TUGAS AKHIR – KI1502

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JALUR TERPENDEK SATU SUMBER PADA GRAPH YANG BERUBAH SECARA DINAMIS

IVAN HENDRAJAYA
NRP 5111100063

Dosen Pembimbing I
Ahmad Saikhu, S.Si., M.T.

Dosen Pembimbing II
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA 2014

TUGAS AKHIR – KI1502

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JALUR TERPENDEK SATU SUMBER PADA GRAPH YANG BERUBAH SECARA DINAMIS

IVAN HENDRAJAYA
NRP 5111100063

Dosen Pembimbing I
Ahmad Saikhu, S.Si., M.T.

Dosen Pembimbing II
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA 2014

(Halaman ini sengaja dikosongkan)

UNDERGRADUATE THESIS – KI1502

ALGORITHM DESIGN AND ANALYSIS FOR SINGLE SOURCE SHORTEST PATH COMPUTATION ON DYNAMICALLY CHANGING GRAPH

IVAN HENDRAJAYA
NRP 5111100063

Supervisor I
Ahmad Saikhu, S.Si., M.T.

Supervisor II
Rully Soelaiman, S.Kom., M.Kom.

DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION TECHNOLOGY
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA 2014

(Halaman ini sengaja dikosongkan)

LEMBAR PENGESAHAN

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JALUR TERPENDEK SATU SUMBER PADA GRAPH YANG BERUBAH SECARA DINAMIS

TUGAS AKHIR
Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Dasar dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
IVAN HENDRAJAYA
NRP 5111100063

Disetujui oleh Dosen Pembimbing Tugas Akhir:

Ahmad Saikhu, S.Si., M.T. .....
NIP 197107182006041001 (Pembimbing I)

Rully Soelaiman, S.Kom., M.Kom. .....
NIP 197002131994021001 (Pembimbing II)

SURABAYA
DESEMBER 2014

(Halaman ini sengaja dikosongkan)

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JALUR TERPENDEK SATU SUMBER PADA GRAPH YANG BERUBAH SECARA DINAMIS

Nama : **Ivan Hendrajaya**
NRP : **5111100063**
Jurusan : **Teknik Informatika**
Fakultas Teknologi Informasi ITS
Dosen Pembimbing I : **Ahmad Saikhu, S.Si., M.T.**
Dosen Pembimbing II : **Rully Soelaiman, S.Kom., M.Kom.**

ABSTRAK

Banyak permasalahan dalam kehidupan sehari-hari yang bisa direpresentasikan sebagai permasalahan jalur terpendek. Solusi untuk permasalahan jalur terpendek pada graph statis saja tidak cukup, karena pada kenyataannya dalam kehidupan sehari-hari lingkungan berubah secara dinamis.

Hal tersebut memunculkan permasalahan jalur terpendek pada graph dinamis yaitu memungkinkan modifikasi pada graph, termasuk penyisipan atau penghapusan edge dengan tetap mempertahankan jalur terpendek tanpa melakukan komputasi ulang dari nol.

Pada Tugas Akhir ini akan diimplementasikan solusi untuk permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif. Untuk setiap modifikasi pada graph, dilakukan update informasi pada vertex yang mengalami perubahan jarak dari vertex sumber atau perubahan parent pada shortest path tree.

Solusi tersebut mempunyai kompleksitas space linear dan kompleksitas time optimal untuk setiap kueri jalur terpendek. Untuk setiap update pada vertex, solusi tersebut mempunyai kompleksitas time $O(\log n)$ *amortized time untuk graph dengan genus tertentu, arboricity tertentu, degree tertentu, treewidth tertentu, pagenumber tertentu. Sedangkan untuk general graph*

dengan n vertex dan m edge, solusi tersebut mempunyai kompleksitas time $O(\sqrt{m}\, log\, n)$ *amortized time untuk setiap update pada vertex.*

Kata kunci: Graph Dinamis, Jalur Terpendek Satu Sumber, Penghapusan Edge, Penyisipan Edge.

ALGORITHM DESIGN AND ANALYSIS FOR SINGLE SOURCE SHORTEST PATH COMPUTATION ON DYNAMICALLY CHANGING GRAPH

Name	**: Ivan Hendrajaya**
NRP	**: 5111100063**
Department	**: Department of Informatics**
	Faculty of Information Technology ITS
Supervisor I	**: Ahmad Saikhu, S.Si., M.T.**
Supervisor II	**: Rully Soelaiman, S.Kom., M.Kom.**

ABSTRACT

Many problems in everyday life can be represented as a shortest path problem. The solution for shortest path problem on static graph is inadequate, because the fact that in everyday life environments change dynamically.

This led to shortest path problem on dynamic graph that allows modification on the graph, including insertion or deletion of edge while maintaining the shortest path without recomputing from scratch.

In this undergraduate thesis will be implemented solution for single source shortest path problem on dynamic graph with positive real edge weights. For each modifications on the graph, information update performed on vertices that either change the distance from the source vertex or change the parent in the shortest path tree.

The space complexity of the solution is linear and the time complexity is optimal for each query on shortest path. For each vertex update, the time complexity of the solution is $O(\log n)$ *amortized time for graph with bounded genus, bounded arboricity, bounded degree, bounded treewidth, bounded pagenumber. Whereas for general graph with n vertices and m edges, the time complexity of the solution is* $O\left(\sqrt{m}\log n\right)$ *amortized time for each vertex update.*

Keywords: Dynamic Graph, Edge Deletion, Edge Insertion, Single Source Shortest Path.

KATA PENGANTAR

Puji syukur saya panjatkan kepada Tuhan Yang Maha Esa yang telah memberikan rahmat dan karunia-Nya, sehingga saya dapat menyelesaikan Tugas Akhir yang berjudul:

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JALUR TERPENDEK SATU SUMBER PADA GRAPH YANG BERUBAH SECARA DINAMIS

Tugas Akhir ini diajukan untuk memenuhi salah satu syarat memperoleh gelar Sarjana Komputer di Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember.

Dengan selesainya Tugas Akhir ini saya harap apa yang telah saya kerjakan dapat memberikan manfaat bagi perkembangan ilmu pengetahuan serta bagi saya selaku penulis Tugas Akhir ini.

Saya selaku penulis menyampaikan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama saya menyelesaikan Tugas Akhir antara lain:

1. Tuhan Yang Maha Esa yang telah memberikan rahmat dan karunia-Nya sehingga saya dapat menyelesaikan Tugas Akhir dengan baik.
2. Keluarga saya yang telah memberikan doa dan dukungan baik secara moral maupun finansial selama saya menyelesaikan Tugas Akhir.
3. Ibu Dr. Eng. Nanik Suciati, S.Kom., M.Kom. selaku Ketua Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember.
4. Bapak Radityo Anggoro, S.Kom., M.Sc. selaku Koordinator Tugas Akhir.

5. Bapak Ahmad Saikhu, S.Si., M.T. selaku Dosen Pembimbing I yang telah memberikan dukungan dan bimbingan kepada saya dalam menyelesaikan Tugas Akhir.
6. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku Dosen Pembimbing II yang telah memberikan ilmu, motivasi, kritik, dan saran kepada saya dalam menyelesaikan Tugas Akhir.
7. Seluruh teman saya dalam menempuh perkuliahan di Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember.
8. Seluruh pihak yang tidak bisa saya sebutkan satu persatu yang telah memberikan dukungan selama saya menyelesaikan Tugas Akhir.

Saya memohon maaf apabila terdapat kekurangan dalam Tugas Akhir ini. Kritik dan saran saya harapkan untuk perbaikan dan pembelajaran di kemudian hari. Semoga Tugas Akhir ini dapat memberikan manfaat yang sebaik-baiknya.

Surabaya, Desember 2014

Penulis

DAFTAR ISI

(Halaman ini sengaja dikosongkan)

DAFTAR GAMBAR

DAFTAR TABEL

(Halaman ini sengaja dikosongkan)

DAFTAR KODE SUMBER

BAB I
PENDAHULUAN

Pada bab ini penulis menjelaskan tentang latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi, dan sistematika penulisan Tugas Akhir

1.1 Latar Belakang

Banyak permasalahan dalam kehidupan sehari-hari yang bisa direpresentasikan sebagai permasalahan jalur terpendek. Permasalahan tersebut diantaranya terdapat pada bidang jaringan transportasi dan komunikasi. Pada kedua bidang tersebut masalah yang dihadapi hampir sama yaitu berapa jarak / biaya / waktu minimal untuk pergi dari A ke B?

Banyak solusi yang efisien untuk permasalahan jalur terpendek pada graph statis. Namun pada kenyataannya dalam kehidupan sehari-hari lingkungan berubah secara dinamis. Seringkali terjadi perubahan pada lingkungan seperti dibuka atau ditutupnya jalan maupun koneksi yang terhubung atau terputus, namun jalur terpendek harus tetap dipertahankan.

Hal tersebut memunculkan permasalahan jalur terpendek pada graph dinamis. Permasalahan yang terjadi adalah bagaimana solusi yang efisien untuk mempertahankan jalur terpendek jika terjadi modifikasi pada graph, termasuk penyisipan atau penghapusan edge. Solusi yang paling mudah tentu saja melakukan komputasi ulang dari nol setiap terjadi modifikasi pada graph, tetapi hal tersebut tentu saja tidak efisien.

Pada Tugas Akhir ini akan diimplementasikan solusi untuk permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif. Solusi tersebut mempunyai kompleksitas *time* optimal untuk setiap kueri jalur terpendek dan kompleksitas *time* yang cukup baik untuk setiap vertex update ketika dilakukan penyisipan atau penghapusan edge. Selain itu solusi tersebut mempunyai kompleksitas *space* linear.

1.2 Rumusan Masalah

Berikut beberapa hal yang menjadi rumusan masalah dalam Tugas Akhir ini:

1. Bagaimana desain algoritma dan struktur data yang efisien untuk permasalahan jalur terpendek pada graph dinamis?
2. Bagaimana implementasi algoritma dan struktur data yang efisien berdasarkan desain yang telah dilakukan?
3. Bagaimana uji coba untuk mengetahui kebenaran dan kinerja dari implementasi yang telah dilakukan?

1.3 Batasan Masalah

Berikut beberapa hal yang menjadi batasan masalah dalam Tugas Akhir ini:

1. Solusi yang diimplementasikan hanya untuk permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif.
2. Solusi yang diimplementasikan hanya untuk graph dengan banyak vertex [2..1000], banyak edge [1..10000], bobot tiap edge [1..100], serta banyak kueri jalur terpendek dan modifikasi pada graph [1..1000].
3. Bahasa pemrograman yang digunakan adalah C++.

1.4 Tujuan

Tujuan dari Tugas Akhir ini adalah mengimplementasikan solusi untuk permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif dan melakukan uji coba untuk mengetahui kebenaran dan kinerja dari implementasi yang telah dilakukan.

1.5 Metodologi

Berikut metodologi yang digunakan dalam Tugas Akhir ini:

1. Penyusunan proposal Tugas Akhir
   Pada tahap ini dilakukan penyusunan proposal Tugas Akhir yang berisi definisi permasalahan jalur terpendek pada graph dinamis beserta gambaran umum mengenai

solusi untuk permasalahan jalur terpendek pada graph dinamis.

2. Studi literatur
   Pada tahap ini dilakukan studi literatur mengenai solusi untuk permasalahan jalur terpendek pada graph dinamis. Literatur yang digunakan antara lain paper, buku referensi, dan artikel yang didapatkan dari internet.
3. Desain
   Pada tahap ini dilakukan desain algoritma serta struktur data dari solusi untuk permasalahan jalur terpendek pada graph dinamis.
4. Implementasi
   Pada tahap ini dilakukan implementasi solusi untuk permasalahan jalur terpendek pada graph dinamis berdasarkan analisis dan desain yang telah dilakukan.
5. Uji coba dan evaluasi
   Pada tahap ini dilakukan uji coba untuk menguji kebenaran dan kinerja dari implementasi yang telah dilakukan. Pengujian kebenaran dilakukan dengan beberapa input (kasus uji), kemudian membandingkan output dari hasil pengujian dengan output dari implementasi yang lain. Pengujian kinerja dilakukan dengan membandingkan kompleksitas yang didapat dari hasil pengujian dengan kompleksitas yang didapat dari hasil analisis. Selain itu dilakukan evaluasi untuk mencari bagian-bagian yang masih bisa dioptimasi dan melakukan perbaikan jika terdapat kesalahan.
6. Penyusunan buku Tugas Akhir
   Pada tahap ini dilakukan penyusunan buku Tugas Akhir yang berisi dokumentasi mengenai solusi untuk permasalahan jalur terpendek pada graph dinamis.

1.6 Sistematika Penulisan

Berikut sistematika penulisan buku Tugas Akhir ini:

1. **BAB I: PENDAHULUAN**
   Bab ini berisi latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi, dan sistematika penulisan Tugas Akhir.
2. **BAB II: TINJAUAN PUSTAKA**
   Bab ini berisi dasar teori mengenai permasalahan dan algoritma yang digunakan dalam Tugas Akhir.
3. **BAB III: DESAIN**
   Bab ini berisi desain algoritma serta struktur data yang digunakan dalam Tugas Akhir.
4. **BAB IV: IMPLEMENTASI**
   Bab ini berisi implementasi berdasarkan desain algoritma serta struktur data yang telah dilakukan.
5. **BAB V: UJI COBA DAN EVALUASI**
   Bab ini berisi uji coba dan evaluasi dari implementasi yang telah dilakukan.
6. **BAB VI: KESIMPULAN DAN SARAN**
   Bab ini berisi kesimpulan dari hasil uji coba yang telah dilakukan dan saran mengenai hal-hal yang masih bisa dikembangkan.

BAB II
TINJAUAN PUSTAKA

Pada bab ini penulis menjelaskan tentang beberapa tinjauan pustaka mengenai permasalahan dan algoritma yang digunakan dalam Tugas Akhir.

2.1 Definisi Graph

Sebuah graph $G = (V, E)$ terdiri dari himpunan vertex V dan himpunan edge E. Sebuah edge (u, v) terdiri dari pasangan vertex u dan vertex v. Vertex u dan vertex v disebut *endpoint* dari sebuah edge (u, v). Edge (u, v) dikatakan *berincident* dengan vertex u dan vertex v. Jika terdapat edge (u, v) dalam sebuah graph, maka vertex u disebut *neighbor* dari vertex v dan vertex v disebut *neighbor* dari vertex u.

