Fast Approximate Max-n Monte-Carlo Tree Search for Ms Pac-Man

Spyridon Samothrakis, David Robles, and Simon Lucas, *Senior Member, IEEE*

***Abstract*—We present an application of Monte-Carlo Tree Search (MCTS) for the game of Ms Pac-Man. Contrary to most applications of MCTS to date, Ms Pac-Man requires almost real-time decision making and does not have a natural end state. We approached the problem by performing Monte-Carlo tree searches on a 5 player max^n tree representation of the game with limited tree search depth. We performed a number of experiments using both the MCTS game agents (for pacman and ghosts) and agents used in previous work (for ghosts). Performance-wise, our approach gets excellent scores, outperforming previous non-MCTS opponent approaches to the game by up to two orders of magnitude.**

***Index Terms*—pac-man, MCTS, max-n, Monte-Carlo**

I. Introduction

With the recent success of Monte-Carlo Tree Search (MCTS) in Computer Go [1], [2] and General Game Playing (GGP) [3], there has been a growing interest in applying these techniques to other board-like strategy games such as Hex [4]. However, there has been very limited research in the application of MCTS in real time video games. In this paper we investigate the use of Monte-Carlo Tree Search in the game *Ms Pac-Man*, a predator/prey-like game popularised in the 80s. From an AI research perspective, there are many aspects of the game worthy of study, but in this particular paper we are mostly focused on the performance of MCTS in a real time game setting. The rest of the paper is organised as follows: in the next section (section II), we discuss the game of Ms Pac-Man and the previous research on this game; in Section III, we describe our Ms Pac-Man simulator and the experimental setup done in order to perform simulations on the game; in section IV we discuss MCTS in more depth together with some of its previous applications; in section V we present the methodology and rationale used in order to create the current set of agents. In section VI we perform four sets of experiments and present the results. Finally, section VII summarises the contributions of this research, and gives directions for future developments. Throughout the text as a convention, we use the word "pac-man" to refer to the pac-man agent, while "Ms Pac-Man" and "Pac-Man" (capitalised first letter) refer to the actual game.

II. Ms Pac-Man

Pac-Man is a classic arcade game originally developed by Toru Iwatani for the Namco Company in 1980. Since its release in Japan, the game became immensely popular, rapidly achieving cult status, and various other versions followed. The best known of these is Ms Pac-Man, released in 1981, which many see as being a significant improvement over the original. Ms Pac-Man introduced a female character, new maze designs and several gameplay changes over the original game. Probably the most important change was the introduction of an element of randomness to the ghosts' behaviour which eliminates the effectiveness of exploiting set patterns or routes, making Ms Pac-Man much more of a game of short-term planning and reactive skill than memorisation. While it is no longer the newest or most advanced example of game development, Pac-Man style games still provide a platform that is both simple enough for AI research and complex enough to require intelligent strategies for successful gameplay. The current Ms Pac-Man's world record is held by Abdner Ashman with $921,360$ points. To put this score in perspective, more than 130 stages were completed in an almost perfect fashion[1]. On the other hand, the AI agent who holds the world record is ICE Pambush [5], who won the competition held in the Ms Pac-Man Competition 2009 IEEE Symposium on Computational Intelligence and Games with a maximum score of $30,010$. There is an enormous difference between the human world record and the agent world record, which indicates that computers are significantly behind the human players and there is much room for improvement with AI techniques. Note that this score of $30,010$ was achieved in the original Ms Pac-Man, using a screen-capture software agent; this will be further discussed in the next section.

Since the source code of the original Ms Pac-Man game is in Z80 assembly language, most of the research done in Pac-Man style games is done with simulators. These vary in how closely they resemble the original game, both functionally and cosmetically, but all the ones that do not use the original assembly code have some significant differences when compared to the original. For this reason, it is hard to make comparisons between agents created by other researchers, who use their own simulators.

A. Ms Pac-Man Gameplay

The player starts in maze A with three lives, and a single extra life is awarded at $10,000$ points. There are four mazes in the game (A, B, C and D), and each one contains a different layout with pills and power pills placed on specific nodes. The goal of the player is to obtain the highest possible score by eating all the pills and power pills in the maze and continue

Spyridon Samothrakis, David Robles and Simon M. Lucas are with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, United Kingdom. (emails: ssamot@essex.ac.uk, davidr@essex.ac.uk, sml@essex.ac.uk).

[1]http://uk.gamespot.com/arcade/action/mspac-man/news.html?sid=6130815

to the next stage. The difficulty of clearing the maze comes from the ghosts. There are four different ghosts in Ms Pac-Man: Blinky (red), Pinky (pink), Inky (cyan) and Sue (yellow). At the start of each level the ghosts start in their lair in the middle and spend some idle time before exiting the lair to chase the pac-man. The time spent in the lair before joining the chase varies with the level, with the ghosts leaving the lair sooner on the harder levels of the game. Each time the pac-man is eaten by a ghost, a life is taken away and pac-man and the ghosts return to their original positions on that maze.

There are four power pills in each of the four mazes, which when eaten, reverse the direction of the ghosts and turn them blue, which mean they can be eaten for extra points. The score for eating each ghost in succession immediately after a power pill (which is worth 50 points) starts at 200 and doubles each time. So, an optimally consumed power pill is worth 3050 (= 50 + 200 + 400 + 800 + 1600). Note that if a second power pill is consumed while some ghosts remain edible from the first power pill consumption, then the ghost score is reset to 200. The more difficult the level, the shorter that time becomes, until the ghosts do not turn blue at all (but they do still change direction). When the ghosts are flashing, they are about to change from blue state back to their normal state, so the player (or agent) should be careful in these situations to avoid losing lives. Another source of points are the extra fruits and prizes that appear and bounce around the maze. These bonus items include cherries, strawberries, peaches, pretzels, apples, pears and bananas, and their values increase with increasing levels of the game.

As previously mentioned, when all the pills and power pills are cleared in a maze, the next maze appears, though with increased speed and difficulty. Also, ghosts travel at half speed through the side escape tunnels which allows pac-man to gain some breathing space, at the risk of being trapped in the tunnel. Another detail of the first stages is that pac-man moves faster when not eating pills than when she is. She is also capable of turning around corners faster than the ghosts, so should make as many turns as possible when the ghosts are on her tail.

B. Previous Research

In recent years, Pac-Man style games have received some attention in Computational Intelligence research. Previous work in Pac-Man has been carried out in different versions of the game (i.e. Pac-Man and Ms Pac-Man simulators), hence an exact comparison is not possible, but we can have a general idea of the performances. Most of these works can be divided in two areas: agents that use CI techniques partially or completely, and controllers with hand-coded approaches. We will briefly discuss the most relevant works in both areas.

One of the earliest studies with Pac-Man was conducted by Koza [6] to investigate the effectiveness of genetic programming for task prioritisation. This work utilised a modified version of the game, using different score values for the items and also a different maze. According to Szita and Lorincz [7], the only score reported on Koza's implementation would have been around 5,000 points in their Pac-Man version. Bonet and Stauffer [8] proposed a reinforcement learning technique for the player, using a very simple Pac-Man implementation. They used a neural network and temporal difference learning (TDL) in a 10 x 10 centred window, but using simple mazes with only one ghost and no power pills. Using complex learning tasks, they showed basic ghost avoidance.

Gallagher and Ryan [9] used a Pac-Man agent based on a simple finite-state machine model with a set of rules to control the movement of Pac-Man. The rules contained weight parameters which were evolved using the Population-Based Incremental Learning (PBIL) algorithm. They ran a simplified version of Pac-Man with only one ghost and no power pills, which reduces the scoring opportunities in the game and removes most of the complexity. However this implementation was able to achieve machine learning at a minimum level. Gallagher and Ledwich [10] described an approach to developing Pac-Man playing agents that learn game play based on minimal screen information. The agents were based on evolving neural network controllers using a simple evolutionary algorithm. Their results showed that neuro-evolution is able to produce agents that display novice playing ability with no knowledge of the rules of the game and a minimally informative fitness function.

