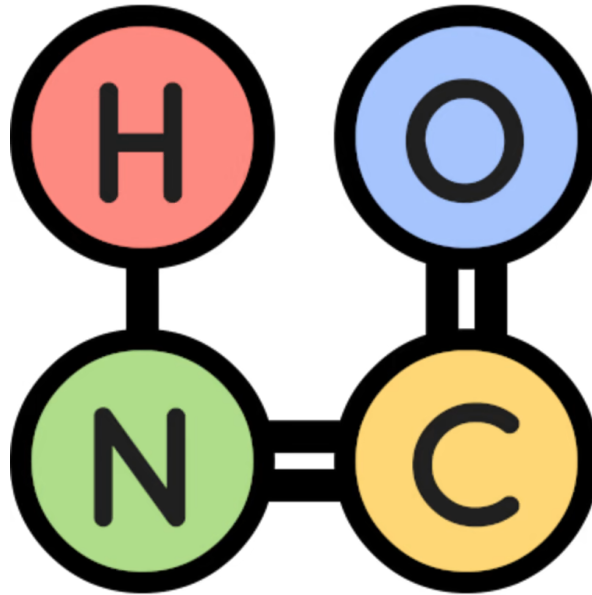


Sokobond with heuristic Search Methods



- Sokobond with heuristic Search Methods
 - Summary
 - Installation and usage
 - Game Controls
 - Algorithms controls
 - Hint - AI Assistance
 - Definition of the game
 - Formulation of the problem as a search problem
 - Uninformed search methods
 - Depth-First Search (DFS)
 - Breadth-First Search (BFS)
 - Informed search methods
 - Best-first Search
 - Greedy Algorithm
 - A* Algorithm
 - Results and analysis
 - Execution time
 - Solution Quality
 - State space explored
 - Conclusions
 - Authors

Summary

The project focuses on developing heuristic search methods for solving one-player solitaire games, with **Sokobond** being the chosen game for implementation. Utilizing Python along with the Pygame library, the project aims to provide both a playable solitaire game for human players and an automated solver capable of tackling various levels of Sokobond puzzles.

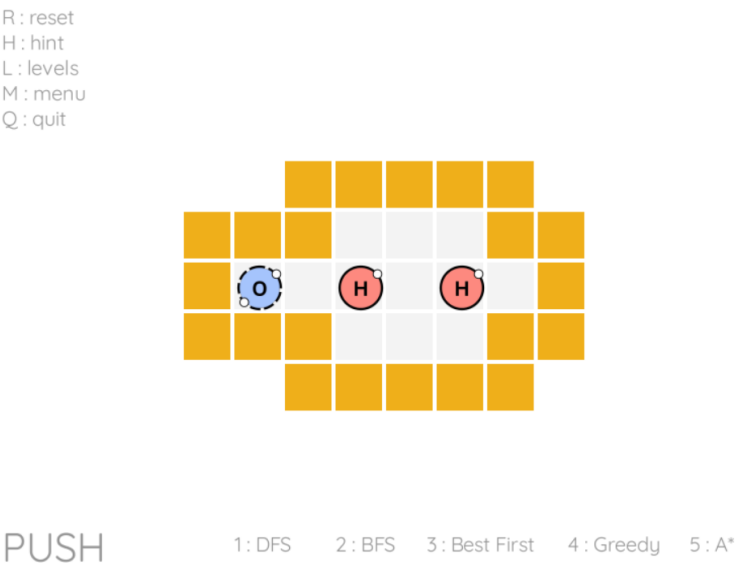
In addition to creating an engaging gaming experience, the project emphasizes the implementation of heuristic search algorithms for solving Sokobond puzzles efficiently. Special attention is given to comparing different uninformed search methods to evaluate their effectiveness in solving the game's puzzles.

By combining game development with artificial intelligence techniques, the project provides a platform for exploring and understanding heuristic search methods in the context of puzzle-solving games. Through experimentation and analysis, the project aims to shed light on the strengths and limitations of various search algorithms in tackling challenging solitaire game scenarios.

Installation and usage

1. Ensure you have Python installed on your system.
2. Install the Pygame library using pip: `pip install pygame`.
3. Download the Sokobond game files from the repository.
4. Run the game script using Python: `python sokobond.py`.

