Monte Carlo Tree Search Applied to Co-operative Problems

Piers R. Williams, Joseph Walton-Rivers, Diego Perez-Liebana, Simon M. Lucas

***Abstract*—This paper highlights an experiment to see how standard Monte Carlo Tree Search handles simple co-operative problems with no prior or provided knowledge. These problems are formed from a simple grid world that has a set of goals, doors and buttons as well as walls that cannot be walked through. Two agents have to reach every goal present on the map. For a door to be open, an agent must be present on at least one of the buttons that is linked to it. When laid out correctly, the world requires each agent to do certain things at certain times in order to achieve the goal. With no modification to allow communication between the two agents, Monte Carlo Tress Search performs well and very "purposefully" when given enough computational time.**

I. Introduction

The research problem studied in this paper consists of how General Game Playing (GGP) agents perform when trying to solve a simple co-operative problem without co-operative abilities, with a focus on Monte Carlo Tree Search (MCTS). GGP is the field of writing Artificial Intelligence (AI) agents that can play a multitude of games without being written specifically for each one individually [1]. GGP in real time video games has a popular competition [2] run frequently.

Games that feature co-operation of some form between human players and AI agents are commonplace. Most however feature very limited forms of co-operation that are typically scripted such as in most First-Person Shooter (FPS) games. Typically FPS games give the mere impression of co-operation, though any player that looks carefully at it will see the tell tale signs of scripting. Where FPS games typically excel at co-operation is in online modes that enable teams of humans to play against each other. Some games even provide squad structures and communication allowing direct command for the purpose of better co-ordination as in *Battlefield 2142* (EA Digital Illusions CE, 2006). Real-Time Strategy (RTS) games also often have a small number of features designed to facilitate communication in a bid to facilitate co-operation. Two games that stand out for co-operation are *Rise of Nations* (Big Huge Games, 2003) and *Empire Earth II* (Mad Doc Software, 2005). Rise of Nations allowed a human and AI player to operate the same set of units and buildings, though no communication was possible at all. This allowed a form of co-operation but the AI operated to its own agenda. Empire Earth 2 allowed for humans and AI agents to co-operate by letting plans be drawn up between them that could also be followed by both the human and AI agent. These allowed a fairly complex set of instructions to be created, despite the simple interface.

A highly popular game that had an entire mode designed for co-operation between humans was *Portal 2* (Valve, 2011). This featured human sized lab test mazes with elements that required players to work together by activating buttons, moving cubes and using intra-dimensional portals to get to the end goal. Both players were required to reach the goal in order to complete the level.

Creating GGP agents that can co-operate with other players would open the door for more flexible agents in games that can work together with human players. This experiment tests how well current techniques can cope without specifically co-operating with each other.

A. Monte Carlo Tree Search

MCTS is a Tree Search algorithm originally proposed in 2006 [3], [4], [5]. MCTS operates in the action space, building an asymmetric tree in memory that biases the search towards the most promising parts of the search space. MCTS estimates the theoretic value of the visited states by performing self-play from the state in the node, to the end of the game. The basic steps for MCTS are:

1) *Selection* - Selection is the stage where the algorithm navigates the search tree, selecting optimal nodes (based on the Tree Policy) until it reaches a leaf node.
2) *Expansion* - Expansion is the stage whereby the leaf node is expanded by adding one of the remaining child states if it is not a terminal node (game end state).
3) *Simulation* - Simulation is the stage where the algorithm forwards the model until a result is achieved. Their is usually a policy that defines how the simulation is made, with the simplest being random possible moves.
4) *Backpropagation* - Backpropagation is the stage where the results of the simulation are propagated up the tree, so that they can influence the selection phase.

MCTS [6] has been applied to a wealth of domains and is one of the primary algorithms in use for GGP [7]. Primary advantages are that, when given a sufficient forward model, MCTS does not require any strategic knowledge about the game itself in order to play.

