ITS
Institut
Teknologi
Sepuluh Nopember

TUGAS AKHIR - KI141502

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI CYCLE PADA TURNAMEN BERBASIS STRONGLY CONNECTED DAN VERTEX PANCYCLIC: STUDI KASUS PERSOALAN SPOJ KLASIK CYCLERUN - RIDING IN CYCLES

DANIEL BINTAR
NRP 5113 100 145

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

Dosen Pembimbing II
Abdul Munif, S.Kom., M.Sc.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2017

(Halaman ini sengaja dikosongkan)

TUGAS AKHIR - KI141502

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI CYCLE PADA TURNAMEN BERBASIS STRONGLY CONNECTED DAN VERTEX PANCYCLIC: STUDI KASUS PERSOALAN SPOJ KLASIK CYCLERUN - RIDING IN CYCLES

DANIEL BINTAR
NRP 5113 100 145

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

Dosen Pembimbing II
Abdul Munif, S.Kom., M.Sc.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2017

(Halaman ini sengaja dikosongkan)

UNDERGRADUATE THESIS - KI141502

ALGORITHM DESIGN AND ANALYSIS FOR CYCLE COMPUTATION ON TOURNAMENT WITH THE PRINCIPLE OF STRONGLY CONNECTED AND VERTEX PANCYCLIC: CASE STUDY ON PROBLEM SPOJ CLASSIC CYCLERUN - RIDING IN CYCLES

DANIEL BINTAR
NRP 5113 100 145

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

Supervisor II
Abdul Munif, S.Kom., M.Sc.

DEPARTMENT OF INFORMATICS
Faculty of Information Technology
Institut Teknologi Sepuluh Nopember
Surabaya, 2017

(Halaman ini sengaja dikosongkan)

LEMBAR PENGESAHAN

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI CYCLE PADA TURNAMEN BERBASIS STRONGLY CONNECTED DAN VERTEX PANCYCLIC: STUDI KASUS PERSOALAN SPOJ KLASIK CYCLERUN - RIDING IN CYCLES

TUGAS AKHIR

Diajukan Guna Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Algoritma dan Pemrograman
Program Studi S1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:

DANIEL BINTAR
NRP: 5113 100 145

Disetujui oleh Dosen Pembimbing Tugas Akhir

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

Abdul Munif, S.Kom., M.Sc. .....
NIP: 198608232015041004 (Pembimbing 2)

SURABAYA
JULI 2017

(Halaman ini sengaja dikosongkan)

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI CYCLE PADA TURNAMEN BERBASIS STRONGLY CONNECTED DAN VERTEX PANCYCLIC: STUDI KASUS PERSOALAN SPOJ KLASIK CYCLERUN - RIDING IN CYCLES

Nama	:	**DANIEL BINTAR**
NRP	:	**5113 100 145**
Jurusan	:	**Teknik Informatika FTIf**
Pembimbing I	:	**Rully Soelaiman, S.Kom., M.Kom.**
Pembimbing II	:	**Abdul Munif, S.Kom., M.Sc.**

Abstrak

Salah satu permasalahan pada turnamen yaitu mendeteksi apakah terdapat cycle dengan jumlah verteks t, untuk setiap t pada range $3 \leq t \leq N$, dengan N adalah total vertex yang ada, dan terdapat satu verteks yang diinginkan untuk ada pada semua cycle tersebut. Jika semua cycle yang diharapkan ada, permasalahannya lainnya adalah bagaimana mendapatkan semua cycle tersebut. Salah satu solusi untuk mendapatkan semua cycle tersebut bisa dilakukan dengan melakukan traversing ke semua kemungkinan cycle, namun pendekatan tersebut tentunya tidak efisien.

Pada Tugas Akhir ini dirancang penyelesaian permasalahan di atas dengan melihat beberapa kondisi apakah suatu turnamen merupakan strongly connected, karena jika memiliki cycle dengan memakai semua verteks pada graf, pasti strongly connected. Jika suatu graf tidak strongly connected, pasti tidak memiliki cycle dengan N verteks. Spesialnya dari turnamen adalah jika turnamen merupakan strongly connected, pasti vertex pancyclic. Vertex pancyclic berarti tiap vertex memiliki cycle dengan jumlah vertex t, untuk setiap t pada range

$3 \leq t \leq N$.

Hasil dari Tugas Akhir ini telah berhasil menyelesaikan permasalahan di atas dengan cukup efisien dengan kompleksitas waktu $O(N^2)$.
***Kata-Kunci**: Turnamen, Vertex Pancyclic, Strongly Connected*

ALGORITHM DESIGN AND ANALYSIS FOR CYCLE COMPUTATION ON TOURNAMENT WITH THE PRINCIPLE OF STRONGLY CONNECTED AND VERTEX PANCYCLIC: CASE STUDY ON PROBLEM SPOJ CLASSIC CYCLERUN - RIDING IN CYCLES

Name	:	**DANIEL BINTAR**
NRP	:	**5113 100 145**
Major	:	**Informatics FTIf**
Supervisor I	:	**Rully Soelaiman, S.Kom., M.Kom.**
Supervisor II	:	**Abdul Munif, S.Kom., M.Sc.**

Abstract

One of tournament problem is to check if there are exist cycles with the number of vertices t, for each t in range $3 \leq t \leq N$*, with N is the total vertices available, and there is one desired vertex to be exist on all cycle. If cycles are exist, another problem is how to get it. One of the solution is traversing to all cycle possibilities, but this approach is certainly not efficient.*

This Final Project is designed to solve the above problem by looking at some conditions whether a tournament is strongly connected or not, because if graph have cycle which using all vertices in the graph, it must be strongly connected. If a graph is not strongly connected, it certainly does not have cycle with N *vertices. The special of the tournament is if it is strongly connected, it definitely is vertex pancyclic. Vertex pancyclic means each vertex has cycle with the number of vertices t, for every t in range* $3 \leq t \leq N$*.*

The result of this Final Project has successfully solved the above problem quite efficiently with the time complexity of $O(N^2)$*.*

Keyword*: Tournament, Vertex Pancyclic, Strongly Connected*

(Halaman ini sengaja dikosongkan)

KATA PENGANTAR

Puji syukur penulis panjatkan kepada Allah Tuhan Yang Maha Esa, yang telah melimpahkan rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul **Desain dan Analisis Algoritma Komputasi Cycle pada Turnamen Berbasis Strongly Connected dan Vertex Pancyclic: Studi Kasus Persoalan SPOJ Klasik CYCLERUN - Riding in cycles**.

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

Penulis mengucapkan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama proses pengerjaan tugas akhir ini hingga selesai, antara lain:

1. Allah Tuhan Yang Maha Esa atas anugerah-Nya yang tidak terkira kepada penulis.
2. Orang tua, saudara serta keluarga penulis yang tiada henti-hentinya memberikan semangat, perhatian, dan doa selama perkuliahan di Jurusan Teknik Informatika ini.
3. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku dosen pembimbing I yang telah memberikan bantuan dan arahan, waktu untuk berdiskusi sehingga penulis dapat menyelesaikan tugas akhir dan perkuliahan di Kampus ini.
4. Bapak Abdul Munif, S.Kom., M.Sc. selaku dosen pembimbing II yang telah banyak memberikan bimbingan dan arahan terutama pembuatan skripsi dalam pengerjaan tugas akhir ini.
5. Teman-teman Kontrakan Berprestasi, Daniel Fablius, Dewangga, Donny, Irsyad, Juan, Wicak, dan Luthfie yang telah membantu penulis dalam menyelesaikan permasalahan dalam mengerjakan tugas akhir, yang selalu menghibur, memotivasi, dan mendukung penulis untuk menyelesaikan tugas akhir ini.

6. Bapak Radityo Anggoro, S.Kom., M.Sc., selaku koordinator TA, dan segenap dosen Teknik Informatika yang telah memberikan ilmu dan pengalamannya.
7. Serta semua pihak yang telah turut membantu penulis dalam menyelesaikan Tugas Akhir ini.

Penulis menyadari bahwa Tugas Akhir ini masih memiliki banyak kekurangan. Sehingga dengan kerendahan hati, penulis mengharapkan kritik dan saran dari pembaca untuk perbaikan ke depannya.

Surabaya, Juli 2017

Daniel Bintar

DAFTAR ISI

(Halaman ini sengaja dikosongkan)

DAFTAR TABEL

(Halaman ini sengaja dikosongkan)

DAFTAR GAMBAR

DAFTAR KODE SUMBER

(Halaman ini sengaja dikosongkan)

BAB I

PENDAHULUAN

Pada bab ini akan dipaparkan mengenai garis besar Tugas Akhir yang meliputi latar belakang, tujuan, rumusan dan batasan permasalahan, metodologi pembuatan Tugas Akhir, dan sistematika penulisan.

1.1 Latar Belakang

Teori graf adalah salah satu cabang ilmu yang sudah ada sejak lama dan banyak sekali diterapkan untuk membantu pemecahan masalah terutama di bidang ilmu pengetahuan dan teknologi informasi. Banyak permasalahan di dunia nyata seperti komputasi jalur terpendek atau informasi yang terkandung pada hubungan pertemanan di media sosial yang dapat dimodelkan dengan bantuan teori graf untuk selanjutnya dilakukan komputasi dan didapatkan hasil yang diinginkan. Dalam beberapa dekade terakhir tidak sedikit algoritma dan struktur data yang diciptakan untuk mengatasi permasalahan graf.

Turnamen adalah *directed graph* yang tiap pasangan verteks terhubung oleh satu *directed edge*. Jika verteks a bisa langsung menuju verteks b, maka verteks b tidak bisa langsung menuju ke verteks a. Jika verteks a tidak bisa langsung menuju verteks b, maka verteks b bisa langsung menuju verteks a.

Permasalahan yang diambil pada Tugas Akhir ini diambil dari persoalan SPOJ CYCLERUN - Riding in Cycles[1]. Permasalahannya yaitu dari sebuah turnamen yang diberikan, dapatkan *cycle* dengan jumlah verteks t, untuk setiap t pada *range* $3 \leq t \leq N$, dengan N adalah total vertex yang ada, dan terdapat satu verteks yang diinginkan untuk ada pada semua *cycle* tersebut. Solusi naif untuk mendapatkan semua *cycle* yang diharapkan adalah dengan melakukan *traversing* dari satu verteks ke semua verteks lainnya sampai membentuk *cycle* yang diharapkan. Kompleksitas waktu eksekusi yang dibutuhkan

adalah $O(N^3)$. Padahal kompleksitas waktu eksekusi yang diharapkan adalah $O(N^2)$. Oleh karena itu, dibutuhkan desain algoritma yang efisien dalam kecepatan untuk permasalahan ini.

Hasil dari Tugas Akhir ini diharapkan dapat menentukan implementasi algoritma yang tepat untuk memecahkan permasalahan di atas secara optimal dan dapat memberikan kontribusi terhadap perkembangan pengetahuan teknologi informasi.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam tugas akhir ini adalah sebagai berikut:

1. Bagaimana menganalisis dan menentukan algoritma untuk menyelesaikan permasalahan mendeteksi apakah suatu turnamen memiliki semua *cycle* dengan jumlah verteks t, untuk setiap t pada *range* $3 \leq t \leq N$, dengan N adalah total verteks yang ada, dan terdapat satu verteks yang diinginkan untuk ada pada semua *cycle* tersebut pada turnamen dengan optimal?
2. Bagaimana menganalisis dan menentukan algoritma untuk menyelesaikan permasalahan mendapatkan semua *cycle* dengan jumlah verteks t, untuk setiap t pada *range* $3 \leq t \leq N$, dengan N adalah total verteks yang ada, dan terdapat satu verteks yang diinginkan untuk ada pada semua *cycle* tersebut pada turnamen dengan optimal?

1.3 Batasan Masalah

Batasan masalah diambil dari SPOJ CYCLERUN - Riding in Cycles[1], yaitu:

1. Batasan waktu untuk satu pencarian 0,1 detik.
2. Batasan besar *file source code* 50000B.

3. Batasan besar memori 1536MB.

1.4 Tujuan

Tujuan dari pengerjaan Tugas Akhir ini adalah:

1. Melakukan analisis dan mendesain algoritma mengenai *strongly connected tournament* untuk menyelesaikan permasalahan SPOJ CYCLERUN - Riding in Cycles.
2. Melakukan implementasi algoritma mengenai *strongly connected tournament* untuk menyelesaikan permasalahan SPOJ CYCLERUN - Riding in Cycles.
3. Mengevaluasi algoritma yang didesain untuk menyelesaikan permasalahan SPOJ CYCLERUN - Riding in Cycles.

1.5 Manfaat

Manfaat dari pengerjaan Tugas Akhir ini adalah mendapatkan algoritma yang optimal untuk menyelesaikan permasalahan komputasi *cycle* pada turnamen.

1.6 Metodologi

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

1. Penyusunan proposal Tugas Akhir
   Pada tahap ini dilakukan penyusunan proposal tugas akhir yang berisi permasalahan dan gagasan solusi yang akan diteliti pada permasalahan SPOJ CYCLERUN - Riding in Cycles.
2. Studi literatur
   Pada tahap ini dilakukan pencarian informasi dan studi literatur mengenai pengetahuan atau metode yang dapat

digunakan dalam penyelesaian masalah. Informasi didapatkan dari materi-materi yang berhubungan dengan algoritma dan struktur data yang digunakan untuk penyelesaian permasalahan ini, materi-materi tersebut didapatkan dari buku, jurnal, maupun internet.

3. Desain
   Pada tahap ini dilakukan desain rancangan algoritma yang digunakan dalam solusi untuk pemecahan permasalahan SPOJ CYCLERUN - Riding in Cycles.
4. Implementasi perangkat lunak
   Pada tahap ini dilakukan implementasi atau realiasi dari rancangan desain algoritma yang telah dibangun pada tahap desain ke dalam bentuk program.
5. Uji coba dan evaluasi
   Pada tahap ini dilakukan uji coba kebenaran implementasi dan uji coba generalisasi. Pengujian kebenaran dilakukan pada sistem penilaian daring SPOJ sesuai dengan masalah yang dikerjakan untuk diuji apakah luaran dari program telah sesuai. Uji coba generalisasi digunakan untuk menguji struktur data generalisasi pada variasi permasalahan lain.
6. Penyusunan buku Tugas Akhir
   Pada tahap ini dilakukan penyusunan buku Tugas Akhir yang berisi dokumentasi hasil pengerjaan Tugas Akhir.

1.7 Sistematika Penulisan

Berikut adalah sistematika penulisan buku Tugas Akhir ini:

1. BAB I: PENDAHULUAN
   Bab ini berisi latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi dan sistematika penulisan Tugas Akhir.
2. BAB II: DASAR TEORI

Bab ini berisi dasar teori mengenai permasalahan dan algoritma penyelesaian yang digunakan dalam Tugas Akhir.

