# Penerapan Algoritma *Branch and Bound* untuk Penyusunan Strategi Penambangan *Item* pada Permainan Growtopia

Naufal Adnan - 13522116
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13522116@std.stei.itb.ac.id

***Abstract*—Algoritma *Branch and Bound* merupakan salah satu algoritma yang dapat digunakan dalam persoalan optimasi yang melakukan pencarian pada pohon ruang status. Strategi penambangan *item* pada permainan Growtopia membutuhkan teknik minimasi jarak sehingga pemain dapat berjalan dari titik awal (*base*) untuk mengambil seluruh *item*, kemudian kembali lagi menuju *base.* Oleh karena itu, penggunaan algoritma *Branch and Bound* dapat menjadi solusi yang efektif untuk mencari rute penambangan yang optimal dalam permainan Growtopia. Hasil percobaan juga menunjukkan bahwa dengan menggunakan algoritma *Branch and Bound* dengan matriks ongkos-tereduksi, diperoleh rute optimal untuk penambangan *item*.**

***Keywords*—branch and bound; optimasi; minimasi; growtopia; reduced cost matrix.***

## I. Latar Belakang

Growtopia merupakan sebuah permainan *sandbox* yang memungkinkan seorang pemain berinteraksi dalam dunia virtual, membangun berbagai struktur, dan mengumpulkan *item.* Salah satu aktivitas utama dalam permainan ini adalah penambangan *item*, di mana pemain harus berjalan dari satu titik ke titik lainnya untuk mengumpulkan berbagai *item* yang tersebar di area permainan. Efisiensi dalam proses penambangan menjadi kunci penting, terutama untuk meminimalkan waktu dan usaha yang diperlukan untuk mengumpulkan semua *item* yang diinginkan dan kembali ke titik awal (*base*).

Fig. 1. Ilustrasi penambangan *item* pada permainan Growtopia (*Sumber: dokumen pribadi*)

Dalam konteks permainan Growtopia ini, persoalan penambangan *item* dapat dimodelkan sebagai masalah *Travelling Salesman Problem* (TSP), di mana seorang pemain (*salesman*) harus mengunjungi sejumlah titik (*item*) dengan total jarak perjalanan seminimal mungkin sebelum akhirnya kembali lagi ke titik awal. Mengingat kompleksitas dari *Travelling Salesman Problem* (*non decision problem*) merupakan persoalan *NP-Hard*, maka dibutuhkan algoritma optimasi yang efektif dan efisien untuk menentukan rute terbaik yang harus diambil oleh pemain.

Algoritma *Branch and Bound* merupakan salah satu teknik yang dapat digunakan untuk menyelesaikan masalah optimasi seperti *Travelling Salesman Problem* (TSP). Algoritma *Branch and Bound* bekerja dengan cara melakukan eksplorasi ruang pencarian secara sistematis melalui struktur pohon. Pada struktur pohon tersebut, setiap simpul (*node*) dalam pohon mewakili sub-masalah dari masalah utama. Algoritma ini menggunakan batasan (*bound*) untuk memotong cabang pohon yang tidak menjanjikan, sehingga mengurangi jumlah sub-masalah yang perlu dievaluasi. Dengan demikian, algoritma *Branch and Bound* mampu menemukan solusi optimal dengan mengeksplorasi lebih sedikit kemungkinan dibandingkan dengan pencarian secara *brute force*.

Penerapan algoritma *Branch and Bound* dalam strategi penambangan *item* pada permainan Growtopia bertujuan untuk menemukan rute penambangan yang paling efisien. Dalam pengembangannya algoritma ini bisa tidak hanya memperhitungkan jarak antar titik, tetapi juga mempertimbangkan berbagai kendala dan batasan yang mungkin ada dalam permainan, seperti waktu yang tersedia, energi pemain, dan hambatan-hambatan di sepanjang jalur penambangan. Dengan menggunakan algoritma *Branch and Bound*, pemain dapat mengoptimalkan perjalanan mereka dalam mengumpulkan *item*, sehingga dapat meningkatkan produktivitas dan efisiensi dalam permainan. Melalui pendekatan ini, diharapkan pemain dapat memperoleh keuntungan maksimal dari waktu dan usaha yang diinvestasikan dalam permainan, serta memberikan kontribusi bagi pengembangan lebih lanjut dalam bidang optimasi *game*.

## II. Teori Dasar

### A. *Algoritma Branch and Bound*

Algoritma *Branch and Bound* (B&B) adalah teknik pencarian sistematis yang digunakan untuk menyelesaikan

masalah optimasi, baik untuk meminimalkan maupun memaksimalkan suatu fungsi objektif dengan memperhitungkan batasan-batasan (*constraints*) tertentu. Algoritma *Branch and Bound* merupakan pengembangan lebih lanjut dari algoritma *Backtracking*. Kedua algoritma ini memiliki persamaan, yaitu melakukan pencarian pada pohon ruang status, serta pemangkasan (*prunning*) cabang pohon ruang status yang tidak mengarah ke solusi. Namun, algoritma *Branch and Bound* digunakan untuk menyelesaikan persoalan optimasi, sedangkan algoritma *Backtracking* umumnya untuk persoalan non-optimasi.

Algoritma branch and bound biasa digunakan untuk persoalan *NP-hard global optimization*. Dalam persoalan optimasi, dikenal istilah solusi memungkinkan dan solusi optimal. Solusi yang memungkinkan adalah suatu simpul dalam pohon ruang status yang memenuhi seluruh batasan yang ditetapkan untuk persoalan tersebut. Solusi optimal merupakan solusi memungkinkan yang memiliki nilai fungsi objektif terbaik (Levitin, 2012).

Algoritma *Branch and Bound* memiliki karakteristik sebagai berikut.

1. Solusi optimal: memastikan bahwa solusi terbaik dapat ditemukan melalui evaluasi menyeluruh terhadap semua kemungkinan jalur yang relevan.
2. Batas (*upper and lower bound*): nilai terbaik dari fungsi objektif pada setiap solusi yang mungkin, dengan menambahkan komponen pada solusi sementara yang direpresentasikan oleh simpul.
3. Pemangkasan (*prunning*): mengeliminasi sub-masalah yang tidak mungkin mengandung solusi optimal atau tidak layak untuk dieksplorasi lebih lanjut.
4. *Backtracking*: kembali ke simpul sebelumnya dalam pohon pencarian ketika mencapai jalan buntu atau ketika ditemukan solusi yang lebih baik.

