Heurestic Analysis

Game of Isolation

Firstly, I have to say that I spend a lot of time trying to find good heuristics for the game of Isolation, but many problems prevent me from finding a really good one. Those problems are :

- Computer not powerful enough. Every simulation takes 20 minutes to execute
- Even for a 20 minutes execution the accuracy variance is to high between 2 distinct simulations
- If I set `NUM_MATCHES = 1` to speed up the computation, I won't have enough training samples to generalize well the accuracy I get
- The results depends on the hardware.

Yet I present to you my analysis

Firstly, here are the options I tried so far :

- **Heuristic that takes into account the distance to center**. The main idea is that the closer to the center we are, the more possibilities of action we have (8 for a Knight)
- **Heuristic that try to eliminate the opponent as quickly as possible**. The action take by the heuristic is then to attack the opponent by reducing his possible future moves
- **Heuristic that switches to another one** once there is « not so many » empty squares on the board.
- Method to improve Improved_ID heuristic by **forcing the choice of a node once there is a tie**
- **Heuristic based on the Knight's Tour Problem**. I tried this one but it doesn't fit the problem as there is an opponent that can cut my path in this game.
- if I'm next to the opponent on the diagonal, I can add point because, the opponent can't block my next move and I might block the next move of my opponent
- Lot's of others that I tried but that I won't discuss because he doesn't led to any good result in practice.

Heuristic Analysis

Closer to the center

This heuristic takes into account the fact that the closer to the center we are, the more possibilities we have to move. Actually if we are on the corner of the game we have only 2 moves at most, when we are at the center of the screen we'll have 8 moves available. Hence we can see the board (game) as a Gaussian where the peak is at the center.
This representation doesn't take into account that the game is filled over time and this is one of the drawback of this heuristic. Yet, on average, this can be a good approximation of the tendency of our heuristic.

2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2

The idea is to use a positive weight to favor the decision when there is a tie. See code below

```
my_loc = game.get_player_location(player)
dist_to_center = sqrt((game.width/2)**2+(game.height/2)**2)
cur_pos_to_center = sqrt( (game.width/2 - my_loc[0] )**2 +
(game.height/2 - my_loc[1])**2 )
norm_dist_to_center = dist_to_center/(cur_pos_to_center +
dist_to_center)

return float(my_moves*(my_moves - opp_moves)/(my_moves + opp_moves)
- opp_moves*(my_moves - opp_moves)/(my_moves + opp_moves))
```

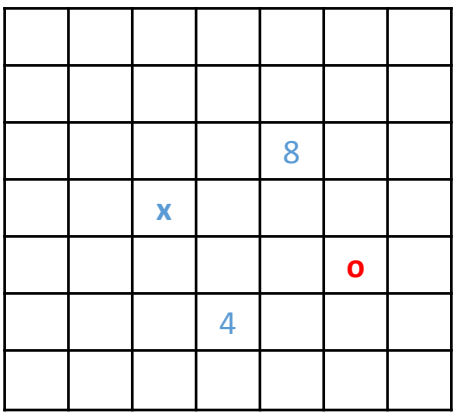
Results :

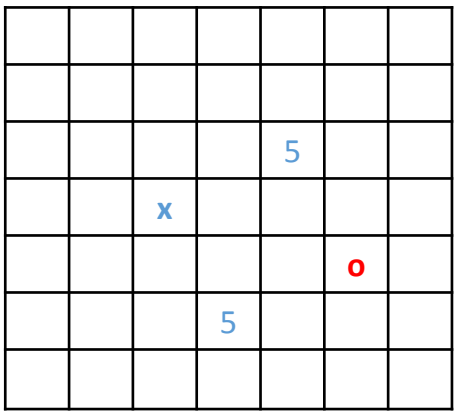
ID_Improved	**70.71%**	Student	**61.43%**
Match 1: ID_Improved vs Random Result: 20 to 0		Match 1: Student vs Random Result: 13 to 7	
Match 2: ID_Improved vs MM_Null Result: 14 to 6		Match 2: Student vs MM_Null Result: 16 to 4	
Match 3: ID_Improved vs MM_Open Result: 12 to 8		Match 3: Student vs MM_Open Result: 10 to 10	
Match 4: ID_Improved vs MM_Improved Result: 13 to 7		Match 4: Student vs MM_Improved Result: 9 to 11	
Match 5: ID_Improved vs AB_Null Result: 16 to 4		Match 5: Student vs AB_Null Result: 14 to 6	
Match 6: ID_Improved vs AB_Open Result: 12 to 8		Match 6: Student vs AB_Open Result: 14 to 6	
Match 7: ID_Improved vs AB_Improved Result: 12 to 8		Match 7: Student vs AB_Improved Result: 10 to 10	