3. BAB III: DESAIN
   Bab ini berisi desain algoritma dan struktur data yang digunakan dalam penyelesaian permasalahan.
4. BAB IV: IMPLEMENTASI
   Bab ini berisi implementasi berdasarkan desain algortima yang telah dilakukan pada tahap desain.
5. BAB V: UJI COBA DAN EVALUASI
   Bab ini berisi uji coba dan evaluasi dari hasil implementasi yang telah dilakukan pada tahap implementasi.
6. BAB VI: KESIMPULAN
   Bab ini berisi kesimpulan yang didapat dari hasil uji coba yang telah dilakukan.

(Halaman ini sengaja dikosongkan)

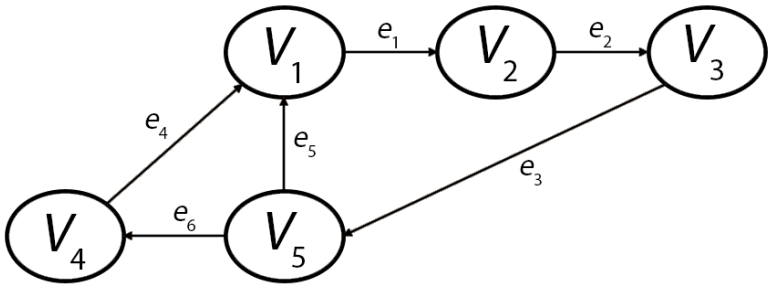
BAB II

LANDASAN TEORI

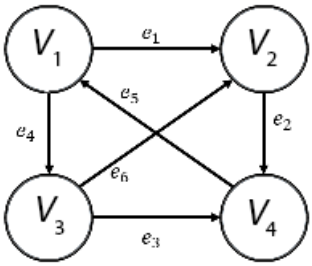
2.1 Graf

Suatu graf merupakan sekumpulan titik pada bidang datar, disebut juga sebagai verteks, dan beberapa diantaranya dihubungkan oleh segmen garis, atau yang disebut juga *edge*. Secara formal, suatu graf G = (V,E), didefinisikan sebagai sekumpulan pasangan 2 buah himpunan, yaitu himpunan tidak kosong V yang berisi kumpulan verteks, dan himpunan E yang berisi sekumpulan *edge*. Didefinisikan pula, suatu *edge* (u,v) merupakan penghubung antara verteks u dan v.

Jika nilai dari *edge* (u,v) sama dengan nilai *edge* (v,u), maka *edge* tersebut berarti tidak berarah, atau disebut juga dengan *undirected* edge, begitupun sebaliknya, apabila tidak sama, maka disebut dengan *directed edge* atau *edge* berarah. Suatu graf pasti berisi *edge* yang sejenis, yaitu berarah atau atau tidak berarah saja. Graf yang terdiri atas *directed edge*, disebut dengan *directed graph* atau graf berarah. Sedangkan graf yang terdiri atas *undirected edge*, disebut dengan *undirected graph* atau graf tidak berarah[2].



Gambar 2.1 *Directed Graph*



Gambar 2.2 *Strongly Connected Graph*

2.2 Turnamen

Turnamen adalah *directed graph* yang tiap pasangan verteks terhubung oleh satu *directed edge*[3]. Jika verteks a bisa langsung menuju verteks b, maka verteks b tidak bisa langsung menuju ke verteks a. Jika verteks a tidak bisa langsung menuju verteks b, maka verteks b bisa langsung menuju verteks a.

2.3 *Strongly Connected Graph*

Graf yang *strongly connected* adalah graf yang tiap verteksnya bisa dicapai oleh semua verteks lainnya, walaupun untuk mencapai verteks lain melewati *edge* lebih dari 1.

Contoh *strongly connected graph* diilustrasikan pada Gambar 2.2. Tiap verteks pada Gambar 2.2 bisa mencapai semua verteks lainnya, yaitu:

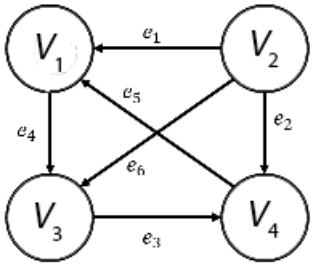
1. Verteks V_1 bisa mencapai verteks V_2 melalui e_1.
2. Verteks V_1 bisa mencapai verteks V_3 melalui e_4.
3. Verteks V_1 bisa mencapai verteks V_4 melalui e_1 dan e_4 .
4. Verteks V_2 bisa mencapai verteks V_1 melalui e_2 dan e_5.
5. Verteks V_2 bisa mencapai verteks V_3 melalui e_2, e_5 dan e_4.
6. Verteks V_2 bisa mencapai verteks V_4 melalui e_2.

7. Verteks V_3 bisa mencapai verteks V_1 melalui e_3 dan e_5.
8. Verteks V_3 bisa mencapai verteks V_2 melalui e_6.
9. Verteks V_3 bisa mencapai verteks V_4 melalui e_3.
10. Verteks V_4 bisa mencapai verteks V_1 melalui e_5.
11. Verteks V_4 bisa mencapai verteks V_2 melalui e_5 dan e_1.
12. Verteks V_4 bisa mencapai verteks V_3 melalui e_5 dan e_4.

2.4 *Strongly Connected Component*

Strongly connected component atau bisa disingkat *SCC* adalah *subgraph* terbesar yang *strongly connected* yang berarti tiap verteks pada *subgraph* bisa mencapai semua verteks lainnya pada *subgraph* tersebut. Jika sebuah graf merupakan *strongly connected* maka hanya memiliki satu *strongly connected component*.

Contoh graf diilustrasikan pada Gambar 2.3. *Subgraph* dengan verteks V_1, V_2, dan V_3 bisa mengunjungi verteks antara satu dengan lainnya pada *subgraph* tersebut, tapi verteks V_2 hanya bisa menuju verteks lain dan tidak bisa dituju oleh verteks lain, maka terdapat dua *strongly connected component* yaitu $\{V_1, V_3, V_4\}$ dan $\{V_2\}$.



Gambar 2.3 Graf

2.5 Struktur Data *List*

List adalah struktur data yang menyimpan data secara berurutan yang memungkinkan penyisipan waktu konstan dan menghapus operasi di manapun dalam urutan, dan iterasi di kedua arah.

Dibandingkan dengan struktur data berurutan standar dasar lainnya (*array*, vektor dan *deque*), *list* tampil secara umum lebih baik dalam memasukkan, mengeluarkan dan memindahkan elemen dalam posisi apapun di dalamnya yang telah diperoleh iterator, dan oleh karena itu juga algoritma yang menggunakan penggunaan secara intensif[4].

Pada *list* C++, terdapat beberapa fungsi yang tersedia pada pustakanya. Beberapa yang dibutuhkan untuk Tugas Akhir ini adalah:

1. *begin*(): fungsi untuk mendapatkan elemen paling depan pada *list*.
2. *end*(): fungsi untuk mendapatkan akhir dari *list*.
3. *empty*(): fungsi untuk mengecek apakah ada element pada *list*.
4. *front*(): fungsi untuk mendapatkan isi dari elemen paling depan pada *list*.
5. *push_front*(X): fungsi untuk memasukkan elemen X ke dalam *list* untuk berada di urutan paling depan.
6. *insert*(X,Y): fungsi untuk memasukkan elemen X ke dalam *list* untuk berada pada sebelum elemen Y.
7. *erase*(X): fungsi untuk mengeluarkan elemen X dari *list*.

2.6 Struktur Data *Stack*

Stack adalah jenis struktur data, yang dirancang khusus untuk beroperasi dalam konteks *LIFO* (*Last In First Out*, di mana elemen disisipkan dan diambil hanya dari satu ujung *stack*[5].

Pada *Stack* C++, terdapat beberapa fungsi yang tersedia pada pustakanya. Beberapa yang dibutuhkan untuk Tugas Akhir ini adalah:

1. *empty*(): fungsi untuk mengecek apakah ada element pada *stack*.
2. *top*(): fungsi untuk mendapatkan elemen yang terakhir dimasukkan pada *stack*.
3. *push*(X): fungsi untuk memasukkan elemen X ke dalam *stack*.
4. *pop*(): fungsi untuk mengeluarkan elemen yang terakhir dimasukkan pada *stack*.

2.7 Algoritma Kosaraju

Algoritma Kosaraju adalah algoritma berbasis *depth-first search* atau disingkat *DFS* yang bisa digunakan untuk mendapatkan *strongly connected component* pada sebuah *directed graph*. Ide dasar dari algoritma ini adalah menjalankan *DFS* dua kali[6]. *DFS* pertama dilakukan pada graf awal dan simpan *traversal post-order* dari tiap verteks. *DFS* kedua dilakukan pada transpos dari graf awal menggunakan urutan *post-order* yang ditemukan pada *DFS* pertama. Karena melakukan *DFS* dua kali, maka kompleksitas waktu eksekusi algoritma ini adalah $O(N^2)$.

2.8 *Vertex Pancyclic*

Vertex pancyclic adalah tiap verteks pada graf yang bisa membentuk *cycle* dengan panjang t, untuk semua t dalam *range* $3 \leq t \leq N$.

Contoh: graf pada Gambar 2.1 merupakan *vertex pancyclic*.

1. Verteks V_1 bisa membentuk *cycle* yaitu:

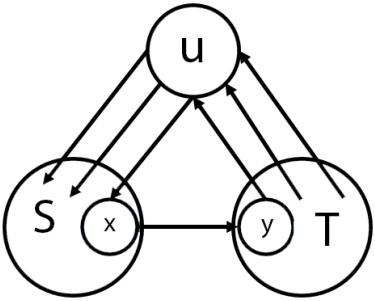
   (a) (V_1, V_2, V_4, V_1)
   (b) $(V_1, V_3, V_2, V_4, V_1)$
2. Verteks V_2 bisa membentuk *cycle* yaitu:
   (a) (V_2, V_4, V_1, V_2)
   (b) $(V_2, V_4, V1, V_3, V_2)$
3. Verteks V_3 bisa membentuk *cycle* yaitu:
   (a) (V_3, V_4, V_1, V_3)
   (b) $(V_3, V_2, V_4, V_1, V_3)$
4. Verteks V_2 bisa membentuk *cycle* yaitu:
   (a) (V_4, V_1, V_2, V_4)
   (b) $(V_4, V_1, V_3, V_2, V_4)$

2.9 *Strongly Connected Tournament*

Strongly connected tournament adalah turnamen yang *strongly connected*. Setiap *strongly connected tournament* yang memiliki jumlah verteks $N(\geq 3)$ pasti *vertex pancyclic*[7].

2.9.1 Tiap Verteks pada *Strongly Connected Tournament* memiliki *directed 3-cycle*

Untuk membuktikan bahwa tiap verteks pada *strongly connected tournament* memiliki *directed 3-cycle*, kita perlu mendefinisikan hal-hal sebagai berikut:
$G =$ *strongly connected tournament*
$u =$ salah satu verteks pada G
$k = 3$
$S = N_G^+(u)$
$T = N_G^-(u)$
Keterangan:
1. k adalah jumlah awal verteks pada *cycle*
2. S adalah himpunan verteks pada *G* yang bisa dituju secara langsung oleh verteks u



Gambar 2.4 Ilustrasi Teorem 2.9.1

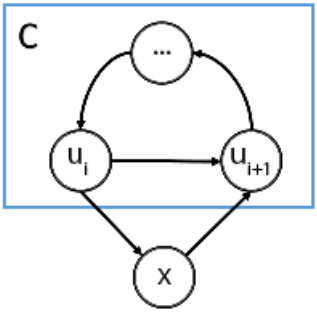
3. T adalah himpunan verteks pada *G* yang bisa menuju secara langsung ke verteks u

Karena *strongly connected*, maka $S \neq \emptyset$ dan $T \neq \emptyset$.
Karena turnamen, maka $T \cup \{u\} = S$.
$(S, T) = (S, S) \neq \emptyset$
Terdapat $x \in S$ dan $y \in T$ yang $(x, y) \in E(G)$, dan oleh karena itu, (u,x,y,u) adalah *directed 3-cycle*.

2.9.2 Jika u terdapat pada *directed cycle* dengan panjang t, $3 \leq t < N$, maka u terdapat pada *directed* $(t+1)$*-cycle*

Untuk membuktikan bahwa u terdapat pada *directed* $(t + 1)$-*cycle*, kita perlu mendefinisikan C sebagai directed t-cycle yang belum terdapat verteks u didalamnya. $C = (u_0, u_1, ..., u_{t-1}, u_o)$
Jika terdapat $x \in V(G) \setminus V(C)$ dan $N_G^+(x) \cap V(C) \neq \emptyset$ dan $N_G^-(x) \cap V(C) \neq \emptyset$, maka terdapat verteks u_i dan u_{i+1} pada C yang mana $(u_i, x), (x, u_{i+1}) \in E(G)$.
Maka u terdapat pada *directed (t+1)-cycle* $(u_0, x, u_1, ..., u_{t-1}, u_0)$.

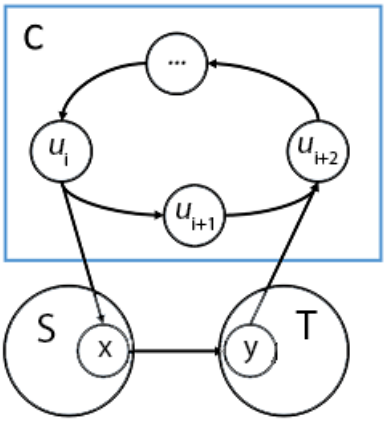
Jika tidak ada,
$S = \{x \in V(G) \setminus V(C) : N_G^+(x) \cap V(C) = \emptyset\}$



Gambar 2.5 Ilustrasi Teorem 2.9.2

$T = \{y \in V(G) \setminus V(C) : N_G^-(x) \cap V(C) = \emptyset\}$
Karena $S \neq \emptyset$ dan $T \neq \emptyset$, maka $(S, T) \neq \emptyset$
Terdapat $x \in S$ dan $y \in T$ yang mana $(x, y) \in E(G)$.
Maka u terdapat pada *directed (t+1)-cycle* $(u_0, x, y, u_2, ..., u_{n-1}, u_0)$.
Keterangan:

1. C adalah *cycle* dengan t verteks
2. $x \in V(G) \setminus V(C)$ adalah verteks pada *G* yang berada di luar C.
3. $N_G^+(x) \cap V(C) \neq \emptyset$ berarti terdapat verteks pada C yang bisa dituju secara langsung oleh verteks X.
4. $N_G^-(x) \cap V(C) \neq \emptyset$ berarti terdapat verteks pada C yang bisa menuju secara langsung ke verteks X. $(u_i, x), (x, u_{i+1}) \in E(G)$ berarti u_i bisa menuju secara langsung x, x bisa menuju secara langsung $u_i + 1$
5. S adalah himpunan verteks yang bisa dituju secara langsung oleh verteks pada C.
6. T adalah himpunan verteks yang bisa menuju secara langsung ke verteks pada C.
7. $(x, y) \in E(G)$ berarti verteks x bisa menuju secara langsung verteks y.



