Making Something Out of Nothing: Monte Carlo Graph Search in Sparse Reward Environments

Marko Tot,[1*] **Michelangelo Conserva,**[1] **Sam Devlin,**[2] **Diego Perez Liebana**[1]

[1] Queen Mary University of London
[2] Microsoft Research

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.

Introduction

Monte Carlo Tree Search (MCTS) (Browne et al. 2012) is a family of algorithms based on Statistical Forward Planning. 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 makes them unsuitable for 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 (Robert and Casella 2004). 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) (Leurent and Maillard 2020) is a modification of the standard MCTS algorithm, in which the tree structure created by the search is changed into a more generic graph. 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 maintaining and even surpassing the performance of the currently available solutions. This is achieved by increasing the sample efficiency of the search through improving the exploration rate. 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.

Related Work

MCTS performs the search by building a tree of future possible states of the game 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

[*]Contact Author: m.tot@qmul.ac.uk

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.

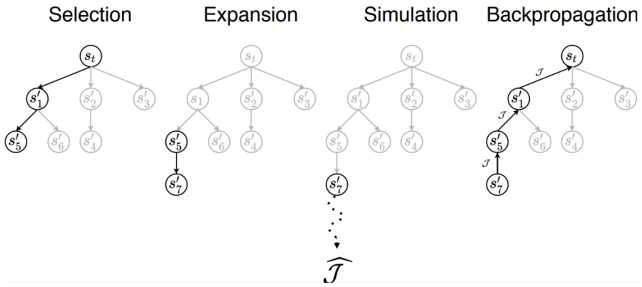


Figure 1: Monte Carlo Tree Search steps.

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 (Leurent and Maillard 2020), as multiple different trajectories may lead to the same state (Childs, Brodeur, and Kocsis 2008) and results in the same states appearing multiple times in the tree, unnecessarily increasing the size of the structure. A recent study (Nelson 2021) shows that dealing with the redundant states decreases the branching factor and consequently the size of the created structure in games like Atari by an order of magnitude. 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 (Kishimoto and Schaeffer 2002) have been used in tree search methods, as a way of combining identical nodes in order to share the value between nodes. Recent advances have been seen in Statistical Forward Planning Methods (SFP) that utilise directed graphs instead of trees to represent the game state space, where each state can appear only once in the graph (Leurent and Maillard 2020; Czech, Korus, and Kersting 2021). On one hand, this transformation causes the search space to lose part of its simplicity. 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.

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.

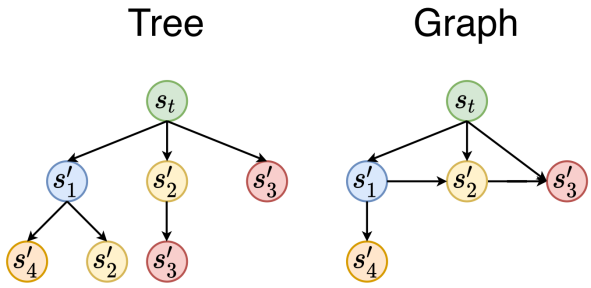


Figure 2: The graph obtained by merging identical states.

With the absence of external rewards, many techniques have been used to provide intrinsic rewards to guide the exploration. In procedural content generation (PCG), novelty has been used to drive the PCG process (Gravina et al. 2019), and to create new and different entities (Liapis, Yannakakis, and Togelius 2015). It has also been used with different reinforcement learning techniques (Jackson and Daley 2019) for improving the policy and encouraging exploration.

Recent work has also shown promise of using novelty in MCTS. 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 or reward novelty (Katz et al. 2017) separates the nodes into novel and not-novel based on the received rewards. ϕ-Exploration bonus algorithm (Martin et al. 2017) detects novel states based on the probability distribution of variables, while feature-based pseudo count novelty 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" (Baier and Kaisers 2021).

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.