Game Controls



- **Movement** : Use the arrow keys to move the hero atom.
- **Reset Level** : Press the 'R' key.
- **Hint** : Press the 'H' key.
- **Level Menu** : Press the 'L' key.
- **Main Menu**: Press the 'M' key.
- **Quit Game** : Press the 'Q' key.

Algorithms controls

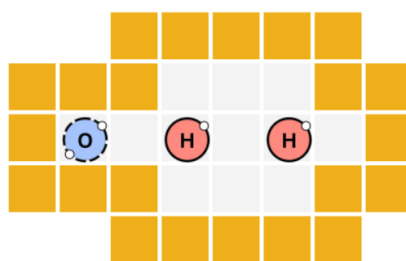
- **Depth-First Search** : Press the '1' key.
- **Breadth-First Search** : Press the '2' key.
- **Best-first Search** : Press the '3' key.
- **Greedy Algorithm**: Press the '4' key.
- *A Algorithm** : Press the '5' key.

After selecting the algorithm, the AI assistant will automatically solve the level, displaying the chosen solution step by step.

Hint - AI Assistance

R : reset
H : hint
L : levels
M : menu
Q : quit

RIGHT



PUSH

1: DFS 2: BFS 3: Best First 4: Greedy 5: A*

- The AI component provides hints and solutions to help you progress through challenging puzzles, using the A* algorithm.
- Press the 'H' key to activate AI assistance, and it will suggest the next move.




Definition of the game

Sokobond is a puzzle game with a chemistry theme. It involves using logic and planning to move atoms around a 2D grid to form specific chemical compounds. Even though the game is centered around creating molecules, you don't need any prior knowledge of chemistry to play and enjoy it.

There are five elements introduced:

- **He** : Helium (0 bonds)
- **H** : Hydrogen (1 bond)
- **O** : Oxygen (2 bonds)
- **N** : Nitrogen (3 bonds)
- **C** : Carbon (4 bonds)

There are also 3 powerups in the game:

-  -> turns the molecule into a snake (if possible)
-  -> Duplicate the connection (if possible)
-  -> Cut the connection.

Formulation of the problem as a search problem

- **State representation** : Board with the molecules distributed in their respective positions.
- **Initial State** : Board with the molecules distributed in their respective initial positions.
- **Objective Test** : All atoms present on the board with no valence electrons (available bonds) remaining.
- **Operators** : Move the atom (Hero) to
 - **Up**. Precondition: the atom can move upwards. Cost: 1.
 - **Down**. Precondition: the atom can move downwards. Cost: 1.
 - **Left**. Precondition: the atom can move to the left. Cost: 1.
 - **Right**. Precondition: the atom can move to the right. Cost: 1.
- **Solution Cost** : Total number of moves required to reach the objective test.

Uninformed search methods

Depth-First Search (DFS)

In DFS, the algorithm traverses depth-wise from the root node to the furthest node possible, exploring each branch completely before moving on to the next branch. This search method is implemented with a depth limit of 30, ensuring that the search does not continue indefinitely and providing a balance between exploration and efficiency.

```
def dfs(board, visited, path, depth, limit):
    nextBoards = Algorithms.getNextBoards(board, visited)
    visited.add(board)

    if board.win():
        return path

    if depth >= limit:
        return []

    for nextBoard, direction in nextBoards:
        path_to_win = Algorithms.dfs(nextBoard, visited, path + [direction], depth
+ 1, limit)
        if path_to_win:
            return path_to_win
    return []
```

Breadth-First Search (BFS)

In BFS, we systematically explore all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. Thus, the graph is explored level by level, starting from the root node, and moves outward in a breadth-wise manner. By implementing BFS with a depth limit of 30, the algorithm ensures efficient exploration while avoiding infinite loops and unnecessary computational overhead.

```
def bfs(board, limit):
    visited = {board}
    queue = deque([(board, [])])

    while queue:
        current_board, path = queue.popleft()

        if current_board.win():
            return path

        if len(path) >= limit:
            continue

        nextBoards = Algorithms.getNextBoards(current_board, visited)

        for nextBoard, direction in nextBoards:
            queue.append((nextBoard, path + [direction]))
            visited.add(nextBoard)

    return []
```

Informed search methods

Best-first Search

In best-first search, we explored the search space by selecting the most promising node according to a heuristic function. In our implementation, we utilized the nearest neighbor heuristic, which prioritizes nodes based on their proximity to the goal state.

