## 

To define the Successor function, we first define the Current location of the agent 1, and agent 2, as following:

Since the state variable doesn’t contain exact information on the agent 1 and agent 2 locations, we have to loop through and find a tile with values 1 and 2.

We have to check that the following conditions hold:

* Player can move to a tile which isn’t gray, or other player
* Player can move to a tile which isn’t a wall

Defining a group of Moves which can be done in a game:

For example, a move of first player upwards is: , where:

Defining a condition which will be checked:

Now, we can define the successor function:

## 

Win(s,i) ⬄ Succ(s,i ) ≠ ∅ ∧ ∀ s’ ∈ Succ(s,i): Succ(s’,(i+1)mod 2) = ∅

Tie(s) ⬄ Succ(s,1 ) = ∅ ∧ Succ(s,2 ) = ∅

## 

The branching factor can be 4 in the initial turn of a any player, if surrounded by unvisited or white tiles. From second turn on, it will be 0-3, since we can never visit the tile which we have already visited.

## 

Simple player iterated on all the directions (4 directions – UP, DOWN, LEFT, RIGHT), checking if they are feasible. If a direction is feasible, the ‘state\_score’ is being calculated. The move with the best score is performed. If state\_scores are equal, FIRST direction which was calculated wins.

According to the ‘state\_score’, it will prefer the state with the LEAST number of further states available for that player, as long as this will not lead to 0.

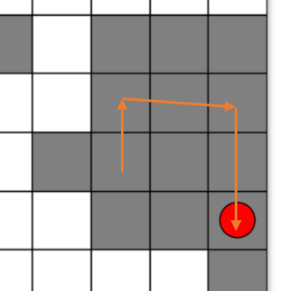
PROS:

* Given a certain area with no obstacles (a large space), the agent will for sure fill all the tiles

CONS:

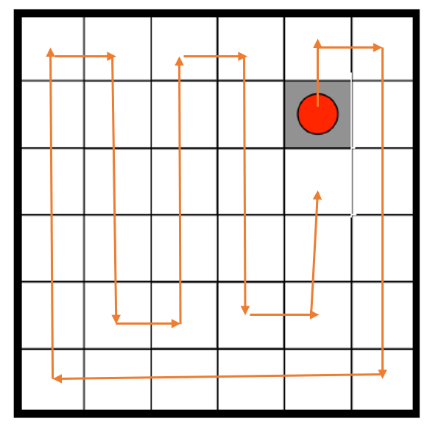
* Sees only 1 step into the future
* Doesn’t take enemy location into account
* Can run into the dead end, while being able to easily avoid it (connects to the first CON)

Example of a dead-end (map number 3)



## 

As stated in PRO, the agent will fill the whole given area in an optimal way, for example here, I have marked it’s path:



## 

The heuristic value includes only the number of successors of a certain state for a certain player. CONs can be similar to the “simple player” tactics, and more:

* This heuristic is not admissible – Sometimes going into a child state which has only 1 successor can lead to victory, while still having other children with more successors, which will result in higher heuristic
* It does not take into account the enemy location
* It looks only 1 step further.

## 

First let’s define 3 components:

1)

AvailableTiles(S) = white\_tiles/total\_tiles. This component offers us an insight on how advanced the game is, the bigger this component is, means we are at a later stage in the game.

2)

GroundDiff(S) = Player\_ground – Opponent\_ground. Player’s ground is defined by the sum of the locations that are available, and closer to the player than to the opponent( in terms of number of moves). The same goes for the opponent’s ground.

3)

Opponent\_dist(S) = the number of moves it will take for the player to reach the opponent.

Given these components we can define a heuristic as follows:

H(S) = AvailableTiles(S)\*GroundDiff(S) - (1-AvailableTiles(S))\*(Opponent\_dist(S)

This approach will be consisting of two stages: In the beginning of the game, meaning when there are more white tiles, the player will try to collect a big territorial advantage on the opponent trying to maximize his ground and minimize the opponent’s ground.

As the game proceeds and the white tiles go and diminish, the player will take an increasingly aggressive approach by trying to get closer to the opponent in order to close him.

These specific components were chosen as they give as close as a full picture of the state of the game as possible, providing crucial information in order to make the next move.

## 

The “anytime” variation of the Minimax algorithm consist in running the usual minimax algorithm with an additional time parameter, and demanding that the algorithm returns the best answer available in that time.

For this purpose we run the rb-minimax algorithm with increasing depths, until the available time runs out.

At the end of each iteration the chosen step is saved and at the end of the allocated time we interrupt the search and return the last chosen step.

This technique is called iterative deepening, and the problem presented in the lecture regarding this technique, is that on average, the time will run out during the last iteration calculation, thus we will return the previous iteration step, but still most of the resources will be used by the last iteration, so we will use a lot of resources, for no gain since we will not manage to terminate the last iteration.

## 

The proposed solution for this problem in the lecture is that in each iteration, we can store the minimax value for each of the sons of the upper level

On the last iteration, thus, half of the sons will represent a search at the deepest level, whereas the rest will represent the search at the previous depth level ( one before the deepest). This will solve the problem partially, since at least for half of the sons we will have used our resources in order to see into the next depth level.

## 

Defining the function:

Looking at the Minimax basic agent (or Alpha-Beta with NO PRUNING (worst case possible, even with child sorting)), the next depth will develop leaves, where is a branching factor.

Define:

We know, that in BFS,

Thus,

Defining number of children developed at current depth as , we get

Assuming that the time to develop each leave is the same among every iteration, we get:

We take the maximum branching factor possible (after the first move), which is .

Thus, we get:

This is of course the worst case, where no pruning occurs, and that EVERY leaf has 3 successors.

11)