Informed Hybrid Game Tree Search

Technical Report TUD–KE–2016–01
Version 1.0, December 22th 2016

Tobias Joppen, Miriam Moneke, Nils Schröder, Christian Wirth and Johannes Fürnkranz
Knowledge Engineering Group, Technische Universität Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Abstract

In this paper, we introduce a universal game playing agent that is able to successfully play a wide variety of video games. It combines the strengths of Monte Carlo tree search with conventional heuristic search into a single hybrid search agent, which is able to select the appropriate strategy based on its observations about the game dynamics. In particular, the agent learns a knowledge base which provides the agent with information such as an approximate transition function, the type of agents and objects that participate in the game as well as their possible effects, heuristics for focusing and pruning the search, and more. This hybrid strategy proved to be successful in the 2015 General Video Game Competition, in which our agent emerged as the clear winner.

1 Introduction

Artificial intelligence has moved from its original goal of providing a general model of cognition towards the task of designing artificial agents that act rationally in a given, specific task [1]. In that, it has been successful in domains as diverse as search agents [2], game playing [3], car control [4] or video games [5]. The solutions for such problems are typically very specific to the problem at hand and cannot be easily transferred to another domain. For this reason, the last decade gave rise to competitions aiming at evaluating AI agents over a broad spectrum of tasks. Games are especially well suited for this setting, as it is possible to define a common framework while still having access to a diverse set of problems. Competitions like the General Game Playing competition (GGP) [6] introduced this idea. In GGP, the agents have to play a previously unseen game, but they do get access to the rules of the game and thus to the environment's transition model. State-of-the-art solutions to GGP exploit this by trying to automatically extract domain knowledge from the environment model [7, 8]. More recent competitions like the General Video Game AI Competition[1] (GVGAI, [9, 10]) make this harder by only allowing to observe interactions with the environment.

In this paper, we discuss an approach to universal game playing that is able to extract domain knowledge out of observations over a wide spectrum of tasks. This is embedded in a Monte Carlo tree search (MCTS) agent [11], currently considered state-of-the-art for many game playing tasks [12]. A drawback of MCTS is that it is not able to cope with trap states and other adversarial search spaces, which makes it, e.g., not useful for games like chess [13]. Therefore, we combine MCTS with a conventional search-based agent. To modulate between these two, we introduce a technique capable of detecting the best agent for a given domain, again only based on observations. This enables us to compute good solutions for both kinds of tasks. The resulting hybrid approach competed in the 2015 General Video Game AI Competition (GVGAI) and won the competition convincingly.

In Section 2, we introduce the domain, setting and general approaches, followed by a general overview of our approach in Section 3. The most important part of our agent is a knowledge base, described in Section 4, which is used for choosing interesting objects (Section 5) and for pruning of the search space (Section 6), among others. Finally, we show the GVGAI results in Section 7, followed by a brief discussion of related work (Section 8) and our conclusions (Section 9).

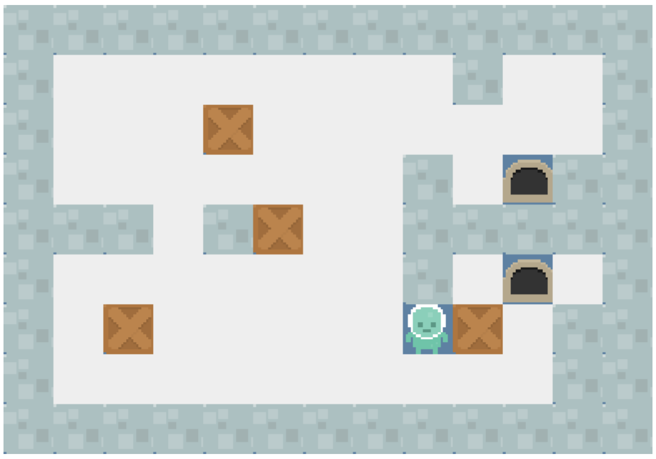
2 Preliminaries

In this section, we will introduce the GVGAI domain (Section 2.1), define a formal framework for our game AI agents (Section 2.2), and recapitulate two common approaches for action selection in games (Section 2.3). The first approach is heuristic search, usually employed for small, deterministic game trees. The second approach, Monte Carlo tree search, is the current state of the art for searching in large and stochastic game trees.

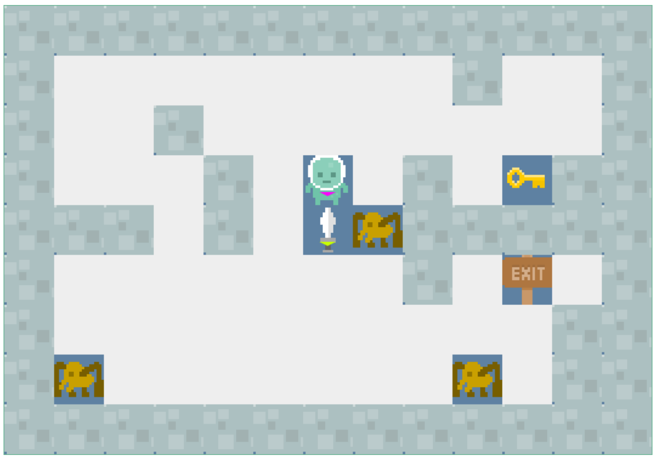
2.1 The GVGAI Domain

In the General Video Game AI Competition (GVGAI), the aim is to create a universal AI, applicable to a wide range of different games. The games are 2D single-player video games, played with a grid-like view in a top down or side view fashion. Many classic games, like *Boulder Dash*[2], *Space Invaders*[3] or *The Legend of Zelda*[4] belong to this category, and can be modeled within this framework. The games encountered within the GVGAI competition range from puzzle games like *Sokoban*[5] (Fig. 2.1a) over dynamic shooting games (e.g. *Space Invaders*, Fig. 2.1c) to role playing games (e.g. simplified *The Legend of Zelda*, Fig. 2.1b). The participants are provided with a framework for implementing an AI controller. The implemented agents do not get access to the rules of the game they are playing, but can obtain detailed information about the game state including object positions, types

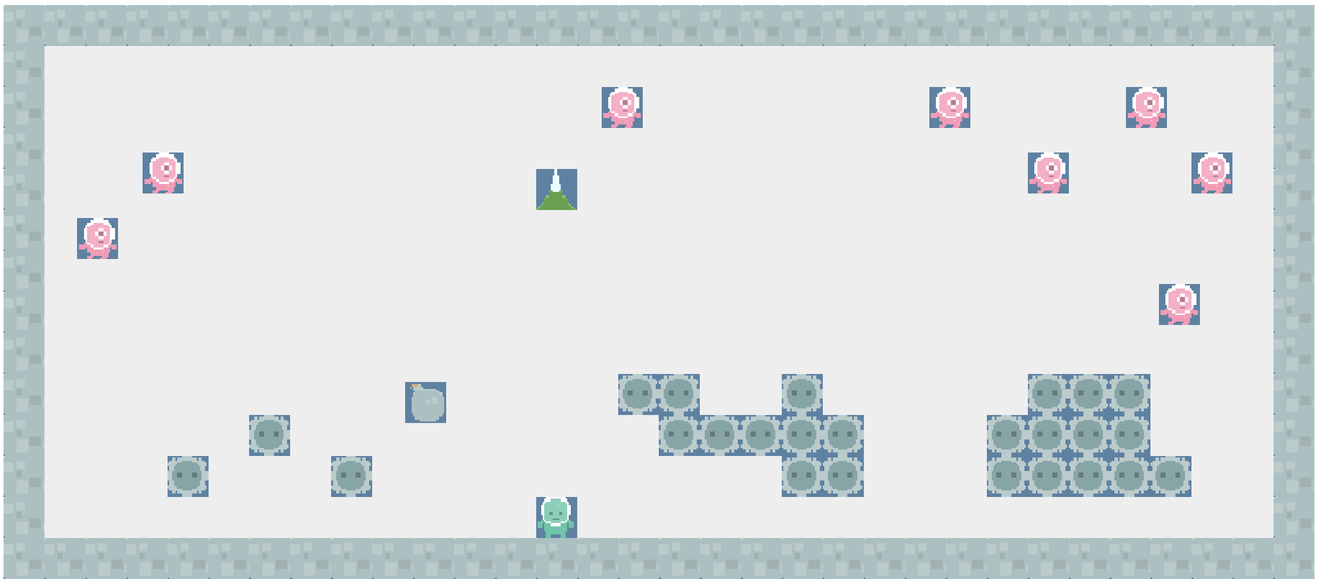
[1] http://www.gvgai.net/
[2] `https://en.wikipedia.org/wiki/Boulder\_Dash`
[3] `https://en.wikipedia.org/wiki/Space\_Invaders`
[4] `https://en.wikipedia.org/wiki/The\_Legend\_of\_Zelda`
[5] `https://en.wikipedia.org/wiki/Sokoban`



