TUGAS AKHIR – KI091391

DESAIN DAN ANALISIS ALGORITMA MODIFIKASI HUNGARIAN UNTUK PERMASALAHAN PENUGASAN DINAMIS PADA STUDI KASUS PERMASALAHAN SPOJ KLASIK 12749

TEGUH SURYO SANTOSO
NRP 5110100164

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

UNDERGRADUATE THESES – KI091391

DESIGN AND ANALYSIS OF MODIFIED HUNGARIAN ALGORITHM FOR DYNAMIC ASSIGNMENT PROBLEM: CASE STUDY ON SPOJ CLASSICAL PROBLEM 12749

TEGUH SURYO SANTOSO
NRP 5110100164

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

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

DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION TECHNOLOGY
SEPULUH NOPEMBER INSTITUTE OF TECHNOLOGY
SURABAYA 2014

DESAIN DAN ANALISIS ALGORITMA MODIFIKASI HUNGARIAN UNTUK PERMASALAHAN PENUGASAN DINAMIS PADA STUDI KASUS PERMASALAHAN SPOJ KLASIK 12749

Nama Mahasiswa : Teguh Suryo Santoso
NRP : 5110 100 164
Jurusan : Teknik Informatika FTIf – ITS
Dosen Pembimbing I : Ahmad Saikhu, S.Si., M.T.
Dosen Pembimbing II : Rully Soelaiman, S.Kom., M.Kom.

Abstrak

Masalah penugasan merupakan masalah optimasi kombinatorial untuk menemukan susunan penugasan yang optimal pada sebuah graf bipartite dan dapat diselesaikan dengan algoritma Hungarian. Ketika susunan penugasan yang optimal ditemukan, bisa saja bobot-bobot yang ada pada graf berubah dan susunan penugasan yang optimal juga berubah. Permasalahan untuk menemukan kembali susunan penugasan yang optimal dari graf yang mengalami perubahan bobot ini dinamakan masalah penugasan dinamis.

Dalam buku ini akan dibahas algoritma Hungarian dinamis untuk menyelesaikan masalah penugasan dinamis. Algoritma Hungarian dinamis bekerja dengan memanfaatkan solusi optimal dari masalah penugasan sebelumnya. Dari serangkaian proses peneletian yang telah dilakukan, didapatkan kesimpulan bahwa algoritma Hungarian dinamis dipengaruhi secara linier oleh banyaknya operasi perubahan dan dipengaruhi secara kuadratik oleh jumlah vertex.

Kata kunci: Algoritma hungarian dinamis, graf bipartite, masalah penugasan dinamis, teori graf.

DESIGN AND ANALYSIS OF MODIFIED HUNGARIAN ALGORITHM FOR DYNAMIC ASSIGNMENT PROBLEM: CASE STUDY ON SPOJ CLASSICAL PROBLEM 12749

Student Name	**: Teguh Suryo Santoso**
NRP	**: 5110 100 164**
Major	**: Teknik Informatika FTIf – ITS**
Supervisor I	**: Ahmad Saikhu, S.Si., M.T.**
Supervisor II	**: Rully Soelaiman, S.Kom., M.Kom.**

Abstract

Assignment problem is a combinatorial optimization problem for finding optimal assignment on a bipartite graph and can be solved with Hungarian algorithm. When the optimal assignment has been found, the weight of bipartite graph could be changed and this may cause the change of optimal assignment. Problem for finding back the optimal assignment of a bipartite graph whose weights has been changed is called dynamic assignment problem.

This book will investigate the dynamic Hungarian algorithm for solving dynamic assignment problem. Dynamic Hungarian algorithm solve the problem with exploiting the optimal solution of previous assignment problem. From a series of research which has been done, it can be conclude that dynamic Hungarian algorithm is affected linearly by the number of change operation and is affected quadraticly by the number of vertex.

Keywords: Bipartite graph , dynamic assignment problem, dynamic hungarian algorithm, graph theory.

LEMBAR PENGESAHAN

DESAIN DAN ANALISIS ALGORITMA MODIFIKASI HUNGARIAN UNTUK PERMASALAHAN PENUGASAN DINAMIS PADA STUDI KASUS PERMASALAHAN SPOJ KLASIK 12749

TUGAS AKHIR

Diajukan untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Komputasi Cerdas dan Visualisasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
TEGUH SURYO SANTOSO
NRP. 5110 100 164

Disetujui oleh Pembimbing Tugas Akhir:

1. Ahmad Saikhu, S.Si., M.T. .....
   NIP. 197107182006041001 (Pembimbing I)
2. Rully Soelaiman, S.Kom., M.Kom. .....
   NIP. 197002131994021001 (Pembimbing II)

SURABAYA
JUNI, 2014

KATA PENGANTAR

Segala puji bagi Allah SWT yang telah melimpahkan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan tugas akhir dengan judul "Desain dan Analisis Algoritma Modifikasi *Hungarian untuk* Permasalahan Penugasan Dinamis pada Studi Kasus Permasalahan SPOJ Klasik 12749".

Penulis berharap semoga apa yang tertulis dalam buku tugas akhir ini bermanfaat bagi pengembangan ilmu pengetahuan saat ini dan dapat memberikan kontribusi bagi jurusan Teknik Informatika ITS, ITS dan bangsa Indonesia. Terselesaikannya buku Tugas Akhir ini tidak terlepas dari bantuan serta dukungan dari semua pihak. Oleh karena itu, penulis mengucapkan terima kasih sebesar-besarnya kepada:

1. Ibu, Ayah dan Adik penulis yang telah memberikan dukungan dan setia menemani penulis baik saat suka maupun duka.
2. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku dosen pembimbing penulis yang telah memberikan banyak arahan, bantuan, nasihat, dan perhatian yang banyak sehingga membuat penulis pantang menyerah dalam menyelesaikan Tugas Akhir ini.
3. Bapak Ahmad Saikhu, S.Si., M.T. selaku dosen pembimbing yang telah memberikan nasihat, arahan, dan bantuan sehingga penulis dapat menyelesaikan Tugas Akhir ini.
4. Bapak Ir. Muhammad Husni, M.Kom. selaku dosen wali penulis yang telah memberikan perhatian dan motivasi kepada penulis selama menjadi mahasiswa di lingkungan Teknik Informatika ITS.
5. Ibu Dr. Nanik Suciati selaku ketua jurusan Teknik Informatika ITS beserta Ibu dan Bapak dosen Teknik Informatika ITS yang lainnya. Terima kasih atas segala ilmu yang telah diberikan. Penulis mohon maaf sebesar-besarnya apabila ada kesalahan yang diperbuat oleh penulis baik sengaja maupun tidak disengaja selama masa perkuliahan.

6. Juga tak lupa kepada semua pihak yang belum sempat disebutkan satu per satu di sini yang telah membantu terselesaikannya tugas akhir ini.

Surabaya, Juni 2014

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR KODE SUMBER

DAFTAR TABEL

BAB I
PENDAHULUAN

Pada bab ini akan dijelaskan bagian dasar dalam penyusunan tugas akhir ini. Penjelasan diawali dengan latar belakang, rumusan masalah, batasan masalah, tujuan, dan metodologi pengerjaan tugas akhir. Terakhir, dijelaskan mengenai sistematika laporan tugas akhir.

1.1 Latar Belakang

Masalah penugasan adalah salah satu masalah optimasi kombinatorial mendasar yang merupakan cabang dari ilmu optimasi atau riset operasi. Secara umum, masalah penugasan adalah sebuah permasalahan untuk menemukan susunan pemberian tugas pada pekerja dengan jumlah *rating* atau bobot yang seoptimal mungkin. Dalam hal ini satu pekerja hanya dapat mengerjakan satu tugas dan satu tugas hanya dapat dikerjakan satu pekerja. Beberapa algoritma dapat digunakan untuk menyelesaikan masalah penugasan ini.

Masalah penugasan dapat diterapkan untuk menyelesaikan masalah transportasi atau masalah-masalah lain pada kehidupan nyata. Dalam prakteknya, masalah penugasan dapat bersifat dinamis, yaitu terdapat kemungkinan perubahan bobot pada beberapa penugasan atau penambahan pekerja dan tugas yang baru akibat faktor tertentu. Ketika hal tersebut terjadi, sususan penugasan yang optimal mungkin saja berubah dari susunan penugasan sebelumnya. Salah satu solusi yang dapat dilakukan untuk mendapatkan susunan penugasan yang optimal adalah dengan melakukan kembali proses komputasi dari awal dengan menggunakan algoritma yang telah ditetapkan. Hal tersebut akan menimbulkan masalah jika perubahan bobot sering terjadi. Proses komputasi akan dilakukan berulang kali dari tahap awal dimana untuk algoritma yang ada sampai saat ini, proses tersebut membutuhkan waktu yang tidak sedikit. Maka dari itu, perlu ada algoritma untuk menyeselaikan masalah penugasan dinamis

dalam mendapatkan susunan penugasan yang optimal ketika terjadi perubahan bobot, penambahan pekerja atau penambahan tugas dengan memanfaatkan solusi yang telah ada agar waktu yang dibutuhkan lebih cepat daripada melakukan proses komputasi dari awal.

Beberapa algoritma yang dapat digunakan untuk menyelesaikan masalah ini adalah algoritma *Hungarian* dan algoritma *Hungarian* dinamis. Algoritma *Hungarian* dapat menyelesaikan masalah penugasan dengan kompleksitas waktu $O(n^4)$ dimana n adalah banyaknya *vertex* pada salah satu bagian dari *bipartite* graph. Namun dengan menggunakan struktur data yang tepat, kompleksitas waktu dapat diturunkan menjadi $O(n^3)$. Sedangkan algoritma *Hungarian* dinamis dapat menyelesaikan masalah penugasan dinamis dengan kompleksitas waktu $O(kn^2)$ dimana k adalah ukuran banyaknya perubahan yang terjadi.

Dari penjelasan di atas, hasil yang diharapkan dalam tugas akhir ini adalah proses komputasi untuk menyeselaikan masalah penugasan dinamis dengan konsumsi waktu dan *memory* yang efisien. Tentunya dilakukan analisis yang komprehensif terlebih dahulu dalam menentukan algoritma yang digunakan untuk menyelesaikan masalah ini.

1.2 Rumusan Masalah

Permasalahan yang akan diselesaikan pada tugas akhir ini adalah:

1. Bagaimana menganalisis dan merancang algoritma yang efisien untuk menyelesaikan masalah penugasan dinamis?
2. Bagaimana mengimplementasikan algoritma yang telah dirancang untuk menyelesaikan masalah penugasan dinamis secara efisien?
3. Bagaimana menguji implementasi dari algoritma yang telah dirancang untuk mengetahui kinerja dari implementasi yang telah dibuat?

1.3 Batasan Masalah

Permasalahan yang dibahas dalam tugas akhir ini memiliki batasan-batasan sebagai berikut:

1. Jenis graf yang digunakan adalah *complete bipartite graph*.
2. Jumlah *vertex* pada masing-masing partisi graf *bipartite* tidak lebih dari 100 buah termasuk setelah terjadi penambahan *vertex*.
3. Jumlah *vertex* pada masing-masing partisi graf *bipartite* harus sama.
4. Jumlah perubahan bobot dan penambahan *vertex* baru tidak lebih dari 10000 kali.
5. Bobot dari tiap penugasan lebih dari atau sama dengan 0 dan mampu ditampung dalam tipe data *integer* 32 bit bertanda.
6. Bahasa pemrograman yang digunakan adalah C++.

1.4 Tujuan

Tujuan dari pengerjaan tugas akhir ini antara lain sebagai berikut:

1. Menganalisis dan merancang algoritma yang efisien untuk menyelesaikan masalah penugasan dinamis.
2. Mengimplementasikan algoritma yang telah dirancang untuk menyelesaikan masalah penugasan dinamis secara efisien.
3. Menguji implementasi dari algoritma yang telah dirancang untuk mengetahui kinerja dari implementasi yang telah dibuat.

1.5 Metodologi

Ada beberapa tahap dalam proses pengerjaan tugas akhir ini, yaitu sebagai berikut:

1. Studi Literatur
   Tahap ini merupakan tahap mempelajari dan menganalisa algoritma *Hungarian* klasik yang masih bersifat statis. Kemudian, mempelajari dan menganalisa algoritma

Hungarian yang telah megalami modifikasi sehingga sudah bersifat dinamis terhadap perubahan bobot.

2. Implementasi
   Tahap ini merupakan tahap implementasi algoritma yang telah dipelajari menjadi aplikasi. Bahasa pemrograman yang digunakan adalah C++, dengan bantuan perangkat lunak wxDev-C++ 7.4.2.596 sebagai *Integrated Development Enviroment* (IDE).
3. Uji coba dan evaluasi
   Tahap ini merupakan tahap pengujian aplikasi dengan data masukan yang telah ditentukan untuk menguji kebenaran hasil implementasi algoritma serta menguji waktu eksekusi aplikasi untuk data masukan yang telah ditentukan. Pada tahap ini juga dilakukan optimasi dari hasil implementasi apabila aplikasi masih kurang efisien.
4. Penyusunan buku tugas akhir
   Tahap ini merupakan tahap penyusunan laporan berupa buku tugas akhir sebagai dokumentasi pelaksanaan tugas akhir yang mencakup seluruh teori, implementasi serta hasil pengujian yang telah dikerjakan.

1.6 Sistematika Laporan

Laporan tugas akhir ini dibagi menjadi 6 bab yang masing-masing menjelaskan bagian-bagian yang berbeda, yaitu:

1. Bab I, Pendahuluan, berisi penjelasan mengenai latar belakang, rumusan masalah, batasan masalah, tujuan penulisan tugas akhir, dan sistematika penulisan laporan.
2. Bab II, Dasar Teori, berisi penjelasan teori-teori yang digunakan sebagai dasar pengerjaan tugas akhir ini..
3. Bab III, Metodologi, berisi rangkaian proses dan algoritma yang digunakan sebagai acuan implementasi.
4. Bab IV, Implementasi, berisi implementasi program yang meliputi penjelasan data masukan, data keluaran program, serta penjelasan masing-masing fungsi kode sumber.

5. Bab V, Uji Coba dan Analisis, berisi rangkaian uji coba yang dilakukan untuk menguji kebenaran dan efisiensi program.
6. Bab VI, Penutup, berisi kesimpulan pengerjaan tugas akhir dan saran untuk pengembangan tugas akhir.

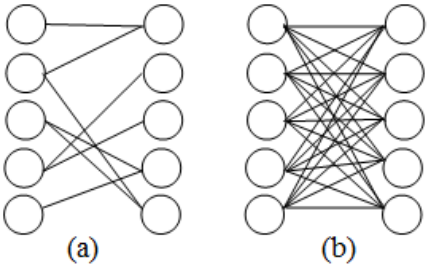
BAB II
DASAR TEORI

2.1 Definisi Umum

Dalam subbab ini akan dibahas definisi-definisi umum yang selanjutnya akan sering digunakan dalam buku ini. Dalam teori graf, banyak sekali istilah yang butuh dijelaskan terlebih dahulu agar pembaca lebih mudah memahami isi dari buku ini yang mana dalam buku ini akan dibahas salah satu permasalahan yang berada dalam *domain* ilmu teori graf.

Sebuah graf $G = (V, E)$ terdiri dari dua himpunan V dan E. Elemen yang berada di dalam himpunan V dinamakan *vertex* atau simpul sedangkan elemen yang berada dalam himpunan E dinamakan *edge* atau sisi. Tiap *edge* terdiri dari satu pasang *vertex* dengan notasi (u, v) yang menandakan terdapat hubungan antara *vertex* u dan *vertex* v dan tiap *vertex* tersebut dinamakan *endpoint*. Sebuah *edge* bisa menghubungkan *vertex* yang sama. *Graf tidak berarah* merupakan graf dimana *edge* (u, v) dan (v, u) dianggap sama. *Graf berarah* merupakan graf dimana *edge* (u, v) dan (v, u) dianggap tidak sama. Dalam graf berarah, *edge* (u, v) menyatakan *vertex* v dapat diakses dari *vertex* u dan *edge* (v, u) menyatakan *vertex* u dapat diakses dari *vertex* v. Jika *vertex* v adalah *endpoint* dari *edge* e maka v dikatakan *bersinggungan* dengan *edge* e. Derajat dari sebuah *vertex* adalah banyaknya *edge* yang bersinggungan dengan *vertex* tersebut. Jika *vertex* u dan *vertex* v adalah *endpoint* dari sebuah *edge* e, maka u adalah *tetangga* v dan v adalah *tetangga* u sehingga dapat dikatakan *vertex* u dan *vertex* v dihubungkan oleh *edge* e. *Graf Bipartite* adalah graf yang *vertex*-nya dapat dibagi menjadi dua himpunan *vertex* S dan T dimana pada tiap himpunan, tidak ada *vertex* yang saling bertetangga. Notasi yang digunakan untuk merepresentasikan graf *bipartite* dalam buku ini adalah $G = (S \cup T, E)$ dimana S dan T adalah himpunan *vertex* S dan T seperti yang telah dijelaskan sebelumnya dan E adalah himpunan *edge*.

Complete bipartite graph adalah graf *bipartite* dimana tiap *vertex* pada satu himpunan bertetangga dengan semua *vertex* pada himpunan yang lainnya. Contoh graf *bipartite* biasa ditunjukkan pada Gambar 2.1(a) dan *complete bipartite graph* ditunjukkan pada Gambar 2.1(b). *Path* adalah deretan *vertex* dimana tiap *vertex* yang bersebelahan adalah *endpoint* dari sebuah *edge* dan *edge* tersebut tidak boleh muncul lebih dari satu kali. *Cycle* adalah *path* yang diawali dan diakhiri oleh *vertex* yang sama.



Gambar 2.1 Contoh graf *bipartite* (a) graf *bipartite* standar (b) *complete bipartite graph*

Subgraf dari graf $G = (V, E)$ adalah graf $H = (W, F)$ dimana $W \subseteq V$ dan $F \subseteq E$. Pada graf $G = (V, E)$, *induced subgraph* dari himpunan *vertex* $W \subseteq V$ adalah subgraf yang himpunan *vertex*-nya adalah W dan semua *edge*-nya memiliki endpoint pada W. Pada graf $G = (V, E)$, subgraf $G \backslash W$ dimana $W \subseteq V$ adalah *induced subgraf* dari himpunan *vertex* $V - W$. *Connected component* adalah subgraf $H = (W, F)$ dimana untuk tiap pasang *vertex* u dan v pada W terdapat setidaknya satu *path* yang berawal di u dan berakhir di v atau berawal di v dan berakhir di u.