Graph berarah merupakan graph dimana edge (u, v) dan edge (v, u) dianggap berbeda, edge (u, v) artinya terdapat edge ke vertex tujuan v dari vertex sumber u sedangkan edge (v, u) artinya terdapat edge ke vertex tujuan u dari vertex sumber v. Graph tidak berarah merupakan graph dimana edge (u, v) dan edge (v, u) dianggap sama.

Sebuah edge (u, v) dikatakan *inedge* dari sebuah vertex v dan *outedge* dari sebuah vertex u. Banyaknya *inedge* yang *berincident* dengan vertex v disebut *indegree* dan banyaknya *outedge* yang *berincident* dengan vertex v disebut out*degree*.

2.2 Permasalahan Jalur Terpendek

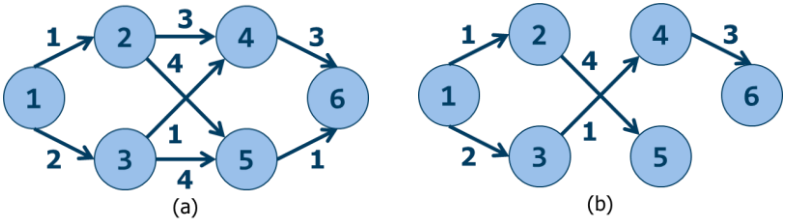
Terdapat beberapa jenis permasalahan jalur terpendek antara lain:

1. *Single source shortest path problem* yaitu menemukan jalur terpendek dari satu vertex sumber ke setiap vertex lain yang terdapat pada graph.
2. *Single destination shortest path problem* yaitu menemukan jalur terpendek ke satu vertex tujuan dari setiap vertex lain yang terdapat pada graph.

3. *Single pair shortest path problem* yaitu menemukan jalur terpendek dari satu vertex sumber ke satu vertex tujuan.
4. *All pairs shortest path problem* yaitu menemukan jalur terpendek dari vertex u ke vertex v untuk setiap pasangan vertex u dan vertex v yang terdapat pada graph.

2.3 Permasalahan Jalur Terpendek Satu Sumber

Diberikan graph berarah $G = (V, E)$ dengan sebuah vertex sumber $s \in V$. Untuk setiap edge $(u, v) \in E$, diberikan bobot edge bilangan real positif yang dinotasikan $w_{u,v}$. Untuk setiap vertex $v \in V$, jarak jalur terpendek ke vertex v dari vertex sumber s dinotasikan $d(v)$ dan didefinisikan dengan fungsi $d: V \rightarrow \mathbb{R}^+$. Graph G mempunyai shortest path tree yang dinotasikan $T(s) = (V_T, E_T)$ dengan root vertex sumber s. $T(v)$ merupakan subtree dari $T(s)$ dengan root vertex v. Untuk setiap vertex $v \in V_T$, terdapat sebuah parent dinotasikan $parent(v)$ (kecuali untuk vertex sumber s) dan himpunan children yang dinotasikan $children(v)$. Edge (u, v) disebut ***tree edge*** jika $(u, v) \in E_T$, sebaliknya disebut ***nontree edge***. Ilustrasi graph dan shortest path tree ditunjukkan dalam **Gambar 2.*1***.



Gambar 2.1 Ilustrasi Graph (a) dan Shortest Path Tree (b)

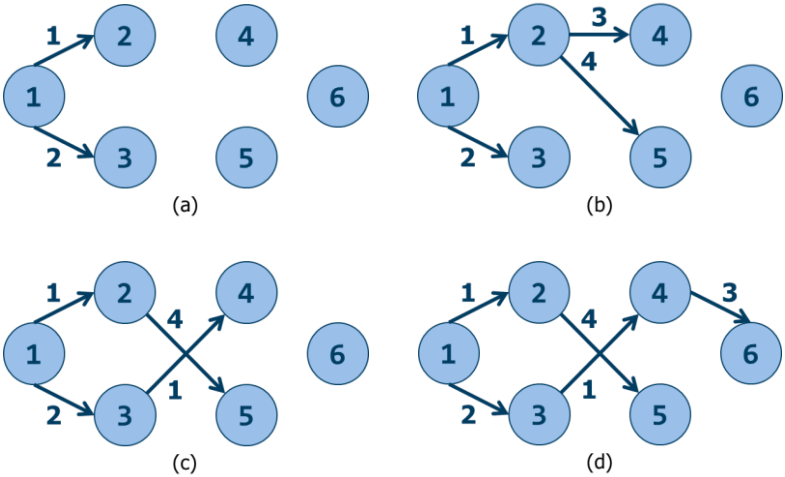
Graph pada **Gambar 2.*1*(a)** mempunyai shortest path tree pada **Gambar 2.*1*(b)** dengan vertex sumber yaitu vertex 1 (sebagai root dari shortest path tree). Edge $(1, 2)$, $(1, 3)$, $(2, 5)$, $(3, 4)$, dan $(4, 6)$ merupakan ***tree edge***. Sedangkan edge $(2, 4)$, $(3, 5)$, dan $(5, 6)$ merupakan ***nontree edge***.

2.4 Permasalahan Jalur Terpendek Satu Sumber pada Graph Dinamis

Pada permasalahan jalur terpendek satu sumber pada graph dinamis, terdapat beberapa jenis modifikasi pada graph antara lain penyisipan atau penghapusan edge dan penambahan atau pengurangan bobot edge. Penyisipan edge atau pengurangan bobot edge disebut sebagai permasalahan *incremental*. Penghapusan edge atau penambahan bobot edge disebut sebagai permasalahan *decremental*. Pada permasalahan *Semi-dynamic* hanya terdapat salah satu diantara permasalahan *incremental* dan permasalahan *decremental*, sedangkan pada permasalahan *Fully-dynamic* dimungkinkan terdapat kombinasi dari permasalahan *incremental* dan permasalahan *decremental*.

2.5 Algoritma Dijkstra

Algoritma Dijkstra merupakan salah satu algoritma yang dapat digunakan untuk menyelesaikan permasalahan jalur terpendek satu sumber pada graph statis dengan bobot edge bilangan real positif. Untuk graph dengan n vertex dan m edge, algoritma Dijkstra mempunyai kompleksitas *time* $O((m+n)\,log\,n)$. Pada awalnya untuk setiap vertex $v \in V$ dilakukan inisialisasi $d(v) = \infty$ dan $parent(v) = \emptyset$, kecuali untuk vertex sumber s dilakukan inisialisasi $d(s) = 0$. Kemudian *enqueue* vertex s dengan *priority* $d(s)$ pada *min priority queue* D. Selama *min priority queue* D tidak kosong, dilakukan *extract min* untuk mendapatkan vertex v dengan *priority* $d(v)$ minimum. Setelah itu dilakukan scanning pada edge (v,u). Jika $d(v) + w_{v,u} < d(u)$, maka dilakukan update yaitu $d(u) = d(v) + w_{v,u}$, $parent(u) = v$, dan *enqueue* vertex u dengan *priority* $d(u)$ pada *min priority queue* D jika $u \notin D$ atau *decrease key* vertex u dengan *priority* $d(u)$ pada *min priority queue* D jika $u \in D$. Ilustrasi shortest path tree yang terbentuk pada setiap iterasi dari graph pada **Gambar 2.*1*(a)** ditunjukkan dalam **Gambar 2.*2***.



Gambar 2.2 Ilustrasi Shortest Path Tree pada Iterasi 1 (a), Iterasi 2 (b), Iterasi 3 (c), dan Iterasi 4 (d)

Pada **Gambar 2.2** dapat dilihat shortest path tree yang terbentuk pada setiap iterasi. Pada iterasi 1, vertex 1 melakukan *scanning* pada edge $(1, 2)$ dan $(1, 3)$ serta mengubah parent dari vertex 2 dan vertex 3. Pada iterasi 2, vertex 2 melakukan *scanning* pada edge $(2, 4)$ dan $(2, 5)$ serta mengubah parent dari vertex 4 dan vertex 5. Pada iterasi 3, vertex 3 melakukan *scanning* pada edge $(3, 4)$ dan $(3, 5)$ serta mengubah parent dari vertex 4. Pada iterasi 4, vertex 4 melakukan *scanning* pada edge $(4, 6)$ serta mengubah parent dari vertex 6. Pada iterasi 5, vertex 5 melakukan *scanning* pada edge $(5, 6)$ namun shortest path tree tidak berubah. Pada iterasi 6, vertex 6 tidak melakukan *scanning* serta shortest path tree tidak berubah.

2.6 Algoritma Dijkstra Dinamis

Algoritma Dijkstra Dinamis merupakan algoritma yang dapat digunakan untuk menyelesaikan permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif. Untuk graph dengan n vertex dan m edge, algoritma Dijkstra Dinamis mempunyai kompleksitas *time* $O(\sqrt{m}\, log\, n)$

amortized time untuk setiap vertex update. Terdapat 2 bagian Algoritma Dijkstra Dinamis yaitu algoritma penyisipan edge dan algoritma penghapusan edge. Algoritma Dijkstra Dinamis berbasis pada algoritma Dijkstra, dimana terdapat *priority queue* yang berisi vertex dengan *priority* jarak jalur terpendek ke vertex tersebut dari vertex sumber. Pada algoritma Dijkstra, ketika sebuah vertex v *didequeue* dari *priority queue*, dilakukan *scanning* pada setiap edge yang *berincident* dengan vertex v. Sedangkan pada algoritma Dijkstra Dinamis, ketika sebuah vertex v *didequeue* dari *priority queue*, hanya dilakukan *scanning* pada subset dari edge yang *berincident* dengan vertex v.

Agar dapat menentukan edge yang harus *discan* dengan tepat, digunakan ***backward level*** dari sebuah edge (v, u) dan sebuah vertex u, relatif terhadap vertex v, dan ***forward level*** dari sebuah edge (u, v) dan sebuah vertex u, relatif terhadap vertex v yang didefinisikan dengan Persamaan **(2.1)** dan Persamaan **(2.2)**.

$$b\_level_v(u) = d(u) - w_{v,u} \quad \textbf{(2.1)}$$

$$f\_level_v(u) = d(u) + w_{u,v} \quad \textbf{(2.2)}$$

Pada algoritma penyisipan edge, jika penyisipan edge menyebabkan $d'(v)$ berkurang hingga lebih kecil dari $b\_level_v(u)$ atau $d'(v) + w_{v,u} < d(u)$, artinya terdapat jalur terpendek baru ke vertex u yang lebih baik dari jalur terpendek ke vertex u saat ini. Jika dilakukan *scanning* pada edge (v, u) dalam urutan b_level yang *nonincreasing*, maka *scanning* hanya dilakukan pada edge yang tepat yaitu edge (v, u) dimana jarak jalur terpendek ke vertex u berkurang.

Pada algoritma penghapusan edge, jika penghapusan edge menyebabkan $d'(v)$ bertambah hingga lebih besar dari $f\_level_v(u)$ atau $d'(v) > d(u) + w_{u,v}$, artinya terdapat jalur terpendek alternatif ke vertex v melalui vertex u. Jika dilakukan *scanning* pada edge (u, v) dalam urutan f_level yang *nondecreasing*, maka *scanning* hanya dilakukan pada edge yang tepat yaitu edge (u, v) dimana jarak jalur terpendek ke vertex v melalui vertex u lebih baik dari jarak jalur terpendek ke vertex v saat ini.

Untuk membatasi banyak edge yang *discan*, dilakukan partisi pada himpunan edge menjadi 2 bagian. Setiap edge (u, v) mempunyai *owner* yaitu salah satu dari vertex u atau vertex v. Untuk setiap vertex v terdapat 2 himpunan yaitu himpunan $owned(v)$ yang anggotanya merupakan edge yang dimiliki vertex v dan himpunan $not\_owned(v)$ yang anggotanya merupakan edge yang *berincident* dengan vertex v tetapi tidak dimiliki oleh vertex v.

Untuk menyimpan himpunan $not\_owned(v)$ dari sebuah vertex $v \in V$ digunakan 2 *priority queue* sebagai berikut:

1. Setiap *inedge* (u, v) disimpan dalam *min priority queue* F dengan priority $f\_level_v(u)$.
2. Setiap *outedge* (v, u) disimpan dalam *max priority queue* B dengan priority $b\_level_v(u)$.

Hal ini dilakukan karena algoritma penghapusan edge membutuhkan *scanning* edge dalam urutan f_level yang *nondecreasing* dan algoritma penyisipan edge membutuhkan *scanning* edge dalam urutan b_level yang *nonincreasing*.

2.6.1 Algoritma Penyisipan Edge

Algoritma penyisipan edge dibagi menjadi 3 tahap. Pada tahap 1, dipilih *owner* dari edge (u, v) yaitu salah satu dari vertex u atau vertex v. Setelah itu dilakukan update f_level_v jika *owner* dari edge (u, v) adalah vertex u. Sedangkan jika *owner* dari edge (u, v) adalah vertex v, maka dilakukan update b_level_u. Jika penyisipan edge (u, v) tidak menyebabkan $d'(v)$ berkurang atau $d(u) + w_{u,v} \geq d(v)$, maka algoritma ini dihentikan. Selain itu jika vertex u dan vertex v tidak terhubung dengan vertex sumber s, maka algoritma ini juga dihentikan. *Graph coloring* digunakan untuk mengetahui vertex mana saja yang memerlukan update untuk mempertahankan struktur data yang digunakan setelah dilakukan penyisipan edge. Pada awalnya untuk setiap vertex $v \in V$ dilakukan inisialisasi $color(v) = white$. Jika $d'(v) < d(v)$ atau vertex v mengalami perubahan jarak jalur terpendek dari vertex sumber s, maka $color(v) = red$. Kemudian dilakukan inisialisasi

untuk tahap 2, yaitu $d'(v) = d(u) + w_{u,v}$, $parent(v) = u$, $color(v) = red$, dan *enqueue* vertex v dengan *priority* $d'(v)$ pada *min priority queue* C.

Pada tahap 2 dilakukan komputasi jalur terpendek dengan cara yang mirip dengan algoritma Dijkstra. Selama *min priority queue* C tidak kosong, dilakukan *extract min* untuk mendapatkan vertex v dengan *priority* $d'(v)$ minimum. Setelah itu dilakukan scanning pada edge (v, u) yang memenuhi salah satu dari kondisi berikut:

1. Vertex v adalah owner dari edge (v, u), dalam hal ini edge (v, u) ***discan by ownership***.
2. Vertex u adalah owner dari edge (v, u) dan $d'(v) < b\_level_v(u)$, dalam hal ini edge (v, u) ***discan by priority***.

