Penggunaan Graf Berbobot untuk Mencari Rute Terpendek Memasuki Site dalam Permainan Valorant

Marchotridyo - 13520119[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520119@std.stei.itb.ac.id

***Abstract*—Valorant adalah suatu permainan bertipe *tactical shooter* yaitu cabang permainan tembak-menembak yang lebih mengedepankan kemampuan berpikir dibandingkan kemampuan mekanikal. Salah satu dari aspek strategi yang lekat pada permainan Valorant adalah memilih rute yang dipilih untuk memasuki sebuah *site*, yaitu tempat tim penyerang menanam bom atau tempat tim bertahan menjinakkan bom. Dikarenakan satu ronde Valorant dibatasi oleh batas waktu yang cukup sempit, pemain harus bisa menentukan rute yang paling optimal. Menggunakan teori Graf, khususnya graf berarah dan algoritma Dijkstra, makalah ini bertujuan untuk mempermudah pemain untuk mengoptimasi permainannya ketika bermain Valorant. Implementasi dari algoritma Dijkstra ini dilakukan dalam bahasa pemrograman Java.**

***Keywords*—Dijkstra, graf berbobot, Java, rute optimal, strategi, Valorant**

I. PENDAHULUAN

Valorant adalah suatu permainan ber-*genre first-person shooter* yang diterbitkan oleh Riot Games pada tahun 2020. Permainan ini dapat dikelompokkan dalam permainan *tactical shooter*, yaitu permainan yang tembak-menembak yang didesain untuk lebih mengedepankan taktik (kemampuan berpikir, misal menyusun strategi atau rute penyerangan) dibandingkan refleks (kemampuan mekanikal, seperti kemampuan menggerakkan *mouse* secara akurat ke lawan). Permainan ini diinspirasi oleh permainan seri *Counter Strike* yang sudah berkembang sejak tahun 1999. Seperti permainan *Counter Strike*, objektif utama dalam permainan Valorant adalah untuk mengaktifkan bom dan/atau mengalahkan tim bertahan (apabila berperan sebagai tim menyerang) atau menjinakkan bom dan/atau mengalahkan tim penyerang (apabila berperan sebagai tim bertahan).

Di permainan Valorant, pemain dapat memilih beberapa karakter untuk dimainkan yang disebut sebagai *agent*. Masing-masing *agent* memiliki kelebihannya masing-masing: ada yang bisa bergerak lebih cepat, ada yang bisa menciptakan asap untuk mengganggu pandangan lawan, ada yang bisa teleportasi, dan sebagainya. Ini menyebabkan taktik berperan cukup penting karena selain pemain harus memiliki kemampuan mekanikal yang baik, pemain juga harus bisa memaksimalkan kelebihan *agent* yang dipilihnya dalam permainannya.

Permainan standar di Valorant adalah permainan bertipe *best of 25*, yaitu tim yang memenangkan 13 ronde pertama adalah tim yang menang. Suatu tim terdiri atas 5 orang, dan satu permainan dapat dimainkan oleh 2 tim. Di awal permainan, satu tim akan dipasang sebagai tim penyerang dan tim satunya akan dipasang sebagai tim bertahan. Ronde berakhir jika waktu ronde habis (akan dijelaskan di paragraf beriktunya), semua pemain di suatu tim berhasil dieliminasi, bom berhasil diledakkan, atau bom berhasil dijinakkan.

Salah satu aspek taktikal yang penting dari setiap ronde Valorant adalah bagaimana suatu tim memaksimalkan waktu yang diberikan padanya di setiap ronde. Suatu ronde memiliki batas waktu awal 100 detik, artinya tim penyerang harus bisa menanam bom atau mengeliminasi tim bertahan sebelum waktu 100 detik tersebut habis. Apabila tim penyerang berhasil menanam bom, batas waktu ronde akan diubah menjadi 45 detik. Dalam 45 detik ini, tim bertahan harus bisa menjinakkan bom. Waktu untuk menjinakkan bom sendiri membutuhkan waktu 7 detik, jadi tim bertahan harus mulai menjinakkan bom paling lambat 38 detik setelah tim penyerang berhasil menanam bom.

