Artificial Intelligence Final Project Report
Friday April 11th 2014
Jean-Sebastien Dery (260 430 688)

1. Introduction

The project was completed in the scope of the Artificial Intelligence course given by the Computer Science department (COMP-424). The purpose of the project was to create an AI system a modified version of the four player board game Halma, which is explained on the Wikipedia page[1]. The basic concept of the game is that four players are placed on a square shaped board, each one at an extremity. Each player has two opponents and one ally, who is placed in the opposite corner. The player will need to exchange his position with his ally in less than 5000 tours, after which the game is considered as a draw. Another restriction from our implementation of the game is that any player must not be in his goal zone after 100 tours, after which he loses the game and is not allowed to play anymore in this round. Every student in the class is expected to create an intelligent agent that will be competing against the other players made by the different students in the class. With this game setup and these constraints, an intelligent player was made in the Java programming language in order to participate to the final competition. The implementation follows the Iterative Deepening A* approach in a multithreaded application where multiple moves are considered in advance, before making a final decision. Multiple approaches were taken in order to arrive at this memory and speed efficient solution. This paper is dedicated to an in depth explanation of the implementation, the alternatives that were considered, and the pros and cons of the solution.

2. Implementation

2.1. Context limitation

From the specific context of the project and the final competition against other players from the class, design choices were made in order to optimize the outcome. The artificial intelligent agent that we needed to design and implement needed to make a decision within a second, when it was his turn to play. This interesting restriction affects how the game should be approached in terms of efficiency and practicability. Especially when you consider the fact that the moves are sent through a socket interface and not via shared memory, and the potential speed limitations of Java with its garbage collector.

This requirement of having one second of computation to make a decision can be considered as “a lot of time” for certain applications. The problem with this requirement, in this particular scenario, is the branching factor in the middle of the game. In order to make a decision regarding which token should be moved where, multiple scenarios must be considered and weighted. The amount of possible moves at the beginning of the game, and toward the end, is relatively small compared to when most of the tokens of all players are in the middle of the board. At this particular moment of the game, referred to as the melee[2], the branching factor is very high. Therefore, if we take a classic approach where the tokens of the opponents and the ally are

[1] Wikipedia. Halma [ONLINE]. Available: http://en.wikipedia.org/wiki/Halma
[2] George I. Bell. (2009). “The shortest game of chinese checkers and related problems”. Electronic Journal of Combinatorial Number Theory 9 (2009) #G01.

considered, like Minimax with Alpha-Beta pruning, the depth at which the algorithm can go is limited. The one second threshold might be elapsed before the algorithm completes and can come with a good move to play, which limits the optimality of the algorithm.

2.2. First approach

The first implementation considered the issues discussed above: the design needed to be fast and be able to give a result even if the computation was not fully completed. This demanded a bit of thinking since the traditional approach is to build a state tree and then search within the tree to find the best desired state. In order to avoid the overhead of searching the tree after building it (which does not provide a solution unless the computation is not completed) a different approach was taken. Instead of only working on a single tree, a priority queue implemented as a heap was used to keep track of the best state found so far. This priority queue would allow the player to return the best desired state at any time. It would happen even though the tree containing all the states was not fully built and evaluated. This was done in order to respect the one-second requirement.

The first implementation was a greedy Best First Search algorithm that was only considering one level of possibility in order to make the decision. The best desired state would be returned, as well as the sequence of moves needed to reach this specific state. The application was single threaded, and no mechanism was implemented to block the execution after one second had elapsed. It was not required since the depth of the tree was of only one, and even with a high branching factor it was not a problem at the time. The heuristic chosen was the manhattan distance between all the token on the board, and the farthest point in the enemy goal zone. This heuristic was computed from scratch for every possible states, which was very inefficient.

