COMP 424 Final Project Report: *Popcoin*

Khalil Virji khalil.virji@mail.mcgill.ca
Department of Computer Science
McGill University
260768416

Curtis Docherty curtis.docherty@mail.mcgill.ca
Department of Computer Science
McGill University
260769767

1. Introduction

Our agent, which we will refer to as *popcoin* throughout this report, is inspired by the game-playing techniques seen in class. It uses Monte-Carlo Tree Search equipped with domain-specific knowledge to explore, expand, and update the game tree, as well as select a promising move at the end of each turn. When designing *popcoin*, we developed several iterations of agents, each having a different approach to the game and their own advantages and disadvantages. We empirically picked the best traits from each of these agents and combined them to form *popcoin*. In addition to testing *popcoin* against the benchmark *random_agent*, we also tested it against our previous iterations and achieved favourable results. This report will cover *popcoin* in detail, as well as previous iterations, challenges, and future improvements.

2. Previous Iterations

During the process of designing *popcoin*, we developed several iterations of agents. In particular, we developed a *heuristic_agent* which uses multiple, hand-crafted, heuristics to evaluate all possible next moves and pick the best one. We also developed an *mcts_agent* which uses Monte-Carlo Tree Search with no domain-specific knowledge to expand the game tree and pick a promising next move. Both these agents have different approaches to the game and they proved to be helpful benchmarks when designing and evaluating *popcoin*.

2.1 *Heuristic_agent*

After playing *Colosseum Survival!* for some time, several strategies became apparent. The first was the strategy of, instead of trying to enclose yourself in the largest area possible, try to enclose your opponent in the smallest area possible. This led us to define the heuristics `number_of_adversary_moves` and `distance_to_adversary`, which we try to minimize. The second strategy was that it is often favourable to control the center of the board, especially during the early stages of the game. This led us to define the heuristic `distance_to_center` which we also try to minimize, but value less as the game progresses. At each turn, the *heuristic_agent* will compute all possible next moves, check for winning/losing moves, score each move based on a weighted combination of the above heuristics, then select its next move as the move with the best score. When evaluating

the *heuristic_agent*, we found it performs perfectly against the benchmark *random_agent*, achieving a 100% win ratio over 100 simulations. We also found it performs well against the *mcts_agent*, achieving a 70% win ratio over 100 simulations, and even against *popcoin* on larger board sizes. However, the *heuristic_agent* suffers from one major flaw in that it frequently exceeds the 2 second time limit to select its next move. Given that it must compute `number_of_adversary_moves` for each possible move before returning, we found the *heuristic_agent* can take 7+ seconds to select a move which is well outside the allotted time. As a result, although the *heuristic_agent* was a strong agent with favourable results, we had to focus our efforts on agents that were flexible to terminate at any chosen time and still deliver a promising move.

2.2 *Mcts_agent*

Due to the time constraints of selecting a move, and the inefficient nature of the *heuristic_agent*, a logical next step was to implement a Monte-Carlo Tree Search (MCTS) agent. This is because MCTS is an anytime algorithm, meaning it can return a valid solution to a problem even if it is interrupted before it ends, and it is expected to find better and better solutions the longer it keeps running [1]. Thus by implementing MCTS, we will be able to avoid the time constraint challenge by simply running the search until the time limit, then returning the best move we have seen so far. This is the core idea and motivation behind the *mcts_agent*.

2.2.1 MCTS Game Tree

Monte-Carlo Tree Search is centered around exploring, expanding, and updating the game tree. Before describing the four steps of the *mcts_agent*, namely selection, expansion, simulation, and back-propagation, it is important to first describe the game tree itself. The game tree consists of nodes and edges. Nodes correspond to game states, and edges correspond to `(x,y,dir)` moves from one game state to another. In each node we store the following information:

- `board`: the current state of the game board
- `my_pos`: the current players position
- `adv_pos`: the opponents position
- `parent`: the parent node, `None` if the node is the root
- `parent_action`: the edge (move) between a node and its parent, `None` if the node is the root
- `children`: an array of child nodes, reachable by a single move from `my_pos`
- `number_of_visits`: the number of times this node has been visited in simulations
- `number_of_wins`: the number of times this node has lead to a win for the adversary in simulations
- `number_of_moves`: the number of possible moves the current player can take