However, it's important to note that this algorithm may not always lead to a solution due to the inherent complexity of the Sokobond game.

```
def bestFirst(board):
    visited = set()
    path = []

    while True:
        if board.win():
            return path

        visited.add(board)
```

```
nextBoards = Algorithms.getNextBoards(board, visited)

if nextBoards == []:
    return []

nextBoard, direction = Algorithms.greedyMove(board, nextBoards)

board = nextBoard
path.append(direction)
```

Greedy Algorithm

The greedy algorithm prioritizes nodes based solely on heuristic information without considering the actual cost of reaching the current position. In our implementation, the cost function ($g(n)$) for the Greedy Algorithm is always set to 0.

This means that the algorithm makes decisions solely based on the heuristic function, which can lead to suboptimal solutions but may be computationally more efficient.

```
def greedySearch(board):
    queue = PriorityQueue()
    visited = {board}

    board.cost = 0
    board.heuristic_estimate = 0
    board.path = []
    queue.push(board)

    while not queue.empty():
        currentBoard = queue.pop()

        if currentBoard.win():
            return currentBoard.path

        nextBoards = Algorithms.getNextBoards(currentBoard, visited)
        for b, direction in nextBoards:
            visited.add(b)
            b.heuristic_estimate = currentBoard.greedyMove(MOVE[direction])
            b.cost = 0
            b.path = currentBoard.path + [direction]
            queue.push(b)

    return []
```

A* Algorithm

A* algorithm combines the benefits of both uniform-cost search and heuristic search. It evaluates nodes based on the sum of two functions: the actual cost $g(n)$, which represents the number of moves to reach the current position, and the heuristic function $h(n)$, which estimates the cost to reach the nearest free atom.

In our implementation, the heuristic function $h(n)$ calculation was designed to consider both the proximity to free atoms and the distance to the closest powerups, assigning them different weights to prioritize connecting atoms first. Specifically, the heuristic function was defined as:

$$h(n) = \text{distance_to_closest_atom}() + 0.1 * \text{distance_to_closest_circle}()$$

This approach allowed the algorithm to balance the importance of forming connections between atoms while also considering the potential benefit of reaching powerups. By adjusting the weights, we aimed to optimize the search strategy for efficiently solving Sokobond puzzles, particularly those involving powerup utilization.

```
def aStar(board):
    queue = PriorityQueue()
    visited = {board}

    board.cost = 0
    board.heuristic_estimate = 0
    board.path = []
    queue.push(board)

    while not queue.empty():
        currentBoard = queue.pop()

        if currentBoard.win():
            return currentBoard.path

        nextBoards = Algorithms.getNextBoards(currentBoard, visited)
        for b, direction in nextBoards:
            visited.add(b)
            b.heuristic_estimate = currentBoard.greedyMove(MOVE[direction]) +
currentBoard.closestCircle(MOVE[direction]) * 0.1
            b.cost = currentBoard.cost + 1
            b.path = currentBoard.path + [direction]
            queue.push(b)

    return []
```