Dari kedua elemen tersebut, maka setiap simpul pencarian diberi nilai batas yang mencerminkan taksiran efisiensi atau biaya dari solusi yang mungkin dicapai melalui simpul tersebut. Pada setiap tahap pencarian, nilai batas dari simpul yang sedang dievaluasi dibandingkan dengan nilai batas dari solusi terbaik yang telah ditemukan sejauh ini. Jika nilai batas simpul yang sedang dievaluasi tidak lebih baik daripada nilai batas solusi terbaik yang ada, maka simpul tersebut dipangkas atau diabaikan dari eksplorasi lebih lanjut. Proses pemangkasan ini dilakukan berdasarkan kriteria tertentu yang tergantung pada jenis persoalan optimasi.

Untuk persoalan maksimasi, simpul dipangkas jika nilai simpul tersebut tidak lebih besar daripada nilai solusi terbaik sejauh ini. Artinya, jika simpul yang sedang dievaluasi memiliki nilai batas yang lebih kecil atau sama dengan nilai batas solusi terbaik saat ini, simpul tersebut tidak akan diekspansi lebih lanjut karena tidak menjanjikan solusi yang lebih baik. Sebaliknya, untuk persoalan minimasi, simpul dipangkas jika nilai simpul tersebut tidak lebih kecil daripada nilai solusi terbaik sejauh ini. Dengan kata lain, jika simpul yang sedang dievaluasi memiliki nilai batas yang lebih besar atau sama dengan nilai batas solusi terbaik saat ini, simpul tersebut diabaikan karena tidak berpotensi memberikan solusi yang lebih efisien.

Pemangkasan ini adalah elemen kunci dari efisiensi algoritma *Branch and Bound*, karena memungkinkan melakukan eliminasi simpul-simpul yang tidak produktif dari ruang pencarian. Dengan begitu algoritma *Branch and Bound* memfokuskan sumber daya komputasi pada jalur-jalur yang lebih menjanjikan dalam menemukan solusi optimal. Pemangkasan ruang pencarian ini dilakukan dengan memperhatikan tiga kasus sebagai berikut.

1. Nilai batas suatu simpul tidak lebih baik daripada nilai batas solusi terbaik sejauh tahap tersebut.
2. Simpul tersebut tidak memenuhi batasan persoalan, dengan kata lain tidak memungkinkan.
3. Solusi pada simpul tersebut tunggal sehingga tidak memiliki pilihan lain. Dalam kasus ini, bandingkan nilai fungsi objektif solusi ini dengan solusi terbaik sejauh tahap tersebut, pilih yang terbaik.

Algoritma *Branch and Bound* menggabungkan beberapa aturan tertentu dalam pembangkitan simpulnya, yang umumnya menggunakan *best-first rule*. Pada algoritma *Breadth-First Search* (BFS) murni simpul berikutnya yang akan diekspansi dilakukan berdasarkan urutan pembangkitannya (FIFO). Sementara pada algoritma *Branch and Bound* setiap simpul diberi sebuah nilai *cost* ($\hat{c}(i)$), yaitu nilai taksiran lintasan termurah ke simpul status tujuan yang melalui simpul status *i*. Simpul berikutnya yang akan di-*expand* tidak lagi berdasarkan urutan pembangkitannya, tetapi simpul yang memiliki *cost* yang paling kecil atau *least cost search* (pada kasus minimasi). Dalam hal ini, algoritma *Branch and Bound* dapat dikatakan merupakan gabungan dari *Breadth-First Search* (BFS) + *least cost search*.

Langkah-langkah dalam algoritma *Branch and Bound* secara umum dapat digambarkan sebagai berikut.

1. Masukkan simpul akar ke dalam antrian. Jika simpul akar adalah simpul solusi (*goal node*), maka solusi telah ditemukan. Hentikan pencarian jika hanya satu solusi yang diinginkan.
2. Jika antrian kosong, pencarian dihentikan.
3. Jika antrian tidak kosong, pilih dari antrian simpul *i* yang mempunyai nilai *cost* $\hat{c}(i)$ paling kecil. Jika terdapat beberapa simpul *i* yang memenuhi, pilih satu secara sembarang.
4. Jika simpul *i* adalah simpul solusi, berarti solusi sudah ditemukan. Hentikan pencarian jika hanya satu solusi yang diinginkan. Pada persoalan optimasi dengan pendekatan *least cost search*, periksa cost semua simpul hidup. Jika *cost*-nya lebih besar dari cost simpul solusi, maka matikan simpul tersebut.
5. Jika simpul *i* bukan simpul solusi, maka bangkitkan semua anak-anaknya. Jika *i* tidak mempunyai anak, kembali ke langkah 2.
6. Untuk setiap anak *j* dari simpul *i*, hitung $\hat{c}(j)$, dan masukkan semua anak-anak tersebut ke dalam antrian.

7. Kembali ke langkah 2.

Penerapan algoritma *Branch and Bound* dapat digunakan untuk menyelesaikan berbagai jenis masalah optimisasi seperti *Traveling Salesman Problem* (TSP), *Knapsack Problem, Resource Allocation, Network Optimization, 16-Puzzle Problem,* dan sebagainya.

*B. Traveling Salesman Problem (TSP)*

*Travelling Salesman Problem* (TSP) merupakan masalah optimisasi klasik dalam ilmu komputer dan matematika diskrit, di mana seorang pedagang harus mengunjungi sejumlah kota, dengan setiap kota dikunjungi tepat satu kali, kemudian kembali ke kota asal dengan total jarak perjalanan yang seminimal mungkin.

Persoalan *Travelling Salesman Problem* (TSP) dapat diformulasikan sebagai berikut.

1. Diberikan $n$ buah kota (*vertex*) serta diketahui jarak (bobot) antara setiap kota satu sama lain.
2. Tujuan: menemukan perjalanan dengan jarak terpendek yang melalui setiap kota satu kali dan kembali ke kota asal.

TSP termasuk dalam kategori masalah *NP-Hard*, yang berarti tidak ada algoritma yang efisien untuk menyelesaikan masalah ini secara umum. Jumlah kemungkinan tur dalam graf lengkap dengan $n$ kota adalah $(n-1)!$, yang membuat pencarian *brute force* tidak praktis untuk jumlah simpul yang besar. Oleh karena itu, diperlukan algoritma optimisasi seperti *Branch and Bound* yang dapat mengurangi ruang pencarian dan menemukan solusi optimal dengan lebih efisien.

Penyelesaian persoalan TSP dengan algoritma *Branch and Bound* dilakukan dengan menentukan *cost* untuk setiap simpul di dalam pohon ruang status yang menyatakan nilai batas bawah (*lower bound*) ongkos mencapai simpul solusi dari simpul tersebut. *Cost* setiap simpul dapat dihitung secara *heuristic* berdasarkan salah satu dari dua cara, yaitu matriks ongkos-tereduksi (*reduced cost matrix*) dari graf dan bobot minimum tur lengkap.

