Engaging Turn-based Combat in The Children of the Galaxy Videogame

Pavel Šmejkal, Jakub Gemrot
Faculty of Mathematics and Physics, Charles University
Ke Karlovu 3
Prague 2, 121 26, Czech Republic

Abstract

In this paper we tackle a problem of tile-based combat in the turn-based strategy (space 4X) video game Children of the Galaxy (CotG). We propose an improved version of Monte Carlo tree search (MCTS) called MCTS considering hit points (MCTS_HP). We show MCTS_HP is superior to Portfolio greedy search (PGS), MCTS and NOKAV reactive agent in small to medium combat scenarios. MCTS_HP performance is shown to be stable when compared to PGS, while it is also more time-efficient than regular MCTS. In smaller scenarios, the performance of MCTS_HP with 100 millisecond time limit is comparable to MCTS with 2 seconds time limit. This fact is crucial for CotG as the combat outcome assessment is precursor to many strategical decisions in CotG game. Finally, if we fix the amount of search time given to the combat agent, we show that different techniques dominate different scales of combat situations. As the result, if search-based techniques are to be deployed in commercial products, a combat agent will need to be implemented with portfolio of techniques it can choose from given the complexity of situation it is dealing with to smooth gameplay experience for human players.

Introduction

In the recent years we have seen the development of superhuman artificial intelligence for games such as Go (Silver et al. 2016), and two player no-limit Texas holdem poker (Bowling et al. 2015). The research in game AI has now turned towards computer strategy games which pose new greater challenges stemming from their large game spaces like Dota 2 and StarCraft. In these games, AI agents (bots) win against human players in small subproblems, usually concerning reflexes and speed as shown by the OpenAI Dota 2 bot [1]. However, bots are mostly incompetent when playing the whole game which also involves high-level decision making and planning (Yoochul and Minhyung 2017). In this paper, we focus on the genre of turn based 4X strategy games represented by the commercial game Children of the Galaxy (CotG) developed by EmptyKeys studio [2]. However, discussion and results in this paper can be transfered to many games of the same genre as they share many game environment properties.

[1] https://blog.openai.com/dota-2/

[2] http://emptykeys.com/Games/Children-of-the-Galaxy

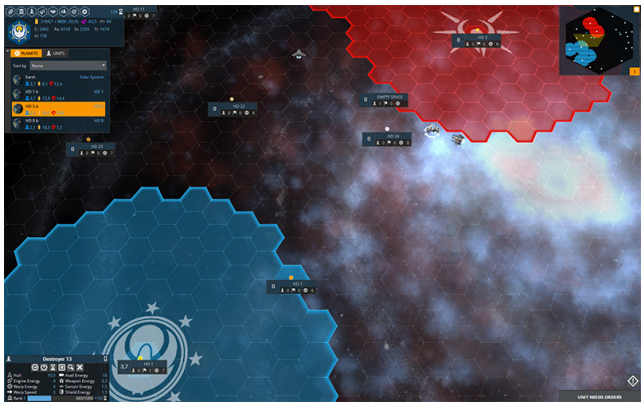

Figure 1: Galaxy view of the game. Color of the hexes indicates which players region they belong to. Top right corner shows minimap. Hexes with stars indicate solar systems.

Children of the Galaxy

CotG is fairly standard 4X game from space, featuring colonization of solar systems, management of colonies, researching new technologies, unit building, and tile-based space combat involving ships of asymmetric powers. Victory can be achieved in a few ways: eliminating all opponents in warfare, researching an ultimate technology, which is economically daunting, or colonizing most of the galaxy. Even though it is clear how to win the game from the human perspective (eXplore, eXpand, eXploit and eXterminate), the winner is determined by the right strategic decisions of how many and where to send one's ships (units in general). One sub-problem of this high-level unit movement planning is the fight or flee decision. Discovering an enemy army advancing to our positions, should an AI engage them with current units at hand or flee and regroup. And if it is to flee and regroup, how many other units should be brought or built to answer the threat? We cannot simply withdraw all other military units as they may have duties elsewhere. It is then hard to determine "what is enough to answer the threat" (or plan an attack). As units in commercial games have asym-