Szita and Lorincz [7] proposed a different approach to playing Ms Pac-Man. The aim was to develop a simple rule-based policy, where rules are organised into action modules and a decision about which direction to move in is made based on priorities assigned to the modules in the agent. Policies are built using the cross-entropy optimisation algorithm. The best-performing agent obtained a score average of 8,186, comparable to the performance of a set of five human subjects played on the same version of the game, who averaged 8,064 points. More recently, Wirth and Gallagher [11] used an agent based on an influence map model to play Ms Pac-Man. The model captures the essentials of the game and showed that the parameters of the model interact in a fairly simple way and that it is reasonably straight-forward to optimise the parameters to maximise game playing performance. The performance of the optimised agents averaged a score of just under 7,000.

The work reported in the papers listed above used nearly as many different simulators as there are papers listed. Unfortunately this makes a meaningful comparison of the abilities of the agents extremely difficult — it is just possible to get a vague idea of the relative playing strengths of these approaches but no more than that.

For this reason one of the authors (Lucas) has developed a screen-capture based competition that uses the original rom-code of the Ms Pac-Man game running on an emulator. The competition has been run at several conferences over the last few years since 2008. Entrants submit a software agent that plays the game by capturing the screen many times per second, processing the captured pixel map to identify the main game objects (pac-man, ghosts, pills etc.) and then generating keyboard-events which are sent to the game window. This allows a standard way to measure the performance of the software agents, and has the benefit of testing performance on the original game rather than some modified (and usually significantly simplified) clone. However, there are a few caveats to this. The performance of the agent can be significantly

impaired by a poor screen-capture and processing kit, and while a specific kit could be specified, none of the existing ones seem to be good enough yet to standardise on. Therefore, this is currently left open as a choice to the entrants in order to promote the development of better kits. For this reason we have not yet tested our agent in screen-capture mode, but this is a priority for future work; to be done properly we are developing a better screen-capture kit that will enable our MCTS agent to show its true ability. Hence, for the moment we base our result on a simulator developed by the authors and freely available for download for the Agent versus Ghost Team Ms Pac-Man competition[2] (originally developed by Lucas [12] and then significantly enhanced).

The papers described next all use some version of this simulator, using the same ghost algorithms (known collectively as *LegacyTeam*) unless otherwise stated. The earlier papers used a single maze (the first maze) version of the game, with the same edible time for each level (hence the game consists of endless repetitions of the first level until all the lives are lost). The later papers (as explained) have also used the full four original mazes, and in some cases reducing the edible time, or (as in the current paper) using no edible time at all. The simulator is described in some detail in the next section.

Lucas [12] proposed evolving neural networks to evaluate the moves. He used Ms Pac-Man because the ghosts behave with pseudo-random movement, and that eliminates the possibility of playing the game using path-following patterns. This work utilised a handcrafted input feature vector consisting of the distances from pac-man to each non-edible ghost, to each edible ghost, to the nearest pill, to the nearest power pill and to the nearest junction. His implementation was able to score an average of $4,780$ points over 100 games.

Robles and Lucas [13] proposed perhaps the first attempt to apply tree search to Ms Pac-Man. The approach taken was to expand a route-tree based on possible moves that the pac-man agent can take to depth 40, and evaluated which path was best using hand-coded heuristics. On their simulator of the game the agent achieved a high score of $40,000$, but only around $15,000$ on the original game using a screen-capture interface. However, it should be emphasised that many of the above quoted scores have been obtained on the different Pac-Man simulators, and therefore only provide the very roughest idea of relative performance.

Burrow and Lucas [14] used a previous version of the simulator to analyse the learning behaviours of TDL and Evolutionary Algorithms, with a simple TD(0) and a standard ES(15 + 15) respectively. Two features were used as input to the function approximators (interpolated table and MLP) to give an estimate of the value of moving to the candidate node: 1) the distance from a candidate node to the nearest escape node, and 2) the distance from a candidate node to the nearest pill along the shortest maze path. Their experiments showed that under that experimental configuration evolution performed significantly better than TDL, although other reward structures or features were not tested.

Also using a previous version of the simulator, Alhejali and Lucas [15] used genetic programming (GP) to evolve a wide variety of pacman agents. A diverse set of behaviours were evolved using the same GP setup in three different versions of the game: 1) single level, 2) four mazes, and 3) unlimited levels. All of them using only one life and the same GP parameters. The function set was created based on Koza [6], including operations such as IsEdible, IsInDanger, IsEnergizersCleared, etc. On the other hand, the terminal set was divided in two groups: data-terminals and action terminals. Most of the data terminals returned the current distance of a component from the agent, such as DISPill, DISEnergizer, DISGhost, etc. As for the action-terminals, each one moved Pac-Man one step toward the target, e.g., ToEnergizer, ToPill, and ToEdibleGhost. The results showed that GP was able to evolve controllers that are well-matched to the game used for evolution and, in some cases, also generalise well to previously unseen mazes. The average score in the experiments was approximately $10,000$ points, with a maximum score of approximately 20,000.

III. Ms Pac-Man Simulator

The Ms Pac-Man simulator used for this research is a re-factored and extended version of the one used in Lucas [12], which is written in object-oriented style in Java with a reasonably clean and simple implementation. Compared to the original several changes and improvements have been made in order to perform Monte-Carlo Tree Search and also to enable much faster path evaluation. This current version of the simulator is a more accurate approximation of the original game, not only at the functional but also at the cosmetic level, and includes the four original mazes. Figure 1 shows a screen shot of each level in action. Nevertheless, there are still important differences with respect to the original game:

- The speed of our pac-man and the ghosts are identical, and pac-man does not slow down to eat pills.
- Our pac-man cannot cut corners, and so has no speed advantage over the ghosts when turning a corner.
- Our ghosts do not slow down in the tunnels.
- Bonus fruits are not present, as their inclusion plays a relatively minor contribution to the score of the game (at least at lower levels).
- No additional life at 10,000 points. This would have been easy to implement, but has little bearing on MCTS-based strategies.
- Perhaps the most significant of all: the ghost behaviours are not the same as in the original game. We developed various ghosts agent controllers with different strategies to use in the simulations, however, they are only an approximation to the original behaviours.

The mazes of the game are modelled as graphs of connected nodes. Each node has two, three or four neighbouring nodes depending on whether it is in a corridor, L-turn, T-junction or a crossroads. After the mazes have been created, a simple efficient algorithm is run to compute the shortest-path distance between every node and every other node in the mazes. These distances are stored in a look-up-table, and allow fast computation of the various controller-algorithm input features.

[2]http://csee.essex.ac.uk/staff/sml/pacman/kit/AgentVersusGhosts.html

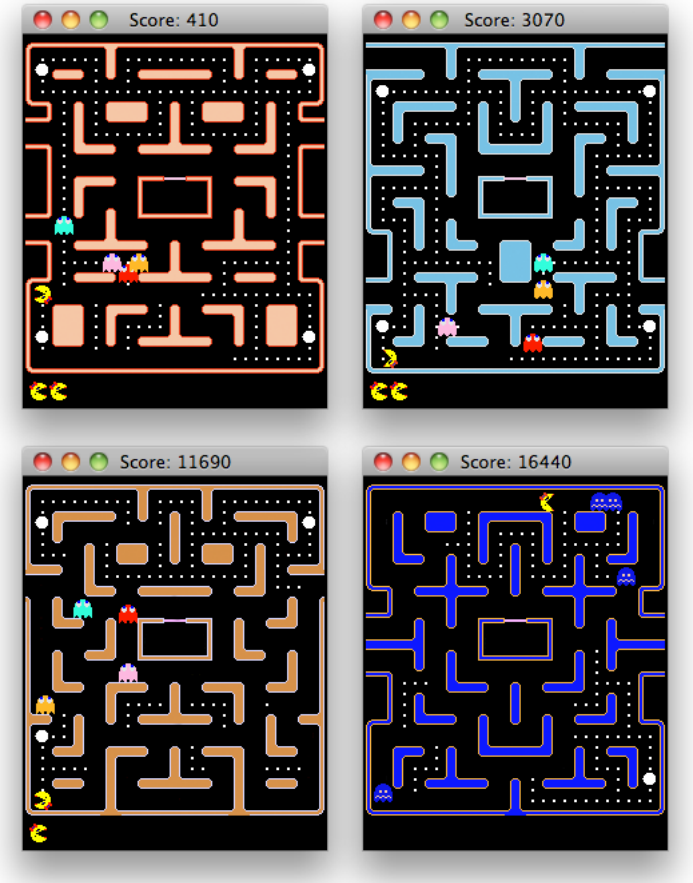


Fig. 1. Snapshots of the four mazes in the Ms Pac-Man Simulator.