Salah satu bentuk memaksimalkan waktu adalah mengatur rute apa yang akan diambil untuk memasuki *site*, yaitu tempat tim penyerang menanam bom atau tempat tim bertahan menjinakkan bom. Pengambilan rute yang terlalu panjang akan memakan waktu yang lama dan dapat berakibat pada kekalahan di ronde tersebut. Gambar di bawah ini mengilustrasikan dua buah rute umum yang sering digunakan untuk memasuki *site* pada *map* Ascent di Valorant (ada beberapa *map* yang bisa dimainkan di game Valorant, salah satunya adalah Ascent).

Gambar 1. Dua rute yang bisa diambil untuk memasuki site pada map Ascent

(Sumber: Dokumen penulis)

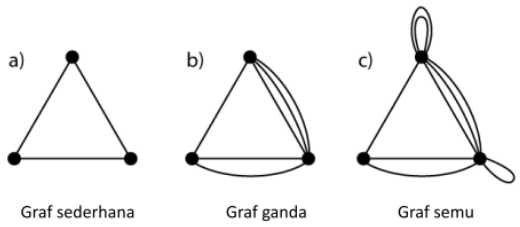
Makalah ini ditujukan untuk menyelesaikan suatu masalah yang ada dalam permainan Valorant yaitu menjawab pertanyaan "apa rute yang harus diambil oleh pemain agar bisa sampai ke *site* dalam waktu yang sesingkat-singkatnya?". Metode yang digunakan untuk menyelesaikannya adalah melalui mengonversi suatu *map* Valorant menjadi suatu graf berbobot dan menganalisis rute terpendek yang dibutuhkan untuk pergi dari simpul A ke simpul B menggunakan algoritma Dijkstra.

II. TEORI DASAR

A. Graf secara umum

Sebuah graf G didefinisikan sebagai $G = (V, E)$ dengan V adalah himpunan simpul yang tidak kosong dan E adalah himpunan sisi yang boleh kosong. Suatu sisi pada E menghubungkan dua buah simpul pada V. Contohnya, sisi $(1,2)$ menghubungkan simpul 1 dengan simpul 2.

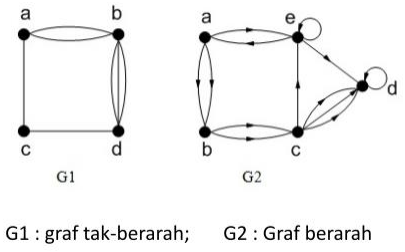
Suatu graf G disebut sebagai graf sederhana apabila tidak ada dua simpul yang dihubungkan oleh lebih dari satu sisi. Graf yang memiliki setidaknya dua simpul yang dihubungkan oleh lebih dari satu sisi dinamakan sebagai graf ganda. Dalam graf yang tidak sederhana, suatu simpul juga bisa memiliki sisi yang menghubungkan simpul tersebut dengan dirinya sendiri. Sisi ini disebut sebagai kalang. Apabila suatu graf memiliki kalang, graf tersebut disebut sebagai graf semu.



Gambar 2. Graf sederhana, graf ganda, dan graf semu.

(Sumber: *Slide* kuliah graf bagian 1, halaman 12)

Graf bisa dikelompokkan menjadi graf tak berarah dan graf berarah tergantung dengan tipe sisi yang digunakan. Pada graf tak berarah, sisi (a, b) menyatakan bahwa simpul a dihubungkan dengan b dan sebaliknya. Sedangkan, pada graf berarah, sisi (a, b) menyatakan bahwa simpul b dapat dijangkau dari simpul a namun tidak berlaku sebaliknya. Sisi ini digambarkan dengan panah, misalkan (a, b) digambarkan oleh sisi dengan penunjuk mengarah ke b, seperti $a \rightarrow b$.



Gambar 3. Graf tak-berarah dan graf berarah.