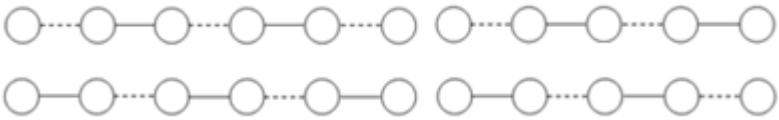
2.2 Matching

Pada subbab ini akan dijelaskan teori tentang *matching*. Terlebih dahulu dibahas pengertian-pengertian yang berkaitan

tentang *matching*. Setelah itu, akan dijelaskan teorema tentang *matching* yang merupakan dasar dari algoritma *Hungarian* dalam menyelesaikan masalah penugasan.

2.2.1 Definisi

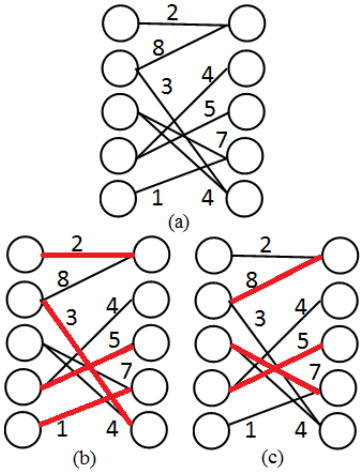
Diasumsikan terdapat Graf $G = (V, E)$ dimana graf tersebut adalah graf tidak berarah dengan himpunan *vertex* V dan himpunan *edge* E serta tiap *edge* $e \in E$ memiliki bobot w_e. Sebuah *matching* pada graf $G = (V, E)$ adalah sebuah himpunan *edge* $M \subseteq E$ dimana tidak ada *edge* yang saling bersentuhan di dalam M. Sebuah *perfect matching* pada graf $G = (V, E)$ adalah sebuah *matching* M dimana tiap *vertex* pada V bersinggungan dengan tepat satu *edge* pada M. Untuk tiap *matching* M pada graf $G = (V, E)$, kumpulan *edge* $e \in M$ adalah *matched edge* sedangkan kumpulan *edge* $e \in E - M$ adalah *unmatched edge*. *Vertex* v adalah *matched vertex* apabila *vertex* tersebut bersinggungan dengan *matched edge*, jika tidak maka *vertex* tersebut adalah *unmatched vertex*. Tiap *matched vertex* v memiliki sebuah *mate* atau *pasangan* yang terletak pada titik akhir dari *matched edge* yang bersinggungan dengan v.



Gambar 2.2 Macam-macam *alternating path*

Alternating path dari sebuah *matching* M adalah *path* dimana urutan *edge*-nya terdiri dari *matched edge* dan *unmatched edge* secara bergantian. Macam-macam *alternating path* dapat dilihat pada Gambar 2.2. *Vertex-vertex* pada urutan 0,2,4 dan seterusnya dinamakan *outer vertex* sedangkan *vertex-vertex* pada urutan 1,3,5 dan seterusnya dinamakan *inner vertex*. *Augmenting path* dari sebuah *matching* M adalah *alternating path* yang

dimulai dari sebuah *unmatched vertex* dan berakhir pada sebuah *unmatched vertex* pula. *Symmetric difference* atau *ring sum* dari dua himpunan *edge* E_1 dan E_2, yang dinyatakan dengan operasi $E_1 \oplus E_2$, adalah *edge-edge* yang berada hanya pada salah satu himpunan E_1 atau E_2 saja.



Gambar 2.3 Perbedaan *Maximum-size matching* dan *maximum-weight matching* (a) graf *bipartite* berbobot (b) *maximum-size matching* (c) *maximum-weight matching*

Kardinalitas dari *matching* M adalah banyaknya *edge* pada M yang dituliskan dengan notasi $|M|$. Bobot dari *matching* M adalah $wt(M) = \sum_{e \in M} w_e$. *Maximum-size matching* pada graf G adalah *matching* M yang memiliki $|M|$ terbesar. *Maximum-weight matching* pada graf G adalah *matching* M yang memiliki nilai $wt(M)$ terbesar. Pada Gambar 2.3, tedapat tiga gambar dimana garis merah menandakan bahwa *edge* tersebut termasuk di dalam *matching*. Gambar 2.3(a) menunjukkan graf *bipartite* berbobot. Gambar 2.3(b) menujukkan *maximum-size matching* dari graf pada Gambar 2.3(a). Hal ini dikarenakan *matching* tersebut tidak dapat dinaikkan kardinalitasnya. Gambar 2.3(c)

menunjukkan *maximum-weighted matching* dari graf pada Gambar 2.3(a). Meskipun *matching* tersebut dapat dinaikkan kardinalitasnya, namun tidak ada *matching* lain yang jumlah bobotnya melebihi 20.

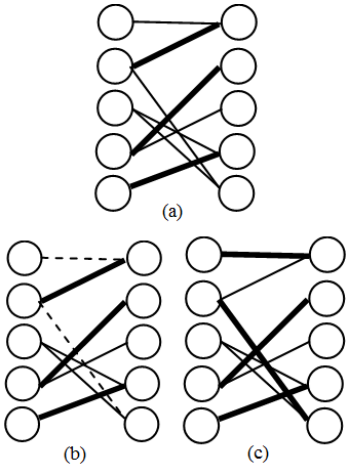
2.2.2 Pembahasan

Dalam subbab ini akan dibahas teori tentang *matching* yang mendasari metode untuk mencari *maximum-size matching* pada graf. Tidak dapat dipungkiri bahwa teori tentang *augmenting path* adalah teori yang sangat mendasari beberapa algoritma untuk *mencari maximum-size matching* dalam sebuah graf. Berikut adalah teorema tentang *augmenting path* beserta pembuktiannya:

Teorema 2.1. (Teorema *Augmenting Path*) Sebuah *Matching* M dalam graf G adalah *maximum-size matching* jika dan hanya jika tidak ada lagi *augmenting path* yang bisa dibentuk sesuai dengan *matching* M [1].

Pembuktian. Misal *matching* M dalam graf G merupakan *maximum-size matching*. Jika ada *augmenting path* P pada graf G yang bisa dibentuk sesuai dengan *matching* M, maka *matching* M dapat digantikan oleh *matching* $M' = M \oplus P$ dimana *matching* M' pasti memiliki kardinalitas $|M| + 1$ dan kontradiksi terjadi. Sebaliknya, jika *matching* M bukan *maximum-size matching*, maka pasti ada *matching* M' yang merupakan *maximum-size matching*. Diasumsikan bahwa subgraf H pada graf G adalah subgraf yang kumpulan *edge*-nya dibentuk oleh $M \oplus M'$. Perlu diperhatikan bahwa tiap *vertex* pada subgraf H memiliki derajat paling banyak 2. *Vertex* dengan derajat 2 pasti bersinggungan dengan satu *edge* pada *matching* M dan satu *edge* pada *matching* M'. Maka, sebuah *connected component* pada subgraf H yang terdiri dari dua *vertex* atau lebih pasti merupakan *cycle* yang memiliki panjang genap (*edge-edge* dari *cycle* adalah *edge* yang terdapat pada *matching* M dan M' secara bergantian) atau *path*

yang terbentuk dari deretan *edge* yang bergantian berada pada *matching M* dan *M'*. Karena *matching M'* memiliki jumlah *edge* yang lebih banyak daripada jumlah *edge* pada *matching M*, maka pasti ada setidaknya satu *path P* pada subgraf *H* yang *edge* pertama dan terakhirnya berada pada *matching M'*. Maka dari itu, *P* adalah *augmenting path* yang dapat dibentuk dari *matching M* karena *vertex* pertama dan terakhirnya tidak berada pada *matching M* dan *path* yang terbentuk adalah deretan *edge* yang bergantian berada pada *matching M* dan *M'*. □



Gambar 2.4 Pembalikan (*flipping*) *augmenting path* (a) *graf bipartite* (b) ditemukan *augmenting path* (c) *augmenting path* yang ditemukan dibalik

Ide dasar dari pencarian *maximum-size matching* menggunakan *augmenting path* yaitu bermula pada sebuah *matching M* (*matching* kosong atau *matching* yang hanya terdiri dari satu *edge*) lalu dari *matching M* dicari *augmenting path* yang sesuai. Lalu dari *augmenting path* yang terbentuk, dapat dibentuk *matching* baru dengan kardinalitas $|M| + 1$ dengan cara membuat *unmatched edge* pada *augmenting path* menjadi *matched edge*

dan begitu juga sebaliknya. Langkah tersebut dijalankan hingga tidak ada lagi *augmenting path* yang dapat dibentuk. Langkah tersebut ditunjukkan seperti pada Gambar 2.4. Pada Gambar 2.4(a) terdapat sebuah *matching* dimana *edge-edge* yang termasuk dalam *matching* ditandai dengan garis tebal hitam. Pada Gambar 2.4(b) ditemukan *augmenting path* yang diawali oleh garis putus-putus dan diakhiri oleh garis putus-putus juga. Pada Gambar 2.4(c) ditunjukkan bahwa dengan ditemukannya *augmenting path*, maka kardinalitas sebuah *matching* dapat dinaikkan menjadi satu lebih banyak dengan membuat *matched edge* menjadi *unmatched edge* dan begitu juga sebaliknya. Langkah ini bisa disebut juga dengan *flipping* atau pembalikan *augmenting path*.

2.3 Bipartite Matching

Dalam subbab ini akan dibahas bentuk khusus dari *matching* yaitu *bipartite matching*. Terlebih dahulu akan dijelaskan beberapa hal yang berkaitan dengan *bipartite matching*. Setelah itu akan dibahas teorema yang mendasari algoritma *Hungarian* dalam menyelesaikan masalah penugasan.

2.3.1 Definisi

Diasumsikan terdapat graf *bipartite* $G = (S \cup T, E)$ dengan bobot tiap *edge* w_e. *Bipartite matching* adalah *matching* yang terdapat pada graf *bipartite*. *Complete matching* dari graf $G = (S \cup T, E)$ adalah *matching* yang memiliki kardinalitas $min\{|S|, |T|\}$. *Regular bipartite graph* adalah graf $G = (S \cup T, E)$ dimana tiap *vertex* pada graf tersebut memiliki derajat yang sama. *k-regular bipartite graph* berarti tiap *vertex* pada graf tersebut memiliki derajat sebanyak k.

2.3.2 Pembahasan

Konsep tentang *bipartite matching* banyak digunakan dalam kehidupan sehari-hari. Sebagai contoh terdapat n pelamar

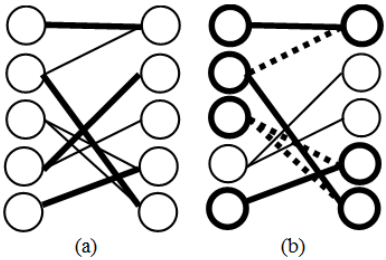
kerja dimana tiap pelamar melamar n pekerjaan. Permasalahan tersebut dapat dimodelkan terlebih dahulu menjadi graf *bipartite* lengkap dengan bobotnya. Permasalahan ini merupakan contoh dari masalah penugasan yang akan dibahas pada subbab selanjutnya.

Salah satu teorema tentang graf *bipartite* yang mendasari penyelesaian masalah penugasan adalah teorema yang diajukan oleh Philip Hall. Teorema tersebut menyatakan ciri-ciri graf *bipartite* yang memiliki *complete matching*. Sebelum membahas teorema tersebut, terlebih dahulu diketahui notasi $N(J)$. Jika terdapat graf *bipartite* $G = (S \cup T, E)$ dan $J \subseteq S$ maka $N(J)$ adalah himpunan *vertex* yang berisi semua tetangga dari *vertex* $i \in J$. Berikut adalah teorema beserta pembuktian teorema *Hall*:

Teorema 2.2. (Teorema *Hall*) Sebuah graf *bipartite* $G = (S \cup T, E)$ dengan $|S| \leq |T|$ mempunyai *complete matching* jika dan hanya jika $|N(J)| \geq |J|$ untuk semua $J \subseteq S$ [2].

Pembuktian. Misal terdapat graf $G = (S \cup T, E)$ yang mempunyai *complete matching* M (semua *vertex* di S ada dalam *matching* M), maka untuk sembarang $J \subseteq S$, *vertex* yang berada di dalam J masing-masing dipasangkan dengan *vertex* pada $N(J)$ yang berbeda. Dari hal tersebut dapat ditarik kesimpulan bahwa $|N(J)| \geq |J|$. Sebaliknya jika G tidak memiliki *complete matching*, maka terdapat *vertex* v pada S yang tidak masuk ke dalam *maximum-size matching* M'. Misal Z adalah himpunan *vertex* yang terdapat pada semua *alternating path* yang dapat dibentuk dimulai dari *vertex* v sesuai dengan *matching* M' (pada *maximum-size matching* tidak ada *augmenting path*, hanya ada *alternating path*). Maka menurut Teorema 2.1, v adalah satu-satunya *vertex* di Z yang tidak berada di dalam *matching* M'. *Vertex* di dalam Z dibagi menjadi dua himpunan, yaitu $A = S \cap Z$ dan $B = T \cap Z$. Karena v adalah satu-satunya *vertex* yang tidak ada pasangannya dan v berada pada A, maka $|A| - 1 = |B|$. Dalam hal ini B adalah $N(A)$ karena semua *vertex* yang berada

pada B terhubung dengan v melalui *alternating path.* Hal tersebut dapat dilihat pada contoh seperti pada Gambar 2.5(a) dan Gambar 2.5(b). Maka dari itu, $|A| - 1 = |N(A)|$ atau $|N(A)| < |A|$ sehingga terjadi kontradiksi. □



Gambar 2.5 Ilustrasi alternating path (a) graf *bipartite* beserta *matching*-nya (b) Semua *alternating path* yang bermula dari *unmatched vertex*

Teorema *Hall* ini juga dikenal sebagai Teorema *Marriage*. Teorema ini sangat berguna baik pada area ilmu teori graf maupun beberapa cabang ilmu matematika lainnya. Teorema ini juga mendasari algoritma *Hungarian* yang akan dibahas pada subbab selanjutnya.

2.4 Algoritma *Hungarian*

Algoritma *Hungarian* adalah algoritma optimasi kombinatorial yang dapat menyelesaikan masalah penugasan dalam waktu polinomial. Banyak permasalahan dalam kehidupan sehari-hari yang dapat dimodelkan menjadi graf lalu solusinya dicari dengan menjalankan algoritma tertentu termasuk juga masalah penugasan. Secara umum *masalah penugasan* atau *assignment problem* dapat dinyatakan dalam beberapa poin sebagai berikut:

- Terdapat n pekerja dan n pekerjaan
- Setiap pekerja akan mengerjakan satu pekerjaan

- Setiap pekerjaan akan dikerjakan oleh satu pekerja
- Terdapat bobot atau *rating* bagi tiap pekerja dalam mengerjakan setiap pekerjaan
- Tujuannya adalah bagaimana mendapatkan n susunan pekerjaan yang optimal.

Nilai optimal dalam hal ini bisa nilai maksimal atau nilai minimal tergantung kebutuhan. Namun dalam buku ini yang diambil adalah nilai maksimal. Masalah tersebut dapat dimodelkan menjadi graf *bipartite* $G = (S \cup T, E)$ terlebih dahulu dimana himpunan *vertex* S dapat diasumsikan sebagai pekerja dan T dapat diasumsikan sebagai pekerjaan. Tiap *edge* (i, j) pada graf G menyatakan bahwa pekerja ke i pada S dapat melakukan pekerjaan j pada T lengkap beserta bobot W_{ij} yang menyatakan bobot pekerja i dalam mengerjakan pekerjaan j.

Algoritma *Hungarian* dikembangkan oleh Harold Kuhn pada tahun 1955 dan memberi nama algoritma ini sebagai algoritma *Hungarian* karena algoritma ini sebagian besar didasarkan pada karya sebelumnya dari dua ahli matematika asal Hungaria: Dénes Konig dan Jeno Egerváry. Selanjutnya akan dibahas teori yang mendasari dan cara kerja dari algoritma *Hungarian*. Namun terlebih dahulu disepakati bahwa masalah yang akan diselesaikan oleh algoritma *Hungarian* yang akan dibahas dalam buku ini adalah menemukan *maximum-weight matching* pada *complete bipartite graph* $G = (S \cup T, E)$ dimana $|S| = |T|$ sesuai dengan definisi masalah penugasan sebelumnya. Karena tiap subset $J \subseteq S$ pada *complete bipartite graph* memenuhi $|N(J)| \geq |J|$, maka kondisi pada Teorema 2.2 terpenuhi sehingga tiap *vertex* pada S mendapatkan pasangan. Karena $|S| = |T|$, maka tiap *vertex* pada T juga mendapatkan pasangan sehingga *matching* yang dihasilkan dalam hal ini adalah *perfect matching*. Untuk selanjutnya digunakan istilah *optimal matching* untuk menyebutkan *matching* yang dimaksud di atas.

Misal terdapat *complete bipartite graph* $G = (S \cup T, E)$ dimana $S = \{1 \ldots n\}$ dan $T = \{1' \ldots n'\}$ dan fungsi bobot $W =$

(w_{ij}) dimana w_{ij} merupakan bobot dari *edge* (i, j') dan diasumsikan bahwa w_{ij} selalu non-negatif, maka algoritma *Hungarian* dapat menemukan *optimal matching* pada G dalam kompleksitas waktu $O(n^3)$. Dalam hal ini $n = |S| = |T|$. Dua *vector* $< u, v >$ dimana $u = (u_1, u_2, u_3 \ldots u_n)$ dan $v = (v_1, v_2, v_3 \ldots v_n)$ adalah *feasible node-weighting* jika memenuhi Persamaan 2.1 berikut:

$$\boldsymbol{u_i + v_j \geq w_{ij}} \text{untuk semua } \boldsymbol{i, j = 1 \ldots n} \tag{2.1}$$

Misal *optimal matching* pada G memiliki total bobot D, maka dapat diketahui dengan jelas bahwa

$$\mathrm{w(M)} \leq \mathrm{D} \leq \sum_{i=1}^{n} u_i + v_i \tag{2.2}$$

Pada Persamaan 2.2, $w(M)$ adalah jumlah bobot dari sembarang *perfect matching* M pada G. Jika dapat ditemukan *perfect matching* M dimana kesetaraan terjadi pada Persamaan 2.2, maka M adalah *optimal matching* dan kondisi tersebut pasti dapat ditemukan. Berikutnya akan disediakan pembuktian dari pernyataan tersebut. Sebelum membuktikan pernyataan tersebut, untuk sembarang *feasible node-weighting* $< u, v >$, $H_{u,v}$ adalah subgraf dari G dengan himpunan *vertex* V dimana tiap *vertex*-nya adalah *endpoint* dari *edge* (i, j') yang memenuhi $u_i + v_j = w_{ij}$. Dalam hal ini $H_{u,v}$ dinamakan *equality subgraph* dari $< u, v >$. Berikut adaah pembuktian yang telah dijanjikan sebelumnya:

Lemma 2.3. Misal $H = H_{u,v}$ adalah *equality subgraph* untuk sembarang *feasible node-weighting* $< u, v >$, maka kondisi pada Persamaan 2.3 berikut:

$$\boldsymbol{\sum_{i=1}^{n} (u_i + v_i) = D} \tag{2.3}$$

terjadi jika dan hanya jika H memiliki *perfect matching*. Dalam hal ini setiap *perfect matching* dari H adalah *optimal matching* dari G.