Standard MCTS plays best when it is able to search far enough to locate states that provide a reward. MCTS tends to stumble when the time or search depth limit given to it is not sufficient to allow it to locate any sequence of actions that provides a reward in order to differentiate the root's children [8]. When MCTS can not find any single state that contained

All authors are with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, UK; email: {pwillic,jwalto,dperez,sml}@essex.ac.uk).

a reward, all children of the node will contain identical values and the algorithm will be forced to make a random choice.

The rest of this paper is structured as follows. Section II describes the problem and the experiment, including the AI controllers that were used. Section III describes the results of the experiment and Section IV provides a discussion of the results. Section V concludes and introduces avenues of future research.

II. The Experiment

The main premise was to put MCTS in a situation where it would rely on other AI Agents in order to achieve its objectives, as well as provide a code base for further work with communication between *Agents*. In order to do this, a problem domain was created.

A. The Problem Domain

We created the following problem domain that is a simple grid world consisting of various objects:

- *Floor* - Floors are passable, and are rendered in Grey.
- *Walls* - Walls are impassable, and are fixed in position. *Walls* are rendered in Black.
- *Agents* - Agents are the moving objects that can activate buttons and the goal. *Agents* are rendered in dark Yellow
- *Doors* - Doors can either be open or closed. When open, the door is passable. When closed, the door is impassable. Doors are open whilst an *Agent* is activating a linked *Button*. If the *Agent* stops activating the *Button* the *Door* will close. *Doors* are rendered in Blue when closed, and are invisible when open.
- *Buttons* - Buttons are passable, and when an *Agent* is on the *Button* it will activate and open the linked *Door*. *Buttons* are rendered in Red
- *Goals* - Goals are passable and reward all *Agents* with a portion of the score. All *Goals* are worth the same amount and the maximum score of 1 is achieved when every *Agent* has visited every *Goal* at least once. Re-visiting a *Goal* has no effect. All *Agents* are aware of the current score achieved, and therefore can calculate if a *Goal* is reached. *Goals* are rendered in Bright Yellow

The *Agents* are provided with a forward model, that allows simulation of potential action pairs up to the end of the game. The *Agents* are not allowed to directly query any information about the game state other than the score at any given state (current or simulated). There is no way to determine the presence of in-game objects other than potentially the *Goal* due to its effect on score.

Each *Agent* is able to make a single action in each game tick. Each agent, after receiving the state of the game, must return a valid action. Once all *Agents* have returned an action, the game is updated with these moves.

The actions are implemented sequentially: if two *Agents* return actions that result in occupying the same space, the *Agent* with the lowest ID number will succeed while the other *Agent* will fail and execute a *No-Op*. The available 5 actions are:

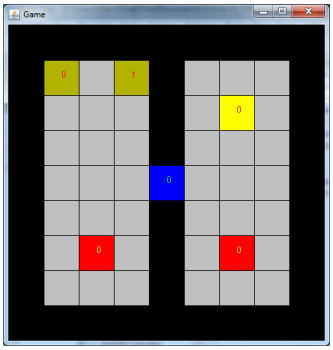

Fig. 1: Map *Single Door* [1]

- *Left* - This will move the *Agent* one grid square to the left. (-1, 0)
- *Right* - This will move the *Agent* one grid square to the right. (1, 0)
- *Up* - This will move the *Agent* one grid square up. (0, -1)
- *Down* - This will move the *Agent* one grid square down. (0, 1)
- *No-Op* - This will not move the *Agent*. It will remain in the same location as the previous game state.

All levels are defined by a simple text file format.

The problems are designed to require *Agents* to co-operate in order to achieve their goals. Unlike other domains where MCTS can solve tasks at its own pace and by itself, this domain requires the agents to sometimes sit patiently on a button for another *Agent* to do something else.

We feel that this creates a very interesting problem domain, due to placing each *Agent* that is typically designed to solve problems by themselves in an environment where they must rely on another *Agent* performing certain sequences of actions without being able to communicate that sequence or anything else.

B. Tournament

For the main data collection, we ran a round robin tournament of maps and Agents. Each Agent was paired with all 7 Agents (including a copy of itself) and played 47 games on each of the 7 maps.

