Checkpoint

IA - Baker’s Dozen Card Solitaire

Baker’s Dozen Game

Baker's Dozen is a card solitaire using a single pack of 52 playing cards. The game is called that because of the 13 columns in the game.

Rules

Setup:

- Deal cards into 13 columns of 4 cards each.
- Move any top/middle kings to the bottom of their column (keep order if two kings).

Objective:

- Move all cards to four foundations.

Foundations:

- Start with aces, build up by suit (ace to king).

Gameplay:

- Only top cards are playable.
- Build tableau cards downward (any suit).
- Empty columns stay empty.

Win:

- All cards in foundations.

Related Work

This thread provides a **detailed discussion on effective strategies and rules** for playing Baker's Dozen Solitaire, offering insights into common challenges and tips for optimizing gameplay:

https://boardgames.stackexchange.com/questions/7611/what-is-a-good-bakers-dozen-solitaire-strategy

This research paper explores **algorithmic approaches** to solving solitaire games, including heuristic search methods, which can be adapted to develop an AI or optimization strategy for Baker's Dozen:

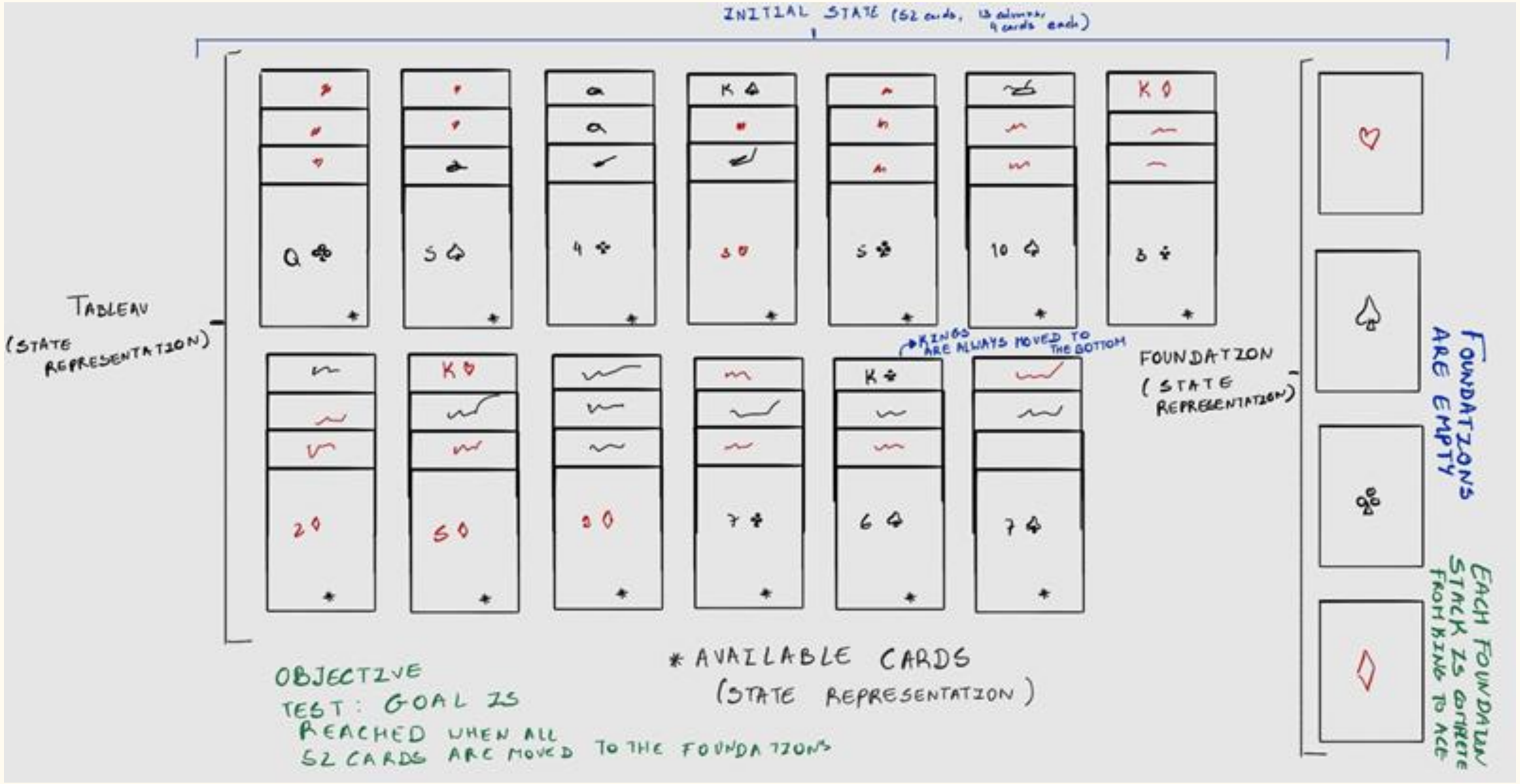
https://ai.dmi.unibas.ch/papers/paul-helmert-icaps2016wshsdip.pdf