Each maze is played twice consecutively, starting in Maze A up to Maze D. When Maze D is cleared, the game goes back to Maze A and continues the same sequence, i.e., (A, A, B, B, C, C, D, D, A, A,...) until game over.

The new design of the simulator allows us to implement AI controllers for the pac-man and the team of ghosts as well, by implementing their controller interfaces: AgentInterface (Listing 1) and GhostTeamController (Listing 2) respectively. Each interface has a single method which takes the game state as input and returns the desired movement directions.

Listing 1. Agent Interface
```
public interface AgentInterface {
    int action(GameStateInterface gs);
}
```

Listing 2. Ghosts Controller
```
public interface GhostTeamController {
    public int[] getActions(GameStateInterface gs);
}
```

During the game, the model of the game will send the GameState to both controllers, and they must return the appropriate actions to take. In the case of the AgentInterface (pac-man) it will return the next direction to take, whereas the Ghosts Controller must return an array with the *desired* directions to move each ghost, and they will be executed as long as they are legal moves for the ghosts. The pac-man is allowed to move in any direction along a corridor at any given time, while the ghosts cannot reverse unless the model of the game determines a ghosts reversal. For example, in Figure 3 the state of the game is on time step 25, s_{25}, in which the action set for pac-man and Blinky (Blinky is the ghost in the bottom left of the maze in Figure 3) are:

$$\mathcal{A}_P(s_{25}) = \{\texttt{north}, \texttt{south}\}$$
$$\mathcal{A}_B(s_{25}) = \{\texttt{north}\}$$

If one of the agents chooses an action that is not part of the respective set of actions, $\mathcal{A}_p(s_{25})$, the simulator will take the default behaviour of the agent.

The GameStateInterface (Listing 3) provides an approximation to a Markov state signal that summarises everything important about the preceding moves that contributed to the current state. Some of the information about the previous moves is lost, but most that really matters for decision-making is retained. Apart from the information of the current state of the game, the GameStateInterface also provide methods that ease the use of Monte-Carlo simulations: copy() and next(). The method copy() returns a copy of the current state of the game. This is helpful while doing game-tree search, since it is necessary to keep copies of the game state in every node. The next() method takes the current game *state*, $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states, and receives a set of *actions* to take for each of the five agents (pac-man and the ghosts), $a_{p,t} \in \mathcal{A}_p(s_t)$ where $\mathcal{A}_p(s_t)$ is the set of actions available in state s_t for agent p. One time step later, as a consequence of their actions, the agents find themselves in a new state, s_{t+1}.

Listing 3. Game State Interface that enables MCTS
```
public interface GameStateInterface {
    GameStateInterface copy();
    void next(int pacDir, int[] ghostDirs);
    Agent getpac-man();
    MazeInterface getMaze();
    int getLevel();
    BitSet getPills();
    BitSet getPowers();
    GhostState[] getGhosts();
    int getScore();
    int getGameTick();
    int getEdibleGhostScore();
    int getLivesRemaining();
    boolean agentDeath();
    boolean terminal();
    void reset();
}
```

The simulator represents the game state in a compact way in order to allow efficient copying of the state, and efficient transmission to remote clients (enabling remote evaluation of ghost teams and agents). The game state contains only references to other immutable objects where possible. For example, the maze object never changes, but the positions and states of the agents change, and the state of the pills change. Bit-sets are used to store whether each pill is present or has already been eaten, enabling the entire state of the game to be represented in a few tens of bytes.

IV. Monte Carlo Tree Search

A. MCTS and Monte-Carlo Tree Search

The idea behind Monte-Carlo algorithms in AI is that the approximation of future rewards (as they are understood in the Markov Decision Process (MDP) sense [16]) can be achieved through random sampling. What this effectively means is that the agent extrapolates to future states in a random fashion and moves to the state with the highest predicted reward. MCTS tries to rectify some of the issues that come with such an approach by combining it with a tree and effectively creating a stochastic form of best-first search. From a game theoretic perspective, the tree is a subtree of the game tree in extensive form [17].

Upper Confidence Bounds in Trees (UCT) (presented in Algorithm 1) can be seen as simply an implementation of MCTS where the "selection" part of the algorithm is provided by ideas borrowed from the study of multi-armed bandit [18] problems. In UCT, each node in the tree is seen as multi-armed bandit. The goal of the search is to "push" more towards areas of the search space that seem more promising. Although there are many different versions of the algorithm, the one presented in [19] is quite commonly used[3]. The algorithm (see Figure 2) can be summarised as follows; starting from the root node, expand the tree by a single node. If the node is a leaf node, estimate its value by performing a roll-out and back-propagate the value to the node's ancestors in the tree. If the node is not a leaf node, keep exploring the tree until one is reached. The most commonly used back-propagation strategy is one that makes direct use of the underlying tree, e.g., for a minmax (negamax) tree, that would include subtracting or adding the result depending on who is the owner of each node in the ancestor list (for an example see Algorithms 1, 2, 3, taken from [19]).

In case all possible nodes have been visited, a common way to distinguish which node to explore further is to assign a value to each node based on the Chernoff-Hoeffding bound. This leads to what is known as the UCB1 policy[18]. Play arm j that maximises

$$\bar{x_j} + C\sqrt{\frac{ln(n)}{n_j}} \tag{1}$$

The symbols $\bar{x_j}$ in Equation 1 denote the average reward from the underlying bandit, and it is the exploitation part of the algorithm. The second part of the above equation is the exploration part. C is a constant (often set to $\sqrt{2}$) [18], n is the sum of all trials and n_j is the number of trials for the j bandit. Finally, $\sum_{j=0}^{j=j_{max}} n_j = n$.

The equation to choose which arm to play (in the case of a tree search which child to follow) can be heavily tuned depending on the underlying distribution. UCB1 (see Algorithm 2 for the pseudocode) should be seen as the "lowest common denominator" policy. If no information about the bandits is available to us, this is probably the correct policy to use. On the other hand, once we have collected enough information, more informed policies should be able to perform better.

[3]Although many leading MCTS Go programs now use very different node-selection formulas based on various heuristics.

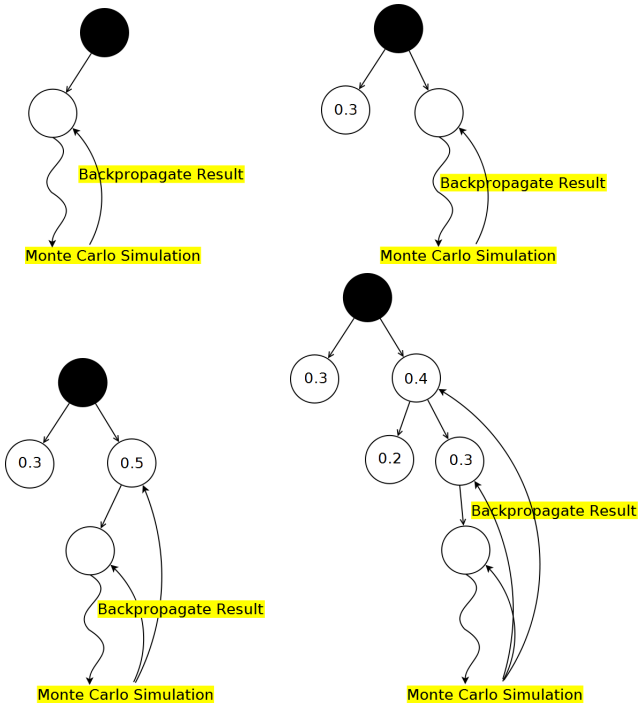


Fig. 2. A sample MCTS search.

