Artificial Intelligence Nanodegree

Heuristic analysis

Project 2: Build a Game-Playing Agent

Antonio Ferraioli

These pages provide a brief discussion about heuristic functions used to solve *Isolation* board game and their relative implementation. Four different heuristic functions are presented, each one is a modified `improved_score` function.

Linear combination score

This function is simply a linear combination

$$\text{Score}(L_1, L_2) = a\, L_1 + b\, L_2$$

where L_1 and L_2 are the number of moves available to the two players and a, b are weights. The code implementation is the following.

```
def custom_score(game, player):

    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    # get legal moves
    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))

    a=1; b=-3;
    return float( a*len(own_moves) + b*len(opp_moves) )
```

Taxicab score

This heuristic function is extremely simple and uses taxicab metric. Here a move has a good score if it involves a large number of successive legal moves L and if the taxicab distance d between players locations is large. If we denote the players locations with P_1 and P_2, we can write:

$$\text{Score}(P_1, P_2, L) \;=\; k\, L\, d$$

where k is a weight. The code implementation is the following.

```
def custom_score_2(game, player):

    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    # get legal moves
    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))

    # get locations
    x, y = game.get_player_location(player)
    X, Y = game.get_player_location(game.get_opponent(player))

    k = 0.5
    return float( (len(own_moves) - len(opp_moves)) * k/(abs(x-X) + abs(y-Y)) )
```

Intersection score

This function is a linear combination with weights *a* and *b* that depends on a combination *D* of the number of moves available to the two players, and the number *I* of legal moves that players share at a certain stage of the game. Intersections should be more influential after several moves, when the game board is "crowded" (say near the end of the game); so it is useful to introduce a term *B* that varies according to the number of blank cells (*B* is the inverse of the number of blank cells). The formula is the following:

$$\text{Score}(B, D, I) = aD + bBI$$

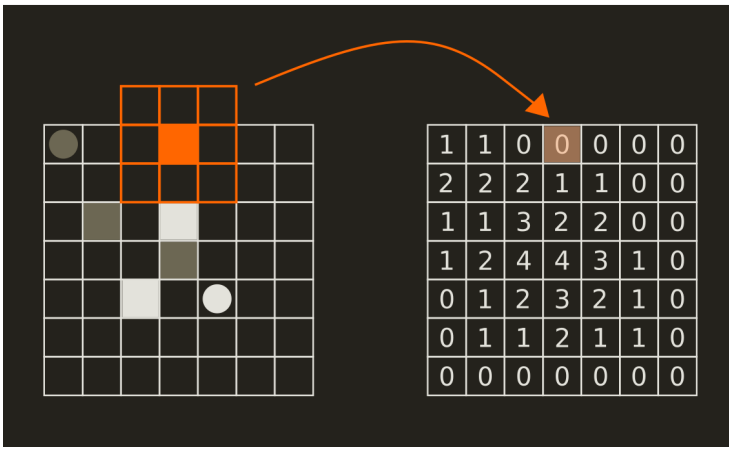
Below the code implementation.

```
def custom_score(game, player):

    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    # get legal moves
    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))

    # get number of blank cells
    blanks = len(game.get_blank_spaces())

    diff = len(own_moves) - 3*len(opp_moves)
```

```
    intersection = [move for move in own_moves if move in opp_moves]

    m = 1; n = 5
    return float(m*diff + 5*(1/blanks)*len(intersection))
```

2D correlation score

This criterion is based on 2D cross-correlation of binary gameboard matrix with a user defined kernel. The 2D correlation mode called "same" is used in order to obtain a resulting matrix with the same size as the original gameboard matrix. In tests a 3x3 kernel matrix, whose entries are all equal to one, is considered. For such kernel the resulting 2D correlated matrix has an intuitive meaning: each of its entries approximately represent the local density of neighboring unavailable squares. One could think of several strategies using this matrix. For example, a "good" move would be a move for which the correlation matrix on the location is small: in other words, the player's neighbor is less crowded. Here low density regions are supported and high density ones are penalized; note that this is just an *assumption*. Moreover, 2D correlation operation intrinsically introduce a numerical imbalance between center positions and boundary positions; this numerical difference could be exploited. The picture below shows a 3x3 kernel with all entries equal to 1 sliding on the game board and the resulting 2D correlation matrix.



Fix a board state: the formula expressing this heuristic score is

$$\text{Score}(L_1, L_2, P, k_i) = (aL_1 + bL_2) - kc_{xy}$$

where L_1 and L_2 denote the numbers of legal moves for each player and c_{xy} is the (x, y)-entry of correlation matrix and depends on kernel entries k_i and on the location $P=(x, y)$. The real numbers a and b represents weights. The values $a = 1$, $b = -3$ are used for testing but these are just arbitrary values because no real exhaustive search for optimal weights was carried out due to lack of time and hardware power. Note that kernel weights can be different from 1, even though in that case it is not so easy to spot a clear pattern in the resulting matrix. For example, the 3x3 kernel

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

has a similar performance, but the intuitive reason for that is not immediate.

The code implementation is the following.

```
def custom_score_4(game, player):

   if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    # get player and opponent moves
    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))
    location = game.get_player_location(player)

    # create game board matrix
    l = np.array(game._board_state)
    l=l[:-3].reshape((game.width,game.height))

    # set the kernel (for simplicity all entries are 1)
    # dimensions and weights can be changed
    ker = np.ones((3,3), dtype=np.float32)

    # 2D cross-correlation matrix
    corr = correlate2d(l,ker,mode='same')

    return float( (len(own_moves) - 3*len(opp_moves)) - 0.5*corr[location] )
```

Tests and heuristics performance

Two tests were performed, the first with `NUM_MATCHES=15` (30 games per opponent) and the second with `NUM_MATCHES=50` (100 games per opponent). The `timeout` parameter was set to 30 for both tests. Lower timeout values (say less than 5) could lead to numerous timeouts. The test script evaluates the performance of the `custom_score` evaluation function against a baseline agent using alpha-beta search and iterative deepening (ID) called `AB_Improved`. The three `AB_Custom` agents use ID and alpha-beta search with the `custom_score` functions defined in `game_agent.py`. Here are the results.