1. Anytime algorithm: `https://en.wikipedia.org/wiki/Anytime_algorithm`

- `possible_moves`: a list of all possible moves the current player can take
- `next_move_index`: the next move to explore in the `possible_moves` list
- `max_node`: `True` if we are the current player, `False` if the opponent is the current player
- `fully_expanded`: `True` if this node and all its children have been fully expanded, `False` if not

2.2.2 Selection

The first step of the *mcts_agent* is to select which node to expand next. Starting from the root of the tree, it will propagate down and return a node based on the following tree policy. Note that `evaluate` is a function that returns a numerical score to a node based on the upper confidence bound seen in class. We chose this bound to balance exploration and exploitation trade-offs, and produce a reasonable estimate for the value of a given game state.

```
select(node):
    if there exits an unexplored move from this node: return node
    else if all children have been fully expanded:
        mark node as fully expanded
        return select(node.parent)
    else:
        for child in node.children:
            if child is fully expanded: continue
            else:
                score = evaluate(child)
                if score is the best seen so far:
                    max_child = child
        return select(max_child)

evaluate(child):
    exploitation = child.number_of_wins/child.number_of_visits
    exploration = 2log(child.parent.number_of_visits)/child.number_of_visits
    return exploitation + sqrt(exploration)
```

2.2.3 Expansion

The selection process returns a node to expand based on the tree policy. Once the *mcts_agent* has this node, it expands its children by finding an unexplored move (which is guaranteed to exist from the tree policy), and initializing the corresponding child node. A detailed description of the expansion step is given below. Note that when a new child node is initialized, it will be the opponents turn, so `my_pos` and `adv_pos` are swapped, and `max_node` is negated.

```
expand(node):
    (x,y,dir) = node.possible_moves[node.next_move_index]
```

```
    board_copy = copy(node.board)
    place_barrier(board_copy, x, y, dir)
    child = new node(
        board: board_copy,
        my_pos: adv_pos,
        adv_pos: (x, y),
        parent: node,
        parent_action: (x,y,dir),
        children: [],
        number_of_visits: 0,
        number_of_wins: 0,
        possible_moves: [all possible moves the child can take],
        number_of_moves: len(possible_moves),
        next_move_index: 0,
        max_node: !node.max_node,
        fully_expanded: False
        )
    node.children.append(child)
    node.next_move_index++
    return child
```

2.2.4 Simulation

After the expansion process returns a new child node added to the game tree, the next step for the *mcts_agent* is to run a simulation on that node. A simulation consists of simulating the remainder of the game from the board state of the input node, picking **random** moves for both players. It will return 1 if the current player won, 0 if the adversary won, and 0.5 if there was a tie. A detailed description of a simulation is given below.

```
simulate(node):
    board_copy = copy(node.board)
    my_pos = node.my_pos
    adv_pos = node.adv_pos
    my_turn = True
    if game is already over:
        node.fully_expanded = True
        return score
    while True:
        if game is over:
            return score
        if my_turn:
            (x,y,dir) = get_random_move(my_pos)
            place_barrier(board_copy, x, y, dir)
            my_pos = (x, y)
            my_turn = False
        else:
```

```
(x,y,dir) = get_random_move(adv_pos)
place_barrier(board_copy, x, y, dir)
adv_pos = (x, y)
my_turn = True
```

2.2.5 Back-propagation

Once a simulation has returned a score, the *mcts_agent* will use it to update the game tree. Starting from the node the simulation was run on, the *mcts_agent* will propagate back up the tree, and update its parents `number_of_visits` and `number_of_wins` attributes accordingly. Note that `number_of_wins` counts the number of adversary wins for a given node, so when back-propagating, `1-score` is added for nodes with the same parity as the simulation node, otherwise `score` is added. In this way, during the selection process, the win rates of child nodes will be an estimate of how many times the parent node will win if it selects the corresponding move, which is what we expect and try to maximize.

```
backpropagate(node, score):
    cur = node
    while cur:
        cur.number_of_visits++
        if cur.max_node == node.max_node:
            cur.number_of_wins += 1-score
        else:
            cur.number_of_wins += score
        cur = cur.parent
```

2.2.6 Initialization

To have a starting point for the MCTS loop of selection, expansion, simulation, and back-propagation, the *mcts_agent* will first initialize the root node and run a simulation on it. It will initialize the root node with the current state of the game board, and set `possible_moves` to all the possible next moves we can select. It will set `number_of_wins` to 1 if the opponent won the simulation, 0 if we won the simulation, and 0.5 if there was a tie. It will also increment `number_of_visits` to 1.