Algorithm 1 playOneSequence(rootNode)

{The entry point of MCTS. It traverses through a tree in an asymmetric fashion. End state rewards are provided as an array (*rewardsArray*), as we can have more than two players.}
node[0] ← rootNode
i ← 0
repeat
  nodeArray[i+1] ← ***descendByUCB1***(nodeArray[i])
  $i \leftarrow i + 1$
until nodeArray [i] is a terminal Node
updateValue(nodeArray, nodeArray[i].rewardsArray)

Note that events in a tree are not independent, so algorithms that would naturally work for bandit problems need some adaptation when applied to tree search. Advantages of MCTS include the fact that it explores the tree asymmetrically and that it gives a natural way to handle uncertainty.

B. MCTS in Games

The popularity of MCTS stems primarily from the fact that it revolutionised Computer Go [19], to the point where computer players actually became competitive against human players (e.g. [20], [21], [22]). Its success lead to widespread acclaim of Monte-Carlo methods, eventually reaching popular media [23]. Example implementations include ($CRAZYSTONE$ [24], $MANGO$ [25], $MOGO$ [19] and $FUEGO$ [26]).

Algorithm 2 descendByUCB1(node)

```
{We now descend through the tree according to UCB1 policy, making this a UCT algorithm. This part of the algorithm can be modified to suit the particular problem (e.g. use UCB-TUNED) }
nb ← 0
for i ← 0 to node.childNode.size() - 1 do
  nb ← nb + node.childNode[i].nb
end for
for i ← 0 to node.childNode.size() - 1 do
  if node.childNode[i].nb = 0 OR MAX_DEPTH_REACHED then
    v[i] ← ∞
  else
    v[i] ← 1.0 - node.childNode[i].value / node.childNode[i].nb + sqrt(2 * log(nb) / node.childNode[i].nb)
  end if
end for
index ← argmax(v[j])
return node.childNode[index]
```

Algorithm 3 updateValue(nodeArray, rewardsArray)

```
{Finally, we can update the values of each node in the path between the newly added node and the root, stored on nodeArray (i.e. the ancestors of the newly added node). Note that each node has a corresponding rewardId. The vector of rewards should be provided by the game.}
nb ← 0
for i ← nodeArray.size() - 2 to 0 do
  value ← rewards[nodeArray[i].rewardId]
  nodeArray[i].value ← nodeArray[i].value + value
  nodeArray[i].nb ← nodeArray[i].nb + 1
end for
```

As a result, MCTS has been applied already to a large number of games [27], [28], [29]. For the most part, the non-Go papers failed to replicate the burgeoning success of MCTS in Go. The area that was identified for improvement [27] was mostly around the concept of doing good Monte-Carlo simulations. Being a best-first search, MCTS relies heavily on the quality of Monte-Carlo simulations, and its performance is greatly affected by them. For example, Gelly et al. [19] report a big boost compared to purely random simulations (from 1647 to 2200 ELO), which can be increased even further with further heuristics of a more general nature like RAVE[30]. Another big issue with the non-Go implementations of MCTS is the lack of comparison with the state of the art. As a consequence there is no way of understanding how well MCTS did compared to other methods.

MCTS is currently very successful in General Game Playing (GGP) [3], a domain that practically prohibits the use of strong heuristics. While General Game Playing in this sense is not fully *general* since it refers to a subset of perfect and complete games, it nevertheless shows that MCTS has the potential of achieving good results in diverse domains.

Finally, there have been some developments for MCTS in some real time video games, mainly from the the perspective of real time strategy games (e.g. [31], [32], [33]). Most real time strategy games harbour an element of imperfection, which is however discarded in these studies, with the results however being exceptionally strong.

V. Methodology

A. Applying MCTS

There has been no "standard" process for applying MCTS, so we are proposing an empirical four step process. The first step is to understand the number of agents and the information content of each game and choose the right tree. For games of complete and perfect information (e.g. chess, Go), a *min-max* tree is commonly used. For games of incomplete but perfect information (e.g. backgammon), *expectimax* trees should be used. Finally for games of imperfect and incomplete information (e.g. poker), *miximax* trees were recently introduced [34].

All the above cases apply naturally to 2 or 2.5 player games. In the case of N-Player games, one can easily extend the algorithm to either max^n [35] or one of its possible variations. The basic principle behind these algorithms is that each player tries to maximise its payoffs independently from the rest.

The second step is understanding the underlying distribution of each arm and tuning the policy equation. This can be done in a number of ways, which can range from tuning Equation 1, to completely replacing it with something that captures the underlying probabilities better. In our case we use the algorithm UCB-TUNED [18] (see Equation 2), which seems to be fairly common in the literature, and initial short runs showed that it works better in our case than UCB1 (although we did perform some experiments with UCB1 (see Equation 1) for comparison purposes, see the experiments section).

$$\bar{x_j} + \sqrt{\frac{ln(n)}{n_j} min\left\{1/4, \bar{x_j^2} - \bar{x_j}^2 + \sqrt{\frac{2ln(n)}{n_j}}\right\}} \quad (2)$$

The third step is to come up with a back-propagation policy. Our strategy is the one used by default in the original Go MCTS implementation and presented in the previous section, with a max^n adaptation [35]. In this case, the end node provides the algorithm with n rewards. Each node in the tree has an associated $rewardId$ and one of the rewards is added accordingly. To put it in another way, each end node has an associated vector of reward $\mathbf{r}$ with as many elements as agents in the game (in our case 5). Each node however has just one reward, based on which agent it belongs to.

The final step is to augment the algorithm with knowledge and/or "guide" the Monte-Carlo simulations. In Go this is achieved by using local patterns [19], which significantly improves the quality of the simulations. In our case a set of heuristics is created, and presented in the next subsection.

B. MCTS on Pac-Man: Problems

The first thing one should notice in Pac-Man is that the game (like most video games, and arguably life) does not easily

converge into winning final state for the pac-man player, which creates a problem for all kinds of tree search algorithms. This problem stems from two facts. On one hand pac-man has an almost infinite number of back-and-forth moves it can do, with no ending apart from it dying. On the other hand, the game can keep going on forever, with pac-man progressing from maze to maze, with no final winning position.

The second issue with the game is that it has a strong timing element; whatever moves one has to perform at each state, it should be calculated in less than $50-60ms$. Although we do not adhere strictly to real-time solutions in this paper, as it is understood that further improvements in code quality and CPU speed can easily bring performance within bounds, nevertheless we tried to avoid timing settings that can never lead to real time game play.

C. Max^n approach to Ms Pac-Man

Ms Pac-Man is a real-time computer game where events in the game occur at the same rate as the events which are being depicted. For instance, one minute of play of the game depicts one minute of character movement. Despite its real-time simultaneous-move nature, the game has simple rules and can be transformed to a turn-based game to apply MCTS directly. There are many ways one can model Pac-Man for MCTS. In our approach, we chose to model Pac-Man as a 5-player game, and base the tree on max^n. Pac-Man is a simultaneous move game, at least theoretically speaking, none of the min-max like trees is really applicable, and one should be searching for mixed strategies (at least for the endgames). As seen recently by [36] however, in practice the strategic advantage of taking this into account is trivial and can be safely ignored.

In order to solve the problem of not having a natural end state, we artificially limit the search tree to a fixed depth. We also restrict pac-man's ability to move back and forth as it pleases within a single tree, making its behaviour similar to a ghost. This (implicitly) creates a number of paths that a search can easily evaluate. Just to clarify this last point, our MCTS pac-man agent is able to move back and forth at each time step (for example, while waiting to see which path a ghost will take), but when expanding each MCTS tree these step-by-step direction reversals are not allowed in order to restrict the growth of the tree.

