VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
SOFTWARE ENGINEERING STUDY PROGRAMME

Real-time game agent using Monte Carlo tree search

Realaus laiko žaidimo agentas naudojant Monte Karlo medžių paiešką

Course work

Author: 4th year 5th group student
Benas Budrys

Supervisor: Assist., Dr. Rokas Astrauskas

Vilnius – 2025

Contents

Introduction

Topic relevance

Among the many disciplines of computer science, artificial intelligence has become a central area of research, focusing on enabling software to make decisions in complex, uncertain environments. Games are especially suited for such experimentation due to their structured rules, well-defined objectives, and diverse challenges. Consequently, game-playing agents have progressed into a critical topic for studying and advancing decision-making strategies.

Monte Carlo tree search (MCTS) is a notable algorithm for game AI that balances exploration and exploitation effectively through random sampling of the search space. The algorithm has proven successful in turn-based games like *Chess* and *Go* [GKS+12; SHS+18], however, adapting it for real-time environments introduces strict time constraints, dynamic gameplay and other challenges.

Research on MCTS is quite extensive and numerous approaches and optimizations have been explored throughout the years [BPW+12], however, real-time applications for MCTS remain less prevalent due to the inherent computational challenges posed by time constraints. Exceptionally MCTS has been successfully applied for playing the real-time game *Ms. Pac-Man* [PWL14] where the authors introduced several innovative adaptations to overcome the challenges of real-time decision-making. These adaptations included a variable-depth tree and tree reuse with a decay factor, allowing the algorithm to retain relevant information while adapting to the game's dynamic state. The agent's effectiveness was demonstrated in several competitions where it achieved top rankings. The application of MCTS to the *Bomberman* game was also explored [KKO+22]. Although not the optimal solution, MCTS proved to be a viable framework for the game's challenges. The authors note challenges specific to *Bomberman* which this paper takes game mechanics from. The agent faced obstacles such as the penalty for death being too high causing it to play too safe and defining a good heuristic function for the state evaluation.

Aims

This paper aims to create an agent for a single-player real-time game that uses Monte Carlo tree search and analyse its performance implications when tuning parameters.

Objectives

To achieve the paper's aims the following objectives were defined:

1. Create a single-player real-time video game according to the defined rules and containing frame rate independent logic.
2. Implement an agent that uses MCTS for decision-making.
3. Create a visualization tool to analyse trees created by the MCTS process.
4. Analyse the algorithm's performance when tuning the simulation update time step and the exploration constant parameters.

1. Game

1.1. Rules

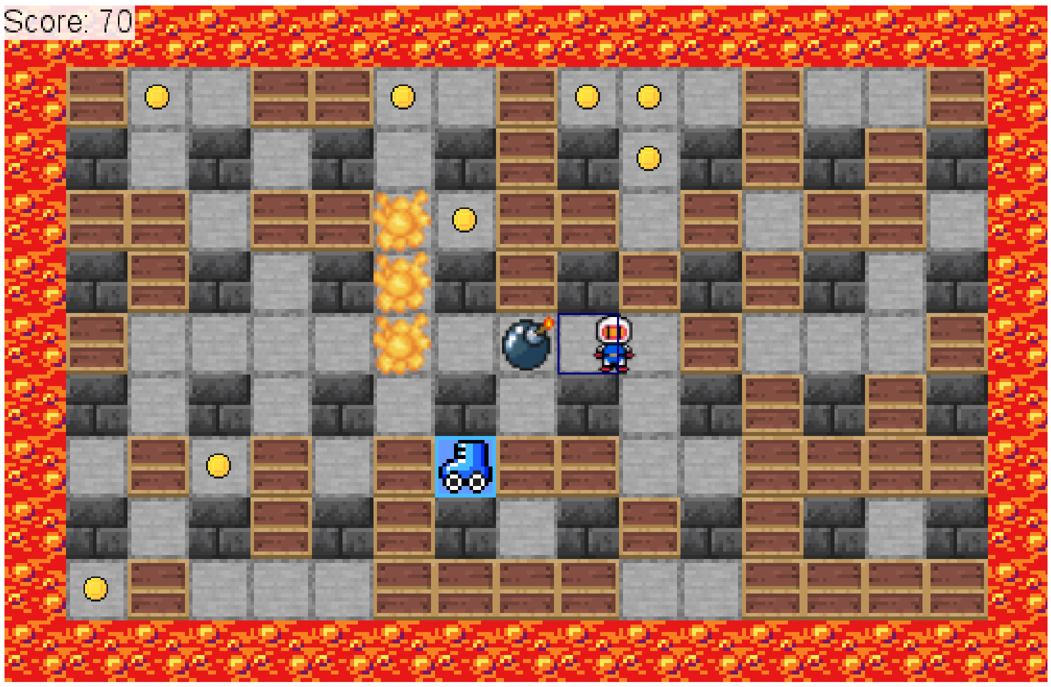


Figure 1. An example screenshot of the game

This section defines the rules and environment of the game that are analysed throughout this paper. Although this is a custom game, a lot of the mechanics are adapted from the popular video game *Bomberman*[1]. Unlike *Chess* or *Go* there is no definitive win state for the player. The goal for the player is to collect points and survive as long as possible without dying.