1) Maps: The 7 maps that were included in the tournament were as follows:

a) SingleDoor*:* This map, depicted in Figure 1, is a basic map with two rooms, a single door and a button on either side.

b) Pathfinding*:* This map, depicted in Figure 2, is the same as *SingleDoor* but without the *Doors* and *Buttons*. This has the aim of testing the *Agent's* ability to solve the simplest problem.

c) SymmetricSingleDoor*:* This map, shown in Figure 3, is a modification of the *SingleDoor* - making the map more symmetric and the action parts closer together.

[1]Displayed numbers are the ID numbers - *Button* ID 0 will open *Door* ID 0 but not *Door* ID 1

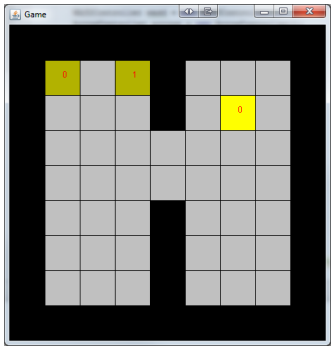

Fig. 2: Map *Pathfinding*

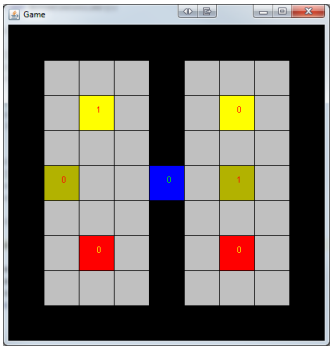

Fig. 5: Map *Side By Side*

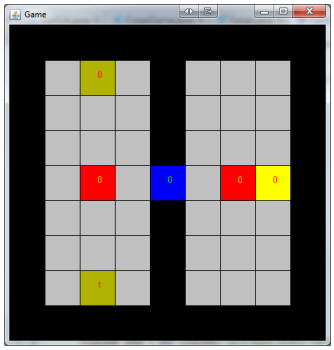

Fig. 3: Map *Symmetric Single Door*

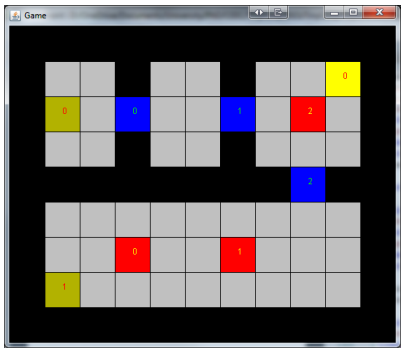

Fig. 6: Map *Airlock*

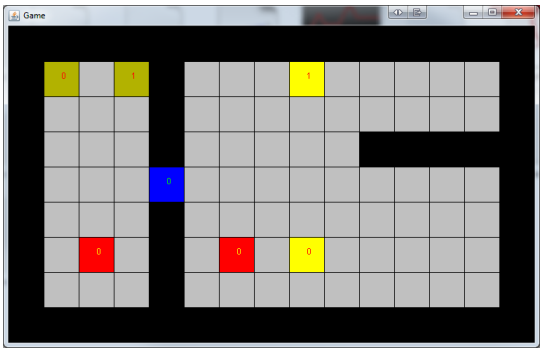

Fig. 4: Map *Extended Side*

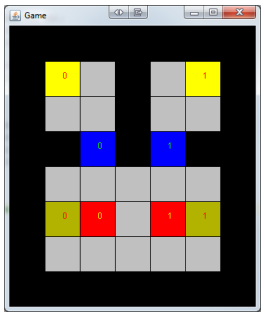

Fig. 7: Map *Butterfly*

d) ExtendedSide*:* This map, shown in Figure 4, is an extension of *SingleDoor* with a second *Goal* and some extra walls in the game. Spreading the action parts away from the door and buttons is the design after this one.

e) SideBySide*:* This map, illustrated in Figure 5, is a modification of *SingleDoor*, including a second *Goal* and making the map partly mirrored. Each *Agent* begins in a separate room to each other instead of together.

f) Airlock*:* This map (see Figure 6) is designed to give wildly different roles to each AI: the top Agent would have to travel through two doors (like an airlock) that the bottom agent had access to the linked *Buttons*. Then the top *Agent* could collect the reward and allow the bottom *Agent* access to the *Goal*.

g) Butterfly*:* This map (see Figure 7) is a map with two rooms each, with a *Goal* and a door. This was designed to force each *Agent* to be let into the *Goal* room, and out of it again.

