Player2 – A General Game Player

1. Introduction

Player2 is a general game player capable of playing games presented in the Game Description Language (GDL). The application is written in the Java programming language and the basis of the program is the Java Basic Player of the Palamedes Project, which originated at Dresden University of Technology. The player uses the Prolog reasoner to evaluate logical statements. As Prolog statements have to be executed, the Eclipse Prolog library is employed.

2. Basic Concepts

2.1 Search Algorithm

The states of a game can be perceived as being organized in a tree structure. The states themselves are the nodes of the tree and the branches are moves that lead to the respective successor state on the next level of the tree. In order to play a game successfully, a player has to search the tree to find all relevant solutions. In general game playing, a solution is characterized as a path through the tree that leads to a leaf node that has a beneficial score for the own role. The responsibility of a search algorithm in the context of a general game player is to find such a solution.

There are three general basic approaches for searching a game tree. The first is to process all nodes in breadth first order. This procedure ensures that all nodes of one level are searched before those of the following level and thus that a potential simple solution is found early. However, this order requires a tremendous amount of memory to temporarily save all the nodes that have to be explored later on.

The second approach is to parse the tree in depth first order. This procedure requires significantly less memory as only the nodes of the current path have to be saved. However, the process might spend too much time on finding complex solutions so that simpler solutions are found too late or not at all due to the limited amount of processing time.

The third approach is a combination of the prior two that is called iterative deepening search. The idea is to search the tree in depth first order but only up to a certain level. Once the tree was searched with this limitation, the maximum level is increased and the search procedure is repeated. Figure 1 shows the order in which the game tree is searched to illustrate the iterative deepening search.

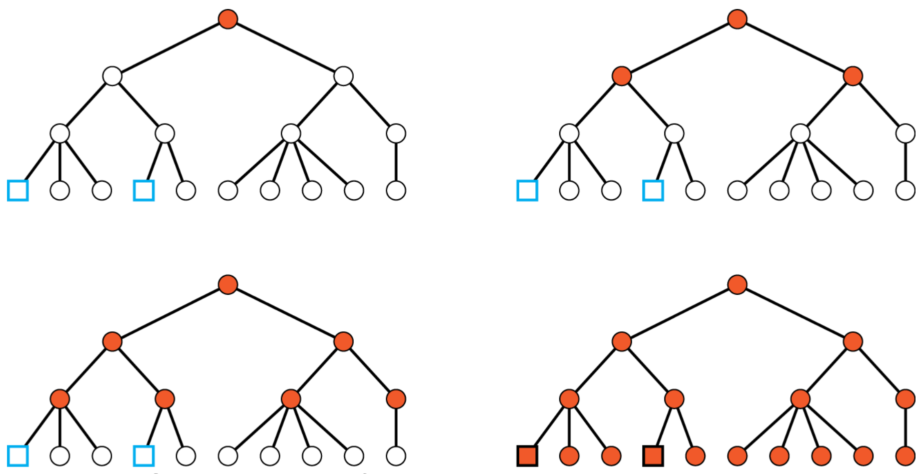
Figure 1: Search order of the iterative deepening search

With this order, potential solutions are found early on while entailing no further memory requirements. The added workload for repeatedly searching the upper nodes of a tree is relatively small so that the benefits of the procedure outweigh its drawbacks. Hence, the iterative deepening search is used as basis for the search algorithm of Player2.

2.2 Obeying to Time Limits

For a general game player to play a game there are two time limits: the start clock and the play clock. The first one is the time for the player to get to know the game and devise a strategy for it. The second one is used by the player to determine an appropriate move for the current game state. When this time limit is not obeyed, the player times out and a random move is played. To avoid this, it is essential that the player finishes its calculations on time.

Player2 uses a separate thread to enforce the time limit. The actual processing is done in the main thread of the program. Whenever a new level of the game state tree is explored (see above) , a flag is checked whether the process should stop because the time limit is about to be hit. At these points, the process could be stopped gracefully. Should none of these points be met before time runs out, the watcher thread intervenes and cancels processing. This way, it is guaranteed that Player2 responds to the messages of the game server in time, which is elementary for legal play.

2.3 State Caching

As the play clock is usually less than the start clock, it is necessary to determine the majority of the potential paths through the game state tree during the preparation for the match. In the playing phase of the game, ideally the player should merely choose one of the predetermined moves. For this purpose, it is necessary to recognize which state the game is currently in so that an adequate move can be returned. Hence, Player2 caches already recognized states. Furthermore, this temporary storing of states is useful if an already known state is re-encountered during the exploration of the game state tree. In this case, the algorithm does not have to search the state and its children again, which leads to a significant increase in processing speed.

3. Advanced Concepts

3.1 Heuristics

As the path to a terminal node in the game state tree might be relatively long, it is not always possible to determine an accurate value for the goal scores of a particular node. Therefore, it may be necessary to make an educated guess about the quality of a game state. Player2 uses a combination of two approaches to determine an estimated quality for a game state.