Jika $d'(v) + w_{v,u} < d'(u)$, maka dilakukan update yaitu $d'(u) = d'(v) + w_{v,u}$, $parent(u) = v$, $color(u) = red$, dan *enqueue* vertex u dengan *priority* $d'(u)$ pada *min priority queue* C jika $u \notin C$ atau *decrease key* vertex u dengan *priority* $d'(u)$ pada *min priority queue* C jika $u \in C$.

Pada tahap 3 untuk setiap vertex v dimana $color(v) = red$ dilakukan update pada vertex u dimana owner edge (u, v) atau (v, u) adalah vertex v sebagai berikut:

1. Jika $color(u) \neq red$, maka dilakukan perubahan owner dari edge (u, v) atau (v, u) menjadi vertex u. Perubahan ini termasuk menghapus edge (u, v) atau (v, u) dari himpunan $owned(v)$ dan himpunan $not\_owned(u)$ serta menyisipkan edge (u, v) atau (v, u) pada himpunan $owned(u)$ dan himpunan $not\_owned(v)$.
2. Jika $color(u) = red$, maka dilakukan update b_level_u untuk edge (u, v) dan f_level_u untuk edge (v, u).

2.6.2 Algoritma Penghapusan Edge

Algoritma penghapusan edge dibagi menjadi 4 tahap. Pada tahap 1, edge (u, v) dihapus dari himpunan $owned(u)$ dan himpunan $not\_owned(v)$ jika *owner* dari edge (u, v) adalah vertex u. Sedangkan jika *owner* dari edge (u, v) adalah vertex v, maka edge (u, v) dihapus dari himpunan $owned(v)$ dan himpunan

$not\_owned(u)$. Jika penghapusan edge (u, v) tidak menyebabkan $d'(v)$ bertambah atau edge (u, v) merupakan ***nontree edge***, maka algoritma ini dihentikan. Selain itu jika vertex u dan vertex v tidak terhubung dengan vertex sumber s, maka algoritma ini juga dihentikan. *Graph coloring* digunakan untuk mengetahui vertex mana saja yang memerlukan update untuk mempertahankan struktur data yang digunakan setelah dilakukan penghapusan edge. Selain itu *graph coloring* juga digunakan untuk menentukan vertex yang mengalami perubahan jarak jalur terpendek dari vertex sumber s dan mencari jalur terpendek alternatif setelah dilakukan penghapusan edge. Pada awalnya untuk setiap vertex $v \in V$ dilakukan inisialisasi $color(v) = white$. Jika $d'(v) > d(v)$ atau vertex v mengalami perubahan jarak jalur terpendek dari vertex sumber s, maka $color(v) = red$. Kemudian dilakukan inisialisasi untuk tahap 2, yaitu *enqueue* vertex v dengan *priority* $d(v)$ pada *min priority queue* M.

Pada tahap 2 dilakukan *graph coloring*. Selama *min priority queue* M tidak kosong, dilakukan *extract min* untuk mendapatkan vertex v dengan *priority* $d(v)$ minimum. Setelah itu dicari ***nonred neighbor*** terbaik dari vertex v dengan cara melakukan scanning pada edge (u, v) yang memenuhi salah satu dari kondisi berikut:

1. Vertex v adalah owner dari edge (u, v), dalam hal ini edge (u, v) ***discan by ownership***.
2. Vertex u adalah owner dari edge (u, v) dan $d(v) > f\_level_v(u)$, dalam hal ini edge (u, v) ***discan by priority***.

Jika terdapat ***nonred neighbor*** terbaik vertex u sehingga $d(u) + w_{u,v} = d(v)$ atau vertex v tidak mengalami perubahan jarak jalur terpendek dari vertex sumber s, maka $parent(v) = u$. Sedangkan jika tidak, maka $color(v) = red$ dan *enqueue* vertex $u \in children(v)$ dengan *priority* $d(u)$ pada *min priority queue* M.

Pada tahap 3 dilakukan komputasi jalur terpendek untuk setiap vertex v dimana $color(v) = red$ dengan cara yang mirip dengan algoritma Dijkstra. Pada awalnya dilakukan inisialisasi *min priority queue* Q dengan cara mencari ***nonred neighbor*** terbaik dari setiap vertex v dimana $color(v) = red$. Jika terdapat ***nonred***

neighbor terbaik vertex u, maka dilakukan update yaitu $d'(v) = d(u) + w_{u,v}$, $parent(v) = u$, dan *enqueue* vertex v dengan *priority* $d'(v)$ pada *min priority queue* Q. Sedangkan jika tidak, maka dilakukan update yaitu $d'(v) = \infty$ dan $parent(v) = \emptyset$. Kemudian selama *min priority queue* Q tidak kosong, dilakukan *extract min* untuk mendapatkan vertex v dengan *priority* $d'(v)$ minimum. Setelah itu dilakukan scanning pada setiap edge (v, u) dimana $color(u) = red$. Jika $d'(v) + w_{v,u} < d(u)$, maka dilakukan update yaitu $d'(u) = d'(v) + w_{v,u}$, $parent(u) = v$, dan *enqueue* vertex u dengan *priority* $d'(u)$ pada *min priority queue* Q jika $u \notin Q$ atau *decrease key* vertex u dengan *priority* $d'(u)$ pada *min priority queue* Q jika $u \in Q$.

Pada tahap 4 untuk setiap vertex v dimana $color(v) = red$ dilakukan update pada vertex u dimana owner edge (u, v) atau (v, u) adalah vertex v sebagai berikut:

1. Jika $color(u) \neq red$, maka dilakukan perubahan owner dari edge (u, v) atau (v, u) menjadi vertex u. Perubahan ini termasuk menghapus edge (u, v) atau (v, u) dari himpunan $owned(v)$ dan himpunan $not\_owned(u)$ serta menyisipkan edge (u, v) atau (v, u) pada himpunan $owned(u)$ dan himpunan $not\_owned(v)$.
2. Jika $color(u) = red$, maka dilakukan update b_level_u untuk edge (u, v) dan f_level_u untuk edge (v, u).

2.7 Sphere Online Judge (SPOJ)

Sphere Online Judge (SPOJ) adalah sistem penilaian *online* yang berisi berbagai jenis permasalahan pemrograman yang dapat diselesaikan oleh pengguna dengan mengirim kode sumber program yang berisi solusi dari permasalahan yang diberikan. Setiap permasalahan mempunyai format masukan yang diberikan dan format keluaran yang diminta. Selain itu setiap permasalahan juga mempunyai batasan tertentu termasuk lingkungan penilaian, batasan waktu, batasan memori, dan batasan dari permasalahan yang dideskripsikan.

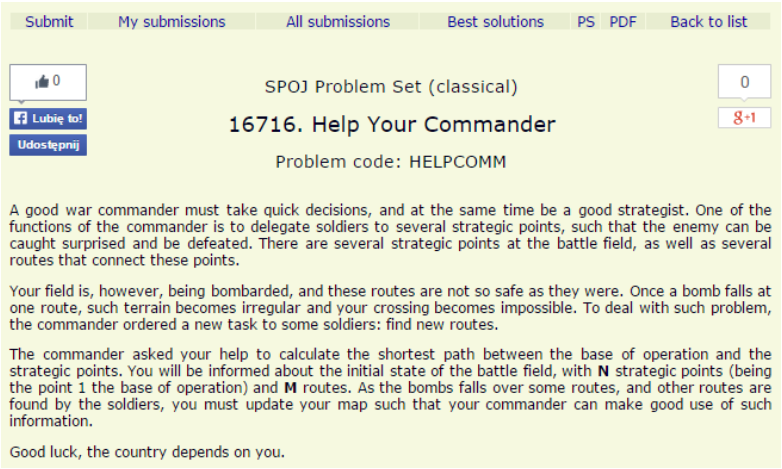
Kode sumber program yang diterima sistem akan dikompilasi dan dijalankan pada lingkungan penilaian sistem. Program akan diuji menggunakan masukan dari berkas masukan permasalahan kemudian hasil keluaran program akan dibandingkan dengan berkas keluaran permasalahan. Sistem akan memberikan umpan balik kepada pengguna antara lain:

1. ***Accepted***, artinya program tidak melanggar batasan yang diberikan dan hasil keluaran program sama dengan berkas keluaran permasalahan.
2. ***Wrong Answer***, artinya hasil keluaran program tidak sama dengan berkas keluaran permasalahan.
3. ***Time Limit Exceeded***, artinya program melanggar batasan waktu yang diberikan.
4. ***Memory Limit Exceeded***, artinya program melanggar batasan memori yang diberikan.
5. ***Runtime Error***, artinya terjadi *error* pada saat program dijalankan.
6. ***Compile Error***, artinya terjadi *error* pada saat kompilasi kode sumber program.

Selain itu sistem juga memberikan umpan balik kepada pengguna mengenai waktu dan memori yang dibutuhkan program pada saat diuji menggunakan masukan dari berkas masukan permasalahan.

2.8 Permasalahan *Help Your Commander* pada SPOJ

Salah satu permasalahan yang terdapat pada SPOJ adalah *Help Your Commander* yang mempunyai nomor 16716 dan kode HELPCOMM. Deskripsi permasalahan ditunjukkan dalam **Gambar 2.*3***.


Submit My submissions All submissions Best solutions PS PDF Back to list

0

Lubię to!

Udostępnij

SPOJ Problem Set (classical)

16716. Help Your Commander

Problem code: HELPCOMM

0

+1

A good war commander must take quick decisions, and at the same time be a good strategist. One of the functions of the commander is to delegate soldiers to several strategic points, such that the enemy can be caught surprised and be defeated. There are several strategic points at the battle field, as well as several routes that connect these points.

Your field is, however, being bombarded, and these routes are not so safe as they were. Once a bomb falls at one route, such terrain becomes irregular and your crossing becomes impossible. To deal with such problem, the commander ordered a new task to some soldiers: find new routes.

The commander asked your help to calculate the shortest path between the base of operation and the strategic points. You will be informed about the initial state of the battle field, with **N** strategic points (being the point 1 the base of operation) and **M** routes. As the bombs falls over some routes, and other routes are found by the soldiers, you must update your map such that your commander can make good use of such information.

Good luck, the country depends on you.

Gambar 2.3 Deskripsi Permasalahan Help Your Commander

Deskripsi singkat permasalahan tersebut adalah diberikan graph berarah dengan N vertex dan M edge. Setiap vertex diberi nomor mulai dari 1 hingga N. Hitung jarak jalur terpendek dari vertex sumber (vertex nomor 1) ke setiap vertex lain yang terdapat pada graph. Jika terjadi modifikasi pada graph termasuk penyisipan atau penghapusan edge, update jarak jalur terpendek dari vertex sumber ke setiap vertex lain yang terdapat pada graph. Berikut merupakan format masukan dari permasalahan tersebut:

1. Masukan terdiri dari beberapa kasus uji, format masukan setiap kasus uji didefinisikan pada poin 2 hingga poin 5.
2. Baris pertama berisi 2 buah integer N dan M yang merepresentasikan banyak vertex dan banyak edge.
3. M baris berikutnya masing-masing berisi 3 buah integer U, V, dan W yang merepresentasikan vertex sumber, vertex tujuan, dan bobot edge.
4. Baris berikutnya berisi 1 buah integer Q yang merepresentasikan banyak operasi.
5. Q baris berikutnya masing-masing berisi deskripsi operasi yang dilakukan antara lain:

5.1 ‘P’ V, artinya keluarkan jarak jalur terpendek dari vertex sumber ke vertex V.
5.2 ‘I’ U V W, artinya sisipkan edge yang menghubungkan vertex sumber U dan vertex tujuan V dengan bobot edge W.
5.3 ‘R’ U V, artinya hapus edge yang menghubungkan vertex sumber U dan vertex tujuan V.

6. Masukan berakhir jika N = M = 0

Berikut merupakan format keluaran dari permasalahan tersebut:

1. Untuk setiap kasus uji, format keluaran didefinisikan pada poin 2.
2. Keluarkan K buah integer masing-masing dalam 1 baris yang merepresentasikan jarak jalur terpendek ke sebuah vertex tujuan, dimana K adalah banyaknya operasi ‘P’ pada masukan. Jika vertex tujuan tidak terhubung dengan vertex sumber, maka keluarkan -1.

Berikut merupakan batasan dari permasalahan tersebut:

1. $2 \leq N \leq 1.000$
2. $1 \leq M \leq 10.000$
3. $1 \leq U, V \leq N$
4. $1 \leq W \leq 100$
5. $1 \leq Q \leq 1.000$
6. Lingkungan penilaian Intel Pentium III 733 MHz.
7. Batasan waktu 1 detik.
8. Batasan memori 256 MB.

Contoh masukan dan keluaran dari permasalahan tersebut ditunjukkan dalam **Gambar 2.*4***. Ilustrasi graph dari permasalahan tersebut ditunjukkan dalam **Gambar 2.*5***.

Example

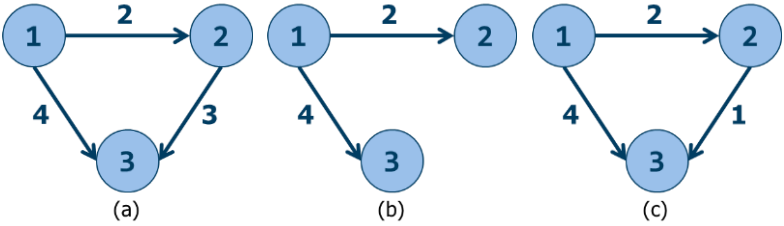
```
Input:

3 3
1 2 2
2 3 3
1 3 4
5
P 3
R 2 3
P 3
I 2 3 1
P 3

Output:

4
4
3
```

Gambar 2.4 Contoh Masukan dan Keluaran Permasalahan Help Your Commander



Gambar 2.5 Ilustrasi Graph Permasalahan Help Your Commander, Graph Sebelum Dilakukan Modifikasi (a), Graph Setelah Dilakukan Penghapusan Edge (b), Graph Setelah Dilakukan Penyisipan Edge (c)

Pada **Gambar 2.5** dapat dilihat ilustrasi graph hasil modifikasi yang dilakukan. Sebelum dilakukan modifikasi, jarak jalur terpendek ke vertex 3 adalah 4. Setelah dilakukan modifikasi penghapusan edge $(2, 3)$, jarak jalur terpendek ke vertex 3 adalah 4. Setelah dilakukan modifikasi penyisipan edge $(2, 3)$ dengan bobot edge 1, jarak jalur terpendek ke vertex 3 adalah 3.