Method

Our proposed algorithm is a highly modified version of the state of the art SFP algorithm called Graph Based Planner (GBP) (Leurent and Maillard 2020). GBP uses a graph structure as the basis for the search, with Upper Confidence Bound for Trees (UCT) (Kocsis and Szepesvári 2006) 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 newly explored node needs to be added to the graph.

Another available planner which uses graphs is Stochastic Graph Based Planner (S-GBP) (Leurent and Maillard 2020), which uses the GBP as its basis, but also incorporates the rollouts as well as keeping 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[1]. 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 simulated, and the value was already backpropagated through the graph.

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

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.

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

, where $\epsilon \sim \mathcal{N}(0, \sigma)$, $\sigma = \max_{n \in \mathcal{N}} \text{ucb}(n)$. 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, those nodes have to be omitted from the node selection process. 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.

[1]In some cases the game state could still change due to change not being caused by the player but by other agents or the environment itself.

Stored rollouts

In MCGS, rollouts are done in order to ascertain the value of a specific node. The agent follows a random action trajectory from the selected node and the value of the node is determined based on the accumulated reward. A valuable addition to the rollout step could be adding the nodes visited throughout the rollout to the graph. In a tree structure, those trajectories would be highly non-optimal due to the randomness of the rollouts and state redundancy. Graphs, on the other hand, allow for more inter-connectivity between nodes. Adding additional nodes to the graph during the rollout phase, combined with leaf parallelisation (Cazenave and Jouandeau 2007) provides a meaningful difference in the exploration of the state space.

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 state is regarded as novel if its occurrence rate is below a certain threshold. Including novelty as a factor adds a new component to the selection step (Equation 2).

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

Novelty inheritance is also enabled. If a node is reached from a novel node it gets a percentage 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.

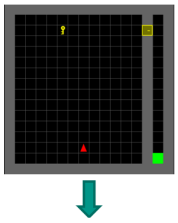
Based on these modifications we present several different ablations of the algorithm based on types of enhancements that were enabled:

- 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

Environment

In this study, we chose MiniGrid (Chevalier-Boisvert, Willems, and Pal 2018) for the evaluation of the agent. MiniGrid is a staple environment for assessing the performance of the algorithms on grid-based levels (Flet-Berliac et al. 2021; Ke et al. 2019; Leurent and Maillard 2019; Jiang, Grefenstette, and Rocktäschel 2020). The agent has been tested in two different versions of the environment, Empty and 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 size of the level is also modifiable to allow for specific difficulty ranges.

The state representation, i.e. the observation received by the agent, consists of features extracted from the environment, as well as the grid that represents the map. The choice of state representation is critical for the creation and maintenance of the graph as well as for the performance of the algorithm. Original MCTS doesn't rely on any state representation as it only encodes the action that is used to transition to the new state, and not the state itself. When using a graph representation, however, the structure of the node is of utmost importance, as different state representations will create different graph structures. An example of a game state and its state representation is presented in Figure 3.



X	Y	Rotation	Has Key	Door open	Door unlocked	Grid (16x16)
7	13	Top	False	False	False	['Wall', 'Wall', 'Wall'...]

Figure 3: DoorKey level and its state representation.

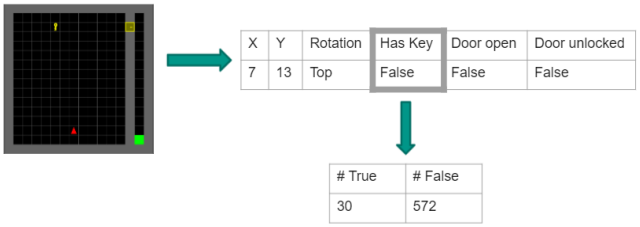
An example of using the novelty method in MiniGrid, based on the given state representation can be seen in Figure 4. This state is not novel based on the *Has Key* feature, where the value *False* is present in 572 of out 602 nodes currently in the graph[2]. On the other hand, if the value was *True*, the state would be regarded as novel due to the low occurrence rate.

An example of utilising novelty search and its effects on the reward signal in MiniGrid can be seen in Figure 5.

