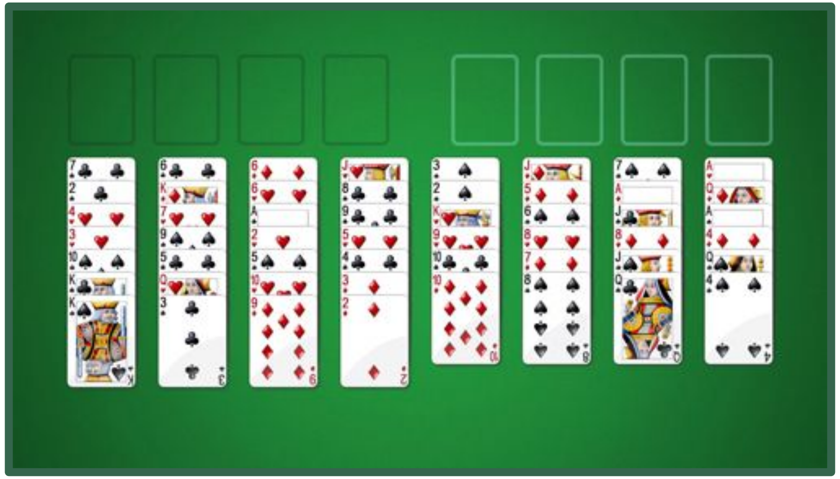
FreeCell Solitaire

AI Project Final Presentation - T03G08

- Afonso Castro @**up202208026**
- Fernando Rodrigues @**up201105620**
- Sofia Bliznyuk @**up202209448**

- Card-based puzzle game played with a standard deck of 52 playing cards.
- Any card at the top of a column or in a Freecell can be moved.
- A card can be placed on another column if it follows descending order and alternates in color (e.g., a red 7 can be placed on a black 8).
- A card can be moved to an empty freecell as a temporary holding space, but only one card per Freecell is allowed.
- A card can be moved to the foundation if it follows the correct ascending order for its suit (e.g., Ace of Spades first, then 2 of Spades, etc.).
- The game is won when all 52 cards are placed in the foundation Piles in the correct order.

- **Tableau (Columns):** Eight columns where cards are placed in descending order and alternating colors.
- **Free Cells:** Four temporary storage spaces for individual cards that can be used to maneuver other cards.
- **Foundation Piles:** Four piles, one for each suit, where cards must be stacked in ascending order from Ace to King.

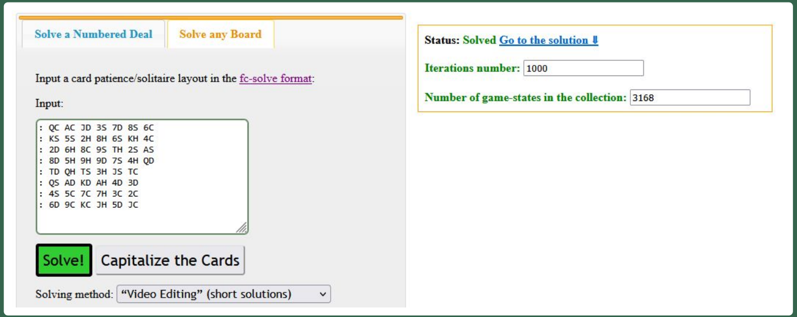


Related Work/References

https://github.com/shlomif/fc-solve/tree/master

https://fc-solve.shlomifish.org/js-fc-solve/text/

- Command line solver written in C
- Various scanning options (DFS, random DFS, best-first, A*)
- Works with several Solitaire-type games
- Online version with limited options



Input a card patience/solitaire layout in the fc-solve format:

Input:

```
: QC AC JD 3S 7D 8S 6C
: KS 5S 2H 8H 6S KH 4C
: 2D 6H 8C 9S TH 2S AS
: 8D 5H 9H 9D 7S 4H QD
: TD QH TS 3H JS TC
: QS AD KD AH 4D 3D
: 4S 5C 7C 7H 3C 2C
: 6D 9C KC JH 5D JC
```