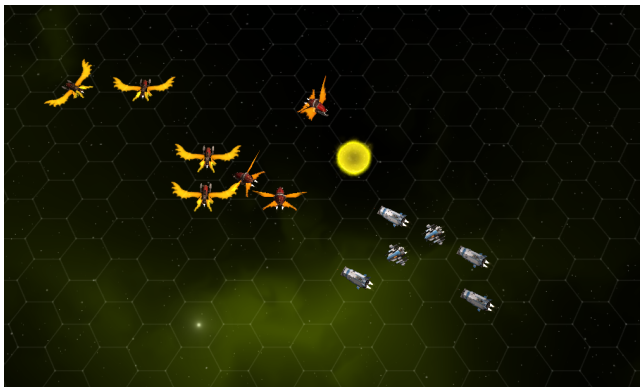

Figure 2: Solar system view with a small combat scenario. Yellow/winged ships are one army and grey/industrial ships the other army.

metrical powers it is hard to come up with formula to assess the strength of an army. This subproblem, assessment of an army's immediate strength, is (again) still complex as the number of combat situations (the type and number of units and the battlefield setup) is high plus it is affected by how well each player can utilize the units strenghts and exploit other units weaknesses in combat. Therefore, the combat capabilities of AI greatly impacts the strength of its army, which in turns affect its fight or flee decisions, which in the end affect what to build or what to research. As the result, we focus solely on solving the combat in CotG here as it is one of the precursor to many high-level decicion-making routines. Our chosen approach is to create a combat simulator that can be used both for the artificial play and to assess outcomes of future encounters.

Related Work

Churchill et al. (2012) try to solve combat in the RTS game StarCraft using improved Alpha-Beta pruning called Alpha-Beta Considering Durations (ABCD). This algorithm addresses the fact that not all units can perform actions each turn (some actions need to cool down) and how do deal with simultaneous actions in the search tree.

As a continuation, Churchill and Buro (2013) present two new search algorithms. The first one is Upper confidence bound for trees (UCT) Considering Durations (UCTCD) which is standard UCT algorithm that considers durative and simultaneous actions. The second algorithm is called Portfolio Greedy Search (PGS). It is an iterative algorithm which works with a portfolio of predefined behaviors (scripts). Given a game state, it assigns the same script to all units. Then it iteratively, for each unit changes its script, performs a playout and if it ends better than the previous assignment it keeps this new script; if not, the last script is used. This is called improvement and it is done in turns for both armies first, one army is improved and then the other one.

Further improvements to the UCT were proposed by Justesen et al. (2014). Inspired by PGS they modify UCTCD to search in the space of script assignments instead of action assignments. This means that unit-actions in each player-action are generated only using predefined scripts.

Our approach, MCTS_HP is similar to the quality-based rewards from Pepels et al. (2014) in terms of using more information from the playout terminal state for assessing quality of MCTS states. We differ from the Pepels et al. (2014) in that we apply MCTS_HP in the script space, where the number of playouts made is severely limited (hundreds in our case, vs. up-to 10^7 in some action spaces) due to the fact that scripts are doing own local searches (e.g., pathfinding).

Combat Background

CotG can be played in more than two players. We, simplify the combat to a two-player game as, e.g., 1v1v1 scenarios are not frequent. Combat takes place in the solar system view (see Figure 2) and is played in turns as well. A player can issue attack and/or move orders to all their ships. Ships have hit-points (HP; represents the amount of damage a unit can take before being destroyed), strength (how much damage it causes during the attack), shields (which reduce incoming damage), attack range, and power that represents movement capabilities of a unit. Attack can be issued to a unit with positive strength, which did not attack this turn, and has an enemy units in its attack-range. Move can be issued to a unit with positive power. For each hex a unit moves it loses one power. Power is recharged at the start of each turn. Unit actions may interleave. Units need not to spend all their power during one turn but it is not transfered to the next turn. After a player ends the turn, they can not issue any more actions until their next turn.

As the result, the branching factor of the combat is huge and grows geometrically with the number of units (and their powers) involved in the combat. For small combat of six units each having the power of two (meaning one unit can move to 18 different hexes), assuming collision-free setup and not considering attack actions, we have the branching factor of 34 millions.

