University of Essex School of Computer Science and Electronic Engineering

Intelligent game playing with AI

Andrei Tiberiu Alexandru
BSc Computer Science
1502053

Supervisor: Dr. Luca Citi
Assessor: Dr. Adrian Clark

Acknowledgements

Many thanks to Dr. Luca Citi, for guidance throughout the exploration of a large research space and for many fruitful discussions. This learning experience would have been a lot less enjoyable without his input.

A big thank you to Radu Adobroaie and Liam Blenkinsop for proof-reading drafts of the paper and offering constructive criticism.

Finally, I am grateful for my family's support throughout a year-long endeavour and for their unwavering belief that I would overcome all obstacles. Their encouraging words have been invaluable.

Abstract

Video-games contain a simulated reality not unlike ours, where the developer has control over constraints and parameters which are inaccessible in the outside world. They are generally structured around tasks that are challenging for humans, and in some cases prove to be more difficult than other activities to which our species is adapted. This makes games ideal for artificial intelligence research and benchmarking. In this project, we build an intelligent agent and test it using the General Video Game AI platform. The agent is a combination of two algorithms with an outstanding track record in decision-making and pathfinding: Monte Carlo Tree Search (MCTS) and A* search. MCTS was used by DeepMind in AlphaGo, the first intelligent agent to beat the human champion in the game of Go. A* is used widely in pathfinding problems in video-games, navigation and parsing grammars in Natural Language Processing. The hybrid agent is benchmarked against a standard implementation of MCTS.

Table of Contents

1 Introduction

Video games have been widely appraised in the research literature as one of the best tests for artificial intelligence. This is due to the various challenging aspects encountered by the player, including dealing with an uncertain environment, planning, making a decision with limited information and within a relatively short time, learning which aspects and outcomes of the game are desirable and which undesirable, as well as generality of objectives. Any difficulties an AI player (also called an agent) may have in its attempts to win one game is compounded by the fact that the next game to play may be different in all aspects. It is in this sense that training an agent to play a multitude of games becomes useful in researching artificial general intelligence (AGI). AGI can be defined as the ability of an agent to do the following [1]:

- Reason under uncertainty;
- Represent and make use of knowledge, and specifically language;
- Learn to adapt to new circumstances.

In theory, an AGI aligned with human values could be tremendously beneficial to humanity by creating new technology (a common example is molecular nanotechnology), conducting automated research (perhaps leading to the development of interstellar travel), and facilitating global policy and coordination problems (e.g. climate change)[2]. There have been significant efforts in the AI safety and existential risk domain made in order to ensure that the discovery of AGI does not precede the adoption of a governance model optimised for the scenario[3].

Whereas the AI safety research sphere approaches AGI from a social and governance perspective, many of the advancements in AI capability come from applications to limited domains with measurable goals and instant feedback. For example, the team at DeepMind recently released an agent titled AlphaStar, competing in the real-time strategy game Starcraft II[4]. It was able to defeat the game's built-in "elite"-difficulty AI, as well as two of the top ranked players worldwide. In contrast with AlphaStar, which was built specifically to compete in Starcraft II games, some agents are built to compete in games that have significantly different features which encourage some play styles and discourage others.

General Video Game AI (GVGAI) was introduced in 2014 as a framework and a competition for general video-game playing[5]. The current version contains ~110 games, some of which were originally created for the Atari 2600 console. All games are two-dimensional. Some games are deterministic, which means that they can be modelled as a sequence of causes and effects; others factor in randomness, usually manifested as changes in the environment. This has an impact on performance since some algorithms fare well in stochastic games by using a technique called statistical sampling. Others attempt to search for the optimal choice exhaustively, which may not be possible. Some games contain non-playable characters (NPCs)—for example Zelda and Pac-Man—, whereas others contain only the player avatar and environment features such as resources or portals. A full classification is offered in [5], taking into account NPC type, whether the game can be won or is simply score-based, how many actions are available to the player etc. The competition has several tracks with different rules. In the 1-player planning track explored in this paper, a time limit of 40ms per turn is imposed, during which the agent must return a valid action. If the agent takes too long, it is disqualified.

One type of algorithm that has occurred many times in the submissions to GVGAI is the Monte Carlo Tree Search (MCTS). It has ranked highly in most of its entries, and an MCTS-based agent titled YOLOBOT won the 2015 competition by overall score. In the following sections, the performance of MCTS is explored through the creation of a hybrid agent. The paper is structured as a review of the relevant research (section 2), followed by a discussion of the design and implementation of the hybrid agent (sections 3 and 4), as well as the test results and the conclusions drawn from them (sections 4 and 6). Section 5 is dedicated to the methodology used while planning and managing this project.

2 Literature review

The literature review is organised as follows: section 2.1 focuses on the Monte Carlo Tree Search algorithm, including a description, algorithm limitations and the motivation for an improved approach. Section 2.2 introduces the idea of a hybrid, an agent that uses a combination of algorithms to achieve better performance. In section 2.3, the discussion focuses on hyper-agents, which choose from a portfolio of algorithms the approach best suited to the problem at hand.

2.1 Monte Carlo Tree Search

In its current form, the Monte Carlo tree search is the result of two papers published in 2006. Coulom[6] is the first to formalise the approach combining tree search with the Monte Carlo sampling method. In [7], Kocsis and Szepesvári apply bandit methods to tree search by implementing the UCT algorithm (Upper Confidence Bounds for Trees). In the years since, MCTS has been applied in many settings, including the game Hex, Chess and, more importantly, Go[8]. Its success can be attributed to its versatility: MCTS does not need domain knowledge to succeed (it is aheuristic). As the search is run, a search tree is generated asymmetrically, favouring more promising outcomes. Finally, given a limited computational budget, MCTS always returns the best result thus far (that is, one does not need to wait for the search to “finish” to retrieve its output)[9].

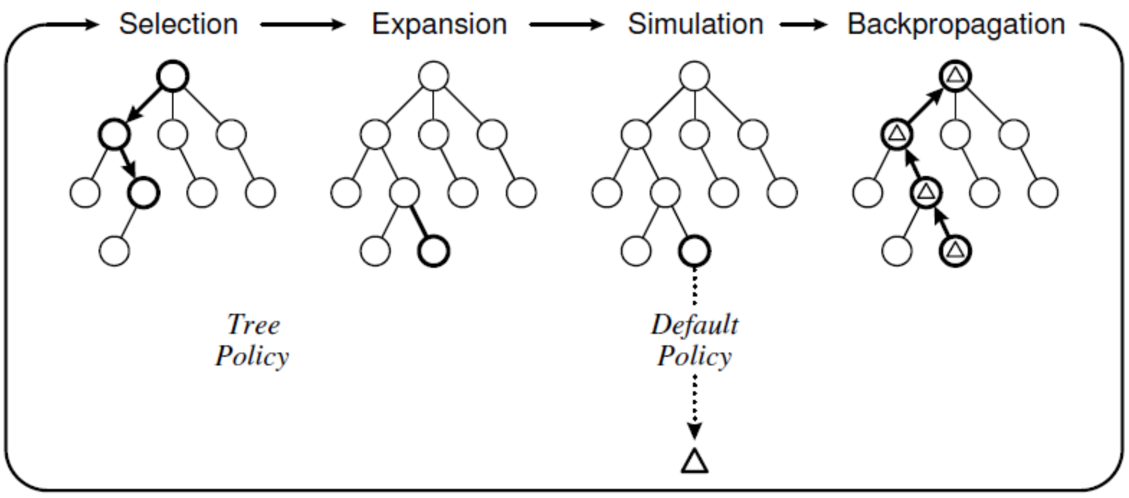
2.1.1 Description

The Monte Carlo Tree Search (MCTS) algorithm builds an asymmetrical search tree that approximates the value of moves in the problem space[9]. To do so, it relies on statistical sampling of possible states until a terminal state or a computational limit is reached. Trees are composed of nodes recording the current problem state, the game-theoretical value of the current state (i.e. how desirable the state is relative to other states), the number of times the node was visited during the search, and other implementation-specific data (see section 2.1.3). The transition function of the tree comprises the actions or moves that are available from a given node. In this paper, these terms are used interchangeably—a “move” in the tree does not equate to moving the player avatar in a game. When one of these actions is applied to the node, the result is a new node called a “child”. MCTS algorithms have 4 distinct steps:

- **Selection**: from the root node, descend the tree until you encounter the expandable node which maximises a utility function. “Expandable” means that the node is not fully-expanded: there are available moves, or actions, which when applied to the node would generate a child node not previously encountered. Selection is applied recursively.
- **Expansion**: apply one or more of the available actions and add the resulting child node(s) to the tree. Together, the selection and expansion step are called the tree policy.
- **Simulation**: starting from the new child node, simulate possible game states by applying a sequence of actions until a terminal state is reached. At its most basic level, the strategy applied during simulation—called a default policy—is a sequence

of random actions which explore the search space. Most often, a terminal state is not reached due to the size of the search space, so a depth constraint is introduced. This roll-out depth is typically determined experimentally and has implications on the algorithm's ability to simulate future states (see 2.1.2). In [10], the authors distinguish between light and heavy playouts. A light playout means that at each step of the simulation, a random action is taken from the set of available actions. On the other hand, a heavy playout consists of the use of a heuristic to choose the action.

- **Backpropagation**: evaluate the final state reached during simulation according to the utility function and back-up that value until the root of the tree is reached. This value, Δ, is added to the previous value of each parent node; at the same time, their visit counts are increased. Backpropagation essentially ensures that the path leading to a promising terminal state is explored more heavily than other, lower-value paths. In the absence of updates for nodes, MCTS becomes a random walk.



(Figure 1: steps of a Monte Carlo Tree Search[9])

The steps are run in a loop until some computational budget is exhausted, such as number of iterations or CPU time. The budget varies according to the purpose of the implementation and can be used to control the accuracy of MCTS. Once the search is completed, the action recommended is the one leading to the "best" child. There are two main criteria used to determine the best child[9]. The algorithm may return the action corresponding to the child with the highest value, or to the child with the most visits. While there is some overlap between the measures, the highest reward and the highest number of visits do not always occur in the same node.