Solving method: “Video Editing” (short solutions)

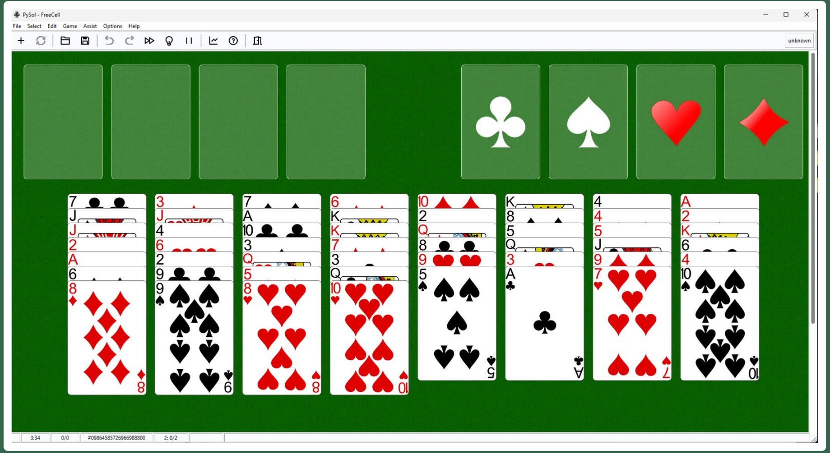
Status: Solved Go to the solution ⇓

Iterations number: 1000

Number of game-states in the collection: 3168

PySolFC

- Popular suite of Solitaire-type games
- fc-solve integration
- Native solver isn't as good



Microsoft Copilot & ChatGPT → Help in explaining concepts and refining code.

The search problem & our approach to it.

State Representation:

- Tableau: 8 columns, each a list of Card(rank, suit, colour) object.
- Free Cells: 4 slots, each containing a Card or None.
- Foundations: Dictionary {suit: [last rank placed]}.
- History: List [Previous States]
- Minutes: None or number of minutes at time of saving.
- Seconds: None or number of seconds at time of saving.

Initial State:

- Tableau: 4 columns with 7 cards, 4 with 6 cards (shuffled deck or preset).
- Free Cells: [None, None, None, None].
- Foundations: {hearts: [last rank placed], diamonds: [...], clubs: [...], spades: [...]}.
- History: []
- Minutes: None
- Seconds: None

Goal Test: Solved if all foundation piles contain 13 cards in ascending order (A → K).
Operators: *Shown in next slide.*

1. **Tableau → Tableau** (move_tableau_to_tableau(state, src, dest))
    - The src tableau column is not empty and its top card can be moved to the destination column `dest`. The top card is removed from the source tableau column, being added to the destination tableau column.
2. **Tableau → Freecell** (move_tableau_to_freecell(state, col))
    - The tableau column `col` is not empty and there is at least one empty free cell. The top card is removed from the tableau column, being placed in an empty free cell.
3. **Freecell → Tableau** (move_freecell_to_tableau(state, fc, col))
    - The free cell `fc` is not empty and the card in it can be moved to the tableau column `col`. The card is removed from the free cell, being added to the tableau column.
4. **Freecell → Foundation** (move_freecell_to_foundation(state, fc))
    - The free cell `fc` is not empty and the card in it can be moved to the foundation. The card is removed from the free cell, being added to the foundation.
5. **Tableau → Foundation** (move_tableau_to_foundation(state, col))
    - The tableau column `col` is not empty and its top card can be moved to the foundation. The top card is removed from the tableau column, being added to the foundation.
6. **Foundation → Tableau** (move_foundation_to_tableau(state, suit, col))
    - The foundation for the suit is not empty and the card there can be moved to the tableau column `col`. The card is removed from the foundation, being added to the tableau column.
7. **Foundation -> Freecell** (move_foundation_to_freecell(state, suit))
    - The foundation for the suit is not empty and there is at least one empty free cell. The card is removed from the foundation, being placed in an empty free cell.