Pembuktian. Misal $\sum_{i=1}^{n}(u_i + v_i) = D$ dan H tidak mempunyai *perfect matching*. Untuk $J \subseteq S$ maka $N(J)$ adalah himpunan *vertex* yang merupakan tetangga dari semua *vertex* $i \in J$. Menurut Teorema 2.2, terdapat J dimana $|N(J)| < |J|$. Misal δ diberi nilai seperti pada Persamaan 2.4 berikut:

$$\boldsymbol{\delta = min\{u_i + v_j - w_{ij}: i \in J, j' \notin N(J)\}} \tag{2.4}$$

dan didefinisikan $< u', v' >$ seperti pada Persamaan 2.5 berikut:

$$u'_i = \begin{cases} u_i - \delta\,, & i \in J \\ u_i \quad\;\; , & i \notin \mathrm{J} \end{cases} \text{ dan } v'_j = \begin{cases} v_j + \delta\,, & j' \in N(J) \\ v_j \quad\;\; , & j' \notin \mathrm{N(J)} \end{cases} \tag{2.5}$$

Maka $< u', v' >$ juga merupakan *feasible node-weighting*. Hal tersebut dikarenakan kondisi $u'_i + v'_j \geq w_{ij}$ hanya mungkin disebabkan saat $i \in J$ dan $j' \notin \mathrm{N(J)}$, namun karena $\delta \leq u_i + v_j - w_{ij}$ maka $w_{ij} \leq (u_i - \delta) + v_j = u'_i + v'_j$. Dari sini kemudian terjadi kontradiksi seperti yang ditunjukkan pada Persamaan 2.6 berikut:

$$D \leq \sum_{\mathrm{i=1}}^{\mathrm{n}} u'_i + v'_i = \sum_{\mathrm{i=1}}^{\mathrm{n}} u_i + v_i - \delta|\mathrm{J}| + \delta|N(J)| < D \tag{2.6}$$

Misal sekarang H memiliki *perfect matching* M. Kesetaraan dalam Persamaan 2.2 terjadi. Hal ini dikarenakan kesetaraan terjadi pada Persamaan 2.1 untuk tiap *edge* di M. Lalu menjumlahkan semua Persamaan 2.1 untuk semua *edge* pada M akan menghasilkan kesetaraan pada Persamaan 2.2. □

Ide dasar dari algoritma *Hungarian* adalah pada mulanya ditetapkan *feasible node-weighting* $< u, v >$ diberi nilai seperti pada Persamaan 2.7 berikut:

$$u_i = \max \{w_{ij} , j = 1,2,3 \dots n\} \text{ dan } v_1 = \cdots = v_n = 0 \qquad (2.7)$$

Jika *equality subgraph* $H_{u,v}$ memiliki *perfect matching* maka *optimal matching* dari graf G sudah didapatkan. Jika tidak, maka dicari $J \subseteq S$ dimana $|N(J)| < |J|$ dan mengubah $< u, v >$ sesuai dengan pembuktian Lemma 2.3. Hal itu akan mengurangi $\sum_{i=1}^{n}(u_i + v_i)$ dan menambah setidaknya satu *edge* (i, j') dengan $i \in J$ dan $j' \notin$ N(J) ke *equality subgraph* baru $H_{u',v'}$. Proses ini diulangi hingga jenis *matching* pada *equality subgraph* yang baru bukan lagi *maximum-size matching*. Pada akhirnya akan didapatkan graf H yang berisi *perfect matching* yang merupakan *optimal matching* dari graf G. Untuk mendapatkan jumlah bobot dari *optimal matching* yang telah didapatkan, dapat diambil dari nilai $\sum_{i=1}^{n}(u_i + v_i)$. Hal tersebut dikarenakan semua *vertex* baik pada S dan T masuk ke dalam *optimal matching* yang didapatkan dan menjumlahkan u_i dan v_i sama dengan menghitung bobot *optimal matching* yang didapatkan.

2.5 Algoritma *Hungarian* Dinamis

Dalam kehidupan sehari-hari, ketika *optimal matching* sebuah graf *bipartite* telah didapatkan, permasalahan tidak berhenti pada saat itu juga. Permasalahan baru muncul saat terjadi perubahan beberapa bobot penugasan yang berarti terjadi pula perubahan bobot pada beberapa *edge* pada graf *bipartite* hasil pemodelan masalah penugasan. Salah satu solusi yang paling mungkin dilakukan yaitu dengan menjalankan ulang algoritma *Hungarian* pada graf yang baru. Hal tersebut tidak menjadi masalah jika jarang terjadi perubahan. Namun pada sistem yang memungkinkan dimana sering terjadi perubahan apalagi dalam rentang waktu yang relatif sebentar, proses penghitungan harus

dilakukan secepat mungkin. Permasalahan inilah yang disebut dengan *masalah penugasan dinamis* atau *dynamic assignment problem*.

Algoritma *Hungarian* dinamis dapat digunakan untuk menyelesaikan masalah penugasan dinamis. Algoritma *Hungarian* dinamis dapat menemukan *optimal matching* pada graf *bipartite* yang mengalami perubahan dengan memanfaatkan hasil perhitungan dari algoritma *Hungarian* (baik algoritma *Hungarian* klasik maupun algoritma *Hungarian* dinamis) sebelumnya. Ide dasar dari algoritma *Hungarian* dinamis adalah mempertahankan nilai *feasible node-weighting* $< u, v >$ hasil algoritma *Hungarian* sebelumnya ketika terjadi perubahan bobot pada graf *bipartite*. Penyesuaian nilai *feasible node-weighting* terhadap perubahan bobot pada graf *bipartite* perlu dilakukan agar selanjutnya algoritma *Hungarian* dapat dijalankan tanpa harus mencari *feasible node-weighting* dari awal. Bisa dikatakan algoritma ini adalah *modifikasi* dari algoritma *Hungarian*. Hal ini dikarenakan cara kerja dari algoritma *Hungarian* dinamis adalah mengubah beberapa hasil dari algoritma *Hungarian* yang telah dijalankan sesuai dengan perubahan bobot yang terjadi. Algoritma *Hungarian* dinamis yang dibahas pada [3] berfungsi untuk mendapatkan *minimum-weight matching*, namun dalam buku ini akan dibahas algoritma *Hungarian* dinamis yang berfungsi untuk mendapatkan *maximum-weight matching*. Macam-macam per-ubahan bobot yang akan dibahas pada buku ini ada empat. Didefinisikan terlebih dahulu *complete bipartite graph* $G = (S \cup T, E)$. Dalam hal ini $n = |S| = |T|$ dimana $S = \{1 \ldots n\}$ dan $T = \{1' \ldots n'\}$ dan fungsi bobot $W = (w_{ij})$ dimana w_{ij} merupakan bobot dari *edge* (i, j') dimana $i \in S$ dan $j' \in T$ dan diasumsikan bahwa w_{ij} selalu non-negatif. Terdapat juga *feasible node-weighting* $< u, v >$ lalu dijalankan algoritma *Hungarian* yang mengasilkan *optimal matching* M dan didapatkan nilai $< u, v >$ yang baru. Empat perubahan yang dimaksud adalah:

1) perubahan nilai bobot pada *edge* (i, j') untuk semua $j' \in T$

2) perubahan nilai bobot pada *edge* (i, j') untuk semua $i \in S$
3) perubahan nilai bobot pada sebuah *edge* (i, j')
4) penambahan dua *vertex* masing-masing pada S dan T

Pada kasus keempat, ditambahkan juga beberapa *edge* sehingga graf G tetap *complete bipartite graph* dan dalam hal ini diasumsikan semua *edge* yang berhubungan dengan *vertex* yang baru saja ditambahkan diberi nilai 0. Berikut adalah penjelasan dari penyesuaian untuk tiap kasus yang telah disebutkan di atas.

2.5.1 Perubahan Nilai Bobot pada *Edge* (i, j') untuk Semua $j' \in T$

Dalam kasus ini terdapat sebuah *vertex* i ∈ S dimana semua *edge* yang bersinggungan dengannya diubah nilai bobotnya. Dari $< u, v >$ hasil algoritma *Hungarian* sebelumnya, penyesuaian yang harus dilakukan adalah mengubah nilai u_i. Nilai u_i diubah menjadi seperti Persamaan 2.8 berikut:

$$u_i = max(w_{ij} - v_j) \text{ untuk } j = 1 \ldots n \tag{2.8}$$

Hal tersebut dilakukan untuk mencari nilai u_i sehingga $u_i + v_j \geq w_{ij}$ untuk semua j. Hal ini dikarenakan $u_i \geq w_{ij} - v_j$ untuk semua j sesuai dengan yang didapat akibat dari Persamaan 2.8. Setelah itu langkah selanjutnya adalah menghapus *edge* (i, j') dari *optimal matching M* karena pasti ada kemungkinan susunan *optimal matching* berubah.

2.5.2 Perubahan Nilai Bobot pada *Edge* (i, j') untuk Semua $i \in S$

Dalam kasus ini terdapat sebuah *vertex* j′ ∈ T dimana semua *edge* yang bersinggungan dengannya diubah nilai bobotnya. Dari $< u, v >$ hasil algoritma *Hungarian* sebelumnya,

penyesuaian yang harus dilakukan adalah mengubah nilai v_j. Nilai v_j diubah menjadi seperti Persamaan 2.9 berikut:

$$v_j = max(w_{ij} - u_i) \text{ untuk } i = 1 \ldots n \quad (2.9)$$

Hal tersebut dilakukan untuk mencari nilai v_j sehingga $u_i + v_j \geq w_{ij}$ untuk semua i. Hal ini dikarenakan $v_j \geq w_{ij} - u_i$ untuk semua i sesuai dengan yang didapat akibat dari Persamaan 2.9. Setelah itu langkah selanjutnya adalah menghapus *edge* (i, j') dari *optimal matching M* karena pasti ada kemungkinan susunan *optimal matching* berubah.

2.5.3 Perubahan Nilai Bobot pada Sebuah *Edge* (i, j')

Dalam kasus ini terdapat *edge* (i, j') i ∈ S dan j′ ∈ T dengan yang bobotnya diubah dari w_{ij} menjadi w'_{ij}. Terdapat beberapa kondisi yang harus diperhatikan yang akan dijelaskan sebagai berikut:

- Jika $w'_{ij} < w_{ij}$
  Pada kasus ini, terdapat dua kondisi yang harus diperhatikan. Dua kondisi tersebut adalah sebagai berikut:
  - Jika pasangan i pada *optimal matching M* adalah *j'*
    Dalam kondisi ini terjadi perubahan bobot pada *edge* yang termasuk dalam *optimal matching M* menjadi lebih kecil. Maka dari itu *edge* (i, j') perlu dihapus dari *optimal matching M*. Nilai u_i dan v_j tetap karena bobot yang baru lebih kecil sehingga ketetapan $u_i + v_j \geq w_{ij}$ tetap terjaga.
  - Jika pasangan i pada *optimal matching M* bukan *j'*
    Dalam kondisi ini terjadi perubahan bobot pada *edge* yang tidak termasuk dalam *optimal matching M* menjadi lebih kecil daripada bobot sebelumnya sehingga tidak ada yang perlu dilakukan.

- Jika $w'_{ij} > w_{ij}$
  Pada kasus ini, terdapat dua kondisi yang harus diperhatikan. Dua kondisi tersebut adalah sebagai berikut:
  - Jika $u_i + v_j \geq w'_{ij}$
    Jika $u_i + v_j \geq w_{ij}$, maka kondisi *feasible node-weighting* tetap terjaga. Dalam hal ini *optimal matching* M tetap *perfect matching* dari *equality subgraph* dari $< u, v >$ sehingga tidak ada yang perlu dilakukan.
  - Jika $u_i + v_j < w'_{ij}$
    Dalam hal ini salah satu dari nilai u_i atau v_j perlu disesuaikan agar kondisi *feasible node-weighting* tetap terjaga. Nilai u_i atau v_j bisa disesuaikan salah satunya saja. Jika dipilih hanya nilai u_i yang disesuaikan maka nilai u_i akan seperti pada Persamaan 2.8. Jika dipilih hanya nilai v_j yang disesuaikan maka nilai v_j akan seperti pada Persamaan 2.9. Setelah itu, jika *vertex j'* bukan pasangan i pada *optimal matching* M, maka terdapat kemungkinan bahwa *optimal matching* berubah. Maka dari itu penghapusan *edge* (i, j') dari *optimal matching M* perlu dilakukan.

2.5.4 Penambahan *Vertex* Masing-masing pada *S* dan *T*

Dalam kasus ini, penambahan dua *vertex* dilakukan terhadap graf G. Satu *vertex* ditambahkan pada S dan satu *vertex* pada T. Misal *vertex* yang ditambahan pada *S* adalah *vertex* p dan *vertex* yang ditambahkan pada T adalah *vertex* q, maka ditambahkan juga beberapa *edge* yang menghubungkan p dengan semua *vertex* pada T dan beberapa *edge* yang menghubungkan q dengan semua *vertex* pada S serta satu *edge* yang menghubungkan p dengan q. Dalam hal ini bobot dari semua *edge* yang baru saja ditambahkan diberi nilai 0. Selain itu ditambahkan juga u_p dan v_q pada $< u, v >$ dengan nilai u_p sama dengan Persamaan 2.10 berikut:

$$u_p = \max(w_{pq}, max(w_{pj} - v_j) \text{ untuk } j = 1 \ldots n) \quad (2.10)$$

dan v_q sama dengan Persamaan 2.11 berikut:

$$v_q = max(w_{iq} - u_i) \text{ untuk } i = 1 \ldots n, p \quad (2.11)$$

Pemberian nilai u_p sebagaimana yang telah dituliskan pada Persamaan 2.10 adalah untuk menyesuakan nilai *feasible node-weighting*. Perlu diingat nilai u_p bisa lebih dari 0 ketika $v_j < 0$ setelah diberlakukannya Persamaan 2.9. Begitu juga dengan nilai v_q yang harus disesuaikan dengan nilai u_i untuk semua i karena terdapat kemungkinan nilai u_i bernilai negatif. Hal ini dapat terjadi saat dilakukan operasi perubahan nilai $< u, v >$ menjadi $< u', v' >$ sesuai dengan pembuktian Lemma 2.3 atau setelah diberlakukannya Persamaan 2.8. Setelah itu langkah selanjutnya adalah menjadikan p dan q sebagai *unmatched vertex*.

Setelah proses penyesuaian dilakukan sesuai dengan jenis perubahan yang terjadi, maka perlu dijalankan algoritma *Hungarian* untuk memperoleh *optimal matching* yang baru. Perlu dicatat bahwa dalam hal ini algoritma *Hungarian* tidak lagi dijalankan dari awal. Dari kasus-kasus di atas, hal paling buruk yang terjadi adalah apabila salah satu *edge* pada *optimal matching* M dihapus. Proses menemukan *optimal matching* hanya dengan satu pasang *vertex* saja yang belum masuk ke dalam *optimal matching* pasti sangat lebih cepat daripada proses menemukan *optimal matching* dari keadaan awal. Terlebih lagi *feasible node-weighting* telah disesuaikan agar nilainya tetap terjaga sesuai dengan *matching M* yang saat ini terbentuk.

2.6 Metode Hungarian

Metode *Hungarian* adalah metode yang juga digunakan untuk menyelesaikan masalah penugasan [4]. Namun pendekatan

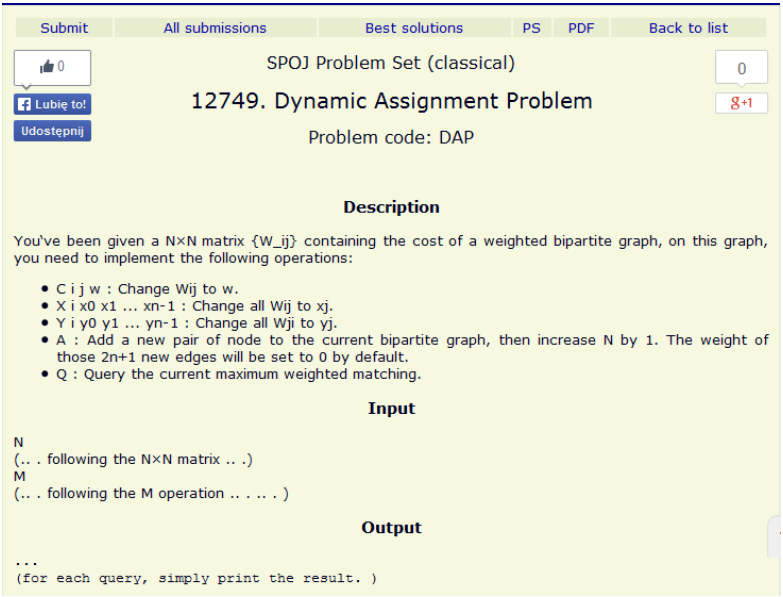
dari metode ini sama sekali tidak ada hubungannya dengan teori graf. Metode ini diperlukan untuk proses uji kebenaran dari algoritma *Hungarian* dinamis yang telah dirancang. Metode ini dipilih karena metode ini sering dipakai dalam menyelesaikan masalah penugasan. Sebagai ganti dari graf, metode ini hanya menggunakan matriks berukuran $n \times n$ dimana n adalah jumlah pekerja atau pekerjaan dalam mencari solusi optimal masalah penugasan. Matriks tersebut berisi bobot dari para pekerja dalam melakukan pekerjaan. Selanjutnya, matriks tersebut dinamakan matriks bobot. Berikut adalah langkah-langkah dari metode *Hungarian*:

1) Tentukan nilai minimal pada tiap baris matriks bobot lalu kurangkan dengan semua nilai pada baris tersebut.
2) Dari matriks bobot hasil dari langkah pertama, Tentukan nilai minimal pada tiap kolom matriks bobot lalu kurangkan dengan semua nilai pada kolom tersebut.
3) Perhatikan nilai 0 pada tiap elemen matriks bobot hasil dari langkah kedua. Jika bisa dibentuk penugasan dari nilai 0 pada tiap elemen matriks maka susunan penugasan yang optimal ditemukan. Penugasan yang dimaksud adalah dalam tiap baris dan kolom pada matriks hanya ada satu elemen yang terpilih. Jika tidak lakukan langkah keempat.
4) Tarik beberapa garis horizontal yang melewati satu baris pada matriks bobot atau garis vertikal yang melewati kolom baris pada matriks bobot seminimal mungkin yang dapat melewati semua nilai 0 pada matriks bobot.
5) Tentukan nilai terkecil dari elemen matriks yang tidak dilewati garis. Kurangkan pada tiap elemen matriks yang tidak dilewati garis dan tambahkan pada elemen matriks yang dilewati oleh dua garis. Pergi ke langkah ketiga.