2) AI Controllers: A set of AI Controllers were created in order to attempt to solve the problem domain. None of the agents have the ability to communicate with another agent and each agent is managed by a single controller.

a) Random: The random controller simply chooses one of the possible 5 actions. This is one of the simplest to implement and run.

b) MCTS: The MCTS controller is a simple implementation of Upper Confidence bound applied to Trees (UCT), with a fixed number of rollouts, UCT tree search depth limit and rollout border. The rollout border is how far in total the forward model will be allowed to progress before the

Budget	Rollouts	UCT Search Limit	Rollout Border
Small	75	3	15
Medium	200	5	30
High	500	10	45

TABLE I: Table of parameters for the 3 MCTS players

Param	Controls	Value
minNum	Number of Macro Actions	3
MaxNum	Number of Macro Actions	10
minLength	Length of Macro Actions	1
maxLength	Length of Macro Actions	5
numChance	Chance of altering Number	0.25
lengthChance	Chance of altering each length	0.8
actionChance	Chance of altering each action	0.75

TABLE II: Table of parameters for the Variable Macro Action Genetic Algorithm

game state is evaluated. No knowledge about the game is provided and the assumption is made that the other player will play randomly. The fixed rollout border makes a significant improvement in iterations per decision (from 1-3 to 500-600 in 40ms [2]). This greatly improves the ability of MCTS to make informed decisions. The score at the end of a rollout is taken from the game state - so if MCTS does not see any player reach a goal, all branches will be equal. For the tournament, three parameter sets were chosen for comparison and are shown in Table I.

c) Macro Action Genetic Algorithm (GAController): The Genetic Algorithm (GA) algorithm was selected for its simplicity and had Macro Actions added in order to improve the forward search capability as well as the amount of computation time per decision available to it. The Macro Action Genetic Algorithm (MAGA) used a population size of 10 and tournament selection of 3. Each candidate was a string of 15 actions - with each action performed 3 times in a row. This meant the MAGA could "see" 45 ticks in the future. The design of the MAGA algorithm was inspired by *Perez et al* [9].

d) Variable Macro Action Genetic Algorithm (VarGA): The Variable Macro Action GA was created to try to solve the shortcomings of MAGA. The primary premise was to allow the GA to evolve the individual macro action lengths and the length of the overall sequence (number of macro actions). The main technique used was a $1+1$ Evolutionary Strategy (ES) [10]. A $1+1$ ES is a very simple GA that maintains a single candidate. On each iteration, the mutation operator is applied to this individual, and it is saved as the new best candidate in case of an improvement in fitness (and discarded otherwise). This kind of ES is much simpler than a full GA, requires much less memory and due to a small computational cost within each iteration, is able to use more of the available time budget.

The algorithm was bounded by a number of parameters (shown in Table II), and a further tuning of these is a source of possible future work in order to explore the potential of this algorithm.

III. Results

Figure 8 shows the ability of each Agent to solve a simple path finding problem. The *GAcontroller*, with its macro actions, is hindered by its inability to make single step moves and has trouble actually walking straight to the target. The *VariGA* does significantly better, due to its ability to perform single step moves. Figure 9 shows that most of the controllers really struggle when the *Door* and *Buttons* are added. This