Gambar 2.6 Ilustrasi Teorem 2.9.2

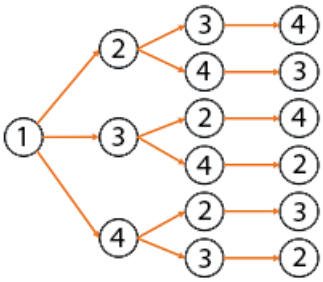
2.10 Permasalahan Riding in Cycles pada SPOJ

Pada situs penilaian daring SPOJ terdapat permasalahan komputasi *cycle* pada turnamen dengan judul soal Riding in Cycles dan kode soal CYCLERUN[1].

Deskripsi singkat dari persoalan tersebut ialah diberikan sebuah turnamen yang terdiri dari N buah verteks dan $N \times N$ matriks yang berisikan arah dari *edge*. Jika pada baris 1 kolom 2 berisi '0', berarti verteks pertama tidak bisa langsung menuju verteks kedua. Jika berisi '1', artinya bisa menuju verteks tersebut. Dipastikan bahwa jika dari kota pertama bisa langsung menuju kedua kedua, maka kota kedua tidak bisa langsung menuju kota pertama, begitu juga sebaliknya.

Berikut merupakan format masukan yang diminta dari permasalahan adalah:

1. Baris pertama terdiri dari sebuah bilangan *integer* N yang merepresentasikan banyaknya verteks.
2. N baris berikutnya, berisi N *character* yang



Gambar 2.7 Ilustrasi Solusi Naif

merepresentasikan *edge*.

Format keluaran dari permasalahan tersebut adalah sebuah string "impossible" jika tidak ada *cycle* yang diharapkan. Jika semua *cycle* yang diharapkan ada, tampilkan isi *cycle* saat verteks tiga sampai N, dimulai dengan 1 dan diakhiri 1, pada baris yang berbeda.

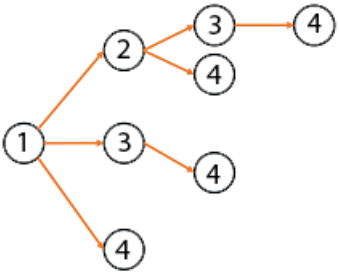
Batasan permasalahan yang diberikan adalah sebagai berikut:

1. $3 \leq N \leq 1000$

2.11 Strategi Penyelesaian Permasalahan

Solusi naif untuk mendapatkan semua *cycle* yang diharapkan adalah dengan melakukan *traversing* dari satu verteks ke semua verteks lainnya sampai membentuk *cycle* yang diharapkan. *Traversing* dimulai dari verteks yang diinginkan untuk berada di semua *cycle*, berpindah ke semua verteks lainnya, kemudian berpindah lagi ke semua verteks lainnya, sehingga membentuk semua kemungkinan *cycle*, diilustrasikan pada Gambar 2.7.

Karena graf berupa turnamen, maka tiap jalan antar kota hanya 1 arah. Contoh jika dari kota kedua bisa ke kota ketiga, maka dari kota ketiga tidak bisa ke kota kedua. Sehingga *traversing* menjadi seperti pada Gambar 2.8. Performa metode



Gambar 2.8 Ilustrasi Solusi Naif

naif ini membutuhkan waktu eksekusi dengan kompleksitas $O(N^3)$. Padahal waktu eksekusi kompleksitas yang diharapkan adalah $O(N^2)$. Sehingga metode naif ini tidak bisa menyelesaikan permasalahan.

Solusi yang ditawarkan adalah dengan memanfaatkan sifat dari *strongly connected tournament*. Karena salah satu *cycle* yang dicari adalah melewati semua verteks, maka graf pasti merupakan *strongly connected*. Jika turnamen merupakan *strongly connected*, maka semua *cycle* yang diharapkan pasti ada[7].

Jika diambil sebuah *cycle*, verteks-verteks di luar *cycle* berada pada kategori:

1. Verteks yang bisa dituju oleh *cycle*, tapi tidak bisa menuju *cycle*
   Semua verteks di dalam *cycle* menuju verteks ini. Supaya verteks ini bisa masuk ke dalam *cycle*, harus mencari verteks yang bisa menuju *cycle*, dan dituju oleh verteks ini.
2. Verteks yang bisa menuju *cycle*, tapi tidak bisa dituju oleh *cycle*
   Semua verteks di dalam *cycle* dituju oleh verteks ini. Supaya verteks ini bisa masuk ke dalam *cycle*, harus mencari verteks yang bisa dituju *cycle*, dan menuju verteks

ini.

3. Verteks yang bisa menuju dan dituju oleh *cycle*
   Ada verteks di dalam *cycle* yang menuju verteks ini, dan ada verteks di dalam *cycle* yang dituju oleh verteks ini.

Saat memasukkan verteks dari kategori 1 dan 2, karena bisa menuju semua verteks dan dituju semua verteks pada *cycle*, bisa masuk di bagian manapun.Setelah memasukkan verteks tersebut, verteks setelah verteks kategori 2 yang baru masuk pada *cycle* bisa dikeluarkan dari *cycle*. Sehingga, jumlah verteks pada *cycle* akan selalu bertambah satu, dari 3 sampai N. Dari pendekatan metode di atas, operasi-operasi yang akan dilakukan adalah:

1. Menentukan apakah turnamen *strongly connected* atau tidak menggunakan Algoritma Kosaraju.
2. Jika tidak, tampilkan "impossible" dan program selesai.
3. Mendapatkan *cycle* dengan jumlah verteks tiga.
4. Mengelompokkan verteks-verteks di luar *cycle* sesuai kategorinya.
5. Jika ada verteks pada kategori 3, ambil satu, cari verteks di dalam *cycle* yang menuju verteks ini, dan verteks setelahnya dituju oleh verteks ini, kemudian verteks ini dimasukkan diantara dua verteks itu.
6. Setelah memasukkan verteks, melakukan pengecekan apakah ada verteks pada kategori 1 dan 2 bisa berubah menjadi kategori 3.
7. Jika masih ada verteks pada kategori 3, kembali ke langkah 4.
8. Jika masih ada verteks di luar *cycle*, keluarkan satu verteks dari *cycle*
9. Cari verteks pada kategori 1 yang bisa menuju verteks kategori 2 yang bisa dituju oleh verteks tersebut, kemudian kedua verteks tersebut dimasukkan ke dalam *cycle*.
10. Kembali ke langkah 6.

Langkah memilih verteks pada kategori 1 dan 2 terlihat

memiliki kompleksitas waktu eksekusi $O(N^2)$ dan karena di dalam iterasi memasukkan *cycle* sampai berjumlah N verteks, menjadi $O(N^3)$. Tapi tidak demikian yang terjadi, karena contoh jika semua verteks pada kategori 1 kecuali 2 verteks terakhir tidak bisa menuju verteks kategori 2 tapi menuju verteks lainnya pada kategori 1, maka saat verteks terakhir masuk ke dalam *cycle* verteks-verteks akan berubah menjadi kategori 3. Maka program ini memiliki kompleksitas waktu eksekusi yang dibutuhkan $O(N^2)$. Kompleksitas dari pendekatan penyelesaian ini sudah memiliki ekspektasi waktu yang mampu menyelesaikan permasalahan ini.

(Halaman ini sengaja dikosongkan)

BAB III

DESAIN DAN PERANCANGAN

Pada bab ini dibahas mengenai desain algoritma untuk menyelesaikan permasalahan SPOJ CYCLERUN - Riding in Cycles.

3.1 Desain Fungsi Naif *Generate3VertexCycles*

Fungsi *Generate3VertexCycles* adalah fungsi pada program naif yang akan membuat semua *cycle* yang berisi tiga verteks, dengan cara mencari vertex a dan b, dimana a bisa langsung dituju verteks yang diinginkan, dan bisa langsung menuju verteks b yang bisa langsung menuju verteks yang diinginkan.

```
Generate3VertexCycles()
 1  for i = 1 to N − 1
 2      if edge[importantVertex][i] = 0
 3          continue
 4      for j = 1 to N − 1
 5          if edge[i][j] = 1 AND
    edge[j][importantVertex] = 1
 6              cycle.push_front(importantVertex)
 7              cycle.push_front(i)
 8              cycle.push_front(j)
 9              flag[importantVertex] ← true
10              flag[i] ← true
11              flag[j] ← true
12              cycles[3].push_back(cycle, flag);
```

Gambar 3.1 *Pseudocode* Fungsi *Generate3VertexCycles*

3.2 Desain Fungsi Naif *ExpandCycles*

Fungsi *ExpandCycles* adalah fungsi pada program naif yang akan menambahkan 1 verteks pada semua *t-vertex cycle* dengan mencari verteks di luar *cycle*, yang bisa dimasukkan diantara 2 verteks pada *cycle*. Fungsi ini akan mengembalikan nilai apakah ada *cycle* dengan banyak verteks $t + 1$.

```
ExpandCycles(t)
 1  flag ← false
 2  for i = 1 to cycles[t].size
 3      cycle ← cycles[y][i].first
 4      visited ← cycles[y][i].second
 5      for j = 1 to N − 1
 6          if visited[j]
 7              continue
 8          for k = 1 to cycle.size
 9              if edge[cycle[k − 1][j] = 1 AND
    edge[j][cycle[k]] = 1
10                  flag ← true
11                  newFlag ← visited
12                  newFlag[j] ← true
13                  newCycle.push_back(cycle[0 − k])
14                  newCycle.push_back(j)
15                  newCycle.push_back(cycle[k−
    cycle.size])
16                  cycles[t + 1].push_back(newCycle,
    newFlag);
17      return flag
```

Gambar 3.2 *Pseudocode* Fungsi *ExpandCycles*

3.3 Deskripsi Umum Sistem Program Naif

Pada subbab ini akan dijelaskan mengenai gambaran secara umum dari algoritma yang dirancang pada fungsi main pada program naif.

Sistem akan menerima masukan jumlah verteks, dan *edges* sebagai hubungan antar verteks.

Setelah mendapat masukan nilai dari *edge*, program akan membuat semua *cycle* dengan 3 verteks. Jika tidak ada *cycle* dengan 3 verteks, kita tampilkan "impossible" dan program selesai. Jika ada, kita iterasi dari 4 sampai N, tambahkan 1 verteks pada semua *cycle*. Jika terdapat *cycle* yang tidak ada, tampilkan "impossible" dan program selesai. Jika semua *cycle* ada, mulai iterasi dari 3 sampai N, dan tampilkan verteks pada *cycle* dengan banyak verteks sesuai urutan iterasi.

```
main()
 1  edge[0..N − 1] ← ReadInput()
 2  Generate3VertexCycles()
 3  if cycles[3].empty
 4       print "impossible"
 5       return
 6  for i = 3 to N − 1
 7       flag ← ExpandCycle()
 8       if !flag
 9            print "impossible"
10            return
11  for i = 3 to N
12       Print(i)
```

Gambar 3.3 *Pseudocode* Fungsi *Main* Naif

3.4 Desain Fungsi *IsStronglyConnected*

Fungsi *IsStronglyConnected* akan mengecek apakah graf hanya memiliki 1 *SCC*. Program akan menghitung jumlah *SCC* berdasarkan Algoritma Kosaraju dengan melakukan 2 kali *complete traversing*. Pada awalnya, tiap verteks diberikan tanda bahwa belum pernah dikunjungi. Kemudian, setiap verteks yang belum pernah dikunjungi akan melakukan *FirstDFS*, yaitu menandai bahwa verteks telah dikunjungi, tiap verteks yang bisa dikunjungi lakukan *FirstDFS*, setelah itu masukkan verteks ke dalam *stack*. Setelah itu, tiap verteks pada *stack* yang statusnya pernah dikunjungi merupakan *SCC* baru. Verteks tersebut akan memanggil *SecondDFS*, yaitu menandai bahwa verteks belum pernah dikunjungi, dan verteks-verteks yang bisa langsung menuju verteks tersebut, akan memanggil *SecondDFS*. Terakhir, program akan mengembalikan nilai apakah jumlah *SCC* 1 atau tidak.

```
FirstDFS(X)
1  visited[X] ← true
2  for i = 0 to N − 1
3      if !visited[i] AND edge[X][i] = 1
4          FirstDFS(i)
5          stackVertex.push(X)
```

Gambar 3.4 *Pseudocode* Fungsi *FirstDFS*

```
SecondDFS(X)
1  visited[X] ← false
2  for i = 0 to N − 1
3       if visited[i] AND edge[i][X] = 1
4            SecondDFS(i)
```

Gambar 3.5 *Pseudocode* Fungsi *SecondDFS*

```
IsStronglyConnected()
 1  scc ← 0
 2  for i = 0 to N − 1
 3       if !visited[i]
 4            FirstDFS(i)
 5  while !stackVertex.empty()
 6       top ← stackVertex.top()
 7       stackVertex.pop()
 8       if visited[top]
 9            scc ← scc + 1
10            SecondDFS(top)
11  return scc = 1
```

Gambar 3.6 *Pseudocode* Fungsi *IsStronglyConnected*

3.5 Desain Fungsi *Generate3VertexCycle*

Fungsi *Generate3VertexCycle* akan membuat *cycle* yang berisi tiga verteks, dengan cara mencari vertex a dan b, dimana a bisa langsung dituju verteks yang diinginkan, dan bisa langsung

```
Generate3VertexCycle()
 1  found ← false
 2  for i = 1 to N − 1
 3      if found
 4          break
 5      if edge[importantVertex][i] = 0
 6          continue
 7      for j = 1 to N − 1
 8          if edge[i][j] = 1 AND
    edge[j][importantVertex] = 1
 9              found ← true
10              Lists[CurrentCycle].push_front(j)
11              Lists[CurrentCycle].push_front(i)
12              Lists[CurrentCycle].push_front(0)
13              firstCycle[0] ← importantVertex
14              firstCycle[1] ← i
15              firstCycle[2] ← j
```

Gambar 3.7 *Pseudocode* Fungsi *Generate3VertexCycle*

menuju verteks b yang bisa langsung menuju verteks yang diinginkan.

3.6 Desain Fungsi *UpdateVertex*

Fungsi *UpdateVertex* akan mengategorikan suatu verteks berdasarkan hubungannya dengan verteks-verteks yang berada di dalam *cycle*. Terdapat 4 kategori verteks, yaitu:

1. *CurrentCycle*, yaitu verteks yang berada pada *cycle*.
2. *FromCycle*, yaitu verteks yang hanya bisa dituju dari verteks pada*cycle*, tapi tidak bisa menuju verteks dalam *cycle*.

```
UpdateVertex(X)
 1  from ← false
 2  to ← false
 3  for i = 0 to Lists[CurrentCycle].end()
 4       from ← from OR edge[i][X] = 1
 5       to ← to OR edge[X][i] = 1
 6  if from AND to
 7       BothCycle.push(X)
 8  else if from
 9            Lists[FromCycle].push_front(X)
10  else if to
11            Lists[ToCycle].push_front(X)
```

Gambar 3.8 *Pseudocode* Fungsi *UpdateVertex*

3. *ToCycle*, yaitu verteks yang hanya bisa menuju verteks pada *cycle*, tapi tidak bisa dituju dari verteks pada *cycle*.
4. *BothCycle*, yaitu verteks yang bisa menuju verteks pada *cycle*, dan juga bisa dituju dari verteks pada *cycle*.