Heurestic Analysis

Eliminate the opponent

Another way of thinking is to try to eliminate the opponent as quickly as possible. The idea is that by doing so we will make pressure on the opponent. So the idea is to maximize the reverse of the possibilities the opponent have to choose. Hence we will return the value : `1/(#opp_moves+1)`. The **one** is added to prevent **division by zero exception**.





I mark in blue 2 different positions for the **x player**. The number corresponds to the **#my_moves** available if I put **X** in this case. So If I consider the heuristic `#my_moves - #opp_moves`, I got either:

- `#my_moves - #opp_moves = 8 - 4 =` **4**
- `#my_moves - #opp_moves = 4 - 5 =` **-1**

So `#my_moves - #opp_moves` will choose to go to case with the 8 in it.

While with `1/(#opp_moves+1)`, the number in blue represents the `#opp_moves` if **X** go in the case with the **blue number**. Hence in this case the weight are the same so the reverse : 1/(5+1) are the same and **X can either decide to go up ot to go down.**

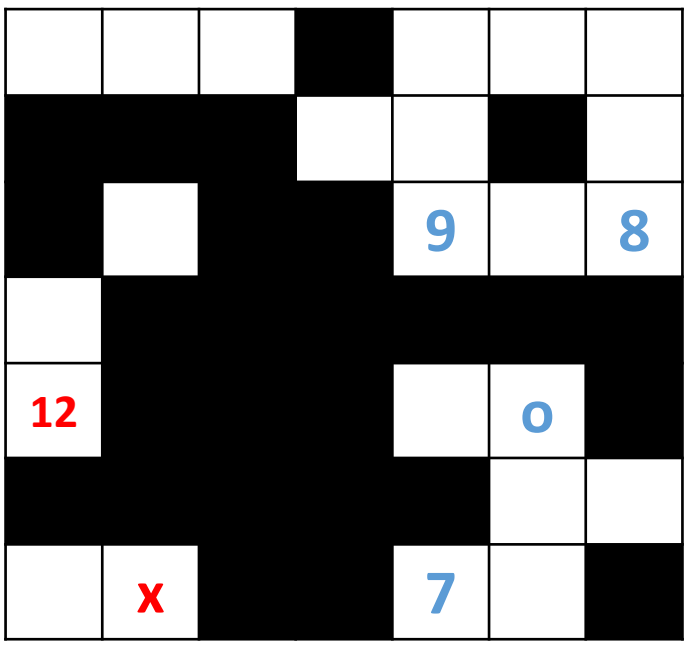
Result:

ID_Improved	**68.57%**	Student	**68.57%**
Match 1: ID_Improved vs Random Result: 17 to 3		Match 1: Student vs Random Result: 17 to 3	
Match 2: ID_Improved vs MM_Null Result: 13 to 7		Match 2: Student vs MM_Null Result: 13 to 7	
Match 3: ID_Improved vs MM_Open Result: 13 to 7		Match 3: Student vs MM_Open Result: 12 to 8	
Match 4: ID_Improved vs MM_Improved Result: 13 to 7		Match 4: Student vs MM_Improved Result: 13 to 7	
Match 5: ID_Improved vs AB_Null Result: 14 to 6		Match 5: Student vs AB_Null Result: 15 to 5	
Match 6: ID_Improved vs AB_Open Result: 13 to 7		Match 6: Student vs AB_Open Result: 12 to 8	
Match 7: ID_Improved vs AB_Improved Result: 13 to 7		Match 7: Student vs AB_Improved Result: 14 to 6	

Heurestic Analysis

Filled squares

Another way of thinking is to say that at some point (when there is lot's of filled square in the board) we might choose to to switch to an heuristic that have `#my_moves` rather than #my_moves - #opp_moves (that prefers to conserve itself) or to an heuristic that computes the longest path possible (assuming the opponent doesn't move).