Experiments

Figure 6 showcases the difference between the structures created after only a couple of iterations of MCTS and MCGS, caused by disregarding 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.

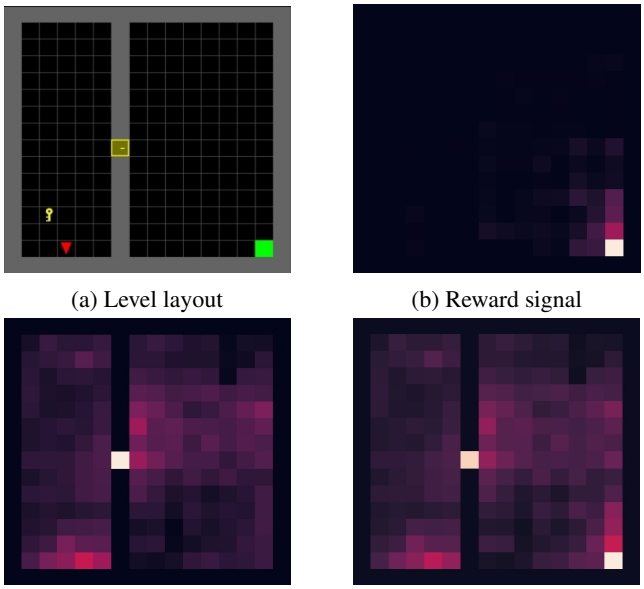
[2]It could still be novel based on the other features.



X	Y	Rotation	Has Key	Door open	Door unlocked
7	13	Top	False	False	False

# True	# False
30	572

Figure 4: Novelty detection in MiniGrid environment



(a) Level layout (b) Reward signal

(c) Novelty signal (d) Combined signal

Figure 5: Level layout and the corresponding value signals.

Empty environment

Table 1 shows the comparison between the state of the art planning algorithms and MCGS with different modifications on an Empty 8×8 environment. 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 250 forward model calls (FMC) per step.

Each algorithm was run with 50 times, each time with a different random seed, to create the variance in the selection, expansion and rollouts phases between runs. From the results, it is evident that the standard MCTS struggles to finish even the simple, Empty 8×8 grid, with only 6% of runs being successful compared to 100% completion rate of other algorithms. While this disparity of the results might come from a very low budget, it shows that in the environments where there are many identical states graph-based search can explore the space much more efficiently.

To determine the effectiveness of specific modifications made for MCGS, the algorithms were run on a larger, 16×16 grid. The results for the Empty 16×16 grid are presented

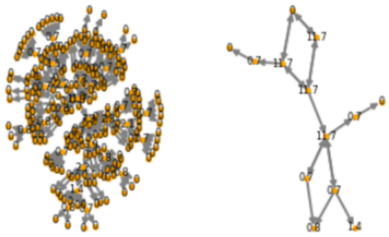


Figure 6: Difference in the tree and the graph structures caused by merging identical states and detecting obsolete actions

Table 1: Different versions of MCGS compared with state of the art planning methods, tested in Empty 8x8 environment.

Algorithm	Steps $Mean \pm STD$	Solve rate
MCTS	97.5 ± 7.2	6%
GBP	14.0 ± 0.0	100%
S-GBP	25.4 ± 1.9	100%
MCGS	39.2 ± 14.4	100%
MCGS+R	22.9 ± 7.0	100%
MCGS+N	35.4 ± 15.8	100%
MCGS+RN	33.5 ± 12.8	100%
MCGS+R*N	35.0 ± 17.0	100%

in the Table 2. Each of the algorithms was run on the same budget of 8000 forward model calls per step.

Similarly to the previous experiment, MCTS fails to solve the environment on any seed. All of the MCGS variants however manage to finish almost all of the levels. It is important to note the big disparity between the number of steps required to finish the levels after including the rollout nodes. However, with the small state space of only 782 different states, all algorithms have been able to explore almost every state and adding the rollout nodes allows the algorithm to explore much faster, in a brute-force like manner, therefore discovering and reaching the goal in fewer steps.

