COMP90054 AI Planning for Autonomy Azul Project Group 09 Report

Jiawei Liao (756560), Yachao Ren (1010409), Albert Tien (758259)

Presentation Video: https://youtu.be/sIPfJZT-K5I

June 2, 2020

1 Introduction

This project aims to develop agents using different techniques to play the Game Azul in the two-player setup.[1]. The best performing agent would participate in a final tournament against other student teams and the staff team agents.

2 Evaluation Metric

The following metrics were used to evaluate and compare the performance of agents over 400 games:

- **Win Rate** (%): Assess the percentage of game won over opponent
- **Average Score** (Points): Assess the ability to minimize penalty given opponent's moves to minimize self score and ability to maximize score given a set of tiles per factory

3 Technique 1: Naive Agents

Naive agents are rule-based and only assesses the player's game_state and the current turn's available moves. The provided naive_Player is an example that ranks moves based on num_to_pattern_line. Naive agents provide us basic insights of how to play the game as a new player, and could be used for pruning in more advanced agents and as benchmark baselines.

3.1 V1: Archur_01.py (Arc_1)

Arc_1 improves on *Naive_player.py*(NP) by fixing a "typo", thus resolves ties of equal num_to_pattern_line using a smaller num_to_floor value. The correction leads to 55.25% win rate against the NP agent.

3.2 V2 and 3: Alb_1 and Alb_1a

*albert_player*_01_*v*3.*py* (Alb_1) uses 3 metrics for better ranking of moves,

- Metric A follows is extended from arc_1 using (num_to pattern_line - num_to_floor_line)
- Metric B rank uses (turn score + end game bonus score)
- Metric C, count_neighbour selects moves surrounded by the most tiles in wall.

*albert_player*_01_*v*3*a.py* (Alb_1a) improved on Alb_1 by removing Metric B's turn score component, since both metric B's turn score component and metric C attempts to select tiles well connected to others. Removing Turn score component eliminated redundancy and transformed a single metric score calculated using addition of sub components into two metrics, so metrics can be ranked by importance.

3.3 Performance Evaluation, Challenges and Improvement

Player vs Player	Win Rate (%)	Score (points)
Np vs Arc_1	42.50% : 55.25%	32.08 : 34.70
Np vs Alb_1	32.25% : 66.50%	32.76 : 40.61
Np vs Alb_1a	30.50% : 66.75%	33.91 : 41.09
Arc_1 vs Alb_1	38.80% : 60.75%	35.85 : 40.47
Arc_1 vs Alb_1a	39.00% : 59.00%	35.11 : 39.34
Alb_1 vs Alb_1a	44.75% : 53.25%	39.63 : 41.35

Table 1: Naive Agents Results (400 runs)

Win rate of alb_1a hits a ceiling after reaching average 67% win rate against agent NP, since naive agent does not consider future states. However, metrics can still select moves beneficial to future turns (e.g. Metric C choose tiles with many neighbours with hope of row, column and set completion in the future). Possible improvement is limited unless agent considers future turns to formulate better metrics. The final baseline chosen was *Albert_player*_01_03*a.py* after eliminating Agents NP, Arc_1 and Alb_1 with 6 comparisons. refer to table (1).

4 Technique Two: Breadth First Search Agents

Breadth First Search(BFS) technique is a search algorithm to search a tree or graph structure data

[1] https://www.ultraboardgames.com/azul/game-rules.php

completely, layer by layer. BFS allows consideration of moves in future turns.

4.1 V2 and 3: Alb_2a and Alb_2b

Albert_player_02_v4, (Alb_2a) considers moves in three consecutive turns, "current turn", "opponent turn" and the next "self turn". For each moves in current turn, BFS finds the best move in the next "self turn" and tie break using best move from "opponent turn". Given that searching every move in 3 turns lead to timeout due to overwhelming number of moves, moves number were pruned after every move rank. Albert_player_02_v6.py (Alb_2b) provided a performance comparison of only using round score as the ranking metric, no extra metrics were used. It's 33.5% low win rate against Alb_2a verified ranking requires multiple metrics.

4.2 Performance Evaluation, Challenges and Improvement

Player vs Player	Win Rate (%)	Score (points)
Alb_1a vs Alb_2a	42.00% : 54.75%	37.29 : 40.51
Alb_1a vs Alb_2b	60.25% : 39.00%	37.61 : 31.42
Alb_2a vs Alb_2b	65.00% : 33.50%	35.76 : 26.59

Table 2: BFS vs naive agent results

One of the benefits of BFS is that it allows consideration of the future turns. However, due to challenges including the potential to timeout, move number requires pruning and BFS loses it's "breadth" advantage, in contrast, Depth first search may lead to performance improvement. Our BFS was implemented to compare moves used a single metric score (round score), as verified in naive agent rank moves using single metric is inaccurately and multiple metrics should be used, thus explaining why the performance was so low compared to Alb_1a.