Here is the representation of the situation in a 7x7 grid. The idea is that at some point it is better to maximize one's chance of survive. So, the idea is to find the longest path available at some point or to switch to a `#my_moves` heuristic

Here we evaluate the longest path accessible from the first case accessible from each player. So the blue player can accesses 3 cases from his current position. I put a blue number for the longest path accessible from each next move. Hence we can see that if player blue doesn't move, player red can play 12 times before dying.
We can also switch to the `#my_moves` heuristic after a certain number of filled square in the game.

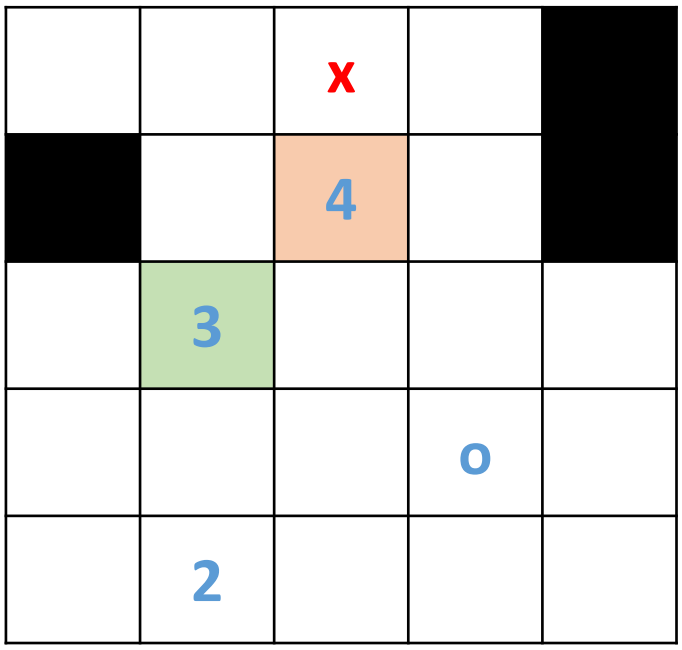
Result:

ID_Improved	**70.71%**	Student	**70.00%**
Match 1: ID_Improved vs Random Result: 17 to 3		Match 1: Student vs Random Result: 16 to 4	
Match 2: ID_Improved vs MM_Null Result: 13 to 7		Match 2: Student vs MM_Null Result: 14 to 6	
Match 3: ID_Improved vs MM_Open Result: 11 to 9		Match 3: Student vs MM_Open Result: 12 to 8	
Match 4: ID_Improved vs MM_Improved Result: 15 to 5		Match 4: Student vs MM_Improved Result: 14 to 6	
Match 5: ID_Improved vs AB_Null Result: 16 to 4		Match 5: Student vs AB_Null Result: 17 to 3	
Match 6: ID_Improved vs AB_Open Result: 14 to 6		Match 6: Student vs AB_Open Result: 13 to 7	
Match 7: ID_Improved vs AB_Improved Result: 13 to 7		Match 7: Student vs AB_Improved Result: 12 to 8	

Heuristic Analysis

Improve ID_improved when there is tie

To enhance an heuristic, one way is to think about how we can improve it. One common idea is that the heuristic can put the same value into two different box (we have a tie). So we can add a weight to favor one path over another. Hence, to improve ID_improved heuristic we going to focus on how to favor one choice over another when there is a tie.



Here is the representation of the situation in a 5x5 grid. The number in blue corresponds to the number of moves of the blue player if he goes to the case with the blue number.

So if the blue **O PLAYER** move to :

4 -> (#my_moves – opp_moves) = 4 – 2 = 2

3 -> (#my_moves – opp_moves) = 3 – 1 = 2

2 -> (#my_moves – opp_moves) = 2 – 2 = 0

So here we **have a tie**. So which one do we want to favor over the other ? If I want to favor the green one (the one with the less number of moves for the opponent I can for example divide by `(#my_moves + #opp_moves)`. If I want to favor the green one I can come up with `(#my_moves - #opp_moves)*#my_moves` for example.