(a) Sokoban



(b) Zelda



(c) Space Invaders or Aliens

Figure 2.1: Example games of the GVGAI competition

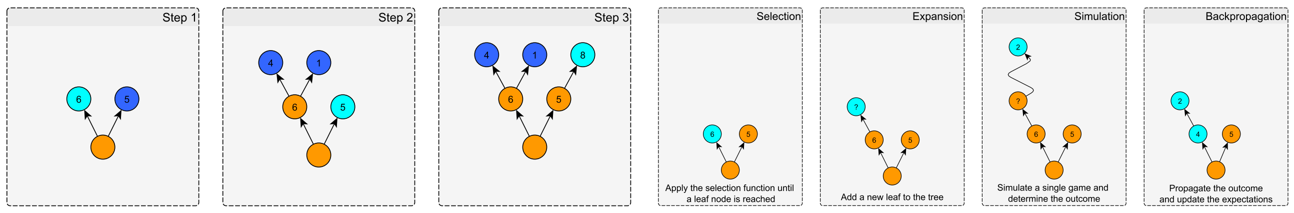
and the player's inventory. Possible objects range from non-player-characters (NPCs) over collectable items (gold, weapons, etc.) to interactive objects like doors or portals. The type information concerns the object and the player's representation (avatar). Note that in some games it is possible for the player to change its avatar. The type information is given as an abstract identifier that does not allow to determine the effect of an object or avatar collision.

The set of available actions is also known. Agents can always choose *no action*, where the player's avatar does nothing. The transition function is not known, but the effect of invoking an action can be observed by calling a (computationally expensive) simulator that returns the next state and whether the player collided with another object. The game progresses in discrete time steps, where it is required to differentiate between the simulation steps and the game steps. Players must return their next move within 40 ms, but within such a so-called *game-tick* as many simulation steps as possible can be performed. Before the start of the game, 1 sec is available for initial calculations.

The problems that arise within the GVGAI domain are manyfold. Classical planning algorithms cannot be used because the transition function is not known. Expensive computations are also not feasible, due to the real-time requirement to perform a move every 40 ms. Furthermore, the complexity of the encountered games varies greatly, which makes it necessary to use a real-time learning algorithm, because it is impossible to guarantee a sample minimum. The diversity of games also makes it difficult to use domain knowledge that could be used for guiding the search or pruning the search space.

2.2 Markov Decision Process (MDP)

A (finite-state) *Markov decision process* (MDP) is defined by a quintuple $(S, A, R, \delta, \gamma)$. Given are a set of *states S, actions A*, and a reward function $R : S \times A$. The probabilistic *state transition function* $\delta : S \times A \times S \to [0,1]$ assumes that $\sum_{s'} \delta(s,a,s') = 1$ for


Step 1
Step 2
Step 3
Selection
Apply the selection function until a leaf node is reached
Expansion
Add a new leaf to the tree
Simulation
Simulate a single game and determine the outcome
Backpropagation
Propagate the outcome and update the expectations

(a) Heuristic search

(b) Monte Carlo tree search

Figure 2.2: Game Tree Search variants

all $(s,a) \in S \times A$. $A(s)$ denotes the set of actions that are possible in state s, i.e., $A(s) = \{a \in A | \sum_{s' \in S} \delta(s,a,s') > 0\}$. $\gamma \in [0,1)$ is a discount factor, which is used to trade off the importance of expected future rewards vs. immediate gains. $S_0 \subseteq S$ defines the set of valid start states. Only states within this collection can be used to initialize the MDP. Additionally, there exists a set $S^F \subseteq S$ s.t. $\forall s \in S^F; A(s) = \emptyset$. This is the set of absorbing or terminal states where no action is possible. In most domains, these are states where the task has been accomplished or where it is impossible to reach an acceptable solution. $\pi : S \times A \rightarrow [0,1]$ denotes a policy by defining the probability of selecting action a in state s where $\sum_{a' \in A(s)} \pi(s,a') = 1$.

2.3 Single-Player Game Tree Search

Single-player game trees are a variant of MDPs. Foremost, terminal states are associated with an outcome, namely victory or defeat. In contrast to classic game trees, we may also encounter score points in some states. As example, consider the Zelda domain (Figure 2.1b) with the task to reach the exit, but where it is also possible to kill spiders for gaining points. Therefore, the reward becomes multi-dimensional (outcome and score), but in real-world scenarios, the score is usually only used to break ties if a game was won or lost (cf. Section 7). Hence, the task in this setting is to find a solution, i.e., a path to a winning terminal state, which also maximizes the score. It is usually not required to find a globally optimal policy, but only one for the current state s to determine which action a to choose next. This is arguably easier, as it is not required to generalize to unseen parts of the state space. Therefore, the tree induced by the MDP is searched for states with a high, cumulative reward and the action that leads to this state is played.

The optimal solution to the search problem can be obtained by searching the game tree exhaustively and replaying the path to the best terminal state found, but this is only possible for small, deterministic games. For large games, it is not possible to traverse the complete game tree. Additionally, when encountering stochastic transitions only an expectation can be defined that is difficult to compute reliably unless a high amount of samples can be collected.

In the following we will show two common approaches for searching deterministic and stochastic game trees.

Heuristic Search (HS)

traverses the game tree by always selecting the node that maximizes a heuristic evaluation function. This function $H(s)$ estimates the expected outcome when moving to a certain state and following an optimal policy afterwards. The algorithm (Figure 2.2a) maintains an open list of unselected nodes (blue) and creates all children for the node with the highest heuristic estimate (light blue). Once all child nodes are generated, the node is removed from the open list and not considered again (orange). In case a terminal node was found, the system saves the shortest path to that node and the outcome (+score). When the algorithm terminates, due to time constraints or an empty open list, the shortest path to the best terminal is played.

Depending on the quality of the heuristic, this can quickly lead to good paths through the state space, but optimality can, in general, not be guaranteed unless we can exhaust the search space or make some assumptions about the heuristic function.[6] Therefore, heuristic search is mainly used in small state spaces or in domains with severe time constraints. This algorithm is also not applicable to stochastic games as it does not consider that invoking the same action twice in a state may result in different child nodes.

Monte Carlo tree search (MCTS)

is usually the algorithm of choice for stochastic games. For an exact computation of the expected outcome of a policy, the agent would need to have complete knowledge of the transition function δ, but this information is often not available. It can also not be approximated reliably from a limited amount of transition samples. Moreover, HS will not be able to compute a reasonable solution if it does not encounter good terminal states within a given search time. Therefore, MCTS can be applied to stochastic or large-scale games where heuristic search meets its limits. The basic idea of MCTS is to view the action selection in a state as a multi-armed bandit problem and sample the outcome in a Monte Carlo fashion. Figure 2.2b shows the general process of MCTS. First a selection strategy is applied until a leaf node is reached. The stored tree is then expanded by adding a new node to this leaf. Its quality is estimated by performing a simulated game, a so called *rollout*, and propagating the achieved outcome as an estimate for all nodes

[6] For example, A^* search [14] can guarantee optimality if the heuristic function does not over-estimate the true costs.

among the selected path. The newly added node was only evaluated once, hence its estimate outcome is the outcome of the first rollout with value 2. The parent node was already evaluated twice and its estimate is therefore $(6+2)/2 = 4$. Due to stochastic effects, rollouts must be performed multiple times in order to compute a reliable estimate. Therefore, the selection strategy trades off exploration of new nodes with exploitation of already promising nodes for improving the estimate. Additionally, in practical scenarios, it can be too expensive to compute rollouts until a terminal state was found so that depth limits should be used.