1. *Cost* Berdasarkan *Reduced Cost Matrix*

   *Cost* untuk setiap simpul dihitung dengan menggunakan matriks ongkos-tereduksi (*reduced cost matrix*) dari graf $G$. Matriks ongkos-tereduksi merupakan matriks yang setiap kolom dan setiap barisnya mengandung paling sedikit satu buah nol dan semua elemen lainnya non-negatif. Pembentukan matriks ongkos-tereduksi dapat dilakukan dengan memilih elemen minimum pada baris $i$ (kolom $j$) yaitu $t$, lalu mengurangkan seluruh elemen pada baris $i$ (kolom $j$) dengan $t$ sehingga menghasilkan sebuah nol pada baris $i$ (kolom $j$) tersebut. Dengan mengulangi proses ini berulang kali akan menghasilkan matriks ongkos-tereduksi.

   Jumlah total elemen pengurang dari semua baris dan kolom menjadi batas bawah (*lower bound*) dari tur dengan total bobot minimum. Nilai ini digunakan sebagai nilai untuk simpul akar pada pohon ruang status.

   Misalkan terdapat persoalan sebuah perjalanan dimulai dari simpul $n$. $A$ adalah matriks tereduksi untuk simpul $R$. Kemudian $S$ adalah anak dari simpul $R$ sehingga sisi ($R$, $S$) pada pohon ruang status berkoresponden dengan sisi ($i$, $j$) pada perjalanan. Jika $S$ bukan simpul daun, maka matriks ongkos-tereduksi untuk simpul S dapat dihitung sebagai berikut.

   1. Ubah semua nilai pada baris $i$ dan kolom $j$ menjadi $\infty$ untuk mencegah agar tidak ada lintasan yang keluar dari simpul $i$ atau masuk pada simpul $j$.
   2. Ubah $A(j, n)$ menjadi $\infty$ untuk mencegah penggunaan sisi ($j$, $n$).
   3. Reduksi kembali semua baris dan kolom pada matriks $A$ kecuali untuk elemen $\infty$ sehingga menghasilkan matriks $B$.

   Secara umum, persamaan fungsi pembatas (*bounding function*) dengan matriks ongkos-tereduksi adalah sebagai berikut.

   $$\hat{c}(S) = \hat{c}(R) + A(i, j) + r$$

   dengan,

   - $\hat{c}(S)$: *cost* perjalanan minimum yang melalui simpul $S$ (simpul di pohon ruang status).
   - $\hat{c}(R)$: *cost* perjalanan minimum yang melalui simpul $R$ ($R$ adalah *parent* dari $S$).
   - $A(i, j)$: *cost* sisi ($i$, $j$) pada graf $G$ yang berkoresponden dengan sisi ($R$, $S$) pada pohon ruang status.
   - $r$: jumlah semua pengurang pada proses memperoleh matriks tereduksi untuk simpul $S$.

   Simpul yang memiliki *cost* ($\hat{c}(S)$) terkecil akan dijadikan simpul ekspansi berikutnya. Perhitungan tiap simpul berikutnya dilakukan dengan tahapan yang sama seperti pada sebelumnya.

2. Cost Berdasarkan Bobot Tur Lengkap

   Pendekatan *heuristic* lain dalam menghitung nilai *cost* untuk setiap simpul dapat dilakukan berdasarkan bobot tur lengkap. Bobot tur lengkap dapat dihitung sebagai berikut.

   $$M: cost = \text{bobot minimum tur lengkap}$$
   $$\geq \tfrac{1}{2} \Sigma \; \textit{bobot sisi } i_1 + \textit{bobot sisi } i_2$$

   Dalam hal ini, sisi $i_1$ dan sisi $i_2$ adalah sisi yang bersisian dengan simpul $i$ dengan bobot minimum.

   Nilai $M$ tersebut dapat digunakan sebagai fungsi pembatas (*bound*) untuk menghitung *cost* setiap simpul di dalam pohon ruang status. Pada penentuan *cost* untuk simpul anak-anaknya, digunakan formula yang

sama namun dengan mengambil sisi yang akan dilalui berikutnya. Misalkan, untuk menghitung *cost* simpul 2 pada persoalan gambar di bawah ini, untuk sisi $i_2$ = b, sisi (*a*, *b*) wajib diambil.

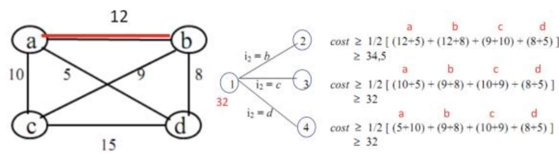



Fig. 2. Ilustrasi penambangan *item* pada permainan Growtopia (*Sumber*:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branchand-Bound-2021-Bagian3.pdf)

Simpul yang memiliki *cost* terkecil akan dijadikan simpul ekspansi berikutnya. Perhitungan tiap simpul berikutnya dilakukan dengan tahapan yang sama seperti pada sebelumnya.

### C. *Penambangan Item pada Permainan Growtopia*

Penambangan *item* dalam permainan Growtopia melibatkan pemain yang mengumpulkan berbagai *item* dengan cara menjelajahi dunia, memecahkan blok, dan berinteraksi dengan elemen-elemen permainan lainnya. *Item-item* yang diperoleh kemudian dapat digunakan untuk berbagai tujuan seperti *crafting*, perdagangan, atau sebagai bagian dari misi permainan. Proses penambangan ini memerlukan strategi untuk mengoptimalkan waktu dan usaha pemain agar mendapatkan *item* yang diinginkan secara efisien.

Masalah penambangan *item* dapat dimodelkan sebagai persoalan dari TSP. Misalnya, jika kita menganggap setiap blok atau lokasi sebagai "kota" dan biaya untuk berpindah antar blok atau lokasi sebagai "jarak" antara kota, maka tugas pemain adalah menemukan jalur penambangan yang meminimalkan total biaya. Dengan begitu, *Mapping* dari permasalahan penambangan *item* pada permainan Growtopia ke persoalan TSP adalah sebagai berikut.

- *Node*: blok atau lokasi *item* di dalam dunia permainan.
- *Edge*: biaya yang dibutuhkan untuk berpindah dari satu blok ke blok lain.
- *Tour*: Urutan tindakan yang diambil oleh pemain untuk menambang *item* secara optimal.

Dengan demikian, algoritma optimasi untuk TSP, seperti Branch and Bound, dapat diadaptasi untuk mengatur urutan optimal dari tindakan penambangan *item* dalam Growtopia.