[2]Intel Core i5-3570, 8GB RAM, Windows 7 Enterprise 64bit

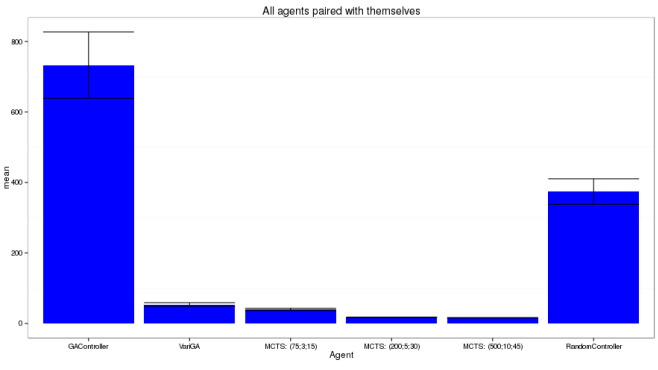

Fig. 8: Average ticks taken to complete *Pathfinding* when paired only with identical Agents

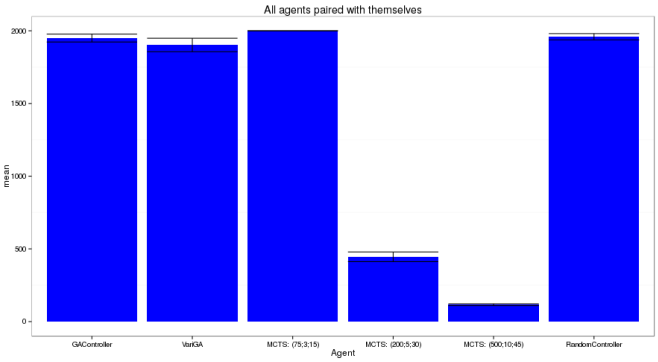

Fig. 9: Average ticks taken to complete *SingleDoor* when paired only with identical Agents

indicates that the challenge of the task is high, although perhaps remarkable that MCTS is able to solve the task.

Figure 10 shows the average score that each AI Agent achieved over all the maps. The two more powerful MCTS Agents performed the best, doing much better than the competition. Figure 11 shows a similar result, with the more powerful Agents typically completing the maps in fewer ticks. *RandomController* is the only deviant here - it outperformed the GA's in score but was worse in average ticks. Comparing the AI *Agents* when paired only with themselves in Figure 12 and Figure 13. The good results for *RandomController* are potentially due to the fact that the 3 MCTS controllers and 2 GA algorithms all based their decisions on having Random as an accomplice.

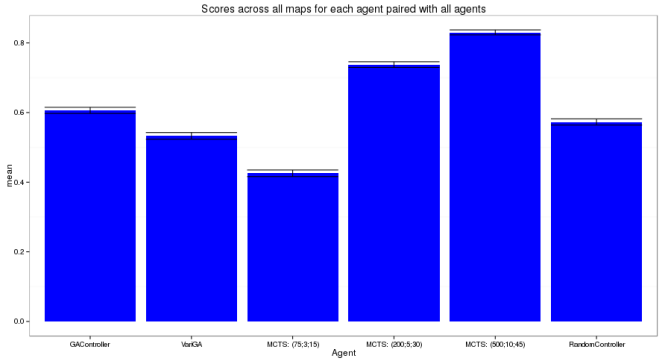

Fig. 10: Average score of each AI Agent over all pairings over all the Maps

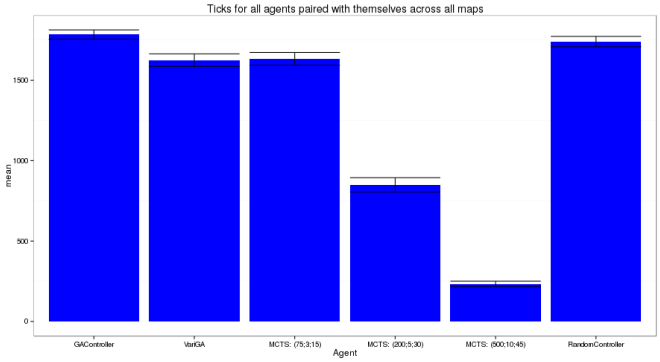

Fig. 13: Average Ticks taken for each AI Agent paired with itself over all the Maps

