Heuristic Search Methods for One Player Solitaire Games

Match the Tiles

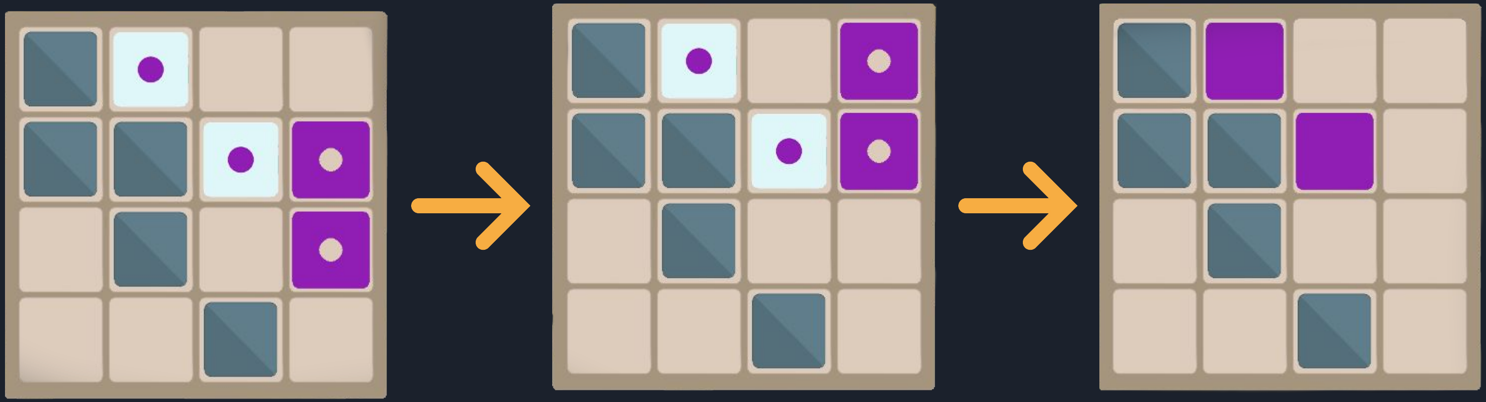
IARTistas
João Diogo Martins Romão, up201806779
João Diogo Vila Franca Gonçalves, up201806162
Rafael Valente Cristino, up201806680

Game Specification

Type of board and pieces | The board is represented by a NxN grid containing obstacles that block the pieces' movement and a target position for each piece (position of the board where each piece must be to win the game). There are K, K<N-1 pieces and each occupies one position (where no obstacle exists) in the grid.

Rules of movement of the pieces | The player moves all pieces simultaneously. They can only move left, right, up or down. Each piece's movement stops if it encounters another piece, an obstacle or the board's border.

Conditions for ending the game | To win the game all pieces need to be in their respectively colored target position.



Problem Formulation

State representation | The state may be represented by a matrix NxN (**board**[,]) where each element can be a piece of a certain color ('pc', where c represents the color), a target piece of a certain color ('tc', where c represents the color), an obstacle ('o') or an empty space (' '). There are $K < N - 1$ pieces. As all the board elements, apart from pieces ('pc') and correspondent targets ('tc), keep their position in the board throughout the game, we decided to save the pieces and correspondent targets in two separate arrays (named, respectively, **pieces** and **targets**), for easier manipulation and verification.

Initial state | The initial state depends on the level being played, as each one has a different configuration. The initial state has no particular distinction from the other states.

Objective test | The objective test consists of verifying if each piece's position is the same as one of the target positions of the same color.
∀ p ∈ **pieces** : ∃ t ∈ **targets** : t.position == piece.position ∧ t.color == p.color.

Operators

Name	Preconditions	Effects	Cost
moveLeft	∃ p ∈ **pieces** : p.x > 0 ∧ **board**[p.y, p.x - 1] != 'o' ∧ **board**[p.y, p.x - 1] != 'p'	∀ p ∈ **pieces** : if **closest 'o' | 'p'** == null then p.x = 0 else p.x = (**closest 'o' | 'p'**).x + 1	1
moveRight	∃ p ∈ **pieces** : p.x < N - 1 ∧ **board**[p.y, p.x + 1] != 'o' ∧ **board**[p.y, p.x + 1] != 'p'	∀ p ∈ **pieces** : if **closest 'o' | 'p'** == null then p.x = N - 1 else p.x = (**closest 'o' | 'p'**).x - 1	1
moveUp	∃ p ∈ **pieces** : p.y > 0 ∧ **board**[p.y - 1, p.x] != 'o' ∧ **board**[p.y - 1, p.x] != 'p'	∀ p ∈ **pieces** : if **closest 'o' | 'p'** == null then p.y = 0 else p.y = (**closest 'o' | 'p'**).y + 1	1
moveDown	∃ p ∈ **pieces** : p.y < N - 1 ∧ **board**[p.y + 1, p.x] != 'o' ∧ **board**[p.y + 1, p.x] != 'p'	∀ p ∈ **pieces** : if **closest 'o' | 'p'** == null then p.y = N - 1 else p.y = (**closest 'o' | 'p'**).y - 1	1