The utility function used to determine the most "urgent" node to expand during the selection step has a large impact on the performance of the MCTS. Repeatedly choosing the same node leads to a depth-first style of expansion, where actions lower in the tree are explored at the expense of nodes closer to the root. Another solution may be to apply each action uniformly; the resulting algorithm, called flat Monte Carlo, works well in some cases, but is suboptimal in scenarios with a large search space (akin to a breadth-first search). These two variations are at the opposite ends of an exploration-exploitation trade-off. The multi-armed bandit problem describes how a rational agent may attempt to maximise their utility in situations where the best action is not known in advance, may change with time or even decrease the more often it is made[11].The process is formalised as a K-armed bandit, which is a slot-machine with K arms that have different pay-outs, all unknown to the player. In this case, the player's goal is to minimise their regret, which is the difference between the best possible reward at each step and the reward received at that step (equivalent to what is lost due to suboptimal playing). The main result of [11] is the Upper Confidence Bound (UCB), a policy that ensures that the expected regret grows only logarithmically, even without knowing the reward distribution. The agent should play machine j that optimises:

$$UCB1 = \overline{x_j} + \sqrt{\frac{2 \ln n}{n_j}}$$

where $\overline{x_j}$ is the average reward from machine j, n is the overall number of plays so far and n_j is the amount of times machine j was played. The first term of the formula corresponds to the exploitation phase; the second, to the exploration phase. When machine j has been played many times relative to the total plays, the second term decreases. This results in UCB1 being relatively small, even if the reward is large, thus encouraging exploration of other options. In some cases, the second term is multiplied by an exploration constant K in order to bias the search. In its application to MCTS, UCB1 translates to picking the child node with the highest reward that has been visited the least amount of times. By convention,

$$UCB1_{unvisited\ child} = +\infty$$

2.1.2 Limitations

Although it's a versatile algorithm, the win rate for MCTS in GVGAI games is not very high. By one measure, it is only 31%[12]. Additionally, MCTS-based agents do not win certain games at all (e.g. Escape and Lemmings). Where there are many available actions, there may not be enough time (or iterations) to try each of them, resulting in random behaviour. The rollout depth introduced in the simulation step is intended as an upper bound for the complexity of the playouts; its drawback is that the algorithm cannot see past the horizon imposed by that depth. If, for instance, a search is run with a rollout depth equal to 10 and a state reached at that depth has a promising value, the tree will follow that path. If a losing terminal state lies just past the deepest explored state, the agent may not be able to redirect in time (e.g. it may venture too close to a non-friendly NPC).

Another drawback is speed. Since the games in the GVGAI framework are all real-time, they have a time limit per move of 40ms, and an agent initialisation time of 1s. This severely limits the number of iterations of the search—and in turn, the accuracy of the tree—as well as the depth of the rollouts achievable. However, such a time limit is realistic if the aim of writing an agent is to play the game as a human does.

2.1.3 Enhancements

Monte Carlo tree search is aheuristic: it does not need domain-specific knowledge to work. This means that it is widely applicable, as well as open for improvement[9]. Although domain-specific knowledge is not necessary, it may provide useful gains in performance. In [10], Drake and Uurtamo investigate whether it is more advantageous to modify the tree policy or the default policy of an MCTS instance applied to the game Go. Modifications to the tree policy result in what they call "move ordering"—a specific method of selecting child nodes to expand. Adjusting the default policy results in "heavy playouts," in the sense that the states simulated are no longer random, instead being guided by a heuristic. Their finding was that biasing the simulations, rather than the selection, results in more significant performance gains.

Browne et al. [9] provide an overview of the literature on MCTS, including variations and enhancements. Some revolve around changes to the tree policy, either by tuning the formula for UCB, or by coming up with novel ways to explore the search tree (for example by using a fixed value for unvisited nodes and UCB for visited nodes). Others rely on game theory to more accurately value each state (or the reward of each node), since more clarity on these rewards should improve the final outcome, regardless of the underlying mechanism for search. Still others are concerned with improvements to the simulation step, with some variations storing the average reward for an action in a separate table so that actions which proved to be useful in a previous context may be repeated.

An interesting direction of research is the encoding of macro-actions into agent behaviour. In a 2017 paper, Perez-Liebana et al.[13] present the creation of games with real-world physics on the GVGAI framework. Most games that were initially developed for GVGAI rely on grid-physics, which has discrete positions (squares) that the agent can traverse. By contrast, real-world physics has continuous states, friction, inertia and other forces. This transition from discrete to continuous is particularly problematic for an agent that relies on search, since the space of possible states increases by several orders of magnitude. To exemplify agent behaviour in this new environment, the authors select two sample algorithms: MCTS and a rolling horizon evolutionary algorithm (RHEA) as test subjects. To offset the increased branching factor of the search tree, macro-actions are used instead of lower-level actions. This has the effect of increasing the depth of the simulation in MCTS, since the algorithm is now simulating actions such as "move to nearest sprite of type *resource*." Part of the findings was that shorter macro-actions result in better performance. This is consistent with the trade-off that shorter sequences correlate to poor exploration, whereas longer sequences result in inefficient navigation of the map (e.g. frequently overshooting the target). The win rates are encouraging, however the set of games that use continuous physics may not be large enough to demonstrate consistent agent performance in these conditions. Furthermore, macro-actions were not tested in grid-physics games, so an agent that does well in one environment may not necessarily do well in the other (this effect is termed "no free lunch," and is discussed later in this section).

Nevertheless, introducing continuous physics into games is an important step towards developing and testing agents that can solve real-world problems. In a sense, the discrete states are a "crutch," a luxury that a bomb-disposal robot, for example, would not have. If macro-actions are available, it is likely that agents will be able to "think" of the environment in more abstract terms; to ignore the low-level, "move one square to the right"-type moves. In a parallel between ant colonies, chess and human brains, Hofstadter[14] discusses the chunking hypothesis, which states that chess grandmasters retain knowledge of a chess board better than beginners because they cluster pieces together and remember the formation of each cluster, as opposed to each piece[15]. To Hofstadter, chunking is not simply a particularity of human memory, but the idea that by thinking in higher-level concepts an individual may see a different way to achieve their goals, in the same way a chess master quite literally sees the best move (findings from the original study are replicated in [16]). Perhaps this extends to any intelligent agent. It certainly seems intuitive that to achieve high-level goals one needs to have at least intermediate-level macro-actions which feed into low-level, atomic moves. At this stage, the correlation is anecdotal for game-playing; further research is needed.

Referring specifically to general video-game playing, a more recent exploration of the enhancement space showed an increase in win rate from 31% (plain MCTS) to 48.4% (a selection of 8 enhancements for MCTS) over 60 games in the GVGAI framework[12]. It is worthwhile detailing some of these in order to get an idea of the approaches taken when modifying the tree search. The first two ideas, Progressive History (PH) and N-Gram

Selection Technique (NST) both relate to repeating an action that has previously led to a high-reward state. PH introduces this bias during selection, whereas NST biases the play-outs. The implementation for GVGAI stores each action and its average reward alongside the current position of the player avatar (an action that was useful in position X is not necessarily useful in position Y). For example, in the game Frogs, the player avatar is a frog that must cross a busy road, then a river stream in which there are logs floating. It is sometimes possible for the frog to cross the road in one smooth move across, as opposed to dithering up and down the road. This smooth move may be accomplished through a macro-action: a sequence of actions compressed into one action. Under the right conditions, a tree search using PH/NST, can discover this macro-action and cross the road seamlessly. However, it would be undesirable to re-apply the same rule when crossing the river stream, since it's less common that the logs will align.

Another proposed improvement is to retain the tree constructed during a previous iteration of the search so that previously-discarded states are not explored again fruitlessly. This may introduce a memory constraint since the new tree would have to be initialised with a sub-tree instead of a root node. Additionally, in non-deterministic games, it may be that applying action a_i from a state s sometimes results in state s_i and other times in state s_j, where $i \neq j$. To account for this, the tree reuse method decays previous results by a factor γ, indicating that previous knowledge about the game may no longer be accurate.

Safety pre-pruning refers to the elimination, from the outset, of those actions which result in a large number of losses. First, for each action the number of immediate losses is determined out of M generated states. Then, only the actions leading to minimum losses are kept; the others are discarded. In conjunction with pre-pruning, the authors propose initialising the tree breadth-first, by generating the first ply of the tree before running MCTS. This serves to balance situations where there are too many actions available from the root node, which would cause MCTS to behave nearly randomly due to its computational constraint.

Loss avoidance is an adaptation that calibrates how "pessimistic" the search is. Normally, upon encountering a losing terminal state at a certain depth, MCTS would avoid the path leading to it. This sometimes causes the avatar to behave in erratic ways, such as in Frogs, where the frog never ventures out into the road because within the roll-out limit no non-losing state is visible—the frog always gets hit by a car. To circumvent this, Soemers et al. suggest not back-propagating the losing state, but manually expanding its siblings and returning the highest reward amongst them; this ensures that MCTS does not discard an entire path due to one negative outcome lower in the tree. However, this introduces significant computational costs if there are many losing states, which means there is even less time for the search to explore alternatives.

One variation that shows promise in the case of GVGAI is knowledge-based evaluation of game states. The default evaluation function for a state relies on the game score, with special considerations for winning or losing states. More specifically, in a winning state the function adds a large positive number to the game score; in a losing state, a large negative number is added. The problem is that in some games it isn't feasible to reach a state that increases the game score from the initial state, because there are many intermediary steps. In Sokoban, the player avatar must push boxes that are in different positions on the map to a specific storage location; the score increases with each box that reaches the destination but does not change otherwise. The difficulty a standard MCTS player (i.e. Monte Carlo tree search with Upper Confidence Bounds) would have with Sokoban is obvious: there is no way to know which moves are desirable, and it typically does not see far enough into the future to find the "box-at-destination" state. Instead, the state evaluation may rely on the computed distance to sprites (objects) on the map. The closer the agent is to a sprite, the higher the reward. The shortest path to each sprite is computed using A* and the distance is factored into the state evaluation function. The solution proposed in [12] takes into account the existence of different types of sprites (NPCs, resources, portals, “movables”) in GVGAI and weights them accordingly. These weights are initially positive and change as the agent interacts with the sprites. If collision with the sprite causes an increase in score, the weight increases; if the score decreases, the weight is reduced to avoid the sprite (for example a non-friendly NPC the player has to kill, like in the game Zelda).