Susunan penugasan yang dihasilkan dari metode ini adalah susunan penugasan yang minimal. Karena dalam buku ini dicari susunan penugasan yang maksimal, maka nilai tiap elemen pada matriks bobot diubah terlebih dahulu menjadi $maxw - w_{ij}$

dimana $maxw$ merupakan nilai terbesar pada matriks bobot dan w_{ij} adalah elemen matriks baris ke-i kolom ke-j. Hal ini dilakukan karena semakin besar nilai sebuah elemen pada matriks, setelah dilakukan perubahan seperti di atas, nilai elemen matriks tersebut akan menjadi semakin kecil dan begitu juga sebaliknya sehingga susunan minimal yang didapatkan sebenarnya adalah susunan maksimal pada matriks bobot yang belum diubah. Proses ini selanjutnya akan disebut sebagai transformasi matriks minimum.

2.7 Soal *Dynamic Assignment Problem* pada Situs Penilaian Daring SPOJ



Gambar 2.6 Deskripsi soal SPOJ klasik 12749

Situs penilaian daring adalah sebuah situs yang berisi berbagai jenis materi soal pemrograman yang dapat diselesaikan dengan cara pengguna mengunggah kode program yang berisi

penyelesaian terhadap masalah tersebut. Tiap-tiap soal memiliki batasan waktu eksekusi dan memori program yang berbeda. Format data masukan yang dibaca dan data keluaran yang dihasilkan oleh program harus sesuai dengan ketentuan pada soal.

Pada situs penilaian daring SPOJ ini terdapat soal yang mendasari penelitian ini. Soal tersebut bernama *Dynamic Assignment Problem* yang memiliki nomor soal 12749 dan kode problem DAP pada SPOJ [5]. Deskripsi mengenai soal dapat dilihat pada Gambar 2.6. Pada soal tersebut diberikan bilangan bulat n yang merupakan ukuran matriks $n \times n$. Setelah itu diberikan bobot-bobot dari matriks $n \times n$ tersebut. Kemudian diberikan bilangan bulat m yang merupakan banyaknya operasi. Setelah itu diberikan m operasi.

Ketentuan-ketentuan dari soal SPOJ klasik 12749 adalah sebagai berikut:

- Diberikan matriks $n \times n$ dimana matriks tersebut merupakan bobot dari *complete bipartite graph*.
- Terdapat beberapa operasi yang harus diimplementasikan sebagai berikut:
  - $X\ i\ x_0\ x_1\ x_2 \ldots x_{N-1}$
    Semua bobot pada baris ke-i pada matriks diganti dengan masukan yang diberikan yaitu diganti dengan x_0 hingga x_{N-1}. Dapat dikatakan bahwa operasi ini merepresentasikan perubahan tipe *update row*.
  - $Y\ i\ y_0\ y_1\ y_2 \ldots y_{N-1}$
    Semua bobot pada kolom ke-i pada matriks diganti dengan masukan yang diberikan yaitu diganti dengan y_0 hingga y_{N-1}. Dapat dikatakan bahwa operasi ini merepresentasikan perubahan tipe *update column*.
  - $C\ i\ j\ w$
    Nilai pada elemen matriks baris ke-i kolom ke-j diganti bobotnya menjadi w. Dapat dikatakan bahwa operasi ini merepresentasikan perubahan tipe *update cell*.

- A
  Ukuran matriks bobot ditambah 1 menjadi $N + 1$. Bobot pada baris ke-$(N + 1)$ dan kolom ke-$(N + 1)$ pada matriks bobot semuanya diatur menjadi 0. Dapat dikatakan bahwa operasi ini merepresentasikan perubahan tipe *add vertex*.
- Q
  Jika sebuah baris pada masukan hanya terdiri dari huruf Q saja berarti hal ini menandakan bahwa program diminta untuk mengeluarkan jumlah bobot *optimal matching* dari graf yang sekarang. Dapat dikatakan bahwa operasi ini adalah operasi *query*.

- Jumlah vertex tidak akan pernah melebihi 100.
- Jumlah operasi tidak akan pernah melebihi 10000 dimana jumlah operasi *query* tidak akan pernah melebihi 1000.
- Nilai bobot dapat ditampung dalam tipe data *integer* 32 bit bertanda.
- Keluaran yang diharapkan adalah bobot *optimal matching* setiap ada operasi *query*.
- Program yang dibuat tidak boleh berjalan dalam waktu lebih dari 10 detik, tidak boleh menghabiskan memory lebih dari 1536 MB dan ukuran kode program yang dibuat tidak boleh melebihi 50000 B.
- Program akan dijalankan pada lingkungan komputer dengan prosesor Intel Pentium G860 3 GHz.

Contoh masukan dan keluaran dari soal dapat dilihat pada Gambar 2.7. Baris pertama pada masukan bernilai 2 yang menunjukkan akan diberikan matriks berukuran 2×2. Dua baris berikutnya merupakan nilai bobot dari matriks. Sehingga didapatkan matriks saat ini adalah $\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$. Baris keempat merupakan banyaknya operasi yang akan dilakukan dimana dalam hal ini nilainya tiga. Baris kelima terdapat karakter Q yang menandakan operasi *query*. Dalam hal ini program diperintahkan

untuk mengeluarkan bobot *optimal matching* dari matriks saat ini. Pada keluaran program baris pertama, dapat dilihat program mengeluarkan angka 2 yang menandakan bahwa bobot *optimal matching* saat ini adalah 2. Kemudian pada baris keenam masukan terdapat operasi perubahan satu elemen pada matriks. Hal ini dapat diketahui dari karakter pertama yang ada pada baris keenam yaitu C. Setelah karakter C terdapat tiga bilangan bulat. Dua bilangan bulat pertama merupakan indeks elemen yang diubah dimana dalam hal ini elemen yang diubah adalah elemen yang terletak pada baris ke-0 dan kolom ke-1. Bilangan ketiga merupakan bobot yang baru pada elemen yang diubah nilainya dimana bobot yang baru nilainya 9 sehingga matriks saat ini adalah $\begin{vmatrix} 1 & 9 \\ 0 & 1 \end{vmatrix}$. Baris ketujuh terdapat karakter Q lagi yang menandakan operasi *query*. Setelah itu program mengeluarkan angka 9 yang menyatakan bobot *optimal matching* pada saat ini.

Example

```
Input 1:
2
1 0
0 1
3
Q
C 0 1 9
Q

Output 1:
2
9
```

Gambar 2.7 Contoh masukan dan keluaran soal

BAB III
METODOLOGI

Dalam bab ini akan dijelaskan metodologi yang dipakai dalam menyelesaikan masalah penugasan dinamis. Menurut penjelasan pada subbab 2.5, masalah penugasan dinamis dapat diselesaikan menggunakan algoritma *Hungarian* dinamis. Karena ide dasar dari algoritma *Hungarian* dinamis adalah mempertahankan *feasible node-weighting* saat ada perubahan bobot lalu menjalankan kembali algoritma *Hungarian*, maka pada bab ini akan dijelaskan perancangan algoritma *Hungarian* terlebih dahulu lalu dijelaskan perancangan algoritma *Hungarian* dinamis yang digunakan untuk menyelesaikan permasalahan penugasan dinamis.

3.1 Perancangan Algoritma *Hungarian*

Menurut subbab 2.4, ide dasar dari algoritma *Hungarian* dalam menemukan *optimal matching* suatu graf *bipartite* berbobot adalah dengan menemukan *feasible node-weighting* $< u, v >$ dimana *equality subgraph* dari $< u, v >$ memiliki *perfect matching*. Selanjutnya, *perfect matching* tersebut merupakan *optimal matching* yang dicari. Didefinisikan terlebih dahulu *complete bipartite graph* $G = (S \cup T, E)$ dimana dalam hal ini $n = |S| = |T|$ dengan $S = \{1 \ldots n\}$ dan $T = \{1 \ldots n'\}$. Terdapat fungsi bobot $W = (w_{ij})$ dimana w_{ij} merupakan bobot dari *edge* (i, j') untuk $i \in S$ dan $j' \in T$ dan diasumsikan bahwa w_{ij} selalu non-negatif. Terdapat juga *feasible node-weighting* $< u, v >$. Pertama-tama, dibuat fungsi untuk menginisialisasi $< u, v >$ dan menetapkan bahwa semua *vertex* pada G adalah *unmatched vertex*. *Pseudocode* dari proses inisialisasi tersebut ada pada Gambar 3.1. Seperti yang bisa dilihat pada Gambar 3.1, inisialisasi $< u, v >$ dilakukan seperti apa yang telah dijelaskan pada subbab 2.4. Terdapat dua larik `mateS` dan `mateT` dimana masing-masing larik menyatakan pasangan dari *vertex-vertex*

pada S dan pasangan dari *vertex-vertex* pada T. Pada awalnya semuanya diberi nilai -1 yang menandakan bahwa semua *vertex* baik di S maupun di T merupakan *unmatched vertex*. Proses inisialisasi perlu dipisah dari proses utama algoritma *Hungarian* agar dapat mempermudah memodifikasi algoritma *Hungarian* saat menjalankan algoritma *Hungarian* dinamis seperti yang telah disebutkan pada subbab ⁋2.5. *Pseudocode* algoritma *Hungarian* dapat dilihat pada Gambar ⁋3.2, Gambar ⁋3.3 dan Gambar ⁋3.4.

```
INIT-HUNGARIAN
   Input :
     -
   Output :
     -

1. for i = 1 to n do
2.    v[i] = -∞
3.    for j = 1 to n do
4.        u[i] ← max(u[i], w[i][j])
5.    od
6.    mateS[i] = mateT[i] = -1
7. od
```

Gambar ⁋3.1 *Pseudocode* fungsi `INIT-`*`HUNGARIAN`*

```
HUNGARIAN
Input :
  n       : banyaknya vertex
  w[n][n] : bobot pada graf bipartite

Output :
  bobot optimal matching

1. INIT-HUNGARIAN
2. nres ← 0
3. for i = 1 to n do
```

Gambar ⁋3.2 *Pseudocode* fungsi *`HUNGARIAN`* (bagian 1)

```
4.    if mates[i] = -1
5.      nres ← nres + 1
6.    fi
7.  od
8.  while nres ≠ 0 do
9.    for i = 1 to n do
10.     m[i] ← false
11.     p[i] ← 0
12.     slack[i] ← ∞
13.   od
14.   aug ← false
15.   Q ← {pilih satu i ∈ S dengan mate[i] = -1}
16.   do
17.     ambil sebuah vertex i dari Q
18.     m[i] ← true
19.     j ← 1
20.     while aug = false and j ≤ n do
21.       if mateS[i] ≠ j
22.         if u[i] + v[j] – w[i][j] < slack[j]
23.           slack[j] ← u[i] + v[j] – w[i][j]
24.           p[j] ← i
25.           if slack[j] = 0
26.             if(mateT[j] = -1)
27.               AUGMENT(j)
28.               aug ← true
29.               nres ← nres – 1
30.             else
31.               masukkan mateT[j] ke Q
32.             fi
33.           fi
34.         fi
35.       fi
36.       j ← j + 1
37.     od
38.     if aug = false and Q = NULL
39.       J ← {i ∈ S : m[i] = true}
40.       K ← {j' ∈ T : slack[j] = 0}
41.       minSlack ← min{slack[j] : j' ∈ (T\K)}
```

Gambar 3.3 *Pseudocode* fungsi *HUNGARIAN* (bagian 2)

```
42.       for each i ∈ J do
43.         u[i] ← u[i] – minSlack
44.       od
45.       for each j' ∈ K do
46.         v[j] ← v[j] + minSlack
47.       od
48.       for each j' ∈ (T\K) do
49.         slack[j] ← slack[j] – minSlack
50.       od
51.       X ← {j' ∈ (T\K) : slack[j] = 0}
52.       if mateT[j] ≠ -1 untuk semua j' ∈ X
53.         for j' ∈ X do
54.           masukkan mateT[j] ke Q
55.         od
56.       else
57.         pilih satu j' ∈ X dimana mateT[j] = -1
58.         AUGMENT(j)
59.         aug ← true
60.         nres ← nres – 1
61.       fi
62.     fi
63.   while aug = false
64. od
65. ans ← 0
66. for i = 1 to n do
67.   ans ← ans + u[i] + v[i]
68. od
69. return ans
```

Gambar 3.4 *Pseudocode* fungsi *HUNGARIAN* (bagian 3)

Pseudocode di atas merupakan *pseudocode* algoritma *Hungarian*. Baris pertama adalah proses inisialisasi yang telah dijelaskan sebelumnya. Baris ke-2 hingga baris ke-7 adalah proses penghitungan berapa *vertex* yang masih berstatus *unmatched vertex*. Kompleksitas waktu dari baris ke-2 hingga baris ke-8 adalah $O(n)$. Baris ke-8 hingga baris ke-64 adalah proses pencarian *optimal matching*. Disepakati bahwa tiap satu iterasi pada bagian ini dinamakan sebuah *fase*. Baris ke-9 hingga baris ke-13 adalah inisialisasi larik `m`, `p` dan `slack`. Nilai `m[i]`

menyatakan apakah *vertex* $i \in S$ merupakan *outer vertex* atau bukan. Nilai `slack[j]` menyatakan nilai terkecil `u[i] + v[j] - w[i][j]` untuk $i = 1 \ldots n$. Lalu nilai `p[j]` menyatakan *vertex* pertama pada i yang membuat nilai `slack[j]` menjadi nilai yang terkecil. Baris ke-14 adalah pemberian tanda jika pada mulanya *augmenting path* belum ditemukan. Pada baris ke-15 terdapat variabel `Q` yang digunakan untuk menampung kandidat *outer vertex* dimana dalam hal ini *outer vertex* awalnya adalah salah satu *vertex* $i \in S$ yang belum masuk ke dalam *optimal matching*. Dari hal ini dapat disimpulkan bahwa semua *outer vertex* pasti merupakan *vertex* pada S. Dalam hal ini dijamin juga bahwa semua *vertex* pasti masuk ke dalam *optimal matching* sesuai dengan Teorema 2.2 dan Lemma 2.3. Baris ke-16 hingga baris ke-63 merupakan proses pencarian *augmenting path*.

Inti dari proses pencarian *augmenting path* pada *pseudocode* baris ke-16 hingga baris ke-63 adalah dengan menentukan sebuah *vertex* pada `Q` yang pada awalnya adalah *unmatched vertex*. Ketika *vertex* tersebut memiliki *edge* yang ada dalam *equality subgraph* maka dilihat apakah *vertex* tetangganya pada *edge* tersebut merupakan *unmatched vertex* atau *matched vertex*. Jika *unmatched vertex* maka pasti *augmenting path* ditemukan. Jika tidak maka pasangan dari tetangganya dimasukkan ke dalam `Q` dan menjadikannya kandidat *outer vertex*. Proses ini diulangi hingga salah satu kandidat *outer vertex pada* `Q` diambil pada saat proses baris ke-17 dan memiliki *edge* pada *equality subgraph* serta *vertex* tetangganya pada *edge* tersebut merupakan *unmatched vertex*. Pada saat pencarian apakah sebuah *outer vertex* memiliki *edge* pada *equality subgraph* atau tidak, direkam juga nilai `slack[j]` dan `p[j]` sebagaimana yang telah disebutkan sebelumnya. Perlu dicatat bahwa sebuah *outer vertex* memiliki *edge* pada *equality subgraph* jika *outer vertex* tersebut menyebabkan nilai `slack[j]` menjadi 0. Proses ini ditunjukkan pada baris ke-17 hingga baris ke-27.

Hasil yang diharapkan setelah proses yang dijelaskan di atas selesai adalah ditemukannya *augmenting path* sehingga dapat menaikkan kardinalitas dari *optimal matching*. Namun jika ada kondisi dimana kandidat *outer vertex* sudah tidak ada dan *augmenting path* belum ditemukan maka hal itu pasti merupakan kondisi yang telah dijelaskan pada subbab 2.4 dimana ada kalanya $|N(J)| < |J|$ dimana J dalam hal ini adalah *outer vertex* dan $N(J)$ adalah *inner vertex*. Hal ini dapat terjadi karena semua *vertex* yang ada terhubung dengan sebuah *unmatched vertex* yang dipilih pada baris ke-15 melalui *alternating path*. Hal yang dapat dilakukan adalah mengubah $< u, v >$ menjadi $< u', v' >$ sebagaimana yang telah dijelaskan pada pembuktian Lemma 2.3. Proses ini dikerjakan pada *pseudocode* baris ke-39 hingga baris ke-50. Baris ke-39 adalah proses pencarian *outer vertex* dimana dalam hal ini informasi didapatkan saat nilai `m[i]` untuk $i \in S$ bernilai benar. Baris ke-39 adalah proses pencarian *inner vertex* dimana dalam hal ini ditandai dengan `slack[j] = 0` untuk $j' \in T$. Baris ke-40 adalah proses pencarian nilai `slack[j]` terkecil yang ditampung dalam variabel `minSlack` dimana j bukan bagian dari *inner vertex*. Lalu baris ke-42 hingga haris ke-47 merupakan proses perubahan $< u, v >$ menjadi $< u', v' >$. Perlu diperhatikan bahwa akibat perubahan yang dilakukan, semua nilai `slack[j]` dimana *j'* bukan bagian dari *inner vertex* pasti berkurang sebesar `minSlack`. Hal tersebut dilakukan pada proses baris ke-48 hingga baris ke-50.