Result:

ID_Improved	**71.43%**	Student	**75.71%**
Match 1: ID_Improved vs Random Result: 16 to 4		Match 1: Student vs Random Result: 18 to 2	
Match 2: ID_Improved vs MM_Null Result: 17 to 3		Match 2: Student vs MM_Null Result: 18 to 2	
Match 3: ID_Improved vs MM_Open Result: 14 to 6		Match 3: Student vs MM_Open Result: 15 to 5	
Match 4: ID_Improved vs MM_Improved Result: 15 to 5		Match 4: Student vs MM_Improved Result: 12 to 8	
Match 5: ID_Improved vs AB_Null Result: 13 to 7		Match 5: Student vs AB_Null Result: 18 to 2	
Match 6: ID_Improved vs AB_Open Result: 12 to 8		Match 6: Student vs AB_Open Result: 12 to 8	
Match 7: ID_Improved vs AB_Improved Result: 13 to 7		Match 7: Student vs AB_Improved Result: 13 to 7	

Conclusion

As I mentioned in the introduction, I didn't come up with a really good heuristic for this problem. I wanted to have a heuristic that succeed more than 80+% of the time but I didn't really find any easy solution. Yet I provide my analysis about which path I chose to follow to find new potentially good heuristic.

Actually, under other circumstances. Here are an idea that can help come up with a good (non trivial) heuristic :
-Find several simple features like `#my_moves`, `#my_moves - #opp_moves`, `#dist_to_opponent`, `#dist_to_square`,...
- pass each features to a supervised machine learning algorithm and tweak each weight corresponding to each features until we maximize the performance of the heuristic.

... But this is far more complex and time consuming than trying to find heuristic by hand

I think the good to find a good heuristic is to play a lot to the game. Also we need to combine easy future and find the right weight that maximize the performance. Finally one good idea is to change the heuristic of hour player over time or over certain condition. For example at the end of the party I can prefer to survive rather than trying to block my opponent. I think we a good adjustment of those ideas and a good computer we can come up with a good heuristic.

Note : The results (score) have a high variance, for some configuration some of my heuristic outperform **ID_Improved** by 5%, for some **ID_Improved** outperform my heuristic by 5%. I commented all heuristics in the python file. According to the benchmark I think attacking heuristic is the best I could come up with. So I choose to call `attacking()` in `custom_score()`

Why is it a good heuristic ?

Attacking heuristic is a good heuristic for several reasons.
Firstly, it is a semantically good heuristic because it tries to reduce the number of possible moves of the opponent. Actually that could be a good strategy used by any common human. This heuristic is **very effective** if my opponent is on the edge of the game, because in this case my opponent doesn't have lot's of choice (see board p2). And so, If there are already filled square my opponent will have even less way out… and I won't let him go out that easily because my heuristic purpose is to decrease is number of way… Yes I'm evil !

Secondly it is a good Heuristic in term of **complexity**. Actually the only thing the heuristic had to compute is the reverse of `#opp_moves`. So it is very efficient to compute, not like ***longest*** heuristic that I implemented and that is recursive (complexity exponential in term of the depth of the tree to visit). So it is very efficient. Furthermore instead of computing `1/#opp_moves` we could just compute `-#opp_moves` which I think is even slightly faster (in term of assembly operation).

Then I added Tweak to add extra point if my player can get closer to the opponent. Why ? Because If I cannot reduce the number of `#opp_moves` at the next step, the idea is that I can maybe reduce it at the step after the next step If I'm close enough at the next step (an so on…)

Finally I added a tweak to favor my next step to be the one with the most number of my moves in case there is a tie between two or more branches of my decision tree. Hence if there is a tie (same weight for different branches if I only compute `1/#opp_moves`), my heuristic will choose the next step that have the most liberty of movement (because now the weight on each branches is : $1/\#opp\_moves + epsilon_{(espilon < 1)}$