1.1.1. Tile map

The game is played on a 17x11 grid consisting of various tiles. The border of the grid contains lava tiles which kill the player and end the game if they collide with it forcing them to stay in the play zone. Tiles inside the play zone follow a *Bomberman*-like layout - at intersections of every second row and column a wall tile is placed. The other tiles are randomly generated with weights described in table 1.

Table 1. Tile generation weights.

Box	▤	49.1%
Empty		40%
Coin	●	10%
Fire up	🔥	0.3%
Speed up	🛼	0.3%
Bomb up	💣	0.3%

[1]Originally released for the Nintendo Entertainment System (NES) in 1983

There are 3 power up tiles that grant the player one of the following:

- Fire up – increase bomb explosion radius by one (maximum of 3 tiles).
- Speed up – increase walking speed (maximum of 4 tiles per second).
- Bomb up – increase the number of bombs the player can place at once (maximum of 3).

As time goes by, tiles are shifted from right to left. The left most column of the play zone is discarded, all of the existing columns are shifted left and then a new column on the right side of the play zone is added following the aforementioned tile generation rules. The player is also shifted by one tile. This results in a natural progression forcing the player to move towards the right side of the map, avoid the lava border and encounter new tiles. The tile shift occurs at second intervals defined by the formula $\max(2, 4 \cdot 0.96^s)$ where s is the number of shifts so far.

1.1.2. Player

Similarly to *Bomberman*, the player does not move through the grid one tile at a time but rather continuously through the tiles in small increments defined by the player's speed. Figure 1 shows the player mid-way between two tiles. A dark blue outline is displayed on a tile to help signify where the player is considered to be on the tile map. The player's starting speed is 2 tiles per second.

Apart from moving, the player may choose to place a bomb. Bombs behave in a *Bomberman*-like fashion – after some time a placed bomb explodes in 4 directions. Each of the 4 directions leaves a trail of explosion tiles from the centre up to the bomb radius and will destroy a box tile if encountered. Figure 1 shows a placed down bomb waiting for it to explode and an explosion trail from a different bomb placed before. At the start, the player is allowed to place 1 bomb at a time.

1.1.3. Scoring

During gameplay, the player is awarded points for different occurred events. Figure 1 shows the score in the top left corner. Rewards are given for the following events:

- 40 points for picking up a coin
- 10 points for surviving a tile map shift
- 100 points for picking up a power up

1.2. Implementation

The game is implemented using the C# programming language with the MonoGame framework. MonoGame is a free, open-source .NET framework for creating cross-platform games. This particular framework was selected due to its ease of use, simplicity in contrast to a fully fledged game engine and familiarity with the technology. The framework takes care of features such as the game loop execution, sprite drawing and user input and is suitable for 2D game development.

1.2.1. Game loop

The game loop is the central component of any video game. Unlike traditional software programs that do not execute anything without user input, a video game is constantly updated through the game loop regardless of user input. A simplified game loop looks like the following:

Algorithm 1. Basic game loop

```
1: while user does not exit do
2:     update                    ▷ Read user input, update logic, AI, physics
3:     draw
4: end while
```

Usually, there is no need to iterate this loop as fast as possible and the loop is iterated at some fixed time step. One of the common values for this is to iterate the game loop 60 times per second, leaving around 16ms for a single loop iteration. Under sufficient resources, the update and draw functions may complete before elapsing the full 16ms, so the game loop should just wait until the full 16ms are elapsed and then continue with the next iteration. The time between the updates is commonly referred to as the *delta time* (Δt). In the discussed scenario the Δt is equal to 16ms, however, this might not always be the case, since the game might be coping with the loop iteration that exceeds the given 16ms time step, so (in this paper's implementation) the time step attempts to be fixed but can vary. In general, behaviour under different time stepping techniques (fixed or variable) is subject to the implementation of the game loop and its advantages and disadvantages are out of the scope of this paper.

MonoGame exposes the Δt in its update function to allow the developer to make the game logic dependent on real time and not the frame rate (this is usually referred to as being frame rate independent). An example of this could be a moving bullet: if 100 real time milliseconds passed, the bullet moves the same distance regardless of whether 3 or 6 game loop iterations were completed. This is further discussed in section 1.2.2.

The bombs in the game also depend on real time. Each bomb accumulates the time it exists. On each update call, the accumulated time is incremented by Δt and if it exceeds the detonation time, bomb explosion logic is carried out.

1.2.2. Physics

The player moves linearly and they can choose a moving direction represented by a 2D unit vector (or $\vec{0}$ if the player wants to stand). The velocity is then calculated as the following:

$$\vec{v} = \vec{dir} \cdot speed \cdot \Delta t$$

The $\vec{v}$ represents the change in the player's position for the current update. $|\vec{v}|$ is proportional to the time since the last update (Δt) allowing the movement to be frame rate independent. The $speed$ scalar is expressed as tiles per second. The player's new position vector is calculated by adding the velocity vector to the current position vector

$$\vec{p} = \vec{p_0} + \vec{v}$$

Before the velocity vector $\vec{v}$ is added to the current position vector $\vec{p_0}$, the magnitude $|\vec{v}|$ is decreased if $\vec{v}$ intersects with any solid tiles (wall, box, bomb). This is achieved by checking the tiles in front of the player's movement direction one tile at a time until a solid tile or the length of $\vec{v}$ is reached. As an alternative it would be possible to just check if the player's end position $(\vec{p_0} + \vec{v})$ overlaps with some solid tile, however, if the velocity is big (above one tile) this method would not take into account tiles in between the player's new and old position and would allow the player to teleport through solid tiles, hence the former method of collision checking was chosen.