A Star Search & Heuristic

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

- **g(n) ->** Cost so far (number of moves made)
- **h(n) ->** Estimated cost to reach the goal (must be optimistic – never overestimate)

Properties:

- Optimal & complete (finds the shortest solution if one exists).
- Exponential time & high memory usage in worst-case scenarios.

Our A* Implementation:

- Uses a priority queue (heap) sorted by f(n).
- Keeps track of the path with a *came_from* dictionary.
- Applies moves uniformly (each move costs 1).
- Applies automatic moves for efficiency, and checks for supermoves as well.

Heuristic Function Highlights:

- **Foundation Score**
  Encourages moving cards to the foundation.
- **Blocked Next Cards**
  Penalizes cards that block the next required card in a tableau column (deeper penalty if buried deeper).
- **Blocked Free Cells**
  Counts occupied free cells, as they reduce flexibility.
- **Free Columns**
  Rewards empty tableau columns to improve maneuverability.

Key Takeaways:

- The heuristic is admissible (optimistic), ensuring h(n) never overestimates the true cost.
- Weights are tuned experimentally to balance foundation progress with board mobility.
- Our approach combines greedy guidance (via h(n)) with uniform-cost search (via g(n)) for an effective solver.

BFS

Algorithm Overview:

- Explores the state space level by level using a FIFO queue.
- Expands all nodes uniformly, ensuring the shortest solution in terms of moves is found.
- Uses a visited set to prevent redundant exploration and a came_from dictionary to reconstruct the solution path.

Key Implementation Points:

- **Queue** (Initializes with the starting game state at depth 0, and appends neighbor states at increasing depths).
- Ignores moves like "foundation_to_freecell" and "foundation_to_tableau" to avoid unhelpful branches.
- Applies automatic moves (*fcm.apply_automatic_moves(neighbor)*) for efficiency, and checks for supermoves as well.
- Once a solved state is found, the solution path is rebuilt using the *came_from* dictionary.

DFS

Algorithm Overview:

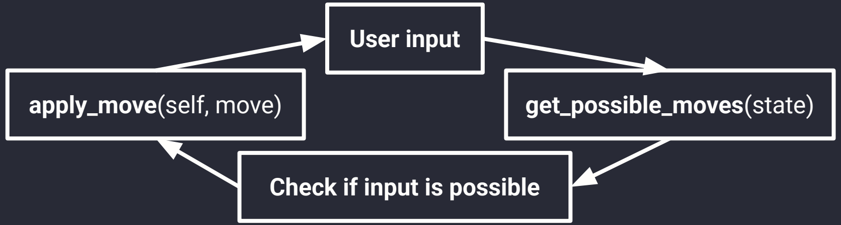
- Explores the state space by diving deep into one path before backtracking.
- Uses a stack (LIFO) to track which state to explore next.
- Includes a depth limit (max_depth=45) to prevent infinite search and excessive memory use.
- Not guaranteed to find the shortest solution, but might be faster in certain scenarios.

Key Implementation Points:

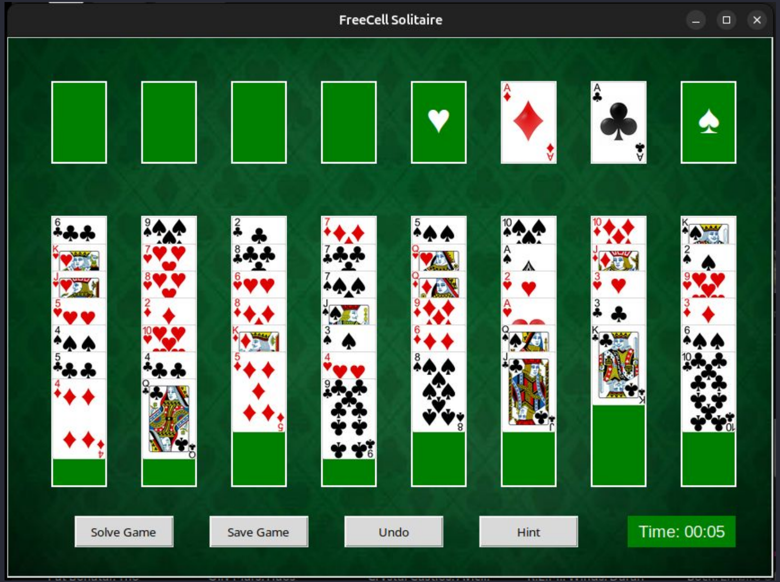
- **Stack:** Initialized with the starting state at depth 0; expands downwards until the limit is reached.
- Ignores moves like "foundation_to_freecell" and "foundation_to_tableau" to avoid unhelpful branches.
- Applies automatic moves (*fcm.apply_automatic_moves(neighbor)*) for efficiency, and checks for supermoves as well.
- Once a solved state is found, the solution path is rebuilt using the *came_from* dictionary.

Work Implementation

- **Main.py** → Entry point
- **FreecellState.py** → Manages game state
- **FreecellMove.py** → Move handling
- **FreecellAI** → Game Solving Algorithms
- **FreecellGUI.py** → GUI implementation
- **FreecellMenu** → Menu Handling
- **Card.py** → Card representation
- **Move.py** → Move representation



```
def __init__(self, tableau, free_cells=None, foundations=None, minutes = None, seconds = None):
        self.tableau = tableau  # 8 tableau columns
        self.free_cells = free_cells if free_cells else [None] * 4  # 4 free cells
        self.foundations = foundations if foundations else {suit: 0 for suit in ['hearts', 'diamonds',
'clubs', 'spades']}
        self.history = []
        self.minutes = minutes
        self.seconds = seconds
```



Automatic Moves implemented to speed up gameplay and algorithms' analysis;
Supermoves implemented for the AI to also speed up algorithms' analysis;
Testing functions using grid search created to test heuristic weights;
Two versions of get_possible_moves created;
Load, Save and Undo Options;
Randomized Hints;
Presets available;
Moves from foundation permitted (but ignored by the AI);
Clock installed;
Memory tracking;

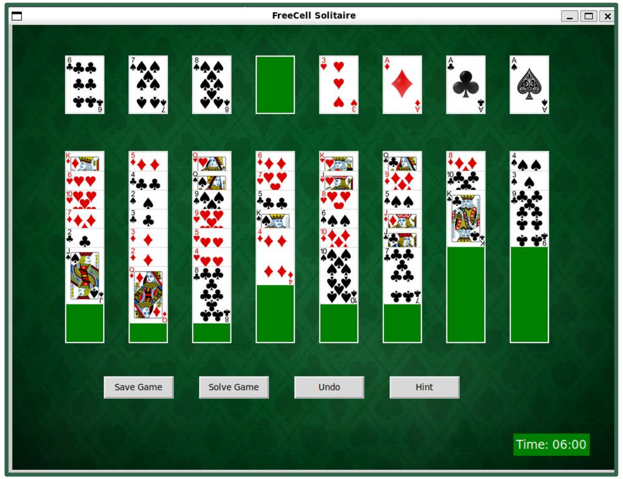
Experimental Results

- **A Star with Heuristics** (foundation_weight = 0.5 , fc_weight = 0, fcol_weight = -0.5, blocked_weight = 0)
- **A Star with Heuristics v2** (foundation_weight = 0.5, fc_weight = 0.2, fcol_weight = -0.5, blocked_weight = 0.3)
- **BFS**
- **DFS** (max_depth=45)