For further details, we refer the reader to [11]. In this paper, we use the UCT selection strategy as introduced by [15].

3 Overall Design of the GVGAI Player

Depending on the domain, different search algorithms will perform best. Hence, we decided to rely on both heuristic search and MCTS. An enhanced heuristic search is used until it can be deduced that this search agent is not able to cope with the given domain. The heuristic search employs search space pruning, potentially leading to good terminal states faster.

In case this heuristic search agent is deemed insufficient, the agent switches to MCTS search for estimating the current, best action. This agent also uses an evaluation heuristic for initializing the UCT selection as well as for guiding the rollouts. Furthermore, we also prune the state space and use a MCTS variant capable of dealing with stochastic domains. Figure 3.3 shows the inner workings of the system. The left side relates to the heuristic search (Section 3.1) where as the right shows the MCTS agent (Section 3.2). The *time limit* branching box checks whether the GVGAI move time limit (Section 2.1) was hit and it is required to return the next action to play. The *optimal*, *search limit* and *stochastic* checks are specific to the heuristic search and determine if to switch to MCTS or play the current, best sequence of actions, as explained in the following heuristic search section. The remaining parts (*determine next states, score states, expand best*) relate to the common heuristic search cycle of selecting and expanding parts of the search space, whereas the selection is guided by a *score heuristic*.

The right part of the figure is a common MCTS loop of *determining next states*, *select next state* and determining if it is not yet *expanded*. In this case, it the state is added to the tree and a rollout gets performed for estimating its expected value (*expand and rollout*). Upon visiting a state for the first time, actions are selected according to a *target heuristic*, also used for biasing the rollouts.

In both cases, heuristic search and MCTS, a *knowledge base* is maintained for pruning that is updated based on the transition information of every encountered step. This database is also relevant for scoring targets with the target heuristic.

3.1 Heuristic Search

As mentioned, the system uses a heuristic search agent for deterministic games and initial search. In such cases, it searches for a solution that can be replayed exactly. Hence, this method can only be applied to deterministic environments. The basic idea is to search the game tree until an approximated, optimal solution is found, even if this requires several game ticks. As the search may take longer than the time that is allotted for choosing the next move, the agent always plays a *no action* move while it searches for a solution path.

During search (see Alg.1), a heuristic value is computed for all reachable states. This value is then used to determine which state to expand next (line 4), guiding the search more quickly into the direction of interesting states. The heuristic

$$H_{HS}(s) = -\phi_t(s) + w_s \sum_{0}^{t} r(s_t) \tag{3.1}$$

is a trade-off between minimizing the amount of time-steps $\phi_t(s)$ and maximizing the cumulative, obtained reward $\sum_0^t r(s_t)$, where w_s is a trade-off parameter that requires manual tuning. Hence, promising states with a high, cumulative reward are visited first while still biasing the search towards a breadth-first exploration style by also considering the amount of required steps. Additionally, parts of the game tree are pruned by rules provided by the knowledge base and by duplicate state identification, as will be explained in Section 6 (lines 6 and 20).

The heuristic search is performed until one of three criteria is met (line 4):

- a stochastic effect was observed
- the estimated optimal solution was found
- a game-tick threshold was reached

The first condition depends on the knowledge base for determining the existence of stochastic effects (see Section 4.2). Furthermore, if *non-player characters* (NPC) exist, the game is also deemed indeterministic because the movement of these characters is not under control of the player. As it is possible that stochastic effects arise later on in a game, a single, separate, look-ahead path is maintained. This path is extended by three *no action* moves for each game tick after an initial search phase. Hence it provides a look-ahead of at least three times the current amount of game ticks.