(Halaman ini sengaja dikosongkan)

BAB III
DESAIN

Pada bab ini penulis menjelaskan tentang desain algoritma serta struktur data yang digunakan dalam Tugas Akhir.

3.1 Deskripsi Umum Sistem

Sistem akan menerima input berupa banyak vertex, banyak edge, dan deskripsi edge yaitu vertex sumber, vertex tujuan, dan bobot edge. Setelah itu dilakukan *preprocess* menggunakan algoritma Dijkstra. Kemudian sistem menerima input berupa banyak operasi.

Terdapat 3 jenis operasi yang dapat dilakukan yaitu:

1. Kueri jarak jalur terpendek dengan memberikan sebuah vertex tujuan.
2. Penyisipan edge pada graph dengan memberikan deskripsi edge yang akan disisipkan yaitu vertex sumber, vertex tujuan, dan bobot edge.
3. Penghapusan edge pada graph dengan memberikan deskripsi edge yang akan dihapus yaitu vertex sumber dan vertex tujuan.

Untuk setiap modifikasi pada graph, dilakukan insert atau delete menggunakan algoritma Dijkstra Dinamis. Untuk setiap kueri jarak jalur terpendek, dikeluarkan jarak jalur terpendek dari vertex sumber ke vertex tujuan yang diberikan. Pseudocode Fungsi Main ditunjukkan dalam **Gambar 3.*1*** dan **Gambar 3.*2***.

```
Main()
1.  input number of vertex, number of edge
2.  input source vertex, destination vertex, weight of
    each edge
3.  Preprocess()
4.  input number of operation
5.  while number of oepration > 0
6.      input type of operation
```

Gambar 3.1 Pseudocode Fungsi Main (1)

```
7.      if type of operation is ‘query’
8.          input destination vertex
9.          output distance to destination vertex from
    source vertex
10.     else if type of operation is ‘insert’
11.         input source vertex, destination vertex,
    weight
12.         Insert(source vertex, destination vertex,
    weight)
13.     else if type of operation is ‘delete’
14.         input source vertex, destination vertex
15.         Delete(source vertex, destination vertex)
16.     decrement number of operation
```

Gambar 3.2 Pseudocode Fungsi Main (2)

3.2 Desain Algoritma

Sistem terdiri dari 3 fungsi utama yaitu Preprocess, Insert, dan Delete. Pada subbab ini akan dijelaskan desain algoritma dari masing-masing fungsi utama tersebut.

3.2.1 Desain Fungsi Preprocess

Fungsi Preprocess dilakukan menggunakan algoritma Dijkstra seperti yang dijelaskan dalam subbab 2.5. Fungsi ini digunakan untuk menginisialisasi jarak jalur terpendek ke setiap vertex tujuan dan membentuk shortest path tree dengan root sebuah vertex sumber. Selain itu juga dilakukan inisialisasi struktur data yang digunakan dalam algoritma Dijkstra Dinamis. Pseudocode Fungsi Preprocess ditunjukkan dalam **Gambar 3.*3*** dan **Gambar 3.*4***.

```
Preprocess()
1.  for each vertex v ∈ V
2.      parent(v) = ∅;
3.      d(v) = ∞
4.  d(s) = 0;
5.  D = ∅;
6.  Enqueue(D,(s, d(s)));
7.  while D ≠ ∅
8.      (v, d(v)) = ExtractMin(D);
```

Gambar 3.3 Pseudocode Fungsi Preprocess (1)

```
9.      for each edge (v,u)
10.         if d(v) + w_{v,u} < d(u)
11.             parent(u) = v;
12.             d(u) = d(v) + w_{v,u};
13.             if u ∉ D
14.                 Enqueue(D,(u, d(u)));
15.             else
16.                 DecreaseKey(D,(u, d(u)));
17. for each vertex e ∈ E
18.     SetOwner(e)
```

Gambar 3.4 Pseudocode Fungsi Preprocess (2)

Baris 1 hingga baris 3 melakukan inisialisasi parent dan jarak untuk setiap vertex pada graph. Baris 4 melakukan inisialisasi jarak ke vertex sumber s. Baris 5 dan baris 6 melakukan inisialisasi *priority queue* D. Baris 8 melakukan *extract min* untuk mendapatkan vertex v dengan *priority* jarak minimum, selama *priority queue* D tidak kosong (baris 7). Baris 9 melakukan scanning pada *outedge* dari vertex v. Jika jarak ke vertex u melalui vertex v lebih kecil dari jarak ke vertex u saat ini (baris 10), maka dilakukan update parent dan jarak dari vertex u (baris 11 dan baris 12). Setelah itu dilakukan *enqueue* (baris 14) jika vertex u tidak terdapat dalam *priority queue* D (baris 13) atau *decrease key* (baris 16) jika vertex u terdapat dalam *priority queue* D (baris 15). Baris 17 dan baris 18 melakukan inisialisasi *owner* untuk setiap edge pada graph.

3.2.2 Desain Fungsi Insert

Fungsi Insert dilakukan menggunakan algoritma penyisipan edge seperti yang dijelaskan pada subbab 2.6.1. Fungsi ini digunakan untuk mempertahankan jarak jalur terpendek, shortest path tree, dan struktur data yang digunakan setelah dilakukan penyisipan edge. Pseudocode Fungsi Insert ditunjukkan dalam **Gambar 3.5**.

```
/***
 * u is the source vertex
 * v is the destination vertex
 * d is the weight of the edge
 */
Insert(u,v,d)
```

```
1.  e = InsertEdge(G,u,v,d);
2.  SetOwner(e);
3.  if d(u) + w_{u,v} ≥ d(v) or (u and v are not connected to s)
4.      return;
5.  parent(v) = u;
6.  d'(v) = d(u) + w_{u,v};
7.  color(v) = red;
8.  C = ∅;
9.  Enqueue(C,(v,d'(v)));
10. while C ≠ ∅
11.     (v,d'(v)) = ExtractMin(C);
12.     for each edge (v,u) ∈ owned(v) or
            ((v,u) ∈ not_owned(v) and d'(v) < b_level_v(u))
13.         if d'(v) + w_{v,u} < d'(u)
14.             parent(u) = v;
15.             d'(u) = d'(v) + w_{v,u};
16.             color(u) = red;
17.             if u ∉ C
18.                 Enqueue(C,(u,d'(u)));
19.             else
20.                 DecreaseKey(C,(u,d'(u)));
21. for each red vertex v
22.     for each edge e ∈ owned(v)
23.         if color(u) ≠ red
24.             ChangeOwner(e);
25.         else
26.             UpdateLevel(e);
27. reset all vertices color to white
```

Gambar 3.5 Pseudocode Fungsi Insert

Baris 1 melakukan insert edge pada graph. Baris 2 melakukan pemilihan owner dari edge (u, v) dan melakukan update struktur data yang digunakan pada vertex u dan vertex v. Baris 4 melakukan return jika penyisipan edge (u, v) tidak menyebabkan jarak ke

vertex v berkurang atau vertex u dan vertex v tidak terhubung dengan vertex sumber s (baris 3). Baris 5 dan baris 6 melakukan update parent dan jarak dari vertex v. Baris 7 melakukan update color dari vertex v karena vertex v mengalami perubahan jarak jalur terpendek dari vertex sumber s. Baris 8 dan baris 9 melakukan inisialisasi *priority queue* C.

Baris 11 melakukan *extract min* untuk mendapatkan vertex v dengan *priority* jarak minimum, selama *priority queue* C tidak kosong (baris 10). Baris 12 melakukan scanning pada *outedge* dari vertex v yang berada dalam himpunan $owned(v)$ dan himpunan $not\_owned(v)$ jika jarak jalur terpendek ke vertex v setelah dilakukan penyisipan edge lebih kecil dari $b\_level_v(u)$. Jika jarak ke vertex u melalui vertex v lebih kecil dari jarak ke vertex u saat ini (baris 13), maka dilakukan update parent dan jarak dari vertex u (baris 14 dan baris 15), kemudian dilakukan update color dari vertex u (baris 16) karena vertex u mengalami perubahan jarak jalur terpendek dari vertex sumber s. Setelah itu dilakukan *enqueue* (baris 18) jika vertex u tidak terdapat dalam *priority queue* C (baris 17) atau *decrease key* (baris 20) jika vertex u terdapat dalam *priority queue* C (baris 19).

Untuk setiap vertex v dimana $color(v) = red$ (baris 21) dan setiap edge yang dimiliki oleh vertex v (baris 22) dilakukan perubahan pemilik edge (baris 24) jika $color(u) \neq red$ (baris 23) atau dilakukan update struktur data yang digunakan (baris 26) jika $color(u) = red$ (baris 25). Setelah itu untuk setiap vertex v pada graph $color(v) = white$ (baris 27).

3.2.3 Desain Fungsi Delete

Fungsi Delete dilakukan menggunakan algoritma penghapusan edge seperti yang dijelaskan pada subbab 2.6.2. Fungsi ini digunakan untuk mempertahankan jarak jalur terpendek, shortest path tree, dan struktur data yang digunakan setelah dilakukan penghapusan edge. Pseudocode Fungsi Delete ditunjukkan dalam **Gambar 3.*6*** dan **Gambar 3.*7***.

```
/***
 * u is the source vertex
 * v is the destination vertex
 */
Delete(u,v)
```

```
1.  e = DeleteEdge(G,u,v);
2.  UnsetOwner(e);
3.  if parent(v) ≠ u or (u and v are not connected to s)
4.      return;
5.  M = ∅;
6.  Enqueue(M,(v,d(v)));
7.  while M ≠ ∅
8.      (v,d(v)) = ExtractMin(M);
9.      (b,l) = BestNonRedNeighbor(v);
10.     if d(v) = l
11.         parent(v) = b;
12.     else
13.         color(v) = red;
14.         for each vertex u ∈ children(v)
15.             Enqueue(M,(u,d(u)));
16. Q = ∅;
17. for each red vertex v
18.     (b,l) = BestNonRedNeighbor(v);
19.     if b = ∅
20.         parent(v) = ∅;
21.         d'(v) = ∞;
22.     else
23.         parent(v) = b;
24.         d'(v) = l;
25.         Enqueue(Q,(v,d'(v)));
26. while Q ≠ ∅
27.     (v,d'(v)) = ExtractMin(Q);
28.     for each edge (v,u)
29.         if color(u) = red and d'(v) + w_{v,u} < d(u)
30.             parent(u) = v;
31.             d'(u) = d'(v) + w_{v,u};
32.             if u ∉ Q
33.                 Enqueue(Q,(u,d'(u)));
34.             else
35.                 DecreaseKey(Q,(u,d'(u)));
```

Gambar 3.6 Pseudocode Fungsi Delete (1)

```
36. for each red vertex v
37.     for each edge (u,v) ∈ owned(v) or (v,u) ∈ owned(v)
38.         if color(u) ≠ red
39.             ChangeOwner(u,v);
40.         else
41.             UpdateLevel(u,v);
42. reset all vertices color to white
```

Gambar 3.7 Pseudocode Fungsi Delete (2)

Baris 1 melakukan delete edge pada graph. Baris 2 melakukan update struktur data yang digunakan pada vertex u dan vertex v. Baris 4 melakukan return jika penghapusan edge (u, v) tidak menyebabkan jarak ke vertex v bertambah atau vertex u dan vertex v tidak terhubung dengan vertex sumber s (baris 3). Baris 5 dan baris 6 melakukan inisialisasi *priority queue* M.

Baris 8 melakukan *extract min* untuk mendapatkan vertex v dengan *priority* jarak minimum, selama *priority queue* M tidak kosong (baris 7). Baris 9 melakukan pencarian ***nonred neighbor*** terbaik dari vertex v. Jika terdapat ***nonred neighbor*** terbaik vertex b sehingga vertex v tidak mengalami perubahan jarak jalur terpendek dari vertex sumber s (baris 10), maka dilakukan melakukan update parent dari vertex v (baris 11). Jika sebaliknya (baris 12), maka dilakukan update color dari vertex v (baris 13) karena vertex v mengalami perubahan jarak jalur terpendek dari vertex sumber s, kemudian dilakukan *enqueue* pada *priority queue* M (baris 15) untuk setiap vertex u yang menjadi child dari vertex v dalam shortest path tree (baris 14). Baris 16 melakukan inisialisasi *priority queue* Q.

Untuk setiap vertex v dimana $color(v) = red$ (baris 17) dilakukan pencarian ***nonred neighbor*** terbaik (baris 18). Jika tidak terdapat ***nonred neighbor*** terbaik vertex b (baris 19), maka dilakukan update parent dan jarak dari vertex v (baris 20 dan baris 21). Jika sebaliknya (baris 22), maka dilakukan update parent dan jarak dari vertex v (baris 23 dan baris 24), kemudian dilakukan *enqueue* pada *priority queue* Q (baris 25). Baris 27 melakukan *extract min* untuk mendapatkan vertex v dengan *priority* jarak minimum, selama *priority queue* Q tidak kosong (baris 26).

Baris 28 melakukan scanning pada *outedge* dari vertex v. Jika $color(u) = red$ dan jarak ke vertex u melalui vertex v lebih kecil dari jarak ke vertex u saat ini (baris 29), maka dilakukan update parent dan jarak dari vertex u (baris 30 dan baris 31). Setelah itu dilakukan *enqueue* (baris 33) jika vertex u tidak terdapat dalam *priority queue* Q (baris 32) atau *decrease key* (baris 35) jika vertex u terdapat dalam *priority queue* C (baris 34).

Untuk setiap vertex v dimana $color(v) = red$ (baris 36) dan setiap edge yang dimiliki oleh vertex v (baris 37) dilakukan perubahan pemilik edge (baris 39) jika $color(u) \neq red$ (baris 38) atau dilakukan update struktur data yang digunakan (baris 41) jika $color(u) = red$ (baris 40). Setelah itu untuk setiap vertex v pada graph $color(v) = white$ (baris 42).

