TUGAS AKHIR - KI141502

STUDI BANDING ALGORITMA PENYELESAIAN PERMASALAHAN MINIMUM PATH OF ALL EDGE COVERAGE PADA DIRECTED ACYCLIC GRAPH PLANAR: STUDI KASUS SPOJ - SKIVER1 (SKIERS)

ALFIAN
NRP 05111640000073

Dosen Pembimbing 1
Rully Soelaiman S.Kom., M.Kom.

Dosen Pembimbing 2
M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

DEPARTEMEN TEKNIK INFORMATIKA
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya, 2020

Halaman ini sengaja dikosongkan

UNDERGRADUATE THESIS - KI141502

COMPARISON STUDY OF SOLUTION ALGORITHM ON MINIMUM PATH OF ALL EDGE COVERAGE PROBLEM IN PLANAR DIRECTED ACYCLIC GRAPH: CASE STUDY SPOJ - SKIVER1 (SKIERS)

ALFIAN
NRP 05111640000073

Supervisor 1
Rully Soelaiman S.Kom., M.Kom.

Supervisor 2
M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

DEPARTMENT OF INFORMATICS ENGINEERING
Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, 2020

Halaman ini sengaja dikosongkan

LEMBAR PENGESAHAN

STUDI BANDING ALGORITMA PENYELESAIAN PERMASALAHAN MINIMUM PATH OF ALL EDGE COVERAGE PADA DIRECTED ACYCLIC GRAPH PLANAR: STUDI KASUS SPOJ - SKIVER1 (SKIERS)

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
Pada
Bidang Studi Algoritma Pemrograman
Program Studi S-1 Teknik Informatika
Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember

Oleh:
Alfian
NRP: 05111640000073

Disetujui oleh Dosen Pembimbing Tugas Akhir:

Rully Soelaiman, S.Kom., M.Kom.
NIP. 197002131994021001 ..... (Pembimbing 1)

M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.
NIP.197402092002121001 ..... (Pembimbing 2)

KEMENTERIAN PENDIDIKAN DAN KEBUDAYAAN INSTITUT TEKNOLOGI SEPULUH NOPEMBER FAKULTAS TEKNOLOGI ELEKTRO DAN INFORMATIKA CERDAS DEPARTEMEN TEKNIK INFORMATIKA

SURABAYA
2020

Halaman ini sengaja dikosongkan

Studi Banding Algoritma Penyelesaian Permasalahan Minimum Path of All Edge Coverage Pada Directed Acyclic Graph Planar: Studi Kasus SPOJ - Skiver1 (Skiers)

Nama : Alfian
NRP : 05111640000073
Departemen : Departemen Teknik Informatika, Fakultas Teknologi Elektro dan Informatika Cerdas, ITS
Pembimbing 1 : Rully Soelaiman S.Kom., M.Kom.
Pembimbing 2 : M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

ABSTRAK

Minimum Path of All Edges Coverage (MPAEC) adalah permasalahan meminimalkan jumlah path yang dibutuhkan untuk mencakup semua edge yang ada pada Directed Acyclic Graph (DAG) dengan ketentuan sebuah jalur mungkin saja dilewati lebih dari sekali. Permasalahan ini bisa diselesaikan dengan pendekatan Minimum Path Cover.

Permasalahan SPOJ - Skiver1 (Skiers) adalah permasalahan MPAEC dengan vertex 1 sebagai vertex awal dan vertex N sebagai vertex terakhir di mana N adalah nomor vertex terakhir pada DAG. Proses traversing (penelusuran) dimulai dari vertex 1 dan berakhir di vertex N, dan edge pada data masukan sudah terurut dari permukaan paling kiri ke permukaan paling kanan. Sehingga dengan keteraturan data tersebut bisa dianalisis apakah keteraturan tersebut bisa dimanfaatkan sehingga bisa ditemukan algoritma penyelesaian yang lebih efisien.

Pada Tugas Akhir (TA) ini dirancang dua penyelesaian untuk permasalahan SPOJ - Skiver1 (Skiers), yaitu: (1) penyelesaian

dengan Minimum Path Cover, dan (2) pendekatan DFS (Depth Firsh Search) dengan Greedy Pruning, Perbandingan dua pendekatan tersebut dalam menyelesaikan permasalahan MPAEC pada studi kasus SPOJ - Skiver1 (Skiers) dibahas lebih lanjut dalam TA ini, di mana diperlihatkan bahwa pendekatan (2) lebih baik dibandingkan (1) dalam studi kasus SPOJ - Skiver1.

Kata kunci: DAG, Greedy, Minimum Path Cover

Comparison Study of Solution Algorithm on Minimum Path of All Edge Coverage Problem in Planar Directed Acyclic Graph: Case Study SPOJ - Skiver1 (Skiers)

Name : Alfian
NRP : 05111640000073
Major : Department of Informatics Engineering , Faculty of Intelligent Electrical and Informatics Technology
Supervisor 1 : Rully Soelaiman S.Kom., M.Kom.
Supervisor 2 : M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

ABSTRACT

Minimum Path of All Edges Coverage (MPAEC) is a minimization problem that minimizes the required paths for covering all edges of the Directed Acyclic Graph (DAG) where an edge may be passed more than once. This problem can be solved by Minimum Path Cover approach.
SPOJ - Skiver1 (Skiers) problem is an MPAEC problem where vertex 1 is the first vertex and vertex N is the last vertex of the DAG. Traversing process start from vertex 1 and ended at vertex N, and the input's edges are sorted from left most face to the right-most face. Thus, by its structured data, it can be analyzed whether that characteristic can be an advantage to be used for finding more efficient solution's algorithm.
In this thesis, two solutions for SPOJ - Skiver1 (Skiers) have been designed: (1) the solution by using Minimum Path Cover, and (2) DFS (Depth First Search) with Greedy Pruning. The comparison

between these two solutions to solve MPAEC problem in the case study of SPOJ - Skiver1 (Skiers) has been discussed in this thesis, where it can be shown that (2) is better than (1).

Keywords: DAG, Greedy, Minimum Path Cover

KATA PENGANTAR

Puji syukur penulis ucapkan kepada Tuhan Yang Maha Esa atas pimpinan, penyertaan dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul:

STUDI BANDING ALGORITMA PENYELESAIAN PERMASALAHAN MINIMUM PATH of ALL EDGE COVERAGE PADA DiRECTED ACYCLIC GRAPH PLANAR: STUDI KASUS SPOJ - SKIVER1 (SKIERS)

Penulisan Tugas Akhir ini dilakukan untuk memenuhi salah satu syarat meraih gelar Sarjana di Departemen Informatika, Fakultas Teknologi Informasi dan Komunikasi, Institut Teknologi Sepuluh Nopember, ITS, Surabaya.

Dengan selesainya Tugas Akhir ini diharapkan apa yang telah dikerjakan penulis dapat memberikan manfaat bagi perkembangan ilmu pengetahuan terutama di bidang teknologi informasi serta bagi diri penulis sendiri.

Penulis mengucapkan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama penulis mengerjakan Tugas Akhir maupun selama menempuh masa studi antara lain:

- Kedua orang tua dan keluarga penulis yang selalu mendoakan dan memberikan dukungan pada setiap aspek kehidupan penulis termasuk dalam pengerjaan Tugas Akhir ini.
- Bapak Rully Soelaiman, S.Kom., M.Kom, selaku Dosen Pembimbing yang telah memberikan ilmu dan selalu memberi

- motivasi serta dukungan kepada penulis baik selama masa kuliah maupun selama pengerjan Tugas Akhir ini.
- Bapak M.M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil., selaku dosen pembimbing yang telah memberikan ilmu dan masukan kepada penulis.
- Departemen Pendidikan dan Kebudayaan Republik Indonesia yang telah memberikan dukungan berupa beasiswa bidikmisi yang sangat membantu bagi penulis untuk berkuliah di ITS.
- Teman-teman, kakak-kakak dan adik-adik administrator Laboratorium Algoritma dan Pemrograman yang memberi pengalaman berkesan selama masa kuliah.
- Seluruh tenaga pengajar, asisten dosen dan karyawan Departemen Teknik Informatika ITS yang telah memberi ilmu dan waktunya yang bermanfaat bagi penulis.
- Seluruh teman penulis yang telah memberi dukungan dan semangat kepada penulis baik dalam kehidupan perkuliahan maupun dalam pengerjaan Tugas Akhir ini.
- Seluruh pihak-pihak yang tidak dapat disebutkan disini yang telah membantu dalam masa perkuliahan penulis.

Penulis memohon maaf apabila masih ada kesalahan dan kekurangan pada Tugas Akhir ini, Penulis berharap ada kritik dan saran yang membangun untuk perbaikan dan pembelajaran, Semoga Tugas Akhir ini dapat menjadi kontribusi penulis dan dapat memberikan manfaat sebaik-baiknya kepada kemaslahatan masyarakat.

Surabaya, Desember 2019

Alfian

DAFTAR ISI

SAMPUL i
COVER iii
LEMBAR PENGESAHAN v
ABSTRAK vii
ABSTRACT ix
KATA PENGANTAR xi
DAFTAR ISI xiii
DAFTAR TABEL xvii
DAFTAR GAMBAR xix
DAFTAR KODE SUMBER xxi
BAB I PENDAHULUAN **1**
1.1 Latar Belakang 1
1.2 Rumusan Masalah 2
1.3 Batasan Masalah 2
1.4 Tujuan 3
1.5 Manfaat 3
1.6 Metodologi 4
1.7 Sistematika Penulisan 5
BAB II DASAR TEORI **7**
2.1 Deskripsi Permasalahan 7
2.2 Deskripsi Umum 9

2.2.2 Algoritma Hopcroft-Karp.....9
2.2.3 Transitive Closure.....10
2.2.4 Topological Sort.....11
2.3 Strategi Penyelesaian dengan Minimum Path Cover.....11
2.3.1 Minimum Path Cover pada Permasalahan MPAEC.....12
2.3.2 Transformasi Edge menjadi Vertex dan Vertex menjadi Edge.....13
2.3.3 Minimum Path Cover dengan Non-Unique Vertex....735
2.3.4 Membangun Bentuk Transitive Closure dari DAG.....17
2.3.5 Maximum Matching dan Minimum Path Cover.....21
2.4 Strategi Penyelesaian menggunakan DFS dengan *Greedy Pruning*.....22
2.4.1 Intuisi Visual pada Permasalahan MPAEC.....22
2.4.2 Kondisi *subpath* sebelumnya dari suatu *edge*.....28
2.4.3 Kondisi *subpath* selanjutnya dari suatu *edge*.....26
2.4.4 Kondisi *subpath* sebelumnya dan selanjutnya dari suatu *edge*.....28
BAB III DESAIN..... 31
3.1 Desain Umum Sistem..... 31
3.2 Desain Solusi Minimum Path Cover..... 31
3.2.1 Desain Class Graph.....31
3.2.2 Desain Class Algoritma Hopcroft-Karp (*HopcroftKarp*) 33
3.2.3 Desain Fungsi Transformasi Edge ke Vertex (*transformEdgeToVertex*)..... 36
3.2.4 Desain Fungsi Pencarian Urutan Topologis (*getTopologicalSort*)..... 37

3.2.5 Desain Fungsi Pembentukan Transitive Closure (*getTransitiveClosure*)..... 38
3.2.6 Desain Fungsi Pencarian Minimum Path Cover (*findMinimumPathCover*)..... 39
3.2.7 Desain Fungsi Utama Solusi Minimum Path Cover (*main*)..... 40
3.3 Desain Solusi DFS dengan Greedy Pruning..... 41
3.3.1 Desain Fungsi DFS dengan Greedy Pruning (*dfsWithPruning*)..... 41
3.3.2 Desain Fungsi Utama Solusi DFS dengan Greedy Pruning(*main*)..... 42
BAB IV IMPLEMENTASI..... 44
4.1 Lingkungan Implementasi..... 45
4.2 Rancangan Data..... 45
4.2.1 Data Masukan..... 46
4.2.2 Data Keluaran..... 46
4.3 Implementasi Solusi dengan Algoritma *Minimum Path Cover*..... 46
4.3.1 *Header* yang Digunakan..... 47
4.3.2 Implementasi Class Graph..... 48
4.3.3 Implementasi Class Algoritma Hopcroft-karp *(HopcroftKarp)*..... 49
4.3.4 Implementasi Fungsi Transformasi Edge ke Vertex *(transformEdgeToVertex)*..... 53
4.3.5 Implementasi Fungsi Pencarian Urutan Topologis *(getTopologicalSort)*..... 54
4.3.6 Implementasi Fungsi Pembentukan *Transitive Closure (getTransitiveClosure)*..... 56

4.3.7 Implementasi Fungsi Pencarian *Minimum Path Cover (findMinimumPathCover)* 58
4.3.8 Implementasi Fungsi Utama Solusi *Minimum Path Cover (main)* 59
4.4 Implementasi Solusi menggunakan DFS dengan *Greedy Pruning* 60
4.4.1 Variabel Global 60
4.4.2 Implementasi Fungsi DFS dengan *Greedy Pruning* (dfsWithPruning) 61
4.4.3 Implementasi Fungsi Utama Solusi Greedy (*main*) 63
BAB V UJI COBA DAN EVALUASI 65
5.1 Lingkungan Uji Coba 65
5.2 Uji Coba Kebenaran dan Perbandingan Kinerja 65
5.3 Analisis Kompleksitas 68
5.3.1 Analisis Kompleksitas Pendekatan Algoritma *Minimum Path Cover* 68
5.3.2 Analisis Kompleksitas Pendekatan Algoritma DFS dengan *Greedy Pruning* 70
5.4 Analisis dan Kesimpulan umum 70
5.5 Pengembangan MPAEC Lebih Lanjut 71
BAB XI KESIMPULAN 73
6.1 Kesimpulan 73
6.2 Saran 74
DAFTAR PUSTAKA 75
BIODATA PENULIS 77