The results also show that adding novelty bonuses to such a small state space doesn't impact the performance of the algorithm. 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.

Table 2: 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	69.7 ± 11.8	100%
MCGS+R	35.1 ± 3.2	100%
MCGS+N	68.2 ± 19.3	96%
MCGS+RN	50 ± 20.2	96%
MCGS+R*N	43.7 ± 15.2	100%

DoorKey environment

DoorKey environment is difficult for planning algorithms 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 probability 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. Changing to this environment increased the state space from 782 to approximately $600,000$.

Each algorithm was run 25 times with the same budget of 8000 forward model calls per iteration and the results are presented in Table 3. Metrics such as average steps and the number of forward model calls were used in order to determine the effectiveness of different algorithms. 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 inherently give any reward to the agent, they have only been used for measuring purposes.

It is evident that MCGS isn't able to solve the levels, reaching the step limit on each of the level layouts. 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. Interestingly, even though they have the same rate of solving the environment of 28%, different amounts of checkpoints are discovered in these two variants. MCGS+N, discovers the goal 32% of the time, while MCGS+R was able to discover the goal in 44% of the tested levels. The reason for the significant drop for MCGS+R is that the paths added to the graph during the rollouts by MCGS+R are not optimal, requiring the agent the do more steps on its path to the goal and consequently not reaching it in time (99 total steps). Using both modifications at the same time though greatly improves the performance of the search. Adding all of the nodes found during the rollout, in conjunction with novelty bonuses reaches the over-

Table 3: Different versions of MCGS compared with state of the art planning methods, tested in DoorKey 16x16 environment.

Algorithm	Checkpoint Discovered			Solved	Mean Steps	Mean FMC	Mean Nodes	New node per FMC (%)
	Key Found	Door Opened	Goal Found					
GBP	100%	100%	96%	64%	80	639, 280	103, 403	16.17
S-GBP	100%	96%	68%	0%	99	781, 300	44, 610	0.17
MCGS	100%	40%	8%	4%	98	1, 209, 535	1, 487	0.12
MCGS+N	100%	68%	32%	28%	90	1, 218, 543	1, 510	0.12
MCGS+R	100%	84%	44%	28%	92	295, 920	17, 427	5.89
MCGS+R*N	100%	88%	72%	72%	73	819, 253	1, 365	0.17
MCGS+RN	100%	100%	100%	96%	64	231, 806	17, 753	7.66

all solve rate of 96% with only 1/25 runs not being solved in time. The benefit is also evident from the lowest average number of steps and the lowest number of FMC. Combining the two enhancements also provides a high new node per FMC of 7.66%, the highest between all of the rollout based methods. 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 of the enhancements separately.

Another important metric to look at is the number of forward model calls each of the algorithm needed to finish the level. There is a huge decrease in forward model calls after adding the rollout nodes. This happens due to an implementation choice of not doing the rollouts from the nodes which have already been added to the graph. Every time an expansion step is active, MCGS expands all of the children of the selected node, if the option to add nodes during the rollouts phase is enabled, many of these nodes will already be added and therefore no additional rollouts will be enacted.

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 harder levels which can be seen when comparing the solve rate of 72% obtained from this version to the 96% of MCGS+RN.

MCGS+RN also outperforms both of the current state of the art planning methods. Given enough computational budget to search a significant portion of the space GBP can solve some of the environments. Without the need to spend the budget on rollouts, this method can search through a large amount of space, creating over 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 number is an order of magnitude higher than in MCGS+RN (17, 753), and two orders of magnitude higher compared to MCGS+R*N (1, 365) the memory consumption is also significantly higher for GBP. On the other hand, S-GBP often discovers the goal, but cannot solve the environment in time. It also on average stores 44, 610.64 nodes, a significantly less amount of nodes compared to GBP, but also still several times more than MCGS+RN.