Where **closest 'o' | 'p'** is the closest obstacle or piece in the direction the piece is moving.

Developed Work

- Implementation of a graphical interface that allows a clean visualization of the algorithms execution and results.
- Implementation of a Command Line Interface that allows a clean visualization of the algorithms execution and results on the command line.
- In the singleplayer mode, the player has the possibility to request a hint. The program returns the direction in which the pieces should be moved, by executing the ***A**** with ***direction*** heuristic that has as initial state the current state of the game.
- In the AI mode, it is possible to choose which algorithm to execute, visualize its execution with clean animations and see the performance analysis.
- In all game modes it is possible to visualize the current moves count, as well as the minimum moves it takes to solve the puzzle.
- **Implemented algorithms**: breadth-first search / uniform cost (in this case equivalent), depth-first search (with a maximum depth of 20), iterative deepening, greedy and A*. In the case of the informed search greedy and A* algorithms, we designed and tested various heuristics.

Heuristics

Initial stage of development

- As a first approach, we implemented a heuristic that takes into account the ***manhattan distances*** from each piece to the correspondent target position.
  We quickly reached the conclusion that this was not an useful heuristic to consider in this search problem as a piece can move multiple places at a time, meaning that a piece could be, for example, 7 positions away from the correspondent target and could reach it with just 2 moves.

- Another heuristic that was taken into account gave higher priority to game states where the pieces are ***aligned*** with the correspondent targets and have ***fewer possibilities of movement***.
  This idea intended to **block** the movement of pieces that are assumed to be in the correct position as they are on the same direction of the target and do not have many chances of moving away, giving more movement freedom to the pieces that are further away from the target.
  This could prove helpful in very specific situations, but as it was verified, it does not prove useful in the vast majority of the possible game states.

Heuristics

Intermediate stage of development

- After the first few initial attempts to design an efficient heuristic, we tried to implement a new one that rewarded the game states where the pieces have a similar ***alignment*** when compared to the correspondent targets.
  With this, a game state is rewarded if the pieces have the same alignment. If the pieces are aligned vertically the state receives the same reward as if they were aligned horizontally. The reward is doubled if the pieces are aligned in both directions and 0 if the pieces are not ordered at all.

- Even considering its limitations, this heuristic received better results that the ones stated previously. There was still margin for improvement as it did not take into consideration several game aspects, such as the obstacles.

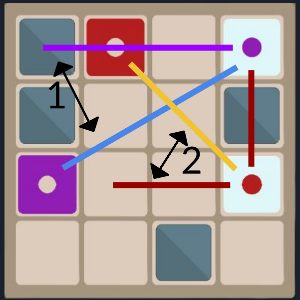
Heuristics

Final stage of development

After experimenting with various heuristics we arrived at one based on the following strategy. For a given game state:

- Reward pieces that are in the position of one of its corresponding targets.
- If a piece is in the direction of one of its corresponding targets:
  - Reward it if there is no obstacle in the straight line to the target.
  - Otherwise, penalise it.
- If the piece is not in the direction of one of its corresponding targets:
  - Penalize it with the deviation of its position to the closest straight line path to the target. This value was calculated in the following way:

```
float Yslope = (float)(target.position.y - piece.position.y) / (float)(target.position.x - piece.position.x);  // is infinity if vertical
float Xslope = (float)(target.position.x - piece.position.x) / (float)(target.position.y - piece.position.y);  // is infinity if horizontal
value += 3*Math.Min(Yslope, Xslope);
```



The value that is obtained, before being multiplied by 3 is always between 0 and 1, and never equal to 0 (for it to be 0 the piece would have to be in the direction of one of the targets).