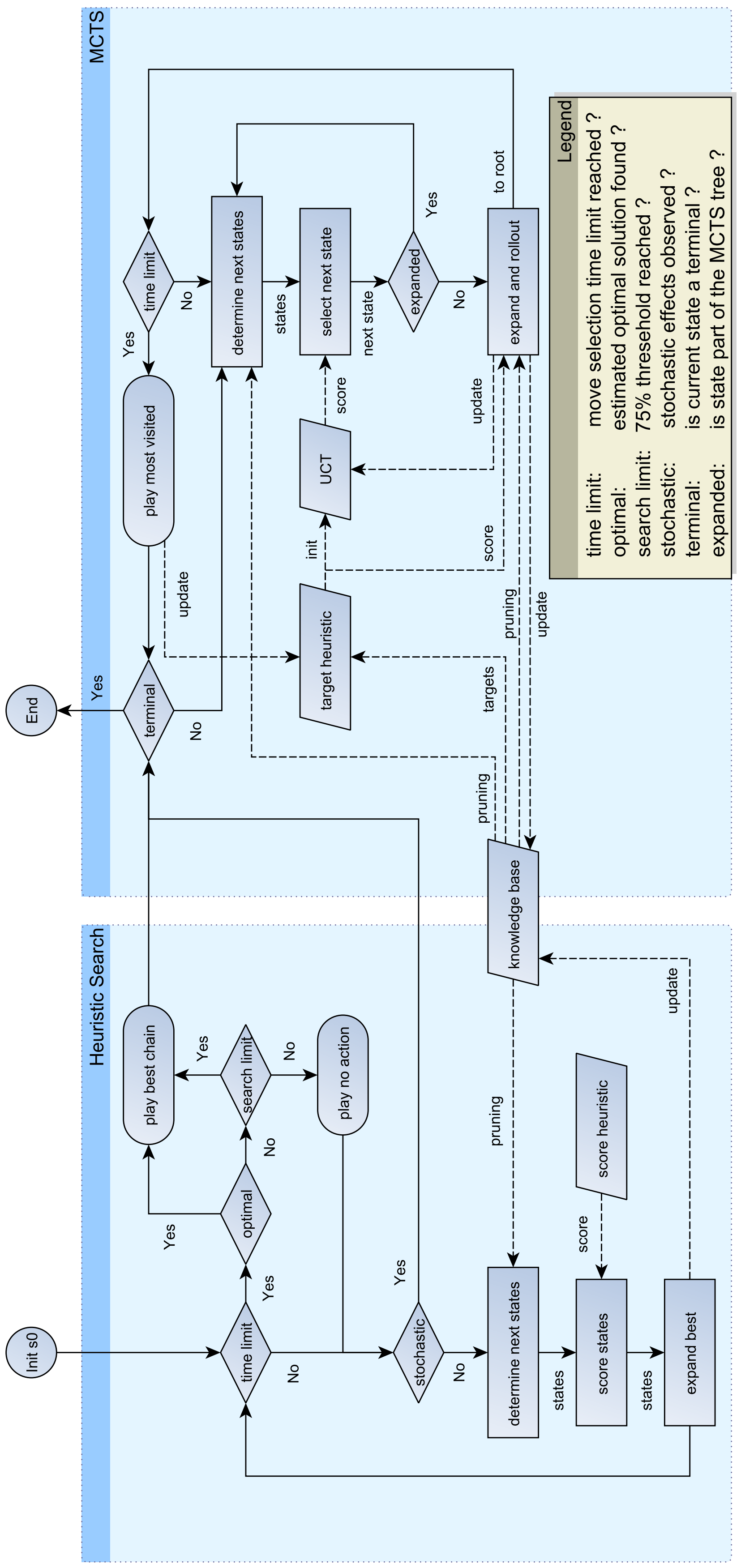


Figure 3.3: Flowchart for the main algorithm

Algorithm 1 Heuristic search algorithm with pruning

```
 1: $Open \leftarrow \{s_0\}$                                ▷ Set open-list to the current root node
 2: $Best \leftarrow \emptyset$
 3: $BestTerminal \leftarrow \emptyset$
 4: while no termination criteria met do
 5:     $s_c \leftarrow \arg\max_{s \in Open} H_{HS}(s)$                ▷ Select best state by heuristic
 6:     $A_c = \text{KB.preprune}(s_c, A(s_c))$                   ▷ Preprune actions
 7:     $Open = Open \setminus s_c$                            ▷ Remove current node from open-list
 8:     if $\text{isTerminal}(s_c)$ & $\text{outcome}(s_c) = victory$ then
 9:         if $\text{score}(s_c) \geq \mathbb{E}_{\text{score}}$ then
10:             return history($s_c$)                   ▷ Play if score is at least expected, maximal score
11:         else
12:             $BestTerminal = \arg\max_{s \in \{BestTerminal, s_c\}} \text{score}(s)$    ▷ Store best, winning terminal state
13:         end if
14:     else
15:         $Best = \arg\max_{s \in \{Best, s_c\}} \text{score}(s)$            ▷ Store best non-victory state
16:     end if
17:     $S_n \leftarrow \cup_{a \in A_c} \delta(s_c, a)$                     ▷ Create all children
18:     $S_n = \text{KB.postprune}(S_n)$                         ▷ Postprune duplicate states
19:     $Open \leftarrow S_n$                                  ▷ Add new states to open-list
20: end while
21: if $|\text{history}(BestTerminal)| < remainingTime$ then   ▷ Determine if their is still time available
22:     return history($Best$)                          ▷ Play score maximizing path
23: else
24:     return history($BestTerminal$)         ▷ Play score maximizing path to a winning terminal
25: end if
```

If no stochastic effects are present, the search continues until the optimal solution is found. As it is computationally infeasible to compute a tight bound on the value of the optimal solution, an estimate for the optimal value is used. The search agent queries the knowledge base for obtaining a list of score changing objects $\vec{O}$, as predicted by a collision classifier system (see Section 4.4). The classifier system also estimates the expected score change $\phi_{\text{score}}(O)$ when colliding with the object O. The estimate is then the sum of all objects that are increasing the score

$$\mathbb{E}_{\text{score}} = \sum_{O \in \vec{O}} \max(0, \phi_{\text{score}}(O)).$$

This is an optimistic estimate, assuming all score changing objects are reachable and it is possible to evade all score reducing objects.

In case it is not possible to find a solution achieving this estimated, optimal score, two game-tick thresholds are considered: the first threshold ensures that the current best solution is always realizable by stopping the search in case the amount of remaining game ticks is just enough for playing this solution. The look-ahead path is used to determine how many game ticks are left till the end of game and the current best solution is executed in case the amount of steps of this solutions is identical to the amount of remaining game ticks (lines 24 and 25). The second threshold is hit in case the search agent uses 75% of the available game ticks. In this case, it is assumed that the heuristic search is not suited for the given domain and the system switches to the MCTS agent. This usually happens if no score changing states are encountered or if obtaining score is not correlated with finding good, terminal states and the search space is unsuited for breath-first expansion. Before that switch is performed, the current score-maximizing path will be played, even if it is no solution (line 27). During the first second of initial search, no termination criteria are considered because this time is used for populating the knowledge base.

3.2 Enhancing MCTS

We use several MCTS enhancements, most notably informed priors and rollout policies [16], backtracking, open loop MCTS [17], early cutoffs with heuristic evaluation and pruning the search space. Algorithm 2 shows the pseudo-code of the resulting algorithm.

The informed priors are used to initialize the UCT values for unvisited state-action pairs (lines 2 and 11). This overcomes the problem of inducing a move ordering for unvisited states. As a prior, we use a heuristic which biases the action selection towards moves leading to promising positions (Section 5). These initial values are later disregarded and overwritten the first time MCTS propagates a value to this state-action pair. The same information is also used to bias the rollout policy, using an ε-greedy strategy ($\varepsilon = 0.2$), increasing the chance to perform meaningful rollouts (line 12). This results in a high chance to select the action which maximizes the heuristic's value while still exploring other actions. In many games, the heuristic target selection is quite reliable,

Algorithm 2 Enhanced MCTS

```
1:  $Tree = s_0$                                            ▷ Set tree to the current root node
2:  $\forall_{a \in A(s_0)} uctValues(s_0, a)$ = KB.informedPriors($s_0, a$)    ▷ Set informed priors
3:  while no time limit do                                  ▷ Determine if it is time to return a move
4:      $s' = s_0$
5:      while $s' \in Tree$ do                              ▷ MCTS selection phase
6:          $s = s'$
7:          $A(s)$ = KB.preprune($s, A(s)$)                 ▷ Preprune actions
8:          $s' \leftarrow \delta(s'|s$, UCT.selectAction($s, A(s)$))   ▷ Apply UCT selection
9:      end while
10:     $Tree \leftarrow \{s, s'\}$
11:     $\forall_{a \in A(s')} uctValues(s', a)$ = KB.informedPriors($s', a$)   ▷ Set informed priors
12:     $(s, a, s')$ = KB.informedRollout($s', A(s')$)      ▷ Apply rollout policy, return last state/action triple
13:     $b = 0$
14:     while isTerminal($s'$) & outcome($s'$) = $defeat$ & $b < 4$ do   ▷ Backtrack up to 4 times
15:         $A(s) = A(s) \setminus a$
16:         $s' \leftarrow \delta(s'|s$, random($A(s)$))    ▷ Try alternative action
17:         $b = b + 1$
18:     end while
19:     UCT.propagateValue(history($s'$), $H_{MCTS}(s')$)   ▷ Propagate heuristic evaluation value
20: end while
21: return $\arg\max_{a \in A(s_0)} uctValue.visits(s_0, a)$   ▷ Return the most visited move
```

therefore, the first rollout in each game tick is performed with $\varepsilon = 0$ to directly verify the computed path. In case the target chooser was not able to determine an interesting position, rollouts are performed randomly.

Another difference to vanilla MCTS is that upon encountering a losing terminal state, we backtrack one step and simulate up to four alternative actions before the information is propagated backwards (line 14–18). In case a non-losing state is found, its value is propagated. This is required to prevent penalizing actions which only lead to losing states based on a suboptimal rollout policy. We also use a maximal rollout depth to prevent spending too much time in useless areas of the state space. Therefore, it is required to be able to evaluate non terminal states for propagating meaningful feedback. The evaluation heuristic (line 19)

$$H_{MCTS}(s) = \phi_{outcome}(s) + w_s \sum_0^t r(s_t) - w_d d(o_c) - w_t \phi_t(s) \tag{3.2}$$

is a weighted (w_s, w_d, w_t) trade-off between the outcome (win/loss) $\phi_{outcome}(s)$, the obtained reward (score) $\sum_0^t r(s_t)$, the distance to the currently most promising object $d(o_c)$ and the number of steps taken $\phi_t(s)$. For the computation of $d(o_c)$, see Section 5. This function is also applied for computing return values of terminal states, but the outcome has the highest influence on the value. Furthermore, it is required to use the open-loop MCTS variant [17] because we have to deal with stochastic domains. In contrast to closed-loop MCTS, nodes in the tree are not identified by a single, underlying game state, but by the action history that has lead there. Due to the stochastic effects, it is now possible that one node represents multiple states and the complete action chain from the root node must be replayed for ever MCTS iteration. For reducing the search space quickly, we use pre-pruning (line 7, see Section 6), inline with our heuristic search agent. Post-pruning is not considered in the MCTS setting, because it can be costly but is only rarely helpful in stochastic domains. In case we need to return an action for playing (line 21), we play the most visited move as this is more stable than playing the move with the highest expectation.

4 Knowledge Base

For searching game trees efficiently, it is necessary to reduce the search space because most game trees are too large for exhaustive search. If we had access to a world model, this could be achieved by restricting the search to actions resulting in relevant states. However, within the GVGAI task, the environment model is unknown. Therefore, it is only possible to determine transitional effects by observation. To make efficient use of these observations, it is also necessary to generalize them to new states. This section describes the knowledge base that is used to learn and apply this information. We use a rule-based prediction system for deterministic effects, but statistical principles like frequency and expected move distance are also used to capture stochastic effects.