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3		AB_Custom_4	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	23	7	24	6	21	9	**26**	4	24	6
2	MM_Open	16	14	**20**	10	**20**	10	**20**	10	**20**	10
3	MM_Center	23	7	23	7	21	9	**26**	4	25	5
4	MM_Improved	22	8	**23**	7	20	10	20	10	16	14
5	AB_Open	16	14	17	13	16	14	**18**	12	**18**	12
6	AB_Center	18	12	16	14	19	11	17	13	**20**	10
7	AB_Improved	**16**	14	14	16	8	22	13	17	10	20
	Win Rate:	63.8%		65.2%		59.5%		**66.7**%		63.3%	

Match #	Opponent	AB_Improved		AB_Custom		AB_Custom_2		AB_Custom_3		AB_Custom_4	
		Won	Lost	Won	Lost	Won	Lost	Won	Lost	Won	Lost
1	Random	78	22	**88**	12	83	17	84	16	87	13
2	MM_Open	**75**	25	68	32	62	38	63	37	71	29
3	MM_Center	72	28	77	23	79	21	81	19	**83**	17
4	MM_Improved	63	37	**68**	32	61	39	**68**	32	63	37
5	AB_Open	50	50	**59**	41	48	52	50	50	49	51
6	AB_Center	54	46	**61**	39	59	41	53	47	49	51
7	AB_Improved	**52**	48	42	58	43	57	49	51	46	54
	Win Rate:	63.4%		**66.1**%		62.1%		64.0%		64.0%	

A first look reveals that *there is no large performance gap between different functions*. The poorer performance (`AB_Custom_2`) corresponds to the *taxicab score* that, in any case, performs similarly to `AB_Improved`. Greater distance between player locations is not so incisive, at least under these conditions. Overall there is a slight convenience for using *linear combination score* function (`AB_Custom`) and *Intersection score*.

Intersection score is partially based on the `improved_score` algorithm but provides additional tweakings through weights *a* and *b* and the intersection feature. Non-void intersections are limitations in player's mobility, one can emphasize or ignore this feature acting on weight *b*. In

the final stages of a game, a move that involves intersections can make the difference because it determines obstructions, loss of mobility, even the end of the game.

2D correlation score is almost useless except against agents using `center_score`. The neighbors are "blurry" and not so meaningful. It would be fine to learn the kernel weights somehow and give an interpretation to the resulting matrix. Another 2D correlation score shortcoming is its computational expensiveness. 2D correlation score performs quite good compared to `center_score`; other scores tend to underestimate peripheral positions in favor of central positions and this is generally not the case for 2D correlation score (because it penalizes crowded positions and central positions).

Recommended heuristic

As observed before, it is not easy to recommend a particular evaluation function observing performance ratios because

- the examined evaluation functions give similar performance ratio (and this is confined in the range 59% - 67%);
- score functions performance ratio is extremely variable (several tests give different results).

However, the best heuristic one can choose is *linear combination score* (used by `AB_Custom` player) for the following reasons.

Performance in tournament

In tournaments (see tables), this function gives the overall best results against other scores. Tests may mislead but (on average) its performance is no worse than `improved_score` (opponent: `AB_improved`), as shown below for a sets of 400 games.

```
                    Match #   Opponent     AB_Custom
                                           Won | Lost
                       1     AB_Improved   204 |  196
-------------------------------------------------------------------------
                              Win Rate:      51.0%
```

Computational cost

The algorithm is the cheapest of the four custom functions examined because, *on average*, the other three

- all have to store – *other than* the legal moves for both players – additional game board submatrices data (locations, board state, blank spaces, intersections);
- make more operations (except checks and legal moves storage, linear combination score performs only 2 variable assignments, two array length evaluation, 2 multiplications, 1 sum and 1 type conversion for each call).

On the testing device, the average execution time for a single call to linear combination score is $\sim 10^{-5}$ seconds, for the expensive 2D correlation score is $\sim 10^{-4}$ seconds.

Depth searched in game tree

Search depth is quite shallow, it rarely goes beyond depth 20. On a set of 50 games against `AB_improved` agent, search depth value was ≥30 only 18 times. Things are essentially the same with the other three custom scores.

How does the heuristic treat the player and the opponent differently?

If a=1 and b is negative, this evaluation function seeks for higher number of player possible moves while getting in the opponent's way: a player should have more options (more mobility) than its opponent. So, for simplicity, fix a=1; the the function adequacy depends on the coefficient b. In fact, since numbers of legal moves are expressed by non-negative integers, once a and b are set all the distinct possible values assumed by $a\,L_1 + b\,L_2$ are

$$\{ -7, -6, -5, \ldots, 5, 6, 7 \}$$

for b=-1 and

$$\{ -21, -20, -19, \ldots, 5, 6, 7 \}$$

for b=-3. It is evident that b influences the function's range. So the problem of finding an optimal b arises: the range should be not too small (loss of diversification, several moves are grouped under the same value) nor too large (dilution of values, some move value does not correspond to the real move importance). In other words, the range should be tailored to the real importance of each move. The value b=-3 was find after several fights between agents using the same score function with a=1 and different values for parameter b.