Fungsi BestNonRedNeighbor digunakan untuk mencari ***nonred neighbor*** terbaik dari vertex v sebagai vertex yang dilalui dalam jalur terpendek alternatif ke vertex v setelah dilakukan penghapusan edge. Pseudocode Fungsi BestNonRedNeighbor ditunjukkan dalam **Gambar 3.*8***.

```
/***
 * v is the vertex to be found its BestNonRedNeighbor
 */
BestNonRedNeighbor(v)
```

```
1.  b = ∅
2.  l = ∞
3.  for each edge (u,v) ∈ owned(v) or
        ((u,v) ∈ not_owned(v) and l > f_level_v(u))
4.          if color(u) ≠ red and d(u) + w_u,v < l
5.              b = u;
6.              l = d(u) + w_u,v;
7.              if d(v) = l
8.                  return (b,l);
9.  return (b,l);
```

Gambar 3.8 Pseudocode Fungsi BestNonRedNeighbor

Baris 1 dan baris 2 melakukan inisialisasi vertex dan jarak yang menjadi ***nonred neighbor*** terbaik dari vertex v. Baris 3 melakukan scanning pada *inedge* dari vertex v yang berada dalam himpunan $owned(v)$ dan himpunan $not\_owned(v)$ jika jarak saat ini lebih

besar dari $f\_level_v(u)$. Jika $color(u) \neq red$ dan jarak jalur terpendek ke vertex v melalui vertex u lebih kecil dari jarak saat ini (baris 4), maka dilakukan update vertex dan jarak yang menjadi ***nonred neighbor*** terbaik dari vertex v (baris 5 dan baris 6). Jika terdapat vertex u sehingga vertex v tidak mengalami perubahan jarak jalur terpendek dari vertex sumber s (baris 7), maka dilakukan return vertex dan jarak yang menjadi ***nonred neighbor*** terbaik dari vertex v.

3.3 Desain Struktur Data

Struktur data yang digunakan harus efisien agar algoritma Dijkstra Dinamis dapat berjalan sesuai dengan kompleksitas *time* yang diberikan. Pada algoritma Dijkstra Dinamis dibutuhkan *list* yang mendukung *random access* artinya setiap elemen yang terdapat dalam *list* dapat diakses secara efisien seperti setiap elemen yang lain tanpa terpengaruh dengan banyaknya elemen yang terdapat dalam *list* tersebut. Namun *list* tersebut juga harus mendukung penyisipan atau penghapusan elemen secara efisien. Terdapat 2 jenis implementasi *list* yang sering digunakan antara lain:

1. *Linked List*
   Linked List mendukung penyisipan elemen dan penghapusan elemen di posisi manapun secara efisien tanpa harus mengatur ulang posisi elemen yang lain. Namun *Linked List* tidak mendukung *random access*, karena *Linked List* hanya mendukung *sequential access* sehingga setiap elemen harus diakses berdasarkan urutan tertentu.
2. *Array List*
   Array List mendukung random access, penyisipan elemen dan penghapusan elemen di posisi akhir secara efisien. Namun *Array List* tidak mendukung penyisipan elemen dan penghapusan elemen di posisi awal atau di posisi tertentu secara efisien, karena mengharuskan pengaturan ulang posisi elemen yang lain.

Tidak ada implementasi *list* yang mendukung random access, penyisipan elemen, dan penghapusan elemen secara efisien. Oleh karena itu dilakukan modifikasi pada *Array List* untuk menangani hal tersebut. Setiap elemen yang terdapat dalam *list* mempunyai indeks yaitu posisi elemen tersebut dalam *list*. Jika dilakukan penyisipan elemen, maka elemen tersebut disisipkan di posisi akhir dan indeks elemen tersebut diisi dengan posisi elemen dalam *list*. Pseudocode Fungsi InsertCustomList ditunjukkan dalam **Gambar 3.*9***. Jika dilakukan penghapusan elemen, maka elemen di posisi akhir ditukar dengan elemen di posisi yang akan dihapus (didapatkan dari indeks elemen tersebut) dan indeks elemen yang berada di posisi akhir diisi dengan indeks elemen yang akan dihapus, kemudian dilakukan penghapusan elemen pada posisi akhir. Pseudocode Fungsi DeleteCustomList ditunjukkan dalam **Gambar 3.*10***.

```
/***
 * e is the element to be inserted
 */
InsertCustomList(e)
```

```
1.  insert e to the end of the list
2.  increment element count
3.  set index of e with its position in the list
```

Gambar 3.9 Pseudocode Fungsi InsertCustomList

```
/***
 * e is the element to be deleted
 */
DeleteCustomList(e)
```

```
1.  if e is not at the end of the list
2.      l = element at the end of the list
3.      move l to the position of e
4.      set index of l with its position in the list
5.  delete element at the end of the list
6.  decrement element count
```

Gambar 3.10 Pseudocode Fungsi DeleteCustomList

Untuk menyimpan $children(v)$ dari sebuah vertex $v \in V_T$ dapat digunakan *Custom List*. Selain itu untuk menyimpan himpunan $owned(v)$ dari sebuah vertex $v \in V$ juga dapat

digunakan *Custom List*. Fungsi Insert hanya melakukan scanning pada *outedge* sedangkan Fungsi Delete hanya melakukan scanning pada *inedge*, sehingga *inedge* (u, v) dan *outedge* (v, u) disimpan dalam 2 *Custom List* agar lebih efisien dalam melakukan scanning.

Pada algoritma Dijkstra Dinamis dibutuhkan struktur data untuk menyimpan elemen dalam urutan tertentu sesuai dengan *priority* elemen tersebut. Terdapat 2 jenis implementasi struktur data yang sering digunakan untuk menangani hal tersebut antara lain:

1. *Priority Queue*
   Priority Queue mendukung penyimpanan elemen berdasarkan urutan *priority* tertentu. *Priority Queue* mendukung akses pada elemen dengan *priority* minimum atau maksimum. Untuk mengakses elemen dengan *priority* minimum atau maksimum berikutnya, harus dilakukan penghapusan elemen.
2. *Set*
   Set mendukung penyimpanan elemen berdasarkan urutan *priority* tertentu. *Set* mendukung akses pada elemen dengan dengan urutan *priority* tertentu.

Fungsi Delete membutuhkan scanning himpunan $not\_owned(v)$ dari sebuah vertex $v \in V$ lebih dari 1 kali (untuk mencari ***nonred neighbor*** terbaik), sehingga struktur data tersebut harus mendukung pengaksesan elemen berdasarkan urutan tertentu tanpa mengubah isi dari struktur data tersebut. Oleh karena itu untuk menyimpan himpunan $not\_owned(v)$ dari sebuah vertex $v \in V$ digunakan *set* untuk menggantikan *priority queue*.

(Halaman ini sengaja dikosongkan)

BAB IV
IMPLEMENTASI

Pada bab ini penulis menjelaskan tentang implementasi berdasarkan desain algoritma serta struktur data yang telah dilakukan.

4.1 Lingkungan Implementasi

Lingkungan implementasi yang akan digunakan sebagai berikut:

1. Perangkat Keras
   Processor Intel® Core™ i7-4702MQ CPU @ 2.20GHz
   RAM 8.00 GB
   64-bit Operating System, x64-based processor
2. Perangkat Lunak
   Operating System Windows 8.1
   Integrated Development Environment Code::Blocks 13.12

4.2 Implementasi Struct Neighbor

Struct Neighbor digunakan untuk merepresentasikan neighbor. Terdapat variabel vertex neighbor dan jarak ke vertex tersebut. Kode Sumber Implementasi Struct Neighbor ditunjukkan dalam **Kode Sumber 4.*1***.

```
1.  struct Neighbor {
2.      Vertex *vertex;
3.      int distance;
4.      Neighbor(Vertex    *vertex,    int    distance):
    vertex(vertex), distance(distance) {}
5.      bool operator<(const Neighbor &other) const {
6.          return this->distance < other.distance;
7.      }
8.  };
```

Kode Sumber 4.1 Implementasi Struct Neighbor

4.3 Implementasi Struct Edge

Struct Edge digunakan untuk merepresentasikan edge pada graph. Terdapat variabel vertex sumber, vertex tujuan, vertex owner dari edge tersebut, dan jarak edge. Selain itu juga terdapat beberapa variabel lain untuk implementasi struktur data yang digunakan dalam algoritma Dijkstra Dinamis. Kode Sumber Implementasi Struct Edge ditunjukkan dalam **Kode Sumber 4.*2***.

```
1.  struct Edge {
2.      Vertex *source, *destination, *owner;
3.      int distance, color, index, ownedIndex,
    notOwnedIndex, neighborIndex;
4.      Edge(Vertex *source, Vertex *destination, int
    distance, int index):
5.          source(source), destination(destination),
    owner(NULL), distance(distance), color(-1),
6.          index(index), ownedIndex(-1), notOwnedIndex(-
    INF), neighborIndex(-1) {}
7.  };
```

Kode Sumber 4.2 Implementasi Struct Edge

4.4 Implementasi Struct Vertex

Struct vertex digunakan untuk merepresentasikan vertex pada graph. Terdapat variabel parent dari vertex tersebut dan jarak dari vertex sumber ke vertex tersebut. Selain itu juga terdapat beberapa variabel lain untuk implementasi struktur data yang digunakan dalam algoritma Dijkstra Dinamis. Kode Sumber Implementasi Struct Vertex ditunjukkan dalam **Kode Sumber 4.*3***, **Kode Sumber 4.*4***, **Kode Sumber 4.*5***, **Kode Sumber 4.*6***, **Kode Sumber 4.*7***, **Kode Sumber 4.*8***, **Kode Sumber 4.*9***, dan **Kode Sumber 4.*10***.

```
1.  struct Vertex {
2.      Vertex *parent;
3.      int distance, color, index, childrenIndex;
4.      Vertex(int index): parent(NULL), distance(INF),
    color(-1), index(index), childrenIndex(-1) {}
```

Kode Sumber 4.3 Implementasi Struct Vertex (1)

Terdapat variabel neighbor yang menyimpan edge dimana vertex tersebut menjadi vertex sumber. Fungsi insertNeighbor dan eraseNeighbor digunakan untuk menyisipkan dan menghapus edge dalam variabel neighbor. Fungsi tersebut diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*9*** dan **Gambar 3.*10***.

```
5.      vector < Edge * > neighbor;
6.      bool insertNeighbor(Edge *edge) {
7.          if   (edge->source   !=   this   ||   edge-
    >neighborIndex != -1)
8.              return false;
9.          edge->neighborIndex = this->neighbor.size();
10.         this->neighbor.push_back(edge);
11.         return true;
12.     }
13.     bool eraseNeighbor(Edge *edge) {
14.         if   (edge->source   !=   this   ||   edge-
    >neighborIndex == -1)
15.             return false;
16.         if    (edge->neighborIndex    <    (int)this-
    >neighbor.size() - 1) {
17.             this->neighbor[this->neighbor.size()    -
    1]->neighborIndex = edge->neighborIndex;
18.             this->neighbor[edge->neighborIndex]     =
    this->neighbor[this->neighbor.size() - 1];
19.         }
20.         edge->neighborIndex = -1;
21.         this->neighbor.pop_back();
22.         return true;
23.     }
```

Kode Sumber 4.4 Implementasi Struct Vertex (2)

Terdapat variabel children yang menyimpan vertex yang menjadi child dari vertex tersebut dalam shortest path tree. Fungsi setParent digunakan untuk mengisi variabel parent dengan sebuah vertex parent serta menyisipkan vertex tersebut sebagai child dari vertex parent. Fungsi insertChild dan eraseChild digunakan untuk menyisipkan dan menghapus vertex dalam variabel children. Fungsi tersebut diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*9*** dan **Gambar 3.*10***.

```
24.     vector < Vertex * > children;
25.     bool setParent(Vertex *vertex) {
26.         if (vertex != NULL)
27.             return vertex->insertChild(this);
28.         if (this->parent != NULL)
29.             return this->parent->eraseChild(this);
30.         return true;
31.     }
32.     bool insertChild(Vertex *vertex) {
33.         if (vertex->parent == this)
34.             return false;
35.         if (vertex->parent != NULL)
36.             vertex->parent->eraseChild(vertex);
37.         vertex->parent = this;
38.         vertex->childrenIndex         =         this-
    >children.size();
39.         this->children.push_back(vertex);
40.         return true;
41.     }
42.     bool eraseChild(Vertex *vertex) {
43.         if (vertex->parent != this)
44.             return false;
45.         if   (vertex->childrenIndex   <   (int)this-
    >children.size() - 1) {
46.             this->children[this->children.size()    -
    1]->childrenIndex = vertex->childrenIndex;
47.             this->children[vertex->childrenIndex]   =
    this->children[this->children.size() - 1];
48.         }
49.         vertex->parent = NULL;
50.         vertex->childrenIndex = -1;
51.         this->children.pop_back();
52.         return true;
53.     }
```

Kode Sumber 4.5 Implementasi Struct Vertex (3)

Fungsi getNumber digunakan untuk menentukan nomor *list* atau *set* yang digunakan untuk menyimpan himpunan edge (*owned* atau *not_owned*) yang *berincident* dengan vertex tersebut. Setiap *inedge* disimpan dalam *list* atau *set* nomor 0 dan setiap *outedge* disimpan dalam *list* atau *set* nomor 1.

