LAPORAN TUGAS BESAR I

Implementasi *Minimax Algorithm* dan *Local Search* pada Permainan *Dots and Boxes*

Laporan dibuat untuk memenuhi salah satu tugas mata kuliah

IF3170 Inteligensi Buatan

Disusun oleh:

Adiyansa Prasetya Wicaksana	**13520044**
Andhika Arta Aryanto	**13520081**
Sarah Azka Arief	**13520083**
Aira Thalca Avila Putra	**13520101**

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

DAFTAR ISI

1. Penjelasan Objective Function

Objective function yang digunakan adalah menghitung *score point* (1 box = 20 poin) pada state saat ini ditambah dengan perhitungan berdasarkan konfigurasi *state* dan kondisi permainan saat ini. Perhitungan *score point* dilakukan karena tujuan akhir dari game adalah memaksimalkan poin agen dan meminimalkan poin musuh. Perhitungan berdasarkan konfigurasi state dilakukan untuk memaksimalkan probabilitas agen mendapatkan poin.

```python
def evaluate(self, state: GameState) -> int:
        score = 20*(self.count_point(state, self.my_turn) - self.count_point(state, not self.my_turn))
        if self.is_game_over(state):
            if self.count_point(state, self.my_turn) > self.count_point(state, not self.my_turn):
                score += 1000
            else:
                score -= 1000
        score_config = 0
        for i in range (0, state.board_status.shape[0]):
            for j in range (0, state.board_status.shape[1]):
                if abs(state.board_status[i][j]) == 3:
                    score_config += 10
                elif abs(state.board_status[i][j]) == 2:
                    score_config -= 2
        if (self.my_turn == state.player1_turn):
            score += score_config
        else:
            score -= score_config
        return score
```

Gambar 1.1 *Evaluation Function*

Berikut penjelasan pemberian poin berdasarkan konfigurasi *state*:

- *Score* box dengan banyak garis terisi sebanyak 3 memiliki dua kasus
    - Saat giliran agen, mendapat nilai plus 10 karena agen bisa menyelesaikan box nya dan mendapatkan poin.
    - Saat giliran musuh, mendapat nilai minus 10 karena artinya jika ada konfigurasi box dengan 3 garis, musuh bisa mendapatkan nilai dengan mudah
- Box dengan garis sebanyak 2 memiliki dua kasus

- Saat giliran agen, mendapatkan nilai minus 2 karena agen tidak mau membentuk garis tambahan di box tersebut karena hal itu bisa menyebabkan musuh bisa mendapatkan tambahan poin.
- Saat giliran musuh, mendapatkan nilai plus 2 karena kita ingin agen membentuk garis tambahan di box tersebut sehingga membentuk box dengan konfigurasi 3 garis.

Selain itu terdapat juga perhitungan poin saat state mencapai game over, agen akan memberi nilai +1000 jika perhitungan heuristic menyebabkan game over dan agen menang karena inilah output yang kita harapkan. Sebaliknya nilai -1000 jika musuh menang.

2. Implementasi *Minimax Alpha Beta Pruning*

Seperti yang kita ketahui, kompleksitas waktu algoritma *Minimax* adalah eksponensial. Untuk permainan *Dots and Boxes* dengan banyak titik sebanyak 4 x 4, maka ada 24 garis yang dapat dibentuk atau bisa dibilang ada 24 step dalam sebuah *game* untuk mencapai *goal state*. Dengan *branching factor* yang juga cukup besar (rata-rata *move* pada setiap *step* adalah 12 kemungkinan), maka tidak *feasible* untuk melakukan pencarian *Minimax* dari awal hingga akhir state. Maka dari itu, proses pencarian *move* yang kami lakukan memiliki *max_depth*, yaitu maksimal kedalaman *tree* yang akan diproses, atau dalam kata lain banyaknya step yang akan diperhitungkan dalam setiap *move* agen.

Algoritma *Minimax* dengan *Alpha Beta Pruning* yang kami buat memiliki beberapa parameter yaitu :

i. State sebagai **Initial State**, Merupakan state awal dari pencarian yang menginterpretasikan kondisi game saat giliran bot.
ii. *Depth*, Merupakan nilai ketinggian *tree* yang akan dicek atau bisa disebut banyak step yang akan dicek.
iii. *Alpha* dan *Beta*, merupakan nilai pembatas yang akan memangkas *expand node* yang tidak mungkin menjadi solusi.
iv. *Agent turn* berupa boolean, merupakan nilai yang mengindikasikan apakah algoritma harus memaksimumkan nilai atau meminimumkan nilai bergantung player yang sedang mendapat giliran.
v. *Timeout* berupa *integer* yang akan dijadikan acuan untuk menghentikan fungsi apabila durasi pencarian sudah melebihi *timeout* yang telah ditentukan. *Default value* dari parameter ini adalah 5 sesuai spesifikasi.