3.7 Desain Fungsi *Init*

Fungsi *Init* merupakan fungsi yang bertujuan untuk melakukan inisialisasi *cycle* awal berjumlah 3 verteks, kemudian mengelompokan verteks sesuai dengan kategorinya masing-masing dengan cara memanggil fungsi *Generate3VertexCycle* untuk membentuk *cycle* dan *UpdateVertex* untuk mengategorikan verteks.

```
Init()
1  Generate3VertexCycle()
2  for i = 0 to N
3       if i! = firstCycle[0] AND i! = firstCycle[1]
   AND i! = firstCycle[2]
4            UpdateVertex(i)
```

Gambar 3.9 *Pseudocode* Fungsi *Init*

3.8 Desain Fungsi *ReduceCycle*

Fungsi *ReduceCycle* akan mengeluarkan satu verteks dari dalam *cycle* sehingga jumlah verteks pada *cycle* akan berkurang satu. Karena fungsi ini akan dipanggil saat akan memasukkan verteks yang bisa menuju semua verteks pada *cycle* ke dalam *cycle* setelah verteks pertama, maka verteks kedua bisa dikeluarkan.

```
ReduceCycle()
1  pos ← Lists[CurrentCycle].begin() + 1
2  vertex ← *pos
3  Lists[CurrentCycle].erase(pos)
4  Lists[BothCycle].push_front(vertex)
5  Address[Vertex] ← Lists[BothCycle].begin()
6  Type[vertex] ← BothCycle
```

Gambar 3.10 *Pseudocode* Fungsi *ReduceCycle*

```
UpdateAfterNewVertexInCycle()
 1  it ← Lists[FromCycle].begin()
 2  while it! = Lists[FromCycle.end()
 3       if edge[∗it][X] ==′ 1′
 4            BothCycle.push(∗it)
 5            it ← Lists[FromCycle].erase(it)
 6       else
 7            it ← it + 1
 8  it ← Lists[ToCycle].begin()
 9  while it! = Lists[ToCycle.end()
10       if edge[X][∗it] ==′ 1′
11            BothCycle.push(∗it)
12            it ← Lists[ToCycle].erase(it)
13       else
14            it ← it + 1
```

Gambar 3.11 *Pseudocode* Fungsi *UpdateAfterNewVertexInCycle*

3.9 Desain Fungsi *UpdateAfterNewVertexInCycle*

Fungsi *UpdateAfterNewVertexInCycle* akan melakukan pengecekan dan merubah jika terdapat verteks pada kategori *FromCycle* yang bisa menuju langsung verteks yang baru saja masuk *cycle* dan *ToCycle* yang bisa bisa dituju langsung verteks yang baru saja masuk *cycle* sehingga verteks tersebut masuk kategori *BothCycle*.

3.10 Desain Fungsi *AddOneVertex*

Fungsi *AddOneVertex* akan menambahkan 1 verteks dari kategori *BothCycle* ke dalam *cycle* dengan mencari verteks pada

```
AddOneVertex()
 1  X ← BothCycle].top()
 2  BothCycle.pop()
 3  iterator ← Lists[CurrentCycle].begin()
 4  next ← iterator + 1
 5  for next to List[CurrentCyle].end()
 6       if edge[iterator][X] = 1 AND edge[X][next] = 1
 7            position ← iterator
 8            Found ← true
 9            break
10       iterator ← iterator + 1
11  if !Found
12       position ← Lists[CurrentCycle].back()
13  Lists[CurrentCycle].insert(position, X)
14  UpdateAfterNewVertexInCycle(X)
```

Gambar 3.12 *Pseudocode* Fungsi *AddOneVertex*

cycle yang menuju verteks pada kategori *BothCycle*, dan verteks setelahnya bisa dituju oleh verteks yang sama pada kategori *BothCycle*. Kemudian verteks pada pada kategori *BothCycle* itu akan dimasukkan diantara 2 verteks pada *cycle* tersebut. Sehingga banyak verteks pada *cycle* bertambah 1.

3.11 Desain Fungsi *AddTwoVertex*

Fungsi *AddTwoVertex* akan menambahkan 2 verteks, masing-masing dari kategori *FromCycle* dan *ToCycle* ke dalam *cycle* dengan mencari verteks pada *FromCycle* yang menuju verteks pada kategori *ToCycle*. Kedua verteks tersebut bisa dimasukkan di posisi dimana saja pada *cycle*. Pada program ini,

verteks tersebut dimasukkan setelah verteks pertama pada *cycle*. Supaya jumlah verteks pada *cycle* hanya bertambah 1, maka sebelum verteks tersebut dimasukkan ke dalam *cycle*, fungsi *ReduceCycle* akan dipanggil untuk mengeluarkan 1 verteks pada *cycle*, kemudian verteks pada pada kategori *FromCycle* dan *ToCycle* akan dimasukkan ke dalam *cycle*. Sehingga banyak verteks pada *cycle* bertambah 1.

```
AddTwoVertex()
 1  found ← false
 2  from ← Lists[FromCycle].begin()
 3  to ← Lists[ToCycle].begin()
 4  for from to Lists[FromCycle].end()
 5       for to to Lists[ToCycle].end()
 6            if edge[*from][*to] = 1
 7                 V1 ← *from
 8                 V2 ← *to
 9                 found ← true
10  position ← Lists[CurrentCycle].begin()
11  position ← position + 1
12  Lists[FromCycle].erase(from)
13  Lists[CurrentCycle].insert(position, V1)
14  Lists[ToCycle].erase(to)
15  Lists[CurrentCycle].insert(position, V2)
16  UpdateAfterNewVertexInCycle(V1)
17  UpdateAfterNewVertexInCycle(V2)
```

Gambar 3.13 *Pseudocode* Fungsi *AddTwoVertex*

```
ExpandCycle()
1  if !BothCycle.empty()
2        AddOneVertex()
3  else
4        ReduceCycle()
5        AddTwoVertex()
```

Gambar 3.14 *Pseudocode* Fungsi *ExpandCycle*

3.12 Desain Fungsi *ExpandCycle*

Fungsi *ExpandCycle* akan menambahkan verteks dari luar *cycle* ke dalam *cycle* membuat banyak verteks pada *cycle* bertambah 1. Jika terdapat verteks pada kategori *BothCycle*, maka verteks tersebut yang diambil. Karena jika tidak ada verteks pada *BothCycle*, maka pasti ada verteks pada pasti ada verteks *FromCycle* yang bisa menuju verteks pada *ToCycle*. Jika masih terdapat verteks pada kategory *BothCycle*, belum tentu verteks *FromCycle* bisa menuju verteks *ToCycle*, tapi bisa menuju verteks pada *BothCycle*. Karena jika ingin menambahkan verteks dari *FromCycle* dan *ToCycle* akan bertambah 2 verteks, maka 1 verteks akan dikeluarkan dari *cycle*.

3.13 Deskripsi Umum Sistem

Pada subbab ini akan dijelaskan mengenai deskripsi dari algoritma yang dirancang pada fungsi main.

Sistem akan menerima masukan jumlah vertex, dan *edges* sebagai hubungan antar verteks. Setelah mendapat masukan, sistem akan melakukan pengecekan apakah turnamen merupakan

strongly connected atau tidak. Jika tidak *strongly connected*, kita tampilkan "impossible" dan program selesai. Jika *strongly connected*, kita melakukan inisialisasi *cycle*, yaitu mendapatkan *cycle* dengan 3 vertex. Lalu tampilkan isi *cycle*. Kemudian kita iterasi dari 4 sampai N, masukan verteks dari luar *cycle* ke dalam *cycle*, dan tampilkan isi *cycle*.

```
main()
1  edge[0..N − 1] ← ReadInput()
2  if !IsStronglyConnected()
3       print "impossible"
4       return
5  Init()
6  Print()
7  for i = 4 to N
8       ExpandCycle()
9       Print()
```

Gambar 3.15 *Pseudocode* Fungsi *Main*

(Halaman ini sengaja dikosongkan)

BAB IV

IMPLEMENTASI

Pada bab ini dibahas mengenai implementasi.

4.1 Lingkungan Implementasi

Pada bab ini dijelaskan mengenai implementasi dari desain algoritma penyelesaian SPOJ CYCLERUN - Riding in Cycles.

1. Perangkat Keras
   (a) *Processor* Intel Core i7-6700 CPU @ 2.60GHz.
   (b) *Memory* 8GB.
2. Perangkat Lunak
   (a) Sistem operasi Windows 10 64-bit
   (b) *Integrated Development Environment* DevC++ 5.11.

4.2 Rancangan Data

Pada subbab ini dijelaskan mengenai desain data masukan yang diperlukan untuk melakukan proses algoritma, dan data keluaran yang dihasilkan oleh program.

4.2.1 Data Masukan

Data masukan adalah data yang akan diproses oleh program sebagai masukan menggunakan algoritma yang telah dirancang dalam tugas akhir ini.

Data masukan berupa berkas teks yang berisi data dengan format yang telah ditentukan pada deskripsi SPOJ CYCLERUN - Riding in Cycles. Pada masing-masing berkas data masukan, baris pertama berupa sebuah bilangan bulat N yang merepresentasikan jumlah verteks yang ada pada berkas tersebut. Untuk sebanyak N baris selanjutnya, masukan berupa sebuah baris masukan yang terdiri dari N buah *integer*, yang merepresentasikan *edge* pada graf. Jika pada baris 2 pada kolom 3 terdapat angka 1, maka verteks 2 bisa langsung menuju verteks

3. Jika pada baris 4 pada kolom 3 terdapat angka 0, maka verteks 4 tidak bisa langsung menuju verteks 3.

4.2.2 Data Keluaran

Jika graf memiliki $N - 2$ *cycle* dengan banyak verteks dari 3 sampai N dan tiap *cycle* diawali oleh verteks 1, maka data keluaran berupa N baris yang tiap barisnya berisi urutan isi verteks diawali dan diakhiri oleh verteks 1. Jika tidak ada, maka keluaran berupa string "impossible".

4.3 Implementasi Algoritma

Pada subbab ini akan dijelaskan tentang implementasi proses algoritma secara keseluruhan berdasarkan desain yang telah dijelaskan pada Bab 3.

4.3.1 *Header* yang Diperlukan

Implementasi algoritma untuk menyelesaikan SPOJ CYCLERUN - Riding in Cycles membutuhkan lima buah *header* yaitu *iostream*, *stack*, dan *list*, seperti yang terlihat pada Kode Sumber 4.1.

```
1 #include<iostream>
2 #include<stack>
3 #include<list>
```

Kode Sumber 4.1 *Header* yang diperlukan

Header iostream berisi modul untuk menerima masukan dan memberikan keluaran. *Header stack* berisi struktur data yang digunakan untuk menyimpan data dalam penggunaan operasi *LIFO* (*last-in first-out*). *Header list* berisi struktur data yang digunakan untuk menyimpan data yang dalam penggunaan menghapus datanya memiliki kompleksitas linear.

4.3.2 Variabel Global

Variabel global untuk memudahkan dalam mengakses data yang digunakan lintas fungsi. Kode sumber implementasi variabel global dapat dilihat pada Kode Sumber 4.2.

```
1  using namespace std;
2  const int maxVertex = 1001;
3  const int importantVertex = 0;
4  int n;
5  char edge[maxVertex][maxVertex];
6  bool visited[maxVertex], inStack[maxVertex];
7  stack<int> checkPan;
8  int firstCycle[3];
9  list<int> Lists[3];
10 const int CurrentCycle = 0;
11 const int FromCycle = 1;
12 const int ToCycle = 2;
13 stack<int> BothCycle;
```

Kode Sumber 4.2 Variabel Global

4.3.3 Implementasi Fungsi *Main* pada Program Naif

Berdasarkan pada desain fungsi *main* pada program naif yang telah dirancang pada Gambar 3.3, awalnya program menerima masukan data berupa N sebagai banyaknya verteks. Pada N baris berikutnya, program akan membaca input data berupa string $edge$. Kemudian program akan memanggil fungsi $Generate3VertexCycles$ yang berfungsi untuk membuat semua *cycle* dengan 3 verteks. Jika tidak ada satupun *cycle* dengan 3 verteks, program akan menampilkan "impossible" dan selesai. Jika ada, maka program akan terus berjalan. Program akan melakukan perulangan sampai *cycle* berisi N verteks dengan memanggil fungsi $ExpandCycles$, yang akan menambahkan 1 verteks ke dalam semua *cycle*. Jika ada *cycle* yang tidak ada, program akan menampilkan "impossible" dan selesai. Jika

semua *cycle* yang diharapkan ada, program akan melakukan perulangan dari 3 sampai N dan menampilkan verteks pada *cycle* dengan jumlah verteks sesuai urutan perulangan. Fungsi Implementasi fungsi $main$ dapat dilihat pada Kode Sumber 4.3.

```
1  int main(){
2    cin >> N;
3    for(int i=0;i<N;i++)cin >> edge[i];
4    Generate3VertexCycles();
5    if(cycles[3].empty()){
6      cout << "impossible" <<endl;
7      return 0;
8    }
9    for(int i=3;i<N;i++){
10     bool flag = ExpandCycles(i);
11     if(!flag){
12       cout << "impossible" <<endl;
13       return 0;
14     }
15   }
16   for(int i=3;i<=N;i++)
17     Print(i);
18 }
```

Kode Sumber 4.3 Fungsi *main*

4.3.4 Implementasi Fungsi *Generate3VertexCycles*

Bagian ini adalah implementasi fungsi *Generate3VertexCycles* dari hasil perancangan pada Gambar 3.1 untuk membuat semua *cycle* dengan berjumlah 3 verteks. Implementasi dari fungsi *Generate3VertexCycles* dapat dilihat pada Kode Sumber 4.4.

```
1  void Generate3VertexCycles(){
2    for(int i=1;i<N;i++){
3      if(edge[0][i]=='0')continue;
4          for(int j=1;j<N;j++){
5            if(edge[i][j]=='1' && edge[j][0]=='1'){
6              vector<int> cycle;
7                   cycle.push_back(0);
8                   cycle.push_back(i);
9                   cycle.push_back(j);
10                  cycle.push_back(0);
11                  map<int,bool> flag;
12                  flag[0] = true;
13                  flag[i] = true;
14                  flag[j] = true;
15                  cycles[3].push_back(make_pair(cycle,
                        ↪ flag));
16           }
17         }
18   }
19 }
```

Kode Sumber 4.4 Fungsi *Generate3VertexCycles*

4.3.5 Implementasi Fungsi *ExpandCycles*

Bagian ini adalah implementasi fungsi *ExpandCycles* dari hasil perancangan pada Gambar 3.2 untuk menambahkan verteks pada semua *cycle* dengan t verteks dan mengembalikan info apakah ada *cycle* dengan $t + 1$ verteks. Implementasi dari fungsi *ExpandCycles* dapat dilihat pada Kode Sumber 4.5.

```
1  bool ExpandCycles(int t){
2    bool flag = false;
3    for(int i=0;i<cycles[t].size();i++){
4      vector<int> cycle = cycles[t][i].first;
5          map<int,bool> visited = cycles[t][i].second;
6          for(int j=1;j<N;j++){
7            if(visited.count(j)==1)continue;
8            for(int k=1;k<cycle.size();k++){
9              if(edge[cycle[k-1]][j]=='1' && edge[j][
                   ↪ cycle[k]]=='1'){
10                 flag = true;
11               vector<int> newCycle;
12                 map<int,bool> newFlag;
13                 newFlag.insert(visited.begin(),
                       ↪ visited.end());
14                 newFlag[j] = true;
15                 for(int l=0;l<k;l++)
16                 newCycle.push_back(cycle[l]);
17                 newCycle.push_back(j);
18                 for(int l=k;l<cycle.size();l++)
19                   newCycle.push_back(cycle[l]);
20               cycles[t+1].push_back(make_pair(newCycle,
                     ↪ newFlag));
21             }
22           }
23     }
24   }
25   return flag;
26 }
```

Kode Sumber 4.5 Fungsi *ExpandCycles*

4.3.6 Implementasi Fungsi *Main*

Berdasarkan pada desain fungsi *main* yang telah dirancang pada Gambar 3.15, awalnya program menerima masukan data berupa N sebagai banyaknya verteks. Pada N baris berikutnya, program akan membaca input data berupa *string edge*.