This first implementation rapidly become a problem because of its inability to think further than a single move, and its inefficiency in computing the heuristic for a state. Moreover, no penalty was introduced in the calculations of the value function for moves that were in the initial goal zone. This lack of penalty resulted in tokens that would stay in this zone even after 100 turns. The inability of the algorithm to think further than a single move particularly caused problems at the end of the game. It happened when only one or two empty spots needed to be filled in the destination goal zone since the token would need to jump back in order to proceed in the game. This resulted in games that were not able to complete and the failure to beat any adversary beside the provided random player. After the results of the first implementation, the need to proceed further in the implementation of an artificial intelligent agent felt strong.

2.3. Final implementation

Multiple trials were executed before arriving to the final result, but this section will mainly focus on the final implementation that was submitted. In order to be able to fill the goal zone and make better choices, a modified version of the Iterative Deepening A* algorithm was implemented. The strategy is to consider all the possible moves for the player given a board state for one turn. Only the player's moves are considered, the opponents' and ally's moves are not. The exploration must go further than a single action and consider multiple hops, since they make a big impact on

the performance of the player. Therefore, all set of moves and combination of hops are considered until the player must return a move with a null “to” and null “from” in order to terminate the round (if there was hops). As these desired states are explored, their value (with the aid of the heuristic) is being updated instead of being calculated from scratch (like in the first implementation). A lot of computation time is being saved by adding this small change in the code base. After the value of the state has been computed, it is added to a heap where the head represents the best desired state found so far. This allows the agent to make a decision even if the set of all possible moves has not been completely explored. This, again, is to ensure a decision can be made when the one second timeout occurs.

After the first level of possibilities has been explored, the best four states are taken from the heap in order to be explored further for five more turns. The decision of only taking the four best states came from the fact that the player has limited time to make a decision, and it is unlikely (not impossible) that the worst states will result in better moves for the next two rounds. Moreover, since the algorithm is only considering the player’s move, the state of the board will be way different than the actual predictions after six moves. There is therefore no reason of exploring further if we are not considering the other players. After all the possible moves were explored for two more turns for the player, the best desired state is chosen.

As the states are being explored and added to the heap, the list of moves needed to reach this desired state is built and saved in the state at the same time. Therefore, whenever the final decision is made, there is no need to search the tree in order to figure out how to reach this state, since the list is saved within its data structure. This feature is a trade off between computational time and memory usage, since all states will have their own list. Since we were allowed 500 MB of RAM for the execution of the program, it was determined that memory was not a problem for this particular situation. The data structure used for the board states is shown in Figure 2.3.1 (on next page) as the BoardState class.

Another improvement that was introduced in the final solution was a penalty for tokens that are in the goal zones and a penalty for moves that go out of the goal zone. This penalty, introduced in the computation of the state’s value, will add priority to these moves since all tokens must be out of the initial goal zone before the 100 turns. The heuristic for a token will also be equal to zero whenever it is in the target goal zone, but the value of a state is still the summation of the heuristic value of all tokens for the player.