DAFTAR TABEL

Tabel 2.1 *Matrix edge list* pada DAG 14
Tabel 2.2 Urutan hasil proses iterasi transformasi 15
Tabel 2.3a Hasil iterasi 1 dari *vertex* D 18
Tabel 2.3b Hasil iterasi 1 dari *vertex* E 18
Tabel 2.3c Hasil iterasi 1 dari *vertex* C 19
Tabel 2.3d Hasil iterasi 1 dari *vertex* B 19
Tabel 2.3e Hasil iterasi 1 dari *vertex* A 20
Tabel 3.1 Tabel Penjelasan variabel *class HopcroftKarp* 34
Tabel 4.1 Penjelasan variabel pada *class Graph* 49
Tabel 4.2 Penjelasan variabel *class HopcroftKarp* 52
Tabel 4.3 Penjelasan variabel fungsi *transformEdgeToVertex* . 54
Tabel 4.4 Penjelasan variabel pada fungsi *getTopologicalSort* . 56
Tabel 4.5 Penjelasan variabel pada fungsi *getTransitiveClosure* 57
Tabel 4.6 Penjelasan variabel pada fungsi *findMinimumPathCover* 59
Tabel 4.7 Penjelasan variabel global 60
Tabel 4.8 Penjelasan variabel pada fungsi *dfsWithPruning* . . . 62
Tabel 4.9 Penjelasan variabel fungsi utama penyelesaian menggunakan DFS dengan *Greedy Pruning* 64
Tabel 5.1 Perbandingan kinerja dengan hasil umpan balik dengan pengujian 10 kali dari situs SPOJ 68

Halaman ini sengaja dikosongkan

DAFTAR GAMBAR

Gambar 2.1 Contoh graf permasalahan SPOJ-Skiver1 (Skiers) 8
Gambar 2.2 DAG sebelum menjadi bentuk *transitive closure* . 10
Gambar 2.3 *Transitive Closure* dari DAG 10
Gambar 2.4 *Topological sort* Dari DAG 11
Gambar 2.5 Graf untuk permasalahan MPAEC dengan *edge* bernomor 13
Gambar 2.6 Hasil transformasi graf dari *edge* menjadi *vertex* dan sebaliknya 15
Gambar 2.7 *Minimum Path Cover* pada DAG 16
Gambar 2.8 *Minimum Path cover* pada bentuk *transitive closure* dari DAG 16
Gambar 2.9 DAG dengan *incidency matrix*-nya 17
Gambar 2.10 DAG dalam bentuk *transitive closure* dengan *incidency matrix*-nya 20
Gambar 2.11 DAG dan bentuk bipartite graph-nya 21
Gambar 2.12a Iterasi penyelesaian intuitif ke-1 23
Gambar 2.12b Iterasi penyelesaian intuitif ke-2 23
Gambar 2.12c Iterasi penyelesaian intuitif ke-3 24
Gambar 2.12d Iterasi penyelesaian intuitif ke-4 24
Gambar 2.12e Iterasi penyelesaian intuitif ke-5 25
Gambar 2.13 Ilustrasi-1 *pruning* pada DAG 26
Gambar 2.14 Ilustrasi-1 *pruning* pada DAG dengan *subpath* yang diubah menjadi *edge* 27
Gambar 2.15 Ilustrasi-2 *pruning* pada DAG 28
Gambar 2.16 Ilustrasi-2 *pruning* pada DAG dengan *subpath* yang diubah menjadi *edge* 29
Gambar 2.17 Ilustrasi-3 *pruning* pada DAG 30
Gambar 5.1 Umpan balik sebanyak 10 kali untuk solusi DFS dengan *Greedy Pruning* 66

Gambar 5.2 Umpan balik sebanyak 10 kali untuk solusi xi *Minimum Path Cover* 67
Gambar 5.3 *Diagram ketergantungan untuk solusi menggunakan Minimum Path Cover* 69

DAFTAR KODE SUMBER

Kode Sumber 3.1 *Pseudocode class* Graph 32
Kode Sumber 3.2a *Pseudocode class HopcroftKarp* 33
Kode Sumber 3.2b *Pseudocode constructor class HopcroftKarp* 33
Kode Sumber 3.2c *Pseudocode* fungsi *addEdge* 34
Kode Sumber 3.2d *Pseudocode* fungsi *bfs* 35
Kode Sumber 3.2e *Pseudocode* fungsi *dfs* 35
Kode Sumber 3.2f *Pseudocode* fungsi *findMaxMatch* 36
Kode Sumber 3.3 *Pseudocode* fungsi *transformEdgeToVertex* . 37
Kode Sumber 3.4 *Pseudocode* fungsi *getTopologicalSort* . . . 38
Kode Sumber 3.5 *Pseudocode* fungsi *getTransitiveClosure* . . 39
Kode Sumber 3.6 *Pseudocode* fungsi *findMinimumPathCover* . 40
Kode Sumber 3.7 *Pseudocode* fungsi utama solusi *Minimum Path Cover* 41
Kode Sumber 3.8 *Pseudocode* fungsi *dfsWithPruning* 42
Kode Sumber 3.9 *Pseudocode* fungsi utama menggunakan DFS dengan *Greedy Pruning* 43
Kode Sumber 4.1 *Header* yang digunakan pada solusi menggunakan *Minimum Path Cover* 47
Kode Sumber 4.2 Implementasi *class Graph* 48
Kode Sumber 4.3a Implementasi *class HopcroftKarp* 49
Kode Sumber 4.3b Implementasi *class HopcroftKarp* 50
Kode Sumber 4.3c Implementasi *class HopcroftKarp* 50
Kode Sumber 4.3d Implementasi *class HopcroftKarp* 51
Kode Sumber 4.3e Implementasi *class HopcroftKarp* 51
Kode Sumber 4.4 Implementasi fungsi *transformEdgeToVertex* 53
Kode Sumber 4.5 Implementasi fungsi *getTopologicalSort* . . . 55
Kode Sumber 4.6a Implementasi fungsi *getTransitiveClosure* . 56
Kode Sumber 4.6b Implementasi fungsi *getTransitiveClosure* . 57
Kode Sumber 4.7 Implementasi fungsi *findMinimumPathCover* 58

Kode Sumber 4.8a Implementasi fungsi fungsi utama penyelesaian menggunakan *Minimum Path Cover* 59
Kode Sumber 4.8b Implementasi fungsi fungsi utama penyelesaian menggunakan *Minimum Path Cover* 60
Kode Sumber 4.9 Variabel global yang diperlukan 61
Kode Sumber 4.10 Implementasi fungsi *dfsWithPruning* 61
Kode Sumber 4.10 Implementasi fungsi *dfsWithPruning* 62
Kode Sumber 4.11 Implementasi fungsi utama penyelesaian menggunakan DFS dengan *Greedy Pruning* . . 64

BAB I

PENDAHULUAN

1.1 Latar Belakang

Minimum Path of All Edges Coverage (MPAEC) adalah permasalahan meminimalkan jumlah path yang dibutuhkan untuk mencakup semua edge yang ada pada *Directed Acyclic Graph* (DAG) dengan ketentuan sebuah jalur mungkin saja dilewati lebih dari 1 kali. Permasalahan ini bisa diselesaikan dengan mencari *Minimum Path Cover* pada DAG tersebut.

Permasalahan SPOJ - Skiver1 (Skiers) [1] adalah permasalahan MPAEC dengan *vertex* 1 sebagai *vertex* awal dan *vertex* N sebagai *vertex* terakhir dengan N adalah nomor *vertex* terakhir pada DAG dan proses *traversing* dimulai dari *vertex* 1 dan berakhir di *vertex* N, dan *edge* pada data masukan sudah terurut dari permukaan paling kiri ke permukaan paling kanan. Sehingga dengan keteraturan data tersebut bisa dianalisis apakah pendekatan menggunakan algoritma DFS dengan *Greedy Pruning* bisa dilakukan.

Tugas Akhir ini mengacu pada analisis algoritma DFS dengan *Greedy Pruning* pada studi kasus SPOJ - Skiver1 (Skiers) dan juga perbandingannya dengan algoritma *Minimum Path Cover*.

Hasil dari Tugas Akhir ini adalah memberikan kontribusi khususnya pada ilmu pengetahuan dan teknologi informasi, yaitu mampu memberikan penjelasan dan pemahaman mengenai analisis Algoritma DFS dengan *Greedy Pruning* pada permasalahan pada SPOJ - Skiver1 (Skiers) dan bagaimana perbandingan solusi menggunakan *Minimum Path Cover* dan solusi menggunakan DFS dengan *Greedy Pruning*.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam Tugas Akhir ini adalah seperti dibawah ini.

1. Bagaimana penyelesaian MPAEC pada Studi Kasus SPOJ - Skiver1 (Skiers) menggunakan Algoritma *Minimum Path Cover*?
2. Bagaimana penyelesaian MPAEC pada Studi Kasus SPOJ - Skiver1(Skiers) menggunakan Algoritma DFS dengan *Greedy Pruning*?
3. Bagaimana perbandingan antara algoritma *Minimum Path Cover* dan algoritma DFS dengan *Greedy Pruning* sebagai penyelesaian MPAEC pada Studi Kasus SPOJ - Skiver1 (Skiers)?

1.3 Batasan Masalah

Permasalahan yang dibahas pada Tugas Akhir ini memiliki beberapa batasan, yaitu sebagai berikut ini.

1. Perbandingan antara *algoritma Minimum Path Cover* dan algoritma DFS dengan *Greedy Pruning* sebagai penyelesaian SPOJ - Skiver1 (Skiers).
2. Algoritma DFS dengan *Greedy Pruning* yang dibahas terbatas pada analisis intuitif yang logis.
3. Algoritma *Minimum Path Cover* yang dibahas terbatas pada analisis intuitif yang logis.
4. Pembuktian kebenaran didasarkan pada hasil *submission* di *online judge* SPOJ.

Batasan pada SPOJ adalah seperti dibawah ini.

1. Implementasi algoritma menggunakan bahasa pemrograman C++.
2. Batas nilai N adalah $1 < N < 5001$ dengan N adalah jumlah *vertex*.

3. Input *vertex* terurut dari bidang paling kiri ke bidang paling kanan.
4. *Traversing* yang dilakukan dimulai dari *vertex* 1 dan berhenti di *vertex* ke N.
5. Batas waktu yang diberikan adalah 0,310 detik.

1.4 Tujuan

Tujuan dari pembuatan Tugas Akhir ini adalah sebagai berikut.

1. Melakukan analisis dan implementasi algoritma *Minimum Path Cover* pada permasalahan MPAEC.
2. Melakukan analisis dan implementasi algoritma DFS dengan *Greedy Pruning* pada permasalahan MPAEC pada kasus SPOJ - Skiver1 (Skiers).
3. Melakukan perbandingan antara algoritma *Minimum Path Cover* dan algoritma DFS dengan *Greedy Pruning* pada kasus SPOJ - Skiver1 (Skiers).

1.5 Manfaat

Manfaat yang diperoleh dari Tugas Akhir ini adalah sebagai berikut.

1. Mampu memberikan pemahaman dan penjelasan analisis algoritma *Minimum Path Cover* pada permasalahan MPAECpada kasus SPOJ - Skiver1(Skiers).
2. Mampu memberikan pemahaman dan penjelasan analisis algoritma DFS dengan *Greedy Pruning* pada permasalahan MPAEC pada kasus SPOJ - Skiver1(Skiers).
3. Memberikan hasil perbandingan antara algoritma *Minimum Path Cover* dan algoritma DFS dengan *Greedy Pruning* pada kasus SPOJ - Skiver1 (Skiers).

1.6 Metodologi

Metodologi yang digunakan dalam pengerjaan Tugas Akhir ini adalah sebagai berikut.

1. Penyusunan proposal Tugas Akhir

Pada tahap ini dilakukan penyusunan proposal Tugas Akhir yang berisi permasalahan pada permasalahan klasik SPOJ - Skiver1 (Skiers) serta gagasan solusi yang akan dibahas pada Tugas Akhir ini.

2. Studi literatur

Pada tahap ini dilakukan pencarian informasi dan studi literatur mengenai pengetahuan atau metode yang dapat digunakan dalam penyelesaian masalah. Informasi didapatkan dari materi-materi yang berhubungan dengan algoritma yang digunakan untuk penyelesaian permasalahan ini yang didapat dari buku, jurnal, maupun internet.

3. Desain

Pada tahap ini dilakukan desain rancangan algoritma yang digunakan dalam solusi untuk pemecahan permasalahan klasik SPOJ - Skiver1 (Skiers).

4. Implementasi perangkat lunak

Pada tahap ini dilakukan implementasi dari rancangan desain ke dalam bentuk program dalam bahasa pemrograman C++.

5. Uji coba, perbandingan antar solusi dan evaluasi

Pada tahap ini dilakukan uji coba kebenaran implementasi yang di lakukan pada sistem penilaian daring SPOJ serta juga perbandingan kinerja antara solusi *Minimum Path Cover* dengan solusi DFS dengan *Greedy Pruning.*

6. Penyusunan buku Tugas Akhir

Pada tahap ini dilakukan penyusunan buku Tugas Akhir yang berisi dokumentasi hasil pengerjaan Tugas Akhir.

1.7 Sistematika Penulisan

Berikut adalah sistematika penulisan buku Tugas Akhir ini.

1. BAB I: PENDAHULUAN
   Pada Bab ini akan dibahas latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi dan sistematika penulisan Tugas Akhir.
2. BAB II: DASAR TEORI
   Pada bab ini akan dibahas dasar teori yang digunakan serta analisis algoritma penyelesaian permasalahan MPAEC.
3. BAB III: DESAIN
   Pada bab ini akan dibahas tentang desain algoritma untuk menyelesaikan permasalahan SPOJ - Skiver1 (Skiers) sesuai dengan dasar teori yang dibahas di bab 2.
4. BAB IV: IMPLEMENTASI
   Bab ini berisi implementasi algoritma dalam bahasa pemrograman C++ berdasarkan desain algoritma yang telah dibuat pada bab 3.
5. BAB V: UJI COBA DAN EVALUASI
   Bab ini berisi uji coba kebenaran implementasi yang dilakukan pada sistem penilaian daring SPOJ serta juga perbandingan kinerja antara solusi *Minimum Path Cover* dengan solusi DFS dengan *Greedy Pruning*.
6. BAB VI: KESIMPULAN
   Bab ini berisi kesimpulan yang didapat dari hasil uji coba yang telah dilakukan.

Halaman ini sengaja dikosongkan

BAB II

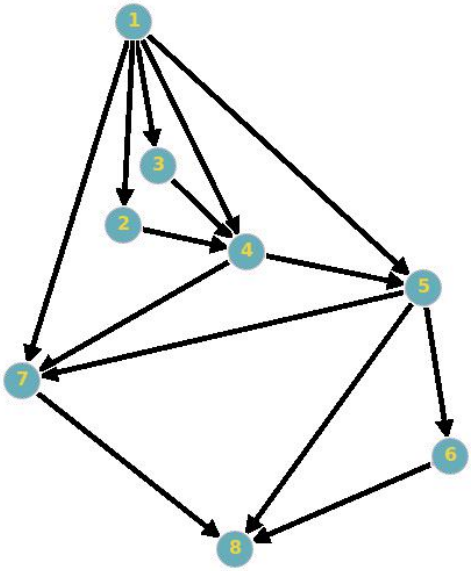
DASAR TEORI

Pada bab ini akan dibahas dasar teori yang digunakan serta analisis algoritma *Minimum Path Cover* dan DFS dengan *Greedy Pruning* sebagai penyelesaian permasalahan MPAEC.