Proses yang dilakukan di atas setidaknya akan menambah satu *edge* pada *equality subgraph* yang baru dimana *edge* tersebut bersinggungan dengan *vertex j'* yang tidak termasuk dalam *inner vertex*. Misal *vertex j'* adalah semua *vertex* yang tidak termasuk dalam *inner vertex*, jika salah satu *j'* memiliki `slack[j] = 0` serta *vertex* tersebut adalah *unmatched vertex*, maka *augmenting path* ditemukan. Jika tidak maka semua pasangan dari *vertex j'* yang memiliki nilai `slack[j] = 0` dimasukkan ke dalam `Q`. Proses tersebut dilakukan dari baris ke-51 hingga baris ke-61. Terakhir, dihitung jumlah *feasible node-weighting* yang

merupakan bobot *optimal matching* yang dicari. Jika ingin mengetahui masing-masing pasangan pada *optimal matching* maka akan dapat mengetahui pasangan dari *vertex* ke-i pada S atau membaca larik `mateT` maka cukup dengan membaca larik `mateS` akan dapat mengetahui pasangan dari *vertex* ke-j pada T.

```
AUGMENT
Input :
  j : outer vertex berakhirnya augmenting path
Output :
  -

1. do
2.    i ← p[j]
3.    mateT[j] ← i
4.    next = mateS[i]
5.    mateS[i] ← j
6.    if next ≠ -1
7.      j ← next
8.    fi
9. while(next ≠ -1)
```

Gambar 3.5 *Pseudocode* fungsi *AUGMENT*

Iterasi baris ke-8 hingga baris ke-64 berjalan paling banyak n kali. Perlu dicatat bahwa tiap *vertex* pasti masuk ke dalam `Q` tidak lebih dari sekali karena sebuah *vertex* masuk ke `Q` jika nilai ada nilai `slack[j]` pasangannya memiliki nilai 0 dan nilai `u[i] + v[j] – w[i][j]` tidak mungkin kurang dari 0 karena sifat *feasible node-weighting* itu sendiri. Maka dari itu, hal terburuk yang dapat terjadi apabila semua *vertex* pada S masuk ke `Q` dimana iterasi dari baris ke-16 hingga baris ke-62 berjalan n kali. Dapat diketahui dengan jelas juga bahwa iterasi dari baris ke-20 hingga baris ke-37 berjalan maksimal n kali. Semua proses yang ada pada baris 39, 40, 41, dan 52 masing-masing memiliki kompleksitas $O(n)$. Proses baris ke-42 hingga baris ke-44, proses baris ke-45 hingga baris ke-47 dan proses baris ke-48 hingga proses baris ke-50 masing-masing juga memiliki kompleksitas

$O(n)$. Maka dapat disimpulkan bahwa satu fase algoritma *Hungarian* memiliki kompleksitas $O(n^2)$ sehingga kompleksitas total Algoritma *Hungarian* adalah $O(n^3)$.

Perlu diperhatikan juga terdapat fungsi `AUGMENT` yang digunakan untuk meningkatkan kardinalitas sebuah *matching* saat ditemukan *augmenting path*. Cara kerja dari fungsi itu adalah dengan membuat *matched edge* menjadi *unmatched edge* dan begitu juga sebaliknya. *Pseudocode* dari fungsi `AUGMENT` adalah seperti pada Gambar 3.5.

3.2 Perancangan Algoritma *Hungarian* Dinamis

Sesuai dengan apa yang telah dibahas pada subbab 2.5, algoritma *Hungarian* dinamis bisa dikatakan sebagai modifikasi dari algoritma *Hungarian*. Hal ini dikarenakan algoritma *Hungarian* dinamis disusun oleh beberapa penyesuaian akibat perubahan bobot sebagaimana yang telah disebutkan pada subbab 2.5. Setelah penyesuaian dilakukan, maka dijalankan kembali satu fase algoritma *Hungarian* karena hal paling buruk yang dapat terjadi adalah ketika hilangnya satu *matched edge* pada *optimal matching*. Pada subbab ini akan dibahas *pseudocode* algoritma *Hungarian* dinamis.

Ada empat jenis perubahan yang telah dibahas pada subbab 2.5. Masing-masing perubahan memiliki penyesuaian sendiri-sendiri. Sebelum membahas hal ini terlalu dalam, terlebih dahulu jenis-jenis perubahan yang telah disebutkan disingkat penyebutannya sehingga akan lebih mudah dalam membahas metode yang digunakan. Berikut adalah penyingkatan penyebutan jenis-jenis perubahan:

1) perubahan nilai bobot pada *edge* (i, j') untuk semua $j' \in T$ disingkat menjadi ***update row***
2) perubahan nilai bobot pada *edge* (i, j') untuk semua $i \in S$ disingkat menjadi ***update column***
3) perubahan nilai bobot pada sebuah *edge* (i, j') disingkat menjadi ***update cell***

4) penambahan dua *vertex* masing-masing pada S dan T disingkat menjadi ***add vertex***

Bersadarkan keempat jenis perubahan di atas, akan dibahas satu per satu *pseudocode* penyesuaian untuk masing-masing jenis perubahan. Perubahan pertama disingkat menjadi *update row*. Hal ini didasarkan pada sifat perubahannya sendiri dimana jika data bobot direpresentasikan pada larik dua dimensi `w` dengan dimensi pertama menandakan elemen pada S dan dimensi kedua menandakan elemen pada T, maka perubahan tersebut tidak lain adalah perubahan semua nilai pada sebuah baris pada larik `w`. Pernyesuaian yang dilakukan adalah seperti yang telah dibahas pada Persamaan 2.8 dan diberi nama dengan fungsi *update-row*.

```
UPDATE-ROW
Input :
  r       : baris yang akan diubah
  newW[] : larik yang berisi bobot baru
Output :
  -

1. u[r] ← -∞
2. for j = 1 to n do
3.    w[r][j] ← newW[j]
4.    u[r] ← max(u[r], w[r][j] - v[j])
5. od
6. mateT[ mateS[r] ] ← -1
7. mates[ r ] ← -1
```

Gambar 3.6 *Pseudocode* fungsi *UPDATE-ROW*

Gambar 3.6 adalah *pseudocode* dari fungsi *UPDATE-ROW* sebagaimana yang telah dijelaskan pada subbab 2.5.1. Baris ke-1 hingga baris ke-4 adalah proses perubahan bobot dan penyesuaian *feasible node-weighting* u. Baris ke-5 dan baris ke-6 merupakan penghapusan *edge matching* yang bersinggungan dengan *vertex* ke-r pada S. Berikutnya adalah penyesuaian yang dilakukan

untuk perubahan tipe *update column* akan dijelaskan pada Gambar 3.7.

```
UPDATE-COLUMN
Input :
  c      : kolom yang akan diubah
  newW[] : larik yang berisi bobot baru
Output :
  -

1. v[c] ← -∞
2. for i = 1 to n do
3.   w[i][c] ← newW[i]
4.   v[c] ← max(v[c], w[i][c] – u[i])
5. od
6. mateS[ mateT[c] ] ← -1
7. mateT[ c ] ← -1
```

Gambar 3.7 *Pseudocode* fungsi *UPDATE-COLUMN*

Gambar 3.7 adalah *pseudocode* dari fungsi *UPDATE-COLUMN* sebagaimana yang telah dijelaskan pada subbab 2.5.2. Sekilas *pseudocode* ini mirip dengan *pseudocode* sebelumnya. Baris ke-1 hingga baris ke-4 adalah proses perubahan bobot dan penyesuaian *feasible node-weighting* v. Baris ke-5 dan baris ke-6 merupakan penghapusan *edge matching* yang bersinggungan dengan *vertex* ke-c pada T. Berikutnya adalah penyesuaian yang dilakukan untuk perubahan tipe *update cell*. Gambar 3.8 merupakan *pseudocode* untuk penyesuaian ketika hanya terjadi perubahan pada sebuah *cell* yang diberi nama fungsi *UPDATE-CELL*. *Pseudocode* ini mengikuti apa yang telah dijelaskan pada subbab 2.5.3. Baris ke-1 hingga baris ke-4 merupakan proses penghapusan *edge* ketika perubahan terjadi pada *edge* yang ada di dalam *optimal matching* dan bobot yang sekarang lebih kecil dari bobot sebelumnya. Baris ke-5 hingga baris ke-13 merupakan *pseudocode* untuk penyeseuaian *feasible node-weighting* u sehingga nilai $< u, v >$ tetap terjaga. Hal ini dikarenakan bobot yang sekarang lebih besar dari bobot yang sebelumnya dan

menyebabkan `u[i] + v[j] < w[i][j]`. Jika *edge* yang nilainya membesar tersebut tidak berada pada *optimal matching* maka ada kemungkinan susunan *optimal matching* berubah. Hal itu sudah ditangani pada proses baris ke-10 hinga baris ke-13.

```
UPDATE-CELL
Input :
  i, j : koordinat cell yang diubah nilainya
  newW : bobot baru pada w[i][j]
Output :
  -

1. if newWeight < w[i][j] and mateS[i] = j
2.   w[i][j] ← newW
3.   mateT[ mateS[i] ] ← -1
4.   mateS[ i ] ← -1
5. else if newW > w[i][j] and u[i] + v[j] < newW
6.   w[i][j] ← new
7.   u[i] ← -∞
8.   for k = 1 to n do
9.     u[i] ← max(u[i], w[i][k] – v[k])
10.  od
11.  if mateS[i] != j
12.    mateT[ mateS[i] ] ← -1
13.    mateS[ i ] ← -1
14.  fi
15.else
16.  W[i][j] ← newW
17.fi
```

Gambar 3.8 *Pseudocode* fungsi *UPDATE-CELL*

Selanjutnya akan dibahas *pseudocode* ketika terjadi perubahan jenis *add vertex*. Gambar 3.9 menunjukkan *pseudocode* untuk perubahan jenis *add vertex*. Seperti yang telah dijelaskan pada subbab 2.5.4, ketika terjadi penambahan *vertex*, nilai `u[n]` dan nilai `v[n]` disesuaikan mirip seperti *pseudocode* pada Gambar 3.6 dan *pseudocode* pada Gambar 3.7.

```
ADD-VERTEX
Input :
  -
Output :
  -

1. n ← n + 1
2. u[n] ← -∞
3. for j = 1 to n do
4.   u[n] ← max(u[n], w[n][j] - v[j])
5. od
6. v[n] ← -∞
7. for i = 1 to n do
8.   v[n] ← max(v[n], w[i][n] - u[i])
9. od
```

Gambar 3.9 *Pseudocode* fungsi *ADD-VERTEX*

Keempat jenis perubahan pada algoritma *Hungarian* dinamis telah dijelaskan di atas. Sekarang akan dibahas *Pseudo-code* dari algoritma *Hungarian* dinamis. Gambar 3.10 merupakan *pseudocode* fungsi *Hungarian* dinamis. Yang perlu diperhatikan dari jalannya algoritma ini adalah proses yang terjadi pada baris ke-12. Algoritma *Hungarian* dijalankan lagi tanpa harus menginisialisasi nilai *feasible node-weighting* dan susunan *matching* karena keduanya didapat dari algoritma *Hungarian* sebelumnya. Tentunya algoritma *Hungarian* yang dijalankan pada baris ke-12 dijalankan sebanyak k fase dimana k adalah banyaknya *edge* yang telah dihapus dari *optimal matching* karena perubahan-perubahan yang terjadi sebelumnya. Maka dari itu, kompleksitas dari algoritma ini adalah $O(kn^2)$ tiap ada permintaan bobot *optimal matching* terkini.

3.3 Perancangan Program Pembuatan Data Uji Coba

Pada subbab ini akan dibahas bagaimana cara merancang program untuk pembuatan data uji coba kinerja dari implementasi algoritma *Hungarian* dinamis yang telah dibuat. Program akan membuat data secara acak sesuai dengan parameter-parameter

yang telah diberikan dan data tersebut akan disimpan dalam sebuah *file*. Parameter-parameter yang dimaksud adalah banyaknya *vertex*, banyaknya operasi, interval terjadinya *query* dan interval terjadinya penambahan *vertex*. Dalam hal ini banyaknya operasi adalah jumlah dari banyaknya perubahan ditambahkan dengan banyaknya *query* dan banyaknya penambahan *vertex*.

```
HUNGARIAN-DINAMIS
Input :
  n       : banyaknya vertex
  w[n][n] : bobot pada graf bipartite
Output :
  bobot optimal matching saat diminta

1. HUNGARIAN
2. while ada perubahan do
3.   if jenis perubahan = update row
4.     UPDATE-ROW
5.   else if jenis perubahan = update column
6.     UPDATE-COLUMN
7.   else if jenis perubahan = update column
8.     UPDATE-CELL
9.   else if jenis perubahan = update column
10.    ADD-VERTEX
11.  else diminta bobot optimal saai ini
12.    Jalankan fungsi HUNGARIAN tanpa melakukan
       proses baris pertama
13.    Kembalikan bobot optimal matching proses
       di atas
14.  fi
15.od
```

Gambar 3.10 *Pseudocode* fungsi *HUNGARIAN*-DINAMIS

Inti proses kerja program ini adalah membuat matriks $n \times n$ secara acak dimana n adalah banyaknya *vertex*. Setelah itu dibuat m operasi secara acak juga. Pertama ditentukan tipe operasi secara acak sesuai dengan banyaknya jenis perubahan seperti yang disebutkan pada Gambar 3.10. Setelah itu

dibangkitkan angka-angka secara acak yang merepresentasikan jenis perubahan yang telah ditentukan. Jika angka-angka yang dibangkitkan secara acak tadi tidak menyebabkan penghapusan salah satu *edge* yang ada dalam *optimal matching* maka dibangkitkan lagi angka-angka hingga menyebabkan penghapusan salah satu *edge* yang ada dalam *optimal matching*. Hal ini dilakukan agar program implementasi yang dibuat benar-benar bisa dipastikan kinerjanya dalam menghadapi perubahan-perubahan yang mungkin terjadi. Parameter kedua dan ketiga dari program utama berupa interval agar operasi jenis *query* dan penambahan *vertex* dapat dikontrol jumlahnya. Hal ini dilakukan karena operasi tipe *query* tidak berpengaruh pada penghapusan *edge* dan operasi tipe penambahan *vertex* perlu dikontrol agar jumlah *vertex* tidak melebihi batasan yang ditentukan, yaitu 100 *vertex*. *Pseudocode* dari program dapat dilihat pada Gambar 3.11 hingga Gambar 3.14.

```
DATA-GENERATOR
Input :
  n  : banyaknya vertex
  m  : bobot pada graf bipartite
  q  : interval query
  a  : interval penambahan vertex

Output :
  File yang berisi data masukan untuk program
  implementasi algoritma Hungarian dinamis

println(n)
for i = 1 to n do
for j = 1 to n do
    tmp ← random()
    W[i][j] ← tmp
    print(tmp)
    if j != n-1
      print(" ")
```

Gambar 3.11 *Pseudocode DATA-GENERATOR* (bagian 1)

```
9.      fi
10.   od
11.   println()
12. od
13. initHungarian()
14. hungarian()
15. println(m)
16. i ← 1
17. while i <= m do
18.   if i % addInterval == 0
19.     println("A")
20.     u[n] ← -∞
21.     for c = 1 to n+1 do
22.       u[n] ← max(u[n], W[n][c] - v[c])
23.     od
24.     v[n] ← -∞;
25.     for r = 0 to n+1 do
26.       v[n] ← max(v[n], W[r][n] - u[r])
27.     od
28.     n++
29.     i++
30.     continue
31.   fi
32.   if i % queryInterval == 0
33.     i++;
34.     println("Q")
35.     hungarian()
36.     continue
37.   fi
38.   type ← (random() % MAXTYPE) + 1
39.   if type == 1
40.     changedRow ← random()%n
41.     if mateS[changedRow] == -1
42.       continue
43.     else
44.       mateT[ mateS[changedRow] ] ← -1
45.       mateS[ changedRow ] ← -1
46.     fi
47.     print("X ", changedRow)
48.     for j = 1 to n do
```

Gambar 3.12 *Pseudocode DATA-GENERATOR* (bagian 2)

```
49.       tmp ← random()
50.       W[changedRow][j] ← tmp
51.       print(" ", tmp)
52.     od
53.     u[changedRow] ← -∞;
54.     for c = 1 to n do
55.       u[changedRow] ← max(u[changedRow],
W[changedRow][c] - v[c])
56.     od
57.     println()
58.     i++
59.   else if type==2
60.     changedColumn ← random() % n
61.     if mateT[changedColumn] == -1
62.       continue
63.     fi
64.     else
65.       mateS[ mateT[changedColumn] ] ← -1
66.       mateT[ changedColumn ] ← -1
67.     fi
68.     print("Y ", changedColumn)
69.     for j = 1 to n do
70.       tmp ← random()
71.       W[j][changedColumn] ← tmp;
72.       print(" ", tmp)
73.     od
74.     v[changedColumn] ← -∞
75.     for r = 1 to n do
76.       v[changedColumn] ← max(
v[changedColumn], W[r][changedColumn] - u[r])
77.     od
78.     println()
79.     i++;
80.   else if(type==3)
81.     changedRow ← random()%n
82.     changedColumn ← random()%n
83.     tmp ← random();
84.     if (tmp < W[changedRow][changedColumn])
&& (mateS[changedRow] == changedColumn)
85.       if(mateS[changedRow] == -1)
```

Gambar 3.13 *Pseudocode DATA-GENERATOR* (bagian 3)

```
86.          continue
87.        else
88.          mateT[ mateS[changedRow] ] ← -1
89.          mateS[changedRow] ← -1
90.        fi
91.        W[changedRow][changedColumn] ← tmp
92.      else if (tmp >
W[changedRow][changedColumn]) && (u[changedRow] +
v[changedColumn] < tmp)
93.        if mateS[changedRow] == changedColumn
94.          continue
95.        else
96.          mateT[ mateS[changedRow] ] ← -1
97.          mateS[changedRow] ← -1
98.        fi
99.        W[changedRow][changedColumn] ← tmp
100.     u[changedRow] ← -∞
101.     for c = 1 to n do
102.       u[changedRow] ← max(u[changedRow],
W[changedRow][c] - v[c]);
103.     od
104.   else
105.     continue;
106.   fi
107.   println("C ", changedRow, " ", changedColumn,
tmp)
108.   println()
109.   i++
110.fi
111.od
```

Gambar 3.14 *Pseudocode DATA-GENERATOR* (bagian 4)

Pada *pseudocode* di atas, baris ke-1 hingga baris ke-12 pada Gambar 3.11 dan Gambar 3.12 merupakan proses pembuatan matriks berukuran $n \times n$ dimana nilai tiap elemen matriks diberi nilai secara acak. Setelah itu dijalankan algoritma *Hungarian* untuk mendapatkan *optimal matching*-nya. Kemudian dibuat m operasi secara acak kecuali pada operasi *query* dan *penambahan vertex* yang dibuat secara periodik sesuai dengan besaran interval yang ditentukan. Baris ke-13 hingga baris ke-31

merupakan proses pembangkitan bilangan acak untuk operasi tipe penambahan *vertex*. Baris ke-32 hingga baris ke-37 merupakan proses pembangkitan bilangan acak untuk operasi tipe *query*. Pada proses ini dijalankan juga algoritma *Hungarian* dinamis untuk mendapatkan *optimal matching*-nya. Baris ke-38 hingga baris ke-58 pada Gambar 3.13 merupakan proses pembangkitan bilangan acak untuk operasi tipe *update row*. Dalam hal ini bilangan acak yang dibangkitkan diperiksa terlebih dahulu apakah menyebabkan sebuah *edge* dalam *optimal matching* dihapus atau tidak. Jika tidak maka proses ini dilewati. Pengecekan ini akan dilakukan pada operasi tipe *update column* dan *update cell*. Alasan dari pengecekan ini telah dikemukakan sebelumnya. Baris ke-59 hingga baris ke-79 merupakan proses pembangkitan bilangan acak untuk operasi tipe *update column* dan baris ke-80 hingga baris ke-110 pada Gambar 3.14 merupakan proses pembangkitan bilangan acak untuk operasi tipe *update cell*.