In the example on the left, the lines of each target's represent the closest straight line path to each target. In the case of the purple piece, from it to the top target, the deviation to the closest straight line path would be less than one (arrow 1). In the case of the red piece, either one of the two red lines represented could be considered the closest straight line path to the target, because the piece has the same deviation from each of them. The value that will be penalized is 1 (arrow 2).

Experimental Results

	BFS/Uniform Cost				DFS				Iterative Deepening				Greedy with manhattan heuristic				Greedy with direction heuristic			
	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)
Lv4	19	17	5	352256	4	6	5	512000	43	39	5	413696	18	16	7	458752	12	17	7	165205
Lv5	25	16	6	555690	17	13	8	421888	125	103	8	618496	16	8	6	379562	20	12	6	462848
Lv6	17	12	6	512000	12	10	6	330410	73	53	6	496981	9	8	6	267605	12	8	6	477866
Lv7	66	35	9	622592	43	56	10	546733	196	219	10	570709	30	25	10	518826	32	25	12	518826
Lv8	100	67	13	626688	79	81	14	574805	452	490	14	548864	72	50	14	488789	32	21	17	360448
Lv9	186	27	13	611669	59	29	18	491520	431	319	18	525653	34	25	13	374101	23	20	13	375466
Lv10	118	29	6	120285	27	14	13	866986	393	329	13	1033557	48	20	6	980309	69	55	6	970752
Lv11	164	43	5	1568768	394	157	16	1432234	220	105	5	1372160	212	56	20	1395370	570	145	9	1189205
Lv12	84	53	7	1485482	68	52	20	1245184	312	357	10	1329834	29	20	7	854698	75	41	9	1196032

Experimental Results

	Greedy with alignment heuristic				Greedy with random heuristic				A* with manhattan heuristic				A* with direction heuristic				A* with alignment heuristic			
	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)	t(ms)	visited nodes	moves	memory (byte)
Lv4	26	16	5	457386	17	13	5	395946	16	13	5	5332480	27	18	7	4436906	26	18	5	446464
Lv5	22	11	6	288085	23	14	6	417792	20	16	6	331776	33	14	6	581632	29	17	6	446464
Lv6	14	10	6	391850	15	10	6	491520	17	11	6	513365	15	11	6	371370	13	11	6	352256
Lv7	26	24	9	378197	39	29	9	339968	42	33	9	453290	34	31	9	517461	42	34	9	368640
Lv8	52	33	17	360448	89	65	17	443733	88	61	13	499712	77	55	13	492885	113	59	13	535210
Lv9	36	22	13	466944	36	26	13	456021	41	25	13	376832	32	25	13	443733	93	27	13	521557
Lv10	168	83	6	887466	313	161	6	794624	27	13	6	740010	40	28	6	932522	75	27	6	958464
Lv11	29	9	5	830122	511	154	9	1221973	72	18	5	951637	215	50	5	1290240	61	13	5	1190570
Lv12	345	194	53	1067690	419	238	7	999424	22	17	7	724992	51	36	7	1149610	59	38	7	1279317

Conclusion

- All algorithms were able to solve all puzzles. Since the puzzles have very distinct characteristics, some algorithms performed very well on some of them and very poorly on the rest.
- In the general case the algorithm that performed the best was the ***A* Search*** with the ***direction heuristic***. However, in the simpler puzzles, the simplest search algorithms (BFS, DFS, Iterative Deepening) and heuristics obtained very good results as solutions are trivial and are found almost instantly.
- The ***Greedy Search*** had very interesting results in the medium difficulty puzzles, having in some of them surpassed the results of the A*. However, while sometimes the greedy search solutions were faster than the A*'s, those would rarely be the optimal solution, that is, the solution with the least number of steps possible. The A* in those cases would take longer but come up with a shorter path.
- Taking into consideration that performing an ***opposite move*** to the one that was performed before brings no new information / game states and can be ignored had a strong impact into the efficiency of the solution.

References and Technologies

References

Game | https://play.google.com/store/apps/details?id=net.bohush.match.tiles.color.puzzle

Sliding tiles AI algorithms | https://visualstudiomagazine.com/articles/2015/10/30/sliding-tiles-c-sharp-ai.aspx

Composition of Basic Heuristics for the Game 2048 | https://theresamigler.files.wordpress.com/2020/03/2048.pdf

Deep Copy of an Object with C# | https://stackoverflow.com/questions/129389/how-do-you-do-a-deep-copy-of-an-object-in-net

Technologies

Unity (version 2019.4.21f1) for the graphical interface.

C# for the game logic and search algorithms.