Another enhancement that proves particularly useful for hybrid agents is deterministic game detection. The premise is that if the agent is aware of whether a game is random or deterministic, it can tailor its strategy to maximise the score. In a random game, it is unlikely that exhaustive search will reach a meaningful outcome, since making a move does not always result in the same state, hugely increasing the search space. For example, in the game Aliens, the agent can only take three actions: left, right and “use,” which shoots the player cannon. Moreover, the aliens always move in a predetermined pattern, approaching the bottom of the screen whilst moving left-to-right, then right-to-left. For such a game, where from a state s_0, the state s_1 that results from applying “use” can be reliably determined, a heuristic search such as A* may fare relatively well (because it is known where the player avatar will be, where the aliens will be, and what the result of “use” is). However, in a game such as Pac-man, where the ghosts sometimes move randomly, it is impossible to fully map the search space.

In the context of a relatively wealthy enhancement space, a particularly pertinent question is whether optimisation for one class of problems offset by a decrease in performance in another class. In [17], the authors discuss this under the guise of the “no free lunch” (NFL) theorem, which states that the average performance of an optimised algorithm tends to be the same as the average performance for all other such algorithms. This means that an improvement in one area is paid for elsewhere. Specifically regarding general video-game playing, Ashlock et al. [18] find evidence that this may not be problematic, since an algorithm which is too general is not useful anyway. They go further and state that the games humans find interesting are only a small subset of the large space of games available. Imagine an equivalent of Aliens where an alien player must “repel” human invaders; further picture that due to some sort of civilisational quirk, the alien player would deem invasion a winning situation. Writing an agent general enough to play both our version of Aliens and the reverse, alien version of Aliens may be an interesting experiment, but completely useless in practice.

The solution proposed by Ashlock et al. is to have specialised algorithms combined in a portfolio which a hyper-agent may choose from. The reasoning here is that some algorithms are better in the beginning of a game, whereas some are more naturally suited to the endgame. This echoes a common practice in chess engines: when the agent has detected that near to its current position there is a win state, it switches to a heuristic search such as A* to find the shortest path to that state (e.g. “mate in 4 moves”). To determine which algorithm is suitable for each game, the authors propose a method of classification. Developments in game classification and feature selection are discussed in section 2.3.

2.2 Hybrids

In the context of the strengths and weaknesses of the Monte Carlo method, as well as an awareness of the NFL theorem, some of the research in general video-game playing focuses on a hybrid approach: an agent comprising two or more algorithms which interact. The hope here is that together, the algorithms are more performant than when taken individually. It may also be the case that one of the algorithms that form a hybrid cannot act as an agent on their own, as is the case with exhaustive search in non-deterministic environments.

One of the possibilities that has received a lot of attention is a combination of Monte Carlo tree search and genetic algorithms. In a genetic algorithm (GA), a possible solution is modelled as an individual which is part of a population of programs. At each generation, individuals are cloned, mutated (where one or more parts of an individual is changed randomly) or crossed-over (two possible solutions serve as parents to create a new individual whose structure is a mix of its parents' structures). Additionally, a fitness function is computed for each individual to quantify how closely it resembles a solution; this is can be a score to maximise or an error to minimise. After a number of generations, the fittest individual may approximate a solution to the problem relatively accurately. One MCTS/GA hybrid uses a weighting of features to bias the roll-outs in the simulation step of the search[19]. Whereas the features are hand-coded, the matrix of weights is evolved through a genetic algorithm, whose individuals constitute MCTS play-outs. The results were promising for the game of Space Invaders, where a pre-evolved agent (i.e. whose feature weights were evolved before game-play) outperformed a standard MCTS by a mean score of 953 to 674.

Using the same idea, Alhejali and Lucas[20] devise a method where domain knowledge is used to evolve MCTS roll-outs for the game Ms. Pac-Man. In Ms. Pac-Man, the player must eat food pellets on a maze-like map, while being chased by 4 ghosts. Contact with the ghosts causes the player to lose a life; the game is over after all 3 lives are lost. Also scattered throughout the map are power pills, which, when ingested, allow the player to consume the ghosts themselves for a score bonus. In this solution, domain knowledge is encoded as parameters in the genetic algorithm, with features such as direction and distance to one of the edible ghosts, distance to a power pill or even logical aspects such as whether the player is in danger. A closer look at the features reveals similarities to the knowledge-based evaluation enhancement described in the previous section. The result of the experiment shows an improvement over standard MCTS (average score 32,641 to 28,116) and hints at the potential of hybrids to perform better than their constituent algorithms.

Further work is presented in [21], where the authors perform a cross-test on variations of MCTS that use random roll-outs (standard MCTS), evolutionary adaptation as described in [19], a knowledge base without evolution and a knowledge base with evolution. The results show that for the first three types, the win-rate is between 20% and 25%, whereas for the latter, it is 49.2%. An interpretation may be that employing a knowledge base without a way of filtering out inconsequential knowledge is not useful; similarly, genetic variation that is not anchored in knowledge of the game results in ineffectual agents that only marginally improve on the performance of standard MCTS.

Another type of hybrid combines MCTS with a rolling-horizon evolutionary algorithm (RHEA)[22]. In RHEAs, an individual represents a sequence of actions. The RHEA evolves the best possible sequence given a computational constraint (the 40ms time limit for GVGAI) and returns the first action in the sequence as the next move to be made by the agent. Here, there are different ways of hybridisation: adding MCTS roll-outs to RHEA (an integrated hybrid), sharing the computational constraint between the two (an alternative hybrid), as well as several enhancements related to domain knowledge. Layering simulations on top of RHEA is conceptually simple: first, the current action sequence is evolved; then, roll-outs are played from the last state, thereby allowing the agent to see further into the future. This has the benefit of allowing an agent to detect paths that lead to a possible loss early in the game (as described earlier when discussing the roll-out depth). Finally, the fitness of the sequence is computed taking into account the simulated states. The alternative hybrid entails using the two algorithms separately: the agent runs a RHEA for a certain amount of time, then a Monte Carlo search for the remaining time until the 40ms are reached. In [22], the two are given equal weight, although there may be an optimal time-distribution that is not 50-50.

The enhancements proposed for the integrated hybrid are sequence planning (reusing a sequence of moves), occlusion detection (removing actions that occur in an action sequence without impacting its fitness) and NPC attitude-detection. The latter explores whether in the current game, NPCs are friendly or adversarial. This information is unknown to the agent when the game starts and may change throughout play (like in Pac-Man). Because of this, agents that are incentivised through knowledge-based state evaluation to explore interactions with NPCs may find themselves consistently losing games where NPCs are unfriendly. Conversely, agents that are too pessimistic or cautious may never progress in games where winning is dependent on interaction with NPCs. One variation is that summarised in section 2.1.3, where this interaction is calibrated by learning weights associated with sprites on the map. The attitude-detection in [22] suggests learning the nature of an NPC before making a move towards or away from it through simulation, then applying an extra reward or a penalty, depending on its friendliness. To eliminate statistical ambiguity when comparing the resulting agents, the authors compute the Elo-ratings[23] of each and rank them according to their wins, draws and losses. The best rank is claimed by the integrated hybrid without enhancements, closely followed by the same hybrid but with attitude-detection. The integrated hybrid has the same number of wins as a standard MCTS but loses less often (4 losses as opposed to 14). This suggests there is an improvement, though it is not large enough to win more games (it just draws more). The alternative hybrid fares worse than standard MCTS or a standard RHEA; it is possible that the gains in performance resulting from the more robust hybrid approach are not visible given the time limit. Indeed, it is likely that the context-switching leaves neither algorithm with enough time to find a meaningful outcome.

At least some of the interpretation of these results is based on human understanding of the problem-space, which introduces a bias in how performance is analysed. To be able to carry out an efficient analysis of an agent's track record, it must be possible to extract at least some indication as to why it makes certain decisions. If an agent is opaque—a black box—, even if it is moderately successful it is difficult to improve it since there is no window into its behaviour. Moreover, as evidenced by research into robustness[24] and game difficulty analysis[22], it is often the case that an agent does well in one type of game but has a poor win-rate for others. To address these issues, as well as to find more data points relating to performance (not just win, draw and loss rate), Bravi et al.[25] investigate which in-game metrics offer information about an agent's behaviour. To retain domain-independence (i.e. to avoid biasing the results towards particular games), the authors only collect agent- and not game-related features. Some of these agent features are the recommended action, the probability vector for available actions, a value vector for possible states and the percentage of the computational budget used. The first three features create basic explainability (or transparency): why the agent chose the recommended action. Keeping track of the budget used allows for like-for-like comparison between agents, some of which always maximise the time used (attempting to use the entire 40ms at each turn), whereas some are more budget-saving. If it's possible to achieve the same performance with less time, it may be possible for the faster algorithm to be enhanced (by employing the time left) or for the slower algorithm to be improved (by modifying internal parameters); all other things being equal, it is better to have a faster agent than a slow one. Tracking these metrics allowed the authors to observe repeated behaviours such as the agent avoiding certain types of objects, killing NPCs or simply maximising the score.

For their study into agent robustness, Perez et al.[24] organise a set of modifications to game mechanics that lower agent performance. It is important to note that these modifications are not game-specific, so the challenge is greater regardless of the type of game played. For example, the authors introduce a score penalty for each move, forcing the agents to reach a solution with minimal hesitation; they also discount future rewards based on the depth of the state they are encountered relative to the current state. By far the most impactful modification is the introduction of noise, which is unpacked into two variations: a noisy world and a noisy forward model. The forward model is the functionality within the GVGAI framework that allows an agent to explore states in the future without actually making the moves towards those states; it is what makes Monte Carlo simulations possible. For the noisy world variation, every time an action a is recommended by the agent, there is a probability that an action a' will be taken instead. The noisy forward model includes noise for the action to be taken, as well as noise for the actions explored during simulation. Two of the agents studied which employed forms of best-first search reported the largest drop in performance, whereas a variation of MCTS called open-loop Monte Carlo tree search (OLMCTS) does best out of a set of hybrids, RHEA and heuristic search controllers. This may be traced to OLMCTS's lack of domain knowledge, which makes it resilient when noise is introduced, whereas the heuristic agents cannot adapt to unexpected cases as easily. In other words, knowledge is harmful when incomplete: the agents that relied on their heuristics for an accurate picture of the game find themselves at a loss when something unknown occurs. Despite the decrease in performance, this does not mean these agents should be discarded; it should be possible to complete their knowledge by informing them of the possibility of noise, in which case behaviour will be adjusted to reflect the uncertainty of each action taken.