The amount of time used by AI for one turn is in hundreds of milliseconds rather then seconds, i.e., a human player will not cherish the situation when they need to await on 8 AIs taking their turns for more than several seconds.

Search techniques

In this section we introduce existing search techniques and our enhancements to these approaches.

Monte-Carlo Tree Search

MCTS tries to stochastically determine the value of nodes to avoid building the whole minimax tree. Opposed to Alpha-Beta pruning, search done in MCTS is not uniform but rather guided towards the most promising nodes which allows it to handle much bigger branching factors. MCTS algorithm starts with the current game state as the root node. It is iterative, and each iteration has four steps:

1. *Selection* – the game tree is traversed from the root. The most promising child is selected recursively, until a node which is not fully expanded (does not have all possible children) is found.

2. *Expansion* – a new child node is created for the node found in selection.
3. *Simulation* – a random playout from the new child node is performed.
4. *Backpropagation* – the result of the playout is propagated from the child node to the root node. Number of playouts and wins is updated for each node along the path.

Listing 1 shows the MCTS algorithm in pseudocode.

Probably the most common variant of MCTS is called Upper-confidence bounds for trees (UCT). It sees the child selection as an arm selection in the n-armed bandit problem (Auer, Cesa-Bianchi, and Fischer 2002). The algorithm selects child node which maximizes the value of upper-confidence bounds formula:

$$UCB1(i) = \frac{w_i}{n_i} + c * \sqrt{\frac{ln(n_p)}{n_i}} \quad (1)$$

Where i is the current node, w_i is the number of wins in the current node, n_i is the number of visits of the current node, n_p is the number of visits of the parent of the current node, and c is a constant usually determined empirically. In this work we use UCT and MCTS interchangeably. An extensive survey and background of MCTS methods can be found in (Browne et al. 2012).

Due to the immense branching factor and nature of the game, MCTS cannot be applied directly and some more clever approaches are necessary.

Listing 1: Regular MCTS algorithm

```
def MCTS(timeLimit, currentState):
  elapsed = 0.0
  root = new Node(currentState)
  while elapsed < timeLimit:
    selected = selection(root)
    newNode = expansion(selected)
    value = simulation(newNode)
    backpropagation(newNode, value)
    elapsed += Time.GetDeltaTime()

def selection(node):
  while true:
    if not node.IsFullyExpanded
      or node.IsTerminal:
      return node
    node = SelectBestChild(node)

def expansion(node):
  return node.GenerateNewChild()

def simulation(node):
  finalState = playout(node.State)
  if isWinner(player_1, finalState):
    return 1
  else
    return -1

def backpropagation(node, value):
  while node != null:
    node.VisitedCount++
    node.Value += value
    node = node.Parent
```

Scripts

Scripted behaviors are the simplest and most commonly used in computer games. In this work we focus on scripts controlling a single unit. We use scripts to express intention and let the implementation find some execution of this intention.

We define script as a function $s : G \times \mathbb{N} \rightarrow A$ where G is a set of all possible game states and A is a set of all possible actions; i.e., for a game state and index of a unit it returns an action this unit should perform. Some examples of scripts are: (Churchill, Saffidine, and Buro 2012)

- *Attack-Closest* – Find the closest unit and attack it. If the closest unit is not in weapons range, go towards it.
- *Attack-Value* – Find unit u with the highest $v(u) = \frac{damage(u)}{hp(u)}$ value in weapons range and attack it. If there is none, go to the closest unit.
- *No-Overkill-Attack-Value* (NOKAV) – Similar to Attack-Value but if a target unit was already assigned a lethal damage by another unit we ignore it.
- *Kiter* – If there are no units in weapons range, go to the closest unit. If there are units in weapons range, attack the one NOKAV selects. If you attacked, move away from the enemy.

Some sort of search or iteration is part of all the scripts presented above. This search is, however, local and focused – we know exactly what we are looking for and usually some sort of heuristic function is used to try the best candidates first.

