Turning Zeroes into Non-Zeroes: Sample Efficient Exploration with Monte Carlo Graph Search

Marko Tot, Michelangelo Conserva, Diego Perez Liebana
Queen Mary University of London
London, United Kingdom

Sam Devlin
Microsoft Research
Cambridge, United Kingdom

***Abstract*—Monte Carlo Tree Search (MCTS) has proven to be a staple method in Game Artificial Intelligence for creating agents that can perform well in complex environments without requiring domain-specific knowledge. The main downside of this planning based algorithm is the high computational budget needed to recommend an action. The fundamental cause of this is a vast search space caused by a high branching factor, and the difficulty to create a good heuristic function to guide the search without leveraging domain-specific knowledge. Recent advances in the field proposed a new planning based method called Monte Carlo Graph Search (MCGS), which uses a graph instead of a tree to plan its next action, reducing the branching factor and consequently increasing the performance of the search. In this paper, we propose several modifications that optimize the performance by increasing the sample efficiency of MCGS. The use of frontier for node selection, improving the rollout phase by doing stored rollouts, and a generalized approach to guide the search by incorporating a domain-independent online novelty detection method. Together these enhancements enable MCGS to solve sparse reward environments while using a significantly lower computational budget than MCTS.**

I. Introduction

Monte Carlo Tree Search (MCTS) [1] is a family of algorithms based on Statistical Forward Planning (SFP). They operate by searching for the optimal action to take given the current state of the environment and work by constructing a tree of possible future states that are used to select the best action.

The main downside of Monte Carlo methods is the number of rollouts needed to get a precise estimate of the state value, which takes up a large portion of the computational budget and limits their use in real-time tasks. As these rollouts are typically done with a random policy for enacting actions, i.e. the agent chooses its actions without any heuristics, these methods rely on the Law of Large Numbers to get a precise estimate of the values for each action given a particular state [2]. To ascertain the value of a node, Monte Carlo methods need to do the rollout step for every new node they encounter, so their performance is directly reliant on the number of new nodes. Even if a single step in the environment is fast, the sheer amount of rollouts required to choose a single action limits the applicability of these methods, especially in sparse reward environments that require a lot of exploration. In sparse reward environments, most of the rollouts don't obtain any valuable information, thereby effectively wasting the computational budget.

Monte Carlo Graph Search (MCGS) is a modification of the standard MCTS algorithm, in which the tree structure created by the search is changed into a more generic graph. This modification has proven to increase performance in environments where the reward signal is rich [3]. However, MCGS has not previously been applied to sparse reward environments, in which MCTS methods are known to struggle.

This study presents a combination of enhancements that can be added to the current state of the art MCGS algorithm to decrease the necessary computational budget while surpassing the performance of the currently available solutions. This is achieved by increasing the sample efficiency of the search through improving the exploration rate. We propose the addition of three modifications, a maintained frontier of nodes to reduce the overhead of the algorithm, storing of the nodes during the rollouts, which pushes the frontier further away from the currently explored space, and a novelty metric that is used as the primary factor for selection phase before a reward is obtained. Together these enhancements enable MCGS to solve sparse reward environments while using a significantly lower computational budget than MCTS.

II. Related Work

MCTS performs the search by building a tree of future possible game states and can be summarised in four steps, reported in Figure 1. These four steps compose one iteration, and iterations are repeated until the computational budget is depleted, and the best action obtained from the tree structure is enacted.

i) *Selection*. Starting from the root node, i.e. the current state of the game s_t, a child selection policy descends through the tree until it reaches a leaf node s'_j.
ii) *Expansion*. The selected leaf node is expanded by adding child node/nodes, based on the actions available to the agent from that state.
iii) *Rollout*. A Monte Carlo simulation using a random rollout policy is used to approximate the return from the new child state. $\widehat{\mathcal{J}}(s'_j) = \sum_{t=0}^{\infty} \mathcal{R}(s'_{j+t}, \pi_{\texttt{Random}})$
iv) *Backpropagation*. The result of the rollout is used to update the value of the trajectory from the expanded node

up until the root. This value is subsequently used by the selection policy during the next iteration.