Kemudian program akan memanggil fungsi *IsStronglyConnected* yang berfungsi untuk mengecek apakah graf yang didapat dari input merupakan *strongly connected* atau tidak. Jika tidak, program akan menampilkan "impossible" dan program selesai. Jika iya, maka program akan terus berjalan. Program akan memangil fungsi *Init* yang berfungsi melakukan inisialisasi *cycle* dengan jumlah verteks minimal pada *cycle* yaitu 3 verteks. Berikutnya program akan memanggil fungsi *Print* yang menampilkan isi urutan verteks dari *cycle*. Setelah itu akan melakukan perulangan sampai *cycle* berisi *N* verteks untuk memanggil fungsi *ExpandCycle*, yang akan menambahkan 1 verteks ke dalam *cycle*, dan juga memanggil fungsi *Print*. Implementasi fungsi *main* dapat dilihat pada Kode Sumber 4.6.

```
1  int main(){
2    cin >> n;
3    for(int i=0;i<n;i++)cin >> edge[i];
4    if(!IsStronglyConnected()){
5      printf("impossible\n");
6      return 0;
7    }
8    Init();
9    Print();
10   for(int i=4;i<=n;i++){
11    ExpandCycle();
12    Print();
13   }
14 }
```

Kode Sumber 4.6 Fungsi *Main*

4.3.7 Implementasi Fungsi IsStronglyConnected

Bagian ini adalah implementasi fungsi *IsStronglyConnected* dari hasil perancangan fungsi *IsStronglyConnected* pada Gambar 3.6 untuk mengecek apakah suatu graf merupakan *strongly connected* atau tidak. Struktur data yang dipakai adalah *stack*.

Implementasi dari fungsi *IsStronglyConnected* dapat dilihat pada Kode Sumber 4.7.

```
1  void FirstDFS(int x){
2    visited[x] = true;
3    for(int i=0;i<n;i++){
4      if(!visited[i] && edge[x][i]=='1')
5            FirstDFS(i);
6    }
7    checkPan.push(x);
8  }
9  void SecondDFS(int x){
10   visited[x] = false;
11   for(int i=0;i<n;i++){
12     if(visited[i] && edge[i][x]=='1')
13           SecondDFS(i);
14   }
15 }
16 bool IsStronglyConnected(){
17   int scc = 0;
18   for(int i=0;i<n;i++)
19     if(!visited[i])FirstDFS(i);
20   while(!checkPan.empty()){
21     int top = checkPan.top();
22     checkPan.pop();
23     if(visited[top]){
24       scc++;
25       SecondDFS(top);
26         }
27   }
28   return scc == 1;
29 }
```

Kode Sumber 4.7 Fungsi *IsStronglyConnected*

4.3.8 Implementasi Fungsi *Init*

Bagian ini adalah implementasi fungsi *Init* dari hasil perancangan fungsi *Init* pada Gambar 3.9 untuk melakukan pembuatan *cycle* awal berjumlah 3 verteks dan mengategorikan

verteks yang berada di luar *cycle* sesuai dengan hubungannya terhadap verteks-verteks yang berada di dalam *cycle*. Implementasi dari fungsi *Init* dapat dilihat pada Kode Sumber 4.8.

```
1  void Init(){
2    Generate3VertexCycle();
3    for (int i = 0; i < n; i++)
4          if (i != firstCycle[0] && i!= firstCycle[1] &&
               ↪ i!= firstCycle[2])
5              UpdateVertex(i);
6  }
```

Kode Sumber 4.8 Fungsi *Init*

4.3.9 Implementasi Fungsi *Generate3VertexCycle*

Bagian ini adalah implementasi fungsi *Generate3VertexCycle* dari hasil perancangan pada Gambar 3.7 untuk membuat *cycle* dengan berjumlah 3 verteks. Struktur data yang digunakan untuk menyimpan *cycle* adalah *list*, karena dibutuhkan waktu yang cepat untuk proses penghapusan dan pemasukan elemen di dalam *cycle*. Implementasi dari fungsi *Generate3VertexCycle* dapat dilihat pada Kode Sumber 4.9.

4.3.10 Implementasi Fungsi *UpdateVertex*

Bagian ini adalah implementasi fungsi *UpdateVertex* dari hasil perancangan pada Gambar 3.8 untuk mengecek dan mengubah dari verteks pada kategori *FromCycle* dan *ToCycle* menjadi berada pada kategori *BothCycle*. Implementasi dari fungsi *UpdateVertex* dapat dilihat pada Kode Sumber 4.10. Struktur data yang digunakan untuk menyimpan *FromCycle* dan *ToCycle* adalah *list*, karena dibutuhkan waktu yang cepat untuk proses penghapusan dan pemasukan elemen di dalam *FromCycle* dan *ToCycle*. Struktur data yang digunakan untuk menyimpan

BothCycle adalah *stack*, karena elemen *BothCycle* yang akan dipakai selalu yang terakhir dimasukkan ke dalam *BothCycle*.

```
1  void Generate3VertexCycle(){
2    bool found = false;
3    for(int i=1;i<n && !found;i++){
4      if(edge[0][i]=='0')continue;
5      for(int j=1;j<n && !found;j++){
6        if(edge[i][j]=='1' && edge[j][0]=='1'){
7          found = true;
8          Lists[CurrentCycle].push_front(j);
9          Lists[CurrentCycle].push_front(i);
10         Lists[CurrentCycle].push_front(importantVertex)
               ↪ ;
11         firstCycle[0] = importantVertex;
12         firstCycle[1] = i;
13         firstCycle[2] = j;
14       }
15     }
16   }
17 }
```

Kode Sumber 4.9 Fungsi *Generate3VertexCycle*

```
1  void UpdateVertex(const int X) {
2   bool From = false, To = false;
3      for(auto it = Lists[CurrentCycle].begin();it!=Lists
           ↪ [CurrentCycle].end();++it){
4          From |= (edge[*it][X] == '1');
5          To |= (edge[X][*it] == '1');
6          }
7      if (From && To)
8          BothCycle.push(X);
9          else if (From)
10         Lists[FromCycle].push_front(X);
11     else if (To)
12         Lists[ToCycle].push_front(X);
13 }
```

Kode Sumber 4.10 Fungsi *UpdateVertex*

4.3.11 Implementasi Fungsi *ExpandCycle*

Bagian ini adalah implementasi fungsi *ExpandCycle* dari hasil perancangan pada Gambar 3.14 untuk menambahkan verteks pada *cycle*. Implementasi dari fungsi *ExpandCycle* dapat dilihat pada Kode Sumber 4.11.

```
1  void ExpandCycle(){
2    if(!BothCycle.empty())
3                    AddOneVertex();
4          else{
5                    ReduceCycle();
6                    AddTwoVertex();
7          }
8  }
```

Kode Sumber 4.11 Fungsi *ExpandCycle*

4.3.12 Implementasi Fungsi *AddOneVertex*

Bagian ini adalah implementasi fungsi *AddOneVertex* dari hasil perancangan pada Gambar 3.12 untuk menambahkan 1 verteks dari kategori *BothCycle* ke *cycle*. Implementasi dari fungsi *AddOneVertex* dapat dilihat pada Kode Sumber 4.12.

4.3.13 Implementasi Fungsi *AddTwoVertex*

Bagian ini adalah implementasi fungsi *AddTwoVertex* dari hasil perancangan pada Gambar 3.13 untuk menambahkan 1 verteks dari *FromCycle* dan 1 verteks dari *ToCycle* ke dalam *cycle*. Implementasi dari fungsi *AddTwoVertex* dapat dilihat pada Kode Sumber 4.13.

4.3.14 Implementasi Fungsi *ReduceCycle*

Bagian ini adalah implementasi fungsi *ReduceCycle* dari hasil perancangan pada Gambar 3.10 untuk mengeluarkan 1

verteks dari *cycle*. Implementasi dari fungsi *ReduceCycle* dapat dilihat pada Kode Sumber 4.14.

4.3.15 Implementasi Fungsi *UpdateAfterNewVertexInCycle*

Bagian ini adalah implementasi fungsi *UpdateAfterNewVertexInCycle* dari hasil perancangan pada Gambar 3.11 untuk mengecek dan mengubah kategori verteks dari *FromCycle* dan *ToCycle* menjadi *BothCycle*, sesuai dengan hubungannya dengan verteks yang baru saja masuk ke dalam *cycle*. Implementasi dari fungsi *UpdateAfterNewVertexInCycle* dapat dilihat pada Kode Sumber 4.15.

```
1  void AddOneVertex(){
2          int X = BothCycle.top();
3          list<int>::iterator position;
4          list<int>::iterator it = Lists[CurrentCycle].
               ↪ begin(), next = it;
5          ++next;
6          bool Found = false;
7          for(;next!=Lists[CurrentCycle].end() && !Found
               ↪ ;++next, ++it){
8                  if(edge[*it][X]=='1' && edge[X][*next
                       ↪ ]=='1'){
9                  position = it;
10                 position++;
11                 Found = true;
12                 }
13         }
14         if(!Found)position = Lists[CurrentCycle].end();
15         BothCycle.pop();
16         Lists[CurrentCycle].insert(position,X);
17         UpdateAfterNewVertexInCycle(X);
18 }
```

Kode Sumber 4.12 Fungsi *AddOneVertex*

```
1  void AddTwoVertex(){
2          int V1,V2;
3          bool found = false;
4          list<int>::iterator from = Lists[FromCycle].
               ↪ begin(),to;
5          for(;from!=Lists[FromCycle].end() &&!found;++
               ↪ from){
6                  for(to = Lists[ToCycle].begin();to!=
                       ↪ Lists[ToCycle].end() && !found
                       ↪ ;++to){
7                          if(edge[*from][*to]=='1'){
8                                  V1 = *from;
9                                  V2 = *to;
10                                 found = true;
11                                 break;
12                         }
13                 }
14                 if(found)break;
15         }
16         list<int>::iterator position = Lists[
               ↪ CurrentCycle].begin();
17         position++;
18         Lists[FromCycle].erase(from);
19         Lists[CurrentCycle].insert(position,V1);
20         Lists[ToCycle].erase(to);
21         Lists[CurrentCycle].insert(position,V2);
22         UpdateAfterNewVertexInCycle(V1);
23         UpdateAfterNewVertexInCycle(V2);
24 }
```

Kode Sumber 4.13 Fungsi *AddTwoVertex*

```
1  void ReduceCycle() {
2   list<int>::iterator position = Lists[CurrentCycle].
        ↪ begin();
3   position++;
4   int vertex = *position;
5   Lists[CurrentCycle].erase(position);
6   Type[vertex] = BothCycle;
7   Lists[BothCycle].push_front(vertex);
8   Address[vertex] = Lists[BothCycle].begin();
9  }
```

Kode Sumber 4.14 Fungsi *ReduceCycle*

```
1  void ReduceCycle() {
2   list<int>::iterator position = Lists[CurrentCycle].
        ↪ begin();
3   position++;
4   int vertex = *position;
5   Lists[CurrentCycle].erase(position);
6   Type[vertex] = BothCycle;
7   Lists[BothCycle].push_front(vertex);
8   Address[vertex] = Lists[BothCycle].begin();
9  }
```

Kode Sumber 4.15 Fungsi *UpdateAfterNewVertexInCycle*

BAB V

PENGUJIAN DAN EVALUASI

5.1 Lingkungan Uji Coba

Lingkungan uji coba menggunakan sebuah komputer dengan spesifikasi perangkat lunak dan perangkat keras sebagai berikut:

1. Perangkat Keras
   (a) *Processor* Intel Core i7-6700 CPU @ 2.60GHz.
   (b) *Memory* 8GB.
2. Perangkat Lunak
   (a) Sistem operasi Windows 10 64-bit
   (b) *Integrated Development Environment* DevC++ 5.11.

5.2 Uji Coba Kebenaran

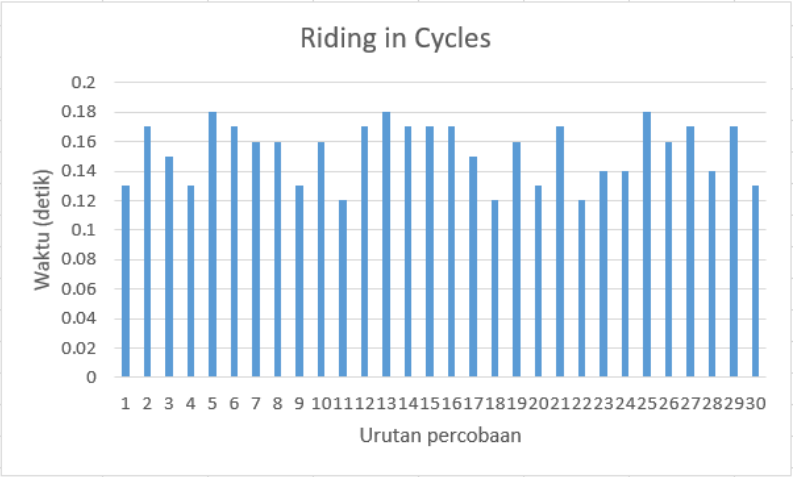
Uji coba kebenaran dilakukan dengan analisis penyelesaian sebuah contoh kasus dengan pendekatan penyelesaian yang telah dijelaskan pada bab 2 dengan hasil dari luaran program dan pengumpulan berkas kode sumber hasil implementasi ke situs sistem penilaian daring SPOJ. Permasalahan yang diselesaikan adalah Riding in Cycles dengan kode CYCLERUN.