## III. Perencanaan Rute Penambangan Menggunakan Algoritma Branch and Bound

### A. *Deskripsi Persoalan*

Dalam penerapan algoritma *Branch and Bound*, dimodelkan sebuah *field* tambang permainan Growtopia ke dalam petak-petak. Petak yang memiliki *item* akan ditandai dengan warna kuning, sementara petak tanpa *item* akan dibiarkan kosong berwarna putih. Untuk mengimplementasikan perencanaan rute penambangan, terdapat skenario permainan yang mungkin terjadi untuk aksi penambangan pada permainan Growtopia sebagai berikut.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | BASE | | | | | |
| 1 | | | | | | | | | | | |
| 2 | | | | | | | | | 1 | | |
| 3 | | | | | | | | | | | |
| 4 | 2 | | | | | | | | | | |
| 5 | | | | | 3 | | | | | | |
| 6 | | | | | | | | | | | |
| 7 | | | | 4 | | | | | | | |
| 8 | | | | | | | | | | | |
| 9 | | | | | | | | | | | |
| 10 | | | | | | | | | | | 5 |

Fig. 3. Skenario penambangan *item* pada permainan Growtopia (*Sumber: dokumen pribadi*)

Pada awal permainan pemain berada pada posisi titik awal (*base*). Pemain hanya bisa bergerak dalam arah horizontal atau vertikal, yang setiap pergerakannya dapat berjalan satu langkah ke arah *East, South, West,* dan *North.*

### B. *Jarak antar Simpul*

Jarak antar simpul dapat dihitung menggunakan pendekatan *Manhattan Distance*. *Manhattan Distance* merupakan metode pengukuran jarak antara dua titik dalam ruang *grid*, seperti pada kasus permainan Growtopia yang dimodelkan dalam petak-petak. *Manhattan Distance* digunakan dalam perhitungan jarak antar simpul karena sesuai dengan pergerakan pemain yang dibatasi pada sumbu horizontal dan vertikal. Dalam hal ini, *Manhattan Distance* dihitung dari banyaknya petak secara horizontal dan vertikal terdekat untuk menuju posisi petak yang sesuai.

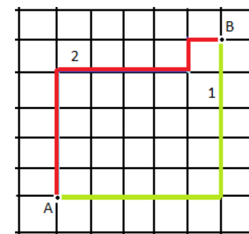



Fig. 4. Ilustrasi Jarak Menggunakan *Manhattan Distance* (*Sumber:* https://www.geeksforgeeks.org/count-paths-with-distance-equal-to-manhattan-distance/)

Formula untuk menghitung *Manhattan Distance* adalah sebagai berikut.

$$\textit{Manhattan Distance} = |x_1 - x_2| + |y_1 - y_2|$$

Dengan $x_1$ dan $y_1$ adalah koordinat titik awal, sementara $x_2$ dan $y_2$ adalah koordinat titik tujuan.

Implementasi perhitungan *Manhattan Distance* menggunakan bahasa *python* dari skenario pada gambar 3, diperoleh jarak antara simpul (*item*) sebagai berikut.

**Algoritma:**

```
def manhattan_distance(point1, point2):
    return abs(point1[0] - point2[0])+
           abs(point1[1] - point2[1])

def calculate_distance_matrix(points):
    n = len(points)
    distance_matrix = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            distance_matrix[i][j] =
                  manhattan_distance(points[i],
                                     points[j])
    return distance_matrix

# instansiasi titik tiap simpul (item)
points = [(5, 0), (8, 2), (0, 4),
          (4, 5), (3, 7), (10, 10)]
distance_matrix = calculate_distance_matrix(points)
```

**Hasil:**

| No | BASE | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| BASE | 0 | 5 | 9 | 6 | 9 | 15 |
| 1 | 5 | 0 | 10 | 7 | 10 | 10 |
| 2 | 9 | 10 | 0 | 5 | 6 | 16 |
| 3 | 6 | 7 | 5 | 0 | 3 | 11 |
| 4 | 9 | 10 | 6 | 3 | 0 | 10 |
| 5 | 15 | 10 | 16 | 11 | 10 | 0 |

Fig. 5. Perhitungan *Manhattan Distance* pada Persoalan (*Sumber: dokumen pribadi*)

### C. *Pencarian Rute Penambangan dengan Reduced Cost Matrix*

#### 1. *Node*

Pencarian rute penambangan dilakukan pada pohon ruang status. Node (*simpul*) dibentuk dengan menyimpan informasi level (*depth*) yang menunjukkan kedalaman dari simpul pada pohon ruang status, *path* yang menunjukkan jalur yang dilewati untuk menuju simpul ke-*i*, *cost* (biaya) sebagai *bound* untuk simpul, dan *reduced matrix* yang menyimpan matriks ongkos tereduksi yang dihasilkan. Objek *Node* dibandingkan berdasarkan atribut *cost*-nya, yang mana akan mengembalikan *True* jika objek saat ini memiliki biaya yang lebih rendah daripada objek lainnya, dan *False* jika sebaliknya.

```
class Node:
    def __init__(self, level, path, cost,
                 reduced_matrix):
        self.level = level
        self.path = path
        self.cost = cost
        self.reduced_matrix = reduced_matrix

    def __lt__(self, other):
        return self.cost < other.cost
```

#### 2. *Reduced Cost Matrix*

Pada perhitungan sebelumnya telah diperoleh jarak antar simpul. Hal tersebut dapat digunakan sebagai matriks ketetanggan untuk membantu perhitungan *cost* di setiap simpul pada pohon ruang status. Implementasi perhitungan untuk menghasilkan matriks ongkos-tereduksi dilakukan dengan mengurangkan setiap baris kemudian kolom dari matriks menggunakan metode reduksi. Jumlah semua pengurang pada proses memperoleh matriks tereduksi juga dihitung. Berikut implementasi dari *function reduce_matrix* yang dibuat dalam bahasa *python.*

```
def reduce_matrix(matrix):
    row_reduction = np.min(matrix, axis=1)
    row_reduction[row_reduction == np.inf] = 0
    reduced_matrix = matrix -
                     row_reduction[:, np.newaxis]

    col_reduction = np.min(reduced_matrix, axis=0)
    col_reduction[col_reduction == np.inf] = 0
    reduced_matrix = reduced_matrix - col_reduction

    cost = np.sum(row_reduction) +
           np.sum(col_reduction)
    return reduced_matrix, cost
```

Fungsi *calculate_cost(matrix, path)* menghitung total biaya perjalanan berdasarkan jalur yang diberikan. Fungsi ini menjumlahkan biaya perjalanan antara setiap pasangan kota berturut-turut dalam jalur tersebut dengan menjumlahkan elemen-elemen yang sesuai dari *cost* dari matriks. Iterasi dilakukan dari awal hingga satu elemen sebelum akhir jalur untuk memastikan semua segmen perjalanan dihitung, yang kemudian dikembalikan sebagai hasil total biaya perjalanan.

```
def calculate_cost(matrix, path):
    cost = 0
    for i in range(len(path) - 1):
        cost += matrix[path[i], path[i+1]]
    return cost
```

#### 3. Algoritma *Branch and Bound*

Fungsi *branch_and_bound(matrix, start)* adalah implementasi algoritma *Branch and Bound* untuk menyelesaikan masalah Traveling Salesman Problem (TSP). Algoritma ini menggunakan struktur data *priority queue* untuk mengelola simpul yang perlu dieksplorasi, di mana setiap simpul mewakili suatu tahap atau jalur parsial dalam pencarian jalur optimal.