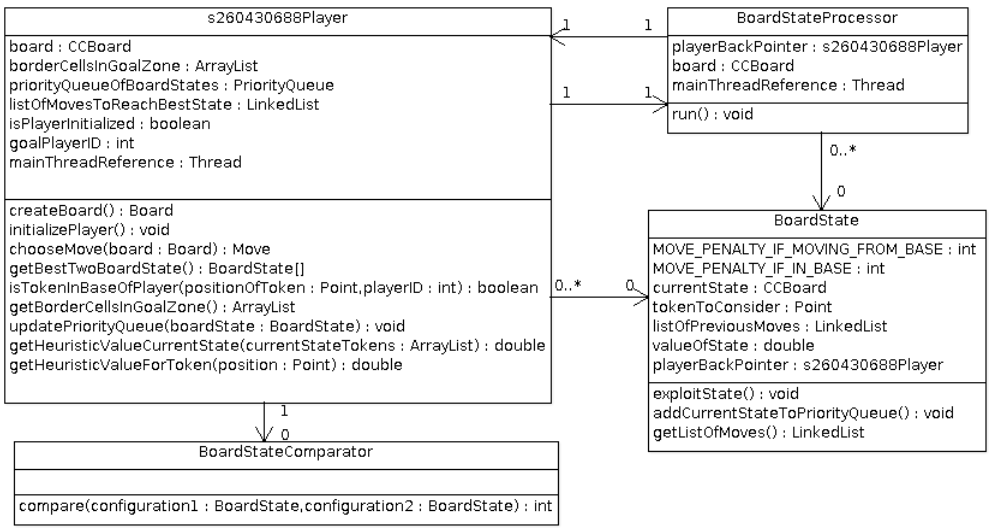

Figure 2.3.1. The UML Class Diagram of the implemented artificial intelligent agent.

Finally, in order to fit the one second requirement, the main thread creates a second thread (BoardStateProcessor in Figure 2.3.1.) that will explore the different possibilities and build the priority queue. While this is being accomplished, the main thread sleeps and waits for 850 milliseconds or a signal from the BoardStateProcessor before waking up and return the best move. Since I did not wanted all moves to take 850 milliseconds even if the computation was completed earlier, the BoardStateProcessor thread wakes up the main thread whenever it has finished.

3. Theoretical basis

This is also discussed in section 2, but the implementation uses a modified version of Iterative Deepening A*. Only the first level is fully searched, then only the four best states are searched further. The iteration is bounded by a depth of six, and the heuristic is admissible since the cost of a move is constant and the heuristic is the manhattan distance between the token and the farthest point in the goal zone. Then, the heuristic is equal to zero whenever the token is in the goal zone. The heuristic is not consistent though, since the current heuristic will not be smaller or equal to the heuristic after a move plus its displacement cost.

4. Advantages and disadvantage

4.1. Advantages of the implementation

The main advantage of the solution is that it is an iterative process which gives good result, does not take a lot of memory and computational time. It is also not a greedy implementation since it explores six turns in advance before making a final decision, trying to optimize for the future instead of the current round. It is also a fast algorithm since after exploring all the possibilities for the current round, it takes the best four states and explore them further. This avoid exploring again all the states since it is more likely that the best solution resides in the top states. This will allow the execution to complete rapidly, even in the middle of the game when the branching

factor is humongous. With this speed factor, the application is likely to end before the one second threshold has elapsed, but a safety is still implemented. After 900 milliseconds, the best solution found so far will be returned. This will happen even if the computation is not completed because the heap is being built at the same time as the tree of possibilities is explored.

4.2. Disadvantages of the implementation

The main disadvantage of the solution is that only the moves of the player are considered. The ally and the opponents are assumed to stay the same when going into further turns. It is an unrealistic assumptions since it is not allowed to not make a move during a turn. It can also compromise the optimality of the artificial agent since it is not thinking about potential good positions in which the other players can place it. This strategy will limit the depth at which it is still valid to go since the deeper it goes, the less reliable the board state will be. Finally, the tokens are not forced to move in pack, which sometimes leave single token in the back. These token will take longer turn to reach the target goal zone since they are less likely to take advantage of multiple hops.

5. Tried approaches

This subject is discussed in section 2, but a few things were tried between the first approach and the final implementation. The A* algorithm was tested, but was failing to give a result before the one second threshold due to the high branching factor in the middle of the game. This is the main reason why an iterative deepening approach has been chosen. An implementation that was considering the ally moves in the exploration of the further turns was also tried, but proved to be not optimal against other players with different agent. An approach where all moves would need to wait 850 milliseconds was also tried, but it ended up being a waste of time to wait when the computation was completed.

6. Further improvements

The current implementation could be improved in multiple ways. One of the first thing that should be implemented is the exploration of the state tree in parallel with the aid of multiple thread. A thread pool should be implemented in order to bound the number of threads running, and it should speedup the computation of the best state. This improvement could then allow the agent to consider the opponents' and ally's tokens and would allow it to explore more turns. Then, a Minimax algorithm with Alpha-Beta pruning could be efficiently implemented and would provide good results.