MCTS in script space

This variant of MCTS assigns scripts rather than actions to units. For a set of n units $u_1, u_2, ..., u_n$ a node in the search tree is a game state with these units and an edge between two nodes is a vector of scripts $(s_1, s_2, ..., s_n)$ where script s_i returns a unit-action for unit u_i. We can easily map this vector of scripts to player-action[3] by giving current game state to each script which returns a unit-action and we can modify this state for the next script. E.g., a unit-action for unit u_i is $a_i = s_i(g_{i-1}, i)$, where $g_i = a_i(g_{i-1})$ and g_0 is the initial game state in the node. Player-action is then a vector of unit-actions $(a_1, a_2, ..., a_n)$.

A playout is performed by generating player-actions from random vectors of scripts. We first generate a vector of scripts $(s_1, s_2, ..., s_n)$ where $\forall i \in 1...n : s_i \in S$ and s_i is chosen uniformly randomly and S is a set of scripts the MCTS uses. Then from this vector, we generate a player-action as explained above and apply it to the game state. Then we do the same thing for the other player and iterate until we reach a terminal state.

[3]Cumulative action of all units for player in one turn.

Compared to the action space search, the branching factor is greatly limited by the number of scripts the units can choose from. For u units and s scripts we have a branching factor of s^u.

MCTS considering HP

Given enough iterations, MCTS will find better actions using random playouts, making iteration speed crucial. In strategy games, high iteration speeds are, however, not always achievable due to the immense complexity of the game. For example, Justesen et al. (2014) were able to achieve between 5 to 100 iterations during 40ms with their UCT algorithm in simplified simulation of StarCraft. In CotG, even with simplified game simulation and scripts we are not able to perform thousands of iterations in a 100ms time frame. Thus, we try to guide the search by specifying the problem a little better and modifying MCTS accordingly.

In standard UCT algorithm, a playout returns 1 or -1 representing which player won, making the UCT maximize the number of wins. But it is oblivious to the way how was the victory achieved and how does the winning state look. That is fine for games such as Go, where win/loss is all that matters. In many cases, however, the UCT is used only for part of the game and win/loss is just an interpretation of the whole game state. In strategy games, it is important to win the combat scenario, but it usually does not mean we won the whole game. Better definition of the problem is that we want to maximize enemy losses while minimizing ours. To account for this, we propose *Monte-Carlo tree search considering hit points* (MCTS_HP) algorithm.

MCTS_HP algorithm

The algorithm is identical to MCTS in script space, except for the simulation and backpropagation part. From simulation the MCTS_HP returns a value corresponding to the remaining hitpoints of the winning army. During the backpropagation, this value is normalized to a $[-1, 1]$ interval. This normalization is performed specifically at each node on the path from the child to root by dividing hitpoints remaining from the playout by hitpoints of the army in the given itermediate node.

For example, in Figure 3 the simulation starts with 210 HP of units. In a few moves, this is reduced to 10 HP. Then we perform a playout which ends with 9 HP of units remaining. That outcome is normalized in the node with 10 HP to value of 0.9 (we won while loosing only 10% of our units). However, in the node where we had our whole army with 210 HP, the normalization yields a value of 0.04 (from that state we lost most of our army).This reflects the fact, that the playout went pretty well (we won while losing only 1 HP), however, we may have done some bad choices between the 210 HP state and the 10 HP state.

Now let us look at the algorithm more formally. The parts which are different from regular MCTS discussed earlier are shown in Listing 2.

In MCTS_HP simulation returns:

$$playoutValue = hp(p_1) - hp(p_2) \tag{2}$$

Where $hp(x)$ returns sum of HP of all units of player x. Note that either $hp(p_1)$ or $hp(p_2)$ must be zero, otherwise the game state would not be terminal.