BAB IV
IMPLEMENTASI

Pada bab ini akan dibahas tentang implementasi yang dilakukan berdasarkan apa yang telah dijabarkan dan dirancang pada bab III.

4.1 Lingkungan Implementasi

Bahasa yang digunakan pada implementasi ini adalah bahasa pemrograman C++ dengan bantuan *Integrated Development Environment* (IDE) wxDev-C++ versi 7.4.2.596. Sistem operasi yang digunakan adalah Windows 7 Home 32 bit. Perangkat keras yang digunakan adalah prosesor Intel(R) Core(TM) i3 CPU M350 2.27 GHz dengan *Random Access Memory* (RAM) sebesar 2.00 GB.

4.2 Implementasi

Pada subbab ini akan dijelaskan masing-masing implementasi dari algoritma *Hungarian* dan algoritma *Hungarian* dinamis sebagaimana yang telah dijelaskan pada subbab ⁋3.1 dan subbab ⁋3.2 dan implementasi dari program uji coba yang dijleaskan pada subbab ⁋3.3. Sebelumnya juga akan dijelaskan rancangan data masukan dari program yang dibuat.

4.2.1 Rancangan Data Masukan

Pada subbab ini akan dijelaskan rancangan data yang diperlukan untuk proses uji coba kinerja algoritma. Perancangan data diperlukan agar program yang dibuat dapat beroperasi sesuai dengan fungsinya dan agar dapat dimengerti oleh pengguna.

Rancangan data masukan disesuaikan dengan soal pada situs penilaian daring SPOJ yang berjudul *Dynamic Assignment Problem* seperti yang telah dijelaskan pada subbab ⁋2.7. Baris pertama dari masukan adalah N yang menyatakan jumlah *vertex* pada graf. Baris kedua hingga baris $N + 1$ masing-masing terdiri

dari N angka yang menyatakan matriks bobot dari *complete bipartite graph* yang akan dicari *optimal matching*-nya. Baris ke-$(N+2)$ terdapat sebuah bilangan bulat M yang menyatakan berapa banyak operasi yang akan terjadi. M baris berikutnya akan berisi sebagai berikut:

1. $X\ i\ x_0\ x_1\ x_2 \ldots x_{N-1}$
2. $Y\ i\ y_0\ y_1\ y_2 \ldots y_{N-1}$
3. $C\ i\ j\ w$
4. A
5. Q

Kelima operasi di atas telah dijelaskan maksudnya pada subbab 2.7 begitu juga batasan-batasan dari data masukan.

4.2.2 Penggunaan *Library*, Konstanta dan Variabel Global

Pada subbab ini akan dibahas penggunaan *library*, konstanta dan variabel-variabel yang akan digunakan secara global pada program dimana variabel-variabel ini akan digunakan pada beberapa fungsi. Berikut adalah potongan kode yang menyatakan penggunaan *library* dan konstanta:

```
1.  #include<cstdio>
2.  #include<cstring>
3.  #include<list>
4.  #include<algorithm>
5.  using namespace std;
6.  typedef long long LL;
7.  const LL INF = (LL)1e15;
8.  const int MAXV = 105;
```

Kode Sumber 4.1 Potongan kode pemanggilan *library* dan penggunaan konstanta

Pada Kode Sumber 4.1, terlihat bahwa terdapat empat *library* yang dipanggil. Didefinisikan juga variabel `LL` yang digunakan untuk menyingkat variabel `long long` agar saat pendeklarasian variabel `long long` cukup mengetikkan `LL` saja

sebaga ganti `long long`. Konstanta `INF` menandakan bahwa dalam program ini nilai *infinity* atau tak hingga dari program ini adalah 10^{15}. Konstanta `MAXV` menandakan bahwa jumlah *vertex* pada graf *bipartite* yang akan digunakan dalam program ini tidak akan pernah melebihi 100 *vertex* sesuai dengan batasan yang ditetapkan pada bab I. Berikut adalah tabel yang menjelaskan variabel-variabel global yang akan digunakan pada implementasi program:

Tabel 4.1 Tabel daftar variabel global (bagian 1)

No	Nama Variabel	Tipe	Penjelasan
1	`N`	`int`	Jumlah *vertex* mula-mula
2	`W`	`long long[][]`	Digunakan untuk menyimpan bobot *complete bipartite graph*
3	`mateS`	`int[]`	Digunakan untuk menyimpan pasangan *vertex* di himpunan *vertex S* graf *bipartite* pada *matching*
4	`mateT`	`int[]`	Digunakan untuk menyimpan pasangan *vertex* di himpunan *vertex T* graf *bipartite* pada *matching*
5	`u`	`long long[]`	Digunakan untuk menyimpan *feasible node-weighting u*
6	`v`	`long long[]`	Digunakan untuk menyimpan *feasible node-weighting v*

Tabel 4.2 Tabel daftar variabel global (bagian 2)

7	`slack`	`long long[]`	Digunakan untuk menyimpan nilai `u[i]+v[j]-W[i][j]` minimal untuk tiap `j`
8	`p`	`int[]`	Digunakan untuk menyimpan *vertex* keberapa pada himpunan *vertex S* graf *bipartite* yang menyebabkan nilai `slack[j]` menjadi minimal
9	`m`	`bool[]`	Digunakan untuk menandai bahwa sebuah *vertex* pada himpunan *vertex S* graf *bipartite* merupakan *outer vertex* atau tidak
10	`Q`	`list<int>`	Digunakan untuk menampung kandidat *outer vertex*

Potongan kode yang menyatakan pendeklarasian variabel global sesuai dengan Tabel 4.1 dan Tabel 4.2 di atas adalah seperti pada Kode Sumber 4.2.

```
1.  int N, mateS[MAXV], mateT[MAXV], p[MAXV];
2.  LL u[MAXV], v[MAXV], slack[MAXV];
3.  LL W[MAXV][MAXV];
4.  bool m[MAXV];
5.  list<int> Q;
```

Kode Sumber 4.2 Potongan kode pendeklarasian variabel global

4.2.3 Implementasi Algoritma *Hungarian*

Sebelum algoritma *Hungarian* dijalankan, terlebih dahulu dibaca masukan berupa matriks dua dimensi yang menyatakan bobot dari graf yang akan dicari *optimal matching*-nya. Dalam hal ini graf yang dimaksud selalu berbentuk *complete bipartite graph* dengan bobot nonnegatif. Misal graf tersebut dinyatakan dengan $G = (S \cup T, E)$, maka nilai baris ke-i kolom ke-j pada matriks masukan menyatakan bobot dari *edge* yang mengubungkan sebuah vertex $i \in S$ dan sebuah vertex $j \in T$. Seperti yang telah dijelaskan pada subbab 4.2.1, pada baris pertama masukan terdapat bilangan N yang menyatakan jumlah *vertex* pada *complete bipartite graph* diikuti dengan N baris dimana untuk tiap baris ke-i, $i \in S$, terdapat N bilangan yang menyatakan bobot *edge-edge* yang terhubung dengan *vertex* i. Potongan kode untuk membaca masukan matriks ditunjukkan oleh Kode Sumber 4.3.

```
1.  void readMatrix()
2.  {
3.    scanf("%d",&N);
4.    for(int i=0;i<N;i++)
5.      for(int j=0;j<N;j++)
6.        scanf("%lld",&W[i][j]);
7.  }
```

Kode Sumber 4.3 Fungsi pembacaan masukan matriks bobot

Setelah matriks bobot disimpan dalam sebuah larik dua dimensi `W`, dijalankan fungsi `INIT-HUNGARIAN` seperti yang ditunjukkan pada Gambar 3.1. Fungsi ini akan menginisialisasi nilai feasible node weighting sesuai dengan Persamaan 2.7. Potongan kode dari implementasi fungsi `INIT-HUNGARIAN` pada Gambar 3.1 ditunjukkan pada Kode Sumber 4.4.

Pada *pseudocode* algoritma *Hungarian* yang ditunjukkan oleh Gambar 3.2, terdapat pemanggilan fungsi `AUGMENT` yang digunakan untuk meningkatkan kardinalitas saat sebuah

augmenting path ditemukan dengan membalikkan status *edge* pada *augmenting path* dimana *matched edge* diubah menjadi *unmatched edge* dan begitu juga sebaliknya. Fungsi AUGMENT telah ditunjukkan pada Gambar 3.5. Potongan kode implementasi fungsi AUGMENT pada Gambar 3.5 ditunjukkan pada Kode Sumber 4.5.

```
1.  void initHungarian()
2.  {
3.    memset(mateS, -1, sizeof(mateS));
4.    memset(mateT, -1, sizeof(mateT));
5.    for(int i=0;i<N;i++)
6.    {
7.      u[i] = -INF;
8.      for(int j=0;j<N;j++)
9.        u[i] = max(u[i], W[i][j]);
10.     v[i] = 0;
11.   }
12. }
```

Kode Sumber 4.4 Kode implementasi fungsi INIT-HUNGARIAN pada Gambar 3.1

```
1.  void augment(int j)
2.  {
3.    int i, next;
4.    do{
5.      i = p[j]; mateT[j] = i;
6.      next = mateS[i]; mateS[i] = j;
7.      if(next!=-1) j = next;
8.    }while(next!=-1);
9.  }
```

Kode Sumber 4.5 Kode implementasi fungsi AUGMENT pada Gambar 3.5

Setelah fungsi AUGMENT diimplementasikan, berikut ini akan diimplementasikan algoritma *Hungarian*. Potongan kode implementasi algoritma *Hungarian* ditunjukkan pada Kode Sumber 4.6, Kode Sumber 4.7 dan Kode Sumber 4.8.

```
1.  LL hungarian()
2.  {
3.    int nres = 0;
4.    for(int i=0;i<N;i++)
5.      if(mateS[i]==-1)
6.        nres++;
7.    while(nres>0)
8.    {
9.      for(int i=0;i<N;i++)
10.     {
11.       m[i] = false;
12.       p[i] = -1;
13.       slack[i] = INF;
14.     }
15.     bool aug = false;
16.     Q.clear();
17.     for(int i=0;i<N;i++)
18.       if(mateS[i]==-1)
19.       {
20.         Q.push_back(i);
21.         break;
22.       }
23.     do{
24.       int i, j;
25.       i = Q.front();
26.       Q.pop_front();
27.       m[i] = true;
28.       j = 0
29.       while(aug==false && j<N)
30.       {
31.         if(mateS[i]!=j)
32.         {
33.           LL minSlack = u[i] + v[j] -
    W[i][j];
34.           if(minSlack < slack[j])
35.           {
```

Kode Sumber 4.6 Kode implementasi algoritma *Hungarian* (bagian 1)

```
36.                 slack[j] = minSlack;
37.                 p[j] = i;
38.                 if(slack[j]==0)
39.                 {
40.                   if(mateT[j]==-1)
41.                   {
42.                     augment(j);
43.                     aug = true;
44.                     nres--;
45.                   }
46.                   else
47.                     Q.push_back(mateT[j]);
48.                 }
49.               }
50.             }
51.             j++;
52.           }
53.           if(aug==false && Q.size()==0)
54.           {
55.             LL minSlack = INF;
56.             for(int k=0;k<N;k++)
57.               if(slack[k] > 0)
58.                 minSlack = min(minSlack,
slack[k]);
59.             for(int k=0;k<N;k++)
60.               if(m[k])
61.                 u[k] -= minSlack
62.             int x = -1;
63.             bool X[MAXV];
64.             for(int k=0;k<N;k++)
65.               if(slack[k] == 0)
66.                 v[k] += minSlack;
67.               else
68.               {
69.                 slack[k] -= minSlack;
```

Kode Sumber 4.7 Kode implementasi algoritma *Hungarian* (bagian 2)

```
70.             if(slack[k]==0 && mateT[k]==-
1)
71.               x = k;
72.             if(slack[k]==0)
73.               X[k] = true;
74.             else
75.               X[k] = false;
76.           }
77.         if(x==-1)
78.         {
79.           for(int k=0;k<N;k++)
80.             if(X[k])
81.               Q.push_back(mateT[k]);
82.         }
83.         else
84.         {
85.           augment(x);
86.           aug = true;
87.           nres--;
88.         }
89.       }
90.     }while(aug==false);
91.   }
92.   LL ans = 0;
93.   for(int i=0;i<N;i++)
94.   ans += (u[i] + v[i]);
95.   return ans;
96. }
```

Kode Sumber 4.8 Kode implementasi algoritma *Hungarian* (bagian 3)

Potongan kode implementasi algoritma *Hungarian* di atas tidak jauh berbeda dengan apa yang ditunjukkan oleh *pseudocode* fungsi `HUNGARIAN` yang ada pada Gambar 3.2.

4.2.4 Implementasi Algoritma *Hungarian* Dinamis

Pada subbab ini akan dibahas implementasi algoritma *Hungarian* dinamis. Implementasi algoritma ini mengacu pada semua *pseudocode* yang telah dijelaskan pada subbab ⁋3.2. Namun demikian, implementasi tiap jenis perubahan tidak dibuat dalam fungsi yang berbeda-beda seperti yang ditunjukkan pada Gambar ⁋3.6 hingga Gambar ⁋3.9 melainkan dijadikan satu karena implementasi tiap jenis perubahan relatif sederhana. Kode implementasi algoritma *Hungarian* dinamis ditunjukkan pada Kode Sumber ⁋4.9, Kode Sumber ⁋4.10 dan Kode Sumber ⁋4.11.

```
1.  void dynamicHungarian()
2.  {
3.    char type[2];
4.    LL w;
5.    int i, j;
6.    scanf("%s",type);
7.    if(type[0]=='C')
8.    {
9.      scanf("%d%d%lld",&i, &j, &w);
10.     if((w<W[i][j]) && (mateS[i]==j))
11.     {
12.       W[i][j] = w;
13.       if(mateS[i]!=-1)
14.       {
15.         mateT[ mateS[i] ] = -1;
16.         mateS[ i ] = -1;
17.       }
18.     }
19.     else if((w>W[i][j]) && (u[i]+v[j]<w))
20.     {
21.       W[i][j] = w;
22.       u[i] = -INF;
23.       for(int c=0;c<N;c++)
24.         u[i] = max(u[i], W[i][c]-v[c]);
```

Kode Sumber ⁋4.9 Kode implementasi algoritma *Hungarian* dinamis (bagian 1)

```
25.       if(mateS[i]!=j)
26.       {
27.         mateT[ mateS[i] ] = -1;
28.         mateS[i] = -1;
29.       }
30.     }
31.     else W[i][j] = w;
32.   }
33.   else if(type[0]=='X')
34.   {
35.     scanf("%d",&i);
36.     for(int c=0;c<N;c++)
37.       scanf("%lld", &W[i][c]);
38.     if(mateS[i]!=-1)
39.     {
40.       mateT[ mateS[i] ] = -1;
41.       mateS[ i ] = -1;
42.     }
43.     u[i] = -INF;
44.     for(int c=0;c<N;c++)
45.       u[i] = max(u[i], W[i][c]-v[c]);
46.   }
47.   else if(type[0]=='Y')
48.   {
49.     scanf("%d",&j);
50.     for(int r=0;r<N;r++)
51.       scanf("%lld", &W[r][j]);
52.     if(mateT[j]!=-1)
53.     {
54.       mateS[ mateT[j] ] = -1;
55.       mateT[ j ] = -1;
56.     }
57.     v[j] = -INF;
58.     for(int r=0;r<N;r++)
```

Kode Sumber 4.10 Kode implementasi algoritma *Hungarian* dinamis (bagian 2)

```
59.         v[j] = max(v[j], W[r][j]-u[r]);
60.     }
61.     else if(type[0]=='A')
62.     {
63.       i = j = N++;
64.       u[i] = -INF;
65.       for(int c=0;c<N;c++)
66.         u[i] = max(u[i], W[i][c]-v[c]);
67.       v[j] = -INF;
68.       for(int r=0;r<N;r++)
69.         v[j] = max(v[j], W[r][j]-u[r]);
70.     }
71.     else if(type[0]=='Q')
72.       printf("%lld\n", hungarian());
73. }
```

Kode Sumber 4.11 Kode implementasi algoritma *Hungarian* dinamis (bagian 3)

Cara kerja implementasi algoritma di atas adalah dilakukan terlebih dahulu pembacaan pada masukan apa tipe perubahan yang akan dilakukan. Baris ke-7 hingga baris ke-32 menunjukkan implementasi ketika terjadi perubahan tipe *update cell*. Hal tersebut sesuai dengan *pseudocode* yang ditunjukkan pada Gambar 3.8. Baris ke-33 hingga baris ke-46 menunjukkan implementasi ketika terjadi perubahan tipe *update row*. Hal tersebut sesuai dengan *pseudocode* yang ditunjukkan pada Gambar 3.6. Baris ke-47 hingga baris ke-60 menunjukkan implementasi ketika terjadi perubahan tipe *update column*. Hal tersebut sesuai dengan *pseudocode* yang ditunjukkan pada Gambar 3.7. Baris ke-61 hingga baris ke-70 menunjukkan implementasi ketika terjadi perubahan tipe *add vertex*. Hal tersebut sesuai dengan *pseudocode* yang ditunjukkan pada Gambar 3.9. Baris ke-71 dan baris ke-72 menjalankan algoritma *Hungarian* kembali tanpa melakukan inisialisasi *feasible node-weighting* dan tanpa menghapus *matching* yang masih ada untuk mendapatkan bobot *optimal matching* pada graf dengan bobot yang baru.

4.2.5 Implementasi Fungsi Utama Program

Pada fungsi utama program ini akan dijalankan fungsi-fungsi yang telah dijelaskan di atas kecuali fungsi augment karena fungsi tersebut hanya dipanggil di dalam fungsi hungarian. Potongan kode fungsi utama program ditunjukkan pada Kode Sumber 4.12.

```
1.  int main()
2.  {
3.    readMatrix();
4.    initHungarian();
5.    LL ans = hungarian();
6.    int M;
7.    scanf("%d",&M);
8.    while(M--) dynamicHungarian();
9.    return 0;
10. }
```

Kode Sumber 4.12 Potongan kode fungsi utama program

Pada Kode Sumber 4.12 di atas, alur kerja dari fungsi utama adalah membaca masukan matriks terlebih dahulu dengan menjalankan fungsi `readMatrix`. Setelah itu dijalankan fungsi `initHungarian` untuk menentukan nilai awal dari *feasible node-weighting* dan membuat semua *vertex* menjadi *unmatched vertex* dengan memberi nilai larik `mateS` dan `mateT` semuanya menjadi -1. Setelah itu dijalankan algoritma *Hungarian* untuk mendapatkan *optimal matching* dari *complete bipartite graph* dengan bobot yang ada. Kemudian dibaca jumlah perubahan yang mungkin terjadi lalu dijalankan algoritma *Hungarian* dinamis untuk menyelesaikan masalah penugasan dinamis.