Algoritma beroperasi selama *priority queue* tidak kosong. Pada setiap iterasi, simpul dengan biaya terendah diambil dari antrian. Jika simpul ini merupakan simpul daun (semua kota telah dikunjungi), maka akan dihitung biaya total. Jika biaya saat ini lebih rendah dari biaya terbaik yang ditemukan sejauh ini, biaya dan jalur terbaik diperbarui. *Priority queue* juga diperbarui untuk hanya menyimpan simpul-simpul dengan biaya lebih rendah dari biaya terbaik saat ini (*prunning*).

Jika simpul yang diambil bukan simpul terakhir, algoritma mengeksplorasi simpul anak yang mungkin dengan

menambahkan setiap kota yang belum dikunjungi ke jalur saat ini. Matriks kemudian direduksi lagi dan *cost* dihitung. Jika biaya baru ini lebih rendah dari biaya terbaik, simpul anak yang baru dibuat dimasukkan ke dalam *priority queue*.

```
def branch_and_bound(matrix, start):
    n = len(matrix)
    initial_matrix, cost = reduce_matrix(np.copy(matrix))
    root = Node(0, [start], cost, initial_matrix)
    pq = [root]

    best_cost = np.inf
    best_path = None

    while pq:
        current = heapq.heappop(pq)

        print(f"Current node path: {current.path}")
        print(f"Reduced cost matrix:
             \n{current.reduced_matrix}")
        print(f"Cost: {current.cost}")
        input("Press Enter to continue...")

        if current.level == n - 1:
            final_cost = calculate_cost(matrix,
                           current.path + [start])
            if final_cost < best_cost:
                best_cost = final_cost
                best_path = current.path + [start]
            pq = [node for node in pq
                    if node.cost < best_cost]
            heapq.heapify(pq)
            continue

        for i in range(n):
            if i not in current.path:
                new_path = current.path + [i]
                new_matrix = np.copy(current.reduced_matrix)

                for j in range(n):
                    new_matrix[current.path[-1], j] = np.inf
                    new_matrix[j, i] = np.inf
                new_matrix[i, start] = np.inf

                reduced_matrix, add_cost = reduce_matrix(new_matrix)
                new_cost = current.cost +
                        current.reduced_matrix[current.path[-1], i] +
                        add_cost

                if new_cost < best_cost:
                    child_node = Node(current.level + 1, new_path,
                                   new_cost, reduced_matrix)
                    heapq.heappush(pq, child_node)

        for node in pq:
            print(f"\nExpanded node path: {node.path}")
            print(f"Cost: {node.cost}")
            input("Press Enter to continue...")

    return best_path, best_cost
```

## 4. Hasil Pohon Ruang Status

Tabel 1. Tabel Pohon Ruang Status Penelusuran Rute Penambangan

| Simpul Hidup | Simpul Ekspansi |
| --- | --- |
| Iterasi ke-1<br>Current node path: [0]<br>Reduced cost matrix:<br>[[inf 0. 2. 1. 4. 5.]<br>[ 0. inf 3. 2. 5. 0.]<br>[ 4. 5. inf 0. 1. 6.]<br>[ 3. 4. 0. inf 0. 3.]<br>[ 6. 7. 1. 0. inf 2.]<br>[ 5. 0. 4. 1. 0. inf]]<br>Cost: 38.0 | node path: [0, 2] - Cost: 40.0<br>node path: [0, 1] - Cost: 41.0<br>node path: [0, 3] - Cost: 41.0<br>node path: [0, 4] - Cost: 42.0<br>node path: [0, 5] - Cost: 43.0 |
| Iterasi ke-2<br>Current node path: [0, 2]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf inf 2. 5. 0.]<br>[inf 5. inf 0. 1. 6.]<br>[ 3. 4. inf inf 0. 3.]<br>[ 6. 7. inf 0. inf 2.]<br>[ 5. 0. inf 1. 0. inf]]<br>Cost: 40.0 | node path: [0, 3] - Cost: 41.0<br>node path: [0, 1] - Cost: 41.0<br>node path: [0, 2, 3] - Cost: 42.0<br>node path: [0, 4] - Cost: 42.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 2, 5] - Cost: 46.0 |
| Iterasi ke-3<br>Current node path: [0, 3]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf 3. inf 5. 0.]<br>[ 3. 4. inf inf 0. 5.]<br>[inf 4. 0. inf 0. 3.]<br>[ 5. 6. 0. inf inf 1.]<br>[ 5. 0. 4. inf 0. inf]]<br>Cost: 41.0 | node path: [0, 1] - Cost: 41.0<br>node path: [0, 4] - Cost: 42.0<br>node path: [0, 2, 3] - Cost: 42.0<br>node path: [0, 3, 2] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 3, 5] - Cost: 44.0 |
| Iterasi ke-4<br>Current node path: [0, 1]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[inf inf 3. 2. 5. 0.]<br>[ 1. inf inf 0. 1. 6.]<br>[ 0. inf 0. inf 0. 3.]<br>[ 3. inf 1. 0. inf 2.]<br>[ 2. inf 4. 1. 0. inf]]<br>Cost: 41.0 | node path: [0, 1, 5] - Cost: 41.0<br>node path: [0, 4] - Cost: 42.0<br>node path: [0, 2, 3] - Cost: 42.0<br>node path: [0, 3, 2] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 1, 2] - Cost: 46.0<br>node path: [0, 1, 3] - Cost: 46.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 2, 4] - Cost: 44.0 |
| Iterasi ke-5<br>Current node path: [0, 1, 5]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[ 1. inf inf 0. 1. inf]<br>[ 0. inf 0. inf 0. inf]<br>[ 3. inf 1. 0. inf inf]<br>[inf inf 4. 1. 0. inf]]<br>Cost: 41.0 | node path: [0, 1, 5, 4] - Cost: 41.0<br>node path: [0, 2, 3] - Cost: 42.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 4] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 3, 2] - Cost: 42.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 1, 2] - Cost: 46.0<br>node path: [0, 1, 3] - Cost: 46.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 1, 5, 2] - Cost: 45.0<br>node path: [0, 1, 5, 3] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0 |
| Iterasi ke-6<br>Current node path: [0, 1, 5, 4]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[ 1. inf inf 0. inf inf]<br>[ 0. inf 0. inf inf inf]<br>[inf inf 1. 0. inf inf]<br>[inf inf inf inf inf inf]]<br>Cost: 41.0 | node path: [0, 2, 3] - Cost: 42.0<br>node path: [0, 4] - Cost: 42.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 3, 2] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 1, 5, 4, 2] - Cost: 42.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 1, 2] - Cost: 46.0<br>node path: [0, 1, 3] - Cost: 46.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 1, 5, 2] - Cost: 45.0<br>node path: [0, 1, 5, 3] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 1, 5, 4, 3] - Cost: 42.0 |