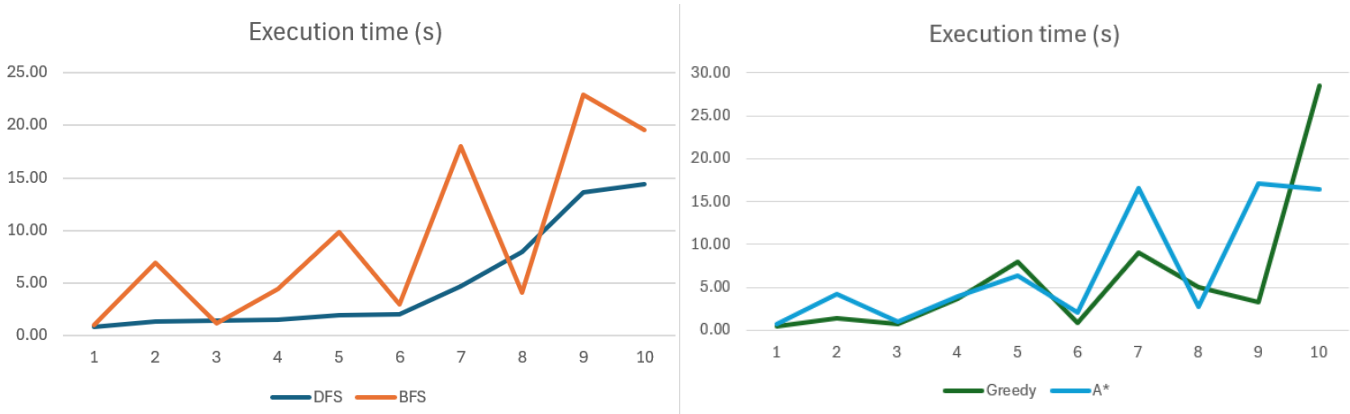
Results and analysis

Due to time constraints, only the first 10 levels were utilized to test the algorithms.

Execution time

Level	DFS	BFS	Best-First	Greedy	A*
1	0.80	1.01	-	0.38	0.75
2	1.35	6.91	-	1.37	4.22
3	1.44	1.13	-	0.76	0.92
4	1.54	4.39	1.07	3.60	3.99
5	1.90	9.83	-	7.94	6.38
6	2.02	2.98	-	0.85	2.11
7	4.70	18.00	1.12	9.00	16.56
8	7.94	4.08	-	4.99	2.78
9	13.65	22.88	-	3.24	17.08
10	14.38	19.58	-	28.48	16.36

The execution time varied for each algorithm across different levels, as illustrated in the provided table.



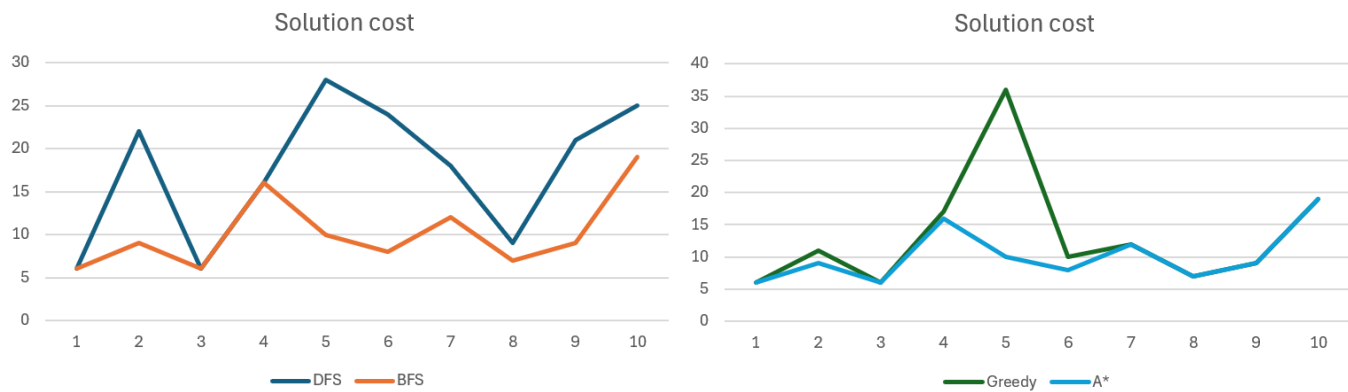
Comparing uninformed search methods, BFS exhibited much longer execution times than DFS, which is expected due to their more exhaustive search strategies.

In informed search methods, the A* Algorithm tended to show a similar pattern to BFS, as it shares similarities with BFS. It's noteworthy that the Best-First Search method, despite yielding solutions in only 2 out of 10 levels, demonstrated significantly quicker execution times compared to all other algorithms.

Solution Quality

Level	DFS	BFS	Best-First	Greedy	A*
1	6	6	-	6	6
2	22	9	-	11	9
3	6	6	-	6	6
4	16	16	17	17	16
5	28	10	-	36	10
6	24	8	14	10	8
7	18	12	-	12	12
8	9	7	-	7	7
9	21	9	-	9	9
10	25	19	-	19	19

The table above depicts the solution quality achieved by different algorithms.