2.2.7 Picking a final move

The *mcts_agent* will keep exploring, expanding, and updating the game tree until near the time limit. Once near the time limit, it will exit the MCTS loop, and select a final move to return. It does this by searching through the children of the root node. If any of the children cause a win, it will immediately return that move. If any of the children cause a loss, it will avoid playing that move completely (unless there is no other option). For the rest of the children, it will select the child that has been visited the most, a term referred to as the most **robust child** [1], and return its corresponding move.

```
final_move(root):
    for child in root.children:
```

```
            if child is fully expanded and causes a win:
                return child.parent_action
            else if child is fully expanded and causes a loss:
                continue
            else:
                count = child.number_of_visits
                if count is the highest seen so far:
                    best_move = child.parent_action
        if no best_move:
            return random_move
        else:
            return best_move
```

2.2.8 Results

Putting all the previous steps together, the *mcts_agent* works as follows.

```
mcts_agent:
    root = initialize
    while time left:
        node_to_expand = select(root)
        new_child = expand(node_to_expand)
        score = simulate(new_child)
        backpropagate(new_child, score)
    return final_move(root)
```

The most favourable trait about the *mcts_agent* is that it has the flexibility to terminate at any given time, and return the best move based on current estimates. By setting the timeout for the MCTS loop to 0.05 seconds before the time limit, giving the `final_move` function 0.05 seconds to return, we found that the *mcts_agent* never exceeds the time limit to select a move. This is a strong improvement from the previous *heuristic_agent*. In addition, we found that the *mcts_agent* achieves a 100% win ratio against a *random_agent* meaning that it is still able to select promising moves under the time constraint. However, when testing its performance against the *heuristic_agent*, we found that the *heuristic_agent* still performs much better, achieving a win rate of 70% across 100 simulations. We hypothesized that this is because the *heuristic_agent* encompasses domain-specific knowledge and strategies whereas the *mcts_agent* does not. As a result, we searched for a way to combine MCTS with domain-specific knowledge, which is the core idea behind *popcoin*.

3. *Popcoin*

Popcoin combines the anytime nature of the *mcts_agent* with the domain-specific knowledge of the *heuristic_agent*. It uses the same MCTS approach as the *mcts_agent* but with some minor adjustments. In particular, it biases the selection process by adding heuristics to the evaluation function, and it biases the final move selection by taking into account the average number of visits for child nodes. It also attempts to share information between turns to avoid re-initializing the game tree from scratch.

3.1 Adding Heuristics to the Evaluation Function

To incorporate domain-specific knowledge, *popcoin* makes use of the `number_of_adv_moves` and `distance_to_adv` heuristics in its evaluation function. Since a nodes win ratio is is always between 0 and 1, it normalizes `number_of_adv_moves` and `distance_to_adv` to allow for easy comparison. It does this by dividing `number_of_adv_moves` by the maximum number moves an adversary can take given the `max_step`, and dividing `distance_to_adv` by the farthest away an adversary could be given the board size. To allow for this, we added the following two attributes to each node:

- `normalized_number_of_adv_moves`: `number_of_adv_moves` / `max_number_of_adv_moves`
- `normalized_distance_to_adv`: `distance_to_adv` / `max_distance_to_adv`

Now, in the evaluation function, instead of returning `exploration + exploitation` as before, *popcoin* will split up the `exploration` value into a weighted combination of the input nodes win ratio, its `normalized_number_of_adv_moves` score, and its `normalized_distance_to_adv` score. This is done to further bias the selection process to select nodes with a favourable win ratio, as well as a good heuristic score. When testing out this feature, we found that *popcoin* out-performed the previous *mcts_agent*, but still was losing the majority of games against the *heuristic_agent*. As a result, we kept looking for ways to incorporate domain-specific knowledge into *popcoin*.