| | |
|---|---|
| Iterasi ke-7<br>Current node path: [0, 2, 3]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf inf inf 5. 0.]<br>[inf inf inf inf inf inf]<br>[inf 4. inf inf 0. 3.]<br>[ 4. 5. inf inf inf 0.]<br>[ 5. 0. inf inf 0. inf]]<br>Cost: 42.0 | node path: [0, 4] - Cost: 42.0<br>node path: [0, 3, 2] - Cost: 42.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 1, 5, 4, 2] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 1, 5, 4, 3] - Cost: 42.0<br>node path: [0, 2, 3, 4] - Cost: 42.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 1, 2] - Cost: 46.0<br>node path: [0, 1, 3] - Cost: 46.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 1, 5, 2] - Cost: 45.0<br>node path: [0, 1, 5, 3] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 2, 3, 1] - Cost: 50.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 2, 3, 5] - Cost: 49.0 |
| Iterasi ke-8<br>Current node path: [0, 4]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf 3. 2. inf 0.]<br>[ 4. 5. inf 0. inf 6.]<br>[ 3. 4. 0. inf inf 3.]<br>[inf 7. 1. 0. inf 2.]<br>[ 5. 0. 4. 1. inf inf]]<br>Cost: 42.0 | node path: [0, 3, 2] - Cost: 42.0<br>node path: [0, 1, 5, 4, 2] - Cost: 42.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 2, 3, 4] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 1, 5, 4, 3] - Cost: 42.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 4, 2] - Cost: 46.0<br>node path: [0, 4, 5] - Cost: 44.0<br>node path: [0, 1, 3] - Cost: 46.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 1, 5, 2] - Cost: 45.0<br>node path: [0, 1, 5, 3] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 2, 3, 1] - Cost: 50.0<br>node path: [0, 2, 3, 5] - Cost: 49.0<br>node path: [0, 4, 1] - Cost: 53.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 4, 3] - Cost: 46.0<br>node path: [0, 1, 2] - Cost: 46.0 |
| Iterasi ke-9<br>Current node path: [0, 3, 2]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf inf inf 5. 0.]<br>[inf 4. inf inf 0. 5.]<br>[inf inf inf inf inf inf]<br>[ 4. 5. inf inf inf 0.]<br>[ 5. 0. inf inf 0. inf]]<br>Cost: 42.0 | node path: [0, 1, 5, 4, 2] - Cost: 42.0<br>node path: [0, 2, 3, 4] - Cost: 42.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 1, 5, 4, 3] - Cost: 42.0<br>node path: [0, 3, 2, 4] - Cost: 42.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 1, 2] - Cost: 46.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 4, 2] - Cost: 46.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 1, 3] - Cost: 46.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 1, 5, 2] - Cost: 45.0<br>node path: [0, 1, 5, 3] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 2, 3, 1] - Cost: 50.0<br>node path: [0, 2, 3, 5] - Cost: 49.0<br>node path: [0, 4, 1] - Cost: 53.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 4, 3] - Cost: 46.0<br>node path: [0, 3, 2, 1] - Cost: 50.0<br>node path: [0, 4, 5] - Cost: 44.0<br>node path: [0, 3, 2, 5] - Cost: 51.0 |
| Iterasi ke-10<br>Current node path: [0, 1, 5, 4, 2]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf 0. inf inf]<br>[ 0. inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]]<br>Cost: 42.0 | node path: [0, 2, 3, 4] - Cost: 42.0<br>node path: [0, 3, 2, 4] - Cost: 42.0<br>node path: [0, 1, 5, 4, 2, 3] - Cost: 42.0<br>node path: [0, 1, 5, 4, 3] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 1, 2] - Cost: 46.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 4, 2] - Cost: 46.0<br>node path: [0, 4, 5] - Cost: 44.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 1, 5, 2] - Cost: 45.0<br>node path: [0, 1, 5, 3] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 2, 3, 1] - Cost: 50.0<br>node path: [0, 2, 3, 5] - Cost: 49.0<br>node path: [0, 4, 1] - Cost: 53.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 4, 3] - Cost: 46.0<br>node path: [0, 3, 2, 1] - Cost: 50.0<br>node path: [0, 3, 2, 5] - Cost: 51.0<br>node path: [0, 1, 3] - Cost: 46.0 |