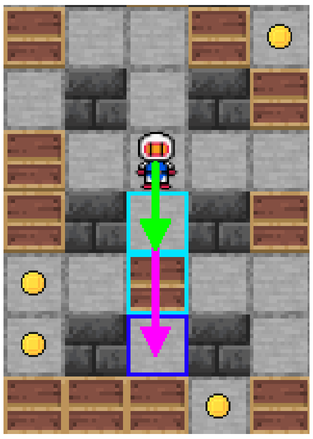


Figure 2. $\vec{v}$ (pink), adjusted $\vec{v}$ (green), evaluated tiles (light blue), not evaluated tiles (dark blue)

Apart from solid tiles, there are enterable tiles such as coins and power ups. If $\vec{v}$ is big (above one tile), then enterable tiles are triggered by the aforementioned collision system if the player crosses them. In the usual case, enterable tiles are triggered if, after moving, the player ends up on the enterable tile. This rounds the player's position to the nearest grid position since the player can be mid-way through tiles.

The player is the size of a single tile, meaning that when a player is at an intersection, they must perfectly align with other tiles if they wish to change their movement axis (horizontal or vertical). To address that, similarly to *Bomberman*, when a player is in such a situation there is a small axis snap threshold applied to the player's position as seen in fig. 3.

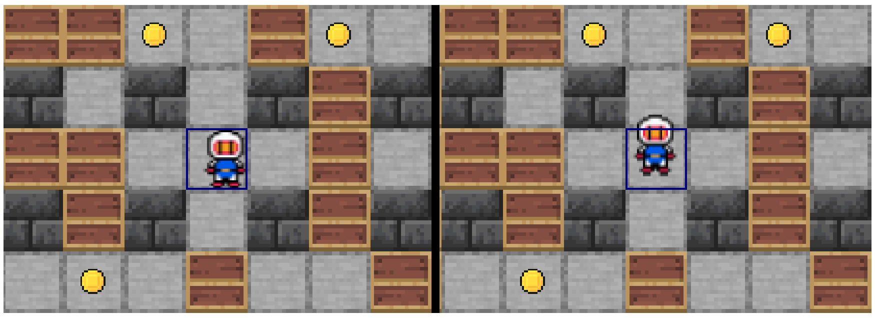


Figure 3. A player not perfectly aligned with the column (left) was snapped to perfectly fit the column (right) after moving up

2. Agent

2.1. MCTS

Monte Carlo tree search (MCTS) is a decision-making algorithm primarily used for planning and game-playing. It explores and analyses the most promising moves by building a search tree composed of game states as the nodes and possible actions as edges representing the transitions from one game state to another. The algorithm uses random simulations to estimate the value of an action and evaluate its desirability. It is designed to be run at any point in time to find the next most promising action to take. A single MCTS run repeats 4 phases of the algorithm until a time constraint is reached.

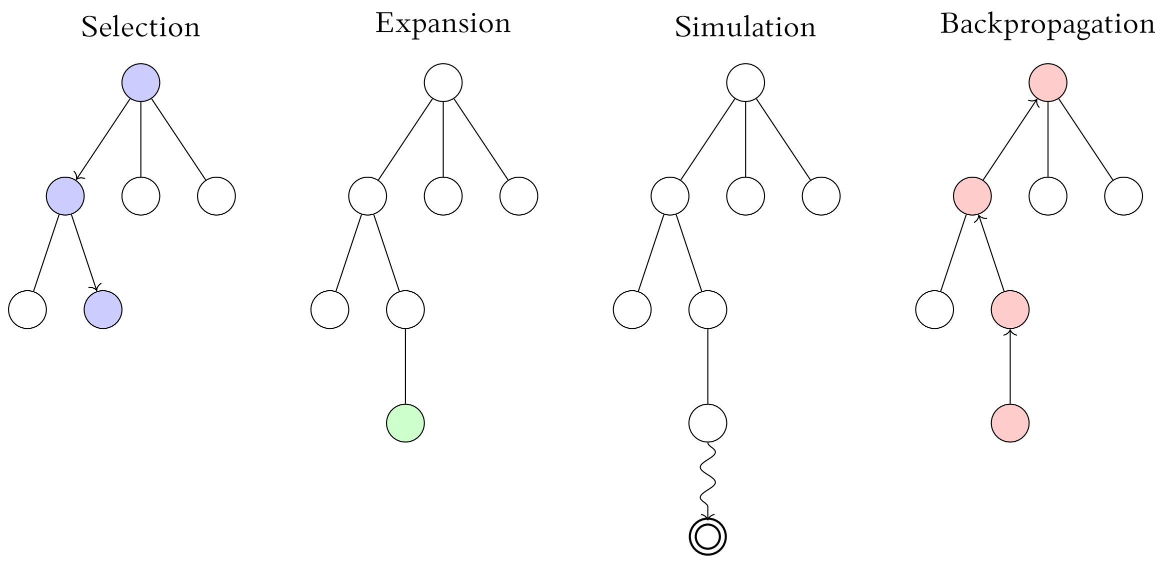


Figure 4. Phases of an MCTS iteration

1. **Selection**. Starting at the root node, the tree is traversed until a leaf node is found. A leaf node is a node that has unexplored actions or is terminal. The nodes are selected using a *selection policy* that typically balances exploitation and exploration.
2. **Expansion**. One or more new child nodes are added to the previously selected leaf node (if the node is not terminal). These new nodes represent unexplored states.
3. **Simulation** (Rollout). From the newly added node state, a simulation is carried out to the end of the game or a fixed depth. The simulation nodes are chosen using a *default policy*, which usually is choosing random actions.
4. **Backpropagation**. The result from the simulation phase is backpropagated up the tree to update statistics that reflect how promising actions are.

