Heuristic analysis

Christopher Brian Currin

Analysis

```
def max_player_path(game, player, max_path=0.):
    """
    Near end game it is important to have depth-first over breadth-first search to determine winner
    """
    moves = game.get_legal_moves()  # gets active players moves
    if game.is_winner(player):
        return max_path
    elif game.is_loser(player):
        return 0
    for move in moves:
        new_game = game.forecast_move(move)  # plays move into copy of the game (and changes
active player)
        mp = max_player_path(new_game, player, max_path + 1)
        max_path = max(max_path, mp)
    return max_path


def heuristic6(game, player):
    return max_player_path(game, player)
```

The heuristic searches until the end of the game and gives a score based on the maximum path it took. Future heuristics could find the minimum route to win.

As the heuristic essentially ignores the depth specified in minimax or alphabeta methods, this heuristic focuses on depth-first search rather than breadth-first search of possible outcomes. Because there are so many possibilities at the beginning of a game, this is very expensive, and is instead useful only towards the end of the game. Near the end of the game it is more useful to know who actually wins or loses than an estimate, and hence the usefulness of the heuristic.

```
def max_player_path_alt(game, player, max_path=0.):
    moves = game.get_legal_moves()  # gets active players moves
    if len(moves) == 0:
        return max_path
    for move in moves:
        new_game = game.forecast_move(move)  # plays move into copy of the game (and changes
active player)
        mp = max_player_path(new_game, player, max_path + 1)
        max_path = max(max_path, mp)
    return max_path


def heuristic5(game, player):
    """
    Returns the difference between the maximum depth a player can go for their moves
    """
    return max_player_path_alt(game, player) - max_player_path_alt(game,
game.get_opponent(player))
```

The heuristic Is similar to the previous one, except the value return from the subroutine (max_player_path_alt) is a max_path regardless of who wins or loses. The score returned by the heuristic is then a comparison between the player and the opponent.

```
def heuristic4(game, player):
    """
    The number of open spaces around the location of the player
    """
    location_player = game.get_player_location(player)
    blank_spaces = game.get_blank_spaces()
    open_space_player = 0.0
    for (m, n) in [(1, 0), (0, 1), (-1, 0), (0, -1), (1, 1), (-1, -1), (1, -1), (-1, 1)]:
        if (location_player[0] + m, location_player[1] + n) in blank_spaces:
            open_space_player += 1
    return open_space_player
```

The heuristic values the amount of open space around a player. It does better than ID_improved by a relatively small margin. Because it quantifies open space around a player and not actual moves, this can be misleading near the end of the game where moving to a space with open space around it may not actually be that useful for knight-style movement. It does however seem to perform quite well middle-game when mixed with other heuristics.

```python
def heuristic3(game, player):
    """
    The number of open spaces around the location of the player's future move
    """
    best_num_open_spaces = 0.0
    for legal_move in game.get_legal_moves():
        new_game = game.forecast_move(legal_move)
        location_player = new_game.get_player_location(player)
        blank_spaces = new_game.get_blank_spaces()
        open_space_player = 0.0
        for (m, n) in [(1, 0), (0, 1), (-1, 0), (0, -1), (1, 1), (-1, -1), (1, -1), (-1, 1)]:
            if (location_player[0] + m, location_player[1] + n) in blank_spaces:
                open_space_player += 1
        best_num_open_spaces = max(best_num_open_spaces, open_space_player)
    return best_num_open_spaces
```

Similar to the previous heuristic except it does an approximate lookahead for each possible move too (in an attempt to better the end-game poor performance), but does poorer in practice.

```python
def heuristic2(game, player):
    """
    Difference between the number of open spaces around the location of the players
    """
    location_player = game.get_player_location(player)
    location_opp = game.get_player_location(game.get_opponent(player))
    blank_spaces = game.get_blank_spaces()
    open_space_player = 0.0
    open_space_opp = 0.0
    for (m, n) in [(1, 0), (0, 1), (-1, 0), (0, -1), (1, 1), (-1, -1), (1, -1), (-1, 1)]:
        if (location_player[0] + m, location_player[1] + n) in blank_spaces:
            open_space_player += 1
        if (location_opp[0] + m, location_opp[1] + n) in blank_spaces:
            open_space_opp += 1
    return open_space_player - open_space_opp
```

Similar to the previous heuristics (heurisic3 and heuristic4), but compares the score to the opponent, so that it favours moves that result in more open space than the opponent. In the early game this is therefore quite arbitrary, but definitely performs fairly well middle game. Performance is about the same as heuristic4 (slightly better than ID_improved).

```
def heuristic1(game, player):
    """
    Number my moves vs opposition moves
    similar to improved score
    (no player win/lose values as this heuristic is used in the preferred heuristics where this is not
necessary)
    """
    num_moves_player = len(game.get_legal_moves(player))
    num_moves_opposition = len(game.get_legal_moves(game.get_opponent(player)))
    return float(num_moves_player - num_moves_opposition)
```

Similar to the improved_score supplied heuristic, but included here due to slight performance improvement when mixed with other heuristics. That is, it does not check whether a move results in a win or lose; as in a mixed-heuristic method, this is typically done by a max_path_length heuristic such as heuristic 5 or 6.