| | |
|---|---|
| Iterasi ke-11<br>Current node path: [0, 2, 3, 4]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf inf inf inf 0.]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf 5. inf inf inf 0.]<br>[ 5. 0. inf inf inf inf]]<br>Cost: 42.0 | node path: [0, 1, 5, 4, 2, 3] - Cost: 42.0<br>node path: [0, 3, 2, 4] - Cost: 42.0<br>node path: [0, 2, 3, 4, 5] - Cost: 42.0<br>node path: [0, 1, 5, 4, 3] - Cost: 42.0<br>node path: [0, 3, 4] - Cost: 44.0<br>node path: [0, 5] - Cost: 43.0<br>node path: [0, 2, 4] - Cost: 44.0<br>node path: [0, 1, 2] - Cost: 46.0<br>node path: [0, 2, 5] - Cost: 46.0<br>node path: [0, 4, 2] - Cost: 46.0<br>node path: [0, 4, 5] - Cost: 44.0<br>node path: [0, 3, 5] - Cost: 44.0<br>node path: [0, 1, 4] - Cost: 49.0<br>node path: [0, 1, 5, 2] - Cost: 45.0<br>node path: [0, 1, 5, 3] - Cost: 44.0<br>node path: [0, 3, 1] - Cost: 48.0<br>node path: [0, 2, 3, 1] - Cost: 50.0<br>node path: [0, 2, 3, 5] - Cost: 49.0<br>node path: [0, 4, 1] - Cost: 53.0<br>node path: [0, 2, 1] - Cost: 48.0<br>node path: [0, 4, 3] - Cost: 46.0<br>node path: [0, 3, 2, 1] - Cost: 50.0<br>node path: [0, 3, 2, 5] - Cost: 51.0<br>node path: [0, 2, 3, 4, 1] - Cost: 52.0<br>node path: [0, 1, 3] - Cost: 46.0 |
| Iterasi ke-12<br>Current node path: [0, 1, 5, 4, 2, 3]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]]<br>Cost: 42.0 | *Goal*<br>(melakukan *prunning*)<br>node path: [0, 1, 5, 4, 3] - Cost: 42.0<br>node path: [0, 3, 2, 4] - Cost: 42.0<br>node path: [0, 2, 3, 4, 5] - Cost: 42.0 |
| Iterasi ke-13<br>Current node path: [0, 1, 5, 4, 3]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[ 0. inf inf inf inf inf]<br>[inf inf 0. inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]]<br>Cost: 42.0 | node path: [0, 3, 2, 4] - Cost: 42.0<br>node path: [0, 2, 3, 4, 5] - Cost: 42.0 |
| Iterasi ke-14<br>Current node path: [0, 3, 2, 4]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf inf inf inf 0.]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf 5. inf inf inf 0.]<br>[ 5. 0. inf inf inf inf]]<br>Cost: 42.0 | node path: [0, 2, 3, 4, 5] - Cost: 42.0 |
| Iterasi ke-15<br>Current node path: [0, 2, 3, 4, 5]<br>Reduced cost matrix:<br>[[inf inf inf inf inf inf]<br>[ 0. inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf inf inf inf inf inf]<br>[inf 0. inf inf inf inf]]<br>Cost: 42.0 | *Empty*<br>(sudah tidak ada simpul dengan cost ≤ 42.0) |
| Best path: [0, 1, 5, 4, 2, 3, 0]<br>Best cost: 42.0 | |

Dari hasil yang diperoleh, implementasi algoritma *Branch and Bound* dengan *Reduced Cost Matrix* untuk masalah pencarian rute penambangan dalam permainan Growtopia memberikan solusi optimal dengan mengurangi jumlah simpul yang perlu dieksplorasi dan pemangkasan simpul yang tidak menjanjikan. Algoritma *Branch and Bound* umumnya memiliki kompleksitas waktu eksponensial dalam kasus terburuk, karena semua kemungkinan jalur harus dieksplorasi dalam kasus terburuk. Namun, teknik reduksi matriks dan pemangkasan

simpul dengan biaya lebih tinggi dari biaya terbaik saat ini membantu mengurangi jumlah simpul yang perlu dieksplorasi.

Dari hasil pohon ruang status untuk pencarian rute penambangan dengan *reduced cost matrix*, terdapat sebanyak 15 iterasi untuk mengunjungi simpul hidup. Dari iterasi ke-1 hingga iterasi ke-11, simpul ekspansi tampak selalu bertambah. Hal ini karena tidak ada simpul yang dapat dipangkas, artinya tidak ada simpul dengan nilai batas yang lebih besar dari nilai batas solusi terbaik saat ini. Pada kondisi tersebut kebetulan belum ada simpul yang mencapai daun (solusi). *Prunning* mulai dilakukan pada iterasi ke-12, di mana pada tahap ini status pohon sudah mencapai daun (solusi). Ketika status pohon pencarian sudah mencapai solusi, jika terdapat nilai *cost* di *priority queue* yang memiliki nilai batas simpul lebih besar daripada nilai batas solusi terbaik sejauh ini (*lower bound*), maka akan dipangkas, dihapus dari dari *priority queue*. Hal ini karena simpul tersebut telah melebihi batas *lower bound* dan membuat simpul menjadi tidak *promising* lagi (tidak menuju ke solusi yang optimal). Kemudian pada iterasi ke-13, 14, dan 15 simpul yang dikunjungi sudah tidak lagi menghasilkan simpul ekspansi dengan nilai batas yang lebih baik dari nilai batas solusi terbaik sejauh ini sehingga tidak ada simpul ekspansi yang dihasilkan. Proses berhenti pada iterasi ke-15 di mana *priority queue* telah kosong. Hasil dari pencarian rute ini diperoleh rute penambangan yang optimal dengan jalur 0-1-5-4-2-3-0 dengan total *cost* sebesar 42. Simpul 0 dalam hal ini adalah *base.*

Kompleksitas kasus terburuk dari algoritma *Branch and Bound* sama dengan kompleksitas algoritma *Brute Force*. Hal ini karena dalam kasus terburuk, mungkin saja tidak akan pernah mendapatkan kesempatan untuk memangkas simpul mana pun. Namun, dalam praktiknya, algoritma *Branch and Bound* bekerja sangat baik tergantung pada berbagai instansiasi dari persoalan *Traveling Salesman Problem* (TSP). Kompleksitas juga bergantung pada pemilihan fungsi batas (*bounding function*) karena fungsi ini yang menentukan berapa banyak simpul yang nantinya akan bisa dipangkas. Dengan kata lain, efektivitas algoritma *Branch and Bound* sangat dipengaruhi oleh seberapa baik fungsi batas yang digunakan dapat memperkirakan biaya minimum. Dengan begitu, semakin baik fungsi batasnya, semakin banyak simpul yang bisa dipangkas, dan semakin efisien algoritma tersebut dalam menyelesaikan masalah TSP.

Dari segi kompleksitas memori, memori yang digunakan untuk menyimpan matriks biaya tereduksi dan *priority queue* dapat menjadi sangat besar seiring dengan bertambahnya jumlah simpul, mengakibatkan penggunaan memori yang intensif. Akan tetapi, kembali lagi kepada poin sebelumnya, bahwa efektivitas bergantung pada pemangkasan (prunning) terhadap simpul-simpul yang tidak mencapai tujuan maupun rute yang optimal. Pemangkasan sangat bergantung pada kualitas heuristik yang digunakan untuk memperkirakan batas bawah (*lower bound*) biaya. Jika heuristik tidak akurat, jumlah simpul yang perlu dieksplorasi bisa meningkat signifikan.