During backpropagation, this value returned from playout is in each node n on the path from leaf to root mapped to interval $[-1, 1]$ and added to the nodes current value as usual. The mapping is performed as follows:

$$nodeValue_n(playoutValue) = \frac{playoutValue}{hp_n(p)} \tag{3}$$

Where $hp_n(p)$ returns sum of HP of all units of player p for the state in node n.

$$p = \begin{cases} p_1, & \text{if } playoutValue > 0 \\ p_2, & \text{otherwise} \end{cases} \tag{4}$$

This means that p_1 tries to maximize the amount of HP remaining for his units and minimize it for enemy units. We can be sure that the $nodeValue$ is always in the $[-1, 1]$ interval because units cannot gain HP; therefore,

$$playoutValue \leq hp_{node}(p) \forall node \in path \tag{5}$$

Where path contains all nodes on the path from the leaf where the playout was performed to the root. Example of this mapping for one node can be seen in Figure 4.

Listing 2: MCTS_HP algorithm is identical to MCTS. Only instead of Simulation and Backpropagation we use Simulation_HP and Backpropagation_HP respectively.

```
def simulation_HP(node):
  finalState = playout(node.State)
  playoutValue =
    finalState.HP[player_1] -
    finalState.HP[player_2]
  return playoutValue

def backpropagation_HP(node, value):
  while node != null
    node.VisitedCount++
    p1_nodeVal=value/node.HP[player_1]
    p2_nodeVal=value/node.HP[player_2]
    if value < 0
      node.Value += p2_HP
    else
      node.Value += p1_HP
    node = node.Parent
```

Now we can look at the example in Figure 3 and apply our new terminology. Since we perform the mapping again in each node, in different nodes the value of a playout may mean different things. In the example we perform a playout from node with $hp_{leaf}(p_1) = 10$ and the playout ends up having $playoutValue = 9$, the mapped value in the leaf will be $nodeValue = 0.9$ which looks like a very good result. If we did not remap and just propagated this value up the tree, all nodes on the path would increase their value by 0.9 and their chances to be selected would increase. What if, however, having 10 HP was not very good in the first place? It could happen so that we started in an advantage with 210 HP, then ended up with only 10 and then managed to barely

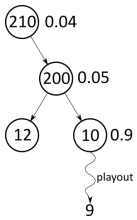

Figure 3: Example of possible backpropagation and mapping in MCTS_HP. Numbers in the nodes are HP remaining, i.e., $hp_n(p_1)$. The numbers next to nodes represent normalized values, i.e., $nodeValue$ of given playout with $playoutValue = 9$.

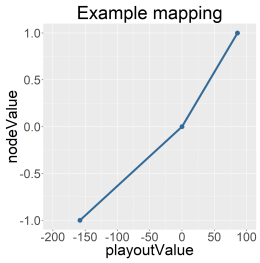

Figure 4: Example mapping of playout value to interval $[-1, 1]$ for a node n with $hp_n(p_1) = -158$ and $hp_n(p_2) = 86$.

win with 9 HP. By remapping the HP value at each node, we can preserve this information and in the starting node with $hp_{start}(p_1) = 210$ this will be considered very weak win.

As we see in Figure 4, the mapping is linear for both subintervals $[-1, 0)$ and $[0, 1]$. In case of this mapping very weak wins and very weak loses, i.e., nodeValue in small interval around 0, change the actual value of a node only slightly. However, the binary result (win/loss) is still completely opposite. This may be problematic for games where we care more about the binary result. To give more value to the win/loss result we could push the mapping interval away from 0 and map for example to interval $[-1, -0.8] \cup [0.8, 1]$ or we could just use different function than linear.

Application of this approach is not limited just to script space searches. This approach would work the same in a regular MCTS in action space and other MCTS variations. And it is applicable to problems where the goal is not only to win, but we also care for quality of the victory and it is quantifiable (such as HP in our case). Values other than HP could also be used in our case such as unit value.

Related work

Since this enhanced MCTS method proved to be very useful and to our knowledge was not used in context of strategy games we researched other areas and found a similar technique in (Tom and Müller 2009) where the authors use enhanced UCT on a game called Sum of Switches (SOS). One of the enhancements, called Score Bonus, should distinguish between strong and weak wins by returning a value in interval $[0; \gamma]$ for losses and in $[1 - \gamma; 1]$ for wins from the playout. The values are scaled linearly in these intervals. Values 0.1, 0.05, and 0.02 were tried as γ, however, the Score Bonus was unable to improve the gameplay of UCT in SOS.