Looking at things in perspective, there exist agents whose strengths and weaknesses may be complimentary, when properly configured; there are metrics that give a glimpse into an algorithm's behaviour, and there are environment features proven to impact the performance of some techniques. Is there a way to compile this information into useful knowledge about how to create a general video game agent?

2.3 Hyper-agents

A hyper-agent is an algorithm designed such that there are two levels of organisation: a high-level algorithm and a set of low-level algorithms. The set of sub-agents is also known as the algorithm portfolio[26]. The hyper-level chooses one of the low-level algorithms to carry out a task, based on its features and known agent strengths and weaknesses. However, it's possible that the hyper-agent does not have much information about the environment or know any features about the task at hand; it may also be unclear how some of its sub-agents behave in the current conditions. Formally, this is known as the algorithm selection problem[27]; it has spawned research into metaheuristics, feature selection and algorithm explainability.

2.3.1 Heuristics

A heuristic is a rule of thumb; a small, verifiable principle that guides an action. Humans use heuristics in day-to-day decision-making, which is governed by uncertainty. Often, the choices made using imperfect information may rely on previous occurrences of similar circumstances, on social principles, or on memory availability[28]. Importantly, they do not rely on explicit calculations of probability, on use of Bayes' theorem or on a unified rational decision-making strategy. It is perhaps not baseless to say that human cognition is not adapted to perfectly rational decisions, but to decisions that were good enough to ensure survival. Parallels may be made with search agents, whose performance in an immense combinatorial space without any heuristic is less than enviable. On the other hand, introducing a heuristic doesn't remove error; indeed, it may not even be the best heuristic for that algorithm and specific problem instance. Algorithms can be said to reflect human bias, insofar as their heuristics are human-crafted. The field of hyper-heuristics grew in an attempt to reduce this unhelpful human influence.

A hyper-heuristic is a rule about how to select or generate a heuristic in order to make search more applicable in the general. It is a heuristic about a heuristic. By another definition, a hyper-heuristic is either a search method itself or a learning mechanism that optimises the heuristic chosen for the algorithm itself. This latter definition encloses a possible classification: a hyper-heuristic that learns, and one that does not. Learning occurs from feedback about heuristic performance. A concrete example: a hyper-heuristic H_1 has a set of heuristics A, B and C it can apply to a search algorithm. Assuming an identical search scenario repeated 1,000 times, A finds a solution in 3.2s on average, B in 3.5s and C in 2.8s. There are two types of learning: online and offline. If, while the search is running H_1 notices that heuristic C seems to have the best performance and allocates more weight towards C, thus applying it more often, online learning has occurred. On the other hand, if after the 1,000 runs H_1 can encode a rule of the type "in this scenario, C does best out of this set of heuristics," it has learnt the optimality of C offline. Put otherwise, the difference lies in the point where the search is adjusted. Learning from feedback about performance was an important theme for hybrid agents, and it is equally important for hyper-heuristic agents. If the interpretation of feedback is delegated to a learning mechanism as opposed to a human designer, it may be possible to eliminate human bias towards what we intuitively feel would work.

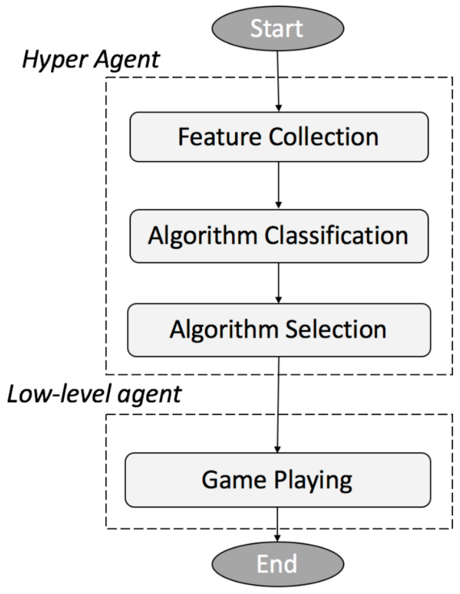
Hyper-heuristics can also be classified on a different criterion: whether they construct a solution or refine an initial guess. Constructive hyper-heuristics start with a blank slate and a pool of heuristic components. At each stage of the game, these components are benchmarked against each other and the best one is added to the solution. When a full solution is reached and the task is completed, the hyper-heuristic is stopped. The other version is called a selective hyper-heuristic (in some sources, "perturbation" or "local-search"). It starts with a rough guess and repeatedly searches the neighbourhood of similar solutions, benchmarking their performance at each step. The termination condition of the process is not as clear as it is for constructive hyper-heuristics and may depend on a domain-specific acceptance state. Different acceptance states lead to different solutions.

Examples of applications of hyper-heuristics abound. In one study[29], a selection mechanism called tabu search is combined with simulated annealing to determine the optimal size of shipping boxes in order to maximise volume used (i.e. leave as little empty space in a box as possible). When a heuristic is applied, if it does not result in an improvement of the search it is added to a "tabu" list. This list is emptied only when, at the end of the current iteration, none of the available heuristics resulted in an improvement. Using this mechanism, the search can explore which of the heuristics is optimal at each move. Another application is for exam and class timetabling[30], where assigning courses and exams to time slots is modelled as a graph-colouring problem. Each node represents an event, while edges are conflicts between two events. Constructing a timetable is done iteratively by searching through a set of 5 heuristics to efficiently schedule the remaining events. These heuristics take into account the number of conflicts of an event with other events, the number of conflicts weighted by the number of students involved in the event, the number of enrolments, number of conflicts with events already scheduled and number of time slots available for each event. Other applications are in vehicle routing, the bin packing problem (where an agent must pack objects into a finite number of containers so that the least amount of containers are used) and the Boolean satisfiability problem[31].

Whether hand-crafted heuristics for a particular problem are optimal is an open question. Another concern that arises is whether their application is optimal, i.e. if the hyper-agent can always tell which heuristic to apply given a set of problem features. In [27], Smith-Miles casts the algorithm selection problem as a learning problem: how a hyper-agent can learn to differentiate amongst its subagents, as well as what aspects of their design, performance or robustness are relevant to the choice. As in [25], the author highlights the need for algorithms to be more transparent, so that researchers can find features that are likely to correlate with algorithm performance. A classifier can use a collection of features and parameters coupled with data about performance to output which algorithm is well-suited for a given type of game. Compiling a useful collection of features has a direct impact on the quality of the output, so finding the highest-information gain features are critical (those features that reveal the most about an optimal game-algorithm pairing). These can be game, algorithm or performance features. In the case of algorithm features, for MCTS the number of expanded nodes, leaves (nodes with no children), the tree width and depth and distribution of node attributes all offer additional information about the inner mechanism which could be used by the classifier. For performance, time and other computational resources are routinely used as benchmark.

2.3.2 As applied to video-games

Mendes, Togelius and Nealen[32] demonstrate the viability of the hyper-heuristic method by classifying GVGAI games based on their features, then training a support-vector machine hyper-agent and a J48 decision tree algorithm to select the appropriate controller. Features used in the classification are divided in groups according to their focus: resources (whether the game has resources, if the avatar has resources, how many different types etc.), NPCs (number and type of NPCs), sprites (movable or immovable) and game mechanics (dimensions, if the player can attack etc.). Their hyper-agents use offline learning to train their models prior to gameplay, but also update the dataset by collecting features during gameplay. Figure 2 shows the sequence of actions that their hyper-agent takes at each game tick.



(Figure 2: hyper-agent gameplay with GVGAI[32])

The algorithm portfolio in [32] consists of previous winners of the GVGAI competition *adrienctx*, *JinJerry* and *YOLOBOT* as well as four sample controllers included with the framework: MCTS and its open-loop variant, a rolling-horizon evolutionary algorithm and a greedy hill-climbing agent which picks the state with the highest reward at each game tick. While training the models, the game features with the highest information gain were found to be the types of NPCs, types of resources that the player has and whether the player can move vertically. The subagents picked according to these features were *adrienctx, YOLOBOT* and *JinJerry*, suggesting that the hyper-agent quickly learned to exploit the most powerful algorithms and steered clear of the weaker ones (for example, there was no occurrence of the hill-climbing algorithm). The two hyper-agents outperformed the strongest subagent, *YOLOBOT*, by 721 wins for the SVM, 709 for the J48 decision tree and 654 for *YOLOBOT*, out of a total of 1,025 game plays. Results seem to hint at the potential of hyper-heuristic gameplay, with many questions remaining open for research, such as finding more or better features to extract from the game environment (or whether it's possible to extract these manually when the game space increases as it did with the introduction of real-world physics in [13]).

Using machine learning to classify tasks, then choosing algorithms known to perform well may seem like an added step when the task itself can be learned using a neural network. On the other hand, it may be the case that to achieve the kind of generality needed for strong AI, the agent would need some sort of algorithmic toolbox that it can refer to as needed. There is strong intuitive appeal to the idea—although most of cognition is invisible to humans, there are common strategies that people use, for example when problem-solving or when trying to learn a new language. In the same way that original formulations of AI encoded knowledge as expert systems (the "what," declarative knowledge), perhaps strategy can be encoded as modular algorithms and heuristics that can be mixed and matched for the situation at hand (the "how," procedural knowledge). In this sense, the problem can be approached from two angles: improving classifiers and finding better features (machine learning) or coming up with yet more efficient algorithms (algorithm creation). With the recent machine learning advances, this does not seem unattainable, though AGI is still some way away.