Mixed heuristics

```
def heuristic_main(game, player):
    """
    weight number of moves more highly at the beginning of the game, then amount of open space
    and finally, maximum path length (winner will have max length)
    each measure for player is compared to opponent's measure
    """
    prop = len(game.get_blank_spaces()) / total_board_spaces
    if prop < 0.2:
        return heuristic5(game, player)
    return prop * heuristic1(game, player) + heuristic2(game, player)
```

The heuristic calculates how far into the game the game it is (`prop`) and biases against heuristic1 over the course of the game. For end game (defined as prop < 0.2), heuristic5 is solely used, which tries to search to the end of the game for a winner. Performs consistently better than ID_improved by about 10-15%.

```
def heuristic_main_staggered(game, player):
    """
    use number of moves at the beginning of the game, then amount of open space
    and finally, maximum path length (winner will have max length)
    each measure for player is compared to opponent's measure
    """
    game_stage = len(game.get_blank_spaces()) / total_board_spaces
    if game_stage > 0.7:
        # early game
        return heuristic1(game, player)
    elif game_stage < 0.2:
        # late game
        return heuristic5(game, player)
    else:
        # middle game
        return heuristic2(game, player)
```

Similar to previous heuristic, except heuristics are solely responsible for certain points in the game. The heuristic performs significantly worse than the above heuristic. Clearly a weighted balance between both heuristic1 and heuristic2 measures is better than a single measure, even if it is faster to computer a single heuristic.

```python
def heuristic_mainalt(game, player):
    """
    weight number of moves more highly at the beginning of the game, then amount of open space
    and finally, maximum path length (winner will have max length)
    """
    prop = len(game.get_blank_spaces()) / total_board_spaces
    if prop < 0.2:
        return heuristic6(game, player)
    return prop * heuristic1(game, player) + heuristic2(game, player)
```

Similar to previous heuristic_main, but uses heuristic6 instead of heuristic5 (see each one's explanation for more information). Performs slightly better than heuristic_main.

```python
def heuristic_main_alt_01(game, player):
    """
    weight number of moves more highly at the beginning of the game, then amount of open space
    and finally, maximum path length (winner will have max length)
    """
    prop = len(game.get_blank_spaces()) / total_board_spaces
    if prop < 0.1:
        return heuristic6(game, player)
    return prop * heuristic1(game, player) + heuristic2(game, player)


def heuristic_main_alt_03(game, player):
    """
    weight number of moves more highly at the beginning of the game, then amount of open space
    and finally, maximum path length (winner will have max length)
    """
    prop = len(game.get_blank_spaces()) / total_board_spaces
    if prop < 0.3:
        return heuristic6(game, player)
    return prop * heuristic1(game, player) + heuristic2(game, player)
```

The above 2 heuristics are slight variations in terms of defining end game (prop).

```python
def heuristic_main_alt_less(game, player):
    """
    weight number of moves more highly at the beginning of the game, then amount of open space
    """
    prop = len(game.get_blank_spaces()) / total_board_spaces
    return prop * heuristic1(game, player) + heuristic2(game, player)
```

Only relies on heuristic1 and heuristic2.

```python
def heuristic_main_alt_less_more(game, player):
    """
    weight number of moves more highly at the beginning of the game, then amount of open space
    near end game (when improved score gives a winner or loser, prop won't have an effect due to the
infinities)
    """
    prop = len(game.get_blank_spaces()) / total_board_spaces
    return prop * improved_score(game, player) + heuristic2(game, player)
```

Similar to above, but uses provided improved score heuristic to the use of infinities for end game results, which should be favoured.

```
def heuristic_mainalt_staggered(game, player):
    """
    use number of moves at the beginning of the game, then amount of open space
    and finally, maximum path length (winner will have max length)
    each measure for player is compared to opponent's measure
    """
    game_stage = len(game.get_blank_spaces()) / total_board_spaces
    if game_stage > 0.7:
        # early game
        return heuristic1(game, player)
    elif game_stage < 0.2:
        # late game
        return heuristic6(game, player)
    else:
        # middle game
        return heuristic2(game, player)
```

Similar to heuristic_mainalt and heuristic_main_staggered, and performs poorly.

Performance table

Heuristic	**Performance (% wins)**
ID_Improved	67.14%
heuristic_mainalt	73.57%
heuristic_mainalt_staggered	62.86%
heuristic_main_staggered	57.86%
heuristic_main	78.57%
heuristic_main_alt_less	72.86%
heuristic_main_alt_less_more	73.57%
heuristic_mainalt_prop	67.86%
heuristic1	57.86%
heursitic2	69.29%
heuristic3	57.86%
heuristic4	69.29%
heuristic5	NA
heuristic6	NA

Recommendations

heuristic_main, heuristic_mainalt and heuristic_main_alt_less_more have similar performance all consistently better than ID_improved. My recommendation would be to heuristic_main as it has the *best* (and *most consistent*) performance compared to ID_improved. It also scales well in terms of computing power. In terms of scaling the size of the board, heuristic_main_alt_less_more may scale better due to avoid depth-first search after a relatively arbitrarily-defined end-game point.

Possible improvements:

For long/complicated heuristics such as depth-first heuristics (heuristic5 and 6), it may be prudent to pass in the time_left object to return *some* value in case of a time_out (or close to it).

Consider reflections of the board near the beginning of the game to speed up computations so depth for iterative algorithms can be deeper.