Algoritma ini mengeluarkan keluaran berupa **action** yang akan dilakukan agen beserta nilai *objective functionnya.* Langkah - langkah proses pencarian pada Algoritma *Minimax* dengan *Alpha Beta Pruning adalah sebagai berikut :*

1. Pertama-tama, akan dilakukan iterasi yang memanggil fungsi alphabeta dengan nilai *alpha* -infinity dan *beta* infinity agar keduanya pasti ter-*update*. Sebelum

iterasi dimulai akan disimpan waktu *timeout* berupa 4 detik dari proses dimulai untuk di-*passing* sebagai nilai *timeout* pada fungsi alphabeta. Iterasi dilakukan sebanyak langkah yang mungkin dilakukan dan akan berhenti apabila iterasi sudah selesai atau batas *timeout* sudah terlewati.

```python
# Mengembalikan aksi yang akan dilakukan
    def get_action(self, state: GameState) -> GameAction:
        self.my_turn = state.player1_turn
        self.hit_timeout = False
        timeout = time.time() + 4
        move_possible = np.count_nonzero(state.row_status == 0) + np.count_nonzero(state.col_status == 0)
        for i in range(1, move_possible + 1):
            if time.time() > timeout:
                break
            temp_action = self.alphabeta(state, i, -np.inf, np.inf, True, timeout)[1]
            if(self.hit_timeout == False and temp_action != None):
                self.action = temp_action
        return GameAction(self.action.action_type, (self.action.position[1], self.action.position[0]))
```

2. Untuk setiap pencarian, lakukan pengecekan apakah saat ini merupakan *base case*, yaitu jika ***depth*** sudah bernilai 0 atau permainan sudah berakhir. Jika salah satunya terpenuhi, lakukan perhitungan objective function dari state saat ini dan *return* nilainya, jika tidak lanjut ke step selanjutnya

```python
if depth == 0 or self.is_game_over(state):
    return self.evaluate(state), None
```

3. Cek apakah saat ini merupakan turn agen atau musuh, jika turn agen kita akan memaksimalkan nilai, jika turn musuh kita akan meminimalkan nilai.
4. *Generate* seluruh *move* yang mungkin dari state sekarang lalu inisiasi nilai *best_score* saat ini dengan nilai **-infinity** jika kita akan memaksimalkan dan nilai **infinity** jika kita akan meminimalkan nilai. Tujuannya adalah agar pada proses perhitungan neighbour yang pertama, nilai *best_score* ini pasti berubah. Lalu inisiasi variabel *best_move* dengan *move* pertama yang ada pada array seluruh *move* yang mungkin.

```python
all_moves = self.get_all_moves(state)
```

```python
def get_all_moves(self, state: GameState) -> list:
    all_moves = []
    for i in range (0, state.row_status.shape[0]):
        for j in range (0, state.row_status.shape[1]):
            if state.row_status[i][j] == 0:
                all_moves.append(GameAction("row", (i, j)))
    for i in range (0, state.col_status.shape[0]):
        for j in range (0, state.col_status.shape[1]):
            if state.col_status[i][j] == 0:
                all_moves.append(GameAction("col", (i, j)))
    random.shuffle(all_moves)
    return all_moves
```

5. Lakukan looping, untuk seluruh *move* yang ada pada *possible move* yang sudah di *generate,* kita *copy* state saat ini dan **jalankan simulasi** melakukan *move* yang sedang di iterasi terhadap state yang sudah di copy.Lalu lakukan pemanggilan alphabeta kembali (rekursif) dengan state yang sudah diupdate (state simulasi) dan *depth* yang berkurang satu (karena sudah ada satu *move* yang dijalankan), dengan nilai boolean player bergantung pada hasil simulasi. Jika hasil simulasi tidak terjadi perubahan *turn* (dalam hal ini berhasil membuat box), maka nilai boolean di*assign* dengan nilai boolean sebelumnya, namun jika terjadi perubahan *turn* maka nilai boolean pada pemanggilan rekursif diganti menjadi negasi dari nilai boolean sebelumnya.