2.1 Deskripsi Permasalahan

Berikut adalah deskripsi permasalah SPOJ - Skiver1 (Skiers) [1]: "Pada suatu gunung terdapat beberapa jalur ski dan 1 ski *lift*. semua jalur dimulai dari stasiun ski *lift* di atas lalu menuju stasiun ski lift di bawah. Setiap pagi beberapa pekerja akan mengecek jalur ski. Mereka bersama-sama naik lift ke stasiun atas lalu berselancar di jalur ski dan setiap pekerja hanya berselancar 1 kali. Jalur yang dilewati tiap pekerja mungkin ada yang sama, setiap jalur selalu mengarah ke bawah. Peta dari jalur ski adalah jaringan antara lapangan yang dihubungkan dengan jalan. Setiap lapangan berada pada ketinggian yang berbeda dan 2 lapangan hanya dihubungkan dengan maksimal 1 jalan, lalu berapakah jumlah pekerja yang dibutuhkan untuk mengecek semua jalur ski tersebut?".

Dari deskripsi di atas, dijelaskan bahwa setiap jalur selalu mengarah ke bawah dengan ketinggian tiap lapangan yang berbeda, dan juga jalur dimulai dari 1 stasiun ski *lift* di atas menuju stasiun ski *lift* di bawah sehingga model graf untuk permasalahan di atas adalah graf planar berarah yang dimulai dari 1 *vertex* dan berakhir ke 1 *vertex*, contoh ilustrasinya sebagai berikut.



Gambar 2.1 Contoh graf permasalahan SPOJ-Skiver1 (Skiers)

Dari deskripsi di atas juga dijelaskan hal yang ingin dicapai yaitu mencari jumlah pekerja minimal untuk mengecek semua jalur ski, Hal ini bisa dianggap setara dengan mencari jumlah *path* minimal untuk mencakup semua jalur ski, atau bisa disebut dengan mencari *Minimum Path of All Edge Coverage* (MPAEC). Sebagai contoh, graf pada Gambar 2.1 memiliki jumlah minimal path sebanyak 5 untuk mencakup seluruh *edge* pada graf, seperti dibawah ini:

path 1: [1,7,8]
path 2: [1,2,4,8]
path 3: [1,3,4,5,7,8]
path 4: [1,4,5,8]
path 5: [1,2,9,13,15]

Terdapat 2 pendekatan penyelesaian yang akan dibahas yaitu pendekatan dengan *Minimum Path Cover* dan pendekatan dengan DFS (*Depth Firsh Search*) dengan *Greedy Pruning*. Pada pendekatan dengan *Minimum Path Cover* akan dibahas beberapa algoritma terkait seperti algoritma *Hopcroft-Karp*, *Topological*

Sort serta juga *Transitive Closure.* Dibahas pula penggunaan algoritma-algoritma tersebut untuk menyelesaikan permasalahan MPAEC.
Pada pendekatan DFS dengan *Greedy Pruning* juga akan dibahas karakteristik data uji soal SPOJ - Skiver1 (Skiers) yang bisa dimanfaatkan dalam penyusunan strategi penyelesaian MPAEC menggunakan DFS dengan *Greedy Pruning.*

2.2 Deskripsi Umum

Pada subbab ini akan dibahas beberapa teori yang akan digunakan pada penyelesaian dengan *Minimum Path Cover* maupun *DFS* dengan *Greedy Pruning.*

2.2.1 Minimum Path Cover

Path cover adalah *set of vertex* dari *directed graph* dimana masing-masing *vertex* terhubung setidaknya dengan 1 *path*, sedangkan *minimum path cover* adalah permasalahan di mana tujuannya meminimumkan jumlah *set* dari *path cover* [2].

Minimum Path Cover pada *non-Directed Acyclic Graph* tergolong *NP-Hard* namun pada *Directed Acyclic Graph* (Graf Berarah) tergolong class P yang bisa diselesaikan dengan algoritma *Maximum Cardinality Bipartite Matching* (MCBM) [3].

2.2.2 Algoritma Hopcroft-Karp

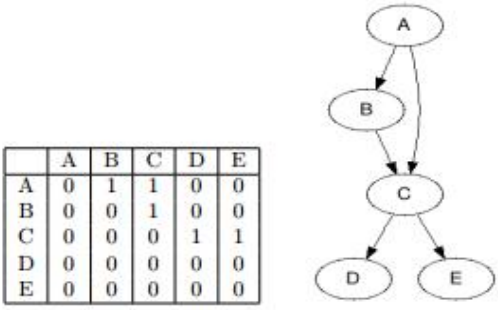
Pada *Minimum Path Cover* terdapat bagian di mana perlu untuk mencari *Maximum Cardinality Bipartite Matching* (MCBM) [4], yaitu banyaknya *match* yang terjadi antara 2 set pada *bipartite graph* (contoh pada Gambar 2.11). Dengan berbagai pilihan algoritma yang ada, penulis memilih algoritma Hopcroft-Karp karena Algoritma Hopcroft-Karp adalah algoritma penyelesaian MCBM yang memiliki kompleksitas $O(|E| \sqrt{|V|})$ [4].

2.2.3 Transitive Closure

Pada *Minimum Path Cover* dengan kasus khusus dimana sebuah *vertex* bisa menjadi anggota lebih dari 1 *path,* maka diperlukan bentuk *transitive closure* [4] dari graf awal, Detail tentang hal ini akan dibahas pada subbab 2.3.3 selanjutnya.
Transitive Closure memiliki definisi sebagai berikut:

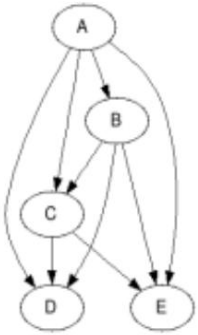
Untuk $a,b,c \in A$, maka (a,b) , $(b,c) \in R \Rightarrow (a,c)$.

Gambar 2.2 adalah contoh definisi DAG sederhana serta Gambar 2.3 adalah bentuk *transitive closure*-nya.



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	0
E	0	0	0	0	0

Gambar 2.2 DAG sebelum menjadi bentuk transitive closure



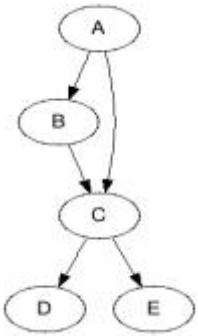
	A	B	C	D	E
A	0	1	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	0	0
E	0	0	0	0	0

Gambar 2.3 *Transitive Closure* dari DAG

2.2.4 Topological Sort

Dalam proses pembangunan bentuk *transitive closure* pada graf terdapat kebutuhan untuk mengetahui urutan topologi pada graf, hal ini akan dibahas lebih detail di subbab 2.3.4.

Topological Sort adalah pengurutan *vertex* dari DAG ke dalam bentuk linier dimana setiap *edge uv*, *vertex u* terletak sebelum *vertex v* dalam urutan [4].



Gambar 2.4 *Topological sort* Dari DAG

Dari Gambar 2.4, maka hasil *topological sort*-nya adalah A,B,C,D,E atau A,B,C,E,D. Hasil *topological sort* memang bisa lebih dari 1 solusi. *Topological sort* bisa dilakukan dengan mengimplementasikan DFS dan struktur data Stack.

2.3 Strategi Penyelesaian dengan Minimum Path Cover

Pada subbab ini akan dijelaskan tahap-tahap penyelesaian dengan *Minimum Path Cover*. Pada subbab 2.3.1 akan dibahas hubungan permasalahan MPAEC dengan *Minimum Path Cover*, serta juga pada subbab 2.3.2 sampai 2.3.5 akan dibahas tahap-tahap penyelesaian permasalahan MPAEC dengan model penyelesaian *Minimum Path Cover*.

2.3.1 Minimum Path Cover pada Permasalahan MPAEC

Pada permasalahan MPAEC, setiap *edge* setidaknya menjadi anggota dari suatu *path.* Pada contoh kasus uji Gambar 2.1, jumlah *path* minimal yang dibutuhkan adalah 5, dengan kumpulan *path* sebagai berikut:

path 1: [1,7,8]
path 2: [1,2,4,8]
path 3: [1,3,4,5,7,8]
path 4: [1,4,5,8]
path 5: [1,2,9,13,15]

setiap *vertex* (dari *vertex* 1 sampai *vertex* 15) menjadi anggota dari salah satu atau beberapa *path*, solusi diatas tidaklah unik, artinya ada solusi lain yang serupa dengan jumlah *path* sebanyak 8.

Berdasarkan deskripsi sebelumnya, terdapat persamaan dengan permasalahan *Minimum Path Cover* yaitu mencari *set of vertex* dimana setiap *vertex* menjadi anggota suatu *set*.
Perbandingan kedua permasalahan tersebut adalah sebagai berikut.

- *Minimum Path of All Edges Coverage* (MPEC) *:* mencari *set of edges* dimana setiap *edge* menjadi anggota suatu *path.*
- *Minimum Path Cover: mencari set of vertices* dimana setiap *vertex* menjadi anggota suatu *path.*

Dari perbandingan tersebut, dapat disimpulkan permasalahan MPAEC bisa diselesaikan dengan jalan menyelesaikan permasalahan *Minimum Path Cover.* Lebih dahulu dimodelkan data permasalahan MPAEC menjadi bentuk model data *Minimum Path Cover,* yaitu dengan mentransformasikan semua *edge* pada MPAEC menjadi *vertex* dan *vertex* menjadi *edges*, hal ini akan dibahas pada bahasan 2.3.2.

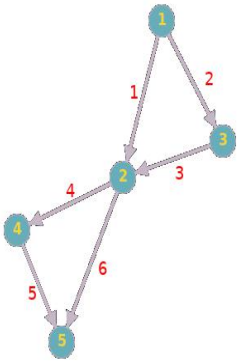
2.3.2 Transformasi Edge menjadi Vertex dan Vertex menjadi Edge

Transformasi *edge* menjadi *vertex* dan *vertex* menjadi *edge* bisa dilakukan dengan cara menghubungkan setiap *edge* pada *vertex u* dengan setiap *edge* pada *vertex v.*

Langkah-langkah transformasi di atas dapat dilihat dibawah ini.

1. Lakukan penomoran pada *edge.*
2. Buat *adjacency list adj* sebagai representasi graf yang baru dengan panjang sama dengan banyaknya edge yang ada pada graf asal.
3. Lakukan iterasi untuk setiap *edge* e_s pada *edge list*, untuk setiap e_x dimana e_x adalah *edge* dimana *vertex source*-nya adalah v_d (*vertex destination* pada e_s), hubungkan *vertex* dengan nomor e_x dengan *vertex* dengan nomor e_s dan disimpan pada *adj*.

Gambar 2.5 menggambarkan contoh sederhana graf untuk permasalahan MPAEC.



Gambar 2.5 Graf untuk permasalahan MPAEC dengan *edge* bernomor

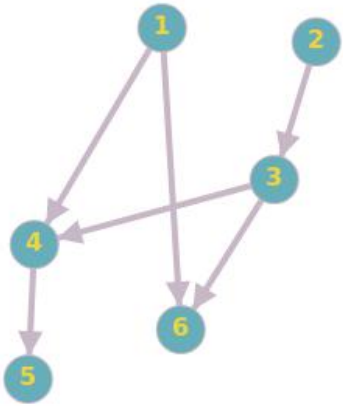
Tabel 2.1 *Matrix edge list* pada DAG

Nomor Edge	Nomor Vertex Source	Nomor Vertex Destination
1	1	2
2	1	3
3	3	2
4	2	4
5	4	5
6	2	5

Pada Gambar 2.5, setiap *edge* diberi nomor yang akan menjadi nomor *vertex* yang baru setelah transformasi.
Tabel 2.2 adalah proses iterasinya.

Tabel 2.2 Urutan hasil proses iterasi transformasi

urutan iterasi	edge list
1	[1,4],[1,6]
2	[1,4],[1,6],[2,3]
3	[1,4],[1,6],[2,3],[3,4],[3,6]
4	[1,4],[1,6],[2,3],[3,4],[3,6],[4,5]
5	[1,4],[1,6],[2,3],[3,4],[3,6],[4,5]
6	[1,4],[1,6],[2,3],[3,4],[3,6],[4,5]

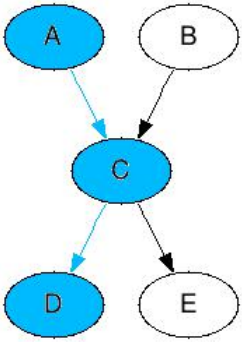


Gambar 2.6 Hasil transformasi graf dari *edge* menjadi *vertex* dan sebaliknya

Gambar 2.6 menggambarkan representasi graf hasil transformasi. Untuk desain *pseudocode* tahap ini bisa dilihat pada subbab 3.2.3.

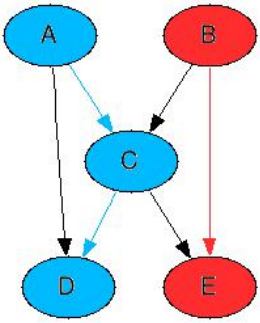
2.3.3 Minimum Path Cover dengan Non-Unique Vertex

Pada permasalahan MPAEC setiap *edge* bisa berada di lebih dari 1 *path,* dan karena dilakukan transformasi *edge* menjadi *vertex* maka setiap *vertex* hasil transformasi bisa dilewati lebih dari 1 *path,* namun jika dilakukan pencarian *Minimum Path Cover* langsung pada graf hasil transformasi maka akan menghasilkan jawaban yang keliru, misal *Minimum Path Cover* untuk DAG sederhana pada Gambar 2.7a, hasil yang didapat adalah 1 sedangkan hasil yang benar adalah 2.