To further compare the state of the art with the main MCGS variants, additional metrics for each of the subgoals are presented separately in Table 4. Step metrics represent the performance of the agent in regards to optimality, nodes stored in the graph equate to the memory consumption of the algorithm and the number of forward model calls reflects the computational budget in which the algorithm can solve the environment.

MCGS+RN variant outperforms both of the state of the art planning algorithms in all three metrics. MCGS+R*N presents itself as a promising alternative solution, which allows for a trade-off between computational budget and memory consumption while still maintaining most of the optimality of MCGS+RN. Compared to MCGS+RN, it used almost 6 times more FMC, but reduced the number of stored nodes by a factor of 8. This trade-off could be useful when the state representations themselves become large enough, where the high memory consumption can become an issue.

Even though 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. As an example, for a hard level where the key is far away from the door, presented in Figure 7, after 50 runs with the different agent seeds, the average number of steps required to completely solve the environment was 68.5 with the standard deviation of 12.7, while the optimal route would take 40 steps.

There are two reasons for this sub-optimal solution. Firstly, 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 the goal. Until the agent discovers the goal, it is purposelessly moving around the environment. For the hard level presented in Figure 7, the average number of steps required to discover the goal was 21.0 with the standard deviation of 11.3. 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 somewhere

Table 4: Subgoal discovery metrics for GBP, S-GBP and MCGS+RN and MCGS+R*N.

Algorithm	Key Discovered			Open Door Discovered			Goal Discovered		
	Step	Node	FMC	Step	Node	FMC	Step	Node	FMC
GBP	8.5	115	663	22.9	6,970	38,681	38.0	103,404	442,645
S-GBP	1.0	132	840	15.8	7,436	118,790	46.1	16,931	364,422
MCGS+RN	1.3	127	2,653	12.3	2,529	25,990	25.2	5,416	56,569
MCGS+R*N	1.1	59	3,608	22.0	441	242,140	28.2	697	314,703

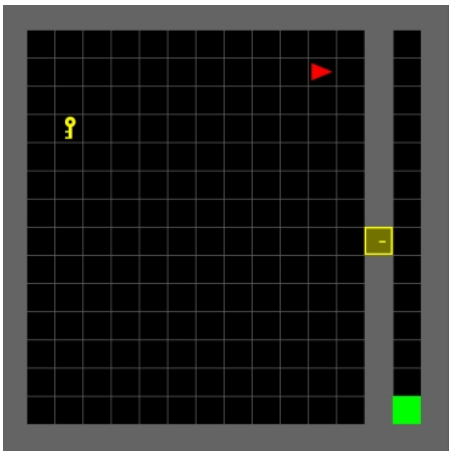


Figure 7: DoorKey 16x16 - Hard level layout

along its path to the goal. The average discovered path length was 47.6 steps with the standard deviation of 6.8 while the optimal route would take 40 steps. Combined, these two factors are the cause of the disparity between the optimal play and the obtained solution. Figure 8 shows the comparison between optimal solution and the solutions from different seeds of MCGS+RN.

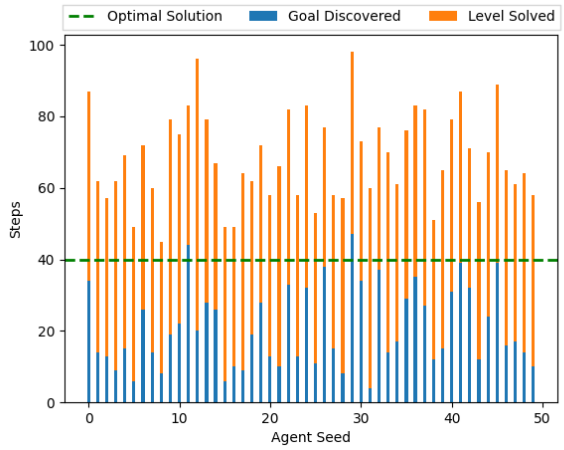


Figure 8: Impact of various agent seeds on the performance of MCGS+RN.