3.2 Adding Heuristics to Picking a Final Move

After some testing, we noticed that *popcoin* was performing significantly worse on larger board sizes. This is due to the large branching factor of the game which causes the average `number_of_visits` for the children of the root to be very low under the time constraints. On a 10×10, 11×11 or 12×12 board, we noticed the average `number_of_visits` for children of the root was routinely 1. We realized that just because a node had a perfect win ratio after one simulation, did not necessarily mean that it was a good move to take. As a result, we decided to bias the final move selection with the heuristic `average_visits` which is the average number of visits of the roots children. Now, the lower `average_visits` is, the lower *popcoin* will value the win ratio over the heuristic value. The higher `average_visits` is, the higher it will value the win ratio over the heuristic value. After implementing this, we found that *popcoin* performs much better on larger board sizes, but still loses to the *heuristic_agent* 50% of the time.

3.3 Achieving a Good Exploration to Exploitation Ratio

After analyzing `average_visits` for different board sizes, we found that the optimal exploration factor ($\sqrt{2}$ in the previous evaluation function) to achieve balance between exploration and exploitation, varied by the branching factor. We came to the conclusion that the larger the branching factor, the more *popcoin* should value exploration, so it can improve win ratio estimates for more possible moves. Thus, instead of an exploration factor of $\sqrt{2}$ for all board sizes, we implemented *popcoin* to use an exploration factor `c`, which was dependent on the board size. After this, we saw an improvement for smaller board sizes, but we struggled to find a good value for larger board sizes.

After some more digging, we realized we had overlooked the fact that the number of possible moves generally decreases throughout the game. As a result, for large board sizes, we decided to make `c` depend on the number of possible moves instead. The idea behind this is that as the number of moves decreases throughout the game, so should the value of `c` to maintain a good exploration to exploitation ratio.

3.4 Sharing information between turns

To avoid having to recompute the entire game tree from scratch at the beginning of each turn, *popcoin* runs a quick search to check if the current game state has already been expanded. It does this in two steps. The first step is once a final move has been selected, the root of the tree is propagated down to the corresponding child node. This node corresponds to the state of the game board once the selected move has been played, and it will be the opponents turn. Then, once the opponent has selected a move, *popcoin* will search through the roots children and check if the opponent took a move that it has already expanded. If yes, it will set the root to that child and start the MCTS loop. If no, it will have no choice but to re-initialize the game tree from scratch. When testing out this feature we found that *popcoin* is sometimes able to avoid re-initialization, and when it does, it picks a strong next move. This is because it starts the MCTS loop with a better estimate of the current game state and its children, rather than if it had re-initialized. However, due to the large branching factor of the game, *popcoin* is often forced to re-initialize, especially for large board sizes, but it is a helpful feature nonetheless.

3.5 Results

Once we had incorporated all the above features into *popcoin*, all that was left to do was test against previous iterations, and tweak the heuristic weights. After many simulations, we settled on our final version of *popcoin*. It beats the *mcts_agent* approximately 80% of the time, with a higher win percentage for larger board sizes. It beats the *heuristic_agent* 50% percent of the time, with a higher win percentage for smaller board sizes. It also performs quite well against a human agent, proving to be a real challenge to beat.

4. Future Improvements

We believe the fundamental idea behind *popcoin* is strong. The more simulations it is able to run, the more it values the win ratio. If it does not have time to run many simulations, it will rely more heavily on heuristics to make reasonable moves. That being said, *popcoin* could certainly be improved. Firstly, since we could only run so many simulations, the weights for the heuristics we used were certainly not optimal. If we had more time, we would have liked to implement a genetics algorithm to find optimal weights for these heuristics. This algorithm would run many simulations, pick the best performing agents and pass their heuristic weights onto the *next generation*. This process would repeat until convergence on the optimal weights. Secondly, we would have liked to experiment with changing the scores associated with a win, tie, and loss. Currently, *popcoin* scores a win as a 1, a tie as a 0.5 and a loss as a 0. However, we hypothesize that it may be better to score a game based on the ratio of captured areas. For example, a game where we covered much more area than

our opponent would be scored higher than a game where we only won by a few squares. Finally, we thought of some other heuristics which we could have incorporated into *popcoin*. Heuristics such as staying away from the edges of the board, and not moving to a square with two barriers could have improved performance, and if given more time we would have liked to incorporate them.

References

[1] Guillaume Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 04:343–357, 2008.