Fungsi getLevel digunakan untuk menentukan *priority* dari edge yang disimpan dalam himpunan *not_owned*. Fungsi tersebut menentukan *level* dari sebuah edge sesuai dengan Persamaan **(2.1)** dan Persamaan **(2.2)**.

```
54.     int getNumber(Edge *edge) {
55.         if (this == edge->source)
56.             return 1;
57.         if (this == edge->destination)
58.             return 0;
59.         return -1;
60.     }
61.     int getLevel(Edge *edge) {
62.         int number = this->getNumber(edge);
63.         assert(edge->owner != this && number != -1);
64.         int level = edge->source->distance + edge-
    >distance;
65.         if (number == 1)
66.             level = edge->destination->distance -
    edge->distance;
67.         return level;
68.     }
```

Kode Sumber 4.6 Implementasi Struct Vertex (4)

Terdapat variabel owned yang menyimpan edge yang dimiliki oleh vertex tersebut. Fungsi insertOwned dan eraseOwned digunakan untuk menyisipkan dan menghapus edge dalam variabel owned. Fungsi tersebut diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*9*** dan **Gambar 3.*10***.

```
69.     vector < Edge * > owned[2];
70.     bool insertOwned(Edge *edge) {
71.         int number = this->getNumber(edge);
72.         if (edge->owner != NULL || number == -1)
73.             return false;
74.         edge->owner = this;
75.         edge->ownedIndex            =            this-
    >owned[number].size();
76.         this->owned[number].push_back(edge);
77.         return true;
78.     }
```

Kode Sumber 4.7 Implementasi Struct Vertex (5)

```
79.     bool eraseOwned(Edge *edge) {
80.         int number = this->getNumber(edge);
81.         if (edge->owner != this || number == -1)
82.             return false;
83.         if     (edge->ownedIndex     <     (int)this-
    >owned[number].size() - 1) {
84.             this->owned[number][this-
    >owned[number].size()   -   1]->ownedIndex   =   edge-
    >ownedIndex;
85.             this->owned[number][edge->ownedIndex]   =
    this->owned[number][this->owned[number].size() - 1];
86.         }
87.         edge->owner = NULL;
88.         edge->ownedIndex = -1;
89.         this->owned[number].pop_back();
90.         return true;
91.     }
```

Kode Sumber 4.8 Implementasi Struct Vertex (6)

Terdapat variabel notOwned yang menyimpan edge yang *berincident* dengan vertex tersebut tetapi tidak dimiliki oleh vertex tersebut. Fungsi pushNotOwned digunakan untuk menyisipkan edge dalam variabel notOwned. Fungsi topNotOwned digunakan untuk mencari edge dengan *priority* minimum atau maximum dalam variabel notOwned. Fungsi popNotOwned digunakan untuk menghapus edge dengan *priority* minimum atau maximum dalam variabel notOwned. Fungsi eraseNotOwned digunakan untuk menghapus edge dalam variabel notOwned.

```
92.     set < pair < int, Edge * > > notOwnedSet[2];
93.     bool pushNotOwned(Edge *edge) {
94.         int number = this->getNumber(edge);
95.         if (edge->owner == this || number == -1)
96.             return false;
97.         if (edge->notOwnedIndex != -INF)
98.             this->eraseNotOwned(edge);
99.         int priority = this->getLevel(edge);
```

Kode Sumber 4.9 Implementasi Struct Vertex (7)

```
100.        edge->notOwnedIndex = priority;
101.        if (number == 1) priority = -priority;
102.        this->notOwnedSet[number].insert(
    make_pair(priority, edge));
103.        return true;
104.    }
105.    int topNotOwned(int number, Edge *&edge) {
106.        if (this->notOwnedSet[number].empty()) {
107.            edge = NULL;
108.            if (number == 1) return -INF;
109.            return INF;
110.        }
111.        set < pair < int, Edge * > > :: iterator it =
    this->notOwnedSet[number].begin();
112.        edge = it->second;
113.        int priority = it->first;
114.        if (number == 1) priority = -priority;
115.        assert(edge->owner != this && number == this-
    >getNumber(edge) && edge->notOwnedIndex == priority);
116.        return priority;
117.    }
118.    bool popNotOwned(int number) {
119.        if (this->notOwnedSet[number].empty())
120.            return false;
121.        set < pair < int, Edge * > > :: iterator it =
    this->notOwnedSet[number].begin();
122.        this->notOwnedSet[number].erase(it);
123.        return true;
124.    }
125.    bool eraseNotOwned(Edge *edge) {
126.        int number = this->getNumber(edge);
127.        if (edge->owner == this || number == -1 ||
    edge->notOwnedIndex == -INF) return false;
128.        int priority = edge->notOwnedIndex;
129.        edge->notOwnedIndex = -INF;
130.        if (number == 1) priority = -priority;
131.        this->notOwnedSet[number].erase(
    make_pair(priority, edge));
132.        return true;
133.    }
134.};
```

Kode Sumber 4.10 Implementasi Struct Vertex (8)

4.5 Implementasi Struct Graph

Struct Graph digunakan untuk merepresentasikan graph. Terdapat variabel numberVertex yang menyimpan banyak vertex pada graph, numberEdge yang menyimpan banyak edge pada graph, serta vertex dan edge yang menyimpan vertex dan edge pada graph. Kode Sumber Implementasi Struct Graph ditunjukkan dalam **Kode Sumber 4.*11***, **Kode Sumber 4.*12***, **Kode Sumber 4.*13***, **Kode Sumber 4.*14***, **Kode Sumber 4.*15***, **Kode Sumber 4.*16***, **Kode Sumber 4.*17***, dan **Kode Sumber 4.*18***.

```
1.  struct Graph {
2.      int color, numberVertex, numberEdge;
3.      Graph(): color(0), numberVertex(0), numberEdge(0)
    {}
4.      Vertex *vertex[1001];
5.      Edge *edge[11011];
```

Kode Sumber 4.11 Implementasi Struct Graph (1)

Terdapat variabel edgeMap yang menyimpan *key* yang dibentuk dari indeks vertex sumber dan vertex tujuan dan *value* indeks edge pada graph. Variabel ini digunakan untuk mencari indeks edge yang *berincident* dengan vertex sumber dan vertex tujuan dalam variabel edge. Fungsi insertEdge dan eraseEdge digunakan untuk menyisipkan dan menghapus edge dalam variabel edge. Fungsi tersebut diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*9*** dan **Gambar 3.*10***.

```
6.      map < int, int > edgeMap;
7.      Edge   *   insertEdge(Vertex   *source,   Vertex
    *destination, int distance) {
8.          this->numberEdge++;
9.          this-
    >edgeMap.insert(make_pair(Graph::getHash(source,
    destination), this->numberEdge - 1));
10.         Edge *edge = new Edge(source, destination,
    distance, this->numberEdge - 1);
11.         this->edge[this->numberEdge - 1] = edge;
12.         source->insertNeighbor(edge);
13.         return edge;
14.     }
```

Kode Sumber 4.12 Implementasi Struct Graph (2)

```
15.     Edge    *    eraseEdge(Vertex    *source,    Vertex
    *destination) {
16.         map < int, int > :: iterator it = this-
    >edgeMap.find(Graph::getHash(source, destination));
17.         if (it == this->edgeMap.end())
18.             return NULL;
19.         int index = it->second;
20.         this->edgeMap.erase(it);
21.         Edge *edge = this->edge[index];
22.         source->eraseNeighbor(edge);
23.         if (index < this->numberEdge - 1) {
24.             it                 =                 this-
    >edgeMap.find(Graph::getHash(this->edge[this-
    >numberEdge - 1]->source, this->edge[this->numberEdge
    - 1]->destination));
25.             it->second = index;
26.             this->edge[this->numberEdge - 1]->index =
    index;
27.             this->edge[index]    =    this->edge[this-
    >numberEdge - 1];
28.         }
29.         this->edge[this->numberEdge - 1] = NULL;
30.         this->numberEdge--;
31.         return edge;
32.     }
```

Kode Sumber 4.13 Implementasi Struct Graph (3)

Fungsi getHash digunakan untuk membentuk *key* yang digunakan dalam variabel edgeMap. Fungsi tersebut membentuk key yang unik untuk setiap pasangan vertex yang menjadi endpoint dari sebuah edge.

```
33.     static   int   getHash(Vertex   *source,   Vertex
    *destination) {
34.         return (source->index << 10) + destination-
    >index;
35.     }
```

Kode Sumber 4.14 Implementasi Struct Graph (4)

Fungsi setOwner digunakan untuk memilih *owner* dari sebuah edge. Fungsi ini memilih *owner* dengan membandingkan edge yang dimiliki oleh vertex sumber dan vertex tujuan. Jika banyak *inedge* dari vertex tujuan lebih kecil dari banyak *outedge* dari vertex sumber, maka dipilih vertex tujuan sebagai owner dari edge tersebut dan dilakukan penyisipan edge pada himpunan *owned* vertex tujuan dan himpunan *not_owned* vertex sumber. Jika sebaliknya, maka dipilih vertex sumber sebagai owner dari edge tersebut dan dilakukan penyisipan edge pada himpunan *owned* vertex sumber dan himpunan *not_owned* vertex tujuan.

Fungsi unsetOwner digunakan untuk menghapus *owner* dari sebuah edge. Jika *owner* dari edge tersebut adalah vertex sumber, maka dilakukan penghapusan edge pada himpunan *owned* vertex sumber dan himpunan *not_owned* vertex tujuan. Jika sebaliknya, maka dilakukan penghapusan edge pada himpunan *owned* vertex tujuan dan himpunan *not_owned* vertex sumber.

Fungsi changeOwner digunakan untuk mengubah owner dari sebuah edge. Jika *owner* dari edge tersebut adalah vertex sumber, maka dilakukan penghapusan edge pada himpunan *owned* vertex sumber dan himpunan *not_owned* vertex tujuan dan dilakukan penyisipan edge pada himpunan *owned* vertex tujuan dan himpunan *not_owned* vertex sumber. Jika sebaliknya, maka dilakukan penghapusan edge pada himpunan *owned* vertex tujuan dan himpunan *not_owned* vertex sumber dan dilakukan penyisipan edge pada himpunan *owned* vertex sumber dan himpunan *not_owned* vertex tujuan.

Fungsi updateLevel digunakan untuk melakukan update *priority* pada himpunan *not_owned*. Jika *owner* dari edge tersebut adalah vertex sumber, maka dilakukan update priority pada himpunan *not_owned* vertex tujuan. Jika sebaliknya, maka dilakukan update priority pada himpunan *not_owned* vertex sumber.

```
36.     static void setOwner(Edge *edge) {
37.         if    (edge->destination->owned[0].size()    <
    edge->source->owned[1].size()) {
38.             edge->destination->insertOwned(edge);
39.             edge->source->pushNotOwned(edge);
40.         }
41.         else {
42.             edge->source->insertOwned(edge);
43.             edge->destination->pushNotOwned(edge);
44.         }
45.     }
46.     static bool unsetOwner(Edge *edge) {
47.         if (edge->owner == edge->source) {
48.             edge->source->eraseOwned(edge);
49.             edge->destination->eraseNotOwned(edge);
50.         }
51.         else if (edge->owner == edge->destination) {
52.             edge->destination->eraseOwned(edge);
53.             edge->source->eraseNotOwned(edge);
54.         }
55.         else
56.             return false;
57.         return true;
58.     }
59.     static bool changeOwner(Edge *edge) {
60.         Vertex *oldOwner, *newOwner;
61.         oldOwner = edge->owner;
62.         if (oldOwner == edge->source)
63.             newOwner = edge->destination;
64.         else if (oldOwner == edge->destination)
65.             newOwner = edge->source;
66.         else
67.             return false;
68.         oldOwner->eraseOwned(edge);
69.         newOwner->eraseNotOwned(edge);
70.         newOwner->insertOwned(edge);
71.         oldOwner->pushNotOwned(edge);
72.         return true;
73.     }
```

Kode Sumber 4.15 Implementasi Struct Graph (5)

```
74.     static bool updateLevel(Edge *edge) {
75.         if (edge->owner == edge->source)
76.             edge->destination->pushNotOwned(edge);
77.         else if (edge->owner == edge->destination)
78.             edge->source->pushNotOwned(edge);
79.         else
80.             return false;
81.         return true;
82.     }
```

Kode Sumber 4.16 Implementasi Struct Graph (6)

Fungsi getBestNonRedNeighbor digunakan untuk mencari ***nonred neighbor*** terbaik dari sebuah vertex. Fungsi tersebut diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*8***.

```
83.     static   Neighbor   getBestNonRedNeighbor(Vertex
    *vertex) {
84.         Neighbor bestNonRedNeighbor(NULL, INF);
85.         for (vector < Edge * > :: iterator it = vertex-
    >owned[0].begin(); it != vertex->owned[0].end(); it++)
    {
86.             int distance = (*it)->source->distance +
    (*it)->distance;
87.             if ((*it)->source->color != graph->color
    && distance < bestNonRedNeighbor.distance) {
88.                 bestNonRedNeighbor.vertex  =  (*it)-
    >source;
89.                 bestNonRedNeighbor.distance         =
    distance;
90.                 if (distance == vertex->distance)
91.                     return bestNonRedNeighbor;
92.             }
93.         }
```

Kode Sumber 4.17 Implementasi Struct Graph (7)

```
94.         for (set < pair < int, Edge * > > :: iterator
    it = vertex->notOwnedSet[0].begin(); it != vertex-
    >notOwnedSet[0].end() && bestNonRedNeighbor.distance
    > it->first; it++) {
95.             int   distance   =   it->second->source-
    >distance + it->second->distance;
96.             if (it->second->source->color != graph-
    >color && distance < bestNonRedNeighbor.distance) {
97.                 bestNonRedNeighbor.vertex    =    it-
    >second->source;
98.                 bestNonRedNeighbor.distance         =
    distance;
99.                 if (distance == vertex->distance)
100.                     return bestNonRedNeighbor;
101.            }
102.        }
103.        return bestNonRedNeighbor;
104.    }
105.};
```

Kode Sumber 4.18 Implementasi Struct Graph (8)

4.6 Implementasi Fungsi Preprocess

Fungsi Preprocess diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*3*** dan **Gambar 3.*4***. Implementasi Fungsi Preprocess ditunjukkan dalam **Kode Sumber 4.*19*** dan **Kode Sumber 4.*20***.

```
1.  void preprocess() {
2.      priority_queue < Neighbor > DPQ;
3.      Vertex *source = graph->vertex[0];
4.      source->distance = 0;
5.      for                     (DPQ.push(Neighbor(source,
    0)); !DPQ.empty(); ) {
6.          Neighbor now = DPQ.top();
7.          DPQ.pop();
8.          now.distance = -now.distance;
9.          if (now.distance == now.vertex->distance)
```

Kode Sumber 4.19 Implementasi Fungsi Preprocess (1)

```
10.             for (vector < Edge * > :: iterator it =
    now.vertex->neighbor.begin();   it   !=   now.vertex-
    >neighbor.end(); it++) {
11.                 int distance = now.distance + (*it)-
    >distance;
12.                 if  (distance  <  (*it)->destination-
    >distance) {
13.                     (*it)->destination-
    >setParent(now.vertex);
14.                     (*it)->destination->distance    =
    distance;
15.                     DPQ.push(Neighbor((*it)-
    >destination, -distance));
16.                 }
17.             }
18.     }
19.     for (int i = 0; i < graph->numberEdge; i++)
20.         Graph::setOwner(graph->edge[i]);
21. }
```

Kode Sumber 4.20 Implementasi Fungsi Preprocess (2)

Fungsi Preprocess membutuhkan *decrease key* jika vertex terdapat dalam *priority queue* (baris 13 hingga baris 16 pada **Gambar 3.*4***). Namun *priority queue* pada Standard Template Library C++ tidak mendukung *decrease key*. Oleh karena itu dilakukan *enqueue* vertex pada *priority queue* (baris 15), tetapi hanya dilakukan *scanning* jika vertex tersebut belum pernah dikunjungi atau *priority* vertex tersebut pada *priority queue* sama dengan jarak jalur terpendek ke vertex tersebut (baris 9).