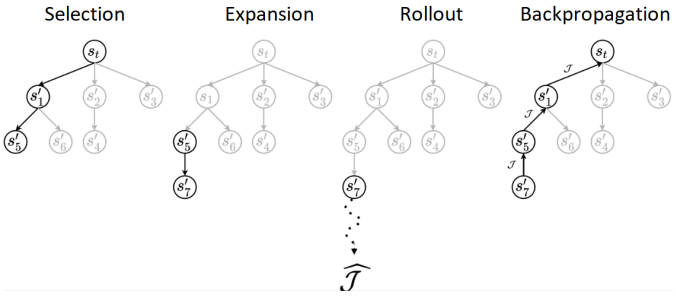


Fig. 1: Monte Carlo Tree Search steps.

The reason in favour of using trees for performing the search lies in the theoretical simplicity of such a structure, which allows defining simple yet effective rules for applying search methods. However, the assumption of a tree structure does not accurately portray the underlying structure of a game state space. This simplicity of the tree structure comes at the cost of redundancy of states [3], as different trajectories may lead to the same state [4] and result in the same states appearing multiple times in the tree, unnecessarily increasing the size of the structure. Reducing the total number of nodes in the graph leads to a lower number of rollouts required to evaluate them, which in turn reduces the required computational budget.

When two identical nodes are merged, the tree becomes a graph. Transposition tables [5] have been used in tree search methods, as a way of combining identical nodes. Utilising this new state-space created by merging nodes required adaptation of the selection and backpropagation steps [6]. On one hand, this transformation causes the search space to lose part of its simplicity and increase the computational overhead of the algorithm. On the other hand, the budget required to search over the state-action space can be dramatically reduced. An example of such transformation is shown in Figure 2.

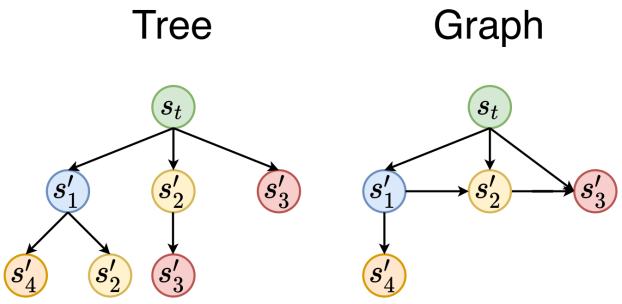


Fig. 2: The graph obtained by merging identical states.

Techniques that reduce the number of states in the structure are particularly useful in environments where the reward signal does not give enough insight to guide the algorithm. Many of these environments have sparse rewards, where for almost all of the states, the agent does not get any reward. This means that the agent is not receiving informative feedback for the actions it takes, practically transforming MCTS into an uninformed search algorithm, and uninformed search is directly affected by the size of the state-space. Recent work has also shown promise of using novelty in MCTS to guide the exploration. Determining novel states can be an effective way of enhancing the selection process in cases where nodes have the same value. Several criteria have been used in order to determine the novelty of the state. Heuristic novelty [7] separates the nodes into novel and not-novel based on the received rewards. ϕ-Exploration bonus algorithm [8] detects novel states based on the probability distribution of variables, while feature-based pseudo count builds on top of this idea but instead of using atomic variables uses probability distribution over composite features. An example of a feature for a game of chess could be "Black still has both knights" [9].

However, the addition of novelty methods with planning based algorithms has only been used on top of an already existing heuristic evaluation, still relying on domain knowledge and handcrafting of the reward signal for the specific environment. While this enhancement shows that novelty can provide additional information to the agent, it doesn't tackle the issue of sparse rewards. Furthermore, the use of graph based planners has not been explored in sparse reward environments where sample efficiency and exploration techniques are of a much higher priority.

III. Method

Our proposed algorithm is a highly modified version of the state of the art SFP algorithm called Graph Based Planner (GBP) [3]. GBP uses a graph structure as the basis for the search, with Upper Confidence Bound for Trees (UCT) [10] formula for the selection step, without doing any rollouts. By skipping the rollout step this algorithm spends most of its computational budget on expanding new nodes, which allows it to explore a large state space. However this way of exploration also heavily impacts the memory consumption of the algorithms, as each expanded node needs to be added to the graph.

Another available planner which uses graphs is Stochastic Graph Based Planner (S-GBP) [3], which uses the GBP as its basis but also incorporates the rollouts and also keeps value bounds for each of the nodes instead of a single value.

As MCTS allows multiple identical states to exist in the tree, the size of the created structures doesn't effectively represent the explored state space. Graph search brings two main benefits: it allows aggregation of states and easy detection of obsolete actions.