Declarative and procedural knowledge could both be said to belong to instrumental rationality—what an agent knows about the world and what it knows how to do in order to achieve its goals. There is one more type of knowledge: the "why." The ethics of a general AI is by far the most impactful aspect of its invention. Given humanity's close calls throughout history, especially since the invention of nuclear weaponry, it is important to understand the leverage our species would achieve with the invention of AGI—leverage that could lead to creation and flourishing, or to destruction and evenf extinction. Small steps towards better governance, towards more transparent technology (hence explainability) and towards higher awareness of the risks may compound into significant benefits in the near- and far-future.

3 Project aims

The goal of the project is an exploration of the hybrid and hyper-agent methods for general video-game playing. As detailed in the initial report[33], the objective was a hybrid agent that uses MCTS and A* to play games on the GVGAI framework. The agent should follow framework constraints such as the computational budget for initialisation (1s) and move decision (40ms), as well as being compatible with the game-playing mechanics. Implementation specifics are discussed in section 4.2.2. Also, part of the goal is to benchmark the new agent against other successful GVGAI competitors; this was reduced to testing against the standard MCTS implementation due to the computationally intensive task of playing several levels per game for 110 games. Testing is discussed in depth in section 4.3.

Throughout the project, a lot of attention was directed towards a good way to think about the problem of algorithm hybridisation: which algorithms work well with MCTS, how they should be combined and what the technical obstacles would be in doing so. Specifically, it remained unclear for some time whether it is optimal to time-share between A* and MCTS or to use A* as a path-finding subagent and MCTS as a hyper-agent. Most of the types of agents considered did not get implemented due to time constraints. A review of the ideas that were explored as well as a discussion of the choices made is presented in section 4.2.1.

4 Technical documentation

The outcome of the project is a hybrid between Monte Carlo Tree Search and Time-Bound A* search with three enhancements: a knowledge-based evaluation of game states, a simple implementation of move history called *inertia*, as well as breadth-first tree initialisation.

4.1 Software used

As the agent needs to be compatible with the GVGAI framework, it is implemented in Java. An agent is defined as one or more classes of which one must implement the `AbstractPlayer` interface. This interface defines a constructor and an `act` method that gets called at each game tick. The `act` method must return an action belonging to the enumeration `Types.ACTIONS`, which in turn encompasses all known actions for a player in GVGAI. Notably, for each game type, the actions available are only a subset of all actions defined; for example, in Aliens one cannot move up and down, whereas in Zelda, the avatar can move up, down, left, right and apply the `USE` action to attack enemies. The implementation of both MCTS and TBA* uses no outside libraries but depends heavily on GVGAI specifics.

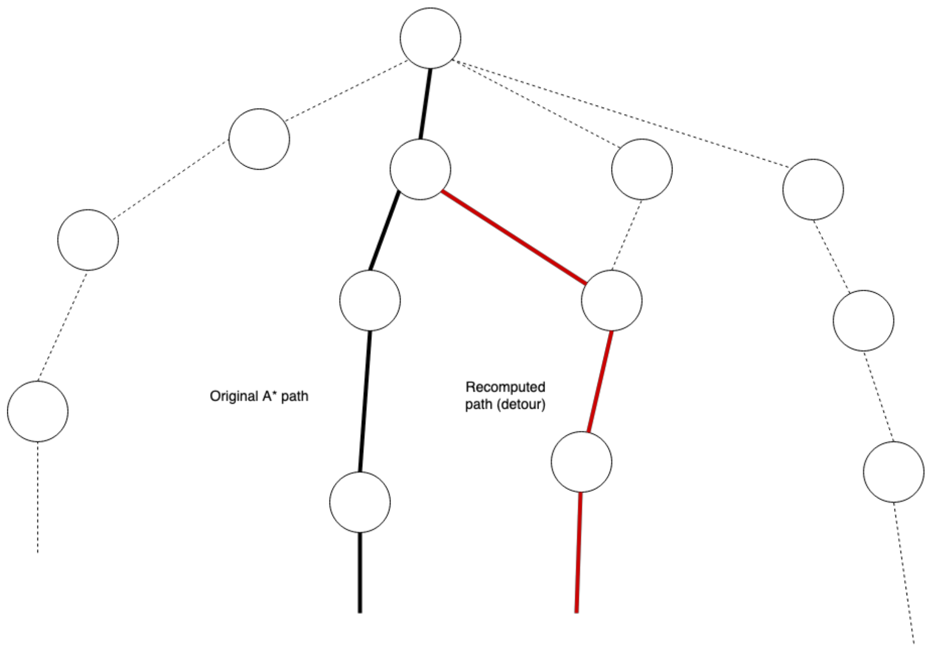
The design process was made easier by since it was known that the resulting agent would have to integrate with the GVGAI framework. In fact, this was a useful starting point, since many of the details were known before designing the algorithms, such as implementing `AbstractPlayer` and communicating with the forward model through a `StateObservation` object. The latter encompasses all the game knowledge available to the player. A few examples are the score, whether the game is finished (which then ramifies into a winning and losing state), the available actions, the player health points as well as the two-dimensional coordinates of sprites on the map (resources, portals and NPCs). The player interacts with this `StateObservation` object, advancing the state by applying an action. Another key component of the agent-game interaction is the `ElapsedCpuTimer` object, which keeps track of the time spent by the controller per move. According to competition rules, a controller that deliberately overshoots the 40ms limit per move while still returning an action within 50ms has the `NIL` action applied, which has no effect on the game. Controllers that do not return any action within 50ms are disqualified from the competition. A final point about the framework itself is that includes sample controllers. As of late 2018, these were Monte Carlo tree search, *adrienctx* (the winner from 2014, an open-loop expectimax tree search), a rolling-horizon evolutionary algorithm, a greedy hill-climbing algorithm as well as two very simple controllers (one picks a random action, the other simply applies `NIL` at each tick). These allow prospective participants to investigate some of the common solutions instead of starting from scratch. Additionally, one rule of the competition is that if an agent is submitted to participate its authors agree to publish the source code; as such, the public has access even to *YOLOBOT*, one of the highest-ranking agents to date.

To test the resulting hybrid against MCTS the rank sum test[34] was used as implemented in the Python library `scipy`[35]. This was opted for despite the change in programming language due to `scipy` being one of the foremost libraries for statistical analysis globally, as well as due to the intuitive implementation of `ranksums` in the `stats` package. Microsoft Excel was used to interpret the results manually and generate the visualisations.

4.2 Design and implementation

4.2.1 Design process

The agent was designed iteratively, through successive refinement of the initial A*-MCTS idea. One of the ideas explored is a hybrid that has a high-level MCTS (HL-MCTS) algorithm and two low-level algorithms: another MCTS (LL-MCTS) and A* search. HL-MCTS would consider macro-actions, which are then delegated to either LL-MCTS and A* on the basis of a heuristic. This heuristic was investigated further from the viewpoint of game determinism. In [12] a method of checking for game-determinism was proposed. An equivalent implementation would be to have an expected reward model: an action is recommended by one of the two low-level agents, the expected reward of this action is stored; if, when taken, the action results in a different reward, the game probably contains randomness. This verification would be the basis of cycling between the two low-level agents, since it is known that heuristic search does not deal well with randomness and noise[24]. Another possibility is to use the initialisation time of 1s to run A* search to find the optimal path to an end state. Then, at each tick, the agent checks that the actual reward equals the expected reward. If it does, the next move on the path is made; if it does not, a move is picked by LL-MCTS and the A* path is recomputed (fig. 3). Appendix 2 contains a flowchart describing the mechanics of such an agent.



(Figure 3: adjusting a previously computed path with A*)

An alternative to using a heuristic to decide between the subagents is time-sharing, as mentioned in [22]. It is likely that this results in suboptimal performance but was chosen due to its simplicity.

One obstacle to using heuristic search as a player of real-time games is that most search algorithms are not anytime algorithms (these return a solution even if interrupted). This is an essential property for a GVGAI agent, as there is a hard cut-off point at 40ms and it is not acceptable to return an action only some of the time. Variants of A* search that work in real-time are part of the literature on re-planning algorithms. Broadly, these work by limiting how much of the search is done at a time, based on the computational budget available. Anytime Dynamic A*[36], for example, reuses partial solutions and decreases a suboptimality bound if there is time left, making the search more accurate. ADA* has been successfully applied to robot path-planning. Another re-planning algorithm is time-bounded A*, presented in section 4.2.2. The alternatives for the heuristic search were not evaluated for their benefits and drawbacks—instead the choice was for the simpler, easier to implement algorithm that could be adapted to GVGAI. Further work may be centred on comparisons between hybrids that use different re-planning algorithms, perhaps without other enhancements so performance is accurately measured.

Also worth evaluating is the design of macro-regions. A macro-region is a cluster of positions on the map grouped in a particular way. These regions could be based on map layout (e.g. a 9x9 grid map could be clustered into 3x3 squares, like in Sudoku), or on the presence of an interesting feature (e.g. a macro-region containing an NPC or a portal. Layout-based clusters are easier to implement, but don't necessarily contain any useful information, whereas feature-based clusters can be tricky to define. For instance, NPCs move around the map and portals may only be enabled on acquiring a resource, meaning that clusters would have to be recomputed dynamically. At least for grid-based games, where at each tick sprites can only move in 4 directions, this could have another use: knowing the general area of an enemy NPC. However promising, this was not implemented because it would introduce additional complexity at the action level, with macro-actions such as "move to the door" or "move to square Y" being necessary to take advantage of a macro-region layout.

4.2.2 Monte Carlo tree search and time-bounded A* search

Initially, the MCTS was implemented separately from the GVGAI sample (denoted sampleMCTS from now on) since it would offer more insight into the key concepts of the search. This agent followed the structure presented in [9] under "Algorithm 2", an implementation which uses UCT as the basis of the tree policy. Unfortunately, some of the constraints in the pseudocode conflicted with the specifics of the framework, causing small pieces of functionality to be unnecessarily complicated and error-prone—for example the best action could not be returned directly, as actions are not intrinsically connected to the resulting child node. Instead, in GVGAI each action and each child node correspond to an index—this is how the connection is retraced between the two. For simplicity, the sampleMCTS algorithm was adapted to the purposes of this paper. The reasoning behind reductions of scope is discussed in section 5.1.