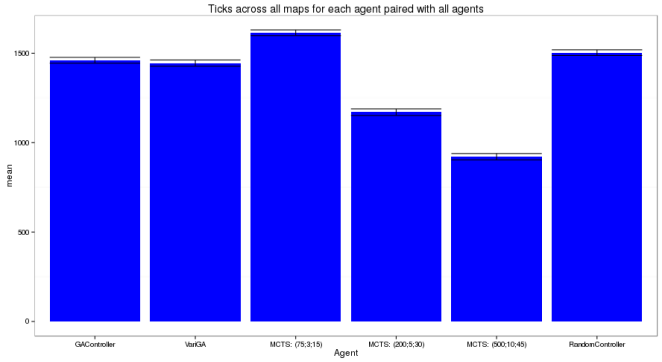

Fig. 11: Average Ticks taken for each AI Agent over all pairings over all the Maps

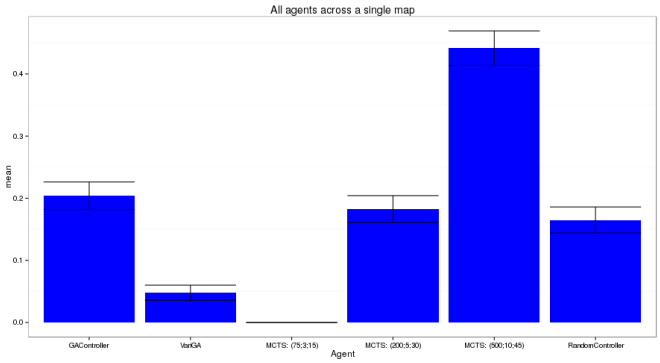

Fig. 14: Average Score for each AI Agent paired with itself over *Airlock*

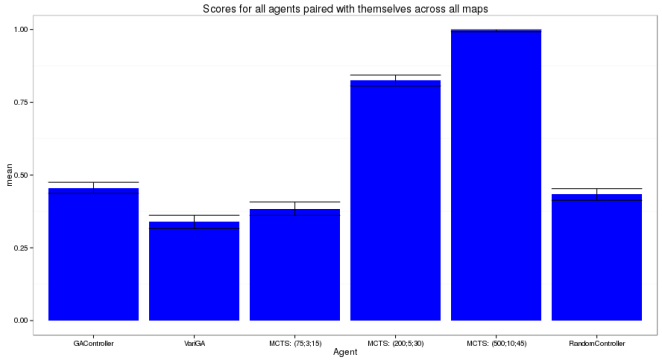

Fig. 12: Average Scores for each AI Agent paired with itself over all the Maps

Airlock - shown in Figure 6 - poses a particular problem for many of the agents (see Figure 14). The asymmetric nature of the level, and the delayed reward caused great difficulty for the Agents. Relying on two button presses and the other agent to get through the open doors - in order - greatly reduced performance. The top agent, *High MCTS*, only managed an average score of 0.44 compared to its total average of just over 0.8.

IV. Discussion

A. Random Assumptions

The MCTS and AI algorithms are required to run simulations of the game state in order to function. Even just a single look ahead of this problem domain requires the algorithm to make a selection of what the other player would do. In all cases, the assumption of a random strategy was chosen due to the availability only of agents that could play generally. Whilst nesting these could enhance play, as explored by Tristan Cazenave [11], it comes at a significant computational cost and must in the case of GGP feature a very simple or Random player at the base of the stack. Random is exceedingly fast to calculate, and provides the ability for techniques like MCTS to sample how its behaviour works almost regardless of what the other agent does. This gives MCTS the ability to make moves that it believes would help a worst case scenario player - something that in this problem domain is likely to also help a more intelligent player. When MCTS is sitting on the button, an intelligent player would go through the open door and a random player would eventually go through it. MCTS even has the option of locking the random player on the other side, forcing it to eventually complete some other task.

B. MCTS

MCTS performed very well in this problem domain. As seen above, in Figure 13, the medium and high budget implementations provided the two quickest completion times across all maps. The differences shown were significant as well as being the only two agents to complete on average in under 1000 ticks. Figure 12 showed that the medium and high budget implementations scored significantly better than all other agents.