In that respect, an end node can be either the natural end of the game (a ghost eats pac-man) or the end of a tree, with $TREE\_DEPTH = c$. However, intuitively this would result in pac-man running around the maze, trying to avoid ghosts, since finding a natural end state (e.g. eating all the pills in the maze) is beyond the tree depth, at least at the beginning of the game. In order to avoid this scenario, we have created a function called $gpn(m,p)$ (which stands for "game preferred node"). This function assigns the binary value of one to a certain node in the tree path, and leaves the rest of the tree to zero. That way we can set a target for the search (and we will see how we use this target in the next section).

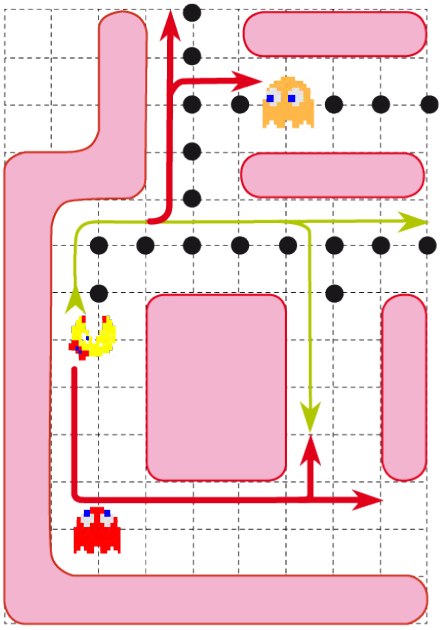


Fig. 3. Example of path creation

$$gpn(m,p) = \begin{cases} node_1, & \text{if } c(m,p) = 1 \\ node_2, & \text{if } c(m,p) = 2 \\ node_3, & \text{if } c(m,p) = 3 \end{cases} \quad (3)$$

$$c(m,p) = \begin{cases} 1, & \text{if } \exists g_i : Ed(g_i) \wedge dpg(m,p) < G \\ 2, & \text{if } \exists! g_i : Ed(g_i) \wedge dpn(m,p) < P \\ 3, & \text{otherwise} \end{cases} \quad (4)$$

$$\begin{aligned} node_1 &= \arg\min_n \mathbf{d_2}(p, g_i) \\ node_2 &= \arg\min_n \mathbf{d_2}(p, n_{xy}) \\ node_3 &= \arg\min_n \mathbf{d_2}(n^p_{xy}, node_2) \end{aligned} \quad (5)$$

$$dpg(m,p) = \mathbf{d_2}(\arg\min_n \mathbf{d_2}(p, g_i), p) \quad (6)$$

$$dpn(m,p) = \mathbf{d_2}(\arg\min_n \mathbf{d_2}(p, n_{xy}), p) \quad (7)$$

In Equation 4 m, is the current maze, p is the pac-man location, g_i is one ith ghost, n is a node, d_2 is the pre-calculated shortest path distance, n^p is one of the nodes that surround pac-man and P is the furthest away distance MCTS can see given its maximum depth size (usually set to $(MCTS.TREE\_DEPTH/5 - 1)$. Function $Ed(g_i)$ returns if a ghost is edible. Equations 6 and 7 specify the distance from the closest ghost and the distance from closest node respectively. The function $c(m,p)$ returns a different value depending on which of three specific cases apply. Case 1 is when there is an edible ghost in the map and the distance to the ghost is smaller than $G = 20$. Case 2 is when there are no edible ghosts and the distance of the closest pill/power is smaller than P. Case 3 is the catch-all for when there are no edible ghosts and the nearest pill is outside the search depth of the tree. In each case, a corresponding node is marked as a

"target" node using Equation 3 (and consequently equation 5). In the first case the nearest ghost is the target, in the second case the nearest pill, while in the third case a node en route to the nearest pill. Note that case three exists purely because it is frequently the case that pac-man will find itself in a position where all the natural target nodes are beyond the search depth. This solves the problem by effectively making a node within the search tree a target node.

In the natural endgame scenarios where the end of the maze is reached or pac-man gets eaten, rewards are set up in an obvious fashion. In the case where the maze is cleared pac-man gets a reward of 1, while all ghosts get a reward of 0. In case pac-man dies, the ghosts get a reward of 1, while pac-man gets a reward of 0. In all other cases, the rewards for the ghosts are set proportionally to the inverse distance between the ghost and pacman. Thus:

$$r(g_i) = \begin{cases} 0.5/((\mathbf{d_2}(g_i, p) + 1)), & \text{if pac-man not eaten} \\ 1, & \text{if pac-man eaten} \end{cases} \quad (8)$$

In Equation 8, first part, the rewards are set so that the closer a ghost is to pac-man the higher the reward is. This makes minimal impact when the ghosts are close, where rewards are primarily acquired through pac-man's death, but it makes a difference when the ghosts are too far away and cannot reach pac-man's node within their tree search.

Now, for pac-man the reward function is:

$$r(p) = \begin{cases} 1.0, & \text{if last pill in map is eaten} \\ 0.8, & \text{if preferred node } gpn(m,p) \text{ is hit} \\ 0.0, & \text{if pac-man dies} \\ 0.6, & \text{otherwise} \end{cases} \quad (9)$$

Hitting the preferred node (Equation 9) means that in the current path evaluated by MCTS, the preferred node was accessed and it is part of the search.

The above reward equations are used both by the MCTS pac-man agent and the MCTS ghost team. Thus, in all cases the reward vector is:

$$\mathbf{r} = [r(p), r(g_1), r(g_2), r(g_3), r(g_4)] \quad (10)$$

In the ghost team however, the first ghost does NOT follow the action proposed by MCTS, but rather follows the route that minimises distance between itself and the pac-man agent ($\mathbf{d_2}(g_i, p)$)).

In almost every equation presented above there is some ad hoc variable. For example pac-man receives a reward of 0.8 when a preferred node is hit. The values for these variables are a result of short trial and error experimentation and are part of the heuristics. There are some obvious criteria involved (e.g. a reward for a preferred node should be higher than the reward for a non-prefered node - 0.8 Vs 0.6 in our case), however there is no "hard" scientific justification for these values, as there is not for the equations themselves; they are part of the chosen set of heuristics.

VI. Experiments

We performed four sets of experiments. The first three experiments are meant to portray the idiosyncrasies of the specific MCTS implementations and how heuristics, tree search depth and the number of simulations affect our agents. The fourth and final experiment is meant to demonstrate the strength of the MCTS approach compared to previous approaches in this area using a version of the simulator used in previous published research [12], [14].

In each set of experiments a number of different individual experiments are performed. At first we progressively increase the number of simulations from the set $\{100, 200, 300, 400\}$ and vary the depth of the search tree from 100 to 450 in increments of 50. We perform two experiments like this, each one with a different UCB function ($UCB1$ and $UCB-TUNED$). We also perform a run where we vary the tree depth from 100 to 950, but this time we fix the time in milliseconds the agent has before he has to perform a move. We test for the set of $\{20, 30, 40, 50, 60\}$ms. Here it is important to note that it takes almost half a second[4] to have an agent performing 400 iterations in a tree of 400, making these kinds of setups prohibitive for real time simulations. However, the results are still of interest, as CPU speed is increasing by the day and a multi-threaded setup can easily bring the results within the bounds of real time game play.

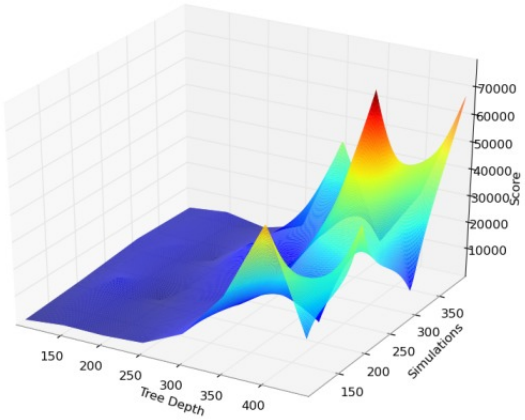
In the first three experiments the edible time of the ghosts is set to 3 (almost non-existent). There are no random ghost reversals and the each agent has one life. This way we hope to make it as hard as possible for the pac-man agent, thus minimising the time for each experiment. In each test run, we only run one game, while keeping fixed all the initial random seeds. This way we hope to make scores between different runs comparable, as the agents effectively play the same game. An alternate (and better) approach would be to perform a number of runs for each depth-time/simulations combination. However, due to the high scores achieved by some of the agents, acquiring descriptive statistics from multiple test runs is prohibitive, as some agents run for days even with a single life.