The agent interacts with the environment through different actions. At each step of the game, it must choose one of the given actions in order to progress the game to the next state. Some actions, however, might not have any impact on the game due to special circumstances. Using action 'Drop object' while carrying no objects, doing 'Move Forward' while being in front of a wall, trying to unlock the door without carrying the correct key, or being in the middle of a cut-scene where actions might not have any effect on the game.

While still being registered as actions, the game will progress, but the underlying game state might not change.Any action that doesn't affect the game state is an obsolete action. If an obsolete action is registered during the expansion phase while using MCGS, the node is not added to the graph, and the rollout and backpropagation phases can be skipped as those states have already been evaluated, and the value was already backpropagated through the graph.

We present three different modifications that increase the sample efficiency of the basic MCGS.

A. Frontier

A common technique for selecting a node for expansion in the Monte Carlo family of algorithms, also used in GBP, is the UCT formula, which balances the exploration and exploitation of the game space. We propose an alternative node selection approach consisting of maintaining a set of leaf nodes (frontier) that are yet to be expanded. Every node added to the graph is also added to the frontier. The selection of the node is done according to Equation 1, which includes the UCT formula as well as added random noise.

$$\texttt{SelectedNode} = \underset{s \in \mathcal{S}}{\operatorname{argmax}} \left(\text{ucb}(s) + \epsilon\right) \tag{1}$$

where $\epsilon \sim \mathcal{N}(0, \sigma)$, $\sigma = \max_{n \in J} \text{ucb}(s)$, and $\mathcal{S}$ represents the set of selectable nodes in the frontier. The frontier also allows for a simpler process of updating the graph and determining which nodes are reachable from the root node. Since the complexity of the graph structure can make certain nodes in the graph disconnected from the root after each step in the environment, those nodes have to be omitted from the node selection process in the next step. Even though marked as unfit for the node selection process, these nodes are still kept in the graph as a potential merging of the states could render those nodes reachable again through a different action trajectory.

B. Stored rollouts

In MCGS, rollouts are done in order to ascertain the value of a specific node. A valuable addition to the rollout step could be adding the nodes visited throughout the rollout to the graph. This technique has previously been tested with the MCTS [11]. On one hand adding additional nodes to the underlying structure during the rollout phase provides a meaningful difference in the exploration of the state space. On the other as the information about the visited states needs to be stored in the structure, increasing the memory consumption of the algorithm. When used with MCTS, every state visited during the rollout is added to the tree structure. Those trajectories are likely to be non-optimal due to the randomness of the rollouts and redundancy of the state. Compared to MCTS, the graph structure in MCGS allows for inter-connectivity between nodes and does not require us to store every visited state, just the newly explored ones, and the optimality of the path generated during the rollout can be modified by detecting obsolete actions and merging the states. Alleviating some of the memory issues as well.

C. Novelty

Novelty is detected by online analysis of the states using a count-based technique. During the search, after a new state has been encountered, features of the state are compared to the ones already stored in the graph. For each feature, we count how many times its value has been seen in the graph, and the feature of that state is regarded as novel if its occurrence rate is below a certain threshold. For every novel feature in the state, the corresponding node in the graph is given a novelty bonus. Including novelty as a factor adds a new component to the selection step (Equation 2).

$$\texttt{SelectedNode} = \underset{s \in \mathcal{S}}{\operatorname{argmax}} \left(\text{ucb}(s) + \epsilon + \text{novelty}(s)\right) \tag{2}$$

Novelty inheritance is also enabled. If a node is reached from a novel node it gets 50% of the parent's node novelty. This encourages further exploration from the area where a novelty was found. Including novelty in the algorithm creates a reward signal that guides the algorithm to select nodes from the frontier which are new and different, consequently improving exploration.

Novelty methods can also be combined with adding the nodes during the rollout. A potential issue with maintaining the graph created in this way would be the requirement to store a large number of nodes. Restricting the addition of nodes to only add the trajectories which contain novel states could alleviate the issue of rapidly increasing graphs, i.e. decrease the memory consumption, while preserving most of the benefits regarding the exploration.