Dari hasil uji coba kebenaran yang dilakukan pada situs penilaian SPOJ menggunakan metode naif seperti pada Gambar 5.1, dapat dilihat bahwa program mendapat keluaran *time limit exceeded*, yang berarti waktu eksekusi program melebihi batas waktu. Sedangkan hasil uji coba kebenaran yang dilakukan menggunakan program berbasis *strongly connected* dan *vertex pancyclic* seperti pada Gambar 5.2, mendapat keluaran *accepted*, yang berarti berhasil menyelesaikan permasalahan.

Gambar 5.1 Hasil Uji Coba pada Situs Penilaian SPOJ Metode Naif

19760512		2017-07-09 04:56:46	Riding in cycles	**accepted** edit ideone it	0.12	17M	CPP

Gambar 5.2 Hasil Uji Coba pada Situs Penilaian SPOJ



Gambar 5.3 Grafik Ujicoba Waktu

5.3 Uji Coba Kinerja

Program diuji sebanyak 30 kali pada situs penilaian SPOJ untuk melihat variasi waktu dan dibutuhkan program. Hasil uji coba sebanyak 30 kali dapat dilihat pada Gambar 5.3.

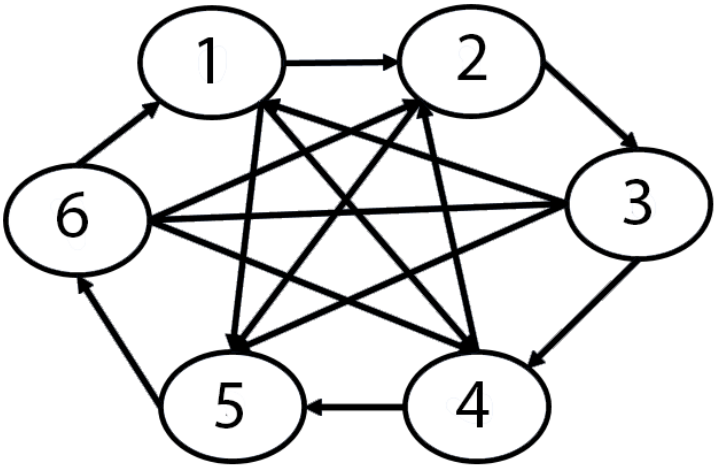
Dari hasil uji coba sebanyak 30 kali, seluruh kode sumber mendapat keluaran *Accepted*, dengan waktu minimum sebesar 0,12 detik dan rata-rata 0,1533 detik.

5.4 Uji Coba Menggunakan Contoh Kasus

Program diuji menggunakan data dari contoh kasus uji, dengan masukan sesuai pada Gambar 5.4 dan diilustrasikan pada Gambar 5.5.

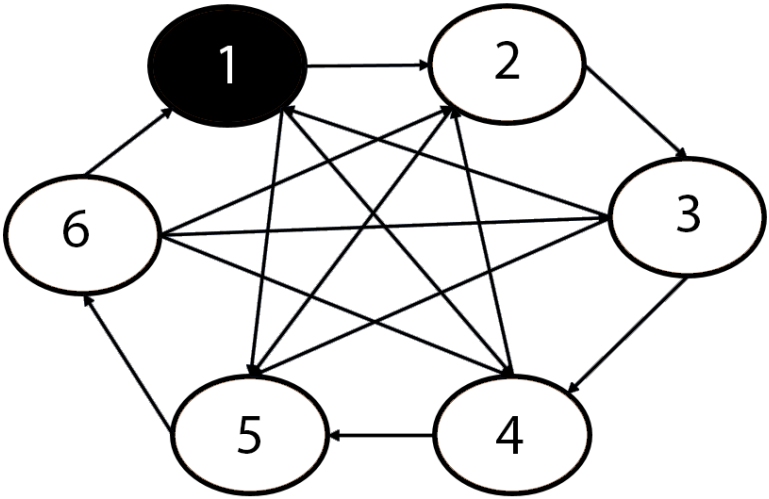
```
6
010110
001010
100110
010010
000001
111100
```

Gambar 5.4 Data Masukan untuk Uji Coba



Gambar 5.5 Ilustrasi Graf dari Masukan

Dijelaskan pada subbab 3.13 yang pertama dilakukan adalah menggunakan Algoritma Kosaraju. Tiap verteks yang belum dikunjungi akan melakukan *FirstDFS*, yaitu menandai verteks telah dikunjungi, tiap verteks yang bisa dikunjungi lakukan *FirstDFS*, lalu masukkan verteks pada *stack*.

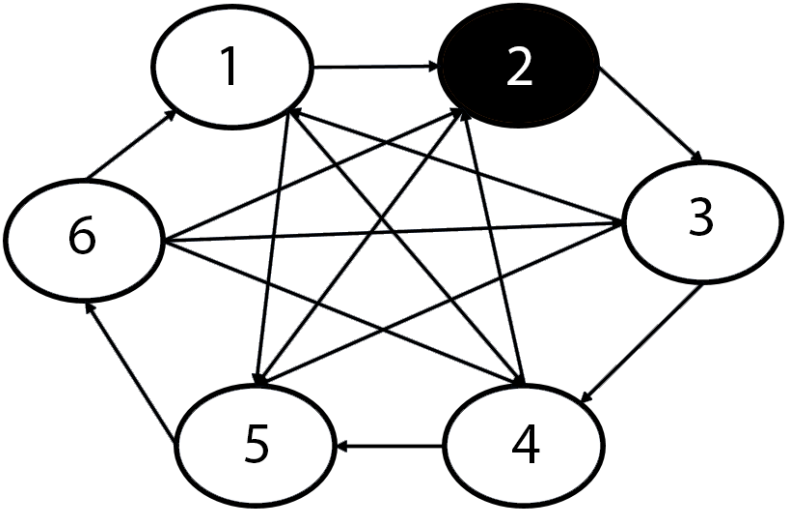


Gambar 5.6 Cek *Strongly Connected* (1)

Tabel 5.1 Mendapatkan *Stack* (1)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	belum dikunjungi	-
3	belum dikunjungi	-
4	belum dikunjungi	-
5	belum dikunjungi	-
6	belum dikunjungi	-

DFS dimulai dari verteks 1. Status verteks 1 berubah menjadi sudah dikunjungi. Karena verteks 1 bisa langsung mengunjungi verteks 2, dan verteks 2 belum dikunjungi maka selanjutnya pindah ke verteks 2.

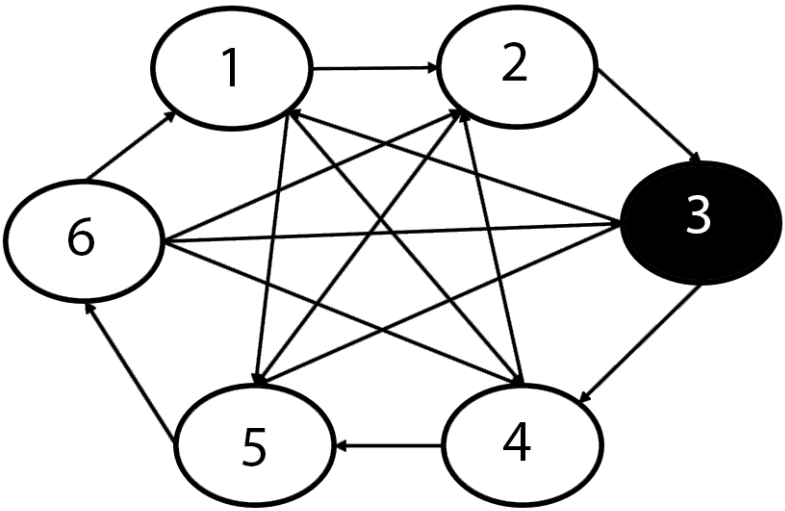


Gambar 5.7 Cek *Strongly Connected* (2)

Tabel 5.2 Mendapatkan *Stack* (2)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	belum dikunjungi	-
4	belum dikunjungi	-
5	belum dikunjungi	-
6	belum dikunjungi	-

Status verteks 2 berubah menjadi sudah dikunjungi. Karena verteks 2 bisa langsung mengunjungi verteks 3, dan verteks 3 belum dikunjungi, maka selanjutnya pindah ke verteks 3.

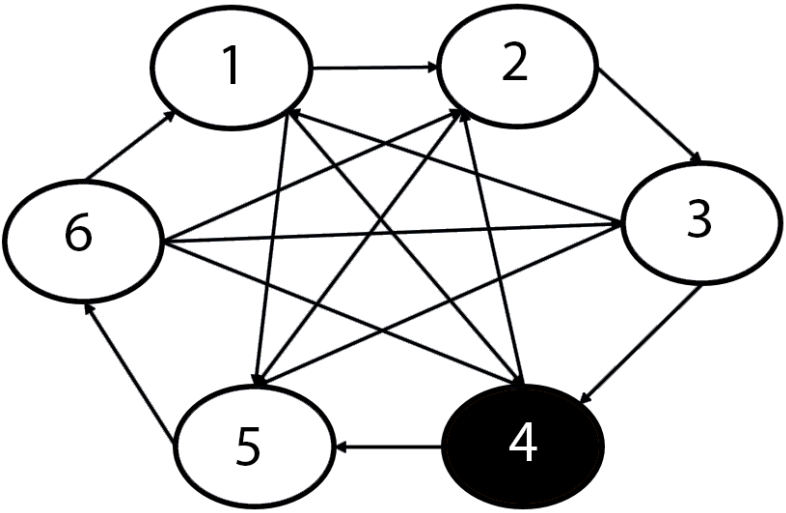


Gambar 5.8 Cek *Strongly Connected* (3)

Tabel 5.3 Mendapatkan *Stack* (3)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	-
4	belum dikunjungi	-
5	belum dikunjungi	-
6	belum dikunjungi	-

Status verteks 3 berubah menjadi sudah dikunjungi. Karena verteks 3 bisa langsung mengunjungi verteks 4, dan verteks 4 belum dikunjungi maka selanjutnya pindah ke verteks 4.

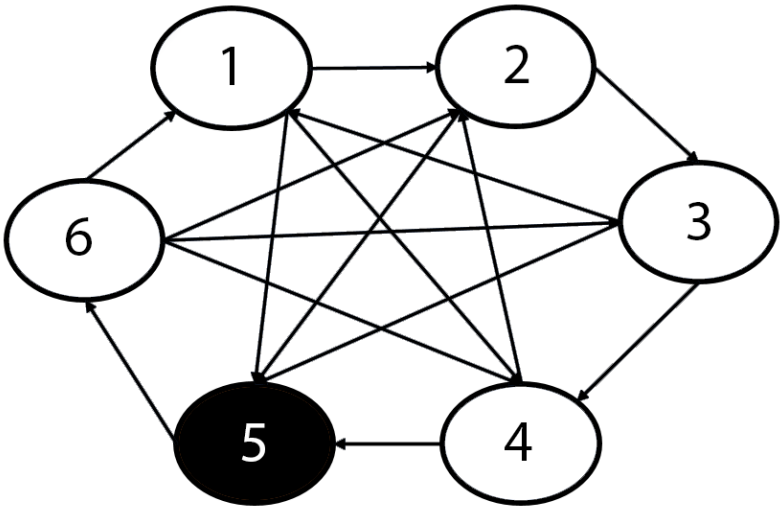


Gambar 5.9 Cek *Strongly Connected* (4)

Tabel 5.4 Mendapatkan *Stack* (4)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	-
4	sudah dikunjungi	-
5	belum dikunjungi	-
6	belum dikunjungi	-

Status verteks 4 berubah menjadi sudah dikunjungi. Karena verteks 4 bisa langsung mengunjungi verteks 5, dan verteks 5 belum dikunjungi maka selanjutnya pindah ke verteks 5.

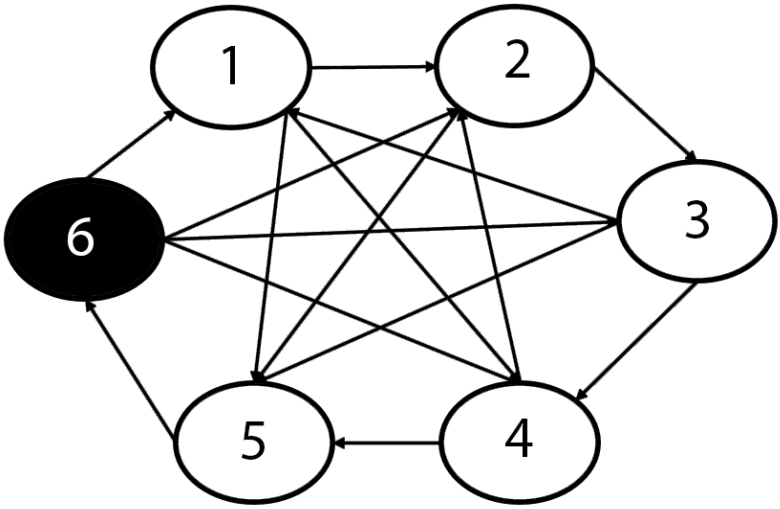


Gambar 5.10 Cek *Strongly Connected* (5)

Tabel 5.5 Mendapatkan *Stack* (5)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	-
4	sudah dikunjungi	-
5	sudah dikunjungi	-
6	belum dikunjungi	-

Status verteks 5 berubah menjadi sudah dikunjungi. Karena verteks 5 bisa langsung mengunjungi verteks 6, dan verteks 6 belum dikunjungi maka selanjutnya pindah ke verteks 6.

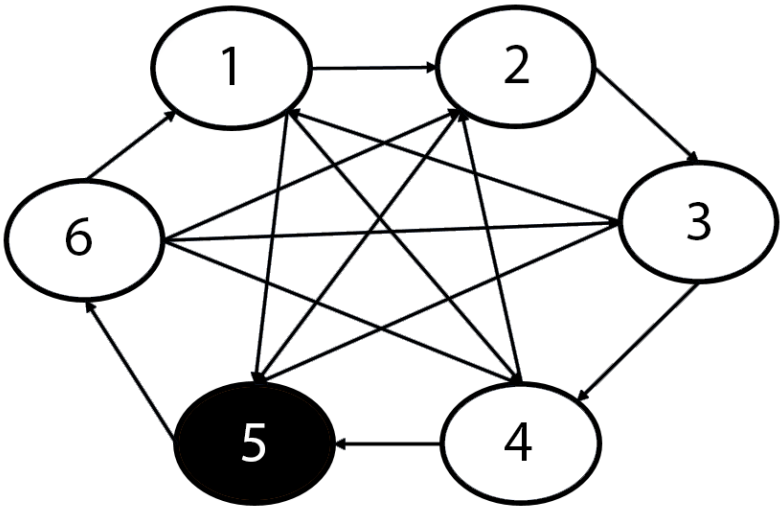


Gambar 5.11 Cek *Strongly Connected* (6)

Tabel 5.6 Mendapatkan *Stack* (6)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	-
4	sudah dikunjungi	-
5	sudah dikunjungi	-
6	sudah dikunjungi	-

Status verteks 6 berubah menjadi sudah dikunjungi. Karena status semua verteks yang bisa dituju oleh verteks 6 sudah dikunjungi, maka verteks 6 dimasukkan ke dalam *stack*, dan kembali verteks yang memanggil verteks 6, yaitu verteks 5.