The authors do not elaborate why did the Score Bonus fail to improve performance in SOS, although they mention that value of $\gamma = 0.02$ slightly improved performance in 9x9 Go. Results of our approach are contradictory, but our problem and method are different as well. First, we use much wider range of values from the whole interval $[-1, 1]$ and more aggressively exploit the entire range of different states. Second, in our case the algorithm closely matches the problem. In SOS and Go, the UCT solves the whole game whereas in our case the UCT solves just a subproblem, the result of which matters to the rest of the game.

Experiments

To evaluate different AI players, we performed several round-robin tournaments with different number of units. Battles were asymmetrical in sense of positioning. For each army, a center point was chosen randomly and around this point units were randomly distributed. We used two basic types of units, both without any upgrades. Destroyer — a unit with high damage but short range and low HP. And Battleship — a unit with medium weapon range, lower damage, but high HP and shields. Our results present average of different ratios of these units to show common case scenarios, e.g., an army in 5vs5 scenario may contain 3 battleships and 2 destroyers, or 1 battleship and 4 destroyers etc.

Because the game is turn-based and one player must go first, we evaluate each battle scenario from both sides. First, one player controls army 1 and the other player controls army 2. Afterwards, we switch sides, i.e., army 1 always goes first but once player 1 controls it and once player 2 does. We call this battle where players switch sides symmetric battle and if a player manages to win both battles in a symmetric battle we say that he achieved a sym-win where winning each sub battle is simply called win. To bring the scenario closer to reality of the game, we also sum HP of all the ships of the winning player. If a symmetric battle ends 1:1 we look at how many HP did each player's ships end with and the one with more HP left achieves a sym-win.

Algorithms we chose for our tournament were the two scripts, Kiter and NOKAV, where the NOKAV script behaves very similarly to the current AI present in the game. PGS with different response and iteration counts which is encoded as PGS_I_R where I is improvement count and R is response count. Specific configurations are PGS_1_0, PGS_3_3, PGS_5_5. All PGS algorithms have time limit per Improvement set to 500 milliseconds. To be competetive to other approaches we introduced playout caching to PGS to improve playout evaluation counts. MCTS and MCTS_HP with three different execution time limits were chosen. The encoding is MCTS_TIME or MCTS_HP_TIME where TIME stands for time in milliseconds. Times were 100ms, 500ms, and 2000ms. All experiments were performed on Intel Core i7 2600K @ 4.5Ghz with 16GB of DDR3 RAM, SSD, running Windows 10 Professional.

Error bounds in the graphs represent 95% confidence interval. Because the MCTS algorithms are not deterministic

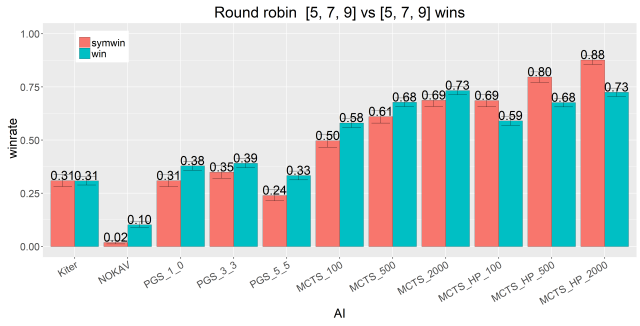

Figure 5: Average win rates of 5vs5. 7vs7 and 9vs9 round robin tournaments.

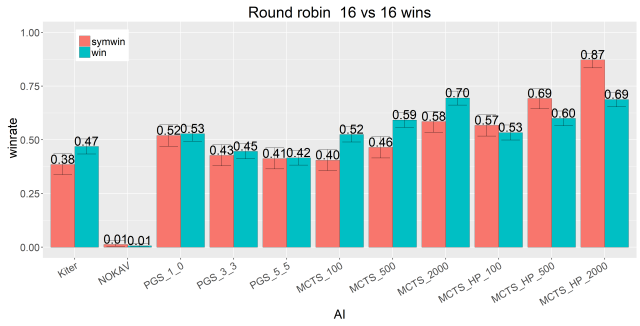

Figure 6: Win rates of 16vs16 round robin tournament.

we run each battle 5 times and consider them as separate battles. All the algorithms, which need a portfolio of scripts, use Kiter and NOKAV. Based on preliminary results we use a very simple move ordering for MCTS methods where we try the assignments involving the Kiter script first.