After the time constraint is reached and MCTS finishes iterating, the action from the root's children with the most visits is chosen as the final answer.

2.2. Tree

Each node in the MCTS search tree represents a particular game state. For this paper's game, the state consists of the following:

1. Player
    (a) Physics information (position, movement direction, speed)
    (b) Score
    (c) Bomb information (placed bombs, constraints)
    (d) Health (alive or dead)
2. Tile map
    (a) Width
    (b) Height
    (c) The grid position and type of each tile

Creating new nodes in the tree requires duplicating the existing game state object in memory. The game state is also serializable to JSON, further discussed in section 2.7.

In traditional turn-based games, the game advances in discrete steps prompted by the player's actions. A player makes a move, the state transitions into a new state and so on. This naturally translates into the game state tree. A real-time game advances continuously, so a discretization of this continuity is needed to construct the tree. The granularity for this cannot be too high because most state changes may not represent any significant events risking computational resources and, on the other hand, it cannot be too low because significant events might be missed (e.g. missing an intersection). The chosen approach constructs the tree with states where the player moves 1 full tile. More correctly, when transitioning from one state to another in the tree the **time is advanced by the time it takes the player to move 1 full tile**. This is a reasonable approach which aligns the player with the grid. It also implies that the player will be performing actions one tile at a time. If the agent chooses to move left, the player will move left until it moves one full tile and then the next action can be applied. A new action will never be applied when a player is mid-way between two tiles.

Within the specified time interval between two states, the player can perform the following actions, corresponding to edges in the MCTS search tree:

- Stand
- Move (up, down, left, right)
- Place bomb and move (up, down, left, right)

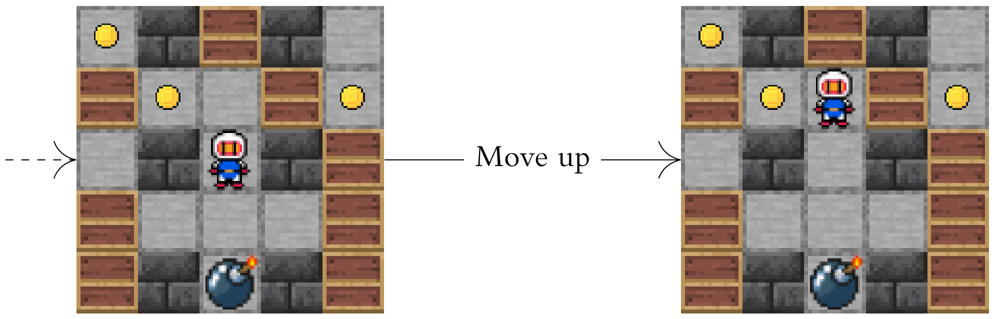


Figure 5. A single MCTS tree transition between two nodes (states)

Since the player's speed is expressed as tiles per second, we can easily calculate the time it takes the player to move one tile.

$$t_{tile} = \frac{1}{speed}$$

To transition between states as shown in fig. 5, we can take the parent state (left) and run its update function with Δt (time step) set to t_{tile} which will advance the time exactly by t_{tile} and the player will be positioned as shown in the child state (right). Despite the player ending up in the correct position, this approach overlooks the fact that the end result may differ from the real game. The update function updates objects (players, bombs) sequentially and using big time steps (such as t_{tile}) may miss important events, as opposed to multiple update calls with smaller time steps. As an extreme example, let's say that a player moves 1 tile per second, a bomb explodes after 1 second, the explosion trail lingers for 1 second and the player is heading in a straight path where 4 tiles neighbour a bomb and the 5th tile is not in any bomb radius. Calling the update function with Δt set to 5 seconds as shown in fig. 6 (left) will result in the player being moved 5 tiles down (the player lives) and bombs exploding, creating an explosion trail and disappearing all in a single update call. In some cases, the sequence in which objects are updated (player then bombs or bombs then player) may lead to different results, although in this case, it does not.

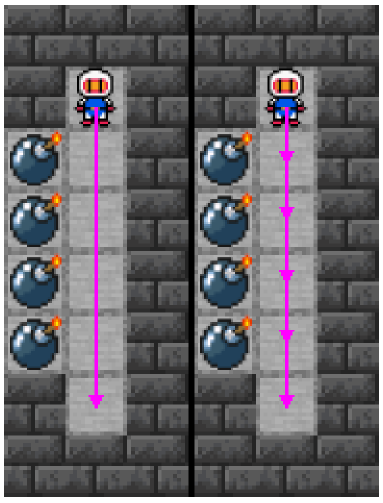


Figure 6. Moving in a single big time step vs moving in multiple smaller time steps

Comparing this to 5 update calls with Δt set to 1 second as shown in fig. 6 (right), the player will die, since at some update call (either call #1 or #2, depending on the object update sequence) the player will be inside the explosion trail. Due to this, it is important to split t_{tile} into smaller steps when transitioning the game state for MCTS. If the desired Δt for an update is $t_{update} = \frac{1}{60}s \approx 16ms$ then the number of update calls is calculated as

$$n = \left\lfloor \frac{t_{tile}}{t_{update}} \right\rfloor$$

Since the t_{tile} might not split exactly into t_{update} we need to do one additional update call with the remaining time

$$t_{remainder} = t_{tile} \mod t_{update}$$

The Δt for an update (t_{update}) in an MCTS simulation is analysed in section 3.