Based on these modifications we present several different ablations of the algorithm[1] based on types of enhancements that were enabled:

- MCGS - Monte Carlo Graph Search with the frontier
- MCGS+R - MCGS with stored rollouts
- MCGS+N - MCGS with novelty search
- MCGS+RN - MCGS with stored rollouts and novelty
- MCGS+R*N - MCGS with stored novel rollouts and novelty

IV. Environment

In this study, we chose MiniGrid [12] for the evaluation of the agent. MiniGrid is a staple environment for assessing the performance of the algorithms on grid-based levels [13]–[16]. The agent has been tested in three different versions of the environment, Empty, DoorKey and Stochastic DoorKey. In the Empty environment, the agent needs to reach the goal located in the grid, while in the DoorKey version there is an additional requirement of picking up the key, unlocking the door, and then reaching the goal. The stochastic setting also adds a chance of action failure for each enacted action.

The state representation, i.e. the observation given to the agent, consists of features extracted from the environment, and the grid that represents the map. The choice of state

[1]The code is available at: https://github.com/markotot/MonteCarloGraphSearch

representation is crucial for the creation and maintenance of the graph and influences the performance of the algorithm as different state representations create different graph structures.

An example of using the novelty method in MiniGrid can be seen in Figure 3. This state is not novel based on the *Has Key* feature, where the value *False* is present in 572 of out 602 nodes in the graph. On the other hand, if the value was *True*, the state would be regarded as novel due to the low occurrence rate. The effect of utilising novelty search with the reward signal in MiniGrid is presented in Figure 4, where each cell in the heat map represents the mean value of the signal the agent received for that position after backpropagation.

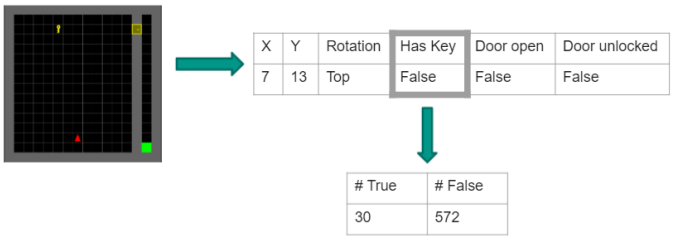
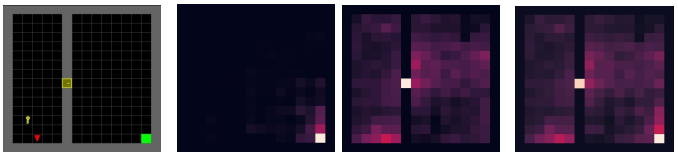


Fig. 3: Novelty detection in MiniGrid environment



(a) Level (b) Reward (c) Novelty (d) Combined

Fig. 4: Level layout and the corresponding value signals.

V. Experiments

Figure 5 showcases the difference between the structures created after only a couple of iterations of MCTS and MCGS, caused by detecting obsolete actions and merging of the identical nodes in a sample MiniGrid level. These two structures contain the same amount of information, the only difference is in the redundancy of nodes in the tree.

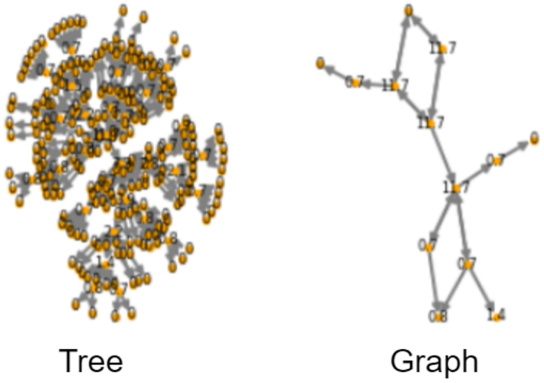


Fig. 5: Difference in the tree and the graph structures caused by merging identical states and detecting obsolete actions

A. Empty environment

Table I shows the comparison between the state of the art planning algorithms and MCGS with different modifications on an Empty 16×16 environment. We chose vanilla UCT based MCTS, with the exploration constant set to 1, as the baseline, and GBP and S-GBP [3] as they represent the most recent work on MCGS that was created for the similar, gridworld environment in mind. The cut-off was set to 99 steps: if the agent can't reach the goal in 99 steps, the game ends. One step in the environment corresponds to one action.

The performance of the agent is measured with two indicators: whether the goal was reached at all, and how many steps were used to do so. All algorithms were run on the same budget of 8000 forward model calls (FMC) per step.

TABLE I: Different versions of MCGS compared with state of the art planning methods, tested in Empty 16x16 environment.