4.7 Implementasi Fungsi Update

Fungsi Update digunakan untuk update struktur data yang digunakan pada vertex yang mengalami perubahan jarak jalur terpendek setelah dilakukan penyisipan atau penghapusan edge (baris 21 hingga baris 26 pada **Gambar 3.*5*** dan baris 36 hingga baris 41 pada **Gambar 3.*7***). Implementasi Fungsi Update ditunjukkan dalam **Kode Sumber 4.*21***.

```
1.  void doUpdate(vector < Vertex * > &vertexVec) {
2.      for (vector < Vertex * > :: iterator it =
    vertexVec.begin(); it != vertexVec.end(); it++)
3.          for (int i = 0; i < 2; i++)
4.              for (int j = 0; j < (int)(*it)-
    >owned[i].size(); j++) {
5.                  Edge *edge = (*it)->owned[i][j];
6.                  int color = edge->source->color;
7.                  if (i == 1)
8.                      color = edge->destination->color;
9.                  if (color != graph->color) {
10.                     Graph::changeOwner(edge);
11.                     j--;
12.                 }
13.                 else
14.                     Graph::updateLevel(edge);
15.             }
16. }
```

Kode Sumber 4.21 Implementasi Fungsi Update

Fungsi Update membutuhkan *color* dari vertex *neighbor* dalam melakukan update struktur data yang digunakan (baris 23 hingga baris 26 pada **Gambar 3.*5*** dan baris 38 hingga baris 41 pada **Gambar 3.*7***). Agar efisien dalam melakukan *Graph Coloring*, maka digunakan indeks (variabel *color* pada graph) yang akan bertambah setiap dilakukan penyisipan atau penghapusan edge. Jika variabel *color* pada vertex sama dengan indeks (variabel *color* pada graph), maka *color* dari vertex tersebut *red*. Jika sebaliknya, maka *color* dari vertex tersebut *white*.

4.8 Implementasi Fungsi Insert

Fungsi Insert diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*5***. Implementasi Fungsi Insert ditunjukkan dalam **Kode Sumber 4.*22*** dan **Kode Sumber 4.*23***.

```
1.  void doInsert(Vertex *source, Vertex *destination, int
    distance) {
2.      Graph::setOwner(graph->insertEdge(source,
    destination, distance));
3.      distance += source->distance;
4.      if (distance >= destination->distance)
5.          return;
6.      destination->setParent(source);
7.      destination->distance = distance;
8.      destination->color = graph->color;
9.      priority_queue < Neighbor > CPQ;
10.     vector < Vertex * > redVertex;
11.     redVertex.push_back(destination);
12.     for       (CPQ.push(Neighbor(destination,       -
    distance)); !CPQ.empty(); ) {
13.         Neighbor now = CPQ.top();
14.         CPQ.pop();
15.         now.distance = -now.distance;
16.         if (now.distance == now.vertex->distance) {
17.             for (vector < Edge * > :: iterator it =
    now.vertex->owned[1].begin();   it   !=   now.vertex-
    >owned[1].end(); it++) {
18.                 distance   =   now.distance   +   (*it)-
    >distance;
19.                 if  (distance  <  (*it)->destination-
    >distance) {
20.                     (*it)->destination-
    >setParent(now.vertex);
21.                     (*it)->destination->distance    =
    distance;
22.                     (*it)->destination->color       =
    graph->color;
23.                     redVertex.push_back((*it)-
    >destination);
24.                     CPQ.push(Neighbor((*it)-
    >destination, -distance));
25.                 }
26.             }
```

Kode Sumber 4.22 Implementasi Fungsi Insert (1)

```
27.             for (Edge *edge; now.distance < now.vertex->topNotOwned(1, edge); ) {
28.                 now.vertex->popNotOwned(1);
29.                 distance = now.distance + edge->distance;
30.                 if (distance < edge->destination->distance) {
31.                     edge->destination->setParent(now.vertex);
32.                     edge->destination->distance = distance;
33.                     edge->destination->color = graph->color;
34.                     redVertex.push_back(edge->destination);
35.                     CPQ.push(Neighbor(edge->destination, -distance));
36.                 }
37.             }
38.         }
39.     }
40.     doUpdate(redVertex);
41.     graph->color++;
42. }
```

Kode Sumber 4.23 Implementasi Fungsi Insert (2)

Fungsi Insert membutuhkan *decrease key* jika vertex terdapat dalam *priority queue* (baris 17 hingga baris 20 pada **Gambar 3.*5***). Namun *priority queue* pada Standard Template Library C++ tidak mendukung *decrease key*. Oleh karena itu dilakukan *enqueue* vertex pada *priority queue* (baris 24 dan baris 35), tetapi hanya dilakukan *scanning* jika vertex tersebut belum pernah dikunjungi atau *priority* vertex tersebut pada *priority queue* sama dengan jarak jalur terpendek ke vertex tersebut (baris 16).

Agar efisien dalam melakukan *Graph Coloring*, maka untuk mengubah *color* dari vertex menjadi *red* (pada awalnya *color* dari setiap vertex adalah *white*) (baris 7 dan baris 16 pada **Gambar 3.*5***) variabel *color* pada vertex diisi dengan indeks (variabel *color* pada graph) (baris 22 dan baris 33).

4.9 Implementasi Fungsi Delete

Fungsi Delete diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*6*** dan **Gambar 3.*7***. Implementasi Fungsi Delete ditunjukkan dalam **Kode Sumber 4.*24*** dan **Kode Sumber 4.*25***.

```
1.  void doRemove(Vertex *source, Vertex *destination) {
2.      Edge     *edge     =     graph->eraseEdge(source,
    destination);
3.      if (edge == NULL || !Graph::unsetOwner(edge))
4.          return;
5.      delete edge;
6.      if (destination->parent != source || (source-
    >distance == INF && destination->distance == INF))
7.          return;
8.      priority_queue < Neighbor > MPQ, QPQ;
9.      vector < Vertex * > redVertex;
10.     for (MPQ.push(Neighbor(destination, -destination-
    >distance)); !MPQ.empty(); ) {
11.         Neighbor now = MPQ.top();
12.         MPQ.pop();
13.         now.distance = -now.distance;
14.         Neighbor         bestNonRedNeighbor         =
    Graph::getBestNonRedNeighbor(now.vertex);
15.         if              (now.distance              ==
    bestNonRedNeighbor.distance)
16.             now.vertex-
    >setParent(bestNonRedNeighbor.vertex);
17.         else {
18.             now.vertex->color = graph->color;
19.             redVertex.push_back(now.vertex);
20.             for (vector < Vertex * > :: iterator it =
    now.vertex->children.begin();   it   !=   now.vertex-
    >children.end(); it++)
21.                 MPQ.push(Neighbor((*it),      -(*it)-
    >distance));
22.         }
23.     }
24.     for (vector < Vertex * > :: iterator it =
    redVertex.begin(); it != redVertex.end(); it++) {
```

Kode Sumber 4.24 Implementasi Fungsi Delete (1)

```
25.         Neighbor         bestNonRedNeighbor         =
    Graph::getBestNonRedNeighbor(*it);
26.         if (bestNonRedNeighbor.vertex == NULL) {
27.             (*it)->setParent(NULL);
28.             (*it)->distance = INF;
29.         }
30.         else {
31.             (*it)-
    >setParent(bestNonRedNeighbor.vertex);
32.             (*it)->distance                          =
    bestNonRedNeighbor.distance;
33.             QPQ.push(Neighbor(*it,            -(*it)-
    >distance));
34.         }
35.     }
36.     for (; !QPQ.empty(); ) {
37.         Neighbor now = QPQ.top();
38.         QPQ.pop();
39.         now.distance = -now.distance;
40.         if (now.distance == now.vertex->distance)
41.             for (vector < Edge * > :: iterator it =
    now.vertex->neighbor.begin();   it   !=   now.vertex-
    >neighbor.end(); it++) {
42.                 int distance = now.distance + (*it)-
    >distance;
43.                 if   ((*it)->destination->color   ==
    graph->color   &&   distance   <   (*it)->destination-
    >distance) {
44.                     (*it)->destination-
    >setParent(now.vertex);
45.                     (*it)->destination->distance    =
    distance;
46.                     QPQ.push(Neighbor((*it)-
    >destination, -distance));
47.                 }
48.             }
49.     }
50.     doUpdate(redVertex);
51.     graph->color++;
52. }
```

Kode Sumber 4.25 Implementasi Fungsi Delete (2)

Fungsi Delete membutuhkan *decrease key* jika vertex terdapat dalam *priority queue* (baris 32 hingga baris 35 pada **Gambar 3.*6***). Namun *priority queue* pada Standard Template Library C++ tidak mendukung *decrease key*. Oleh karena itu dilakukan *enqueue* vertex pada *priority queue* (baris 46), tetapi hanya dilakukan *scanning* jika vertex tersebut belum pernah dikunjungi atau *priority* vertex tersebut pada *priority queue* sama dengan jarak jalur terpendek ke vertex tersebut (baris 40).

Agar efisien dalam melakukan *Graph Coloring*, maka untuk mengubah *color* dari vertex menjadi *red* (pada awalnya *color* dari setiap vertex adalah *white*) (baris 13 pada **Gambar 3.*6***) variabel *color* pada vertex diisi dengan indeks (variabel *color* pada graph) (baris 18). Selain itu komputasi jalur terpendek hanya dilakukan jika *color* dari vertex tersebut *red* (baris 29 pada **Gambar 3.*6***) atau variabel *color* pada vertex sama dengan indeks (variabel *color* pada graph) (baris 43).

4.10 Implementasi Fungsi Main

Fungsi Main diimplementasikan sesuai dengan Pseudocode yang ditunjukkan dalam **Gambar 3.*1*** dan **Gambar 3.*2***. Implementasi Fungsi Main ditunjukkan dalam **Kode Sumber 4.*26***.

```
1.  int main(int argc, char *argv[]) {
2.      for (int numberVertex, numberEdge; scanf("%d%d",
    &numberVertex, &numberEdge) == 2 && !(numberVertex ==
    0 && numberEdge == 0); ) {
3.          graph = new Graph();
4.          graph->numberVertex = numberVertex;
5.          for (int i = 0; i < numberVertex; i++)
6.              graph->vertex[i] = new Vertex(i);
7.          for (int i = 0; i < numberEdge; i++) {
8.              int source, destination, distance;
9.              scanf("%d%d%d",  &source,  &destination,
    &distance);
10.             source--; destination--;
11.             graph->insertEdge(graph->vertex[source],
    graph->vertex[destination], distance);
12.         }
```

Kode Sumber 4.26 Implementasi Fungsi Main (1)

```
13.         preprocess();
14.         int query;
15.         for (scanf("%d", &query); query--; ) {
16.             char command;
17.             for (; (command = toupper(getchar())) !=
    EOF && command != 'R' && command != 'I' && command !=
    'P'; );
18.             switch (command) {
19.                 case 'R': {
20.                     int source, destination;
21.                     scanf("%d%d",            &source,
    &destination);
22.                     source--; destination--;
23.                     doRemove(graph->vertex[source],
    graph->vertex[destination]);
24.                     break;
25.                 }
26.                 case 'I': {
27.                     int     source,     destination,
    distance;
28.                     scanf("%d%d%d",          &source,
    &destination, &distance);
29.                     source--; destination--;
30.                     doInsert(graph->vertex[source],
    graph->vertex[destination], distance);
31.                     break;
32.                 }
33.                 case 'P': {
34.                     int destination;
35.                     scanf("%d", &destination);
36.                     destination--;
37.                     int     distance     =     graph-
    >vertex[destination]->distance;
38.                     if (distance == INF)
39.                         distance = -1;
40.                     printf("%d\n", distance);
41.                     break;
42.                 }
43.             }
44.         }
```

Kode Sumber 4.27 Implementasi Fungsi Main (2)

```
45.         for (int i = 0; i < numberVertex; i++)
46.             delete graph->vertex[i];
47.         for (int i = 0; i < numberEdge; i++)
48.             delete graph->edge[i];
49.         delete graph;
50.     }
51.     return 0;
52. }
```

Kode Sumber 4.28 Implementasi Fungsi Main (3)

BAB V
UJI COBA DAN EVALUASI

Pada bab ini penulis menjelaskan tentang uji coba dan evaluasi dari implementasi yang telah dilakukan.

5.1 Lingkungan Uji Coba

Lingkungan uji coba yang akan digunakan sebagai berikut:

1. Perangkat Keras
   Processor Intel® Core™ i7-4702MQ CPU @ 2.20GHz
   RAM 8.00 GB
   64-bit Operating System, x64-based processor
2. Perangkat Lunak
   Operating System Windows 8.1
   Integrated Development Environment Code::Blocks 13.12

5.2 Skenario Uji Coba

Pada subbab ini akan dijelaskan skenario uji coba yang akan dilakukan. Skenario uji coba terdiri dari uji coba kebenaran dan uji coba kinerja.

5.2.1 Uji Coba Kebenaran

Uji coba kebenaran dilakukan dengan mengirim kode sumber implementasi yang telah dilakukan ke situs SPOJ. Permasalahan yang akan diselesaikan adalah *Help Your Commander* seperti yang dijelaskan pada subbab 2.8. Setelah kode sumber implementasi dikirimkan ke situs SPOJ, dapat dilihat umpan balik sistem pada situs SPOJ seperti yang dijelaskan pada subbab 2.7. Hasil uji coba pada situs SPOJ ditunjukkan dalam **Gambar 5.*1***.

Gambar 5.1 Hasil Uji Coba pada Situs SPOJ

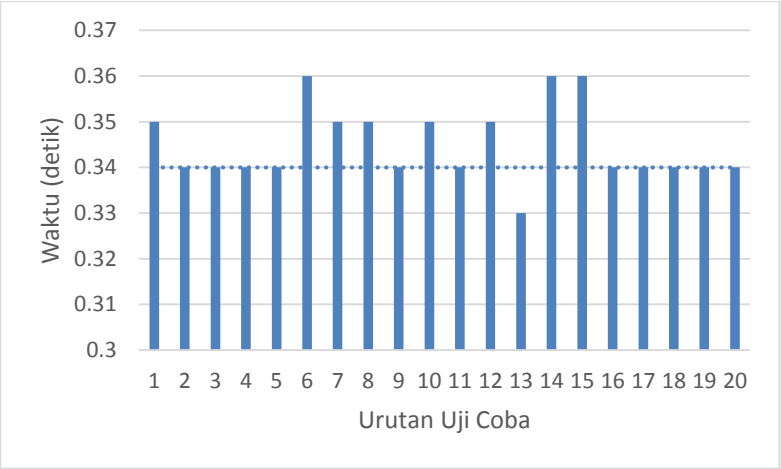
Dari hasil uji coba yang telah dilakukan, kode sumber program yang dikirimkan mendapat umpan balik ***Accepted***. Waktu yang dibutuhkan program adalah 0,37 detik dan memori yang

dibutuhkan program adalah 4,1 MB. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan berhasil menyelesaikan permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif dengan benar.