2.3. Selection

The selection phase recursively traverses the tree until it finds a node that either has unexplored actions or is terminal. A common selection policy is the Upper Confidence Bound 1 applied to trees (UCT) formula [KS06]

$$\frac{w}{n} + c\sqrt{\frac{\ln N}{n}}$$

Where w is the number of wins of the considered node, n is the number of visits of the considered node, N is the number of visits of the considered node's parent and c is the exploration parameter, either $\sqrt{2}$ or chosen empirically. The left term of the sum promotes exploitation while the right term promotes exploration. This formula is used as the basis for this paper's selection policy as well. It is important to note that the left term of this formula is the average win rate which is in the range $[0; 1]$. This assumes that the player either wins or loses and it is not directly suitable for a scenario where a win state does not exist. In a game with a scoring system instead of a win state, it is possible to normalize the simulation reward into the range $[0; 1]$, if a maximum achievable reward exists. This can be done by either dividing the reward by the maximum or multiplying c by the maximum. This paper's game does not have a maximum achievable reward, however, we can still try to estimate and find it empirically which is explored later in the paper. The final selection policy is the following:

$$\frac{r_{total}}{n} + c \cdot \sqrt{2} \cdot \sqrt{\frac{\ln N}{n}} \tag{1}$$

Where r_{total} is the total reward (more on simulation rewarding in section 2.4) accumulated throughout all of the simulations, c is the multiplier of the exploration constant ($\sqrt{2}$) and other parameters are the same as in UCT. From any node, we can perform selection with the following algorithm:

Algorithm 2. Selection

```
function Select(node)
    if node has unexplored actions then
        return node
    end if
    if node is terminal then
        return node
    end if
    return Select(child of node with highest UCT)
end function
```

2.4. Simulation

The simulation phase is used to estimate a reward for the expanded node. In a basic MCTS implementation, the game is simulated using a random default policy which selects a random possible action uniformly. This paper's MCTS implementation does uniformly select random actions, however, the set of actions to be chosen from is constructed following domain-specific knowledge:

- The player may not try to move into neighbouring solid tiles. This applies to both "Move" and "Place bomb and move" actions. Moving into a solid tile would result in the same result as standing, which should be discouraged unless it's the safest option.
- The player may not try to move into a lava or explosion tile. This prohibits moving directly into an existing explosion tile, not into a potential bomb explosion radius.
- The player should only place bombs when they neighbour at least one box tile. Placing a bomb down and waiting until it explodes is an important event since bombs are necessary for paving the way towards survival, and it takes a lot of time (multiple actions) to regain the ability to place another bomb.

Moreover, in a basic MCTS implementation the game is simulated until termination (win or loss), however, since this paper's game does not have a win state, it is important to introduce a depth limit d_{limit} to the simulation. The simulation ends when the player dies (game is terminated) or the simulation depth d_{limit} is reached. The chosen value for d_{limit} is 40, which, taking into account that the t_{tile} is between $[0.25, 0.5]$ seconds (the player's speed may change due to power ups), corresponds to $[10; 20]$ seconds of simulated gameplay. The simulation applies actions and transitions from state to state as mentioned in section 2.2.

In addition to the game score defined in section 1.1.3, the simulation uses other terms to reward or penalize the player for ending up in a particular state. The simulation reward is composed of the following terms:

- Game score (s) gained during the simulation

$$r_{game} = s_{end} - s_{start}$$

- Column reward. This rewards the player more as they approach the right side of the tile map. Since the tile map is shifting left, staying as right as possible (indicated by the player's position on the x axis $p.x$) for maximum survival is encouraged.

$$r_{column} = 20 \cdot \lfloor p.x \rceil$$

- Survival coefficient in the range of $[0; 1]$ which increases as the player survives for longer. Dying early in the simulation (not reaching the depth limit d_{limit}) should be penalized.

$$c_{survival} = \frac{d_{simulation}}{d_{limit}}$$

The final reward r_{final} is calculated as

$$r_{final} = c_{survival} \cdot (r_{game} + r_{column})$$

and the value r_{final} gets backpropagated in the next phase.

2.5. Expansion & Backpropagation

The expansion phase is not subject to many customizations. If the node is not terminal, an action is randomly selected from the node's unexplored action list. A new child node is created by taking the selected node's game state and transitioning it to the new state with the chosen action as described in section 2.2.

Backpropagation is also a straightforward phase as the only data backpropagated back up the tree is the simulation's reward r_{final} accumulated in r_{total}. The node's visit count n is also incremented.

2.6. Continuously selecting actions

The previous sections covered selecting a single promising action using MCTS. As the game progresses, we need to continuously select actions. In this implementation the runs of MCTS are independent, constructing the tree from scratch. While the game is progressing from the last action another thread can start the MCTS process from the state the player is about to be in.