5 Technique 3: Minimax Agents

Minimax is a technique that combines search algorithms and game theory and is commonly used in two-player zero-sum games.([3]) It was designed to minimise the potential loss assuming the worst case scenario the opponent could create for the player.Minimax is also widely used in repeated games, such as Azul. Though Azul is not strictly zero-sum as two players are having separate boards.
The Minimax value is defined by:

$$\underline{v_i} = max_{a_i} min_{a_{-i}} v_i(a_i, a_{-1})$$

where

- a_i denotes the action of our player
- a_{i-1} denotes the action of the opponent
- v_i is a heuristic score to assess how well the player performs at the end of search depth.

At each depth i the algorithm gives the maximum score of the player and at each depth $i-1$ the algorithm gives the minimum score of the player given the opponent's best counter.

5.1 V1: david_player_03.py (Dav_3)

- Search depth: 3
- Actions pruned to 10 candidates using num_to_pattern_line as the sorting method (Descending Order).
- $v_i = R_{me} - R_{op}$, where R denotes round score
- The action in A_3 with the highest v is the action executed.

The basic Minimax algorithm is complete and is impractical to apply. A deep copy of the game_state needs to be constructed for each branch and only about 1500 copies could be made during the 1 second window. The number of available moves at each turn often exceeds 50 in the earlier stages of a round.
To mitigate this, search depth was restricted to 3. The num_pattern_line of moves was used to sort and prune the moves to 10 candidates at each search depth. At the end of search an end-of round score is calculated for the two players and the score difference is used as the v_i. We found the depth of 3 reasonable as online games taught us that a pattern line should be filled using up to 2 moves per round and most poor moves could be pruned.

Against	Win Rate (%)	Avg Score/Game
vs Alb_1a	89.75% : 9.75%	50.13 : 28.64
25-May Tournament	73.00%	46.33

Table 3: Dav_03 Evaluation Results

Using *Alb_1a* as baseline, *Dav_3* outperformed it by huge margins. It secured a 5th place finish in the practice tournament on 25-May, just slightly below the staff_team_top agent. Its performance is not exactly satisfying as it often leaves multiple non-emptied pattern lines at the round end

and does not consider the effect of end-of-game bonuses. The sorting method could still be improved to resolve ties.

5.2 V2: david_player_03d.py (Dav_3d)

We introduced two changes to the V1.

- Added -num_to_pattern_line as the first tie breaker. Added the second tie breaker to look at tile options where the player has lower numbers of the tile color.
- $V_i = S_{me} - S_{op}$, $S = R + W_E \cdot E + W_L \cdot L$ where E denotes the end of game bonus and L denotes the number of non-empty yet not-filled pattern lines, and W_E, W_L are corresponding weights of E, L.
  W_E is dynamic between 0 and 0.33 with respect to the end of search depth such that it gives a lower weighting to end-game bonuses near the end of each round. W_L is statically set as -1.75. Both weights are tuned through trial and error.

Adding the tie breaker alone produced advantage over Dav_3. (50.75% vs 47.25% win-rate, 38.99 vs 37.97 average score). E and L functions assists the agent to look for opportunities in the end-game bonuses such as completing a full column and avoid leaving non-emptied lines at each round end.

Against	Win Rate (%)	Avg Score/Game
vs Alb_1a	93.75% : 5.50%	52.76 : 26.24
vs Dav_3	71.00% : 27.00%	44.45 : 35.10
30-May Tournament	85.37%	50.54

Table 4: Dav_03d Evaluation Results

Dav_3d performed better against Alb_1a compared to Dav_3 with a higher win-rate and average score. It also displayed dominance over Dav_3 in head-to-head trials with over 70% win-rate and much higher score per game. Dav_3d secured the first place finish in the 30-May tournament with similar win-rate as the staff_team_top agent but higher average score per game.

5.3 V3: david_player_03e.py (Dav_3e)

Last improvements over the V2 include

- Added Alpha-Beta pruning method to cut unnecessary branches.([2])
- Added a bonus of 0.75 for getting the first player token in the S function
- Added a start round Minimax session to search up to depth 4 if the player is the first player to utilise the 5 seconds window.
- Now $v_i = S_{me} - 0.9 \cdot S_{op}$

Alpha-Beta pruning allows slightly increased action size. Start round function was used to perform a deeper search. Since Dav_3e does not store any states, it is only used when our agent is the first player. The 0.9 multiplier to S_{op} allows the agent to take more risks to reach for a higher score.