For the first adaptation, the state evaluation function was changed from a simple assessment of game score to a knowledge-based heuristic that takes into account score, player health, whether the current state is a terminal state (winning or losing) as well as distance to the nearest resource, portal and NPC. The final state reward is characterised by the following equation:

$$Q = score + 100 \cdot health - w_{resource} \cdot d_{resource} - w_{portal} \cdot d_{portal} - w_{NPC} \cdot d_{NPC}$$

where the weights are tuned manually to 100, 100 and 50 for resources, portals and NPCs, respectively. This is to encourage exploration towards resources and portals while remaining cautious around NPCs, since they can be adversaries. Training weights as in [12] would result in better performance since the weights would be fine-tuned as the agent explores, but is out of scope for the current project. A tweak similar to the NPC attitude check[22] would also help to train w_{NPC} to a more accurate value. Transitioning to a knowledge-based evaluation significantly improved the performance of a greedy hill-climbing agent, which for each tick picks the best move available according to the evaluation function.

Second, the tree in MCTS is initialised with breadth-first search to ensure the Monte Carlo search has pre-expanded nodes to which it can assign UCT values. As outlined previously, this is to avoid quasi-random behaviour in games where the branching factor is high and there is not enough time to try all available actions. This is as simple as generating the root node's children by looping through the available actions; the root node is defined as a node whose parent is `null`. Given the grid physics in GVGAI and the number of available actions always being less than 6 (UP, DOWN, LEFT, RIGHT, USE, NIL), a high branching factor is not possible. Instead, the breadth-first initialisation results in a small speed boost in that calls to the expand function are avoided for the first child nodes.

Finally, a concept similar to inertia is implemented when MCTS recommends a move; this is a simple version of progressive history[12] where at each game tick there is a probability that instead of taking the recommended action, the agent takes the action it took in the previous tick. This results in longer sequences of moves and may be one way to achieve simple macro-actions (which do not necessarily have a complex arrangement of atomic actions). The probability is pre-set to 50% which results in equal exploration of longer sequences and normal, one-action-at-a-time gameplay. Further work can be done here to determine experimentally what the optimal value is for action repetition. In [12], previous successful actions are stored together with their context, which allows for more informed moves; there are also other ways of encoding macro-actions, such as through individuals in a genetic algorithm.

The variant of A* used in the hybrid is described in [37]. Time-bounded A* is an adaptation to the original heuristic search that results in an anytime algorithm. TBA* works by having a fixed amount of state expansions occurring at each iteration. Once this limit is reached, the optimal state amongst the ones expanded is traced back to the current position, thereby creating a path from the current state to the destination. Similar to the expansions, the trace-back also only occurs a fixed number of steps at a time. Together, these two limits ensure the real-time property of TBA*. In this implementation, the algorithm quickly hits the maximum memory limit due to the game states stored in the open and closed lists, since the states contain a lot of information. To work around this, states can be evaluated when expanded, but only stored via a unique ID. Alternatively, states that have been in the lists for longer than a specified limit may be pruned, since it is unlikely they will be explored. One more limitation is that in order to work, A* needs a heuristic for the distance from a state to the goal state. This is not easily achieved for the games in GVGAI, since the only information available is about the current state and states previously expanded. One solution may be to use the knowledge-based heuristic described earlier, or to only use TBA* as a path-planning algorithm when the destination is known.

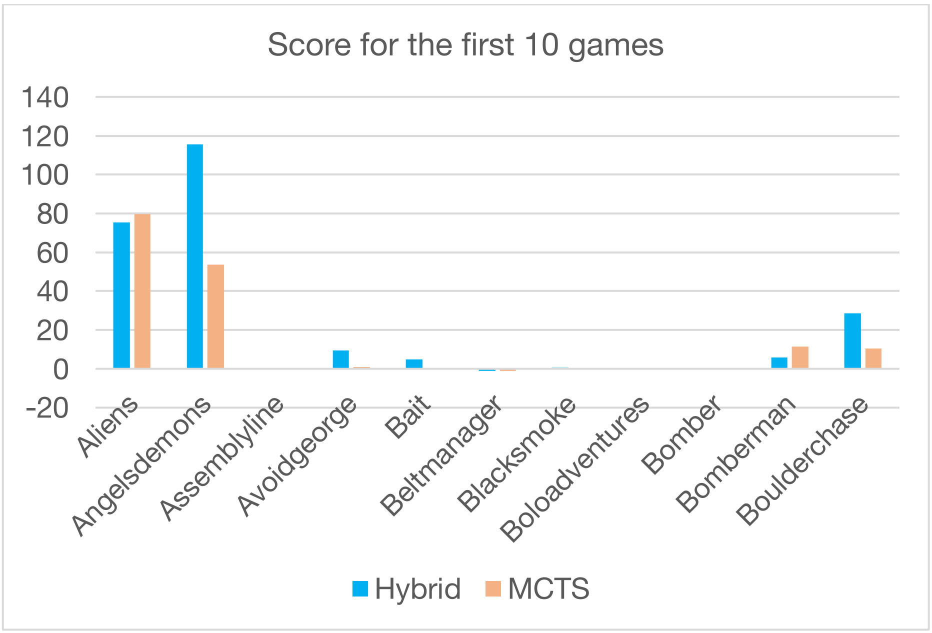
The two algorithms are combined in a time-sharing hybrid. MCTS runs 80% of the time and TBA* for 20% of the time (i.e. at each turn there is a 0.8 probability that MCTS will run for 40ms, otherwise TBA* will run). This solution can be refined by testing for determinism and running MCTS if the game is stochastic, TBA* otherwise; this is done by *YOLOBOT* with MCTS and breadth-first search. An obstacle to creating the hybrid was limited visibility into how the agents interact. Although there are tools within GVGAI that log results and agent moves, sampleMCTS does not output the structure of its tree or the states considered ranked by their reward, both of which would be useful. In the interest of focussing on core functionality, simple output statements were coded in instead of unit tests to evaluate the behaviour of functions in sampleMCTS. These basic logs offered a glimpse into the internal mechanics and helped debug the errors that came up.

4.3 Testing

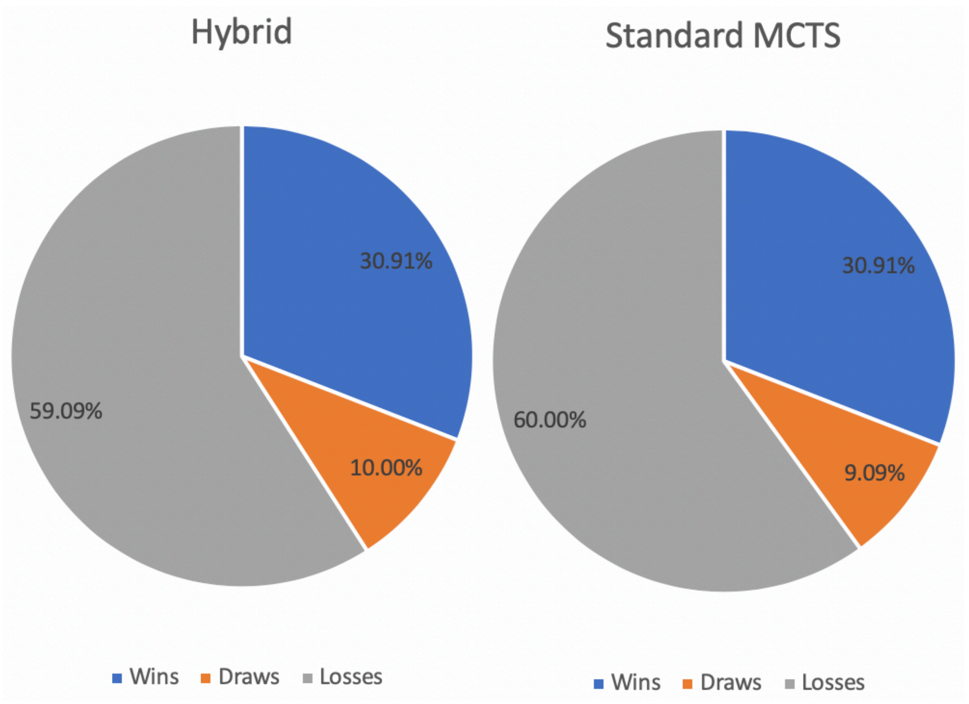
The hybrid agent was tested across all 110 games in the GVGAI framework. Win rate and score were collected. SampleMCTS was picked to benchmark the performance of the new agent due to the versatility of the algorithm in previous iterations of the competition. The test harness was written in Python; it reads the gameplay results from a file and creates two arrays, corresponding to the two agents. These arrays contain either the win rate or the game score. Then, each element from the first array is compared against its corresponding item in the second array to determine whether values in the first sample are more likely to be larger than values in the second sample. The test returns two values: the Wilcoxon rank-sum statistic and the p-value of the test. The null hypothesis for this test is that the two sets of measurements are drawn from the same distribution, i.e. there is no statistically meaningful difference between the sets of values. A small p-value indicates strong evidence against the null hypothesis—which means that one set is likely to contain higher values. The array with the higher values corresponds to the more efficient agent.

For each game, the first level was played 10 times. The game is considered won if more than 50% of plays result in an agent win, whereas the score is the average score of the 10 plays. The tests took approximately 12 hours to run on an Intel i7 processor, 3.3Ghz with 16Gb of RAM at 1867MHz. The rank-sum test was run in three configurations: comparing win rates, scores and scores weighed by their win-rates, respectively. The latter meant that situations where a higher score was achieved despite losing the game were not considered as a performance improvement. For all test cases, the p-value is not small enough to reject the null hypothesis, suggesting very similar performance for the two agents (p-values were 0.935, 0.930 and 0.887 for the three test configs).

On further examination of results by win-rate, both agents win 34 out of 110 games (fig. 5). When comparing scores weighted by the win-rate, MCTS outperforms the hybrid by a total score of 2,828 points to 2,288. There are two games which the hybrid wins that sampleMCTS never wins (Bait and The Shepherd), and several others that the hybrid draws, which the sample loses. On the other hand, games Butterflies, Chip's Challenge, Ghostbuster and Zen Puzzle are won by the sample but drawn by the hybrid. In the future, it may be useful to observe some of the plays to see if any difference in style is apparent. Appendix 1 contains the results structured by game.