In general, the knowledge base is capable of performing the following tasks:

- provide an approximate transition function $\delta : S \times A \rightarrow S$
- prediction of object movements

- prediction of collision effects
- prediction of game specific effects
- determine interesting game states
- pruning the search space

Whenever a state transition (s, a, s') has been observed, the knowledge base tries to extract information for updating the above-mentioned prediction models. The details are explained in the following sections.

4.1 Approximation of the Transition Function

Various parts of our search modules require knowledge of the state transition that will occur when invoking an action. As it is computationally expensive to compute the complete next state in the provided game environment, our agent learns an approximate transition function that can be computed faster. It calls submodules that can predict partial state changes and applies them to the current state. Position changes are captured by a movement prediction module (Section 4.2). Possible effects of the movement (e.g. collisions) can be determined with a collision classification system (Section 4.4). These two modules and their submodules can be called independently in case only partial information is required. The complete, approximate transition function is used for pre-pruning (Section 6).

4.2 Movement prediction

The movement prediction subsystem computes an expected next position for game objects as well as for the player's avatar. For the avatar, the effect is known beforehand, but exceptions can occur. For stochastic non-avatar game objects, we compute the maximal position change per time step, the movement frequency, and objects that block their movement. Deterministic movement is ignored, as it often depends on quite complex patterns, such as objects that try to minimize the distance to another object via path planning. Therefore, the learning process is too computationally expensive, and it is preferable to detect such changes by invoking the real transition function. For the avatar, it can be possible to use game-specific shortcuts, called portals, that can be identified based on information partially provided by the GVGAI framework. The movement prediction information is then used to determine dangerous zones on the board (Section 4.3), possibly resulting in a score reduction or losing the game, as well as for pre-pruning (Section 6).

Avatar movement

The avatar movement is purely deterministic, based on the selected action (e.g. `move-right` will always increase the x-coordinate by 1). Hence, it is in general not required to learn such effects. However, two kinds of exceptions can occur:

(1) When colliding with another object, it may change the resulting position, as determined by the collision prediction system (Section 4.4). Such a change may occur when an object blocks the avatar's movement, in which case the avatar will remain on its current position, or if it collides with a portal (Section 4.2). Portals can cause stochastic state changes, but this happens rarely and they are handled deterministically.

(2) In some games, the avatar may undergo a forced movement (e.g. when it is tossed by a catapult). This is detectable by the object's avatar type. To identify such cases, we compare the expected state change with the encountered change at every state transition. In case the position changes as expected, an avatar-type-specific counter is increased by 1, and decreased by 1 otherwise. Once the counter reaches -20, the avatar is considered to not have control over the movement of this avatar type.

Object movements

For predicting the movement of objects with stochastic transitions, it is first required to identify such objects. Stochastic objects are determined by observing the effects of each action. A game has stochastic objects if executing the same action in a state twice results in different successor states, or if a *no action* move results in a state change. This is checked whenever a *no action* is invoked or an action that is already part of the MCTS tree. In both cases, it is possible to compute the objects that changed, but not all of them need to be stochastic (e.g., it is possible that the effect can be contributed to a forced move, e.g. a push). A forced change requires a second object that caused the change, hence we search for directly neighboring objects that also changed. Objects that do not have such changing objects adjacent to themselves are considered to be stochastic.

As is not possible to predict the exact next position for stochastic objects, we keep track of their statistics. In particular, we derive the *movement frequency*, the *maximal step distance* and possibly *blocking objects*:

Movement frequency.

Some game objects do not move at every time step, but only in intervals. Knowing this, an avatar may move along a path next to a deadly enemy while guaranteeing survival. Therefore, we store the time steps when an object has moved. If the intervals are constant, it is assumed that they will also stay constant for future time steps.

Maximal step distance.

Whenever a game object moves, the distance along each axis is measured and the maximal distance encountered at any time-step is stored. This information is saved per object and axis, allowing to predict the reachable area for each object separately.

Blocking objects.

Due to the stochastic movement, it is not possible to prove that a game object blocks another object, because the exact effects of the objects' movements are not known. Nevertheless, it can be proven that an object is not blocking another object when a state is encountered in which both are at the same position. Therefore, all objects are assumed to be blocking until such a same-position state is encountered.

Portals

Portals are objects that can teleport the avatar or other objects from one position to another. It is important to handle them explicitly, as they can provide substantial shortcuts. Portals have enumerated entries and exits, whereas the numbers are known. An entry always teleports the object colliding with it to an exit with a given number. If there are multiple exits with this number, one of them is chosen at random, but this occurs rarely. The mapping between entries and exists is fixed for each game, but is not known in advance.

A teleport is considered to be a game state transition where the Manhattan distance between the two avatar positions is greater than 2. Whenever this case is encountered, and at least one game object is at the target position in the prior game tick, the collision learner (Section 4.4) assumes that the target of this teleportation is the game object on the target position. This enables the collision learner to predict the next position of an player in case it collides with the teleporter. In case of object movement, portals are indirectly treated, as they are consider as spawners (see. 4.5).

For the player, the portal information gets used to determine the behaviour when invoking a move action (Sec. 4.2).

4.3 Danger Heatmap

Some games include stochastically moving enemies that kill the avatar on collision. The stochastic movement statistics are used to create a danger heatmap, determining the minimal amount of steps until such an enemy is on a certain position. The heatmap is computed by iterating over neighbouring positions for each object in a breadth-first manner (up to depth 10). The object movement statistics (Section 4.2) for each object $o \in \vec{O}$ are then used to determine the minimal amount of steps required to reach the position. The heatmap is computed as

$$\text{heat}(x,y) = \min_{o \in \vec{O}} \left(\text{stepfreq}(o) \left(\frac{|\phi_x(o) - x|}{\text{maxstep}_x(o)} + \frac{|\phi_y(o) - y|}{\text{maxstep}_y(o)} \right) \right)$$

with $\phi_{x/y}(o)$ as the objects current position per axis, $\text{maxstep}_{x/y}(o)$ the saved, maximal step distance and $\text{stepfreq}(o)$ as the movement frequency. The danger value for each position is now the minimal step distance to the next dangerous object or ∞ if it can not be reached within 10 steps. Figure 4.4 shows an example heatmap with $\text{maxstep}(o) = 1$ and $\text{stepfreq}(o) = 1$ for all spiders. This heatmap is then used by the target chooser (Section 5) to compute heuristic values for guiding the MCTS agent (Section 3.2).

4.4 Collisions

In most games, the avatar is required to interact with non-player objects to win the game (e.g., fighting enemies, picking up coins, etc.). The interaction with an object may be direct or indirect via other objects the agent can control (e.g., shooting a missile). Knowledge about the effects of a collision can be used to prune parts of the state space (Section 6) or to evaluate actions (Sections 3.1, 5). To acquire this knowledge, our system learns a *collision classifier* that can predict the outcome of an interaction.

Characterization of collisions and their effects

An interaction always requires two objects to be on neighboring squares and an action that results in a collision of these two objects. In case an action has resulted in such a collision, the environment notifies the agent that a collision has occurred, but the effects

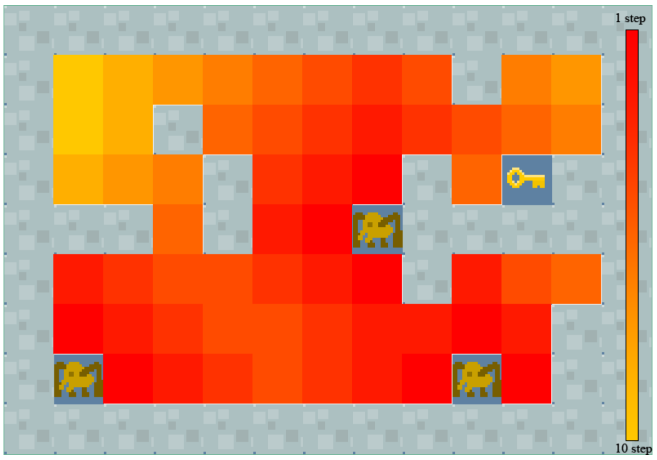


Figure 4.4: Heatmap for the simplified zelda domain.

have to be determined by computing the state difference between the state before and after the collision. As it is not reasonable to consider the complete state information for computing this difference, due to computational and generalization issues, we only consider the following, binary features for describing state differences:

- Change of the avatars type
- Change of the avatars inventory
- Score-change
- Terminal state (No, Yes[Game Won], Yes[Game Lost])
- New game objects
- Disappeared game objects

Of course, more than one of these changes can occur simultaneously.