Jika pencarian memaksimalkan nilai

```
for move in all_moves:
                current = deepcopy(state)
                current = self.apply_action(current, move)
                if(current.player1_turn == state.player1_turn):
                    maxEval = max(maxEval, self.alphabeta(current, depth - 1, alpha, beta, agent_turn, timeout)[0])
                else:
                    maxEval = max(maxEval, self.alphabeta(current, depth - 1, alpha, beta, not agent_turn, timeout)[0])
```

Jika pencarian meminimalkan nilai

```
for move in all_moves:
                current = deepcopy(state)
                current = self.apply_action(current, move)
                if(current.player1_turn == state.player1_turn):
                    minEval = min(minEval, self.alphabeta(current, depth - 1, alpha, beta, agent_turn, timeout)[0])
                else:
                    minEval = min(minEval, self.alphabeta(current, depth - 1, alpha, beta, not agent_turn, timeout)[0])
```

6. Lakukan pengecekan apakah nilai dari node merupakan nilai maksimal atau minimal yang baru (berdasarkan langkah 3 apakah memaksimalkan atau meminimalkan). Jika iya maka ubah best_score dengan nilai node dan ubah best_move dengan *move* yang disimulasikan.

Jika pencarian memaksimalkan nilai

```
if (maxEval > best_score):
    best_score = maxEval
    best_move = move
```

Jika pencarian meminimumkan nilai

```
if (minEval < best_score):
    best_score = minEval
    best_move = move
```

7. Lakukan *update* nilai alpha dan atau beta jika terjadi perubahan batas atas dan bawah.

Jika pencarian memaksimalkan nilai

```
alpha = max(alpha, maxEval)
```

Jika pencarian meminimukan nilai

```
beta = min(beta, minEval)
```

8. Jika nilai alpha dan beta sudah saling memotong (nilai *alpha* sudah melewati *beta)*, lakukan pemangkasan sehingga *move* selanjutnya pada array *possible moves* tidak perlu dicek.

```
if beta <= alpha:
    break
```

9. Fungsi akan me-*return best move* yang selanjutnya akan dieksekusi oleh agen pada *real game*.

```
return best_score, best_move
```

3. Implementasi *Local Search*

Algoritma *Local Search* yang akan digunakan pada penyelesaian permainan Dots and Boxes ini adalah *hill climbing with sideways move*. Penggunaan *local search* pada permainan ini merupakan penerapan *local search* pada suatu permainan yang memiliki *multi agent* dan pada kasus ini algoritma tidak akan menentukan langkah lawan. Berikut apabila dilihat aspek - aspek dari *local search* yang akan digunakan :

- Initial State : State awal dari pencarian adalah kondisi game saat sedang turn bot kita berlangsung atau setelah lawan melakukan gerakan terakhir.
- Neighbor : Semua kemungkinan gerakan dari state permainan sekarang
- Action : Berpindah ke neighbor dengan nilai tertinggi yang dihitung dengan fungsi objektif
- Solution : Move yang dilakukan berikutnya setelah initial state

Jadi, *local search* yang digunakan bersifat iteratif dimana *search* akan digunakan setiap *turn* bot kita akan dimulai. Initial state saat pencarian dilakukan adalah state game saat ini setelah lawan telah memberi gerakannya dan lalu akan dicari neighbour dengan nilai tertinggi. Bila dibuat menjadi langkah yang terstruktur seperti berikut :

1. State awal berupa keadaan permainan saat itu dengan semua garis dan kotak yang sudah terbentuk
2. Bot akan membuat *neighbor* berupa semua kemungkinan gerakan yang bisa dilakukan oleh bot saat itu
3. Dilakukan *for loop* sebanyak semua kemungkinan gerakan, dan pada tiap *loop* agen akan mensimulasikan state baru berdasarkan tiap gerakan
4. State baru yang dihasilkan dihitung nilainya dengan fungsi objektif yang telah dibuat
5. Agen menemukan *neighbor* dengan value tertinggi lalu akan mengembalikan aksi yang akan dilakukan sesuai state game pada *neighbor* tersebut
6. Pencarian selesai dan lalu giliran lawan, *local search* akan dilakukan lagi dari langkah 1 saat sudah kembali giliran bot kami

Adapun algoritma *local search* yang dipilih adalah *hill-climbing with sideway move* dikarenakan pemilihan *successor*-nya. Pemilihan *successor* pada *hill-climbing with sideway move* adalah memilih *neighbor* dengan *objective function* paling tinggi. Sedangkan, pada *simulated annealing* pemilihan *successor*-nya berdasarkan *random neighbor* padahal turn yang terdapat pada game Dots and Boxes itu terbatas sehingga setiap turn berdampak cukup signifikan pada kemenangan. Dan juga *genetic algorithm* tidak dapat digunakan sebagai algoritma karena karakteristiknya yaitu *crossover* dan *mutation* dapat merubah *state* yang bukan seharusnya.