(Sumber: *Slide* kuliah graf bagian 1, halaman 14)

Graf digunakan sebagai model untuk menggambarkan beragam hal. Graf bisa menggambarkan jaringan pertemanan, rute perjalanan suatu maskapai, peta mudik dari Jawa Barat ke Jawa Timur, jaringan makanan pada suatu ekosistem, suatu kompetisi dari awal pertandingan sampai ada pemenang, dan sebagainya. Dalam makalah ini, graf digunakan untuk menggambarkan denah suatu *map* dalam permainan Valorant.

B. Terminologi-terminologi graf

Ada beberapa terminologi yang sering digunakan ketika sedang membahas graf. Terminologi-terminologi yang dimaksud sebagai berikut:

1. Dua simpul disebut *bertetangga* apabila ada sisi yang menghubungkannya. Secara grafik, ada setidaknya satu garis yang menghubungkan kedua simpul tersebut.
2. Suatu sisi $e = (a, b)$ dikatakan *bersisian* dengan simpul a dan simpul b. Secara grafik, sisi e ini merupakan garis yang menghubungkan a dengan b.
3. *Simpul terpencil* adalah simpul yang tidak memiliki sisi yang bersisian dengannya. Secara grafik, simpul ini hanya merupakan titik saja.
4. *Graf kosong* adalah graf dengan $E = \emptyset$, yaitu sisinya merupakan himpunan kosong.
5. *Derajat* dari suatu sisi v, dinotasikan $d(v)$, adalah banyak sisi yang bersisian dengan simpul tersebut.
6. Apabila simpul a dan b dihubungkan dan dapat disusun barisan simpul-simpul dan sisi-sisi yang dilaluinya, misal $a, e_1, v_1, e_2, v_2, \dots, e_n, b$ dengan e sisi dan v simpul, barisan tersebut disebut sebagai *lintasan*. Panjang dari lintasan adalah banyaknya sisi yang ada pada barisan tersebut.
7. Lintasan yang dimulai dan diakhiri pada simpul yang sama disebut sebagai *sirkuit* atau *siklus*.
8. Simpul a dan b disebut *terhubung* apabila ada lintasan dari a ke b. Apabila semua pasang simpul pada suatu graf G terhubung, graf G dikatakan sebagai *graf terhubung*. Apabila kondisi ini tidak dipenuhi, graf G dikatakan sebagai *graf tak terhubung*.
9. Apabila didefinisikan suatu graf $G = (V, E)$ dan terdapat graf $G_1 = (V_1, E_1)$ dengan $V_1 \subseteq V$ dan $E_1 \subseteq E$, G_1 disebut sebagai *upagraf* dari G.
10. *Cut-set* adalah himpunan sisi-sisi yang apabila dihapus dari graf G akan membagi graf G menjadi tidak terhubung, lebih spesifiknya membagi graf G menjadi dua buah komponen.
11. *Graf berbobot* adalah graf yang setiap sisinya memiliki suatu bobot/harga.

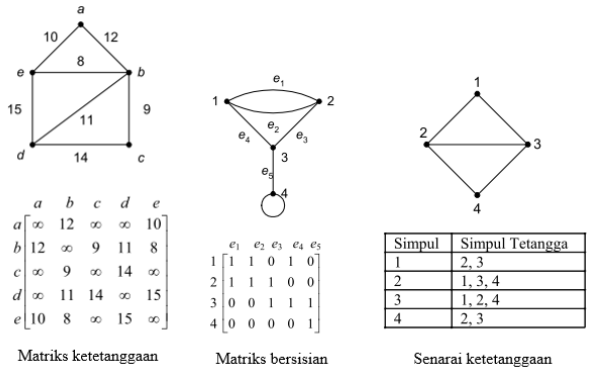
C. Representasi graf

Graf tidak hanya bisa direpresentasikan sebagai suatu gambar yang berisikan simpul-simpul dan sisi-sisi. Khususnya, dalam konteks *programming*, graf harus diimplementasikan dalam bentuk lain. Representasi-representasi graf yang dapat diimplementasikan untuk memecahkan masalah graf menggunakan *program* sebagai berikut:

1. *Matriks ketetanggaan*. Masing-masing baris dan kolom disusun berdasarkan simpul dan elemen a_{ij} menyatakan keterhubungan simpul i dan simpul j. Nilai 1 pada a_{ij} menyatakan kedua simpul tersebut terhubung sedangkan nilai 0 menyatakan sebaliknya.