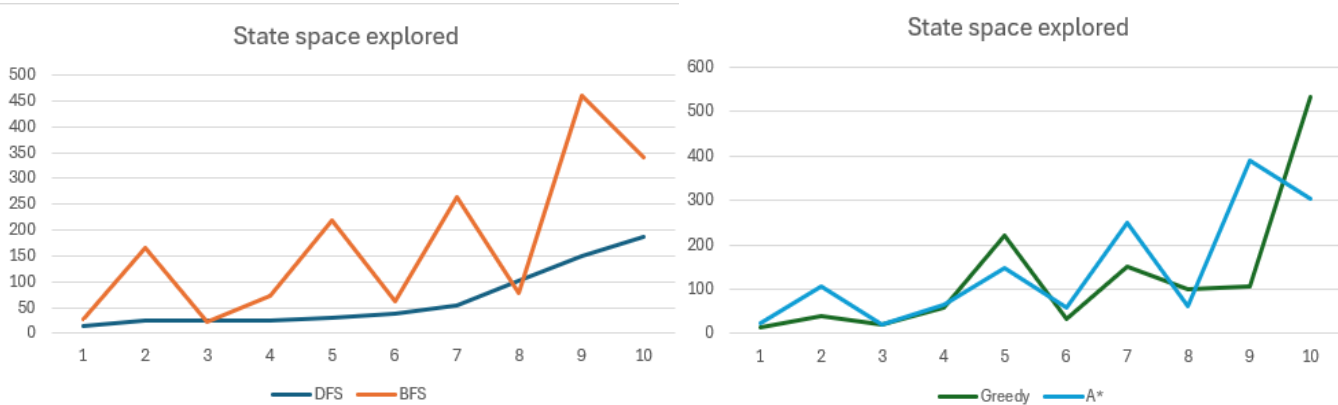


Both BFS and A* Algorithm consistently lead to the optimal solution, demonstrating their effectiveness in finding the most efficient path to solve the Sokobond puzzles.

In contrast, DFS tends to produce suboptimal solutions, with the solution cost scaling up as the complexity of the puzzle increases. This highlights the limitations of DFS in achieving optimal solutions, especially in more challenging scenarios.

State space explored

Level	DFS	BFS	Best-First	Greedy	A*
1	14	27	-	15	23
2	24	166	-	39	107
3	24	23	-	20	21
4	24	72	17	60	66
5	30	219	-	222	148
6	38	62	14	32	59
7	54	263	-	152	249
8	102	77	-	100	62
9	151	459	-	105	391
10	187	340	-	533	305



In terms of memory usage, there is a clear correlation with execution time. BFS and A* Algorithm utilize the most memory since they need to store all visited nodes in memory. This is necessary for their search strategies, which explore the entire state space systematically or guided by heuristics.

On the other hand, the Best-First Search method only visits state spaces corresponding to its best neighbor, resulting in more efficient memory usage. Despite its limited success in finding solutions, it demonstrates efficient memory utilization compared to BFS and A* Algorithm.

Conclusions

In conclusion, this work has provided valuable insights into the performance of various search algorithms in solving Sokobond puzzles. By analyzing the execution time, solution quality, and state space explored by different algorithms, we can draw several conclusions.

While algorithms like BFS and A* consistently lead to optimal solutions, they tend to consume more memory and exhibit longer execution times due to their exhaustive search strategies. On the other hand, algorithms like DFS and Best-First Search may yield suboptimal solutions but offer faster execution times and more efficient memory usage.

Ultimately, the choice of algorithm depends on the specific requirements of the problem and the available computational resources. For scenarios where finding the optimal solution is crucial, algorithms like BFS and A* are preferred despite their higher computational cost. However, for applications where speed and memory efficiency are prioritized, algorithms like DFS and Best-First Search provide viable alternatives.

By understanding the trade-offs between solution quality, execution time, and memory usage, practitioners can make informed decisions when selecting the most appropriate search algorithm for solving Sokobond puzzles or similar problem domains.

Authors

This project was developed by:

- Gonalo Costa
- Lu s Du