### 5. *Branch and Bound vs Greedy*

Alternatif lain untuk pencarian yang cepat dari rute penambangan ini adalah dengan menggunakan algoritma *Greedy*. Algoritma *Greedy* melakukan pendekatan yang mengambil keputusan bertahap dengan memilih pilihan terbaik yang tersedia pada setiap langkahnya tanpa mempertimbangkan dampak jangka panjang dari keputusan tersebut. Dalam konteks pencarian rute penambangan, algoritma *Greedy* akan memilih *item* terdekat yang belum diambil pada setiap langkah, dan terus melakukannya hingga semua *item* diambil, kemudian kembali ke titik awal (*base*). Keunggulan utama dari pendekatan *Greedy* adalah kesederhanaan dan kecepatan eksekusinya, karena tidak memerlukan pencarian mendalam atau penyimpanan data yang kompleks. Namun, kelemahan signifikan dari metode ini adalah kecenderungannya untuk menghasilkan solusi yang terjebak pada *local minima/plateu*, di mana keputusan lokal terbaik tidak selalu mengarah pada solusi global terbaik.

```
def tsp_greedy(matrix):
    n = len(matrix)
    visited = [False] * n
    tour = []

    current_node = 0
    visited[current_node] = True
    tour.append(current_node)

    cost = 0
    for _ in range(n - 1):
        nearest_node = None
        nearest_distance = float('inf')
        for next_node in range(n):
            if (not visited[next_node] and
               matrix[current_node][next_node] < nearest_distance):
               nearest_distance = matrix[current_node][next_node]
               nearest_node = next_node
               cost += nearest_distance
        current_node = nearest_node
        visited[current_node] = True
        tour.append(current_node)
        print(f"path: {tour} - cost: {nearest_distance}")
        input()
    tour.append(tour[0])

    return tour, cost
```

Sebagai contoh pada persoalan instansiasi sebelumnya, dengan algoritma *Greedy* diperoleh rute penambangan 0-1-3-4-2-5-0 dengan total *cost* 52, yang mana rute tersebut bukanlah solusi global terbaik. Dengan demikian, meskipun algoritma Greedy dapat memberikan solusi cepat dan mudah diimplementasikan, solusi yang dihasilkan sering kali tidak seoptimal solusi yang ditemukan menggunakan algoritma yang lebih canggih seperti algoritma *Branch and Bound*.

```
path: [0, 1] - cost: 5

path: [0, 1, 3] - cost: 22

path: [0, 1, 3, 4] - cost: 30

path: [0, 1, 3, 4, 2] - cost: 36

path: [0, 1, 3, 4, 2, 5] - cost: 52

Rute yang diambil: [0, 1, 3, 4, 2, 5, 0]
Cost: 52
```

Fig. 6. Hasil Rute Penambangan Algoritma *Greedy* (*Sumber: dokumen pribadi*)

## IV. PENUTUP

### A. Kesimpulan

penerapan algoritma Branch and Bound dengan *reduced cost matrix* dalam pencarian rute penambangan pada permainan Growtopia dapat menghasilkan solusi optimal dengan mengurangi jumlah simpul yang perlu dieksplorasi serta memangkas simpul yang tidak menjanjikan. Hasil eksperimen menunjukkan bahwa algoritma ini mampu menemukan rute penambangan yang optimal dengan jalur 0-1-5-4-2-3-0 dan total *cost* sebesar 42, dibandingkan dengan algoritma *Greedy* yang menghasilkan jalur sub-optimal dengan biaya lebih tinggi. Dalam konteks permainan Growtopia, algoritma *Branch and Bound* ini mampu mengidentifikasi rute penambangan yang meminimalkan jarak perjalanan, memungkinkan pemain untuk mengumpulkan *item* dengan lebih efisien dan kembali ke titik awal (*base*) dengan biaya minimum.

Penerapan fungsi batas yang efektif dapat secara signifikan meningkatkan kinerja algoritma *Branch and Bound*. Pemilihan fungsi batas yang baik sangat penting karena berperan dalam menentukan berapa banyak simpul yang dapat dipangkas selama proses pencarian solusi. Fungsi batas yang lebih akurat akan menghasilkan lebih banyak pemangkasan simpul, sehingga mengurangi jumlah simpul yang perlu dieksplorasi dan meningkatkan efisiensi algoritma secara keseluruhan dalam menyelesaikan masalah *Traveling Salesman Problem* (TSP).

### B. Saran

Efektivitas algoritma *Branch and* Bound sangat bergantung pada fungsi batas yang digunakan. Oleh karena itu, perlu adanya eksplorasi untuk meningkatkan algoritma *Branch and Bound* dengan mengembangkan dan mengimplementasikan fungsi batas yang lebih efektif dan akurat. Selain itu, optimasi penggunaan memori juga perlu diperhatikan mengingat bahwa kompleksitas memori adalah salah satu tantangan utama dalam algoritma ini saat ukuran ruang meningkat secara signifikan.

## V. UCAPAN TERIMA KASIH

Ucapan rasa syukur dan terima kasih yang sebesar-besarnya hendak saya sampaikan kepada Tuhan Yang Maha Esa karena atas rahmat dan karunia-Nya lah saya diberikan kesehatan sehingga dapat menyelesaikan makalah ini tepat waktu. Ucapan terima kasih juga tak lupa saya sampaikan kepada keluarga saya yang telah mendukung secara penuh dalam proses pendidikan dan perkuliahan yang saya jalani hingga sekarang. Ucapan terima kasih saya ucapkan kepada dosen-dosen pengampu mata kuliah IF2211 Strategi Algoritma, Bapak Rinaldi Munir, Ibu Nur Ulfa Maulidevi, dan Bapak Rila Mandala serta Bapak Monterico Adrian, yang telah memberikan dedikasi dan pengajaran selama semester 4 ini. Akhir kata, saya juga ingin menyampaikan rasa terima kasih kepada seluruh pihak yang telah membantu dan mendukung selesainya makalah ini.

## VI. LAMPIRAN

- Video penjelasan https://youtu.be/7WZYW59Cx90?si=1xN4JdiVbKaoJ-fT
- *Code* percobaan: https://github.com/nanthedom/Makalah-Stima-2024

## REFERENCES

[1] Munir, Rinaldi. 2021. Algoritma Branch & Bound (Bagian 1). Diakses melalui https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branch-and-Bound-2021-Bagian1.pdf pada 11 Juni 2024.

[2] Munir, Rinaldi. 2021. Algoritma Branch & Bound (Bagian 2). Diakses melalui https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branchand-Bound-2021-Bagian2.pdf pada 11 Juni 2024.

[3] Munir, Rinaldi. 2021. Algoritma Branch & Bound (Bagian 3). Diakses melalui https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branchand-Bound-2021-Bagian3.pdf pada 11 Juni 2024.

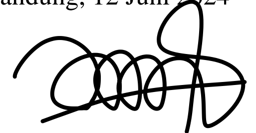
[4] Levitin, A. 2012. Introduction to The Design and Analysis of Algorithms (Third Edition). Edinburgh: Pearson.

[5] Python Software Foundation. 2023. Python 3.12.4 Documentation. Diakses melalui https://docs.python.org/3 pada 11 Juni 2024.

## PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024



Naufal Adnan
13522116