This GitHub repository contains a **Java implementation** of various solitaire games, including Baker's Dozen, serving as a practical reference for coding the game's logic and structure:

https://github.com/jarolrod/java-solitaire

Formulation of the Search Problem

State Representation / Initial State / Objective Test


INITIAL STATE (52 cards, 13 columns, 4 cards each)
TABLEAU (STATE REPRESENTATION)
K ♠
K ♦
K ♦
K ♣
Q ♣
5 ♠
4 ♣
3 ♥
5 ♣
10 ♠
3 ♣
2 ♦
5 ♦
8 ♦
7 ♣
6 ♠
7 ♠
KINGS ARE ALWAYS MOVED TO THE BOTTOM
FOUNDATION (STATE REPRESENTATION)
FOUNDATIONS ARE EMPTY
EACH FOUNDATION STACK IS COMPLETE FROM KING TO ACE
* AVAILABLE CARDS (STATE REPRESENTATION)
OBJECTIVE TEST: GOAL IS REACHED WHEN ALL 52 CARDS ARE MOVED TO THE FOUNDATIONS

Operators

- **Name:** MoveColumnToColumn
- **Preconditions:** Source card is the top card of its column AND (target column is empty OR target column's top card has rank one higher than source card).
- **Effects:** Source card is removed from its current column and added to the target column.
- **Cost:** 1.

- **Name:** MoveCardToFoundation
- **Preconditions:** (Card is Ace AND foundation for its suit is empty) OR (Card is next in sequence for its suit's foundation stack).
- **Effects:** Card is removed from the tableau and added to the foundation.
- **Cost:** 1.

Heuristics/Evaluation function

Heuristic Name	**Description**	**Formula**
Cards in Foundations	More cards in foundations indicate progress toward the goal.	h1(s)=Number of cards in foundations
Available Moves	States with more legal moves are preferred.	h2(s)=Number of available moves
Blocked Cards	Penalizes states where cards are blocked by others.	h3(s)=−Number of blocked cards
Sequential Progress	Rewards states with descending sequences in tableau columns.	h4(s)=Number of columns with descending sequences
Distance to Foundation	Estimates how close each card is to being placed in its foundation.	h5(s)=∑(King−Rank of card)×Available
Weighted Combination	Combines multiple heuristics for better guidance.	h6(s)=w1⋅h1(s)+w2⋅h2(s)+w3⋅h3(s)+...