Gambar 2.7 *Minimum Path Cover* pada DAG

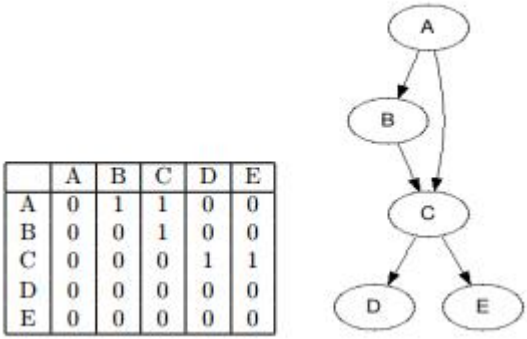
Maka dari itu sebelum dilakukan pencarian *Minimum Path Cover*, Graf harus diubah dulu menjadi bentuk *Transitive Closure*.



Gambar 2.8 *Minimum Path cover* pada bentuk *transitive closure* dari DAG

Untuk pembahasan bagaimana cara membangun *Transitive Closure* akan dibahas pada bahasan 2.3.4.

2.3.4 Membangun Bentuk Transitive Closure dari DAG



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	0
E	0	0	0	0	0

Gambar 2.9 DAG dengan *incidency matrix*-nya

Jika terdapat *edge* berarah yang menghubungkan *vertex* v_1 dengan *vertex* v_2 maka aksesibilitas v_1 merupakan sub-aksesibilitas dari v_2. Misal pada Gambar 2.9, aksesibilitas *vertex* B: {A} merupakan subset dari aksesibilitas *vertex* C: {A,B}, dengan demikian aksesibilitas setiap *vertex* / T*ransitive Closure* bisa didapatkan dengan teknik *Dynamic Programming* (DP) pada aksesibilitas tiap *vertex*.

DP pada *vertex* dilakukan untuk menyalin aksesibilitas dari suatu *vertex* ke *vertex* selanjutnya. Karena suatu *vertex* bisa saja menjadi *destination* dari beberapa *vertex* lainnya (contoh *vertex* C menjadi *destination* dari *vertex* A dan B) maka diperlukan urutan transfer/penelusuran pada *vertex,* hal ini bisa dicapai dengan menggunakan hasil dari *Topological Sort* sebagai urutan penelusuran pada DP.

Simulasi pembentukan *Transitive Closure* dapat dilihat seperti di bawah ini dengan contoh yang diambil dari Gambar 2.9.

Hasil *Topological Sort* secara *ascending*: [D,E,C,B,A]

Tabel 2.3a Hasil iterasi 1 dari *vertex* D

	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

Tabel 2.3b Hasil iterasi 1 dari *vertex* E

	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

Tabel 2.3c Hasil iterasi 1 dari *vertex* C

	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	1	1
D	0	0	0	0	0
E	0	0	0	0	0

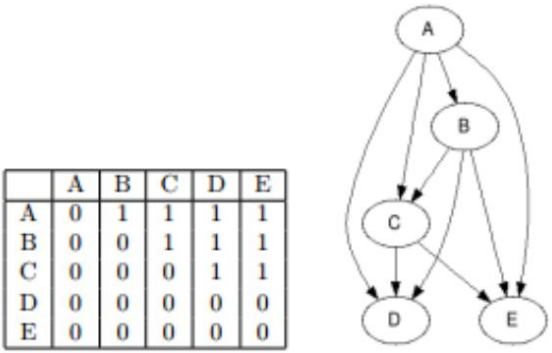
Tabel 2.3d Hasil iterasi 1 dari *vertex* B

	A	B	C	D	E
A	0	0	0	0	0
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	0	0
E	0	0	0	0	0

Tabel 2.3e Hasil iterasi 1 dari *vertex* A

	A	B	C	D	E
A	0	1	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	0	0
E	0	0	0	0	0

Gambar 2.10 adalah graf dalam bentuk *transitive closure*-nya:



	A	B	C	D	E
A	0	1	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	0	0
E	0	0	0	0	0

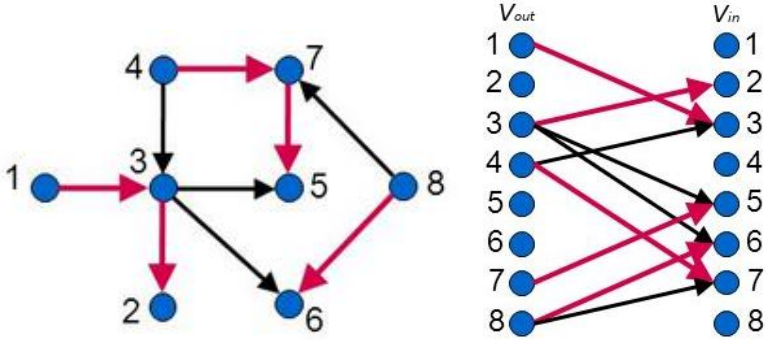
Gambar 2.10 DAG dalam bentuk *transitive closure* dengan *incidency matrix*-nya

Untuk desain *pseudocode* pada tahap ini bisa dilihat pada subbab 3.2.5.

2.3.5 Maximum Matching dan Minimum Path Cover

K¨onigs theorem [4] menyatakan:
"*let G be bipartite graph. Then the maximal cardinality of a matching in G equals the minimal cardinality of a vertex cover: α'(G) = β(G)*.

Untuk mencari *minimum cardinality of a vertex cover,* suatu DAG bisa diubah ke dalam bentuk *bipartite graph* dengan memecah setiap *vertex* menjadi v_{in} dan v_{out} seperti pada Gambar 2.11.



Gambar 2.11DAG dan bentuk bipartite graph-nya

Dengan mencari *Maximum Cardinality Bipartite Graph* (MCBM), maka akan didapatkan *Minimum Cardinality of a Vertex Cover* (MCVC) atau jumlah minimum *vertex* yang dibutuhkan sehingga setiap *edge* terhubung setidaknya pada satu *vertex.* Dari kumpulan *edge* hasil *matching* pada MCVC, bisa didapatkan *path* dengan menggabungkan hasil *matching,* misal dari (v_{out} 1, v_{in} 3) dan (v_{out} 3, v_{in} 2) maka bisa menjadi 1 *path* [1,3,2]. Setiap *path* yang didapat mencakup semua *vertex* dengan jumlah *path* yang minimal dan bisa disebut sebagai *Minimum Path Cover*.

Jumlah *path* yang dihasilkan dari MCBM bisa didapatkan dengan mencari jumlah v_{in} yang bukan menjadi hasil *match* pada MCBM

karena v_{in} tersebut menjadi *source* awal dari *path*. Persamaan 2.1 adalah persamaan untuk menghitung jumlah *Minimum Path Cover*.

$$p = n - M \quad (2.1)$$

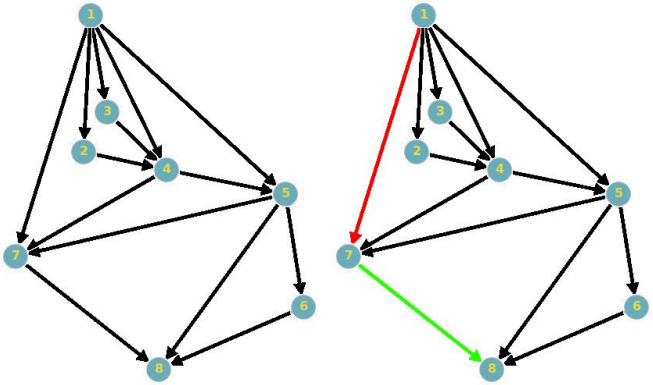
p adalah jumlah path pada *Minimum Path Cover, n* adalah jumlah *vertex* dan *m* adalah maksimum jumlah match pada *bipartite graph*, Desain pseudocode berdasarkan subbab ini bisa dilihat pada subbab 3.2.

2.4 Strategi Penyelesaian menggunakan DFS dengan *Greedy Pruning*

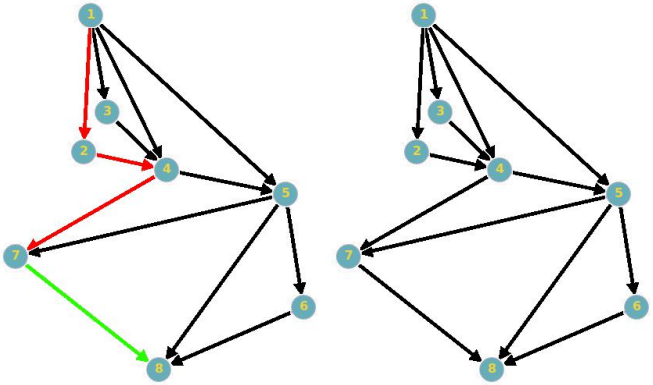
Pada subbab ini akan dijelaskan intuisi visual pada permasalahan MPAEC pada SPOJ - Skiver1 (Skiers), serta penyelesaiannya dengan metode DFS dan *Greedy Pruning*.

2.4.1 Intuisi Visual pada Permasalahan MPAEC

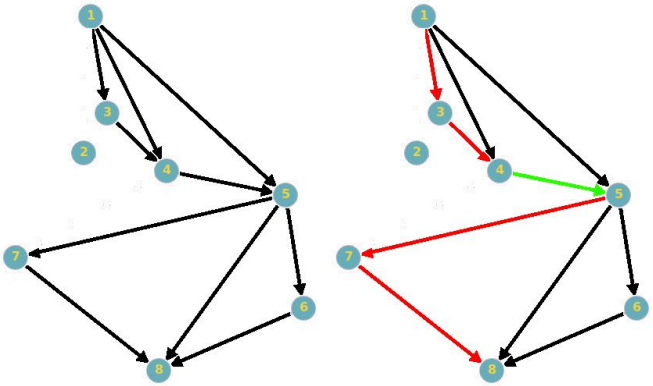
Permasalahan MPAEC bisa diselesaikan dengan intuisi visual dengan memilih *path* paling kiri, lalu menghapus *edge-edge* pada *path* yang tidak akan digunakan pada iterasi selanjutnya. Sebagai contoh, Gambar 2.12a - 2.12e adalah visualisasi iterasi secara intuitif dengan *edge* berwarna hijau adalah *edge* yang masih diperlukan pada iterasi berikutnya dan yang merah adalah *edge* yang tidak lagi diperlukan pada iterasi berikutnya.



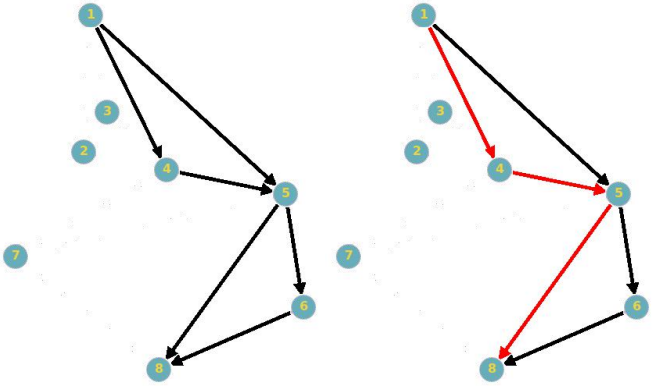
Gambar 2.12a Iterasi penyelesaian intuitif ke-1



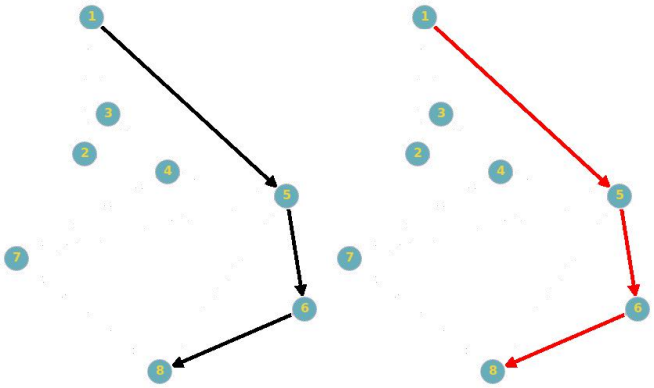
Gambar 2.12b Iterasi penyelesaian intuitif ke-2



Gambar 2.12c Iterasi penyelesaian intuitif ke-3



Gambar 2.12d Iterasi penyelesaian intuitif ke-4



Gambar 2.12e Iterasi penyelesaian intuitif ke-5

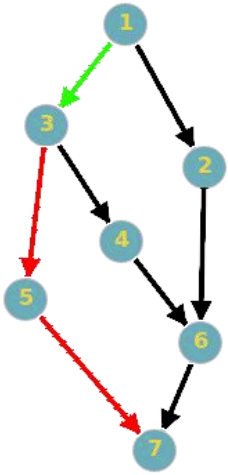
Dari proses iterasi pada Gambar 2.12a - Gambar 2.12e dihasilkan 5 *path* minimal untuk mencakup keseluruhan graf. Penyelesaian intuitif diatas mudah untuk dilakukan secara visual karena urutan dan posisi *edge* terlihat, namun hal ini akan sulit jika diselesaikan dengan program komputer karena terdapat hambatan dimana program harus mengetahui urutan *edge* pada graf tersebut. Untuk permasalahan MPAEC pada kasus uji soal SPOJ - Skiver1 (Skiers), data uji yang diberikan sudah terurut dari kiri ke kanan sehingga tidak ada hambatan dalam mengetahui urutan *edge*. Dengan demikian penyelesaian intuitif diatas bisa dibuatkan algoritma penyelesaiannya dengan syarat data masukan *edge* sudah terurut.

Dengan intuisi visual, setiap *edge* pada *path* yang tidak akan digunakan lagi pada iterasi berikutnya akan dihapus. Maka jika diselesaikan secara algoritmik perlu adanya acuan yang bisa digunakan untuk menentukan apakah suatu *edge* masih dibutuhkan pada iterasi berikutnya atau tidak, hal ini bisa dicapai dengan mengetahui apakah suatu *edge* masih diperlukan oleh *subpath*

sebelum dan selanjutnya dengan memanfaatkan properti *d_in* (*degree in*) dan *d_out* (*degree out*).

2.4.2 Kondisi Subpath Sebelumnya dari Suatu Edge

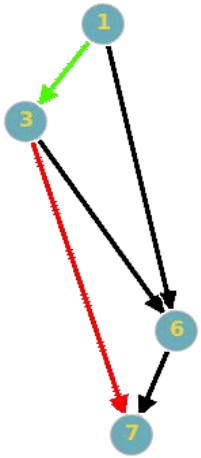
Untuk menentukan suatu *subpath* (i,j) bisa dihapus, harus diketahui apakah *subpath* (i,j) masih diperlukan oleh *subpath* sebelum pada iterasi selanjutnya.
Untuk mengetahui apakah *subpath* (i,j) masih diperlukan oleh subpath sebelumnya adalah dengan mengetahui apakah ada jalur alternatif yang keluar dari v_i , atau bisa didefinisikan dengan kondisi *:* $d\_out_{vi} > 1$ atau $d\_in_{vi} = 0$.