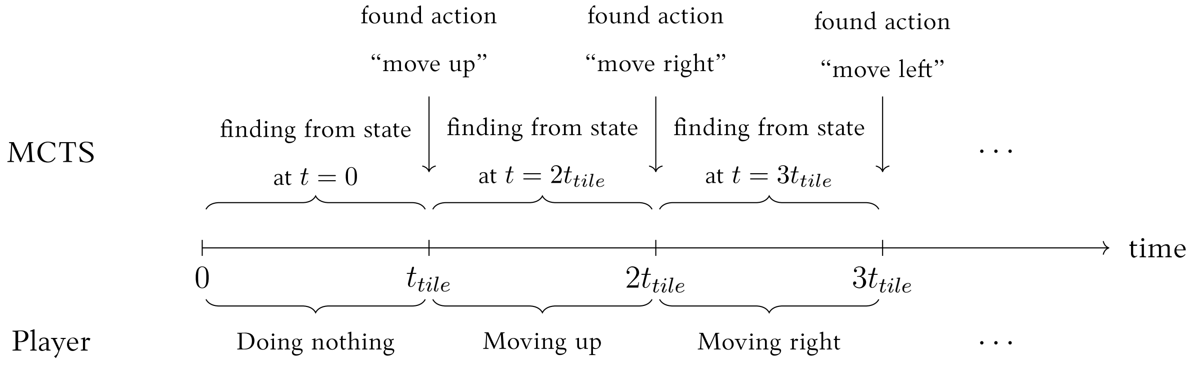
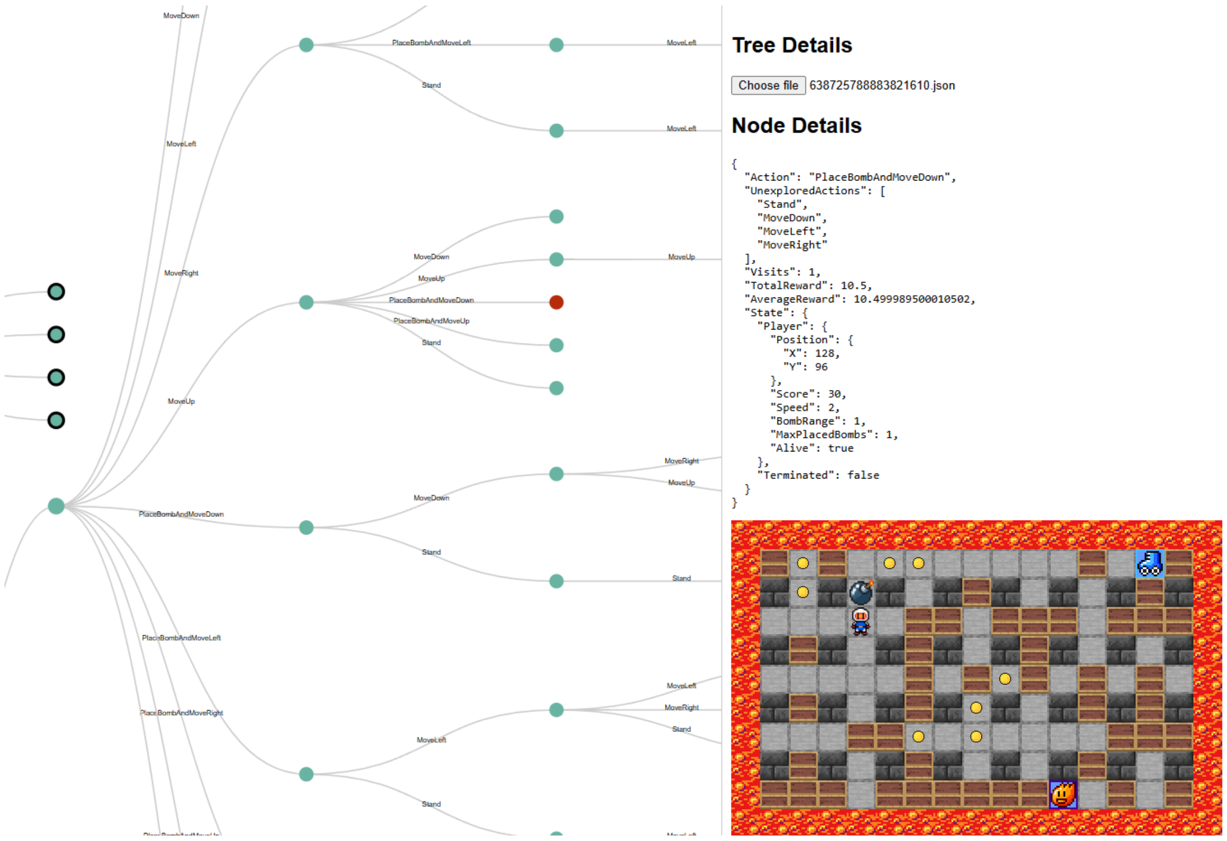


Figure 7. MCTS looking for the next move while the real game is still progressing from the last action. MCTS thread (top), game thread (bottom)

Figure 7 illustrates this behaviour. When the game starts, the player is standing while MCTS searches for the first action to make. When the first action is found, the player starts performing that action and MCTS immediately starts searching for the next action based on the future state (at $t = 2t_{tile}$) and so on. This is achieved by just transitioning the root state (similarly to the expansion phase) with the previously found action. Concurrency must be taken into account when applying the found action from the MCTS thread to the game thread.