4.2.6 Implementasi Program Pembuatan Data Uji Coba

Berikut ini adalah implementasi program pembuatan data uji coba yang telah dibahas pada subbab 3.3. Kode sumber program pembuatan data uji coba ditunjukkan pada Kode Sumber

⁋4.13, Kode Sumber ⁋4.14, Kode Sumber ⁋4.15, Kode Sumber ⁋4.16 dan Kode Sumber ⁋4.17.

```
#include<cstdio>
#include<cstring>
#include<cstdlib>
#include<algorithm>
#include<ctime>
#include<list>
using namespace std;
const int MAXTYPE = 3;
const int MAXN = 105;
typedef long long LL;
const LL INF = (LL)1e15;
const int MAXV = 105;
int N, mateS[MAXV], mateT[MAXV], p[MAXV];
LL u[MAXV], v[MAXV], slack[MAXV];
LL W[MAXV][MAXV];
bool m[MAXV];
list<int> Q;
void initHungarian(){}
void augment(int j){}
LL hungarian(){}
int main(int argc, char **argv)
{
  if (argc != 5) exit(3);
  int M, addInterval, queryInterval;
  sscanf(argv[1], "%d", &N);
  sscanf(argv[2], "%d", &M);
  sscanf(argv[3], "%d", &queryInterval);
  sscanf(argv[4], "%d", &addInterval);
  srand(time(NULL));
  char fileName[32];
  int NN = N;
  N = NN;
```

Kode Sumber ⁋4.13 Kode program pembuatan data uji coba (bagian 1)

```
33.   sprintf(fileName, "N%d-M%d-Q%d-A%d.in",
N, M, queryInterval, addInterval);
34.   freopen(fileName, "w", stdout);
35.   printf("%d\n", N);
36.   for(int i=0;i<N;i++)
37.   {
38.     for(int j=0;j<N;j++)
39.     {
40.       int tmp = rand() * rand();
41.       W[i][j] = tmp;
42.       printf("%d", tmp);
43.       if(j!=N-1) printf(" ");
44.     }
45.     printf("\n");
46.   }
47.   initHungarian();
48.   hungarian();
49.   printf("%d\n",M);
50.   for(int i=1;i<=M;)
51.   {
52.     if(i%addInterval==0)
53.     {
54.       printf("A\n");
55.       u[N] = -INF;
56.       for(int c=0;c<N+1;c++)
57.         u[N] = max(u[N], W[N][c]-v[c]);
58.       v[N] = -INF;
59.       for(int r=0;r<N+1;r++)
60.         v[N] = max(v[N], W[r][N]-u[r]);
61.       N++;
62.       i++;
63.       continue;
64.     }
65.     if(i%queryInterval==0)
66.     {
67.       i++;
```

Kode Sumber 4.14 Kode program pembuatan data uji coba (bagian 2)

```
68.         printf("Q\n");
69.         hungarian();
70.         continue;
71.       }
72.       int type = (rand()%MAXTYPE) + 1;
73.       if(type==1)
74.       {
75.         int changedRow = rand()%N;
76.         if(mateS[changedRow]==-1) continue;
77.         else
78.         {
79.           mateT[ mateS[changedRow] ] = -1;
80.           mateS[ changedRow ] = -1;
81.         }
82.         printf("X %d", changedRow);
83.         for(int j=0;j<N;j++)
84.         {
85.           int tmp = rand() * rand();
86.           W[changedRow][j] = tmp;
87.           printf(" %d", tmp);
88.         }
89.         u[changedRow] = -INF;
90.         for(int c=0;c<N;c++)
91.         u[changedRow] = max(u[changedRow],
W[changedRow][c]-v[c]);
92.         printf("\n");
93.         i++;
94.       }
95.       else if(type==2)
96.       {
97.         int changedColumn = rand()%N;
98.         if(mateT[changedColumn]==-1)
continue;
99.         else
100.        {
101.          mateS[ mateT[changedColumn] ]=-1;
102.          mateT[ changedColumn ] = -1;
```

Kode Sumber 4.15 Kode program pembuatan data uji coba (bagian 3)

```
103.        }
104.        printf("Y %d", changedColumn);
105.        for(int j=0;j<N;j++)
106.        {
107.          int tmp = rand() * rand();
108.          W[j][changedColumn] = tmp;
109.          printf(" %d", tmp);
110.        }
111.        v[changedColumn] = -INF;
112.        for(int r=0;r<N;r++)
113.          v[changedColumn] =
max(v[changedColumn], W[r][changedColumn]-
u[r]);
114.        printf("\n");
115.        i++;
116.      }
117.      else if(type==3)
118.      {
119.        int changedRow = rand()%N;
120.        int changedColumn = rand()%N;
121.        int tmp = rand() * rand();
122.        if((tmp < W[changedRow]
[changedColumn]) && (mateS[changedRow] ==
changedColumn))
123.        {
124.          if(mateS[changedRow]==-1)
continue;
125.          else
126.          {
127.            mateT[ mateS[changedRow] ]=-1;
128.            mateS[changedRow] = -1;
129.          }
130.        W[changedRow][changedColumn] = tmp;
131.        }
132.        else if((tmp>W[changedRow]
[changedColumn]) && (u[changedRow] +
v[changedColumn]<tmp))
```

Kode Sumber 4.16 Kode program pembuatan data uji coba (bagian 4)

```
133.        {
134.          if(mateS[changedRow] ==
changedColumn) continue;
135.          else
136.          {
137.            mateT[ mateS[changedRow] ]=-1;
138.            mateS[changedRow] = -1;
139.          }
140.          W[changedRow][changedColumn] =
tmp;
141.          u[changedRow] = -INF;
142.          for(int c=0;c<N;c++)
143.          u[changedRow] =
max(u[changedRow], W[changedRow][c]-v[c]);
144.        }
145.        else continue;
146.        printf("C %d %d", changedRow,
changedColumn);
147.        printf(" %d", tmp);
148.        printf("\n");
149.        i++;
150.      }
151.    }
152.    return 0;
153. }
```

Kode Sumber 4.17 Kode program pembuatan data uji coba (bagian 5)

Pada Kode Sumber 4.13, dapat dilihat pada baris ke-18 hingga baris ke-20 merupakan fungsi-fungsi yang dapat dilihat pada beberapa kode sumber sebelumnya.

BAB V
UJI COBA DAN ANALISIS

Pada bab ini akan dijelaskan proses uji coba dari program yang telah diimplementasikan pada bab sebelumnya. Hal-hal yang akan dibahas adalah lingkungan uji coba dan skenario uji coba. Skenario uji coba meliputi pengunggahan kode program yang telah dibuat ke situs pengujian daring SPOJ, uji coba kebenaran dan uji coba kinerja disertai dengan analisis kinerja program yang telah dibuat.

5.1 Lingkungan Uji Coba

Pada subbab ini akan dijelaskan lingkungan yang digunakan untuk uji coba program yang telah dibuat. Berikut adalah rincian lingkungan uji coba tugas akhir ini:

1. Perangkat Keras
   a. Prosesor : Intel(R) Core(TM) i3 CPU M350 2.27 GHz
   b. *Memory* : 2.00 GB
2. Perangkat Lunak
   a. Sistem operasi : Windows 7 Home 32 bit
   b. Perangkat pengembang : wxDev-C++ versi 7.4.2.596

5.2 Skenario Uji Coba

Pada subbab ini akan dijelaskan mekanisme uji coba yang dilakukan. Pertama akan dilakukan uji coba dengan mengunggah kode sumber dari program yang telah dibuat ke situs penilaian daring SPOJ pada soal yang berjudul *Dynamic Assignment Problem*. Uji coba selanjutnya yaitu uji coba kebenaran dari implementasi program yang telah dibuat. Dalam hal ini hasil akan diselesaikan masalah penugasan skala kecil dengan menggunakan metode *Hungarian* yang telah dibahas pada subbab 2.6 lalu hasilnya dibandingkan dengan keluaran dari program yang telah dibuat. Uji coba terakhir yaitu uji coba kinerja dari program yang telah dibuat. Dalam hal ini diukur waktu eksekusi program dalam

menyelesaikan masalah penugasan dinamis. Akan dianalisis juga pengaruh ukuran-ukuran data terhadap waktu eksekusi program.

5.2.1 Uji Coba Melalui SPOJ

Dalam uji coba ini, kode sumber dari program yang telah dibuat diunggah ke situs penilaian daring SPOJ pada soal yang berjudul *Dynamic Assignment Problem*. Ketentuan-ketentuan dari soal tersebut telah dijelaskan pada subbab 2.7. Format masukan juga telah disesuaikan dengan yang diminta pada soal tersebut. Setelah kode sumber diunggah, situs SPOJ akan memberikan umpan balik. Jika hasil keluaran program sesuai dengan ketentuan pada soal, maka situs SPOJ akan menampilkan umpan balik berupa tulisan "*Accepted*", tetapi jika sebaliknya, maka akan muncul umpan balik berupa tulisan "*Time Limit Exceeded*", "*Wrong Answer*", atau "*Runtime Error*".

Gambar 5.1 Umpan balik dari situs SPOJ

Dari uji coba ini, didapatkan bahwa kode program hasil implementasi tugas akhir mendapat umpan balik "*Accepted*" seperti pada Gambar 5.1 yang berarti program berjalan dengan benar dengan menggunakan memori sebesar 2.8 *Megabyte* dan waktu eksekusi program terhadap masukan SPOJ sebesar 0.31 sekon. Setelah itu kode program diunggah lagi sebanyak 20 kali untuk mendapatkan waktu eksekusi rata-rata dari program yang telah dibuat. Seperti yang tersaji pada Tabel 5.1, waktu eksekusi program terhadap masukan dari SPOJ adalah sekitar 0.31 detik bergantung pada tingkat kesibukan *server* saat pengumpulan dengan waktu eksekusi program tercepat yang diperoleh adalah 0.29 detik. *Memory* yang digunakan selama 20 kali pengunggahan kode sumber tetap 2.8 *Megabyte*. *Screenshot* status yang diberikan oleh SPOJ dalam pengunggahan kode sumber dapat dilihat juga pada Gambar A.1.

Tabel 5.1 Status pengumpulan kode sumber implementasi ke situs SPOJ sebanyak 20 kali

ID	RESULT	TIME	MEM
11651953	accepted	0.30	2.8M
11651951	accepted	0.32	2.8M
11651950	accepted	0.31	2.8M
11651947	accepted	0.31	2.8M
11651946	accepted	0.31	2.8M
11651942	accepted	0.31	2.8M
11651939	accepted	0.31	2.8M
11651935	accepted	0.31	2.8M
11651934	accepted	0.30	2.8M
11651930	accepted	0.31	2.8M
11651922	accepted	0.30	2.8M
11651921	accepted	0.31	2.8M
11651920	accepted	0.31	2.8M
11651916	accepted	0.31	2.8M
11651914	accepted	0.30	2.8M
11613918	accepted	0.31	2.8M
11613913	accepted	0.31	2.8M
11613911	accepted	0.31	2.8M
11613908	accepted	0.29	2.8M
11613907	accepted	0.31	2.8M

5.2.2 Uji Coba Kebenaran

Dalam subbab ini akan dibahas uji coba kebenaran dari kode program dengan cara membandingkan keluaran dari program dengan hasil dari metode hungarian seperti yang telah dijelaskan pada subbab 2.6. Meskipun telah dilakukan uji coba dengan cara mengunggah kode sumber program ke situs SPOJ, namun uji kebenaran ini perlu dilakukan untuk lebih meyakinkan status kebenaran dari kode sumber ini. Dalam hal ini akan

diselesaikan masalah penugasan dinamis dengan skala problem yang kecil. Akan diberikan matriks bobot dengan ukuran 4×4 degan bobot ditentukan secara acak dan terdapat 11 operasi dimana masing-masing jenis operasi muncul minimal satu kali. Bobot perubahan juga ditentukan secara acak. Format masukan dari masalah penugasan yang akan diselesaikan adalah seperti pada Gambar 5.2.

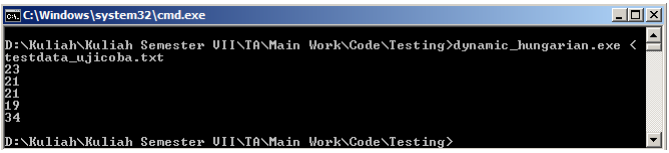
```
1.  4
2.  3 4 6 3
3.  6 7 2 4
4.  4 1 3 4
5.  6 2 4 3
6.  11
7.  Q
8.  X 1 5 3 7 3
9.  Q
10. Y 2 8 5 7 6
11. Q
12. C 3 0 2
13. Q
14. A
15. X 4 3 9 4 1 2
16. Y 4 1 4 9 3 2
17. Q
```

Gambar 5.2 Format masukan uji coba kebenaran

Data tersebut digunakan sebagai masukan dari program yang telah dibuat lalu hasil keluarannya akan dibandingkan apabila dikerjakan secara manual dengan menggunakan meotde *Hungarian*. Keluaran dari program terhadap masukan tersebut ditunjukkan pada Gambar 5.3.

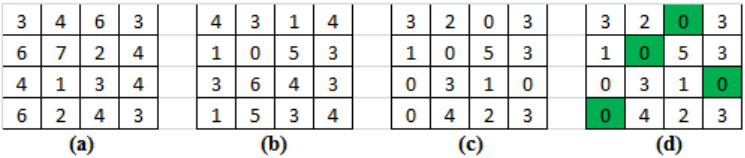
Dapat dicermati pada Gambar 5.3, masukan di atas disimpan terlebih dahulu dalam sebuah file dengan nama *testdata_ujicoba.txt*. Terdapat lima keluaran karena memang terdapat lima operasi *query* pada masukan. Setelah ini akan

ditunjukkan pengerjaan masalah penugasan dinamis dengan menggunakan metode *Hungarian*.



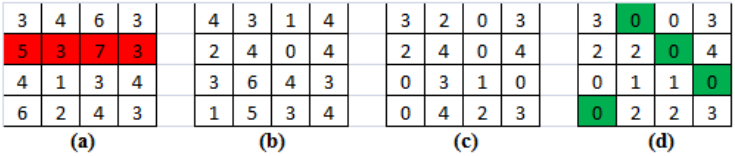
Gambar 5.3 Keluaran program terhadap masukan seperti pada Gambar 5.2

Pada mulanya terdapat matriks bobot dengan ukuran 4×4 seperti pada baris kedua sampai baris kelima pada Gambar 5.2. Karena pada baris ketujuh pada masukan merupakan operasi *query*, maka akan dicari nilai penugasan maksimal. Karena metode *Hungarian* menghasilkan nilai penugasan minimal, maka dari itu data perlu diubah sesuai dengan apa yang telah dijelaskan pada subbab 2.6. Dicari dahulu elemen pada matriks dengan nilai maksimal, lalu tiap elemen dari matriks diubah menjadi nilai maksimal yang didapat dikurangi nilai pada elemen itu sendiri. Proses ini diilustrasikan oleh Gambar 5.4(a) dan Gambar 5.4(b). Setelah itu dijalankan langkah nomor 1 dan nomor 2 seperti yang dijelaskan pada subbab 2.6. Hasil dari operasi tersebut masing-masing ditunjukkan oleh Gambar 5.4(c) dan Gambar 5.4(d). Karena pada Gambar 5.4(d) sudah dapat ditemukan susunan penugasan seperti yang dijelaskan pada langkah nomor 3 pada subbab 2.6, maka susunan optimal (dalam hal ini maksimal) sudah ditemukan. Total bobot optimal adalah $6 + 7 + 6 + 4 = 23$, hasil yang sama dengan keluaran pertama pada program. Elemen matriks pada Gambar 5.4(d) yang berwarna hijau menandakan bahwa pekerja dengan nomor baris elemen tersebut dipasangkan dengan pekerjaan nomor kolom elemen tersebut. Perlu diperhatikan dalam hal ini matriks bobot yang digunakan untuk menemukan bobot penugasan yang optimal adalah matriks pada Gambar 5.4(a).



Gambar 5.4 Proses metode *Hungarian* (a) matriks mula-mula (b) transformasi menjadi matriks minimum (c) matriks hasil pengurangan nilai minimal tiap baris dan kolom (d) susunan *optimal matching* dengan elemen warna hijau

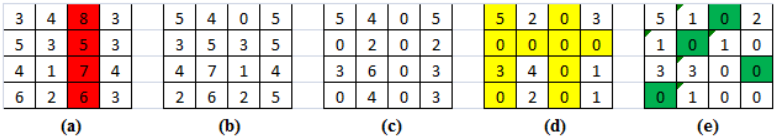
Pada baris kedelapan masukan terdapat operasi perubahan seluruh bobot pada baris kesatu. Lalu pada baris kesembilan, dilakukan lagi operasi *query*. Dilakukan lagi langkah-langkah yang telah dijelaskan pada subbab 2.6. Pada Gambar 5.5(a), baris yang berwarna merah merupakan baris yang diubah nilai bobotnya. Lalu dilakukan proses yang sama seperti sebelumnya dan didapatkan bobot penugasan optimal sebesar $6 + 4 + 7 + 4 = 21$, sama dengan keluaran kedua pada program. Proses keseluruhan dari langkah ini ditunjukkan pada Gambar 5.5.



Gambar 5.5 Proses metode *Hungarian* setelah mengalami perubahan bobot pada baris matriks (a) matriks mengalami perubahan pada baris ke-1 (b) transformasi menjadi matriks minimum (c) matriks hasil pengurangan nilai minimal tiap baris dan kolom (d) susunan *optimal matching* dengan elemen warna hijau

Pada baris kesepuluh masukan terdapat operasi perubahan seluruh bobot pada kolom kedua. Lalu pada baris kesebelas, dilakukan lagi operasi *query*. Pada Gambar 5.6(a), baris yang berwarna merah merupakan baris yang diubah nilai bobotnya.

Lalu dilakukan proses yang sama seperti sebelumnya. Namun dalam hal ini pada saat langkah ketiga tidak langsung ditemukan solusi sehingga harus melakukan langkah keempat dimana dalam hal ini garis horizontal dan vertikal yang minimal untuk melewati semua elemen yang memiliki nilai 0 ditunjukkan pada Gambar ⁋5.6(d) dengan warna kuning. Setelah itu dilakukan pengurangan dan penambahan seperti yang telah dijelaskan pada lagnkah keempat di subbab ⁋2.6. Kemudian didapatkan bobot penugasan optimal sebesar $6 + 4 + 3 + 8 = 21$, sama dengan keluaran ketiga pada program. Proses keseluruhan dari langkah ini ditunjukkan pada Gambar ⁋5.6.