Gambar 2.13 Ilustrasi-2 *pruning* pada DAG

Gambar 2.13 adalah contoh graf dimana *edge* berwarna merah (*path* [3,5,7]) adalah *edge* yang tidak dibutuhkan lagi pada iterasi selanjutnya, sedangkan edge berwarna hijau (*path* [1,3]) masih dibutuhkan pada iterasi berikutnya. Untuk mempermudah analisis,

graf di atas (Gambar 2.13) bisa disederhanakan dengan membaginya dalam beberapa *subpath* dan mengabaikan *vertex* didalam *subpath* yang memiliki d_in=1 dan d_out_v=1.
Edge (1,2) dan (2,6) menjadi *subpath*(1,6), edge (3,4) dan edge (4,6) menjadi *subpath*(3,6), edge (3,5) dan (5,7) menjadi *subpath*(3,7). Dengan mengabaikan *vertex* tertentu, maka graf diatas bisa dipandang sebagai graf berikut (Gambar 2.14).

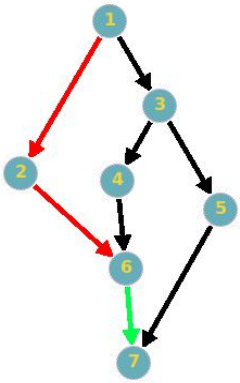


Gambar 2.14 Ilustrasi-2 *pruning* pada DAG dengan *subpath* yang diubah menjadi *edge*

Pada *Subpath* (3,7) Gambar 2.14, tidak ada *subpath* setelahnya, maka pada kasus ini hanya perlu diperhatikan *subpath* sebelumnya. Dengan d_out_{v3} = 2, maka untuk *subpath* (3,7) bisa dihapus karna ada jalur alternatif dari v_3 yaitu *subpath* (3,6).

2.4.3 Kondisi Subpath Selanjutnya dari Suatu Edge

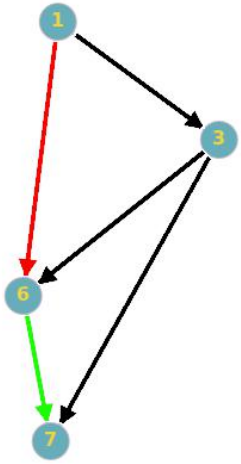
Untuk menentukan suatu *subpath* (i,j) bisa dihapus, harus diketahui apakah *subpath* (i,j) masih diperlukan oleh *subpath* sesudahnya pada iterasi selanjutnya. Untuk mengetahui apakah edge (i,j) masih diperlukan oleh subpath setelahnya adalah dengan mengetahui apakah ada jalur alternatif dimana v_j bisa dicapai tanpa melalui *subpath* (i,j), atau bisa didefinisikan dengan kondisi *:* $d\_in_{vj} > 1$ atau $d\_out_{vj} = 0$.



Gambar 2.15 Ilustrasi-1 *pruning* pada DAG

Gambar 2.15 adalah contoh graf dimana *edge* berwarna merah (pada *subpath* [1,2,6]) adalah *edge* yang tidak dibutuhkan lagi pada iterasi selanjutnya, sedangkan *edge* berwarna hijau (pada *subpath* [6,7]) masih dibutuhkan pada iterasi berikutnya. Untuk mempermudah analisis, graf diatas bisa disederhanakan dengan membaginya dalam beberapa *subpath* dan mengabaikan *vertex* didalam *subpath* yang memiliki d_in=1 dan d_out_v=1.
Edge (1,2) dan (2,6) menjadi *subpath* (1,6), edge (3,4) dan edge (4,6) menjadi *subpath* (3,6), edge (3,5) dan (5,7) menjadi *subpath*

(3,7). Dengan mengabaikan *vertex* tertentu, maka graf di atas bisa dipandang sebagai graf berikut (Gambar 2.16).



Gambar 2.16 Ilustrasi-1 *pruning* pada DAG dengan *subpath* yang diubah menjadi *edge*

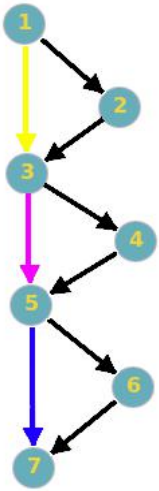
Pada *Subpath* (1,6) Gambar 2.16, tidak ada *subpath* sebelumnya, maka pada kasus ini hanya perlu diperhatikan *subpath* setelahnya. Dengan $d\_in_{v6} = 2$, maka untuk *subpath* (1,6) bisa dihapus karna v_6 bisa diakses melalui *subpath* lain selain *subpath* (1,6) yaitu *subpath* (3,6).

2.4.4 Kondisi Subpath Sebelum dan Selanjutnya dari Suatu Edge

Dari analisis subbab 2.4.2 dan 2.4.3 dapat disimpulkan bahwa untuk melakukan *pruning* harus memperhatikan kondisi *subpath* sebelumnya dan setelahnya dari suatu *edge*.

Pada *subpath* (i,j), jika suatu v_i dan v_j memenuhi 2 kondisi dibawah ini, maka *subpath* (i,j) bisa dihapus.

1. $d\_out_{vi} > 1$ atau $d\_in_{vi} = 0$
2. $d\_in_{vj} > 1$ atau $d\_out_{vj} = 0$



Gambar 2.17 Ilustrasi-3 *pruning* pada DAG

Untuk contoh graf di atas (Gambar 2.17), maka setelah iterasi, *subpath* yang berwarna kuning (*subpath* [1,3]), ungu (*subpath* [3,5]) dan biru (*subpath* [5,7]), *edge* pada *subpath* tersebut bisa dihapus karena hal-hal berikut.

1. *Subpath* kuning, i = 1, j = 3, $d\_out_{vi} = 2$, dan $d\_in_{vj} = 2$, maka *subpath* kuning bisa dihapus
2. *Subpath* ungu, i = 3, j = 5, $d\_out_{vi} = 2$, dan $d\_in_{vj} = 2$, maka *subpath* ungu bisa dihapus
3. *Subpath* biru, i = 5, j = 7, $d\_out_{vi} = 2$, dan $d\_in_{vj} = 2$, maka *subpath* biru bisa dihapus

Untuk pseudocode penyelesaian dapat dilihat pada subbab 3.3

BAB III

DESAIN

Pada bab ini akan dibahas tentang desain algoritma untuk menyelesaikan permasalahan SPOJ - Skiver1 (Skiers) dengan 2 algoritma yang dibahas di bab sebelumnya.

3.1 Desain Umum Sistem

Program akan diawali dengan menerima masukan berupa nilai *n* (banyaknya *vertex* pada graf), dan dilanjutkan dengan *n* baris berikutnya, setiap baris ke-*i* akan diawali dengan masukan nilai *k* (banyaknya masukan selanjutnya pada baris tersebut), dan *k* bilangan berikutnya adalah *vertex* yang dituju oleh *vertex i*. Setelah menerima masukan maka sistem akan mengolah masukan dan kemudian menampilkan hasil akhir berupa jumlah *path* minimal yang dibutuhkan untuk mencakup semua *edge* pada graf.

3.2 Desain Solusi Minimum Path Cover

Pada subbab ini akan dibahas desain *pseudocode* berdasarkan strategi penyelesaian pada subbab 2.3.

3.2.1 Desain Class Graph

Class Graph adalah *class* yang digunakan sebagai penyimpan data graf, dan juga menyimpan data-data yang diperlukan didalam graf seperti jumlah *vertex* dan jumlah *edge.* Desain *class Graph* adalah sebagai berikut.

```
class Graph:
  private int : V,edgeCounter
  private vector of vector of int : adj

  public Graph(int V):
    adj.resize(V+1);

  public addEdge(u, v):
    adj[u].add(v)
    edgeCounter+=1

  public vertexCount():
    return adj.size()-1

  public edgeCount():
    return edgeCounter

  public getAdjacencyList():
    return adj
```

Kode Sumber 3.1 *Pseudocode class* Graph

Fungsi *addEdge* adalah fungsi yang digunakan untuk menambah *directed edge* pada graf dengan parameter (u, v) yang berarti *edge* berarah dari *u* ke *v,* didalam fungsi ini juga akan diperbarui nilai untuk *edgeCounter* yaitu variabel yang digunakan untuk menghitung jumlah *edge.* Selanjutnya fungsi *vertexCount* adalah fungsi yang digunakan untuk mendapatkan jumlah *vertex* pada graf, lalu *edgeCount* adalah fungsi yang digunakan untuk mendapatkan jumlah *edge* pada graf, dan terakhir adalah fungsi *getAdjacencyList* yaitu fungsi yang digunakan untuk mendapatkan representasi graf dalam bentuk *adjacency list.*

3.2.2 Desain Class Algoritma Hopcroft-Karp (*HopcroftKarp*)

C*lass HopcroftKarp* adalah *class* yang menyimpan data *bipartite graph* serta algoritma pencarian *Maximum Cardinality Bipartite Matching* menggunakan algoritma Hopcroft-Karp yang akan digunakan pada proses pencarian *Minimum Path Cover*. Kode Sumber 3.2 adalah desain *class HopcroftKarp* [5] dan Tabel 3.1 adalah penjelasan variabel dari *class HopcroftKarp* .

```
1 public class HopcroftKarp:
2   private vector of vector of int : G
3   private int : n, m
4   private vector of int : match, dist
5
6   public HopCroftKarp(n, m)
7   public addEdge(u, v)
8   public findMaxMatch()
9   private dfs()
10  private bfs()
```

Kode Sumber 3.2a *Pseudocode class HopcroftKarp*

```
1 HopCroftKarp::HopCroftKarp(int n,int m):
2   this->n = n
3   this->m = m
4   G.resize(n+m+1)
5   match.resize(n+m+1)
6   dist.resize(n+m+1)
7   fill(match, 0)
```

Kode Sumber 3.2b *Pseudocode constructor class HopcroftKarp*

Tabel. 3.1 Tabel Penjelasan variabel *class HopcroftKarp*

Variabel	Keterangan
G	representasi graf dalam bentuk *adjacency list*
n	$\|U\|$, pada $G = (U, V, E)$
m	$\|V\|$, pada $G = (U, V, E)$
match	menyimpan hasil *match*, jika pada index ke-i terdapat nilai j, maka ada (i, j) adalah hasil *match*
dist	digunakan untuk proses pembentukan *augmenting path* pada algoritma

Pada Kode Sumber 3.2b bagian *constructor, vector G, match* dan *dist* diubah *size*-nya sebesar (n+m+1) karena *vertex bipartite graph* bagian kiri / *U* akan disimpan dari indeks i, $1 \leq i \leq n$, dan *bipartite graph* bagian kanan / *V* akan disimpan di indeks j, $n+1 \leq j \leq n+m$, maka untuk fungsi *addEdge*(*u, v*) maka *v* akan ditambah sebesar *n*.

```
1 HopCroftKarp::addEdge(u, v):
2     G[u].add(n+v)
3     G[n+v].add(u)
```

Kode Sumber 3.2c *Pseudocode* fungsi *addEdge*

```
HopcroftKarp::bfs():
  queue of int: Q;
  for i=1 to n
    if match[i] == NIL:
      dist[i] = 0
      Q.push(i)
    else:
      dist[i] = INF

  dist[NIL] = INF

  while Q.empty()==false:
    u = Q.front()
    Q.pop()
    if u != NIL:
      len = G[u].size()
      for i=0 to len-1:
        v = G[u][i]
        if dist[match[v]]==INF:
          dist[match[v]] = dist[u] + 1
          Q.push(match[v])

  return dist[NIL]!=INF
```

Kode Sumber 3.2d *Pseudocode* fungsi *bfs*

```
HopcroftKarp::findMaxMatch:
  matching = 0
  while bfs():
    for i=1 to n:
      if match[i]==NIL && dfs(i):
        matching+=1
  return matching
```

Kode Sumber 3.2e *Pseudocode* fungsi *findMaxMatch*

```
HopcroftKarp::dfs(u):
  if u!=NIL:
    len = G[u].size()
    for i=0 to len-1:
      v = G[u][i]
      if dist[match[v]]==dist[u]+1:
        if dfs(match[v]):
          match[v] = u
          match[u] = v
          return true
    dist[u] = INF
    return false

  return true
```

Kode Sumber 3.2f *Pseudocode* fungsi *dfs*

Fungsi *findMaxMatch* pada Kode Sumber 3.2e adalah fungsi pencari *Max Cardinality Bipartite Matching*, fungsi ini memiliki kompleksitas $O(E\sqrt{V})$,di mana E adalah jumlah *edge* dan V adalah jumlah *vertex.*

3.2.3 Desain Fungsi Transformasi Edge ke Vertex (*transformEdgeToVertex*)

Fungsi ini (Kode Sumber 3.3) menerima masukan berupa *class* Graph dan mengembalikan *class Graph* hasil transformasi, desain fungsi ini mengacu pada dasar teori yang sudah dijelaskan pada subbab 2.3.2. Dalam proses fungsi transformasi ini, mula-mula dilakukan penomoran pada *edge* dengan kompleksitas $O(V^2)$, lalu dilanjutkan dengan menghubungkan *edges* yang telah dinomori tersebut dengan memanfaatkan data pada graf masukan, proses ini berjalan dengan kompleksitas $O(V*E^2)$, dengan n merupakan jumlah *vertex* dan m melambangkan jumlah *edge.*

```
transformEdgeToVertex(graph):
  V = graph.vertexCount()
  vector of vector of int: edgesId
  edgesId.resize(V+1)
  adj = graph.getAdjacencyList()
  edge_id = 1
  newGraph = Graph(graph.edgeCount())

  for i=1 to V:
    foreach adj[i] as v:
      edgesId[i].add(edge_id)
      edge_id+=1

  for i=1 to V:
    for j=0 to edgesId[i].size()-1:
      v = adj[i][j]
      foreach edgesId[v] as v_edge:
        newGraph.addEdge(edgesId[i][j], v_edge)

  return newGraph
```