The effects of a collision may not only depend on the avatar type and the object, but also on the avatar's inventory. In order to cope with this, we train one classifier for each pair of avatar type and collision object. Each classifier is trained on a training set that encodes the inventory of the avatar at the point of the collision as inputs, and the observed collision effect(s) as desired output. Figure 4.5a shows a hypothetical training set where five collisions between an avatar and an object have occurred. For example, the first collision occurred when the avatar had 10 objects of type I_1 and 5 of type I_2 in its inventory, and the collision hat the effect that the movement was blocked (the fourth column will be explained in the next section).

The system ignores collisions between non-player objects because the agent does not receive notifications for such collisions. Learning predictors for such cases is infeasible as it is not possible to determine if a partial state change is the outcome of an collision, a stochastic effect or an unseen game mechanic.

Learning of collision classifiers

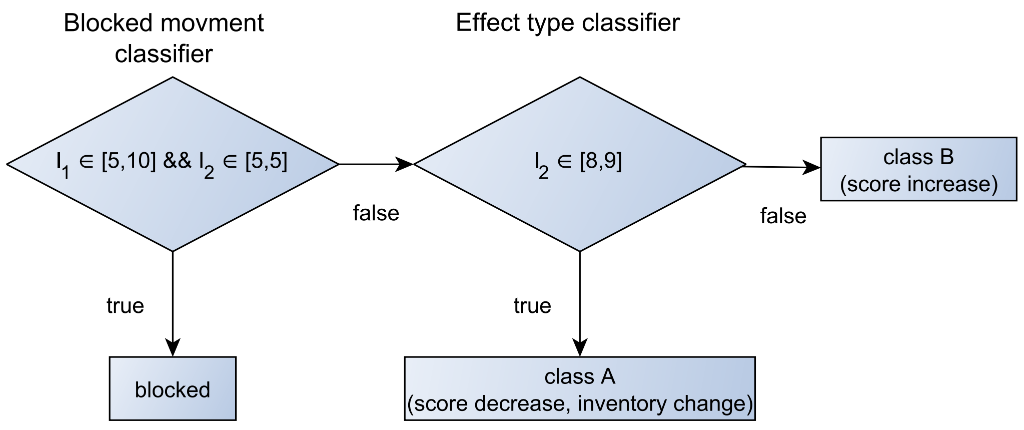
In order to be able to effectively train these collision classifiers, we made a few simplifying assumptions. First, we observed that in most cases, the possible effects of a collision can be categorized into three different groups, namely *blocked movement*, and one of two possible other outcomes *class A* and *class B*, which depend on the concrete avatar/object pair. In the example of Figure 4.5a, these are indicated in the right-most columns. Should more than three different *effect groups* occur, only the first two are used and all others ignored.

For solving the resulting three-class learning problem, we assume that the classes can be represented via hyper-rectangles. More precisely, we learn two classifiers, each in the form of a single conjunctive rule: the first single-rule classifier recognizes cases of blocked movement, whereas a second classifier distinguishes between the other two effect groups.

Blocked Movement classifier: Training examples for the *blocked movement* class are generated by checking whether the agent used a movement action that should move the avatar (Section 4.2) but unexpectedly did not result in such a movement. In case the movement was blocked, the current inventory is used as a positive example for the *blocked movement* class. All other inventories are negative examples, which are forwarded to the second classifier. This classifier either returns, that the movement was blocked or passes the instance on to the effect type classifier.

Input (amount of inventory objects)		Output (collision effect groups)	
Inventory Object I_1	Inventory Object I_2	Output Effects	Class label
10	5	blocked movement	*blocked*
5	5	blocked movement	*blocked*
3	8	score decrease, inventory change	*class A*
7	9	score decrease, inventory change	*class A*
6	10	score increase	*class B*

(a) Training instances for a single avatar/object pair


Blocked movment classifier
Effect type classifier
I1 ∈ [5,10] && I2 ∈ [5,5]
false
I2 ∈ [8,9]
false
class B (score increase)
true
true
blocked
class A (score decrease, inventory change)

(b) Classifier learned from these examples

Figure 4.5: Learning a collision classifier.

Effect Type classifier: This classifier is trained as follows: The first time a non-blocking collision for an object pair is observed, the observed group of effects are labeled as *class A*. If a collision is observed where the effects did not match the effects group of *class A*, it is labeled as *class B*.

The result is a set of two binary classifiers for each possible pair of object/avatar-types. Each classifier learns a single conjunctive rule that is the most specific generalization of the positive training examples. In case a consistent interval cannot be found, the inventory object is considered to be irrelevant.

Consider again the example in Figure 4.5 for one avatar/object pair. As we have seen, the upper part (a) shows the training instances in the order in which they arrive, and the lower part (b) shows the learned classifier pair. The first classifier separates the *blocked movement* outcome from all other effects, whereas the rule is the smallest interval only capturing the examples with the *blocked movement* outcome. The next instance is then the subject to the *score decrease* and *inventory change* effects, labeled as *class A* for the second classifier. The effect group observed at instance 5 (*score increase*) differs from the effects of *class A* and is therefore labeled as *class B*. After seeing all 5 instances, no consistent interval for object I_1 can be learned for the second classifier and the rule therefore decides to ignore I_1 and only consider an interval for object I_2.

Using the collision classifiers

To predict a collision effect, it is necessary to determine whether a collision will occur, which objects are involved and the avatar's current inventory. The inventory and the avatar-type invoking the collision are known beforehand because they are part of the current game state. For deciding whether a collision may occur and which second object would be involved, the system queries the movement prediction mechanism (Section 4.2). Should the action result in two objects in the same position, a collision is assumed. Hence, it is now possible to select the correct classifiers and query them based on the current inventory. An instance is classified as positive, i.e. as *blocked movement* for the block movement classifier, and *effect group A* for the effect type classifier, if the inventory matches all inventory intervals and negative otherwise. In case no consistent intervals have been learned, a majority vote is performed. As it is possible that multiple effect groups belong to *class B*, the majority group is returned.

In some games, objects are able to spawn other objects. In such cases, it is possible that a collision effect is indirectly invoked by colliding with a spawner that simultaneously (i.e., in the same time step) spawns a new object. The effect of the collision then depends on the newly created object, but it occurs during a collision with the spawner. Therefore, we want to map collision effects of spawning objects to their spawner.

To determine spawners, the knowledge base determines newly spawned objects at each step and analyzes their neighborhood. Is the avatar not in the direct neighborhood and nothing else moved or changed in the direct neighborhood of the new object, the object is assumed to be spawned by a spawner. In case such an behavior is observed, the type of object on which the spawned object appeared is assumed to be the spawner. If there are multiple objects on this position, one is chosen according to these rules:

- If a portal exists, it is assumed to be the spawner
- Select objects first that are not yet marked as a spawner

For each spawner, the knowledge base stores which object-type it spawns. Using the collision classifiers, it can be determined whether the spawner is deadly or not, i.e., whether the agent can lose the game by colliding with the spawner at the time it spawns an object. The agent will always avoid collisions with potentially deadly spawners in order to avoid taking unnecessary risks.

5 Target Selection

Considering that we use MCTS with a maximal rollout depth of $n \in \mathbb{N}$ and assuming a current tree depth of $t \in \mathbb{N}$, we cannot observe states that are more than $n+t$ time-steps away. Below, we call $n+t$ the *observation horizon* of the agent. In case the state information obtainable from within the observation horizon is not sufficient to determine the best next action, it is required to guide the avatar to states outside of the horizon (see Section 3.2). This effect occurs quite often, as state transitions are expensive and complex games are mostly subject to large state spaces. Therefore, we introduce a heuristic for selecting possibly interesting states outside the observation horizon, using the data stored in the knowledge base.