2.7. Visualizer

A separate client-side web application was developed to visualize the MCTS trees. The nodes which make up the tree can be serialized to a JSON file. This file can be selected on the web application. The main panel on the page shows the overall tree structure: nodes, actions on edges corresponding to state transitions. It is possible to inspect a node by clicking on it. A side panel will display data about the node such as the number of visits, average reward and a preview of the game at that state.



Tree Details

638725788883821610.json

Node Details

```
{
  "Action": "PlaceBombAndMoveDown",
  "UnexploredActions": [
    "Stand",
    "MoveDown",
    "MoveLeft",
    "MoveRight"
  ],
  "Visits": 1,
  "TotalReward": 10.5,
  "AverageReward": 10.499989500010502,
  "State": {
    "Player": {
      "Position": {
        "X": 128,
        "Y": 96
      },
      "Score": 30,
      "Speed": 2,
      "BombRange": 1,
      "MaxPlacedBombs": 1,
      "Alive": true
    },
    "Terminated": false
  }
}
```

Figure 8. Tree view (left), node information (right)

The application is implemented using vanilla JavaScript with HTML/CSS. The D3.js library is used for the tree visualization. It is possible to pass the `--export` flag to the game's executable to export all of the MCTS trees to JSON during the execution.

3. Analysis

3.1. Methodology

The analysis aimed to evaluate the performance of the real-time game playing MCTS agent by tuning two parameters: the exploration constant multiplier c in eq. (1) and the Δt for a simulation update (t_{update} mentioned in section 2.2). All of the experiments were performed in a deterministic game environment to ensure comparable results. A pseudo-random number generator with a fixed seed was used for the tile map generation and each experiment consisted of running the game 10 times with a specific parameter value while keeping other parameters fixed.

- The parameter c in the selection policy based on UCT directly influences the balance between exploration and exploitation. A range of c values was chosen for experimentation: $\{1, 10, 15, 20, 25, 50, 100, 200, 300, 400\}$. For each value, the game was run 10 times, and the average game score was calculated. This process provided insight into how different exploration constants impact the agent's decision-making and game performance.
- The simulation update Δt controls the granularity of the game's logical updates during simulation. Larger Δt values result in coarser updates, potentially reducing computational demand. A range of Δt values was chosen for experimentation: $\{\frac{1}{60}, \frac{1}{50}, \frac{1}{40}, \frac{1}{30}, \frac{1}{20}, \frac{1}{10}\}$ (in seconds). For each Δt value, the following metrics were recorded:
  - Average game score across 10 runs.
  - Average MCTS iteration count per game, taking into account only the lower three quartiles (the bottom 75% of iteration counts) to mitigate the effect of exponential increases near game-ending scenarios where the player does not have a lot of actions to explore.

The experiment was conducted on a machine equipped with a 12th Gen Intel(R) Core(TM) i7-12700 CPU, ensuring sufficient computation power for consistent results.

3.2. Results

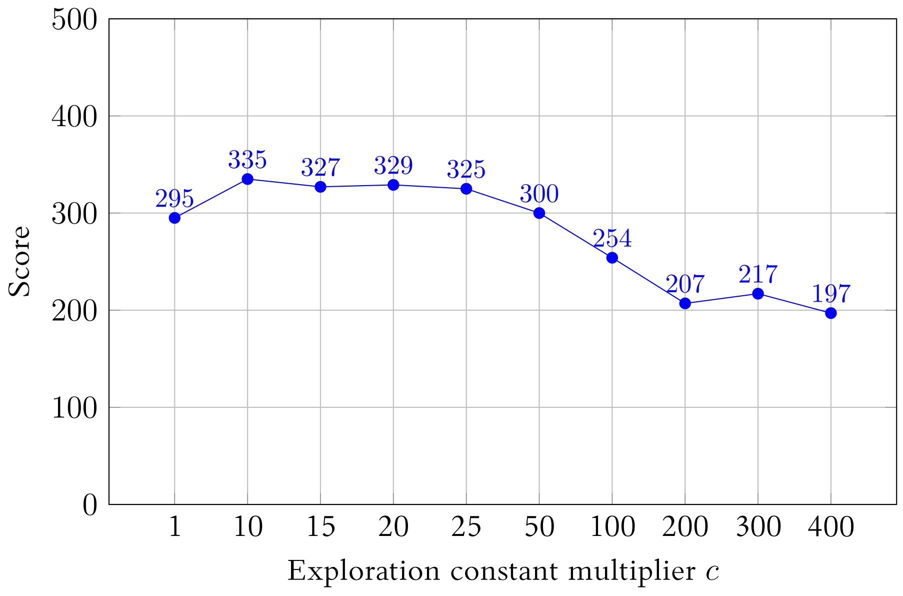


Figure 9. c tuning results

The average game scores for the tested exploration constant multiplier c showed minimal variation. This suggests that the agent's gameplay strategy remains largely unaffected by changes in c. The observation indicates that the agent plays too conservatively, likely avoiding placing bombs. It was observed that the player successfully evades their own bombs, however, struggles to pave a way forward to survive longer. This behaviour may stem from the current reward system, which may not adequately incentivize clearing pathways and moving towards the right side of the screen. A revision of the reward system could be necessary to encourage more balanced exploration and exploitation.