(Figure 4: results for a subset of the games)



(Figure 5: the performance difference is negligible over the 1100 plays)

Since the p-values are not small enough, it's unclear which of the two agents is better (if any). It may be the case that more testing would reveal better results, for example by playing all levels for each game a fixed number of times. In the context of other enhancements resulting in improvements of up to 17.4% [12], it is not immediately clear why the hybrid configuration fails to perform better. Further work is needed to better integrate the two algorithms that make up the hybrid, perhaps using TBA* as a path-finder only instead of as a game-playing agent, as well as quantifying the impact each enhancement has on the performance of MCTS by benchmarking them individually.

5 Project planning

The project was divided into JIRA epics that correlated to deliverables throughout term: initial report, initial oral examination, abstract and poster, final report and the final presentation, demonstration and oral examination. Each epic contains stories that correspond to high-level tasks to be carried out for that deliverable. For example, the final report had the following stories: advanced literature review, building product context, describing objectives and their achievement, product technical challenges, testing plan, quality assurance and reflection. Stories were further divided into small, manageable tasks; for the literature review these are individual papers to summarise; for the product, they were pieces of functionality of the agent. This structure helped guide work towards the most important areas as well as keep track of tasks critical to the project.

As the solution was developed, changes were pushed to the school GitLab server. Papers that were reviewed, Endnote reference libraries and the poster and abstract were included here for ease of use from different machines (at home and at University). Commits were created for mini-milestones when writing the final report just in case there was a need to roll-back the latest section.

5.1 Reflection

Good reflection should include a discussion of project risks. In the initial report[33], the author mentions time-management as the biggest threat to achieving the project objectives. This turned out to be a good estimation. The breadth and depth of the project were mostly unknown beforehand—the author had no previous experience with game-playing other than his own and only brief knowledge about MCTS from a previous module. As problems were encountered along the way, it was a good idea to reduce the scope and abstract away some of the complexity.

Most of the initial work was dedicated to acquiring an in-depth understanding of MCTS by writing it from scratch. This was a moment when the difficulty of writing even a seemingly simple algorithm was underestimated. Many hours were lost attempting to get this initial version to integrate with GVGAI, mainly due to bugs. Even though the agent would eventually compile and return a move within the 40ms, it tended to repeat previous moves, possibly due to a problem with child expansion within the tree policy. In fact, the watershed was when the original implementation of MCTS was abandoned in favour of working on improvements to sampleMCTS.

From then on, it was relatively easy to implement the three enhancements. From a coding perspective these were not complex, but it was important to make sure they did not break any of the existing functionality. To this end, extensive regression testing was done using a subset of games (Aliens being a personal favourite). These enhancements were great at building momentum and encouraged the author to not limit the agent to an enhanced version of MCTS.

Another major obstacle was reached when implementing TBA*, especially since it was done later in the project, with the deadline for the final report looming. The algorithm was largely based on pseudocode provided in [37]. Nonetheless, decisions must be made at the point of implementation: how to integrate with GVGAI and with the other half of the hybrid agent. Towards the end of the spring term the decision was made to redirect efforts to fix A* and enhance MCTS* towards preparing the test results and writing the final report. This resulted in an agent that is only 70% complete, in the author's view.

As expected, the biggest obstacle was managing the project at the same time as other deadlines, while having the right balance between an agent complex enough to be interesting but not too complex to write. There is room for improvement here: dedicating more time to the project earlier in the year would have altered the course later on, making it easier to create multiple versions of the agent to compare. In retrospect, too many resources were spent on understanding the research and not enough on getting hands-on experience with the algorithms. On balance, these risks were managed successfully in that an MCTS/A* hybrid was delivered as per the initial project aim.

Despite no mention of a performance objective in the initial report, it is certainly disappointing to not have any meaningful performance increase over MCTS, although this is most likely because of implementation specifics and not the MCTS/A* hybrid idea. The author considers this exploration of the game-playing AI space an achievement in itself; it was mostly fun and only briefly daunting; most importantly, it was instructive. There are many possibilities and variations within this idea and given more time a definitive result is bound to arise—regardless of its direction (perhaps MCTS and A* do not mix).

An overwhelmingly positive aspect of the project was exposure to the GVGAI framework. Despite having one of the most complex codebases the author has encountered, it implements the game mechanics smoothly, the code is easy to read (including some crucial comments) and testing is done through a simple class with pre-built methods. This project would have been significantly more difficult if the learning curve for the framework itself was steeper. The test harness was almost trivial to write given Python syntax and the simple API call to scipy.

6 Conclusion

In summary, a hybrid agent was created in order to explore performance improvements for the UCT implementation of Monte Carlo tree search. The agent contains an enhanced version of MCTS alongside the TBA* algorithm, an anytime version of A* heuristic search. The enhancements proposed are breadth-first tree initialisation, knowledge-based state evaluation and a simple form of action history called inertia. The hybrid is benchmarked against an implementation of MCTS with UCT through 1100 total game plays on the GVGAI framework. Results are inconclusive, possibly due to no significant performance change, or due to a statistical sample of insufficient size.

Further work should include an in-depth analysis of the impact of enhancements on performance—it may be the case that some of these are harmful overall (in the “no free lunch” sense) since MCTS derives much of its performance from being a versatile algorithm. There is also a lot to explore in the hybrid/hyper-agent area, given a portfolio of sub-agents that contains MCTS and an anytime version of A*. A heuristic based on game features may result in markedly higher win-rate and score for the hyper-agent.

Advances in the field of AI game-playing have wide implications on the possibility of artificial general intelligence. In particular, playing and winning at games which simulate reality closer and closer may be a viable approach towards a fully general intelligence. Furthermore, throughout history, many, if not all scientific advances have been incremental. Researchers build on and rediscover the results of their predecessors until knowledge gives rise to the next big invention or discovery. There is no reason to believe human-level AGI is different.

7 Appendices

7.1. Appendix A

Test results for the MCTS/TBA* hybrid alongside results for sampleMCTS. A win-rate of 1 means the agent won the plays for the game, whereas 0.5 means it drew.

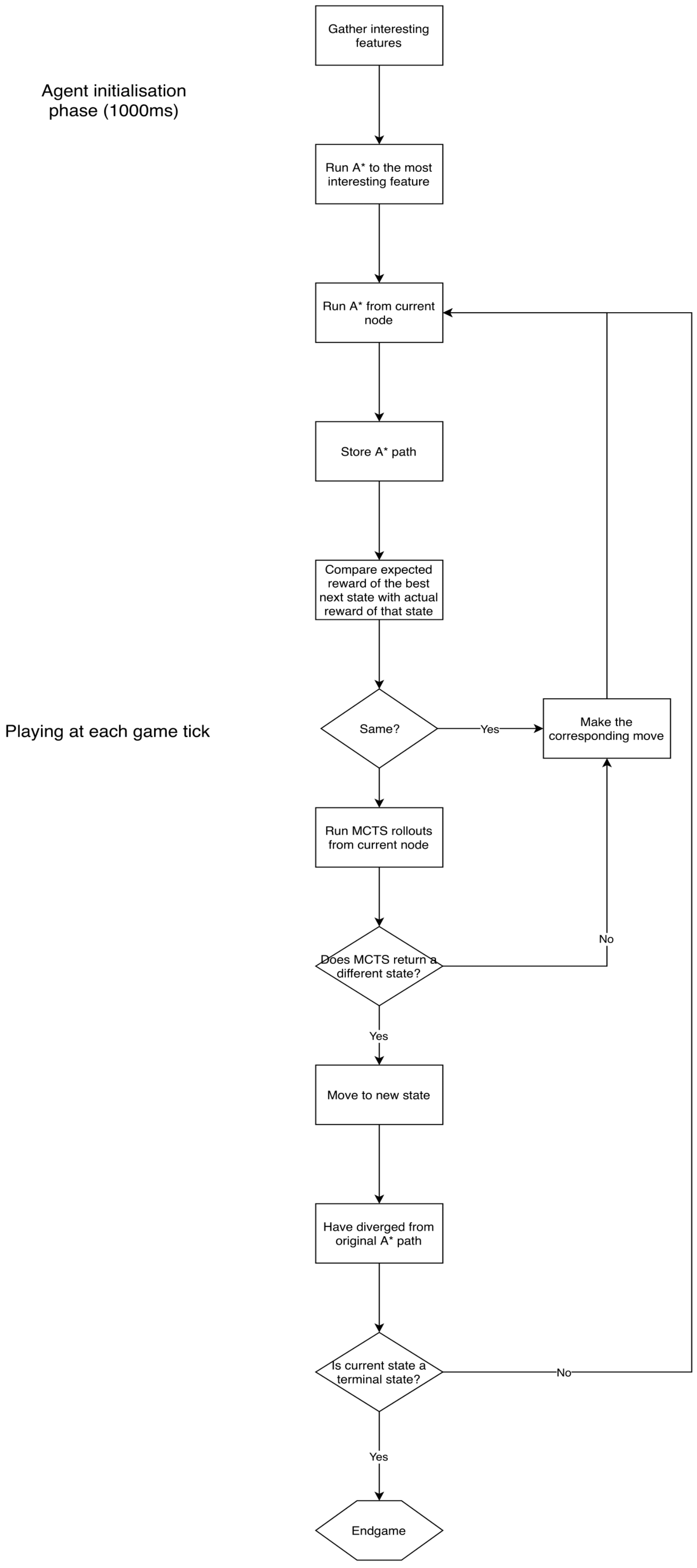
Game	**Hybrid**		**MCTS**	
	Win-rate	**Score**	**Win-rate**	**Score**
Aliens	1	75.5	1	79.5
Angelsdemons	0	115.5	0	53.5
Assemblyline	0	0	0	0
Avoidgeorge	0	9.5	0	1
Bait	1	5	0	0
Beltmanager	0	-1	0	-1
Blacksmoke	0	0.5	0	0
Boloadventures	0	0	0	0
Bomber	0	0	0	0
Bomberman	0	6	0	11.5
Boulderchase	0	28.5	0	10.5
Boulderdash	0	11	0	13
Brainman	0.5	18	0.5	18
Butterflies	0.5	40	1	45
Cakybaky	0	2	0	2
Camelrace	0	-1	0	-1
Catapults	0	3	0	3
Chainreaction	0	0	0	0
Chase	0.5	5	0	4.5
Chipschallenge	0.5	30	1	36
Chopper	1	16	1	14.5