2. *Matriks bersisian*. Baris ke-i menyatakan simpul i dan kolom ke$-j$ menyatakan sisi j. Elemen a_{ij} menyatakan hubungan bersisian antara simpul i dan sisi j. Nilai 1 pada a_{ij} menyatakan simpul i dan sisi j bersisian sedangkan nilai 0 menyatakan sebaliknya.

3. *Senarai ketetanggan*. Berbeda dengan representasi sebelumnya, representasi ini menggunakan *list*, bukan matriks. Representasi ini dilakukan dengan cara mencatat simpul-simpul tetangga yang dimiliki oleh sebuah simpul. Sebagai contoh, apabila simpul 1 berhubungan dengan simpul 2 dan 3, akan dicatat pada tabel bahwa simpul 1 memiliki tetangga berupa simpul 2 dan 3.



Simpul	Simpul Tetangga
1	2, 3
2	1, 3, 4
3	1, 2, 4
4	2, 3

Gambar 4. Macam-macam representasi graf.

(Sumber: *Slide* kuliah graf bagian 1, halaman 62 s.d. 64)

D. Graf berbobot dan algoritma Dijkstra

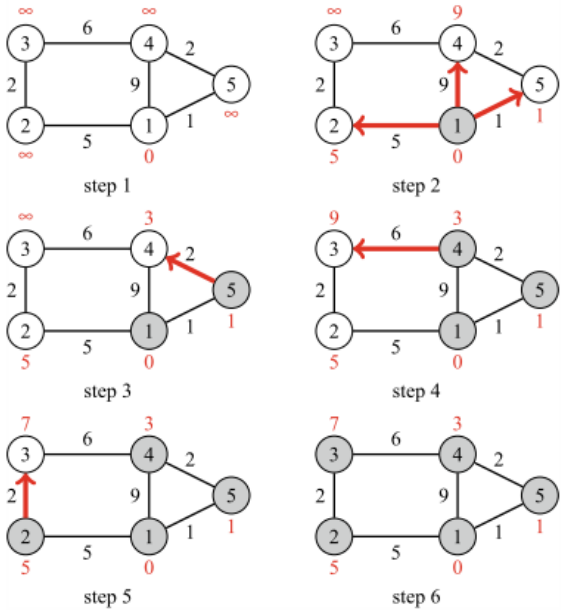
Seperti yang telah dijelaskan pada bagian B., graf berbobot adalah graf yang setiap sisinya memiliki suatu bobot/harga. Bobot ini dapat memiliki arti yang berbeda-beda sesuai dengan pembuat graf. Misalnya, bobot dapat melambangkan harga yang dibutuhkan untuk pergi dari simpul A ke simpul B. Dalam makalah ini, bobot yang digunakan melambangkan jarak dari simpul A ke simpul B, diukur dalam inci.

Penggunaan graf berbobot melahirkan masalah-masalah bertema *shortest path problem* untuk diselesaikan. Seperti namanya, masalah ini bertujuan untuk mencari rute terpendek yang dibutuhkan untuk pergi dari simpul A ke simpul B, tak peduli berapa simpul yang perlu disinggahi terlebih dahulu asalkan total bobot yang dilaluinya adalah bobot yang terkecil. Banyak algoritma yang dikembangkan untuk menyelesaikan permasalahan ini, salah satunya adalah *algoritma Dijkstra* yang ditemukan oleh Edsger Dijkstra pada tahun 1959.