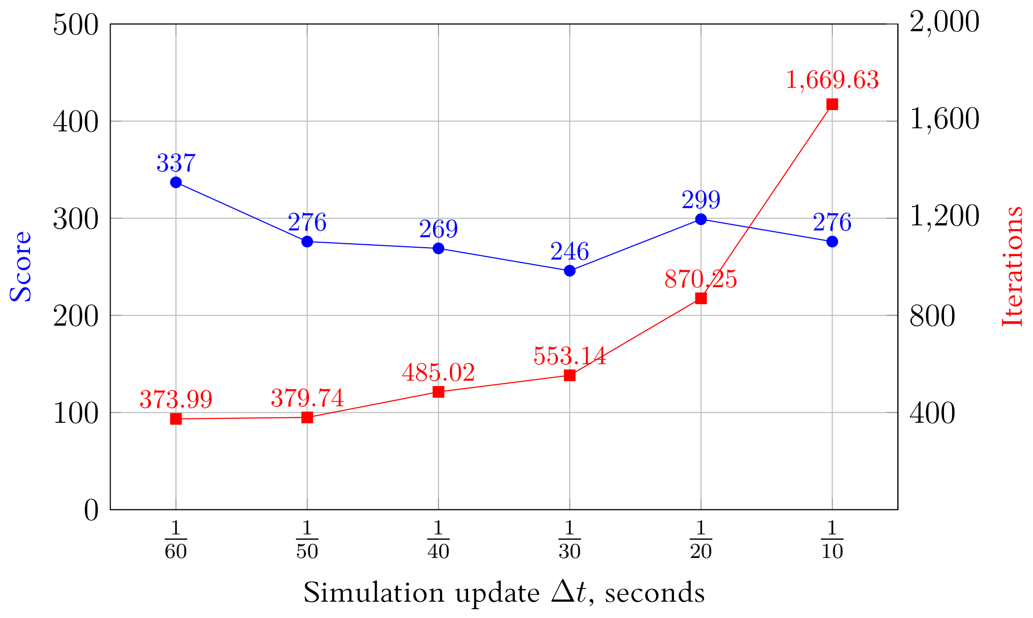


Figure 10. Δt tuning results

The results for Δt revealed two findings:

1. Similar to c, the average game scores remained relatively consistent across different Δt values. This further proves the conservativeness of the agent and the potential need for reward restructuring.
2. The average MCTS iteration count increased as Δt values grew larger. This aligns with the expectation that coarser time steps reduce the computational complexity of simulating the game's logic, allowing more iterations within the same time budget. Analysing the Δt did not lead to any observations where the simulation state fell out of sync with the real game so much that it led to the player dying.

Results and conclusions

In this paper, the following results were achieved:

1. Developed a *Bomberman*-like single-player real-time game. Implemented the game logic in a frame rate independent manner.
2. Implemented a game playing agent using MCTS. Several customizations and enhancements were made to the base implementation making the algorithm more suitable for the real-time domain.
3. Created a visualization tool to analyse and debug the MCTS trees generated by the agent.
4. Analysed properties impacting the algorithm's performance and computational complexity.

The work led to the following conclusions and observations:

- The implemented agent fails to play the game successfully in comparison to a human player. The agent lacks the ability to move towards the right side and survive longer, although it is successful in evading bombs.
- The agent's conservative strategy suggests that gameplay behaviours are primarily dictated by the reward structure rather than fine-tuning of the MCTS parameters. Adjusting the reward system to better balance risk and reward may improve the agent's gameplay performance.
- Larger simulation time step values can enhance computational efficiency without adversely impacting gameplay, providing a potential avenue for optimizing agent performance in real-time scenarios.

References

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, et al. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012, volume 4, number 1, pp. 1–43. Available from: https://doi.org/10.1109/TCIAIG.2012.2186810.

[GKS+12] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, O. Teytaud. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*. 2012, volume 55, pp. 106–113. Available from: https://doi.org/10.1145/2093548.2093574.

[KKO+22] D. Kowalczyk, J. Kowalski, H. Obrzut, M. Maras, S. Kosakowski, R. Miernik. Developing a Successful Bomberman Agent. In: *Proceedings of the 14th International Conference on Agents and Artificial Intelligence*. SCITEPRESS - Science and Technology Publications, 2022. Available from: https://doi.org/10.5220/0010840200003116.

[KS06] L. Kocsis, C. Szepesvári. Bandit Based Monte-Carlo Planning. In: 2006, volume 2006, pp. 282–293. isbn 978-3-540-45375-8. Available from: https://doi.org/10.1007/11871842_29.

[PWL14] T. Pepels, M. H. M. Winands, M. Lanctot. Real-Time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*. 2014, volume 6, number 3, pp. 245–257. Available from: https://doi.org/10.1109/TCIAIG.2013.2291577.

[SHS+18] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. 2018, volume 362, number 6419, pp. 1140–1144. Available from: https://doi.org/10.1126/science.aar6404.

Appendixes

Appendix 1

Source code

The game and the visualizer developed in this paper are publicly available on GitHub: `https://github.com/BenasB/vu-kursinis`

The visualizer tool is available online at `https://benasb.github.io/vu-kursinis/`