Circuit	0	0	0	0
Clusters	0	0	0	0
Colourescape	0	0	0	0
Cookmepasta	0.5	16.5	0	6
Cops	0	5.5	0	3
Crossfire	0	0	0	0
Defem	1	50	1	50
Defender	1	5	0.5	-14.5
Deflection	0	0	0	0
Digdug	0	16.5	0	21.5
Donkeykong	0	7	0	12.5
Doorkoban	0	0.5	0.5	17
Dungeon	0	5.5	0	3.5
Eggomania	0	2	0.5	17.5
Eighthpassenger	0	-10	0	-10
Enemycitadel	0.5	2.5	0.5	2.5
Escape	0	0	0	0
Explore	1	1	1	1
Factorymanager	1	1	1	1
Firecaster	0	14	0	10
Fireman	0	-26	0	-26
Firestorms	0	0	0	0
Freeway	1	1	1	1
Frogs	0	0	0	0
Garbagecollector	0	2	0	3
Ghostbuster	0.5	4.5	1	82.5
Glow	1	1	1	1
Grow	1	7.5	1	7.5
Gymkhana	0	2	0	3
Hungrybirds	0	0	0	0
Iceandfire	0	1	0	1
Ikaruga	0	23	0.5	25
Infection	1	23	1	26
Intersection	1	1	1	1

Islands	1	1	1	1
Jaws	1	1066.5	1	1063.5
Killbillvol1	0	15.5	0	6.5
Labyrinth	1	1	0.5	0.5
Labyrinthdual	0	0	0	0
Lasers	0	0	0	0
Lasers2	0	0	0	0
Lemmings	0	-3	0	-5
Link	0	3650	0	3252
Mirrors	0	1	0	0.5
Missilecommand	1	3.5	1	3.5
Modality	1	1	1	1
Overload	0.5	10.5	0	8.5
Pacman	0	113.5	0	205.5
Pacoban	0	187	0	130
Painter	1	17	1	18
Plants	0	16	0	24.5
Plaqueattack	1	42.5	1	37
Pokemon	1	1	1	1
Portals	0	0	0	0
Racebet	1	1	1	1
Racebet2	1	1	1	1
Realportals	0	0.5	0	0.5
Realsokoban	0	2	0	2
Rivers	0	0	0	0
Roadfighter	1	1	1	1
Roguelike	0	0.5	0	1
Run	1	1	1	1
Seaquest	1	551	1	1065.5
Sheriff	1	8	1	8
Shipwreck	1	196.5	1	24.5
Slide	1	1	1	1
Sokoban	0	0	0	0
Solarfox	0	7	0.5	26

Superman	0	7	0	7
Surround	1	1	1	1
Survivezombies	1	8	1	8
Tercio	0	0	0	0
Thecitadel	0.5	2.5	0	0
Themole	0	5.5	0	3.5
Theshepherd	1	8	0	3
Thesnowman	0	0	0	0
Towerdefense	0	0.5	0.5	100.5
Vortex	0	0	0	0
Waitforbreakfast	0	0	0	0
Watergame	0	0	0	0
Waves	1	52	1	58
Whackamole	1	49.5	1	44.5
Wildgunman	1	8.5	1	10
Witnessprotected	0	0	0	0
Witnessprotection	0	9	0	9.5
Wrapsokoban	0	2.5	0	2.5
X-Racer	0	0	0	0
Zelda	0.5	6	0.5	5.5
Zenpuzzle	0.5	25	1	34

7.2 Appendix B


Agent initialisation phase (1000ms)
Playing at each game tick
Gather interesting features
Run A* to the most interesting feature
Run A* from current node
Store A* path
Compare expected reward of the best next state with actual reward of that state
Same?
Yes
Make the corresponding move
Run MCTS rollouts from current node
Does MCTS return a different state?
No
Yes
Move to new state
Have diverged from original A* path
Is current state a terminal state?
No
Yes
Endgame

8 References

[1] S. Russell and P. Norvig, "Artificial Intelligence A Modern Approach," ed, 2010.

[2] E. Yudkowsky, "Artificial Intelligence as a positive and negative factor in global risk," ed: Machine Intelligence Research Institute, 2008.

[3] N. Bostrom, *Superintelligence : paths, dangers, strategies*.

[4] O. Vinyals *et al.*, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II," ed.

[5] D. Perez-Liebana *et al.*, "The 2014 General Video Game Playing Competition," ed. IEEE Transactions on Computational Intelligence and AI in Games 8(3):1-1, January 2015, 2015.

[6] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," ed. 5th International Conference on Computer and Games, May 2006, 2006.

[7] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," ed. ECML 2006: European Conference on Machine Learning, 2006, pp. 282-293.

[8] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," vol. 529, ed. Nature, 2016, pp. 484-489.

[9] C. Browne *et al.*, "A survey of Monte Carlo tree search methods," ed. IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, March 2012, 2011.

[10] P. Drake and S. Uurtamo, "Move ordering vs. heavy playouts: where should heuristics be applied in Monte Carlo Go?," ed, 2007.

[11] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning,* vol. 47, no. 2-3, pp. 235-256, 05/2002 2002.

[12] D. J. N. J. Soemers, C. F. Sironi, T. Schuster, and M. H. M. Winands, "Enhancements for real-time Monte-Carlo tree search in general video game playing," ed: IEEE Conference on Computational Intelligence and Games (CIG), 2016.

[13] D. Perez-Liebana, M. Stephenson, R. D. Gaina, J. Renz, and S. M. Lucas, "Introducing real world physics and macro-actions to general video game AI," ed. IEEE Conference on Computational Intelligence and Games (CIG), 2017.

[14] D. R. Hofstadter, *Godel, Escher, Bach: An eternal golden braid*. Basic Books, 1979.

[15] W. G. Chase and H. A. Simon, "Perception in chess," *Cognitive Psychology,* vol. 4, pp. 55-81, 1973.

[16] F. Gobet and H. A. Simon, "Expert chess memory: revisiting the chunking hypothesis," *Memory,* vol. 6, pp. 225-255, 1998.

[17] D. H. Wolpert and W. G. Macready, "No Free Lunch Theorems for Optimisation," vol. 1, ed. IEEE Transactions on Evolutionary Computation, 1997.

[18] D. Ashlock, D. Perez-Liebana, and A. Saunders, "General video game playing escapes the No Free Lunch theorem," ed. 2017 IEEE Conference on Computational Intelligence and Games, 2017.

[19] S. M. Lucas, S. Samothrakis, and D. Perez, "Fast evolutionary adaptation for Monte Carlo tree search," ed. Esparcia-Alcázar A., Mora A. (eds) Applications of Evolutionary
Computation. EvoApplications 2014. Lecture Notes in Computer Science, vol 8602., 2014.

[20] A. M. Alhejali and S. M. Lucas, "Using genetic programming to evolve heuristics for a Monte Carlo tree search Ms Pac-Man agent," ed. 2013 IEEE Conference on Computational Inteligence in Games (CIG), 2013.

[21] D. Perez, S. Samothrakis, and S. Lucas, "Knowledge-based fast evolutionary MCTS for general video game playing," ed. 2014 IEEE Conference on Computational Intelligence and Games, 2014.
[22] H. Horn, V. Volz, D. Perez-Liebana, and M. Preuss, "MCTS/EA hybrid GVGAI players and game difficulty estimation," ed. IEEE Conference on Computational Intelligence and Games (CIG 2016), 2016.
[23] A. E. Elo, "The rating of chessplayers, past and present," ed: B. T. Batsford, London, UK, 1978.
[24] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, and S. M. Lucas, "Analyzing the robustness of general video-game playing agents," ed. Published proceedings: IEEE Conference on Computatonal Intelligence and Games, CIG, 2017.
[25] I. Bravi, D. Perez-Liebana, S. M. Lucas, and J. Liu, "Shallow decision-making analysis in general video game playing," ed. 2018 IEEE Conference on Computational Intelligence and Games (CIG), 2018.
[26] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, "A portfolio approach to algorithm selection," *IJCAI International Joint Conference on Artificial Intelligence,* 2003.
[27] K. A. Smith-Miles, "Cross-disciplinary perspectives on meta-learning for algorithm selection," ed. ACM Computing Surveys, Vol. 41, No. 1, Article 6, 2008.
[28] A. Tversky and D. Kahneman, "Judgement under uncertainty: heuristics and biases," *Science,* vol. 185, no. 4157, pp. 1124-1131, 1974.
[29] K. A. Dowsland, E. Soubeiga, and E. K. Burke, "A simulated annealing based hyperheuristic for determining shipper sizes for storage and transportation," *European Journal of Operational Research,* vol. 179, no. 3, pp. 759-774, 16 June 2007 2007.
[30] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu, "A graph-based hyper-heuristic for educational timetabling problems," *European Journal of Operational Research,* vol. 176, pp. 177-192, 2007.
[31] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward, "A classification of hyper-heuristic approaches," *Handbook of Metaheuristics,* vol. 146, pp. 449-468, 2010.
[32] A. Mendes, J. Togelius, and A. Nealen, "Hyper-heuristic general video game playing," ed. 2016 IEEE Conference on Computational Intelligence and Games (CIG), 2016.
[33] A. T. Alexandru, "AI agent for general video games," p. 5, 2018.
[34] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics,* vol. 18, no. 1, pp. 50-60, 1947.
[35] S. community. (2019, 16/04/2019). *SciPy ranksums*. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ranksums.html
[36] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An Anytime, Replanning Algorithm," in *ICAPS*, 2005, vol. 5, pp. 262-271.
[37] Y. Bjornsson, V. Bulitko, and N. R. Sturtevant, "TBA*: Time-bounded A*," *Proceedings of the 21st International Joint Conference on Artificial Intelligence,* 2009.