In each time-step, we compute a feature vector $\vec{\phi}(o)$ for each game object $o \in \vec{O}$ on the board, based on the expected effect when colliding with the avatar (Section 4.4). Additionally, the distance $d(o)$ from the avatar to the object is calculated with a best-first search, The search's state space is defined by the x, y positions on the board with the following cost function for an edge:

$$c(x,y,x',y') = 1 + \frac{80}{\mathrm{heat}(x',y')+1}, \tag{5.3}$$

where $\mathrm{heat}(x,y)$ is the value provided by the danger heatmap (Eq. (4.3)). The heuristic evaluation function for each game object is then calculated by

$$\mathbb{H}(o) = -d(o)\vec{w}^T\vec{\phi}(o), \tag{5.4}$$

where $\vec{w}$ is a user-defined weight vector which trades off the desirability and the distance of the object. The specific elements of the binary feature vector $\vec{\phi}(o)$ are the expected values of the following variable, as determined by the collision classifier:

- score in-/decrease
- winning/losing game termination
- blocks movement
- object is a portal
- object was not yet encountered
- inventory change
- use action is possible

The most interesting object is then picked by $o_c = \arg\max_{o\in\vec{O}} \mathbb{H}(o)$. The MCTS selection is updated by preinitializing the UCT selection values for unseen actions in order to increase the probability that the actions along the shortest path to this object are selected. The rollout heuristic also uses the action minimizing the shortest past, using an ε-greedy strategy with $\varepsilon = 0.2$. Therefore, it is now possible to differentiate actions even without encountering interesting states within the observation horizon. Considering that the knowledge base is not completely reliable requires us to validate the selection from time to time. In case the distance to the selected object does not decrease after multiple steps, the object is disregarded and a new target is chosen. The value of each object is only recalculated once per game-tick, but as it is possible that target objects are moving, for rollouts the action minimizing the Euclidean distance is used when the target is expected to be within 3 steps.

The presented system is only reasonable if most targets are reachable by moving over the board. In case movement is limited to only one axis (e.g. Space Invaders), this can not be assumed. Therefore, the target selection for those one-axis games is simplified by only computing the column with the highest amount of interesting objects, whereas objects closer to the avatar are weighted higher. The chosen target is then this most interesting column.

6 Pruning the Search Space

We use pre- and post-pruning strategies for efficient search space reduction. Pre-pruning uses the approximated transition function (Section 4.1) for computing the expected, next state and prunes improbable outcomes based on this information. Post-pruning invokes an action first and uses the real next state for pruning.

Pre-pruning is used to determine if an action invokes the same state change as the *no action* move. In this case, all actions equivalent to the *no action* are disregarded. In case the agent is determined to not have control over his avatar (see Section 4.2), this is not verified but directly assumed. Additionally, actions leading to a losing state are also pruned as well as actions leading outside the playing area (the size of the playing area is known). In stochastic domains, where it is not possible to reliably predetermine the effect of a movement or collision, this pruning is based on the worst case scenario. If there is a chance that the player will lose the game when invoking the action, the move will be disregarded. A special case occurs when the avatar is subject to an orientation, because moving into a specific direction now requires a move as well as a turn action. Therefore, the pruning of possibly losing states is performed with a two-action look-ahead in this case. In states where all actions are pruned, we still expand actions where it is expected to lose the game, as evaluation for the chance-based estimate. For the MCTS rollout phase, only actions leading outside the playing area are pruned, as this information is computationally very cheap to obtain.

In deterministic environments, as encountered by the heuristic search (Section 3.1), the next state can be approximated rather reliably. Therefore, when performing heuristic search, we compute a state hash code using a *Brent Hash* [18] for the expected next state and compare it with all already encountered hash codes. Duplicate states are not searched again and pruned. The hash code is based on the following state information:

- avatar-type
- avatar ID
- avatar position (x,y)
- avatar orientation (x,y)
- object positions (x,y)
- object IDs
- object types
- inventory item IDs
- inventory item counts

During heuristic search, this hash is also computed for encountered next states (post-pruning) and equivalently used to disregard duplicate states.

7 Results

In the following subsection, we present the GVGAI competition and the scoring principle it uses for ranking the participating agents. This is followed by the results achieved in this competition.

7.1 The GVGAI competition series

The 2015 GVGAI competition [10] consisted of three phases, each phase being associated with a scientific conference, namely the *Genetic and Evolutionary Computation Conference* (GECCO), the *IEEE Conference on Computational Intelligence and Games* (CIG) and the *Computer Science & Electronic Engineering Conference* (CEEC). In each phase, multiple training games were provided, where the game mechanics were known. Additionally, a validation set with unknown rules was used where it was only possible to evaluate the performance of the controller. It was also possible to compare one's results to the results of other participants for these two sets. The final ranking was computed on a previously unknown test set. The rank in each game was primarily determined by number of won games, but the total game scores achieved and the amount of required time steps were used as tie breakers. For the final, over-all ranking, a fixed amount of points per rank (from 1 to 10: 25,18,15,12,10,8,6,4,2,1) was awarded for each game, and the winner was the player with the highest sum of ranking points over all games.

7.2 Competition Results

The presented approach *YOLOBot* competed in all three phases and ended up as the overall winner.[7] The values shown in Table .1 are the cumulative scores awarded, based on the GVGAI score ranking principle. The 5 best agents overall are shown. In each leg, 10 games with 5 levels are played with 10 runs per level. *YOLOBot* was ranked first or third on all legs of the tournament.

Rank	Name	GECCO	CIG	CEEC	Total
1	YOLOBot	**141**	*76*	*89*	**306**
2	Return 42	75	**115**	86	*276*
3	psuko	*89*	*70*	**110**	*269*
4	thorbjrn	*115*	34	32	181
5	NovTea	45	61	52	158

Table .1: Final GVGAI 2015 results, best 5 agents overall

In the third leg, an agent that did not manage to be one of the 5 best agents overall, managed to reach the 2nd place. Table .2 shows the rank and points obtained for each of the test games, with rank 1 results in bold and rank 2 or 3 results in italic. A short game description is available at `http://www.gvgai.net/training_set.php?rg=5` (GECCO), `http://www.gvgai.net/training_set.php?rg=6` (CIG) and `http://www.gvgai.net/training_set.php?rg=8` (CEEC). The column *Develop* shows the results obtained with the version that competed in tourney during that time whereas the *Final* column is obtained from running the latest version that was also used at the *CEEC* competition. On the individual games, *YOLOBot* scored the highest amount of rank one results, as well as top 3 results, showing the generalization power of the approach. For a fair comparison between the final and the development version, it was required to re-run the agent on the same system as used in the competitions. Therefore, we used the online evaluation system offered by the GVGAI tourney, but this evaluation system runs each game only once, not 10 times. The final version improved the obtained results in 7 of the 20 games (the CEEC leg was already played with the last version), with substantial increases in two games (GECCO-7 and GEECO-10) and no substantial decreases. The worse results in the games GECCO-9 and CIG-6 can be explained by chance events.

8 Related work

A key component of our approach is the enhancement of MCTS with the use of domain knowledge in order to be able to use informed priors and rollouts. [16] applied this idea to the game of Go, where it is used to enrich the search with already available domain knowledge. Instead, we use automatically extracted domain knowledge, comparable to [7,8], but in our case the knowledge has been extracted without accessing the rules of the game. This is in line with the approach of [12] to GVGAI, but we do not only analyze abstract values like score, distance or occurrence changes for computing an evaluation heuristic. Our approach is capable of learning an approximate transition function, composed of different submodules that can be used for heuristic evaluation, for pruning the search space, and for suitable priors. A pruning technique for UCT was also introduced by [19], but they utilized two knowledge-free approaches as well as a Go-specific approach.