The first approach is the mobility heuristic. With this procedure, a game state is perceived as being better than another one when the first offers more options for potential moves than the second one. The rationale is that a greater number of potential moves might also mean a greater amount of control over the game. In many games this is particularly true for the last moves before the end.

The second procedure is the novelty heuristic. It perceives a game state as being better the greater the amount of change is in comparison to the initial state. The rationale is that in many games the final state is significantly different from the initial state so that it is beneficial to get as far away from the start state as possible.

Player2 calculates scores for both of these heuristics and averages the sum of both values to obtain a heuristic value for a particular game state.

3.2 Non-uniform Iterative Deepening Search

With the heuristic value for game states it is possible to enhance the blind search algorithm to a guided search that uses hints about which branches of the game state tree are the most promising. For this purpose, the iterative deepening search is extended to the non-uniform iterative deepening search. First, the heuristic values for all children of a particular game state are calculated. The list of states is then sorted descendingly by this value so that potentially good states are explored early on. With this reordering, it is also possible to take a closer look at the states that are most promising. In the non-uniform iterative deepening search, the level limit of the iterative deepening search is extended for the best nodes within the list to create the effective level limit. The effective level limit is then gradually reduced with every further node that is explored until the original level limit is reached. This way, additional search efforts are made for the game states that are most likely to lead to a good solution. Figure 2 shows the search order of the non-uniform iterative deepening search with the effective level limit in place.

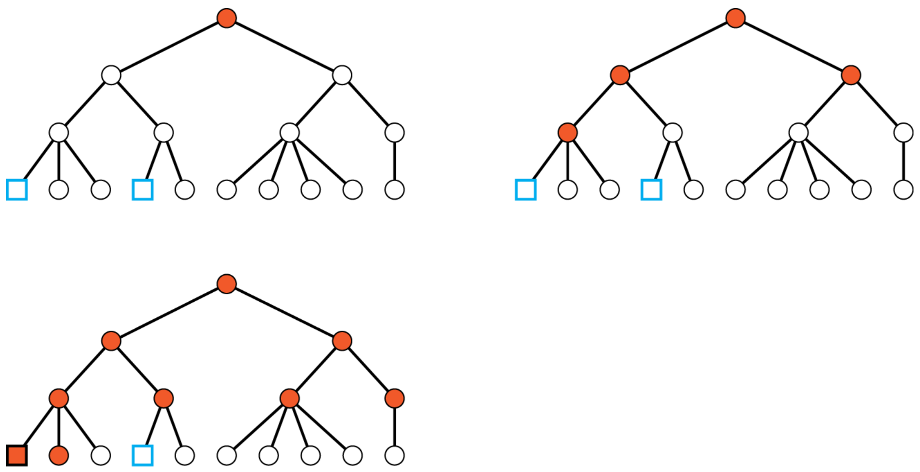
Figure 2: Search order of the non-uniform iterative deepening search.

3.3 Risk Taking

The calculation of heuristic values has additional benefits. There are cases when not all of the child nodes of a particular game state could be explored. Some of the already known successors might be terminal states with a goal score attached to them so that a definite score for some of the nodes is already known. However, it is still possible that the path through the still unknown nodes would yield a better result. If exploring these states would have been an option, the algorithm would have already done so. Thus, there has to be a reason for why these states are still unknown and usually this reason is the lack of time. Hence, the player has to take a certain risk to choose the unknown path. Figure 3 shows an example of a situation where risk taking might be necessary.

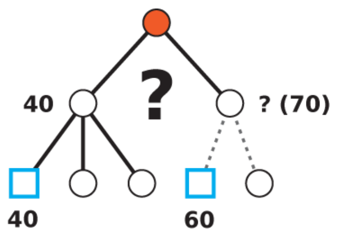

Figure 3: Example of the risk taking mechanism.

In the case of Player2, the unknown path is preferred over an already known value when the heuristic value is at least 30 points better than the definite goal value. In these cases, the player takes a chance to come to a potentially better outcome even though the path is unknown.

3.4 Graceful Degradation

As was already explained, Player2 caches game states to avoid re-encountering and re-exploring equivalent nodes of the game state tree. For this purpose, game states are saved in a hash map that allows fast access to the information associated with a particular state. As the number of potential game states is very large even for simple games, the number of states that can be saved at any given time is usually limited to a subset of all potential states. Therefore, a decision has to be made which states to discard once the program runs out of memory. Player2 uses a graceful degradation mechanism that reduces the number of cached states whenever it is necessary. In this case, the quarter containing the oldest entries is purged from the hash map before a new entry is made. With this mechanism it is possible for Player2 to proceed calculating even though the available memory might be exhausted.

4. Conclusion

Player2 is a general game player, which uses Java and Prolog. It is based on well understood techniques such as the iterative deepening search and state caching. In addition to that, it uses advanced techniques such as graceful degradation and the non-uniform iterative deepening search, which is backed by the calculation of heuristic values. Through these features, Player2 is capable of playing GDL based games both legally and successfully.