Implementation

- **Programming Language Chosen:** Python - Chosen for simplicity, readability, and extensive libraries.
- **Development Environment:** Visual Studio Code, PyCharm.
- **Data Structures:** Lists

Operators - Details

- **Original Name:** MoveColumnToColumn
- **Actual Definiton:** def drop_card(self, card, from_slot, to_animate=False) (and def check_if_can_drop(self, card, from_slot))

- **Name:** MoveCardToFoundation
- **Actual Definiton:** def drop_card(self, card, from_slot, to_animate=False) (and def check_if_can_drop(self, card, from_slot))

Functionality:

- The **drop_card** function attempts to move a card or stack from one slot to another. It first checks if the move is valid using **check_if_can_drop**. If the move is valid (i.e., **moving_stack** is not None), it removes the cards from the from_slot and adds them to the current slot using add_card. If the move is successful, it returns True, otherwise, it returns False.
- The **check_if_can_drop** function checks if a card (or stack) can be dropped into the current slot by validating the move, rejecting it if the slot can't accept the card, or if it's a foundation and the stack has more than one card; otherwise, it returns the valid stack.

Heuristics/Evaluation function - Details

Heuristic Name	Definition	Functionality
Cards in Foundations	def cards_in_foundation(gameplay_screen)	Returns the total number of cards across all foundation piles by summing the length of each foundation's card list in the gameplay_screen.
Available Moves	def available_moves(gameplay_screen)	Counts how many valid moves can be made by checking, for each top card in the slots, how many other slots can accept that card. It returns the total number of such possible moves.
Blocked Cards	def blocked_cards(gameplay_screen)	Counts how many cards are blocking others in the slots by checking if each card is followed by a valid descending card of a different suit; if not, it increments a counter and returns the total number of such blocked card
Sequential Progress	def sequential_progress(gameplay_screen)	Counts how many slot columns have cards in strict descending rank order by checking each slot, returns the total number of such sequentially ordered columns.
Distance to Foundation	def distance_to_foundation(gameplay_screen)	Finds the smallest rank gap between any top slot card and a compatible foundation card, returns how close that card is to being placed in the foundation (with 13 being the farthest). It uses rank values to calculate this distance and returns 13 - closest_distance to reflect progress.
Weighted Combination	def weighted_combination(gameplay_screen)	Computes an overall heuristic score by combining all heuritics

Algorithms - Details

Algorithm	Definition	Functionality
Depth-first Search	class DFS (def dfs(self, depth=0))	Explores possible moves in the game, tracks visited states to avoid cycles and finds a sequence of moves to reach a winning state, allowing retrieval of the next move in the solution path.
Breadth-first search	class BFS (def bfs(self))	This code implements a BFS algorithm to find a winning game state by exploring all possible moves level by level, avoiding revisiting states, and tracking the path to the solution. The get_move method returns the next move from the found solution path.
Uniform Cost	class UniformCostSearch (def search(self))	Explores possible moves, tracks visited states to avoid cycles and finds the least costly sequence of moves to reach a winning state, providing the next move in the solution path.
A* Algorithm	class AStarAlgorithm	This class would implement the A* search algorithm by exploring paths using both cost and heuristic estimates to efficiently find the optimal path to a winning state.
Weighted A*	class WeightedAStar	This class would implement the Weighted A* algorithm, prioritizing paths using a weighted combination of cost and heuristic to potentially speed up the search at the expense of optimality.

Experimental Results

Algorithm: DFS

Execution Time: 8.36 seconds

Memory Usage: 3107.41 KB

Conclusion

Our implementation and evaluation of search algorithms for solving Baker's Dozen highlighted that informed search methods (A* and Weighted A*) would likely outperform the uninformed ones, but we were only able to fully implement DFS and partially implement BFS and Uniform Cost Search (UCS).

While we were unable to fully evaluate A* and Weighted A*, the DFS implementation demonstrated the potential benefits of heuristic guidance in accelerating the search process and reducing the number of explored states. Based on preliminary results, it is expected that Weighted A* would strike a balance between speed and optimality, making it the most practical choice for real-time or resource-constrained settings.

References

https://boardgames.stackexchange.com/questions/7611/what-is-a-good-bakers-dozen-solitaire-strategy
https://ai.dmi.unibas.ch/papers/paul-helmert-icaps2016wshsdip.pdf
https://github.com/jarolrod/java-solitaire
https://www.solitaire.org/bakers-dozen/
https://www.educative.io/answers/what-is-uniform-cost-search

The End