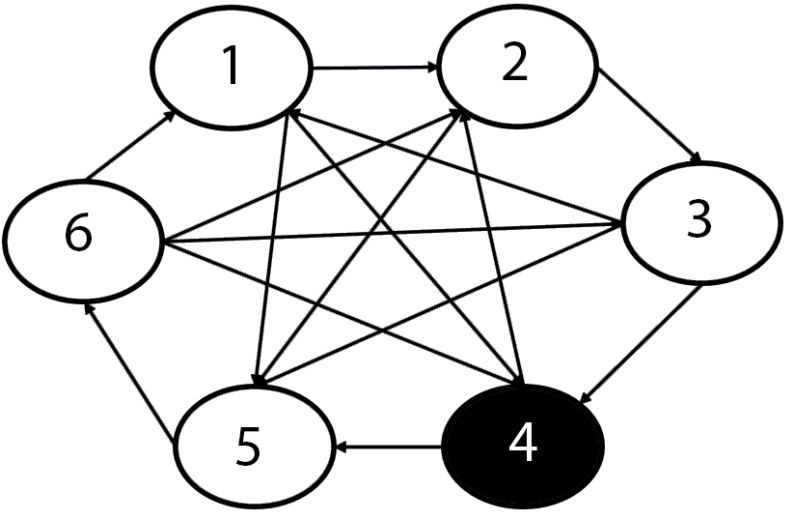


Gambar 5.12 Cek *Strongly Connected* (7)

Tabel 5.7 Mendapatkan *Stack* (7)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	-
4	sudah dikunjungi	-
5	sudah dikunjungi	-
6	sudah dikunjungi	1

Karena status semua verteks yang bisa dituju oleh verteks 5 sudah dikunjungi, maka verteks 5 dimasukkan ke dalam *stack*, dan kembali verteks yang memanggil verteks 5, yaitu verteks 4.

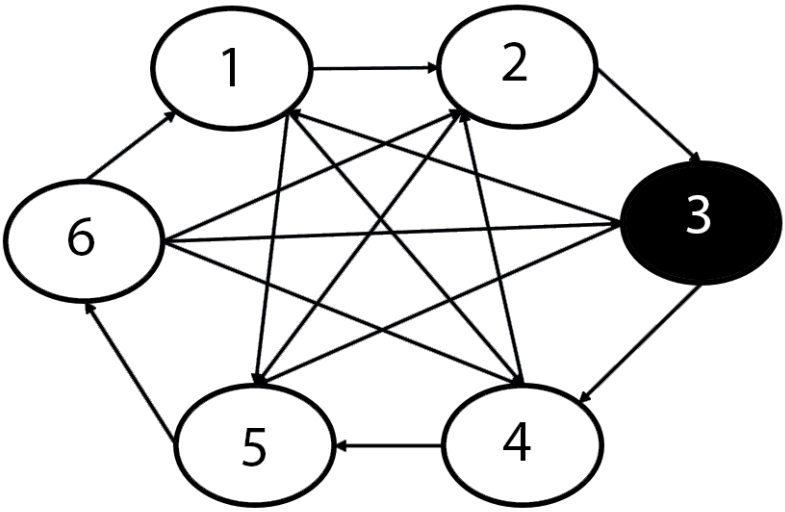


Gambar 5.13 Cek *Strongly Connected* (8)

Tabel 5.8 Mendapatkan *Stack* (8)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	-
4	sudah dikunjungi	-
5	sudah dikunjungi	2
6	sudah dikunjungi	1

Karena status semua verteks yang bisa dituju oleh verteks 4 sudah dikunjungi, maka verteks 4 dimasukkan ke dalam *stack*, dan kembali verteks yang memanggil verteks 4, yaitu verteks 3.

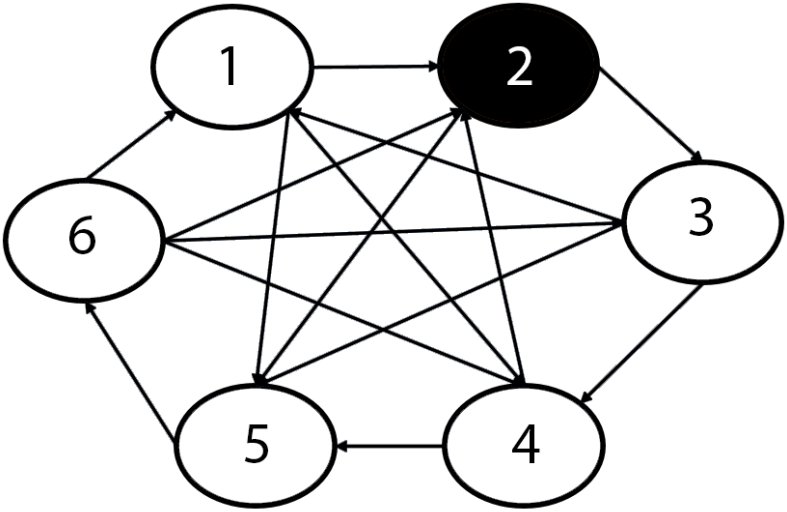


Gambar 5.14 Cek *Strongly Connected* (9)

Tabel 5.9 Mendapatkan *Stack* (9)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	-
4	sudah dikunjungi	3
5	sudah dikunjungi	2
6	sudah dikunjungi	1

Karena status semua verteks yang bisa dituju oleh verteks 3 sudah dikunjungi, maka verteks 3 dimasukkan ke dalam *stack*, dan kembali verteks yang memanggil verteks 3, yaitu verteks 2.

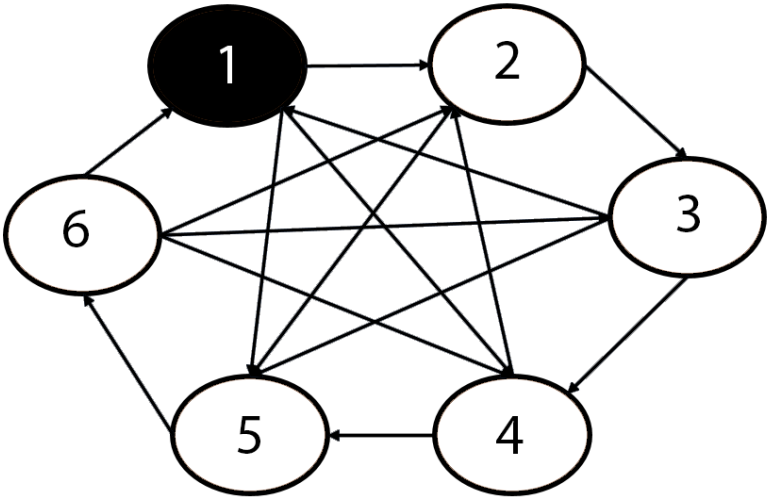


Gambar 5.15 Cek *Strongly Connected* (10)

Tabel 5.10 Mendapatkan *Stack* (10)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	-
3	sudah dikunjungi	4
4	sudah dikunjungi	3
5	sudah dikunjungi	2
6	sudah dikunjungi	1

Karena status semua verteks yang bisa dituju oleh verteks 2 sudah dikunjungi, maka verteks 2 dimasukkan ke dalam *stack*, dan kembali verteks yang memanggil verteks 2, yaitu verteks 1.

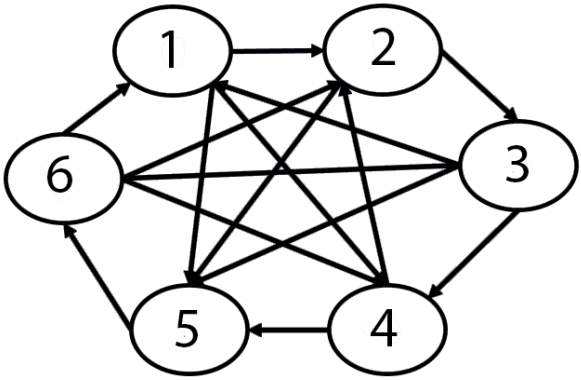


Gambar 5.16 Cek *Strongly Connected* (11)

Tabel 5.11 Mendapatkan *Stack* (11)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	-
2	sudah dikunjungi	5
3	sudah dikunjungi	4
4	sudah dikunjungi	3
5	sudah dikunjungi	2
6	sudah dikunjungi	1

Karena status semua verteks yang bisa dituju oleh verteks 1 sudah dikunjungi, maka verteks 1 dimasukkan ke dalam *stack*. *DFS* yang dimulai dari verteks 1 sudah selesai. Kemudian lakukan *DFS* dari verteks selanjutnya, yaitu verteks 2, 3, 4, 5, dan 6.

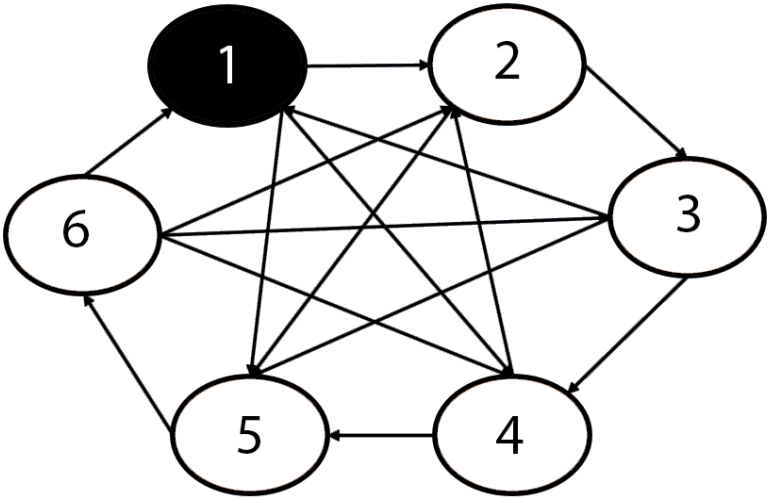


Gambar 5.17 Cek *Strongly Connected* (12)

Tabel 5.12 Mendapatkan *Stack* (12)

Verteks	**Status Dikunjungi**	**Urutan pada *Stack***
1	sudah dikunjungi	6
2	sudah dikunjungi	5
3	sudah dikunjungi	4
4	sudah dikunjungi	3
5	sudah dikunjungi	2
6	sudah dikunjungi	1

Karena status semua verteks sudah dikunjungi, maka tidak ada lagi perubahan pada *stack*. Selanjutnya tiap verteks pada stack yang berstatus sudah dikunjungi akan dijadikan 1 *strongly connected component*. Anggota *strongly connected component* adalah melakukan *DFS* verteks yang bisa menuju langsung verteks tersebut. Satu verteks hanya bisa berada pada satu *strongly connected component*. *Strongly connected component* pertama dimulai dari verteks yang terakhir dimasukkan ke dalam *stack*, yaitu verteks 1.

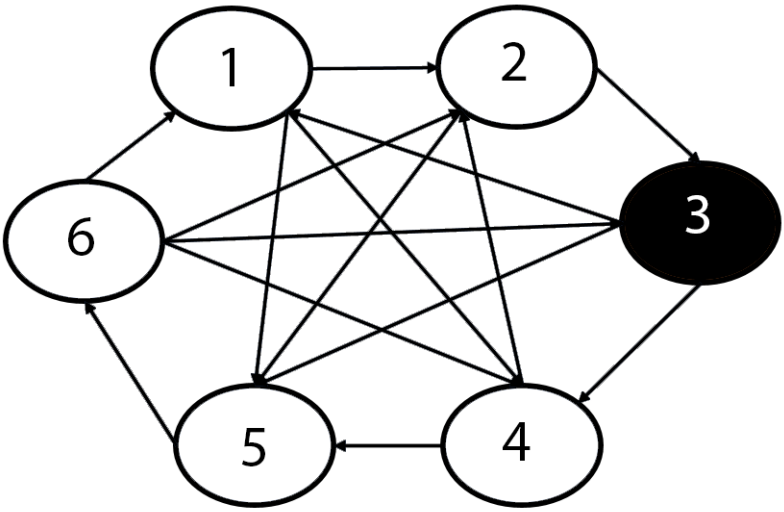


Gambar 5.18 Cek *Strongly Connected* (13)

Tabel 5.13 Mendapatkan *SCC* (1)

Verteks	**Status Dikunjungi**	**Urutan *Stack***	**Nomor *SCC***
1	belum dikunjungi	-	1
2	sudah dikunjungi	5	-
3	sudah dikunjungi	4	-
4	sudah dikunjungi	3	-
5	sudah dikunjungi	2	-
6	sudah dikunjungi	1	-

Status verteks 1 berubah menjadi belum dikunjungi. Verteks 1 masuk pada *SCC* pertama. Karena verteks 1 bisa dikunjungi langsung oleh verteks 3 dan verteks 3 sudah dikunjungi, maka selanjutnya menuju verteks 3.

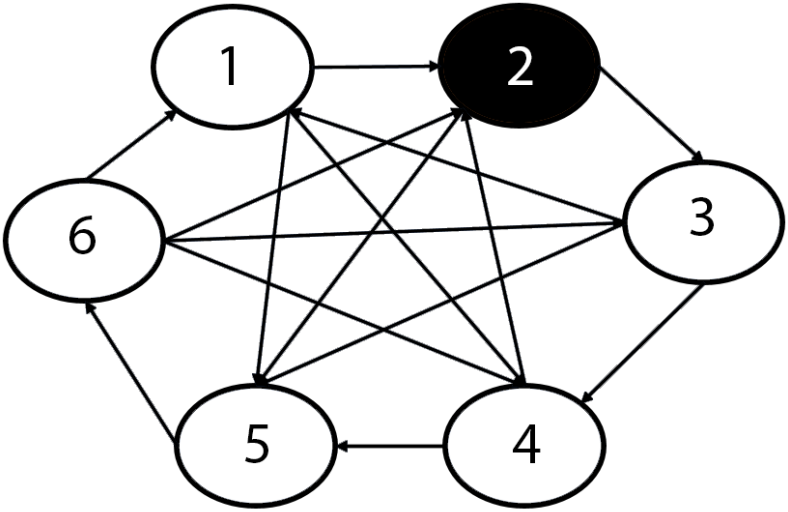


Gambar 5.19 Cek *Strongly Connected* (14)

Tabel 5.14 Mendapatkan *SCC* (2)

Verteks	**Status Dikunjungi**	**Urutan *Stack***	**Nomor *SCC***
1	belum dikunjungi	-	1
2	sudah dikunjungi	5	-
3	belum dikunjungi	4	1
4	sudah dikunjungi	3	-
5	sudah dikunjungi	2	-
6	sudah dikunjungi	1	-

Status verteks 3 berubah menjadi belum dikunjungi. Verteks 3 masuk pada *SCC* pertama. Karena verteks 3 bisa dikunjungi langsung oleh verteks 2 dan verteks 2 sudah dikunjungi, maka selanjutnya menuju verteks 2.