Algoritma Dijkstra yang akan penulis gunakan menggunakan graf yang direpresentasikan dalam bentuk senarai ketetanggaan. Dalam realisasinya, untuk menyimpan data algoritma, akan dibentuk larik bernama *shortestPathTable* untuk mengetahui jarak terpendeknya dan larik *shortestPreviousStopoverTable* untuk mengetahui rute yang diperlukan untuk mencapainya. Kedua larik diimplementasikan menggunakan struktur data HashMap dalam Java. Langkah-langkah algoritma yang diterapkan adalah sebagai berikut:

1. Menandakan simpul awal sebagai simpul *current*.
2. Dari simpul *current*, cek bobot-bobot dari sisi yang bersisian dengan simpul *current*.
3. Apabila bobot pada langkah 2 lebih kecil dari bobot yang disimpan pada *shortestPathTable*, perbarui hasilnya dan perbarui entri simpul *current* pada larik *shortestPreviousStopoverTable* menjadi simpul yang dihubungkan oleh sisi berbobot yang dimaksud.
4. Pengulangan dilakukan dengan cara mengunjungi simpul terdekat yang belum dituju dari simpul *current* dan menjadikannya sebagai simpul *current*.
5. Algoritma selesai ketika semua simpul sudah pernah dikunjungi.

Berikut adalah gambar visualisasi mengenai algoritma Dijkstra yang diambil dari buku *Guide to Competitive Programming Learning and Improving Algorithms Through Contests*.



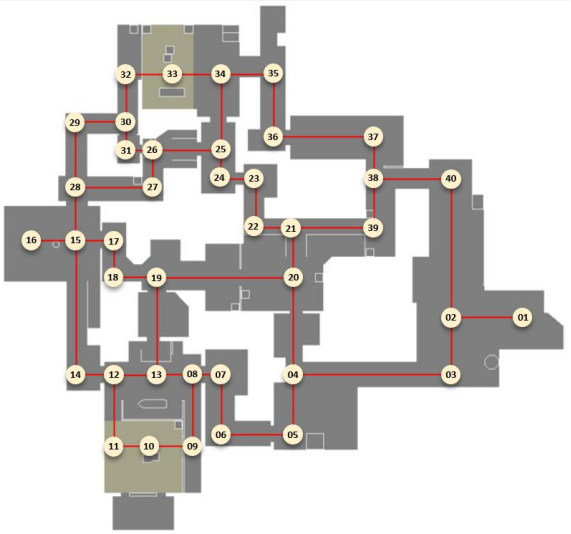
Gambar 5. Visualisasi cara kerja algoritma Dijkstra

Sumber: *Guide to Competitive Programming Learning and Improving Algorithms Through Contests*, halaman 96.

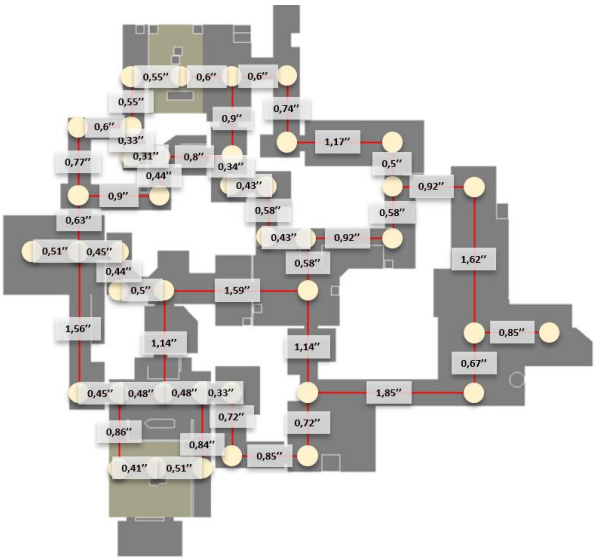
III. ANALISIS KASUS

A. Graf yang digunakan

Graf disusun dengan cara memberikan simpul-simpul pada *map* Ascent dan mengukur jaraknya berdasarkan panjang ruas garis yang dibentuk dalam aplikasi Microsoft PowerPoint (dalam satuan inci). Gambar 6 dan gambar 7 di bawah menunjukkan hasil graf yang dibuat untuk lalu ditransformasikan ke dalam bentuk kode.



Gambar 6. Graf yang menggambarkan *map* Ascent, tanpa bobot.



Gambar 7. Graf yang menggambarkan *map* Ascent, dengan bobot.