A. MCTS pac-man Vs LegacyTeam

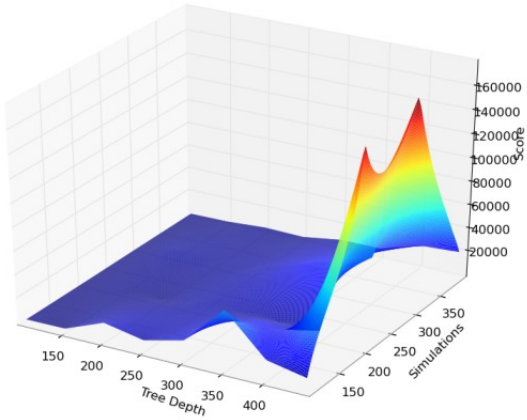
For this experiment The Ghost team is set to the *LegacyTeam*. MCTS is fully aware of this, thus not wasting rollouts in ghost moves that are not possible for this team. In practical terms, this means that for the first three ghosts whose behaviour is deterministic there is only one possible future move (whatever their model dictates) while for the probabilistic ghost we assume its behaviour is unknown, letting max^n search for plausible scenarios.

In the first experiment with this setup, we play a set of games using UCB1. The best performance achieved by this time is 70K at a tree depth of 450 (see Figure 4(a)). It can be observed from the experiments that there is no clear, clean relationship between tree depth, number of simulations and performance. An increased number of simulations with the

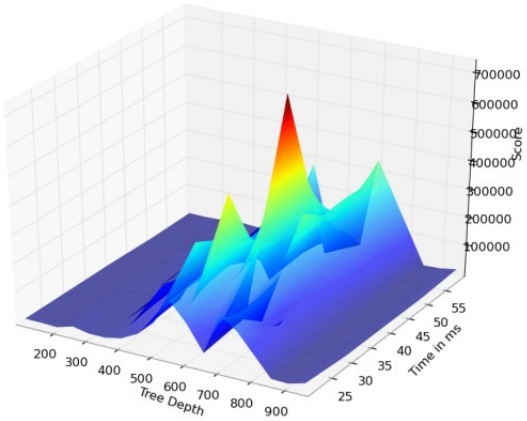
[4]All experiments were run on a Intel(R) Core(TM)2 Duo CPU E8600 @ 3.33GHz machine with java having 512MB of RAM.



(a) $UCB1$, number of simulations Vs tree search depth



(b) $UCB - TUNED$, number of simulations Vs tree search depth



(c) $UCB-TUNED$, time spend per move in ms Vs tree search depth

Fig. 4. Scores of pac-man Vs known *LegacyTeam* (i.e. future states in the tree search take into account the fact that we know where the ghosts are going to move next

same tree will not necessarily lead to better performance, contrary to what one would assume. There is however a clear and strong trend towards this direction, as is evident from the experiments. Our second run is the same as with UCB1, but this time we switched to $UCB-TUNED$ (Figure 4(b)). This effectively doubles the scores acquired by pac-man by almost two, to 160K. Again one can notice here that although tree depth does play a role in the quality of the results, this role is not clear cut, as there are "bumps" in the graph above. In the final setup for this experiment, we play a number of games using a fixed amount of time, while we let the number of evaluations vary (Figure 4(c)). The varying amount of simulations per move allows even better results, with our best score being almost 700K.

B. MCTS pac-man Vs Unknown Team (LegacyTeam)

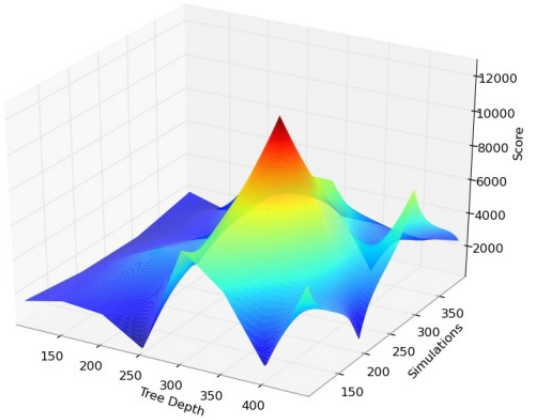
In this set of experiments, although we play against the same team as in the previous experiment, the agent treats the team as an unknown team. This means that the full max^n tree is searched, which should make it much harder for our agent to search for a good strategy.

In the first setup of this experiment, we see a noticeable drop in scores, almost an order of magnitude (see Figure 5(a)), with a high score of 10K. The same trend continues with the $UCB - TUNED$ setup as well (Figure 5(b)). There is a massive drop in score compared with the previous experimental setup (Vs. Known team), but $UCB - TUNED$ still outperforms $UCB1$ with a score of 12K. Finally, in the case where we fix time (Figure 5(c)), we achieve even better results (18K). Note that this is a much tougher version of the game than previous approaches using the same simulator due to the extremely low ghost edible time.

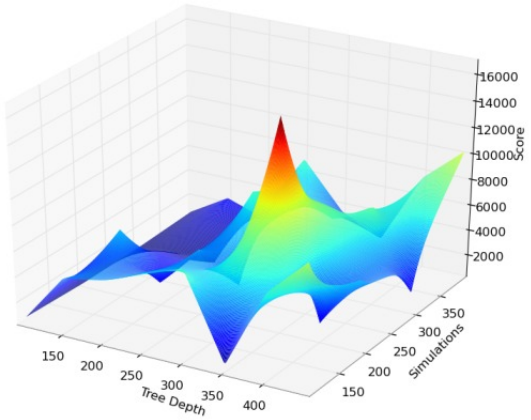
C. Experiment 3: MCTS pac-man Vs MCTS Ghosts

In this experiment we performed a test of MCTS ghosts Vs MCTS pac-man. Please note here, as explained earlier, that both pac-man and the ghost play using the same tree. The MCTS score is evaluated once and followed for all agents, with the exception of the first ghost, which blindly follows pac-man.

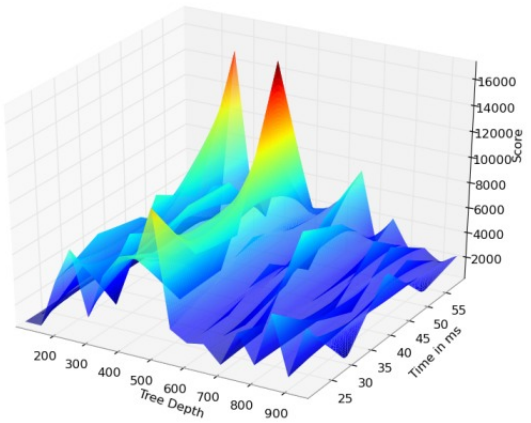
The results in all three setups (see Figures 6(a), 6(b), 6(c))) show pac-man being clearly overpowered by the ghost team,



(a) $UCB1$, number of simulations Vs tree search depth



(b) $UCB-TUNED$, number of simulations Vs tree search depth



(c) $UCB-TUNED$, time spend per move in ms Vs tree search depth

Fig. 5. Scores of pac-man Vs unknown *LegacyTeam* (i.e. when MCTS is not explicitly aware of which team it is performing against)

with very low high scores in all cases (a max of 6K). The reason for the not-so-clear curve in pacman's performance is that the number of iterations changes both for pacman and the ghosts. Thus, a stronger pacman would play with stronger ghosts, which would result in similar results no matter the number of iterations (taking into account some variance as well). If one is to draw a conclusion here, one can say that the MCTS ghosts can easily overpower pac-man, at least in the case of minimum edible time. Admittedly, one can vary asymmetrically the number of iterations of the ghosts in order to balance power. This could be used in order to create weaker "default" MCTS ghost teams for the game, which would provide an interesting direction if one is interested in creating progressively harder ghost teams to play against, adding to the fun element of the game.