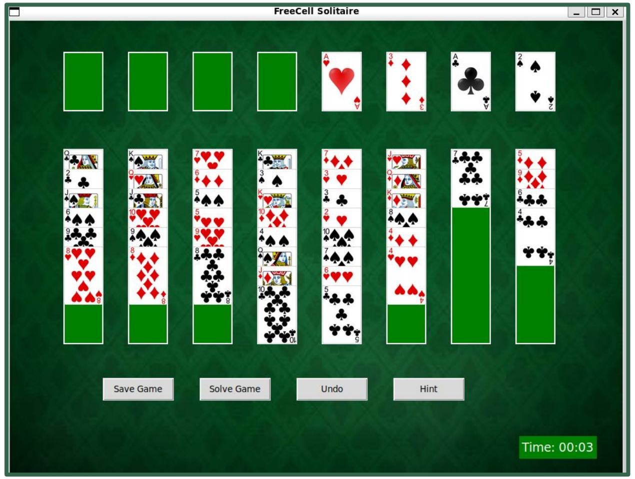
The following three example game states were attempted by the above algorithms, and statistics were collected from their execution.

Heuristic Weights

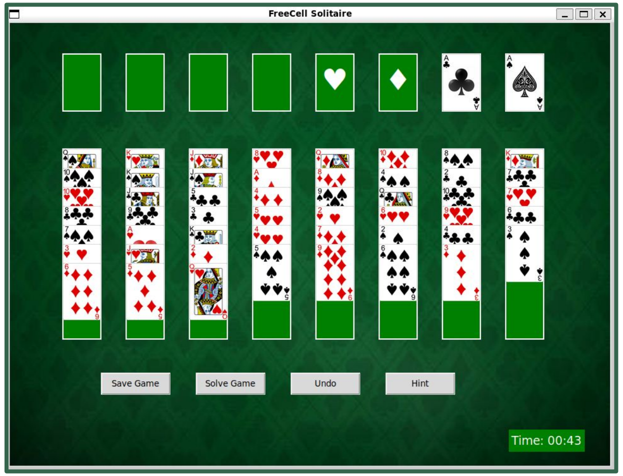
foundation_weight → Encourages progress by prioritizing cards already moved to the foundations.
fc_weight → Penalizes the number of blocked Free Cells (cells holding cards).
fcol_weight → Rewards having free tableau columns by giving a negative score (i.e., lowers the total heuristic).
blocked_weight → Penalizes needed cards that are deeply buried in tableau columns.