Kode Sumber 3.3 *Pseudocode* fungsi *transformEdgeToVertex*

3.2.4 Desain Fungsi Pencarian Urutan Topologis (*getTopologicalSort*)

Fungsi ini menerima masukan berupa *class* Graph dan mengembalikan urutan topologis yang akan digunakan pada proses pembentukan *Transitive Closure*, fungsi ini memiliki fungsi pembantu yaitu *topologicalSortUtil* dimana fungsi pembantu ini adalah fungsi rekursif yang digunakan untuk menelusuri graf dengan *root* yang ditentukan dari fungsi *getTopologicalSort.*
Fungsi ini berjalan dengan kompleksitas O*(V+E),* di mana *V* yaitu jumlah *vertex* dan *E* melambangkan jumlah *edge. Pseudocode* dapat dilihat pada Kode Sumber 3.4.

```
topologicalSortUtil(u, adj, visit, result):
  visit[u] = true

  foreach adj[u] as v:
    if !visit[v]:
      topologicalSortUtil(
        v, adj, visit, result
      )
  result.push(u)
getTopologicalSort(graph):
  V = graph.vertexCount()
  vector of bool: visit
  visit.resize(V+1)
  fill(visit, false)
  stack of int: result

  for i = 1 to V:
    if visit[i] == false:
      topologicalSortUtil(
        i,
        graph.getAdjacencyList(),
        visit, result
      )

  return result
```

Kode Sumber 3.4 *Pseudocode* fungsi *getTopologicalSort*

3.2.5 Desain Fungsi Pembentukan Transitive Closure (*getTransitiveClosure*)

Fungsi ini mengacu pada dasar teori pada subbab 2.3.3 dan 2.3.4. Fungsi ini menerima masukan berupa *class Graph* dengan asumsi grafnya adalah *Directed Acyclic Graph*, lalu hasil fungsi ini adalah bentuk *transitive closure* dari graf masukan dalam representasi *incidency matrix*. Teknik membangun *transitive closure* ini sudah dibahas di bab 2.3.4.

```
getTransitiveClosure(graph):
  seq = getTopologicalSort(graph)
  adj = graph.getAdjacencyList()
  N = adj.size()-1

  vector of bitset(15005): tcAdj
  tcAdj.resize(N+1)
  while !seq.empty():
    u = seq.top()
    seq.pop()
    foreach adj[u] as v:
      tcAdj[v]|=tcAdj[u]
      tcAdj[v][u] = true

  return tcAdj
```

Kode Sumber 3.5 *Pseudocode* fungsi *getTransitiveClosure*

Fungsi ini memiliki kompleksitas *O(V*E)* dengan *V* yaitu jumlah *vertex* dan *E* yaitu jumlah *edge.*

3.2.6 Desain Fungsi Pencarian Minimum Path Cover (*findMinimumPathCover*)

Fungsi ini mengacu pada dasar teori subbab 2.3.5 sebelumnya. Fungsi ini menerima masukan objek *class Graph*, dari objek masukan akan ubah menjadi bentuk *bipartite graph*, pada *G = (U, V, E),* maka setiap *vertex source u* pada *directed edge* akan menjadi anggota *set U,* dan setiap *vertex destination v* maka akan menjadi anggota *set V,* dengan relasi *e*=(*u, v*), *e* menjadi anggota *set E.*

```
findMinimumPathCover(graph):
  N = graph.vertexCount();
  bipgraph = HopCroftKarp(N,N)
  tcAdj = getTransitiveClosure(graph)
  for i=1 to N:
    for j = 1 to N:
      if tcAdj[i][j]:
        bipgraph.addEdge(j,i)

  return N - bipgraph.findMaxMatch()
```

Kode Sumber 3.6 *Pseudocode* fungsi *findMinimumPathCover*

Fungsi ini memiliki kompleksitas $O(E^2)$ dengan *m* adalah jumlah *vertex* graf setelah transformasi atau jumlah *edge* pada graf awal.

3.2.7 Desain Fungsi Utama Solusi Minimum Path Cover (*main*)

Berikut adalah fungsi utama program (Kode Sumber 3.7), dimana alur programnya adalah:

1. Menerima masukan.
2. Menambah data ke *class Graph.*
3. Melakukan transformasi *edge* menjadi *vertex.*
4. Mencari *Minimum Path Cover.*
   a. Mencari urutan topologis.
   b. Transformasi ke bentuk Graf *Transitive Closure* dan ke bentuk *Bipartite Graph.*
   c. Mencari *Maximum Cardinality Matching.*
   d. Melakukan perhitungan dengan rumus.

```
main():
  N = input()
  Graph g(N)
  for i=1 to N-1:
    r = input()
    for j=0 to r-1:
      in = input()
      g.addEdge(i,in)

  g = transformEdgeToVertex(g)
  print(findMinimumPathCover(g))
```

Kode Sumber 3.7 *Pseudocode* fungsi utama solusi *Minimum Path Cover*

Kompleksitas untuk keseluruhan program ini adalah $O(V*E^2)$ dimana V adalah jumlah *vertex* dan E adalah jumlah *edge*.

3.3 Desain Solusi DFS dengan Greedy Pruning

Pada subbab ini akan dibahas desain *pseudocode* berdasarkan strategi penyelesaian pada subbab 2.4 sebelumnya.

3.3.1 Desain Fungsi DFS dengan Greedy Pruning (*dfsWithPruning*)

Fungsi ini adalah fungsi DFS dengan *Greedy Pruning*, seperti yang telah dijelaskan pada bab 2.4 sebelumnya, terdapat aturan *pruning* yang bisa dilakukan jika edge (i,j) memenuhi 2 kondisi di bawah ini:

1. $d\_in_{vj} > 1$ dan ($d\_in_{vi} = 0$ atau $d\_out_{vi} > 1$)
2. $d\_out_{vi} > 1$ dan ($d\_out_{vj} = 0$ *atau* $d\_in_{vj} > 1$)

maka desain algoritma DFS dengan *pruning* adalah sebagai berikut.

```
array of deque of int adj
array of int d_in

dfsWithPruning(u, hasOutBefore):
  if(!adj[u].empty()):
    hasOutBefore = (hasOutBefore && d_in[u] == 1)
                    || adj[u].size() > 1
    next = adj[u].front();
    while(d_in[next] == 1 && adj[next].size() == 1):
        next = adj[next].front()
        adj[u].pop_front()
        adj[u].push_front(next)
    }
    hasInAfter = dfsWithPrunning(
                  adj[u].front(),
                  hasOutBefore
                 )
    if(hasOutBefore && hasInAfter):
      d_in[adj[u].front() ]--
      adj[u].pop_front()

    return d_in[u]>1 || adj[u].size()==0

  else:
    return true
```

Kode Sumber 3.8 *Pseudocode* fungsi *dfsWithPruning*

Fungsi ini memiliki kompleksitas sebesar $O(n+m)$.

3.3.2 Desain Fungsi Utama Solusi DFS dengan Greedy Pruning(*main*)

Fungsi ini adalah fungsi utama yang akan dipanggil pertama ketika mengeksekusi program, fungsi ini bertugas untuk menerima

masukan data, dan juga melakukan pengulangan pemanggilan fungsi *DFS* hingga semua *path* sudah terhapus.

Desain fungsi utama bisa dilihat pada Kode Sumber 3.9.

```
main():
  N = input()

  for i=1 to N-1:
      r = input()
      for j=0 to r-1:
          in = input()
          adj[i].push_back(in)
          d_in[in]++

  d_in[1]=1
  int ans = 0
  while !adj[1].empty():
      ans++
      dfsWithPruning(1,true)

  print(ans)
```

Kode Sumber 3.9 *Pseudocode* fungsi utama menggunakan DFS dengan *Greedy Pruning*

Halaman ini sengaja dikosongkan

BAB IV

IMPLEMENTASI

Pada bab ini akan dibahas implementasi dari *pseudocode* bab 3 dalam bahasa pemrogaman C++. Subbab 4.1-4.7 adalah implementasi solusi dengan algoritma *Minimum Path Cover* dan subbab 4.8-4.9 adalah implementasi solusi dengan algoritma *greedy.*

4.1 Lingkungan Implementasi

Lingkungan implementasi dalam pembuatan Tugas Akhir yang meliputi perangkat keras dan perangkat lunak yang digunakan untuk menjalankan algoritma penyelesaian dengan *Minimum Path Cover* dan algoritma DFS dengan *Greedy Pruning* pada permasalahan klasik SPOJ - Skiver1 (Skiers) adalah sebagai berikut.

1. Perangkat Keras
   - Processor Intel Core i5-5200U @ CPU @ 2.20GHz × 2.
   - RAM 11.6 GiB.
2. Perangkat Lunak
   - Sistem Operasi Linux Mint 19 Cinnamon.
   - IDE Codeblocks.
   - gcc version 7.4.0

4.2 Rancangan Data

Rancangan data masukan dan keluaran dijelaskan pada dua subbab berikut ini.

4.2.1 Data Masukan

Data masukan adalah data yang akan diproses oleh program sebagai masukan yang akan digunakan dalam implementasi algoritma yang telah dirancang dalam Tugas Akhir ini.

Data masukan berupa berkas teks yang berisi data dengan format yang telah ditentukan pada deskripsi permasalahan klasik SPOJ - Skiver1(Skiers). Pada masing-masing berkas data masukan, baris pertama berupa nilai *n* (banyaknya *vertex* pada graf), dan dilanjutkan dengan *n* baris berikutnya, setiap baris ke-*i* akan diawali dengan masukan nilai *k* (banyaknya masukan selanjutnya pada baris tersebut), dan *k* bilangan berikutnya adalah *vertex* yang dituju oleh *vertex i*. Setelah menerima masukan maka sistem akan mengolah masukan dan kemudian menampilkan hasil akhir berupa jumlah *path* minimal yang dibutuhkan untuk mencakup semua *edge* pada graf.

4.2.2 Data Keluaran

Data keluaran yang dihasilkan oleh program berupa suatu bilangan *integer* yaitu jumlah minimal pekerja yang dibutuhkan untuk melalui semua jalur ski.

4.3 Implementasi Solusi dengan Algoritma *Minimum Path Cover*

Pada subbab ini akan dijelaskan tentang implementasi solusi dengan algoritma *Minimum Path Cover* dalam bahasa pemrograman C++ berdasarkan desain yang dijelaskan pada Bab III.

4.3.1 *Header* yang Digunakan

Implementasi solusi menggunakan algoritma *Minimum Path Cover* pada bahasa C++ membutuhkan 7 buah *header* (Kode Sumber 4.1).

```
1 #include <cstdio>
2 #include <vector>
3 #include <queue>
4 #include <stack>
5 #include <bitset>
6 #include <climits>
```

Kode Sumber 4.1 *Header* yang digunakan pada solusi menggunakan *Minimum Path Cover*

Berikut penjelasan *headers* pada Kode Sumber 4.1.

- *Header cstdio* berisi modul untuk menerima masukan dan keluaran.
- *Header vector* berisi struktur data linier untuk menyimpan berbagai macam data.
- *Header queue* berisi struktur data untuk menyimpan data dalam bentuk antrian yang digunakan saat mencari *Max Cardinality Matching* pada *class HoprocroftKarp*.
- *Header stack* berisi struktur data untuk menyimpan data dalam bentuk tumpukan (*stack*) yang digunakan saat mencari urutan topologis graf.
- *Header bitset* berisi struktur data yang digunakan untuk menyimpan data graf dalam bentuk *incidency matrix* yang digunakan sebagai hasil dari fungsi *getTransitiveClosure.*
- *Header climits* berisi konstanta INT_MAX yang digunakan sebagai nilai maksimum jarak yang digunakan kelas *HopcroftKarp*.

4.3.2 Implementasi Class Graph

Implementasi ini (Kode Sumber 4.2) mengacu pada desain di subbab 3.2.1. *Class Graph* adalah *class* yang menyimpan data graf, di dalam *class* ini data graf disimpan dalam bentuk *adjacency list*.

```
 2   int edgeCounter = 0;
 3   vector<vector<int> > adj;
 4 public:
 5   Graph(int V){
 6     adj.resize(V+1);
 7   }
 8
 9   void addEdge(int u, int v){
10     adj[u].push_back(v);
11     edgeCounter++;
12   }
13
14   int vertexCount(){
15     return this->adj.size()-1;
16   }
17
18   int edgeCount(){
19     return this->edgeCounter;
20   }
21
22   vector<vector<int> > getAdjacencyList(){
23     return this->adj;
24   }
25
26 };
```

Kode Sumber 4.2 Implementasi *class Graph*

Tabel 4.1 adalah penjelasan variabel dari Kode Sumber 4.2.

Tabel 4.1 Penjelasan variabel pada *class Graph*

No	Variabel	Tipe Data	Penjelasan
1	edgeCounter	int	jumlah *edge* pada graf
2	adj	vector<vector<int>>	graf dalam bentuk *adjacency list*

4.3.3 Implementasi Class Algoritma Hopcroft-karp *(HopcroftKarp)*

Implementasi ini (Kode Sumber 4.3) mengacu pada desain di subbab 3.2.2. *Class* ini menyimpan data *bipartite graph,* dan juga menyediakan fungsi perhitungan *Max Cardinality Matching* pada *bipartite graph*.

```
class HopCroftKarp {
  vector< vector<int> > G;
  int n, m;
  vector<int> match, dist;

  public:

  HopCroftKarp(int n,int m):
  G(n+m+1),match(n+m+1),dist(n+m+1){
    this->n = n;
    this->m = m;
    fill(match.begin(),match.end(),0);
  }

  HopCroftKarp(){}
```

Kode Sumber 4.3a Implementasi *class HopcroftKarp*

```
17    vector< vector<int> > getGraph(){return G;}
18
19    void addEdge(int u,int v){
20      G[u].push_back(n+v);
21      G[n+v].push_back(u);
22    }
23
24    int findMaxMatch(){
25      int matching = 0, i;
26      while(bfs()){
27        for(i=1; i<=n; i++){
28          if(match[i]==NIL && dfs(i))matching++;
29        }
30      }
31
32      return matching;
33    }
34
35  private:
36    bool bfs(){
37      int i, u, v, len;
38      queue<int> Q;
39      for(i=1; i<=n; i++){
40        if(match[i]==NIL){
41          dist[i] = 0;
42          Q.push(i);
43        }
44        else dist[i] = INF;
45      }
46      dist[NIL] = INF;
47      while(!Q.empty()){
48        u = Q.front(); Q.pop();
```

Kode Sumber 4.3b Implementasi *class HopcroftKarp*

```
49        if(u!=NIL){
50          len = G[u].size();
51          for(i=0; i<len; i++){
52            v = G[u][i];
53            if(dist[match[v]]==INF){
54              dist[match[v]] = dist[u] + 1;
55              Q.push(match[v]);
56            }
57          }
58        }
59      }
60      return (dist[NIL]!=INF);
61    }
63    bool dfs(int u){
64      int i, v, len;
65      if(u!=NIL){
66        len = G[u].size();
67        for(i=0; i<len; i++) {
68          v = G[u][i];
69          if(dist[match[v]]==dist[u]+1){
70            if(dfs(match[v])) {
71              match[v] = u;
72              match[u] = v;
73              return true;
74            }
75          }
76        }
77        dist[u] = INF;
78        return false;
79      }
80      return true;
81    }
82  };
```

Kode Sumber 4.3c Implementasi *class HopcroftKarp*

Tabel 4.2 adalah Tabel Penjelasan variabel untuk Kode Sumber 4.3.