Algorithm	Steps $Mean \pm STD$	Solve rate
MCTS	99 ± 0.0	0%
GBP	26.0 ± 0.0	100%
S-GBP	33.7 ± 0.8	100%
MCGS	34.1 ± 1.33	100%
MCGS+R	30.0 ± 1.24	100%
MCGS+N	31.4 ± 3.68	100%
MCGS+RN	29.1 ± 2.24	100%
MCGS+R*N	29.7 ± 2.85	100%

Even in this simple setting, MCTS fails to solve the environment in any of the runs. All of the MCGS variants however manage to finish all of the levels. With the small state space of only 782 different states, all graph based algorithms have been able to explore every state with little difference in the average number of steps required to solve the level. Adding novelty bonuses, or rollout nodes doesn't impact the performance of the algorithm in this simple environment.

It is also evident that GBP has the best performance in this environment. As the number of different states is very low, it is more effective to skip the rollouts completely and brute force the search.

To analyse the effects of a larger state space on the performance of the algorithms, the environment was changed from Empty to DoorKey.

B. DoorKey environment

DoorKey environment is difficult for planning algorithms that use no domain-specific heuristic for two reasons. Firstly, the large grid makes the sparsity of the reward problem much more evident. Some of the level layouts require the agent to do up to 60 actions to get the reward. Secondly, the presence of the inventory, i.e. the ability of the agent to pick up, and more importantly drop the key. This interaction allows the state space to expand extensively as dropping the key on any spot in the grid creates a whole new set of states. These two factors combined make this environment truly difficult for a planning algorithm, as the chance of the agent traversing the grid from the key to the door without dropping the key somewhere on the way due to random rollouts is extremely low. For a uniform random rollout of 50 actions, which is used for MCGS, the

TABLE II: Different versions of MCGS compared with state of the art planning methods, tested in DoorKey 16x16 environment.

Algorithm	Checkpoint Discovered			Solved	Step	Nodes	New node
	Key Found	Door Opened	Goal Found		$Mean \pm STD$	$Mean \pm STD$	per FMC (%)
GBP	100%	100%	96%	64%	80 ± 20	$103,403 \pm 24,246$	16.17
S-GBP	100%	96%	68%	0%	99 ± 0	$44,610 \pm 10,229$	0.17
MCGS	100%	40%	24%	16%	95 ± 10	$8,939 \pm 1,048$	1.18
MCGS+R	100%	92%	60%	56%	81 ± 22	$32,304 \pm 9,559$	4.99
MCGS+N	100%	100%	96%	92%	60 ± 22	$6,276 \pm 2,113$	1.31
MCGS+RN	100%	100%	100%	100%	45 ± 9	$31,371 \pm 7,924$	8.64
MCGS+R*N	100%	100%	96%	96%	55 ± 20	$6,006 \pm 1,985$	1.34

probability that the key isn't dropped during the rollout is 1 in 2225. Changing to this environment increased the state space from 782 to approximately $600,000$.

Each algorithm was run 25 times, each time on a different level layout, with a budget of 8000 FMC per iteration. The results are presented in Table II. The solved rate and the mean number of steps were used to determine the effectiveness of the algorithms at completing the level, while the number of nodes measured the memory consumption of the algorithm. In addition, the discovery rate for three different checkpoints has been measured: *Key Found*, *Door Opened*, *Goal Found*. Each of them represents if the state where the agent collected the key, opened the door, or found the goal was encountered during the search. Note that discovering these checkpoints does not give any reward to the agent, they have only been used for assessing the search efficiency of each method. Finding the goal checkpoint indicates that the agent discovers the goal state during the search, while solving the environment on the other hand means that the agent actually reached the goal.

It is evident that basic MCGS isn't able to solve the majority of the levels, reaching the step limit on most of the level layouts (84%). Large state space and the sparsity of the rewards cause the rollouts to be ineffective in obtaining any useful information about the value of the states, in order to guide the further search. The addition of novelty bonuses or storing nodes that are encountered during the rollouts increases the performance of the algorithm and allows it to solve some of the levels, showing that on their own, both of these modifications can be used to improve exploration. MCGS+R, solves 56% of the levels, while MCGS+N was able to reach the goal in 92% of the tested levels. Using both modifications at the same time improves the performance of the search even further. Adding all of the nodes found during the rollout, in conjunction with novelty bonuses reaches the overall solve rate of 100%. The benefit is also evident from the lowest average number of steps. Combining the two enhancements also provides a high new node per FMC of 8.64%, the highest between all of the rollout based methods. This means that the exploration rate is 7 times higher compared to MCGS which doesn't utilise stored rollouts or novelty and 50 times higher than S-GBP. Having the advantage of being able to select the node far away from the root which comes from storing rollout nodes, with knowing which of these nodes are promising enabled by novelty detection creates a cohesion that boosts the performance more than each enhancement separately.