Jika dibandingkan dengan *hill climbing* tanpa variasi, *hill climbing with sideways move* lebih menjamin untuk dapat melewati *shoulder*. Selain itu, dibandingkan dengan *stochastic hill-climbing* dikarenakan alasan yang sama yaitu pemilihan *successor* secara random. Dengan beberapa pertimbangan tersebut, terpilih *hill-climbing with sideways move* sebagai algoritma *local search* pada permainan Dots and Boxes.

4. Hasil Pertandingan

a. Bot *Minimax* vs. Manusia

Match Desc.		Player	Score
Match 1	1st Player	Bot *Minimax*	8
	2nd Player	Manusia	1
Match 2	1st Player	Manusia	4
	2nd Player	Bot *Minimax*	5
Match 3	1st Player	Bot *Minimax*	5
	2nd Player	Manusia	4
Match 4	1st Player	Manusia	3
	2nd Player	Bot *Minimax*	6
Match 5	1st Player	Bot *Minimax*	7
	2nd Player	Manusia	2
Wins (*Minimax* : Manusia) = 5 : 0			

b. Bot *Local Search* vs. Manusia

Match Desc.		Player	Score
Match 1	1st Player	Bot *Local Search*	3
	2nd Player	Manusia	6
Match 2	1st Player	Manusia	8
	2nd Player	Bot *Local Search*	1
Match 3	1st Player	Bot *Local Search*	4
	2nd Player	Manusia	5
Match 4	1st Player	Manusia	5

	2nd Player	Bot *Local Search*	4
Match 5	1st Player	Bot *Local Search*	5
	2nd Player	Manusia	4
Wins (*Local Search* : Manusia) = 1 : 4			

c. Bot *Minimax* vs. Bot *Local Search*

Match Desc.		Player	Score
Match 1	1st Player	Bot *Minimax*	7
	2nd Player	Bot *Local Search*	2
Match 2	1st Player	Bot *Local Search*	1
	2nd Player	Bot *Minimax*	8
Match 3	1st Player	Bot *Minimax*	6
	2nd Player	Bot *Local Search*	3
Match 4	1st Player	Bot *Local Search*	4
	2nd Player	Bot *Minimax*	5
Match 5	1st Player	Bot *Minimax*	8
	2nd Player	Bot *Local Search*	1
Wins (*Minimax* : *Local Search*) = 5 : 0			

d. Persentase Kemenangan Bot Minimax dan Local Search

Persentase kemenangan bot *minimax* (dari 10 *match*): **100%**

Persentase kemenangan bot *local search* (dari 10 *match*): **10%**

5. Kesimpulan dan Saran

Algoritma *minimax* dan *local search* dapat digunakan untuk mencari solusi terbaik pada permainan *Dots and Boxes*. Berdasarkan implementasi yang telah dilakukan, didapat bahwa performa dari kedua algoritma tersebut sangat dependen terhadap fungsi objektif yang digunakan pada penerapannya. Oleh karena itu, apabila ingin dilakukan perbaikan maka aspek yang dapat ditingkatkan dan diperbaiki dari kode program berkutat di fungsi objektif yang digunakan seperti membuat rincian objektif program yang lebih optimal. Selain itu, aspek lain yang dapat ditingkatkan seputar pembuatan kode program dapat berupa pembagian kerja yang lebih efisien dan juga pemetaan desain algoritma sebelum menulis kode.

6. Kontribusi Anggota

No.	NIM	Nama	Kontribusi
1.	13520044	Adiyansa Prasetya Wicaksana	Membuat Bot LocalSearch, Membuat Laporan
2.	13520081	Andhika Arta Aryanto	Membuat Bot LocalSearch, Membuat Laporan
3.	13520083	Sarah Azka Arief	Membuat Bot Minimax, Membuat Laporan
4.	13520101	Aira Thalca Avila Putra	Membuat Bot Minimax, Membuat Utility Function, Membuat Laporan