(Sumber: Dokumen penulis)

B. Implementasi graf dan algoritma Dijkstra dalam bahasa Java

Graf didefinisikan oleh vertex-vertex yang disimpan dalam suatu *class* bernama Vertex.

```java
class Vertex {
  int vertexID;
  HashMap<Vertex, Double> adjVertices;
  /* Konstruktor */
  Vertex(int vertexID) {
    this.vertexID = vertexID;
    adjVertices = new HashMap<Vertex, Double>();
  }
  /* Fungsi untuk menambahkan adjacent vertex */
  void addAdjacentVertex(Vertex vertex, double weight) {
    this.adjVertices.put(vertex, weight);
  }
}
```

Sebuah vertex didefinisikan sebagai sebuah objek yang memiliki ID (*vertexID*) dan senarai ketetanggaan (*adjVertices*) yang diimplementasikan menggunakan HashMap. Fungsi *addAdjacentVertex* digunakan untuk menambahkan simpul ke senarai ketetanggaan.

```java
HashMap<Integer, Double> shortestPathTable = new HashMap<Integer, Double>();
HashMap<Integer, Integer> shortestPreviousStopoverTable = new HashMap<Integer, Integer>();
```

Algoritma Dijkstra dimulai dengan menginisialisasi *shortestPathTable* dan *shortestPreviousStopoverTable* sebagaimana yang telah dijelaskan pada bab II.

```java
ArrayList<Vertex> unvisited = new ArrayList<Vertex>();
HashMap<Integer, Boolean> visited = new HashMap<Integer, Boolean>();

shortestPathTable.put(start.vertexID, 0.0);
```

Lalu, dilakukan inisialisasi sebuah larik yang menyimpan simpul-simpul yang belum dikunjungi (*unvisited*) serta sebuah larik yang menandakan simpul apa saja yang sudah dikunjungi (*visited*). Selain itu, dilakukan inisialisasi data *shortestPathTable* yaitu jarak dari simpul *start* ke simpul *start* adalah 0 (karena tidak perlu bergerak ke simpul lain).

```java
Vertex current = start;
while (current != null) {
  visited.put(current.vertexID, true);
  unvisited.remove(current);
```

Algoritma dimulai dengan iterasi yang dilakukan dengan cara mengunjungi simpul-simpul. Simpul yang dikunjungi disebut *current*. Setiap suatu simpul dikunjungi, tandai bahwa simpul tersebut sudah dikunjungi dengan menambah informasi pada larik *visited* dan menghapus informasi pada larik *unvisited*.

```java
visited.put(current.vertexID, true);
unvisited.remove(current);
for (HashMap.Entry<Vertex, Double> entry : current.adjVertices.entrySet()) {
  Vertex adjV = entry.getKey();
  Double weight = entry.getValue();
  if (!visited.containsKey(adjV.vertexID)) {
    unvisited.add(adjV);
  }
  double weightThroughCurrent = shortestPathTable.get(current.vertexID) + weight;
  if (!shortestPathTable.containsKey(adjV.vertexID) || weightThroughCurrent < shortestPathTable.get(adjV.vertexID)) {
    shortestPathTable.put(adjV.vertexID, weightThroughCurrent);
    shortestPreviousStopoverTable.put(adjV.vertexID, current.vertexID);
  }
}
```

Lakukan iterasi untuk melihat setiap simpul tetangga dari *current* yang disimpan pada properti *adjVertices*. Setiap simpul tetangga ditandai sebagai *unvisited* apabila simpul tersebut belum pernah dikunjungi sebelumnya. Setelah itu, dihitung bobot yang perlu ditempuh dari simpul *start* sampai ke simpul hasil iterasi melalui *current*. Bobot tersebut disimpan pada *weightThroughCurrent*. Apabila bobot hasil kalkulasi lebih kecil dari data yang ada pada *shortestPathTable* menuju simpul hasil iterasi, data di *shortestPathTable* diperbarui sesuai nilai dari bobot hasil kalkulasi. Selain itu, untuk menyimpan data rute-rute yang dilalui, rute sejauh ini disimpan dalam *shortestPreviousStopoverTable*

```java
double minWeight = -1.0;
for (Vertex v: unvisited) {
  if (minWeight == -1.0) {
    minWeight = shortestPathTable.get(v.vertexID);
    current = v;
  } else {
    double tmp = shortestPathTable.get(v.vertexID);
    if (tmp < minWeight) {
      minWeight = tmp;
      current = v;
    }
  }
}
if (minWeight == -1.0) {
  current = null;
}
}
```