Tabel 4.2 Penjelasan variabel *class HopcroftKarp*

No	**Variabel**	**Tipe Data**	**Penjelasan**
1	G	vector<vector<int>>	menyimpan graf dalam bentuk *adjacency list*
2	n	int	ukuran set *U bipartite graph* (*U, V, E*)
3	m	int	ukuran set *V bipartite graph* (*U, V, E*)
4	match	vector<int>	menyimpan pasangan *match,* untuk setiap nilai v = *match[i],* maka v berpasangan dengan i dan sebaliknya
5	dist	vector<int>	menyimpan jarak penelusuran pada fungsi *bfs*

4.3.4 Implementasi Fungsi Transformasi Edge ke Vertex *(transformEdgeToVertex)*

Implementasi ini mengacu pada desain di subbab 3.2.3. Fungsi *transformEdgeToVertex* adalah fungsi yang akan menerima masukan berupa objek *class* Graph lalu melakukan transformasi graf dimana setiap *edge* pada graf akan menjadi *vertex* pada graf hasil, hasil fungsi adalah objek *class Graph* hasil transformasi.

```
Graph transformEdgeToVertex(Graph graph){
  int V = graph.vertexCount();
  vector<vector<int> > edgesId(V+1);
  vector<vector<int> > adj = graph.getAdjacencyList();
  Graph newGraph(graph.edgeCount());
  int edge_id = 1;
  for(int i=1; i<=V; ++i){
    for(int v: adj[i]){
      edgesId[i].push_back(edge_id++);
    }
  }

  int v;
  for(int i=1; i<=V; ++i){
    for(int j=0; j<edgesId[i].size();++j){
      v = adj[i][j];
      for(int v_edge : edgesId[v]){
        newGraph.addEdge(edgesId[i][j], v_edge);
      }
    }
  }
  return newGraph;
}
```

Kode Sumber 4.4 Implementasi fungsi *transformEdgeToVertex*

Tabel 4.3 Penjelasan variabel fungsi *transformEdgeToVertex*

No	Variabel	Tipe Data	Penjelasan
1	edge_id	int	id edge, nilainya akan dinaikkan 1 setelah dijadikan id oleh suatu *edge*
2	edgesId	vector<vector<int>>	menyimpan relasi antar *edge* dimana tiap *edge* diberikan id
2	newGraph	Graph	graf baru hasil transformasi
3	adj	vector<vector<int>>	*adjacency list* dari graf awal

4.3.5 Implementasi Fungsi Pencarian Urutan Topologis *(getTopologicalSort)*

Implementasi ini (Kode Sumber 4.5) mengacu pada desain di subbab 3.2.4. Fungi *getTopologicalSort* menerima masukan berupa objek *class* Graph (Kode Sumber 4.2) lalu melakukan pencarian urutan topologis, hasil dari fungsi ini adalah urutan topologis dalam tipe data *stack.* Penjelasan variabel pada Kode Sumber 4.5 bisa dilihat pada Tabel 4.4.

```
void topologicalSortUtil(
  int u, vector< vector<int> > adj,
  vector<bool> &visit, stack<int> &result){

  visit[u] = true;

  for (int v : adj[u]){
    if (!visit[v]){
      topologicalSortUtil(
          v,adj,visit,result
      );
    }
  }
  result.push(u);
}
stack<int> getTopologicalSort(Graph graph){
  int V = graph.vertexCount();
  vector<bool> visit;
  stack<int> result;
  visit.resize(V+1,false);
  for (int i = 1; i <= V; i++){
    if (visit[i] == false){
      topologicalSortUtil(
        i,
        graph.getAdjacencyList(),
        visit, result
      );
    }
  }
  return result;
}
```

Kode Sumber 4.5 Implementasi fungsi *getTopologicalSort*

Tabel 4.4 Penjelasan variabel pada fungsi *getTopologicalSort*

No	Variabel	Tipe Data	Penjelasan
1	V	int	jumlah *vertex*
2	visit	vector<bool>	menyimpan tanda apakah suatu *vertex* sudah dikunjungi
3	result	stack<int>	menyimpan hasil dari *topological sort*

4.3.6 Implementasi Fungsi Pembentukan *Transitive Closure* *(getTransitiveClosure)*

Implementasi ini (Kode Sumber 4.6) mengacu pada desain di subbab 3.2.5. Fungsi *getTransitiveClosure* menerima masukan berupa objek *class* Graph dengan hasil berupa graf dengan *transitive closure* dalam bentuk *incidency matrix*. Implementasi dapat dilihat dihalaman berikutnya.

```
vector<bitset<15005>> getTransitiveClosure
  (Graph graph){

  stack<int> seq = getTopologicalSort(graph);
  vector<vector<int>> adj
    = graph.getAdjacencyList();

  int N = adj.size()-1;
  vector<bitset<15005>> tcAdj;
  tcAdj.resize(N+1);
  int u;
```

Kode Sumber 4.6a Implementasi fungsi *getTransitiveClosure*

```
12   while(!seq.empty()){
13     u = seq.top();
14     seq.pop();
15     for(int v: adj[u]){
16       tcAdj[v]|=tcAdj[u];
17       tcAdj[v][u] = true;
18     }
19   }
20   return tcAdj;
21 }
```

Kode Sumber 4.6b Implementasi fungsi *getTransitiveClosure*

Penjelasan variabel untuk Kode Sumber 4.6 bisa dilihat pada Tabel 4.5.

Tabel 4.5 Penjelasan variabel pada fungsi *getTransitiveClosure*

No	**Variabel**	**Tipe Data**	**Penjelasan**
1	seq	stack<int>	urutan topologis graf dari variabel masukan graph
2	adj	vector<vector<int>>	data graf dalam bentuk *adjacency list*
3	tcAdj	vector<bitset<15005>>	menyimpan data graf dalam bentuk *incidency matrix*

4.3.7 Implementasi Fungsi Pencarian *Minimum Path Cover* *(findMinimumPathCover)*

Implementasi ini (Kode Sumber 4.7) mengacu pada desain di subbab 3.2.6. Fungsi *findMinimumPathCover* menerima masukan berupa objek *class* Graph, dari objek tersebut akan dibentuk *bipartite graph* dengan set *U* yaitu set *vertex destination* dan set *V* yaitu set *vertex source* (bisa dibalik), lalu dari *bipartite graph* tersebut dicari *maximum matching* dan kemudian akan didapatkan jumlah *Minimum Path Cover*. Penjelasan variabel untuk Kode Sumber 4.7 bisa dilihat pada Tabel 4.6.

```
1 int findMinimumPathCover(Graph graph){
2   int N = graph.vertexCount();
3
4   HopCroftKarp bipgraph = HopCroftKarp(N,N);
5   vector<bitset<15005>> tcAdj
6     = getTransitiveClosure(graph);
7
8   for(int i=1;i<=N;++i){
9     for(int j = 1; j <= N; ++j){
10      if(tcAdj[i][j]){
11        bipgraph.addEdge(j,i);
12      }
13    }
14  }
15  int res = N - bipgraph.findMaxMatch();
16
17  return res;
18 }
```

Kode Sumber 4.7 Implementasi fungsi *findMinimumPathCover*

Tabel 4.6 Penjelasan variabel pada fungsi *findMinimumPathCover*

No	Variabel	Tipe Data	Penjelasan
1	N	int	jumlah *vertex* pada graf
2	bipgraph	HopCroftKarp	objek yang menyimpan *bipartite graph* dan memiliki fungsi pencari *maximum cardinality matching*
3	tcAdj	vector<bitset< 15005>>	menyimpan graf dalam bentuk *incidency matrix*
4	res	int	hasil akhir fungsi

4.3.8 Implementasi Fungsi Utama Solusi *Minimum Path Cover* *(main)*

Implementasi ini (Kode Sumber 4.8) mengacu pada desain di subbab 3.2.7. Fungsi ini adalah fungsi yang pertama kali dieksekusi ketika program dijalankan.

```
1 int main(){
2   int N;
3   scanf("%d",&N);
4
5   Graph g(N);
6   for(int i=1;i<N;++i){
7     int r,in;
8     scanf("%d",&r);
```

Kode Sumber 4.8a Implementasi fungsi utama penyelesaian menggunakan *Minimum Path Cover*

```
 9      for(int j=0;j<r;++j){
10        scanf("%d",&in);
11        g.addEdge(i,in);
12      }
13    }
14
15    g = transformEdgeToVertex(g);
16    printf("%d\n",findMinimumPathCover(g));
17    return 0;
18  }
```

Kode Sumber 4.8b Implementasi fungsi utama penyelesaian menggunakan *Minimum Path Cover*

4.4 Implementasi Solusi menggunakan DFS dengan *Greedy Pruning*

Pada subbab ini akan dijelaskan tentang implementasi solusi dengan algoritma *Greedy* dalam bahasa pemrograman C++ berdasarkan desain yang dijelaskan pada Bab III.

4.4.1 Variabel Global

Variabel global digunakan untuk memudahkan pengaksesan data lintas fungsi. Variabel global dapat dilihat pada kode sumber 4.9 dengan penjelasan pada Tabel 4.7.

Tabel 4.7 Penjelasan variabel global

1	adj	deque<int>	representasi graf dalam bentuk *adjacency list*
2	d_in	array of int	menyimpan *degree in* tiap *vertex*

```
deque<int> adj[MAX_V];
int d_in[MAX_V];
```

Kode Sumber 4.9 Variabel global yang diperlukan

4.4.2 Implementasi Fungsi DFS dengan *Greedy Pruning* (dfsWithPruning)

Implementasi ini (Kode Sumber 4.10) mengacu pada desain di subbab 3.3.1. Fungsi melakukan penelusuran menggunakan DFS dan juga menghapus (melakukan *Pruning*) secara *Greedy* untuk *edge-edge* yang tidak akan digunakan diiterasi berikutnya. Iterasi akan diulang oleh fungsi utama (pada subbab 4.4.2) sampai semua *edge* habis.

```
deque<int> adj[MAX_V];
int d_in[MAX_V];

bool dfsWithPruning(int u,bool hasOutBefore){
  if(!adj[u].empty()){
    hasOutBefore = (hasOutBefore && d_in[u] == 1)
                   || adj[u].size() > 1;

    int next = adj[u].front();
    while(d_in[next] == 1 && adj[next].size() == 1){
      next = adj[next].front();
      adj[u].pop_front();
      adj[u].push_front(next);
    }

```

Kode Sumber 4.10a Implementasi fungsi *dfsWithPruning*

```
16      bool hasInAfter = dfsWithPruning(
17                            adj[u].front(),
18                            hasOutBefore
19                          );
20
21      if(hasOutBefore && hasInAfter){
22          d_in[adj[u].front() ]--;
23          adj[u].pop_front();
24      }
25      return d_in[u] > 1 || adj[u].size() == 0;
26    }
27    else{
28        return true;
29    }
30  }
```

Kode Sumber 4.10b Implementasi fungsi *dfsWithPruning*

Penjelasan variabel untuk Kode Sumber 4.10 bisa dilihat pada Tabel 4.8.

Tabel 4.8 Penjelasan variabel pada fungsi *dfsWithPruning*

No	**Variabel**	**Tipe Data**	**Penjelasan**
1	u	int	nomor *vertex*
2	hasOutBefore	bool	bernilai *true* jika ada *vertex v sebelum u* mempunyai *subpath* alternatif
3	hasInAfter	bool	bernilai *true* jika *vertex* setelah *u* bisa diakses dari *subpath* lain

4.4.3 Implementasi Fungsi Utama Solusi Greedy (*main*)

Fungsi utama (Kode Sumber 4.11) adalah fungsi utama yang dieksekusi saat program dijalankan, fungsi ini meliputi proses masukan data, serta iterasi untuk pemanggilan fungsi *dfsWithPruning* sebagai fungsi penelusuran graf. Penjelasan variabel pada Kode Sumber 4.11 bisa dilihat pada Tabel 4.9.

```
int main(){
  int N;
  int r,in;
  scanf("%d",&N);

  memset(d_in,0,sizeof d_in);
  for(int i=1;i<N;++i){
    scanf("%d",&r);
    for(int j=0;j<r;++j){
      scanf("%d",&in);
      adj[i].push_back(in);
      d_in[in]++;
    }
  }

  d_in[1]=1;
  int ans = 0;
  while(!adj[1].empty()){
    ans++;
    dfsWithPruning(1,true);
  }
  printf("%d\n",ans);

  return 0;
}
```

Kode Sumber 4.11 Implementasi fungsi utama penyelesaian menggunakan DFS dengan *Greedy Pruning*

Tabel 4.9 Penjelasan variabel fungsi utama penyelesaian menggunakan DFS dengan *Greedy Pruning*

No	Variabel	Tipe Data	Penjelasan
1	N	int	jumlah *vertex* pada graf
2	ans	int	jumlah *path* minimal untuk melalui semua *edge* pada graf
3	r	int	jumlah masukan *vertex* pada baris ke-i
4	in	int	masukan *vertex* sebagai *vertex destination* dari *vertex source* i

BAB V

UJI COBA DAN EVALUASI

Pada bab ini dijelaskan tentang uji coba dan evaluasi dari implementasi yang telah dilakukan pada tugas akhir ini.

5.1 Lingkungan Uji Coba

Lingkungan yang digunakan untuk uji coba kebenaran ini adalah lingkungan pada situs penilaian daring SPOJ, lingkungan *cluster cube* dengan spesifikasi sebagai berikut.

1. Perangkat Keras
   - *Processor* Intel (R) Pentium G860 CPU @ 3GHz.
   - *Memory* 1536 MB

2. Perangkat Lunak:
   - *Compiler* C++14 (Clang 8.0)

5.2 Uji Coba Kebenaran dan Perbandingan Kinerja

Uji coba kebenaran dilakukan dengan mengirimkan kode sumber program ke situs penilaian daring SPOJ untuk permasalahan SPOJ - Skiver1 (Skiers). Terdapat 2 kode sumber yang dikirim yaitu kode sumber untuk solusi menggunakan *Minimum Path Cover* dan solusi menggunakan DFS dengan *Greedy Pruning*. Pada situs daring tersebut kode akan dieksekusi lalu hasil keluarannya akan dicocokkan dengan hasil yang telah didefinisikan di situs tersebut. Berikut adalah umpan balik dari situs SPOJ.