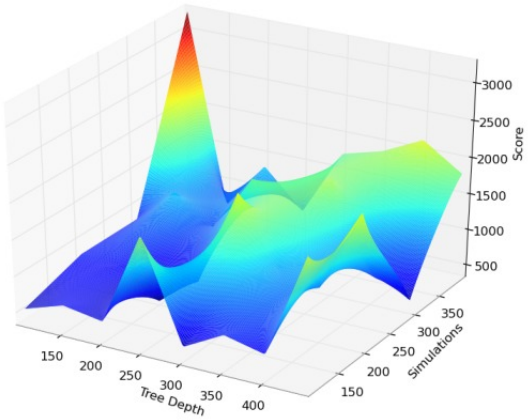
D. Experiment 4: MCTS pac-man Vs Typical LegacyTeam Setup / High Score

In this experiment we aim purely to show the strength of our agent against previous published work using the same simulator[12], [14]. Thus, in this experiment we only use the first stage and we set the ghost edible time to 100. We performed three tests; one with a known model, one an unknown ghost-team model, and non-real time one with an unknown ghost model. Since this experiment is the only one comparable to previously reported experiments with the simulator, which makes result robustness important, for each agent we performed 100 runs. Results are presented in tables I and II.

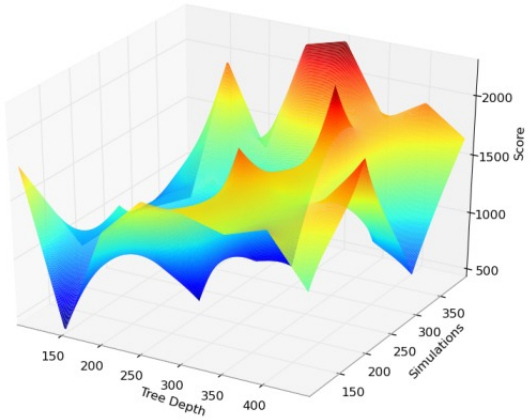
TABLE I
DESCRIPTIVE STATISTICS FOR THE TWO PLAYERS. KM IS THE KNOWN MODEL PLAYER AND UM IS THE UNKNOWN MODEL PLAYER. UM-NR AND UM-R SHOW RESULTS FOR THE UNKNOWN MODEL PLAYER FOR THE REAL-TIME AND THE NON-REAL TIME CASE RESPECTIVELY. S STATISTICS ARE FOR THE NUMBER OF ROLLOUTS WHILE T STATISTICS ARE FOR TIME PER MOVE. TD STANDS FOR TREE DEPTH.

	$MeanS$	$MinS$	$MaxS$	$MeanT$	$MaxT$	TD
KM	300	300	300	45	222	300
UM-NR	300	300	300	74	398	300
UM-R	150	150	300	35	182	300

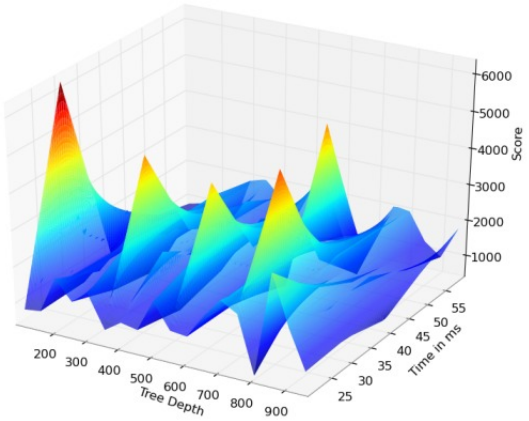
The first thing worth noting here is the very high score, slightly above 2.8M for the known model agent. The best human player has achieved a score of 933,580 [37], however this is not directly comparable to our version, as the ghost



(a) $UCB1$, number of simulations Vs tree search depth



(b) $UCB-TUNED$, number of simulations Vs tree search depth



(c) $UCB-TUNED$, time spend per move in ms Vs tree search depth

Fig. 6. Scores of pac-man Vs MCTS team

TABLE II
SCORES FOR EACH PLAYER, SEE FIGURE I FOR THE THREE ABBREVIATIONS (KM, UM-NR, UM). THE STATISTICS ARE BASED ON 100 GAMES PLAYED FOR EACH ROW OF THE TABLE.

	Mean	Standard Error	SD	Min	Max
KM	546,260	55,980	559,802	4,500	2,807,700
UM-NR	81,979	8,909	89,090	1,080	409,730
UM-R	45,389	4,041	40,407	1,400	207,800

behaviours are somewhat different. On the other hand, this shows that one can aim for such high scores in the real game as well, using similar approaches to ours, but this is out of the scope of this paper. For the record, the highest scoring Pac-man player for the real game (using a screen-capture kit for interfacing the agent to the game) we are aware of achieved a maximum score of $44,910$ [38], and is also based on Monte-Carlo Tree Search[5]. In the "unknown model" player tests, we can see that high scores are also achieved (at least 5 times previously reported results in the non-real time case). This can easily be explained by understanding that while in the known model case, only one ghost has unpredictable behaviour, in the unknown model case, all ghost actions have to be guessed. This creates trees which have a higher branching factor, which in turn requires more search time. Finally, the huge variability observed is explained by the fact that the agent performs simulations not in the real version of the game, but an abstract one. At some point, inconsistencies in the abstract model can easily lead to a premature death. In the real version of the game, one has at least three lives, so one has greater opportunity to amass a high score, and also be less affected by fatal errors (which cause the loss of a single life rather than the end of the entire game).

[5]The paper [38] is currently only available in Japanese.

VII. DISCUSSION & CONCLUSION

We have shown that MCTS can successfully be used in a real time game, getting results that are almost two orders of magnitude better than previous results in the same simulator acquired by evolutionary, reinforcement learning and genetic programming methods. In order to make a critical appraisal, we first need to concentrate on one simple fact: MCTS exploits the model of the game and (in the non-planning tree scenario) models our opponents as being perfect players. This ruthless exploitation of the forward model gives the agent a

characteristic strength that static/reactive evaluation heuristics cannot possibly match. What is important to note here is that it is not the senses that guide our agents behaviour, but rather an internalised model. Senses are practically only used in order to infer the current state, but do not influence behaviour in any other form. On top of all that, the plan the agent is to follow is re-formed at every timestep, making the need for feedback corrections redundant. The only thing that needs to be accurately measured is the current state and that state needs to be mapped to the internal model.

Another reason why the on-line exploitation of the model is so successful (compared to creating off-line controllers using evolution or reinforcement learning) is that even in the case where a reactive behaviour would perform successfully, the necessary function approximation would diminish the performance of the agent. The constant re-planning at every time step we do here alleviates the need for any form of approximator (such as a neural network).

An interesting phenomenon which is evident from the experiments is that in our case more computational time, which invariably results in more simulations, does not necessarily result in better performance. We think that the primary reason behind this is may be as follows: unless some critical threshold is crossed, where the agent clearly plays better, an avalanche of slightly different decisions leads to totally different games. So a slightly better agent might find himself in a difficult situation and fail, whereas an inferior agent will never come to see that state at all! This kind of behaviour has not been observed in any MCTS implementations until now as far as we are aware. However, in most previous efforts to create MCTS agents, the simulations involved playing a real, albeit guided, game until the end. In our case the simulations run an idealised game, with a heuristic end function.

We believe that the approach presented in this paper has great potential for creating generic AI agents. One can easily envisage a procedure where the most important abstract features of a world are modelled (such as the game rules and "equation" of motion) and given to an agent to reason with. The agent can then use MCTS to produce general intelligent (or at least sensible) behaviour with a minimum of domain-specific programming.