Finally, there is a huge increase in the number of nodes that are added to the graph in the modification with the rollouts. This can greatly impact the memory consumption of the algorithm which also has to be taken into account. As the complexity of the environment grows so will the state space and potentially the state representation as well, so keeping the number of stored nodes low while maintaining the performance is certainly a point of interest. One tested modification includes only adding the paths to the novel nodes during the rollout phase (MCGS+R*N). This procedure decreases the number of stored nodes to a level similar to not adding rollout nodes while outperforming both single modification algorithms. However, this constraint does affect the ability of the algorithms to solve the hardest levels which can be seen when comparing the solve rate and the increase in the number of steps required to solve the environment.

Both MCGS+RN and MCGS+R*N outperform the current state of the art planning methods. Given enough computational budget to search a significant portion of the space, GBP can solve 64% of the levels. Without spending the budget on rollouts, this method can search through a large amount of space, creating $103,403$ nodes, which amounts to 16% of the whole state space. We have to be mindful of the number of nodes stored in the graph as well. As this is 3 times higher than in MCGS+RN $(31,371)$, and 15 times higher compared to MCGS+R*N $(6,006)$ the memory consumption is also significantly higher for GBP. S-GBP often discovers the goal but does not solve the environment. It also on average stores $44,610$ nodes, significantly fewer nodes compared to GBP, but also still more than both MCGS+RN and MCGS+R*N.

To further compare approaches, additional metrics for each of the subgoals are presented separately in Table III. Step metrics represent the performance of the agent in regards to optimality, with the ideal result being 1.0, meaning every subgoal was discovered in the first iteration of the algorithm. The number of FMC reflects the computational budget in which the algorithm can solve the environment.

Both MCGS variants outperform the state of the art planning algorithms in both metrics. MCGS+RN is the most sample-efficient on average spending $53,893$ FMC to discover the goal, while MCGS+R*N presents itself as a promising alternative solution, which allows for a trade-off between computational budget and memory consumption while maintaining most of the optimality of MCGS+RN. Compared to MCGS+RN, it used several times more FMC but reduced the

TABLE III: Subgoal discovery metrics for GBP, S-GBP, MCGS+RN and MCGS+R*N in DoorKey 16x16 environment.

Algorithm	Key Discovered $Mean \pm STD$		Open Door Discovered $Mean \pm STD$		Goal Discovered $Mean \pm STD$	
	Step	FMC	Step	FMC	Step	FMC
GBP	8.5 ± 3.7	663 ± 771	22.9 ± 6.3	$38,681 \pm 69,664$	38.0 ± 10.4	$442,645 \pm 201,733$
S-GBP	1.0 ± 0.0	840 ± 869	15.8 ± 21.9	$118,790 \pm 171,800$	46.1 ± 30.6	$364,422 \pm 256,963$
MCGS+RN	1.1 ± 0.3	$1,944 \pm 2,697$	3.7 ± 3.3	$24,852 \pm 26,735$	6.8 ± 5.7	$53,893 \pm 45,323$
MCGS+R*N	1.0 ± 0.0	$1,509 \pm 1,476$	12.5 ± 13.7	$95,051 \pm 108,925$	19.6 ± 17.1	$155,486 \pm 135,296$

TABLE IV: Main MCGS versions compared with state of the art planning method, tested in a stochastic DoorKey 16x16 environment.

Algorithm	Checkpoint Discovered			Solved	Step $Mean \pm STD$	Nodes $Mean \pm STD$	New node per FMC (%)
	Key Found	Door Opened	Goal Found				
S-GBP	100%	88%	56%	0%	99 ± 0	$36,081 \pm 8,606$	4.62
MCGS+R*N	100%	100%	80%	80%	74 ± 22	$4,821 \pm 3,014$	0.81
MCGS+RN	100%	100%	100%	100%	59 ± 17	$29,894 \pm 9,071$	6.25

number of stored nodes by a similar factor. This trade-off could be useful when the state representations become large enough that high memory consumption becomes an issue.

Although MCGS+RN shows the best performance it is necessary to mention that the average number of steps to complete the level is still not optimal. After 50 runs with different seeds on the same 16x16 DoorKey environment, the average number of steps required to solve the environment was 53.5 ± 7.8, while the optimal route takes 40 steps.