Setelah itu dilakukan pengiriman kode sumber implementasi sebanyak 20 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Hasil uji coba pada situs SPOJ sebanyak 20 kali ditunjukkan dalam **Tabel 5.*1***, **Gambar 5.*2***, dan **Gambar A.*1***.

Tabel 5.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 20 Kali

No	Hasil	Waktu (detik)	Memori (MB)
1	Accepted	0.35	4.1
2	Accepted	0.34	4.1
3	Accepted	0.34	4.1
4	Accepted	0.34	4.1
5	Accepted	0.34	4.1
6	Accepted	0.36	4.1
7	Accepted	0.35	4.1
8	Accepted	0.35	4.1
9	Accepted	0.34	4.1
10	Accepted	0.35	4.1
11	Accepted	0.34	4.1
12	Accepted	0.35	4.1
13	Accepted	0.33	4.1
14	Accepted	0.36	4.1
15	Accepted	0.36	4.1
16	Accepted	0.34	4.1
17	Accepted	0.34	4.1
18	Accepted	0.34	4.1
19	Accepted	0.34	4.1
20	Accepted	0.34	4.1



Gambar 5.2 Grafik Hasil Uji Coba pada Situs SPOJ Sebanyak 20 Kali

Dari hasil uji coba yang telah dilakukan, seluruh kode sumber program yang dikirimkan mendapat umpan balik ***Accepted***. Waktu yang dibutuhkan program minimum 0,33 detik, maksimum 0,36 detik, dan yang paling banyak adalah 0,34 detik. Memori yang dibutuhkan program tetap 4,1 MB.

5.2.2 Uji Coba Kinerja

Untuk graph dengan n vertex dan m edge, algoritma Dijkstra Dinamis mempunyai kompleksitas *time* $O(\sqrt{m}\, log\, n)$ *amortized time* untuk setiap vertex update. Uji coba kinerja dilakukan dengan membuat graph acak dengan banyak vertex dan banyak edge bervariasi. Banyak kasus uji dan banyak operasi dibuat tetap agar pengujian dapat menunjukkan pengaruh banyak vertex dan banyak edge terhadap waktu yang dibutuhkan program. Banyak operasi kueri dibuat tetap karena kueri tidak menggunakan algoritma Dijkstra Dinamis. Banyak operasi penyisipan dan penghapusan edge dibuat sama, selain itu operasi penyisipan dan penghapusan edge akan dilakukan secara bergantian agar banyak edge setelah

dilakukan modifikasi pada graph tidak berbeda jauh dengan banyak edge sebelum dilakukan modifikasi pada graph.

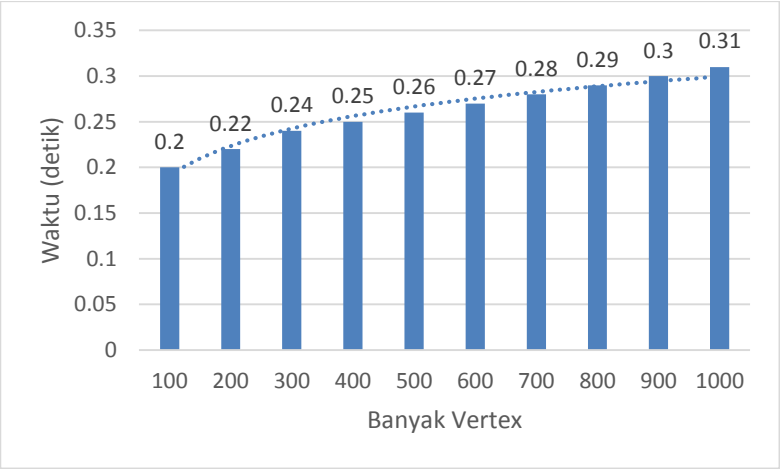
Operasi berulang setiap 4 kali dimana operasi 1 merupakan operasi penghapusan edge, operasi 3 merupakan operasi penyisipan edge, dan operasi 2 dan 4 merupakan operasi kueri. Untuk operasi penyisipan edge, dipilih secara acak edge yang tidak terdapat pada graph. Untuk operasi penghapusan edge, dipilih secara acak edge yang terdapat pada graph. Untuk operasi kueri, dipilih secara acak vertex yang terdapat pada graph.

5.2.2.1 Pengaruh Banyak Vertex Terhadap Waktu

Banyak kasus uji dibuat tetap yaitu 50. Banyak vertex dibuat bervariasi antara 100 hingga 1.000 dengan rentang 100. Banyak edge dibuat tetap yaitu 3.000. Banyak operasi dibuat tetap yaitu 1.000. Dicatat waktu yang dibutuhkan program untuk setiap graph acak yang telah dibuat. Hasil uji coba ditunjukkan dalam **Tabel 5.*2*** dan **Gambar 5.*3***.

Tabel 5.2 Hasil Uji Coba Pengaruh Banyak Vertex Terhadap Waktu

No	Banyak Vertex	Waktu (detik)
1	100	0.2
2	200	0.22
3	300	0.24
4	400	0.25
5	500	0.26
6	600	0.27
7	700	0.28
8	800	0.29
9	900	0.3
10	1000	0.31



Gambar 5.3 Grafik Hasil Uji Coba Pengaruh Banyak Vertex Terhadap Waktu

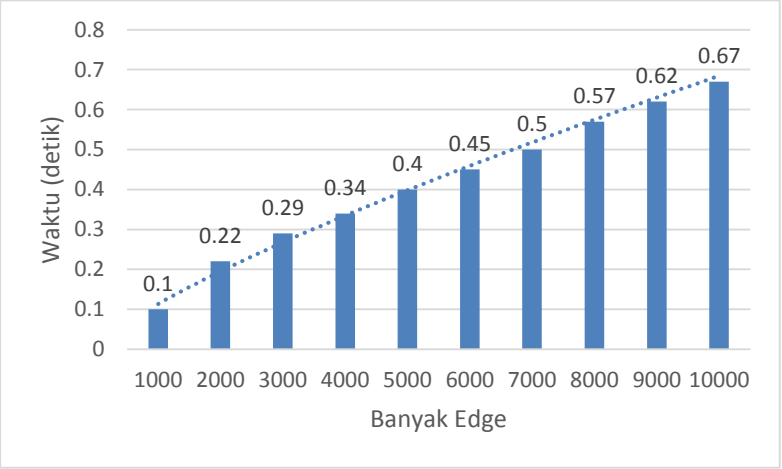
Dari hasil uji coba yang telah dilakukan, dapat dilihat bahwa pertumbuhan waktu yang dibutuhkan program mendekati kurva logaritmik seiring dengan pertumbuhan banyak vertex. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan sesuai dengan kompleksitas *time* dari algoritma Dijkstra Dinamis yang dipengaruhi banyak vertex secara logaritmik.

5.2.2.2 Pengaruh Banyak Edge Terhadap Waktu

Banyak kasus uji dibuat tetap yaitu 50. Banyak vertex dibuat tetap yaitu 1.000. Banyak edge dibuat bervariasi antara 1.000 hingga 10.000 dengan rentang 1.000. Banyak operasi dibuat tetap yaitu 1.000. Dicatat waktu yang dibutuhkan program untuk setiap graph acak yang telah dibuat. Hasil uji coba ditunjukkan dalam **Tabel 5.*3*** dan **Gambar 5.*4***.

Tabel 5.3 Hasil Uji Coba Pengaruh Banyak Edge Terhadap Waktu

No	Banyak Edge	Waktu (detik)
1	1000	0.1
2	2000	0.22
3	3000	0.29
4	4000	0.34
5	5000	0.4
6	6000	0.45
7	7000	0.5
8	8000	0.57
9	9000	0.62
10	10000	0.67



Gambar 5.4 Grafik Hasil Uji Coba Pengaruh Banyak Edge Terhadap Waktu

Dari hasil uji coba yang telah dilakukan, dapat dilihat bahwa pertumbuhan waktu yang dibutuhkan program mendekati kurva akar kuadratik seiring dengan pertumbuhan banyak edge. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan sesuai dengan kompleksitas *time* dari algoritma Dijkstra Dinamis yang dipengaruhi banyak edge secara akar kuadratik.

(Halaman ini sengaja dikosongkan)

BAB VI
KESIMPULAN DAN SARAN

Pada bab ini penulis menjelaskan tentang kesimpulan dari hasil uji coba yang telah dilakukan dan saran mengenai hal-hal yang masih bisa dikembangkan.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi solusi untuk permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif dapat diambil kesimpulan sebagai berikut:

1. Implementasi yang telah dilakukan dapat menyelesaikan permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif dengan benar.
2. Algoritma Dijkstra Dinamis mempunyai kompleksitas *time* $O(\sqrt{m}\log n)$ *amortized time* untuk setiap vertex update. Artinya waktu yang dibutuhkan program dipengaruhi oleh banyak vertex secara logaritmik dan banyak edge secara akar kuadratik.

6.2 Saran

Berikut beberapa saran pengembangan implementasi solusi untuk permasalahan jalur terpendek satu sumber pada graph dinamis dengan bobot edge bilangan real positif:

1. Implementasi yang telah dilakukan menggunakan *priority queue* pada Standard Template Library C++ yang tidak mendukung operasi *decrease key*. Selain itu *priority queue* tersebut diimplementasikan menggunakan *binary heap*. Pengembangan implementasi bisa dilakukan dengan menggunakan *priority queue* yang mendukung operasi *decrease key* dan diimplementasikan menggunakan *fibonacci heap*.

2. Implementasi yang telah dilakukan menggunakan *set* untuk menggantikan *priority queue* yang tidak mendukung akses pada elemen dengan dengan urutan *priority* tertentu. Pengembangan implementasi bisa dilakukan mengganti *set* dengan struktur data yang lebih efisien yang mendukung operasi yang dibutuhkan.

DAFTAR PUSTAKA

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Single-Source Shortest Paths," in *Introduction to Algorithms (3rd ed.)*, The MIT Press, 2009, pp. 643-683.

[2] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully Dynamic Output Bounded Single Source Shortest Path Problem," 1996.

[3] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully Dynamic Algorithms for Maintaining Shortest Paths Trees," *Journal of Algorithms*, vol. 34, no. 2, pp. 251-281, 2000.

[4] "SPOJ.com - Problem HELPCOMM," [Online]. Available: http://www.spoj.com/problems/HELPCOMM/.

(Halaman ini sengaja dikosongkan)

LAMPIRAN A

Not hidden submissions | All submissions

Ivan Hendrajaya: submissions

Previous 1 2 3 4 5 Next >

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
13371433		2015-01-08 00:05:26	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371432		2015-01-08 00:05:16	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371431		2015-01-08 00:05:06	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371430		2015-01-08 00:04:56	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371428		2015-01-08 00:04:46	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371427		2015-01-08 00:04:36	Help Your Commander	accepted edit run	0.36	4.1M	C++ 4.3.2
13371426		2015-01-08 00:04:26	Help Your Commander	accepted edit run	0.36	4.1M	C++ 4.3.2
13371425		2015-01-08 00:04:16	Help Your Commander	accepted edit run	0.33	4.1M	C++ 4.3.2
13371424		2015-01-08 00:04:06	Help Your Commander	accepted edit run	0.35	4.1M	C++ 4.3.2
13371423		2015-01-08 00:03:56	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371422		2015-01-08 00:03:46	Help Your Commander	accepted edit run	0.35	4.1M	C++ 4.3.2
13371421		2015-01-08 00:03:36	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371420		2015-01-08 00:03:26	Help Your Commander	accepted edit run	0.35	4.1M	C++ 4.3.2
13371419		2015-01-08 00:03:16	Help Your Commander	accepted edit run	0.35	4.1M	C++ 4.3.2
13371417		2015-01-08 00:03:06	Help Your Commander	accepted edit run	0.36	4.1M	C++ 4.3.2
13371416		2015-01-08 00:02:56	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371415		2015-01-08 00:02:46	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371413		2015-01-08 00:02:36	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371412		2015-01-08 00:02:26	Help Your Commander	accepted edit run	0.34	4.1M	C++ 4.3.2
13371411		2015-01-08 00:02:15	Help Your Commander	accepted edit run	0.35	4.1M	C++ 4.3.2

Gambar A.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 20 Kali

(Halaman ini sengaja dikosongkan)

BIODATA PENULIS

Ivan Hendrajaya, lahir di Surabaya pada tanggal 30 Agustus 1993, anak kedua dari 2 bersaudara. Penulis telah menempuh pendidikan formal mulai dari jenjang TK hingga S-1 di TKK St. Clara Surabaya (1997-1999), SDK St. Clara Surabaya (1999-2005), SMPK St. Clara Surabaya (2005-2008), SMAK St. Louis 1 Surabaya (2008-2011), dan Jurusan Teknik Informatika Fakultas Teknologi Informasi (FTIF) Institut Teknologi Sepuluh Nopember (ITS) Surabaya (2011-2015). Sejak tahun 2011 hingga saat ini, penulis aktif sebagai anggota Alumni Tim Olimpiade Komputer Indonesia (TOKI). Penulis pernah menjadi panitia (2012 dan 2013) dan juri (2014) pada Olimpiade Sains Nasional (OSN) Bidang Informatika/Komputer, serta asisten (2014) pada Pelatnas 2 TOKI. Ketika masih menempuh perkuliahan di Jurusan Teknik Informatika FTIF-ITS, penulis sering menjadi finalis kompetisi pemrograman tingkat nasional yang diselenggarakan di Institut Teknologi Bandung (ITB), Universitas Indonesia (UI), dan Bina Nusantara University (BINUS). Penulis pernah menjadi Juara III (2013) pada kompetisi Gemastik Kategori Debugging. Selain itu sejak tahun 2012 hingga saat ini, penulis juga aktif sebagai anggota Himpunan Mahasiswa Teknik Computer-Informatika (HMTC) ITS. Penulis pernah menjadi wakil koordinator (2012) dan koordinator (2013) pada National Programming Contest (NPC) Schematics ITS. Selain itu penulis juga pernah menjadi asisten mata kuliah Pemrograman Terstruktur (2012 dan 2013), Algoritma dan Struktur Data (2013 dan 2014), dan Dasar Pemrograman (2014) di Jurusan Teknik Informatika FTIF-ITS.