ACKNOWLEDGEMENTS

This research is partly funded by a postgraduate studentship from the Engineering and Physical Sciences Research Council and from the National Council of Science and Technology of Mexico (CONACYT).

REFERENCES

[1] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo Tree Search," in *Proceedings of the 5th International Conference on Computers and Games (CG2006)*, 2006, pp. 72–83.

[2] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modifications of UCT with Patterns in Monte-Carlo Go," INRIA, Tech. Rep. 6062, 2006.

[3] Y. Bjornsson and H. Finnsson, "CadiaPlayer: A simulation-based general game player," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, pp. 4–15, 2009.

[4] B. Arneson, R. Hayward, and P. Henderson, "Monte Carlo Tree Search in Hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251 –258, 2010.

[5] H. Matsumoto, T. Ashida, Y. Ozasa, T. Maruyama, and R. Thawonmas, "Ice pambush 3," Ritsumeikan University, Tech. Rep., 2009.

[6] J. R. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[7] I. Szita and A. Lorincz, "Learning to play using low-complexity rule-based policies," *Journal of Artificial Intelligence Research*, vol. 30, no. 1, pp. 659–684, December 2007.

[8] J. S. D. Bonet and C. P. Stauffer, "Learning to play Pac-Man using incremental reinforcement learning." 1999, `http://www.ai.mit.edu/people/stauffer/Projects/PacMan/`.

[9] M. Gallagher and A. Ryan, "Learning to Play Pac-Man: An Evolutionary, Rule-Based Approach," in *IEEE Congress on Evolutionary Computation*, 2003, pp. 2462–2469.

[10] M. L. Marcus Gallagher, "Evolving Pac-Man Players: Can We Learn from Raw Input?" in *IEEE Symposium on Computational Intelligence and Games*, 2007.

[11] N. Wirth and M. Gallagher, "An Influence Map Model for Playing Ms Pac-Man," *IEEE Symposium on Computational Intelligence and Games*, 2008.

[12] S. M. Lucas, "Evolving a Neural Network Location Evaluator to Play Ms Pac-Man," *IEEE Symposium on Computational Intelligence and Games*, pp. 203–210, 2005.

[13] D. Robles and S. Lucas, "A Simple Tree Search Method for Playing Ms Pac-Man," in *Proceedings of the 5th International Conference on Computational Intelligence and Games*, 2009, pp. 249–255.

[14] P. Burrow and S. Lucas, "Evolution versus temporal difference learning for learning to play Ms Pac-Man," in *Proceedings of the 5th international conference on Computational Intelligence and Games*. IEEE Press, 2009, pp. 53–60.

[15] A. M. Alhejali and S. Lucas, "Evolving Diverse Ms Pac-Man Playing Agents Using Genetic Programming," in *Proceedings of the UK Workshop on Computational Intelligence (UKCI)*. IEEE Press, 2010.

[16] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in *15th European Conference on Machine Learning (ECML)*, 2006, pp. 282–293.

[17] E. Rasmusen, *Games and information: An introduction to game theory*. Blackwell Pub, 2007.

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

[19] S. Gelly and Y. Wang, "Exploration exploitation in Go: UCT for Monte-Carlo Go," in *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*. Citeseer, 2006.

[20] C. S. Lee, M. H. Wang, C. Chaslot, J. Hoock, A. Rimmel, O. Teytaud, S. R. Tsai, S. C. Hsu, and T. P. Hong, "The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments," in *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, 2009, pp. 73–89.

[21] S. Billouet, J. Hoock, C.-S. Lee, O. Teytaud, and S.-J. Yen, "9x9 go as black with komi 7.5: At last some games won against top players in the disadvantageous situation," in *ICGA Journal*, vol. 32, no. 3, 2009, pp. 241–246.

[22] S.-J. Yen, C.-S. Lee, and O. Teytaud, "Human vs. Computer Go Competition in FUZZ-IEEE 2009," in *ICGA Journal*, vol. 32, no. 3, 2009, pp. 178–180.

[23] R. Blincoe, "Go, going, gone?" The Guardian, 2006. [Online]. Available: http://www.guardian.co.uk/technology/2009/apr/30/games-software-mogo/print

[24] R. Coulom, "Computing ELO ratings of move patterns in the game of Go," in *Computer Games Workshop*. Citeseer, 2007.

[25] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy, "Progressive strategies for Monte-Carlo Tree Search," *New Mathematics and Natural Computation*, vol. 4, no. 3, p. 343, 2008.

[26] M. Enzenberger, M. Muller, B. Arneson, and R. Segal, "FUEGO: An Open-Source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, p. to appear, 2010.

[27] F. Van Lishout, G. Chaslot, and J. Uiterwijk, "Monte-Carlo Tree Search in Backgammon," in *Computer Games Workshop*, 2007, pp. 175–184.

[28] P. Hingston and M. Masek, "Experiments with Monte Carlo Othello," in *IEEE Congress on Evolutionary Computation, 2007. CEC 2007*, 2007, pp. 4059–4064.

[29] G. V. den Broeck, K. Driessens, and J. Ramon, "Monte-Carlo Tree Search in Poker Using Expected Reward Distributions," in *ACML*, 2009, pp. 367–381.

[30] S. Gelly and D. Silver, "Achieving Master Level Play in 9 x 9 Computer Go," in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008, pp. 1537–1540.
[31] M. Chung, M. Buro, and J. Schaeffer, "Monte Carlo Planning in RTS Games," in *IEEE Symposium on Computational Intelligence and Games*, 2005.
[32] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game AI," in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008, pp. 216–217.
[33] R. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in *Proceedings of the 21st International Joint Conference on Artifical intelligence*. Morgan Kaufmann Publishers Inc., 2009, pp. 40–45.
[34] D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron, "Game-tree search with adaptation in stochastic imperfect-information games," *Lecture Notes in Computer Science*, vol. 3846, pp. 21–34, 2006.
[35] C. Luckhardt and K. Irani, "An algorithmic solution of n-person games," in *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI)*, 1986, pp. 158–162.
[36] S. Samothrakis, D. Robles, and S. Lucas, "A UCT agent for Tron: Initial investigations," in *Proceedings of the 6th International Conference on Computational Intelligence and Games*, 2010.
[37] T. Galaxies. (2006) Twin Galaxies - Ms Pac-Man High Scores @ONLINE. [Online]. Available: http://www.twingalaxies.com/index.aspx?c=22&pi=2&gi=3162&vi=1386
[38] N. Ikehata and T. Ito, "Monte Carlo Tree Search in Ms Pac-Man," in *The 15th Game Programming Workshop, IPSJ Symposium Series Vol. 2010/12*, 2010.

Spyridon Samothrakis is currently pursuing a PhD in Computational Intelligence and Games at the University of Essex. His interests include game theory, machine learning, evolutionary algorithms and consciousness.

David Robles did his MSc in Computer Science at the University of Essex in 2008. He is now pursuing a PhD in Computer Science in the area of artificial intelligence in games.

Simon Lucas (SMIEEE) is a professor of computer science at the University of Essex (UK) where he leads the Game Intelligence Group. His main research interests are games, evolutionary computation, and machine learning, and he has published widely in these fields with over 130 peer-reviewed papers, mostly in leading international conferences and journals. He was chair of IAPR Technical Committee 5 on Benchmarking and Software (2002 - 2006) and is the inventor of the scanning n-tuple classifier, a fast and accurate OCR method. He was appointed inaugural chair of the IEEE CIS Games Technical Committee in July 2006, has been competitions chair for many international conferences, and co-chaired the first IEEE Symposium on Computational Intelligence and Games in 2005. He was program chair for IEEE CEC 2006, program co-chair for IEEE CIG 2007, and for PPSN 2008. He is an associated editor of IEEE Transactions on Evolutionary Computation, and the Journal of Memetic Computing. He has given invited keynote talks and tutorials at many conferences including IEEE CEC, IEEE CIG, and PPSN. Professor Lucas was recently appointed as the founding Editor-in-Chief of the IEEE Transactions on Computational Intelligence and AI in Games.