There are two reasons for this sub-optimal solution. First, it takes the algorithm multiple steps to explore the environment and discover the goal, i.e. it depletes the computational budget several times before discovering it. Until the agent discovers the goal, it is purposelessly moving around the environment. For a harder level layout, where the optimal route would take 40 steps, the average number of steps required to discover the goal was 7.2 with the standard deviation of 6.9. Secondly, due to the random rollouts, there is a high chance the path itself is not optimal, which is manifested by the agent occasionally spending an action to drop the key along its path to the goal. The average discovered path length was 46.3 ± 4.8, 6.3 more than the optimal route. Combined, these two factors are the cause of the disparity between the optimal and the obtained solution. Figure 6 shows the comparison between optimal solution and the solutions from different seeds of MCGS+RN.

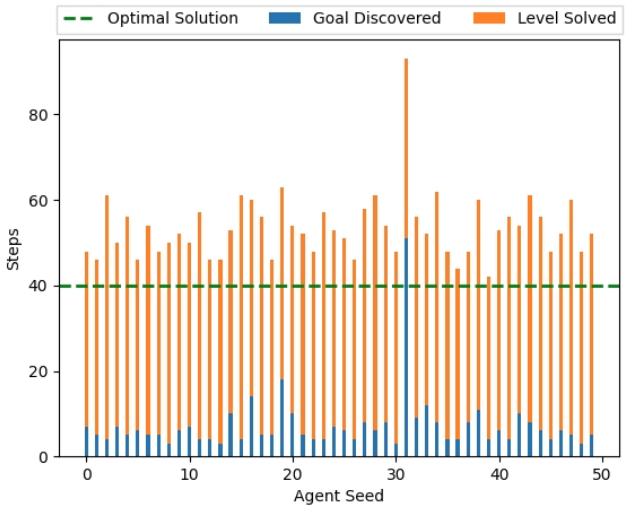


Fig. 6: Impact of agent seed on the performance of MCGS+RN.

C. Stochastic DoorKey environment

The most effective variants, MCGS+RN and MCGS+R*N were then tested on a non-deterministic version of the DoorKey environment. In this setting, every action has a 20% chance of failure, meaning that instead of the selected action, the agent might perform a *DoNothing* action. Table IV shows the performance of the algorithms when presented with an environment with stochastic elements. The GBP version of the algorithm was excluded from the comparison as it was not able to operate in a stochastic environment. Each algorithm was run 25 times, each time on a different level layout, with a budget of 8000 FMC per iteration.

The average amount of steps required to solve the environments increases by approximately 30% for both MCGS+RN and MCGS+R*N. The 20% increase comes as a result of the paths to the goal essentially being 20% longer due to the action failure chance. The cause for the other 10% is most likely due to critical actions like *Picking up the key*, *Opening the door* or *Moving through the door* being failed, and therefore not considered as potential options. This is also reflected in much higher standard deviation, as missing critical states during the expansion or rollouts can be detrimental.

VI. Conclusion and Future Work

In this study, we presented a new planning algorithm that surpasses the current state of the art in the domain of sparse environments. Taking advantage of the graph structure to eliminate node redundancy through merging nodes and disregarding obsolete actions already creates a good basis due to the reduction in the branching factor. We proposed several additions that work symbiotically to allow for planning methods to be effectively used in sparse reward environments. Combining the use of the frontier for the selection step, with storing the nodes during the rollout, and adding a novelty bonus as the intrinsic exploration incentive increases the sample efficiency of MCGS creating an effective way of searching

through large state spaces. Our results demonstrate that combining these modifications shows a significant decrease in the computational budget necessary to discover, and later on reach the goal compared to standard MCTS and current state of the art planning based methods.

Novelty bonuses based on occurrences of features of the state space in the graph provide a generalizable way of giving additional information to the agent and decreases the number of FMC until the goal is reached by a large margin. To further enhance the exploration it could be possible to take into account not only novelty through occurrences of states, but also look for the empowerment of the agent through graph analysis, and the change in state features caused by specific actions to add a surprise factor as well [17], [18].