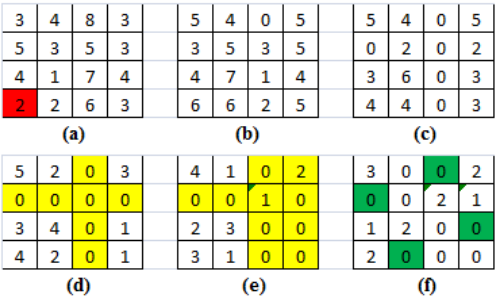


Gambar ⁋5.6 Proses metode *Hungarian* setelah mengalami perubahan bobot pada kolom matriks (a) matriks mengalami perubahan pada kolom ke-3 (b) transformasi menjadi matriks minimum (c) matriks hasil pengurangan nilai minimal tiap baris dan kolom (d) garis vertikal dan horizontal yang melewati semua elemen matriks dengan nilai 0 (e) susunan *optimal matching* dengan elemen warna hijau

Pada baris ke-12 masukan terdapat operasi perubahan sebuah elemen baris ke-3 kolom ke-0. Lalu pada baris ke-13, dilakukan lagi operasi *query*. Pada Gambar ⁋5.7(a), elemen yang berwarna merah merupakan elemen yang diubah nilai bobotnya. Lalu dilakukan proses yang sama seperti sebelumnya dan didapatkan bobot penugasan optimal sebesar $2 + 4 + 5 + 8 = 19$, sama dengan keluaran ketempat pada program. Proses keseluruhan dari langkah ini ditunjukkan pada Gambar ⁋5.7.

Pada baris ke-14 terdapat sebuah operasi untuk meningkatkan ukuran matriks menjadi 5×5. Karena baris dan kolom yang baru masih belum memiliki bobot (nilai bobot sama

dengan 0), maka pada baris ke-15 dan ke-16 terdapat operasi untuk mengubah semua nilai bobot baris dan kolom yang baru. Dilakukan lagi proses yang sama seperti sebelumnya dan didapatkan bobot penugasan optimal sebesar $9 + 3 + 9 + 8 + 5 = 34$, sama dengan keluaran kelima pada program. Proses keseluruhan dari langkah ini ditunjukkan pada Gambar 5.8.



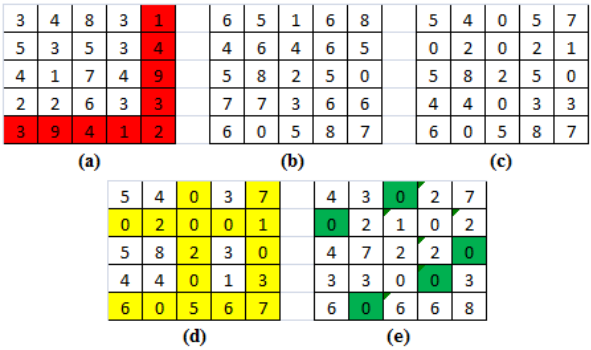
Gambar 5.7 Proses metode *Hungarian* setelah mengalami perubahan bobot pada satu elemen matriks (a) matriks mengalami perubahan pada elemen baris ke-4 kolom ke-1 (b) transformasi menjadi matriks minimum (c) matriks hasil pengurangan nilai minimal tiap baris dan kolom (d), (e) garis vertikal dan horizontal yang melewati semua elemen matriks dengan nilai 0 (f) susunan *optimal matching* dengan elemen warna hijau

Dari kelima keluaran yang dihasilkan, tidak ada perbedaan antara hasi metode *Hungarian* yang dijalankan manual dengan keluaran dari program yang telah dibuat. Hal ini membuktikan bahwa program yang telah dibuat dapat menyelesaikan masalah penugasan dinamis dengan benar.

5.2.3 Uji Coba Kinerja

Kompleksitas waktu dari algoritma *Hungarian* dinamis adalah $O(kn^2)$ setiap ada operasi *query* dimana k adalah

banyaknya perubahan dan *n* adalah banyaknya *vertex*. Dalam bab ini akan dilakukan dua macam uji coba. Pertama, banyaknya *vertex* dibuat bervariasi sedangkan banyaknya operasi dibuat tetap.



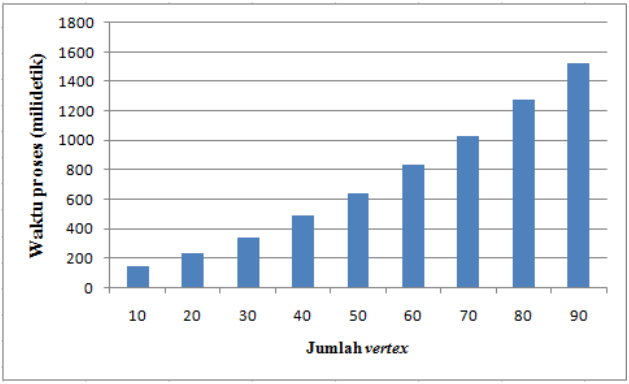
Gambar 5.8 Proses metode *Hungarian* setelah mengalami penambahan baris dan kolom (a) matriks penambahan baris dan kolom (b) transformasi menjadi matriks minimum (c) matriks hasil pengurangan nilai minimal tiap baris dan kolom (d) garis vertikal dan horizontal yang melewati semua elemen matriks dengan nilai 0 (e) susunan *optimal matching* dengan elemen warna hijau

Hal tersebut dilakukan untuk menunjukkan pengaruh banyak *vertex* terhadap waktu eksekusi program. Kedua, banyaknya operasi dibuat bervariasi dan banyaknya *vertex* dibuat tetap. Hal ini dilakukan untuk menunjukkan pengaruh banyak operasi terhadap waktu eksekusi program. Dalam hal ini banyaknya operasi tipe *query* dan *add vertex* dibuat tetap karena operasi *query* tidak menimbulkan perubahan sedangkan operasi tipe *add vertex* dibuat tetap agar tidak melebihi batasan *vertex* dimana paling banyak adalah 100 *vertex*. Implementasi program pembuatan data uji coba sesuai dengan parameter yang ditentukan dan program pengukuran waktu uji coba program hasil implementasi terhadap tiap data dapat dilihat pada Lampiran A.

5.2.3.1 Pengaruh Ukuran *Vertex* Terhadap Waktu

Tabel 5.2 Tabel pengaruh jumlah *vertex* terhadap waktu eksekusi program

Percobaan	Jumlah vertex	Waktu (milidetik)
1	10	140
2	20	234
3	30	337
4	40	486
5	50	642
6	60	834
7	70	1032
8	80	1277
9	90	1527



Gambar 5.9 Grafik pengaruh jumlah *vertex* terhadap waktu eksekusi program

Pada uji coba ini banyaknya *vertex* dibuat bervariasi antara 10 hingga 90 dengan rentang 10 *vertex*. Jumlah operasi ditetapkan sebanyak 10000 operasi dimana tiap operasi kelipatan 10 adalah operasi *query* dan tiap operasi kelipatan 999 adalah operasi *add vertex*. Dicatat waktu eksekusi program terhadap tiap

data dalam satuan milidetik agar pengaruh banyak *vertex* terhadap waktu eksekusi program dapat diamati. Hasil uji coba dari percobaan tersaji dalam Tabel 5.2 dan digambarkan dalam grafik seperti yang terlihat pada Gambar 5.9.

Seperti yang dapat dilihat pada Gambar 5.9, grafik cenderung mendekati kurva kuadratik. Hal ini sesuai dengan kompleksitas dari algoritma *Hungarian* dinamis yang dipengaruhi jumlah *vertex* secara kuadratik.

5.2.3.2 Pengaruh Ukuran Operasi Terhadap Waktu

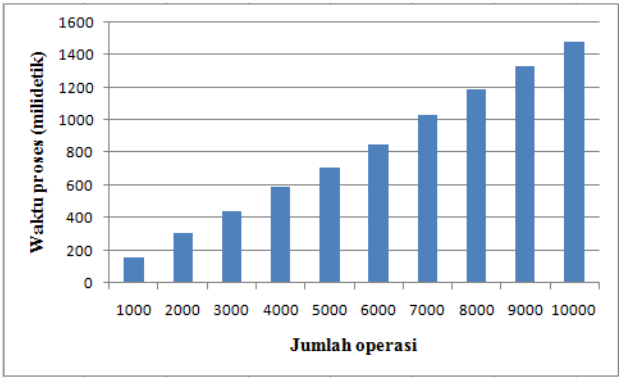
Tabel 5.3 Tabel pengaruh jumlah operasi terhadap waktu eksekusi program

Percobaan	Jumlah operasi	Waktu (milidetik)
1	1000	160
2	2000	305
3	3000	444
4	4000	588
5	5000	710
6	6000	851
7	7000	1034
8	8000	1189
9	9000	1334
10	10000	1483

Pada uji coba ini banyaknya operasi dibuat bervariasi antara 1000 hingga 10000 dengan rentang 1000 operasi. Jumlah vertex ditetapkan sebanyak 90 *vertex*. Ditentukan juga tiap operasi kelipatan 10 adalah operasi *query* dan tiap operasi kelipatan 999 adalah operasi *add vertex*. Dicatat waktu eksekusi program terhadap tiap data dalam satuan milidetik agar pengaruh banyak operasi terhadap waktu eksekusi program dapat diamati. Hasil uji coba dari percobaan tersaji dalam Tabel 5.3 dan

digambarkan dalam grafik seperti yang terlihat pada Gambar 5.10.

Seperti yang dapat dilihat pada Gambar 5.10, grafik cenderung mendekati kurva linier. Hal ini sesuai dengan kompleksitas dari algoritma *Hungarian* dinamis yang dipengaruhi jumlah operasi secara linier.



Gambar 5.10 Grafik pengaruh jumlah operasi terhadap waktu eksekusi program

BAB VI
KESIMPULAN DAN SARAN

Bab ini membahas mengenai kesimpulan yang dapat diambil dari hasil uji coba yang telah dilakukan. Selain kesimpulan, juga terdapat saran yang ditujukan untuk pengembangan program lebih lanjut.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakakan, dapat ditarik beberapa hal dari kinerja algoritma *Hungarian* dinamis dalam menyelesaikan masalah penugasan dinamis. Beberapa hal tersebut adalah sebagai berikut:

1) Algoritma *Hungarian* dinamis dapat menyelesaikan masalah penugasan dinamis dengan benar. Hal ini dapat dibuktikan secara teori maupun implementasi.
2) Kecepatan algoritma *Hungarian* dinamis dipengaruhi oleh banyaknya operasi secara linier dan dipengaruhi oleh banyaknya *vertex* secara kuadratik.

6.2 Saran

Saran yang diberikan dalam pengembangan algoritma *Hungarian* dinamis adalah agar pada penelitian selanjutnya, kompleksitas waktu dari algoritma *Hungarian* dinamis diturunkan menjadi $O(kn\ lg\ n)$ atau bahkan $O(kn)$.

DAFTAR PUSTAKA

[1] Dieter Jungnickel, *Graphs, Networks and Algorithms*, 4th ed. Berlin, Germany: Springer, 2012.

[2] J.A. Bondy and U.S.R. Murty, *Graph Theory*, 3rd ed. Berlin, Germany: Springer, 2008.

[3] G. Ayorkor Korsah, Anthony Stentz, and M Bernardine Dias, "The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs," Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-07-27, July 2007.

[4] Hamdy A. Taha, *Operations Research: An Introduction*, 8th ed. Upper Saddle River, United States of America: Prentice Hall, 2007.

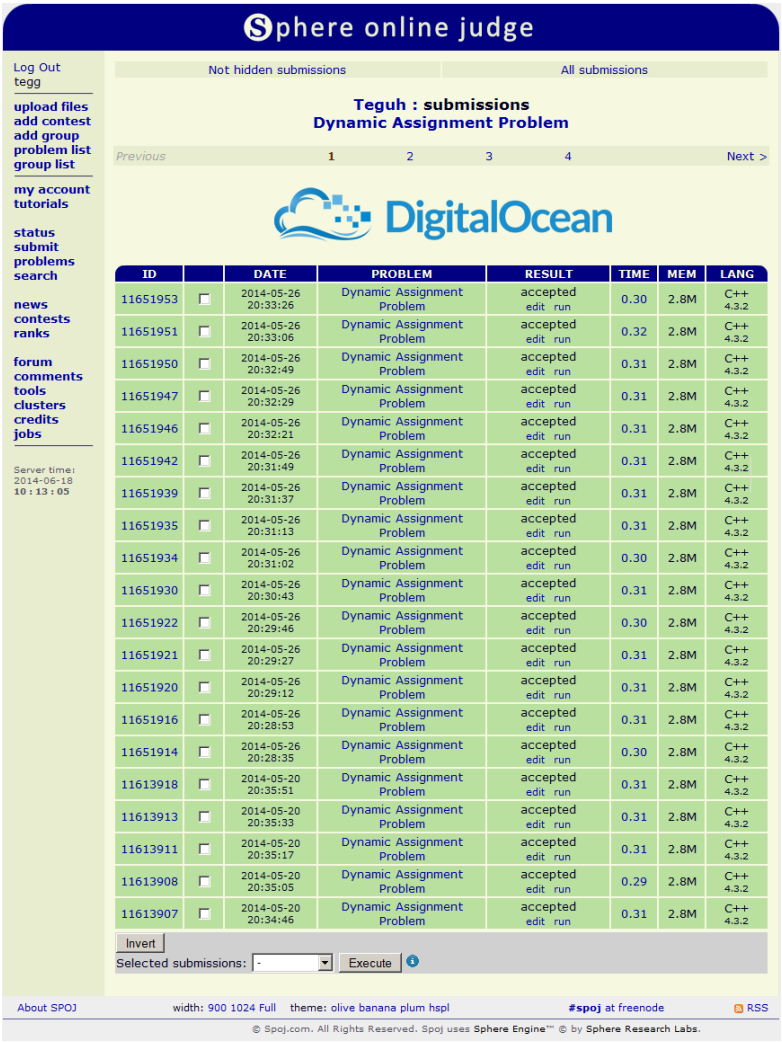
[5] SPOJ. (2012, November) Dynamic Assignment Problem. [Online]. http://www.spoj.com/problems/DAP/

BIODATA PENULIS

Teguh Suryo Santoso, anak pertama dari dua bersaudara yang lahir di Surabaya pada tanggal 12 November 1992. Penulis telah menempuh pendidikan formal mulai dari TK Hidayatul Ummah (1996-1998), SD Hidayatul Ummah (1998-2004), SMP Negeri 1 Surabaya (2004-2007), SMA Negeri 5 Surabaya (2007-2010) dan terakhir sebagai mahasiswa Jurusan Teknik Informatika Institut Teknologi Sepuluh Nopember dengan bidang minat Komputasi Cerdas dan Visualisasi (2010-2014).

Penulis memiliki ketertarikan pada dunia pemrograman sejak penulis belajar pada tingkat SMA. Lulus dari SMA penulis melanjutkan pendidikan di jurusan teknik informatika untuk mengembangkan minatnya. Semasa kuliah penulis aktif mengikuti berbagai lomba pemrograman seperti Gemastik V Kategori Pemrograman, ACM ICPC Asia Jakarta Regional Contest 2012 dan ACM ICPC Asia Jakarta Regional Contest 2013. Selain itu penulis juga pernah penulis pernah menjabat sebagai asisten mata kuliah teori graf dan algoritma dan struktur data.

LAMPIRAN A



Gambar A.1 Status pengumpulan kode sumber pada SPOJ sebanyak 20 kali

```
#include<cstdio>
#include<cstdlib>
const int QUERYINTERVAL = 10;
const int ADDINTERVAL = 999;
const int MAXCHAR = 512;

int main()
{
    int numN, numM;

    int N[] = {10, 20, 30, 40, 50, 60, 70, 80,
90};
    numM = 10000;
    system("mkdir S1");
    for(int i=0;i<9;i++)
    {
        char command[MAXCHAR];
        sprintf(command, "testdata_generator %d
%d %d %d", N[i], numM, QUERYINTERVAL,
ADDINTERVAL);
        system(command);
        system("move *.in S1");
    }

    int M[] = {1000, 2000, 3000, 4000, 5000,
6000, 7000, 8000, 9000, 10000};
    numN = 90;
    system("mkdir S2");
    for(int i=0;i<10;i++)
    {
        char command[MAXCHAR];
```

Kode Sumber A.2 Kode untuk pembuatan data uji coba sesuai dengan parameter yang ditentukan (bagian 1)

```
        sprintf(command, "testdata_generator %d
%d %d %d", numN, M[i], QUERYINTERVAL,
ADDINTERVAL);
        system(command);
        system("move *.in S2");
    }
    return 0;
}
```

Kode Sumber A.3 Kode untuk pembuatan data uji coba sesuai dengan parameter yang ditentukan (bagian 2)

```
#include<cstdio>
#include<cstring>
#include<ctime>
#include<cstdlib>
#include<dirent.h>
const int MAXCHAR = 512;

bool isInputFile(char strFile[])
{
    int len = strlen(strFile);
    if(len < 3) return false;
    if(strFile[len-3]=='.' && strFile[len-
2]=='i' && strFile[len-1]=='n') return true;
    return false;
}

int main()
{
    DIR *dir;
    struct dirent *ent;
    FILE *out;
    int numN, numM;
    out = fopen("Report.txt", "w+");
```

Kode Sumber A.4 Pengukuran waktu uji coba program hasil implementasi (bagian 1)

```
    fprintf(out, "--- Reporting Scenario1 ---
\n");
    if ((dir = opendir (".\\S1\\")) != NULL)
    {
        while ((ent = readdir (dir)) != NULL)
        {
            printf ("%s\n", ent->d_name);
            if(!isInputFile(ent->d_name))
continue;
            char command[512];
            sprintf(command, "dynamic-
hungarian < S1\\%s > a.out", ent->d_name);

            clock_t begin = clock();
            system(command);
            clock_t end = clock();
            double elapsed_secs = double(end -
begin) / CLOCKS_PER_SEC;

            fprintf(out, "%.3lf\n",
elapsed_secs);
            system("del a.out");
        }
        closedir (dir);
    }

    fprintf(out, "--- Reporting Scenario2 ---
\n");
    if ((dir = opendir (".\\S2\\")) != NULL)
    {
        while ((ent = readdir (dir)) != NULL)
        {
            printf ("%s\n", ent->d_name);
            if(!isInputFile(ent->d_name))
continue;
            char command[512];
```

Kode Sumber A.5 Pengukuran waktu uji coba program hasil implementasi (bagian 2)

```
            sprintf(command, "dynamic-hungarian
< S2\\%s > a.out", ent->d_name);

            clock_t begin = clock();
            system(command);
            clock_t end = clock();
            double elapsed_secs = double(end -
begin) / CLOCKS_PER_SEC;

            fprintf(out, "%.3lf\n",
elapsed_secs);
            system("del a.out");
        }
        closedir (dir);
    }
    fclose(out);
    return 0;
}
```

Kode Sumber A.6 Pengukuran waktu uji coba program hasil implementasi (bagian 3)