Against	Win Rate (%)	Avg Score/Game
vs Alb_1a	96.25% : 3.50%	53.63 : 25.38
vs Dav_3	75.00% : 24.25%	45.28 : 34.17
vs Dav_3d	55.00% : 42.33%	41.78 : 38.98
1-Jun Tournament	90.67%	55.13

Table 5: Dav_03e Evaluation Results

Dav_3e manages to perform slightly better against Dav_3d and secured another first player finish in the tournament placing well above the staff_team_top agent. This marks the end of Minimax agents development. If given more time, a tree structure could be implemented to store the search results and reuse them in the next turn.

6 Technique 4: Monte Carlo Tree Search Agents (MCTS)

Monte Carlo Tree Search (MCTS) is a popular heuristic search algorithm for games. ([1]) A general MCTS search iteration comprises of four steps:

1. Selection: Start from Root node and select one of children that has not been simulated,u sing a selection policy.
2. Expansion: Select an unexecuted action of the selected node to execute and expand its child game_state node
3. Simulation: Simulate till the terminal state using a roll-out policy and computes the reward.
4. Back Propagation: Back propagate the rewards to its ancestors recursively.

MCTS is ideal as its tree structure allows its search results to be used for the next turn. It could also adopt similar score functions defined for Minimax.

6.1 V1: david_player_04.py (Dav_4)

The basic MCTS agent was defined by:

- No prunning of actions

- UCB1 selection policy:
  $z = \text{argmax}_{a \in A(s)} Q(s,a) + \sqrt{\frac{2 \ln N(s)}{N(s,a)}}$
- Maximum exploration time to 0.95s per turn
- Simulate till the end of round using random move as the roll-out policy
- The v defined in Dav_3d, $W_E = 1$
- $Q(s,a) = \frac{sum(children.v)}{N}$, where N is the number of visits of the node state.
- After 0.95s, the best child of the root node's transition action is the executed move.

Against	Win Rate (%)	Avg Score/Game
vs Alb_1a	50.25% : 46.00%	25.53 : 23.97

Table 6: Dav_04 Evaluation Results

By looking at the replays, the agent's behavior is very random and exhibits large variance in performance. These are likely due to the random roll-out policy and lack of prunning.

6.2 V2: david_player_04a.py (Dav_4a)

The second iteration introduced:

- Pruning of actions using the Dav_3d sorting method, with size of 12.
- Use the pruning method as the roll-out policy where executed move is the first element in the sorted moves list.
- Redefined the L function for penalising non-emptied lines at the end of the round. For each non-emptied line, add (len(pattern-line) - number of tiles left)/len(pattern-line) to L. W_L is adjusted accordingly to -4.

It is necessary to use a higher quality roll-out policy and constrain the number of branches to improve the quality of the simulation. Both the L and E function have to be re-tuned as MCTS simulates till the end of round instead of only three turns into the future.

Against	Win Rate (%)	Avg Score/Game
vs Alb_1a	97.25% : 2.50%	52.31 : 22.37
vs Dav_3	62.00% : 35.25%	39.52 : 32.66
vs Dav_3e	33.50% : 63.00%	34.03 : 40.18

Table 7: Dav_04a Evaluation Results

The new changes drastically improved the performance over *Dav_4*. *Dav_4a* achieved a higher winrate against *Alb_1a* compared to the best Minimax agent *Dav_3e*. It could beat *Dav_3* but falls short against *Dav_3e* . This is easy to comprehend as our roll-out agent is still very simple and the quality of Q is greatly influenced by the quality of the roll-out policy. The Minimax agent Dav_3e is highly tuned and its conservative counter strategy could punish Dav_4a for being over-optimistic, where the much weaker Alb_1a could not.
If given more time, we would try to use Alb_1a as the roll-out agent and utilise the 5 seconds start round window. We would also try to fine tune the selection policy using UCT.

7 Final verdict: Dav_3e vs Dav_4a

Player vs Player	Win-Rate (%)	Avg Score/Game
Dav_3e vs Alb_1a	96.25% : 3.50%	53.63 : 25.38
Dav_4a vs Alb_1a	97.25% : 2.50%	52.31 : 22.37
Dav_3e vs Dav_4a	63.00% : 33.50%	40.18 : 34.03

Table 8: BFS vs One step agent results

We decided to use Dav_3e in the final tournament. Though it is more likely to experience an upset against a weaker opponent, it has very high winrate and has been thoroughly tested in the practice tournaments. The MCTS agent Dav_4a has great potential if we have more time to improve its roll-out and fine tune parameters.

8 Conclusions

We implemented 4 types of agents to play the Game Azul. The naive and BFS agents were created to better understand the game principle, then we constructed our Minimax and MCTS based agents to utlise the leanings. Minimax agents were thoroughly studied and achieved the best performance but MCTS agents could have higher potential if given more development time.

References

[1] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI." In: *AIIDE*. 2008.

[2] Daniel James Edwards and TP Hart. "The alpha-beta heuristic". In: (1961).

[3] Maurice Sion et al. "On general minimax theorems." In: *Pacific Journal of mathematics* 8.1 (1958), pp. 171–176.