Storing rollouts throughout the search also manifested as a key part of the algorithm. It significantly improved the performance in conjunction with the novelty bonuses. Storing each path visited during the rollouts expanded the frontier greatly and enabled the search to continue from a state further from the root node. Nonetheless, the experiments have presented a high variance during the execution of the algorithm, partially due to the non-optimal paths caused by random rollouts. Disabling certain actions during a portion of the rollouts, or having dynamically adjusted rollout lengths could reduce the variance and give more consistent results over multiple seeds [19]. Following the notion of exploration vs exploitation, using a portion of the computational budget to optimize the best-known path while still using the rest to explore the environment may also lead to an improvement in the path length of the discovered solution. Optimizing the length of the solution path with the addressed issue of the lower computational budget would further improve the effectiveness of MCGS in the environments in which it has been struggling.

A further line of research could also include testing the effects of different exploration incentives such as novelty and stored rollouts in multiplayer environments. It would be interesting to compare the MCGS and the benefits its enhancements provide with the standard planning based methods in both adversarial and cooperative settings.

VII. Acknowledgement

Marko Tot is supported by the 2020 Microsoft Research PhD Scholarship Program. This paper is also funded by EPSRC CDT in Intelligent Games and Game Intelligence (IGGI) EP/S022325/1.

References

[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[2] C. P. Robert and G. Casella, *Monte Carlo Integration*. New York, NY: Springer New York, 2004, pp. 79–122. [Online]. Available: https://doi.org/10.1007/978-1-4757-4145-2_3

[3] E. Leurent and O.-A. Maillard, "Monte-carlo graph search: the value of merging similar states," in *Proceedings of The 12th Asian Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. J. Pan and M. Sugiyama, Eds., vol. 129. PMLR, 18–20 Nov 2020, pp. 577–592. [Online]. Available: https://proceedings.mlr.press/v129/leurent20a.html

[4] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in monte carlo tree search," in *2008 IEEE Symposium On Computational Intelligence and Games*, 2008, pp. 389–395.

[5] A. Kishimoto and J. Schaeffer, "Distributed game-tree search using transposition table driven work scheduling," *Proceedings International Conference on Parallel Processing*, pp. 323–330, 2002.

[6] A. Saffidine, T. Cazenave, and J. Méhat, "Ucd: Upper confidence bound for rooted directed acyclic graphs," *Knowledge-Based Systems*, vol. 34, pp. 26–33, 2012.

[7] M. Katz, N. Lipovetzky, D. Moshkovich, and A. Tuisov, "Adapting novelty to classical planning as heuristic search," in *ICAPS*, 2017.

[8] J. Martin, S. S. Narayanan, T. Everitt, and M. Hutter, "Count-based exploration in feature space for reinforcement learning," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, p. 2471–2478.

[9] H. Baier and M. Kaisers, "Novelty and mcts," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1483–1487. [Online]. Available: https://doi.org/10.1145/3449726.3463217

[10] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[11] D. J. N. J. Soemers, C. F. Sironi, T. Schuster, and M. H. M. Winands, "Enhancements for real-time monte-carlo tree search in general video game playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[12] M. Chevalier-Boisvert, L. Willems, and S. Pal, "Minimalistic gridworld environment for openai gym," https://github.com/maximecb/gym-minigrid, 2018.

[13] Y. Flet-Berliac, J. Ferret, O. Pietquin, P. Preux, and M. Geist, "Adversarially guided actor-critic," in *ICLR*, 2021.

[14] N. R. Ke, A. Singh, A. Touati, A. Goyal, Y. Bengio, D. Parikh, and D. Batra, "Learning dynamics model in reinforcement learning by incorporating the long term future," 2019.

[15] E. Leurent and O. Maillard, "Practical open-loop optimistic planning," *CoRR*, vol. abs/1904.04700, 2019. [Online]. Available: http://arxiv.org/abs/1904.04700

[16] M. Jiang, E. Grefenstette, and T. Rocktäschel, "Prioritized level replay," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 4940–4950. [Online]. Available: https://proceedings.mlr.press/v139/jiang21b.html

[17] D. Gravina, A. Liapis, and G. Yannakakis, "Surprise search: Beyond objectives and novelty," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 677–684. [Online]. Available: https://doi.org/10.1145/2908812.2908817

[18] E. Hartuv and R. Shamir, "A clustering algorithm based on graph connectivity," *Information Processing Letters*, vol. 76, no. 4, pp. 175–181, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019000001423

[19] R. Gaina, S. Lucas, and D. Perez Liebana, "Tackling sparse rewards in real-time games with statistical forward planning methods," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1691–1698, 07 2019.