Very Easy 2 with 3 Moves

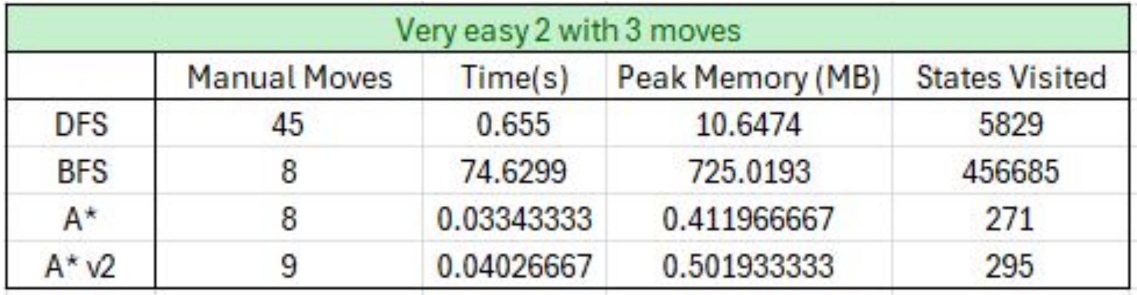


Medium in Progress

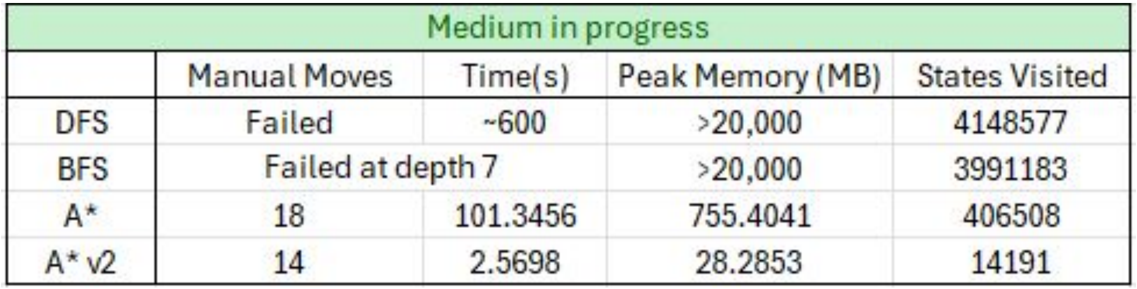


Very Easy 1

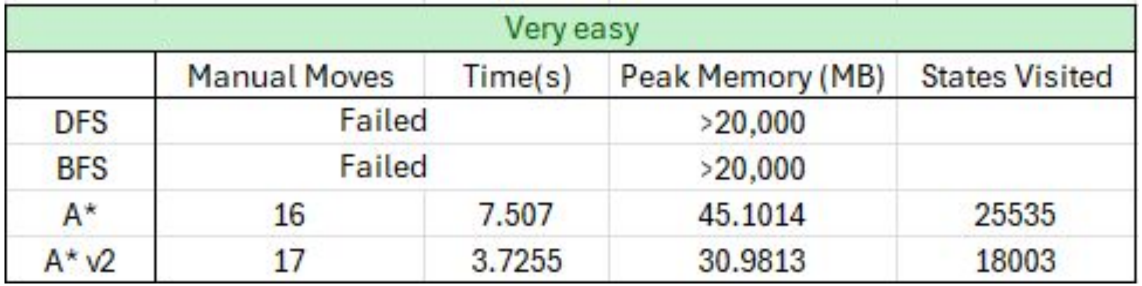
Experimental Results



Very easy 2 with 3 moves				
	Manual Moves	Time(s)	Peak Memory (MB)	States Visited
DFS	45	0.655	10.6474	5829
BFS	8	74.6299	725.0193	456685
A*	8	0.03343333	0.411966667	271
A* v2	9	0.04026667	0.501933333	295



Medium in progress				
	Manual Moves	Time(s)	Peak Memory (MB)	States Visited
DFS	Failed	~600	>20,000	4148577
BFS	Failed at depth 7		>20,000	3991183
A*	18	101.3456	755.4041	406508
A* v2	14	2.5698	28.2853	14191



Very easy				
	Manual Moves	Time(s)	Peak Memory (MB)	States Visited
DFS	Failed		>20,000	
BFS	Failed		>20,000	
A*	16	7.507	45.1014	25535
A* v2	17	3.7255	30.9813	18003

Both DFS and BFS may sometimes fail due to insufficient computer resources, such as memory or RAM, on our end.

Conclusion

This project allowed us to improve our skills in creating basic video games with **Python** and its GUI tool, **Tkinter**.

It also helped us perfect the way we organize our code structure and logic in terms of state management, operators, methods, and other characteristics.

This work enabled us to put into practice our knowledge of algorithms such as **A Star** (with its respective **heuristic**) as well as **BFS** and **DFS**, allowing us to see firsthand the advantages and disadvantages of each algorithm.

A Star proved to be the most efficient time-wise thanks to its heuristic, often finding solutions faster with fewer explored states. However, it can be memory-intensive due to storing a larger number of nodes in memory.

BFS guaranteed the shortest path in terms of the number of moves, but its exploration of all nodes level by level made it slower and even more memory-consuming in larger state spaces.

DFS used less memory compared to the BFS and although it could reach a solution in some cases, it was more likely to get stuck in deep or unhelpful branches and might miss shorter or optimal solutions without proper limiting and due to tableau left-side exploration bias.