Results

Results of our experiments are in figures 5, 6, and 8. Execution time statistics are in figure 9. More complex MCTS approaches work well on small to medium scale combat and at larger sizes greedy PGS approaches prevail. MCTS_HP proved to be equal or superior to the regular MCTS in all scenarios. The NOKAV script, representing the AI currently

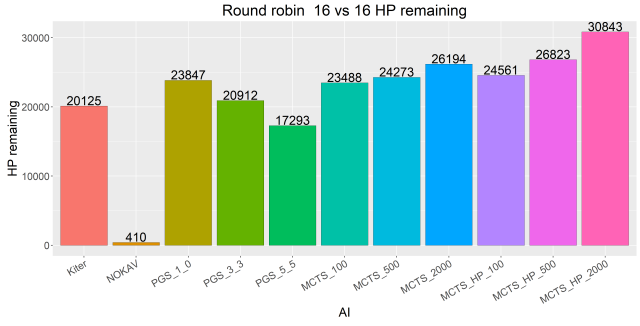

Figure 7: Summed HP remaining of ships after all battles in the 16vs16 round robin tournament.

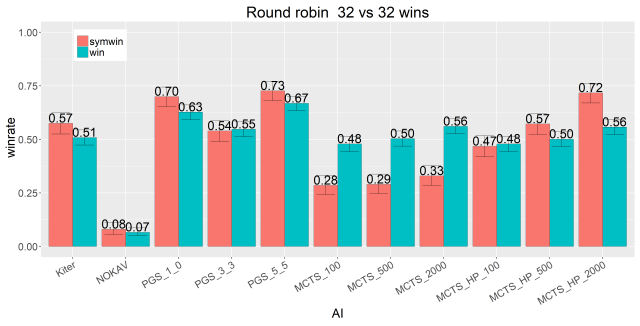

Figure 8: Win rates of 32vs32 round robin tournament.

implemented in the game, was easily outperformed by all other methods.

Interestingly, MCTS_HP does not achieve higher win rates than MCTS given the same time limit. This may be because our mapping for MCTS_HP is linear and does not distinguish much between weak win and weak loss. This, however, corresponds to how combat in strategy games works. If from a whole army of units only one remains after a combat, the result is closer to a draw than to a win and it is very similar to situation when only one enemy unit remains.

In figure 7 we see summed HP remaining after all battles in the 16vs16 tournament. This is complementary information to our sym-win metric and it shows how strong were these symmetric victories. In the game, when an AI engages in a series of battles, no matter how many are won and how many are lost, if it is able to keep more units (in the figure summed as HP) it will be in better strategic position and it will eventually prevail and win the game. In our experiments, the sym-win rate roughly correlated with the HP remaining. In the average 5, 7, and 9 unit tournaments the graphs match almost perfectly. In the 32vs32 case, the HP remaining graph was much less dramatic, and even the worse performing MCTS approaches were able to score almost as much HP remaining as the best MCTS_HP approach.

Since the authors of the PGS algorithm tried only the PGS_1_0 in their work, we did not know how the versions with higher iteration counts would work. Surprisingly, increasing the response and improvement counts does not reliably improve win rates. This stems from the nature of the greedy approach but still, the simplest and fastest PGS_1_0 which just improves the players script assignment once against a fixed opponent works sometimes much better than the variations with more iterations. Especially when considering the execution times (even with our caching optimizations), the PGS_1_0 is the best PGS variation in view of performance/time ratio.

Based on these results we propose to use PGS_1_0 for large combats and when some units are destroyed we can switch to more accurate MCTS_HP which even with a few hundreds of milliseconds time limit outperforms all other methods.