ID	DATE	USER	RESULT	TIME	MEM	LANG
24464772	2019-09-26 11:10:02	alfian	**accepted** edit ideone it	0.07	6.5M	CPP14-CLANG
24464770	2019-09-26 11:09:58	alfian	**accepted** edit ideone it	0.06	6.5M	CPP14-CLANG
24464766	2019-09-26 11:09:45	alfian	**accepted** edit ideone it	0.07	6.5M	CPP14-CLANG
24464764	2019-09-26 11:09:41	alfian	**accepted** edit ideone it	0.08	6.5M	CPP14-CLANG
24464761	2019-09-26 11:09:35	alfian	**accepted** edit ideone it	0.07	6.5M	CPP14-CLANG
24464758	2019-09-26 11:09:29	alfian	**accepted** edit ideone it	0.07	6.5M	CPP14-CLANG
24464757	2019-09-26 11:09:21	alfian	**accepted** edit ideone it	0.07	6.5M	CPP14-CLANG
24464756	2019-09-26 11:09:14	alfian	**accepted** edit ideone it	0.07	6.6M	CPP14-CLANG
24464755	2019-09-26 11:09:08	alfian	**accepted** edit ideone it	0.07	6.5M	CPP14-CLANG
24464752	2019-09-26 11:08:55	alfian	**accepted** edit ideone it	0.07	6.5M	CPP14-CLANG

Gambar 5.1 Umpan balik sebanyak 10 kali untuk solusi DFS dengan *Greedy Pruning*

ID	DATE	USER	RESULT	TIME	MEM	LANG
24334577	2019-09-02 16:50:35	alfian	**accepted** edit ideone it	0.61	148M	CPP14-CLANG
24334576	2019-09-02 16:50:30	alfian	**accepted** edit ideone it	0.62	148M	CPP14-CLANG
24334574	2019-09-02 16:50:11	alfian	**accepted** edit ideone it	0.61	148M	CPP14-CLANG
24334572	2019-09-02 16:50:01	alfian	**accepted** edit ideone it	0.62	148M	CPP14-CLANG
24334569	2019-09-02 16:49:55	alfian	**accepted** edit ideone it	0.62	148M	CPP14-CLANG
24334567	2019-09-02 16:49:49	alfian	**accepted** edit ideone it	0.62	148M	CPP14-CLANG
24334566	2019-09-02 16:49:44	alfian	**accepted** edit ideone it	0.63	148M	CPP14-CLANG
24334565	2019-09-02 16:49:35	alfian	**accepted** edit ideone it	0.62	148M	CPP14-CLANG
24334563	2019-09-02 16:49:29	alfian	**accepted** edit ideone it	0.61	148M	CPP14-CLANG
24334493	2019-09-02 16:39:15	alfian	**accepted** edit ideone it	0.13	6.7M	CPP14-CLANG

Gambar 5.2 Umpan balik sebanyak 10 kali untuk solusi *Minimum Path Cover*

Tabel 5.1 Perbandingan kinerja dengan hasil umpan balik dengan pengujian 10 kali dari situs SPOJ

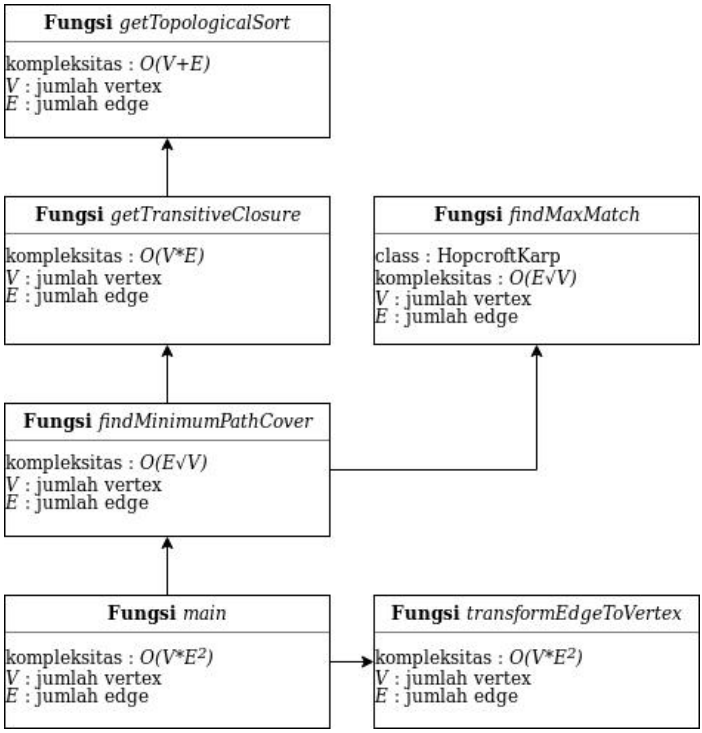
Parameter Perbandingan	**Hasil dengan *Minimum Path Cover***	**Hasil menggunakan DFS dengan *Greedy Pruning***
Waktu Maksimal	0.63 detik	0.08 detik
Waktu Minimal	0.61 detik	0.06 detik
Waktu Rata-Rata	0.62 detik	0.07 detik
Memori Maksimal	148 MB	6.5 MB
Memori Minimal	148 MB	6.5 MB
Memori Rata-Rata	148 MB	6.5 MB
Kompleksitas	$O(V*E^2)$	$O(V+E)$

5.3 Analisis Kompleksitas

Pada subbab ini akan dibahas perbandingan kompleksitas antara solusi menggunakan Algoritma *Minimum Path Cover* dan Algoritma DFS dengan *Greedy Pruning.*

5.3.1 Analisis Kompleksitas Pendekatan Algoritma *Minimum Path Cover*

Berdasarkan desain yang dijelaskan pada subbab 3.2, beberapa fungsi memiliki ketergantungan antar satu dengan yang lain seperti pada Gambar 5.3.



Gambar 5.3 Diagram ketergantungan untuk solusi menggunakan *Minimum Path Cover*

Pada Gambar 5.3, semua pemanggilan fungsi dilakukan sebanyak 1 kali, sehingga total kompleksitas dapat dihitung sebagai berikut.

$$
\begin{aligned}
&Total\ Kompleksitas \\
&= Kompleksitas\ fungsi\ transformEdgeToVertex \\
&+ Kompleksitas\ fungsi\ getTopologicalSort \\
&+ Kompleksitas\ fungsi\ findMaxMatch
\end{aligned}
$$

$$
\begin{aligned}
\text{Total Kompleksitas} &\approx O(V*E^2) + O(V+E) + O(E\sqrt{V}) \\
&\approx O(V*E^2)
\end{aligned}
$$

5.3.2 Analisis Kompleksitas pendekatan Algoritma DFS dengan *Greedy Pruning*

Berdasarkan desain yang dijelaskan pada subbab 3.3, fungsi *dfsWithPruning* berfungsi untuk menelusuri suatu *path,* lalu membuang *edge* yang tidak akan terpakai lagi diiterasi berikutnya. Fungsi *dfsWithPruning* dipanggil didalam perulangan (*for-loop)* pada fungsi utama (Kode Sumber 3.9), sampai tidak ada lagi *edge* yang bisa ditelusuri dari *vertex* 1. Jumlah perulangan adalah jumlah *path* minimal yang dibutuhkan untuk mencakup keseluruhan *edge* pada graf. Algoritma penyelesaian ini memiliki kompleksitas sama dengan DFS yaitu $O(V+E)$ walaupun setiap *edge* mungkin saja dilewati lebih dari sekali, tapi jumlahnya pengulangannya cukup kecil dan bisa diabaikan.

5.4 Analisis dan Kesimpulan Umum

Dibawah ini adalah analisis dan kesimpulan yang didapat dari hasil uji coba yang telah dilakukan.

1. Dengan 10 kali pengujian menggunakan lingkungan situs daring SPOJ didapatkan bahwa solusi menggunakan Algoritma DFS dengan *Greedy Pruning* lebih unggul daripada solusi menggunakan Algoritma *Minimum Path Cover* dalam hal waktu eksekusi (8,86 kali lebih cepat) dan penggunaaan memori (22,77 kali lebih kecil).
2. Algoritma *Minimum Path Cover* yang didesain memiliki kompleksitas $O(V*E^2)$.
3. Algoritma *DFS* dengan *Greedy Pruning* yang didesain memiliki kompleksitas $O(V+E)$.
4. Pengujian yang dilakukan terbatas pada pengujian menggunakan situs daring SPOJ karena keterbatasan penulis dalam membuat data uji berupa graf planar yang acak dalam jumlah besar.

5.5 Pengembangan MPAEC Lebih Lanjut

MPAEC pada DAG bisa dikembangkan dengan menambahkan bobot pada *edge* dan MPAEC yang dicari harus juga menghasilkan total jarak tempuh yang minimal, sehingga lebih efisien secara jumlah petugas dan juga waktu tempuh total yang dialami petugas.

Penyelesaian kasus MPAEC pada Tugas Akhir ini menggunakan algoritma *Minimum Path Cover* yang dimana algoritma ini tidak menggunakan bobot dalam penentuan *path cover*, jika kasus ini dikembangkan lebih lanjut, MPAEC dengan *edge* berbobot berkemungkinan masih bisa diselesaikan dengan mengubah proses *bipartite matching* pada *Minimum Path Cover* atau bahkan *Minimum Path Cover* tidak bisa dikembangkan untuk menyelesaikan MPAEC dengan *edge* berbobot.

Untuk solusi menggunakan DFS dengan *Greedy Pruning* setiap proses penelusuran *edge* selalu diambil berdasarkan urutan *edge,* sedangkan pada permasalahan MPAEC dengan graf berbobot harus mengambil *path* dengan total jarak seminimal mungkin, maka urutan penelusuran pada DFS tidak bisa lagi menggunakan urutan *edge* saja, namun juga harus memperhitungkan juga bobot pada *edge*.

Pengembangan MPAEC dengan menambahkan bobot pada *edge* akan menjadi tantangan baru yang lebih relevan pada penerapannya dikehidupan sehari-hari seperti contohnya kasus petugas ski, dan kasus lain-lainnya yang mungkin berhubungan dengan Tugas Akhir ini.

Halaman ini sengaja dikosongkan

BAB VI

KESIMPULAN

Pada bab ini akan dijelaskan kesimpulan berdasarkan analisis dari hasil uji coba yang telah dilakukan pada bab V.

6.1 Kesimpulan

Dari analisis dan uji coba yang dilakukan terhadap rancangan dan implementasi algoritma untuk menyelesaikan permasalahan klasik SPOJ - Skiver1(Skiers), dapat diambil kesimpulan sebagai berikut.

1. Algoritma *Minimum Path Cover* dapat menyelesaikan permasalahan MPAEC dengan kasus uji soal SPOJ - Skiver1 (Skiers) dengan waktu eksekusi rata-rata selama 0,62 detik dan rata-rata pengunaan memori sebesar 148 MB dengan kompleksitas $O(V*E^2)$.
2. Algoritma *DFS* dengan *Greedy Pruning* dapat menyelesaikan permasalahan MPAEC dengan kasus uji soal SPOJ - Skiver1 (Skiers) dengan waktu eksekusi selama 0,07 detik serta penggunaan memori 6,5 MB dengan kompleksitas $O(V+E)$, algoritma solusi ini lebih cepat dan penggunaan memorinya lebih sedikit dibanding dengan solusi menggunakan algoritma *Minimum Path Cover*.
3. Algoritma *Minimum Path Cover* bisa digunakan untuk menyelesaikan Permasalahan MPAEC dengan memodelkan permasalahannya terlebih dahulu ke model permasalahan *Minimum Path Cover*. Data uji tidak harus memiliki *edge* yang terurut, karena pada prosesnya hal tersebut dapat diabaikan.
4. Algoritma *DFS* dengan *Greedy Pruning* hanya dapat menyelesaikan Permasalahan MPAEC dengan syarat data uji

adalah graf planar dengan *edge* sudah terurut dari kiri ke kanan, seperti pada kasus uji SPOJ - Skiver1 (Skiers).

6.2 Saran

Pada Tugas Akhir ini tentunya terdapat kekurangan serta nilai-nilai yang dapat penulis ambil. Berikut adalah saran-saran yang dapat diambil melalui Tugas Akhir ini.

1. Perbandingan terhadap kedua metode yang telah dibahas di Tugas Akhir ini masih bisa dibuat lebih baik jika mempunyai program *planar graph generator*.
2. Permasalahan pada Tugas Akhir ini masih dikembangkan lagi dengan menambah variabel bobot *edge* pada graf dan hasil yang dicari tidak hanya jumlah *path* yang minimal namun juga harus menghasilkan total jarak yang minimal.

DAFTAR PUSTAKA

[1] Trần Hải Đăng, (2010), Skiver1 (Skiers) [online]. https://www.spoj.com/problems/SKIVER1/.

[2] Mahmoud Radwan, (2014), Flows cuts and matchings [online]. http://mradwan.github.io/algorithms/2014/05/02/flows-cuts-and-matchings/.

[3] Dieter Jungnickel, (2013), Graphs, Networks and Algorithms 4th edition, Heidelberg: Springer.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, dan Clifford Stein, (2000), Introduction to Algorithms Third Edition. USA: MIT Press.

[5] S. Halim dan F. Halim, (2013). Competitive Programming 3. Singapore.

Halaman ini sengaja dikosongkan

BIODATA PENULIS

Alfian, lahir di Tanjung Pinang pada tanggal 04 Nopember 1998, penulis telah menempuh pendidikan di SD 10 Bintan, SMPN 2 Bintan dan SMAN 1 Tanjung Pinang. Penulis melanjutkan studi kuliah program sarjana di Jurusan Informatika ITS.

Selama kuliah di Informatika ITS, penulis mengambil bidang minat Algoritma dan pemrograman (AP). Penulis pernah menjadi asisten dosen dan praktikum untuk mata kuliah dasar pemrograman. Selama menempuh perkuliahan penulis pernah mengikuti berbagai lomba pemrograman tingkat nasional. Penulis juga aktif di kegiatan organisasi seperti *problem setter* di NLC (2017), dan Administrator di Laboratorium Algoritma dan Pemrograman.

Penulis dapat dihubungi melalui surel berikut:
alfian853@gmail.com