9 Conclusion

Monte Carlo tree search lacks efficiency due to the random sampling strategy used for estimating expectations. Under severe computational limits, it is also not possible to perform rollouts until a terminal state with a reliable evaluation can be reached. Therefore, it is required to enhance MCTS with techniques for reducing the search space, guiding rollouts and heuristic evaluations of intermediate states. We show that it is possible to prune the search space using an approximated transition function. For heuristic evaluation and rollout guidance, it is most important to determine interesting states that have not been visited yet. This enables us to use information for search that is not accessible with a basic MCTS strategy. Furthermore, MCTS is not a suitable search algorithm for small, deterministic games, for which heuristic search often yields better results. Therefore, we also introduce a method for determining which search strategy to use, only based on observations. By evaluating this approach within the framework of GVGAI, this algorithm proved to be superior to its competitors and able to generalize over a broad spectrum of games.

[7] The bot is also participating at the 2016 competition at `www.gvgai.net`

	Tourney Ranking					YOLOBot Wins	
Game	YOLOBot	Return 42	psuko	thorbrjn	NovTea	Develop	Final
GECCO-1 (Solarfox)	**1 (25)**	*3 (15)*	4 (12)	–	–	94%	**100%**
GECCO-2 (Defender)	**1 (25)**	–	–	*3 (15)*	–	**80%**	**80%**
GECCO-3 (Enemy Citadel)	–	*2 (18)*	10 (1)	**1 (25)**	–	**0%**	**0%**
GECCO-4 (Crossfire)	**1 (25)**	–	7 (6)	*2 (18)*	6 (8)	88%	**100%**
GECCO-5 (Lasers)	6 (8)	4 (12)	**1 (25)**	–	7 (6)	**2%**	0%
GECCO-6 (Sheriff)	–	–	–	5 (10)	–	98%	**100%**
GECCO-7 (Chopper)	–	5	**1 (25)**	4 (12)	–	32%	**100%**
GECCO-8 (Superman)	*2 (18)*	5 (10)	–	6 (8)	**1 (25)**	**100%**	**100%**
GECCO-9 (WaitForBreakfast)	*3 (15)*	–	4 (12)	**1 (25)**	–	**92%**	80%
GECCO-10 (CakyBaky)	**1 (25)**	5 (10)	6 (8)	9 (2)	7	52%	**80%**
CIG-1	–	**1 (25)**	–	–	*2 (18)*	**0%**	**0%**
CIG-2	–	–	–	–	–	**60%**	**60%**
CIG-3	*2 (18)*	–	–	6 (8)	–	**40%**	**40%**
CIG-4	–	**1 (25)**	9 (2)	–	*2 (18)*	**0%**	**0%**
CIG-5	–	–	9 (2)	–	–	**0%**	**0%**
CIG-6	–	–	*2 (18)*	–	6 (8)	**56%**	40%
CIG-7	6 (8)	–	**1 (25)**	–	7 (6)	**100%**	**100%**
CIG-8	**1 (25)**	*3 (15)*	–	4 (12)	–	**100%**	**100%**
CIG-9	*3 (15)*	**1 (25)**	6 (8)	7 (6)	5 (10)	24%	**40%**
CIG-10	5 (10)	**1 (25)**	*3 (15)*	6 (8)	10 (1)	74%	**80%**
CEEC-1 (ColourEscape)	9 (2)	–	5 (10)	–	*2 (18)*	**20%**	**20%**
CEEC-2 (LaybrinthDual)	6 (8)	4 (12)	10 (1)	5 (10)	–	**80%**	**80%**
CEEC-3 (Shipwreck)	*2 (18)*	**1 (25)**	*3 (15)*	–	10 (1)	**100%**	**100%**
CEEC-4 (Bomber)	*2 (18)*	–	*3 (15)*	5 (10)	**1 (25)**	**14%**	**14%**
CEEC-5 (Fireman)	*2 (18)*	5 (10)	*3 (15)*	4 (12)	–	**4%**	**4%**
CEEC-6 (Rivers)	–	*3 (15)*	–	–	–	**0%**	**0%**
CEEC-7 (ChainReaction)	–	–	–	–	–	**0%**	**0%**
CEEC-8 (Islands)	–	–	**1 (25)**	–	–	**2%**	**2%**
CEEC-9 (Clusters)	**1 (25)**	4 (12)	8 (4)	–	6 (8)	**60%**	**60%**
CEEC-10 (Dungeon)	–	4 (12)	**1 (25)**	–	–	**2%**	**2%**

Table .2: Detailed GVGAI 2015 results, with rank and points (in brackets)

Acknowledgement

This work was supported by the German Research Foundation (DFG). We thank the TU Darmstadt HHLR center for providing the computational resources required for this project.

Bibliography

[1] S. J. Russell and P. Norvig, *“Artificial Intelligence: A Modern Approach"*, Prentice Hall, 1995.

[2] A. N. Langville and C. D. Meyer, *“Google’s pagerank and beyond: the science of search engine rankings"*, Princeton, N.J: Princeton University Press, 2006.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D.Hassabis, “Mastering the game of Go with deep neural networks and tree search", *Nature*, **529**:484–503, 2016.

[4] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling and S. Thrun, “Towards fully autonomous driving: Systems and algorithms", In *Proceedings of the 2011 IEEE Intelligent Vehicles Symposium (IV-11)*, pp. 163–168, 2011.

[5] H. Huang, “Skynet meets the swarm: How the Berkeley Overmind won the 2010 StarCraft AI competition", *Ars Technica*, `http://arstechnica.com/gaming/2011/01/skynet-meets-the-swarm-how-the-berkeley-overmind-won-the-2010-starcraft-ai-competition/`, 2011.

[6] M. Genesereth, Y. Björnsson, “The international general game playing competition", *AI Magazine*, **34**(2):107–111, 2013.

[7] S. Haufe, D. Michulke, S. Schiffel and M. Thielscher, “Knowledge-based general game playing", *Künstliche Intelligenz*, **25**(1):25–33, 2011.

[8] A. Lancucki, *“GGP with advanced reasoning and board knowledge discovery"* Master’s Thesis, University of Wroclaw, 2014.

[9] D. Perez Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couetoux, J. Lee, C. U. Lim and T. Thompson,“The 2014 General Video Game Playing Competition", In *IEEE Transactions on Computational Intelligence and AI in Games*, **PP**(99):1–1, 2015.

[10] D. Perez Liebana, S. Samothrakis, J. Togelius, T. Schaul and S. M. Lucas, “General video game AI: competition, challenges and opportunities", In *Proceedings of the 13th AAAI Conference on Artificial Intelligence (AAAI-16)*, Phoenix, Arizona, USA, pp. 4335–4337, 2016.

[11] 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*, **4**(1):1–43, 2012.

[12] D. Perez Liebana, S. Samothrakis and S. Lucas, “Knowledge-based fast evolutionary MCTS for general video game playing", In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games (CIG-14)*, pp. 1–8, 2014.

[13] R. Ramanujan, A. Sabharwal and B. Selman, “On adversarial search spaces and sampling-based planning", In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-10)*, pp 242–245, 2010.

[14] P. E. Hart, N. J. Nilsson and B. Raphael B, “A formal basis for the heuristic determination of minimum cost paths", *IEEE Transactions on Systems Science and Cybernetics*, **4**(2):100–107, 1968.

[15] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo planning", In *Proceedings of the 17th European Conference on Machine Learning (ECML-06)*, Berlin, Germany, pp. 282–293, 2006.

[16] S. Gelly S and D. Silver D, “Combining online and offline knowledge in UCT", In *Proceedings of the 24th International Conference on Machine Learning (ICML-07)*, pp. 273–280, 2007.

[17] D. Perez Liebana, J. Dieskau, M. Hunermund, S. Mostaghim and S. Lucas, “Open loop search for general video game playing", In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO-15)*, pp. 337–344, 2015.

[18] R. P. Brent, “Reducing the retrieval time of scatter storage techniques", *Communications of the ACM*, **16**(2):105–109, 1973.

[19] J. Huang, Z. Liu, B. Lu and X. Feng, “Pruning in UCT algorithm", In *Proceedings of the 2010 International Conference on Technologies and Applications of Artificial Intelligence (TAAI-10)*, pp. 177–181, 2010.