Setelah semua simpul tetangga diproses, algoritma Dijkstra akan memilih simpul *current* selanjutnya, yaitu suatu simpul yang belum dikunjungi (disimpan dalam larik *unvisited*). Apabila ada lebih dari satu simpul, simpul yang dipilih adalah simpul yang paling dekat. Jika larik *unvisited* adalah larik kosong, maka *current* akan diset sebagai *null* yang menandakan akhir dari algoritma Dijkstra.

Setelah algoritma Dijkstra berakhir, larik *shortestPathTable[x]* akan menyimpan jarak terdekat dari simpul *start* ke simpul *x*. Namun, untuk mengetahui rute yang diperlukan dari simpul *start* ke simpul *x*, informasi yang disimpan pada larik *shortestPreviousStopoverTable* perlu diproses lebih lanjut. Proses tersebut digambarkan oleh potongan kode di bawah ini.

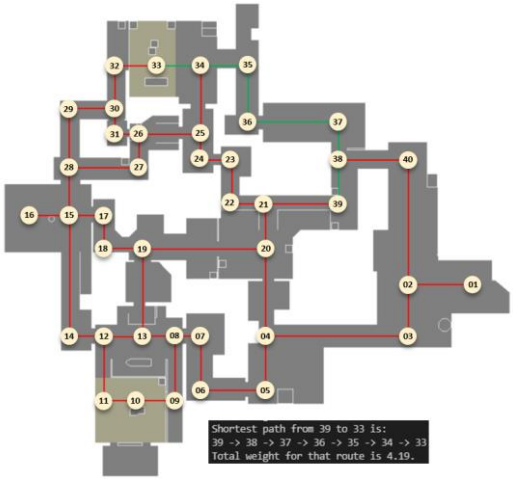
```java
ArrayList<Integer> shortestPath = new ArrayList<Integer>();

int currentID = dest.vertexID;
while (currentID != start.vertexID) {
  shortestPath.add(currentID);
  currentID = shortestPreviousStopoverTable.get(currentID);
}
shortestPath.add(start.vertexID);

System.out.println("Shortest path from " + start.vertexID + " to " + dest.vertexID + " is: ");
/* Cetak array */
for (int i = shortestPath.size() - 1; i >= 0; i --) {
  if (i > 0) {
    System.out.print(shortestPath.get(i) + " -> ");
  } else {
    System.out.println(shortestPath.get(i));
  }
}
System.out.println("Total weight for that route is " + shortestPathTable.get(dest.vertexID) + ".");
```

Data pada *shortestPreviousStopoverTable* diiterasi dan dikumpulkan datanya dari simpul tujuan (*dest*) menuju simpul awal (*start*). Data ini disimpan pada *shortestPath*. Namun, data *shortestPath* ini disimpan secara terbalik karena dimulai dari simpul tujuan. Dengan demikian, output dilakukan dari index terakhir larik *shortestPath* hingga index pertama.

Kode lengkap program dapat dilihat pada *repository* https://github.com/acomarcho/dijkstra-java.

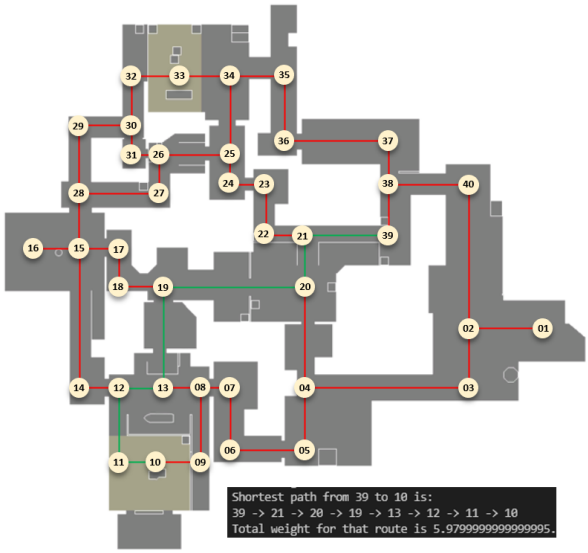
C. Hasil uji coba program

Berikut adalah beberapa contoh hasil eksekusi program untuk mencari jalur terdekat untuk mengunjungi suatu *site*, ditandai oleh simpul 33 dan simpul 10, yang dimulai dari simpul 39.