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 presented several main 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 the algorithm by creating an effective way of searching through large state spaces. The 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 forward model calls 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 in addition to the novelty (Gravina, Liapis, and Yannakakis 2016; Hartuv and Shamir 2000).

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 (Gaina, Lucas, and Perez Liebana 2019). 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 lower computational budget would further improve the effectiveness of MCGS in the environments in which it has been struggling.

References

Baier, H.; and Kaisers, M. 2021. Novelty and MCTS. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '21, 1483–1487. New York, NY, USA: Association for Computing Machinery. ISBN 9781450383516.

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.

Cazenave, T.; and Jouandeau, N. 2007. On the parallelization of UCT. In *Computer Games Workshop*.

Chevalier-Boisvert, M.; Willems, L.; and Pal, S. 2018. Minimalistic Gridworld Environment for OpenAI Gym. https://github.com/maximecb/gym-minigrid.

Childs, B. E.; Brodeur, J. H.; and Kocsis, L. 2008. Transpositions and move groups in Monte Carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games*, 389–395.

Czech, J.; Korus, P.; and Kersting, K. 2021. Improving AlphaZero Using Monte-Carlo Graph Search. In *ICAPS*.

Flet-Berliac, Y.; Ferret, J.; Pietquin, O.; Preux, P.; and Geist, M. 2021. Adversarially Guided Actor-Critic. *CoRR*, abs/2102.04376.

Gaina, R.; Lucas, S.; and Perez Liebana, D. 2019. Tackling Sparse Rewards in Real-Time Games with Statistical Forward Planning Methods. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33: 1691–1698.

Gravina, D.; Khalifa, A.; Liapis, A.; Togelius, J.; and Yannakakis, G. N. 2019. Procedural Content Generation through Quality Diversity. *CoRR*, abs/1907.04053.

Gravina, D.; Liapis, A.; and Yannakakis, G. 2016. Surprise Search: Beyond Objectives and Novelty. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, 677–684. New York, NY, USA: Association for Computing Machinery. ISBN 9781450342063.

Hartuv, E.; and Shamir, R. 2000. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4): 175–181.

Jackson, E. C.; and Daley, M. 2019. Novelty Search for Deep Reinforcement Learning Policy Network Weights by Action Sequence Edit Metric Distance. *CoRR*, abs/1902.03142.

Jiang, M.; Grefenstette, E.; and Rocktäschel, T. 2020. Prioritized Level Replay. *CoRR*, abs/2010.03934.

Katz, M.; Lipovetzky, N.; Moshkovich, D.; and Tuisov, A. 2017. Adapting Novelty to Classical Planning as Heuristic Search.

Ke, N. R.; Singh, A.; Touati, A.; Goyal, A.; Bengio, Y.; Parikh, D.; and Batra, D. 2019. Learning Dynamics Model in Reinforcement Learning by Incorporating the Long Term Future. arXiv:1903.01599.

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

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *Machine Learning: ECML 2006*, 282–293. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-46056-5.

Leurent, E.; and Maillard, O. 2019. Practical Open-Loop Optimistic Planning. *CoRR*, abs/1904.04700.

Leurent, E.; and Maillard, O.-A. 2020. Monte-Carlo Graph Search: the Value of Merging Similar States. In *Asian Conference on Machine Learning*, 577–592. PMLR.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2015. Constrained Novelty Search: A Study on Game Content Generation. *Evolutionary Computation*, 23: 101–129.

Martin, J.; Sasikumar, S. N.; Everitt, T.; and Hutter, M. 2017. Count-Based Exploration in Feature Space for Reinforcement Learning. *CoRR*, abs/1706.08090.

Nelson, M. J. 2021. Estimates for the Branching Factors of Atari Games. arXiv:2107.02385.

Robert, C. P.; and Casella, G. 2004. *Monte Carlo Integration*, 79–122. New York, NY: Springer New York. ISBN 978-1-4757-4145-2.