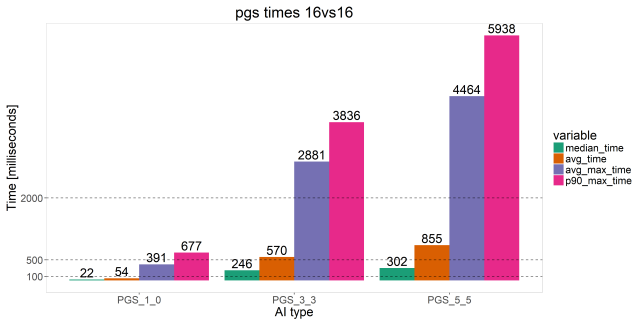

Figure 9: Statistics of execution times for all battles in 16vs16 tournament. MCTS algorithms operate on a fixed time limit and are represented by horizontal lines in 100, 500, and 2000 milliseconds. avg_max_time represents average of maximum values. p90_max_time represents 90th percentile of maximum values.

Conclusion and future work

In this paper we presented extensions to the UCT algorithm in context of unit combat in a commercial 4X game Children of the Galaxy (CotG). Full integration of the AI system to the game is work in progress. To evaluate different combat approaches we used our custom simplified combat simulator called CotG Micro Simulator[4]. The first improvement was MCTS in script space which is similar to the work of Justesen et al. (2014) and searches in space of script assignments instead of unit actions. The second improvement was MCTS considering HitPoints (MCTS_HP) which is based on MCTS in script space but from playout returns a real value in interval $[-1; 1]$ instead of a binary value representing win/loss. The real value represents HP remaining for the winning player, i.e., the quality of the victory. This in turn allows the search to be guided towards more promising states.

Results of our experiments indicated that MCTS_HP is equal or better than a regular MCTS in script space and is well suited for small to medium scale combat. Three Protfolio greedy search (PGS) variations were also evaluated and the simplest one proved to be a viable option for medium to large scale combat where the MCTS methods require too much time to operate effectively. The PGS variations with more iteration counts proved to have questionable performance and unreasonable execution time requirements even with playout caching enabled. Since our MCTS algorithms had only about a hundred iterations for more complex scenarios and low time limits we also shown that MCTS is viable option even in these scenarios where it is not possible to perform hundreds of thousands of playouts.

Future work would be to use our game simulation to implement and benchmark more AI approaches for CotG and turn-based games games in general. To enable faster playout and more iterations some pathfinding methods other than A* could be explored. Variations of the MCTS_HP with different mapping function could be also tried and compared to see the effect of giving more (or less) value to win/loss states. MCTS_HP could be modified to consider not HP but for example value of a unit.

Our improved version of MCTS dominated the regular MCTS approach in our tests. It would be very interesting to see how this algorithm would perform in other games. Our assumption is that in games such as StarCraft where the combat is just a subproblem of the whole game and it is very important how well or bad does it end, MCTS_HP would improve the performance and guide the search better. A full game playing agent with MCTS_HP algorithm for combat should perform much better because accumulating units over time should lead to ultimate victory.

[4]https://bitbucket.org/smejkapa/cog-ai

Acknowledgments

This Research was funded by the Czech Science Foundation (project no. 17-17125Y).

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3):235–256.

Bowling, M.; Burch, N.; Johanson, M.; and Tammelin, O. 2015. Heads-up limit holdem poker is solved. *Science* 347(6218):145–149.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in starcraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 1–8. IEEE.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for rts game combat scenarios. In *AIIDE*, 112–117.

Justesen, N.; Tillman, B.; Togelius, J.; and Risi, S. 2014. Script-and cluster-based uct for starcraft. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 1–8. IEEE.

Pepels, T.; Tak, M. J.; Lanctot, M.; and Winands, M. H. 2014. Quality-based rewards for monte-carlo tree search simulations. In *ECAI*, 705–710.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484–489.

Tom, D., and Müller, M. 2009. A study of uct and its enhancements in an artificial game. In *Advances in Computer Games*, 55–64. Springer.

Yoochul, K., and Minhyung, L. 2017. Humans Are Still Better Than AI at StarCraftfor Now. https://www.technologyreview.com/s/609242/humans-are-still-better-than-ai-at-starcraftfor-now/. Accessed: 2018-06-27.