Gambar 8. Rute terpendek untuk memasuki *site* (simpul 33) dimulai dari simpul 39.

(Sumber: Dokumen pribadi)



Gambar 9. Rute terpendek untuk memasuki *site* (simpul 11) dimulai dari simpul 39.

Selain untuk mengetahui rute terpendek, hasil analisis lebih lanjut terhadap gambar 8 dan gambar 9 menunjukkan bahwa apabila pemain sedang berada pada simpul 39, dilihat dari aspek waktu, pemain lebih baik untuk pergi ke *site* yang dilambangkan oleh simpul 33 dibandingkan pergi ke *site* yang dilambangkan oleh simpul 11.

D. Analisis kompleksitas program

Program yang penulis buat menggunakan larik sederhana untuk menyimpan simpul-simpul yang belum pernah dikunjungi (*unvisited*). Apabila data graf yang disediakan adalah suatu graf lengkap (semua simpul berhubungan dengan simpul-simpul lainnya), pengecekan untuk suatu simpul v_1 dilakukan dengan mengecek v_2, v_3, ..., v_V di mana setiap pengecekan terhadap simpul v_i melakukan pengecekan terhadap simpul-simpul tetangganya juga. Hal ini menyebabkan kompleksitas program menjadi $O(V^2)$ dengan V adalah jumlah simpul yang ada.

IV. KESIMPULAN

Beberapa macam permasalahan dapat direpresentasikan dalam bentuk graf. Salah satunya adalah permasalahan mencari jarak terdekat antar dua titik, dalam kasus ini masalahnya dikhususukan kepada pencarian rute terdekat untuk memasuki *site* pada permainan Valorant. Dengan memodelkan *map* Ascent ke dalam bentuk graf, dapat diketahui rute terdekat untuk memasuki *site* yang diinginkan dari suatu posisi. Pencarian rute terdekat ini dapat dilakukan menggunakan algoritma Dijkstra. Algoritma ini akan mengembalikan panjang rute terpendek dan lintasan berupa simpul-simpul apa saja yang harus ditempuh untuk mencapai tujuannya.

V. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada seluruh pihak yang telah membantu pengerjaan makalah ini sehingga pengerjaan makalah ini dapat dikerjakan tepat pada waktunya. Khususnya kepada Tuhan yang Maha Esa karena tanpa rahmat dan karunia-Nya makalah ini tentu saja tidak akan ada. Selain itu, penulis secara khusus mengucapkan terima kasih kepada keluarga dan Ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen pengampu mata kuliah IF2120 Matematika Diskrit kelas 03. Penulis juga berterima kasih kepada pembuat referensi yang penulis gunakan untuk makalah ini.

REFERENSI

[1] Munir, Rinaldi. 2021. *Graf (Bag. 1): Bahan Kuliah IF2120 Matematika Diskrit*. Merupakan slide kuliah yang diunduh dari *platform* Edunex yang diakses pada tanggal 9 November 2021.

[2] Rosen, Kenneth H. 2019. *Discrete Mathematics and Its Applications, Eighth Edition.* New York: McGraw-Hill Education.

[3] Laaksonen, Antti. 2020. *Guide to Competitive Progrmaming: Learning and Improving Algorithms Through Contests, Second Edition*. Switzerland: Springer Nature.

[4] Wengrow, Jay. 2020. *A Common-Sense Guide to Data Structures and Algorithms: Level Up Your Core Programming Skills, Second Edition*. Pragmatic Bookshelf.

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, 11 Desember 2021

Marchotridyo 13520119