One possible reason for MCTS scoring so well in this problem domain is its use of statistics over hundreds and thousands of simulations to provide it with the ability to act in such a way that it handles all eventualities. Statistically, in certain situations MCTS would only see rewards in the tree when it was situated on a *Button*. This tended to cause the MCTS to travel towards *Buttons*, increasing the possibility that its own simulations would cause it to be situated on the button. Eventually, most other AI *Agents* would cross through the open door.

C. GA

The two GA algorithms did not perform very well in this problem domain. As seen in Figure 12 and Figure 13, when tasked with solving the problem domain with another identical agent, neither the *GAController* or *VariGA* performed well at all. The *VariGA* only performed significantly better than either *RandomController* or *GAController* in *Pathfinding*. In all other cases, *VariGA* performed similarly to the standard *GAController*. The ability to mutate the lengths of individual action sequences made the *VariGA* a more flexible pathfinder that the *GAController* but did not aid its ability to solve the co-operative problems present elsewhere in the experiment.

The GA algorithms do not have the stored tree structure of MCTS with which to gain the statistical model for what happens when they pursue certain actions. This leads to a seemingly poor performance for the GA despite GA and MCTS typically performing equally well in other domains.

D. Random

The *RandomController* performed poorly in this problem domain. On average, the *RandomController* finished under the 2000 tick limit and scored less than half the maximum on average.

E. A* Search

A* search was considered to be unsuitable for this domain, as searching across the action space of two *Agents* was far too high a branching factor.

V. Conclusions

In this paper, we found that a strong MCTS player can solve simple cooperative problems without requiring communication between agents. We also found that a number of other AI techniques experienced difficulty when the problem became cooperative. We hypothesised that MCTS's use of a stored tree guiding its explorations (something that genetic algorithms lack) was a major advantage in solving problems that rely on the other agent. Genetic algorithms were found to perform poorly in the experiment, despite being capable of finding the goal without the co-operative obstacles. Further work is required in this area to determine exactly why MCTS performs well.

A. Further Work

There is scope for more work in a number of areas.

1) Communication: More work can be conducted in the writing of AI's to use communication in order to complete these challenges better amongst themselves. A framework for communication would be devised - with the requirement for fairly GGP like restrictions in mind. Use of the communication framework would need to likely be learned by the *Agents* during play, posing a particular challenge for the future.

2) Optimisations: Future work should also include some work to ensure that the AI Agents have their parameters set correctly. The *High MCTS* performed well - showing the power of having the correct parameters compared to the *Low MCTS*. The *VariGA* has 7 parameters, and it is possible that by fine tuning their values, a higher performance could be found.

3) Problem Domain: More work can also be done on experimenting with the problem domain itself. Expanding the object types to include different techniques such as pickup-able objects that can be placed on *Buttons*, *Buttons* that remain active for a period of time after vacating them, as well as *Doors* that will be open permanently if all (or a subset of) *Buttons* are activated. The achievable scope modifying the problem domain is fairly limitless.

References

[1] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI magazine*, vol. 26, no. 2, p. 62, 2005.

[2] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition," *IEEE Transactions on Computational Intelligence and AI in Games*, p. (to appear) DOI: 10.1109/TCIAIG.2015.2402393, 2015.

[3] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and games*. Springer, 2007, pp. 72–83.

[4] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Machine Learning: ECML 2006*. Springer, 2006, pp. 282–293.

[5] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved Monte-Carlo Search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton *et al.*, "A Survey of Monte Carlo Tree Search Methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[7] H. Finnsson and Y. Björnsson, "Simulation-Based Approach to General Game Playing," in *AAAI*, vol. 8, 2008, pp. 259–264.

[8] D. Perez, P. Rohlfshagen, and S. M. Lucas, "Monte Carlo Tree Search: Long-term versus Short-term Planning," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 2012, pp. 219–226.

[9] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 351–358.

[10] T. Bäck, D. Fogel, and Z. Michalewicz, Eds., *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, Bristol, 2000.

[11] T. Cazenave, "Reflexive Monte-Carlo Search," in *Proc. Comput. Games Workshop*, 2007, pp. 165–173.