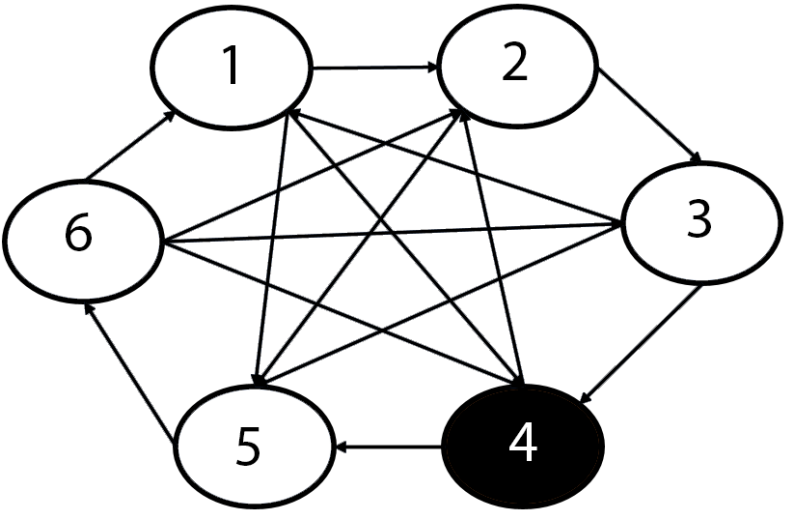


Gambar 5.20 Cek *Strongly Connected* (15)

Tabel 5.15 Mendapatkan *SCC* (3)

Verteks	Status Dikunjungi	Urutan *Stack*	Nomor *SCC*
1	belum dikunjungi	-	1
2	belum dikunjungi	5	1
3	belum dikunjungi	4	1
4	sudah dikunjungi	3	-
5	sudah dikunjungi	2	-
6	sudah dikunjungi	1	-

Status verteks 2 berubah menjadi belum dikunjungi. Verteks 2 masuk pada *SCC* pertama. Karena verteks 2 bisa dikunjungi langsung oleh verteks 4 dan verteks 4 sudah dikunjungi, maka selanjutnya menuju verteks 4.

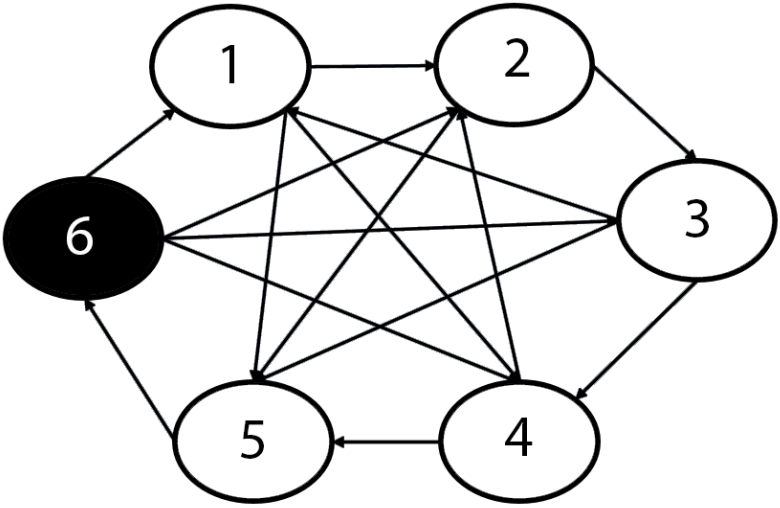


Gambar 5.21 Cek *Strongly Connected* (16)

Tabel 5.16 Mendapatkan *SCC* (4)

Verteks	**Status Dikunjungi**	**Urutan *Stack***	**Nomor *SCC***
1	belum dikunjungi	-	1
2	belum dikunjungi	5	1
3	belum dikunjungi	4	1
4	belum dikunjungi	3	1
5	sudah dikunjungi	2	-
6	sudah dikunjungi	1	-

Status verteks 4 berubah menjadi belum dikunjungi. Verteks 4 masuk pada *SCC* pertama. Karena verteks 4 bisa dikunjungi langsung oleh verteks 6 dan verteks 6 sudah dikunjungi, maka selanjutnya menuju verteks 6.

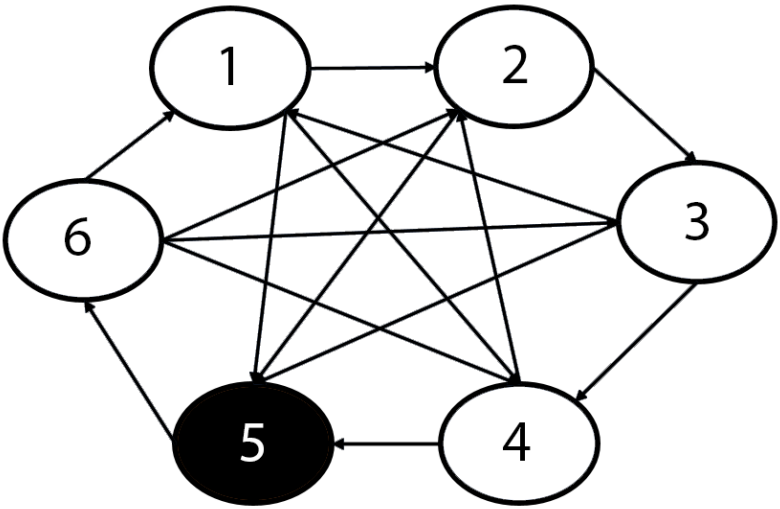


Gambar 5.22 Cek *Strongly Connected* (17)

Tabel 5.17 Mendapatkan *SCC* (5)

Verteks	Status Dikunjungi	Urutan *Stack*	Nomor *SCC*
1	belum dikunjungi	-	1
2	belum dikunjungi	5	1
3	belum dikunjungi	4	1
4	belum dikunjungi	3	1
5	sudah dikunjungi	2	-
6	belum dikunjungi	1	1

Status verteks 6 berubah menjadi belum dikunjungi. Verteks 6 masuk pada *SCC* pertama. Karena verteks 6 bisa dikunjungi langsung oleh verteks 5 dan verteks 5 sudah dikunjungi, maka selanjutnya menuju verteks 5.

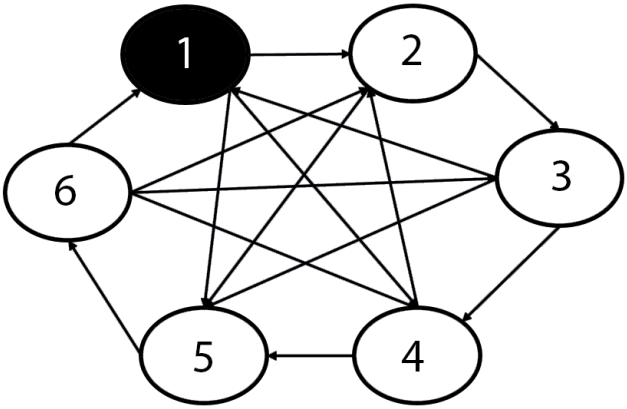


Gambar 5.23 Cek *Strongly Connected* (18)

Tabel 5.18 Mendapatkan *SCC* (6)

Verteks	**Status Dikunjungi**	**Urutan *Stack***	**Nomor *SCC***
1	belum dikunjungi	-	1
2	belum dikunjungi	5	1
3	belum dikunjungi	4	1
4	belum dikunjungi	3	1
5	belum dikunjungi	2	1
6	belum dikunjungi	1	1

Status verteks 5 berubah menjadi belum dikunjungi. Verteks 5 masuk pada *SCC* pertama. Karena semua verteks yang bisa langsung menuju verteks 5 berstatus belum dikunjungi, maka kembali ke verteks yang memanggil verteks 5, yaitu verteks 6, yang juga semua verteks yang bisa menujunya berstatus belum



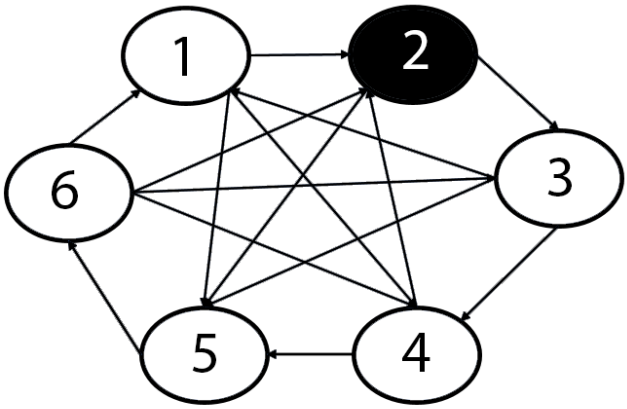
Gambar 5.24 *3-Vertex Cycle* (1)

dikunjungi. Begitu juga dengan verteks yang memanggil sebelumnya yaitu verteks 4, 2, 3, dan 1. Maka *strongly connected component* pertama berhasil dibentuk, dan *strongly connected component* selanjutnya akan dibentuk dari verteks yang masih berada pada *stack*. Tapi, karena semua verteks yang berada pada stack berstatus belum dikunjungi, tidak ada lagi *strongly connected component* yang bisa dibentuk. Maka, turnamen ini adalah *strongly connected*.

Turnamen yang *strongly connected* pasti memiliki *cycle* yang kita harapkan. Langkah pertama yang kita lakukan adalah membentuk *3-vertex cycle* dengan mencari verteks x yang bisa dituju verteks yang kita harapkan ada di semua *cycle*, dan menuju suatu verteks y yang bisa menuju verteks yang kita harapkan. Pada kasus ini, verteks yang kita harapkan untuk ada pada semua *cycle* adalah verteks 1.

Iterasi pertama diilustrasikan pada Gambar 5.24. Karena verteks 1 adalah verteks yang kita harapkan, tidak bisa dijadikan verteks x, maka lanjut ke iterasi selanjutnya.

Iterasi selanjutnya diilustrasikan pada Gambar 5.25. Karena



Gambar 5.25 *3-Vertex Cycle* (2)

verteks 2 bisa dituju oleh verteks 1, maka verteks 2 bisa dijadikan verteks x. Selanjutnya adalah mencari verteks y.

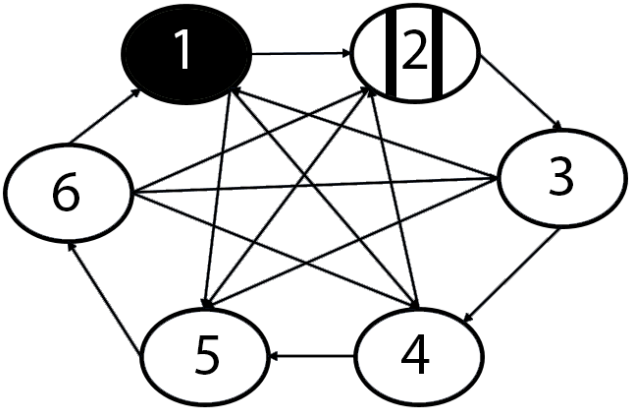
Iterasi selanjutnya diilustrasikan pada Gambar 5.26. Karena verteks 1 adalah verteks yang kita harapkan, tidak bisa dijadikan verteks y, maka lanjut ke iterasi selanjutnya.

Iterasi selanjutnya diilustrasikan pada Gambar 5.27. Karena verteks 2 adalah verteks x, tidak bisa dijadikan verteks y, maka lanjut ke iterasi selanjutnya.

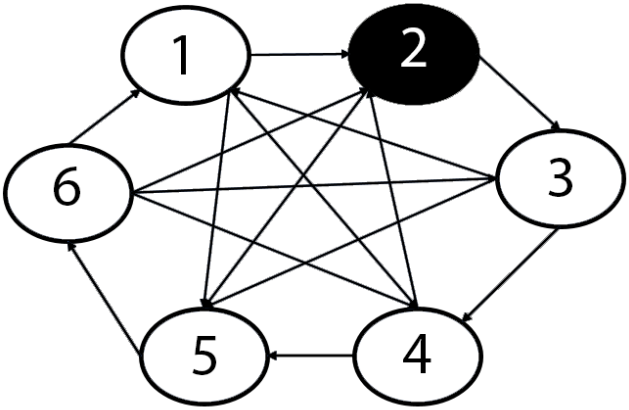
Iterasi pertama diilustrasikan pada Gambar 5.28. Karena verteks 3 bisa langsung dituju oleh verteks 2 dan bisa menuju verteks 1, maka verteks 3 bisa menjadi verteks y.

Setelah berhasil membentuk *3-vertex cycle*, kita harus mengategorikan verteks di luar *cycle* menjadi 3 kategori, yaitu:

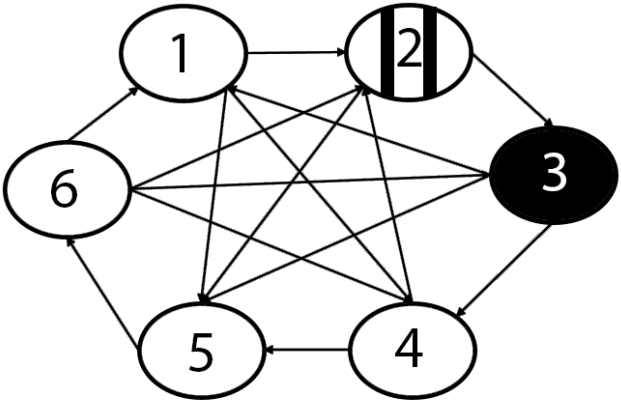
1. *BothCycle*, verteks yang bisa langsung menuju verteks pada *cycle* dan bisa langsung dituju dari verteks pada *cycle*
2. *FromCycle*, verteks yang hanya bisa langsung dituju dari verteks pada *cycle*
3. *ToCycle*, verteks yang hanya bisa menuju verteks pada *cycle*



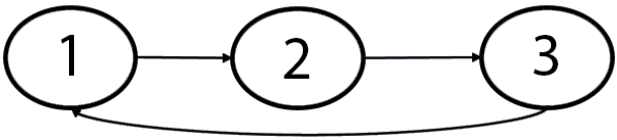
Gambar 5.26 *3-Vertex Cycle* (3)



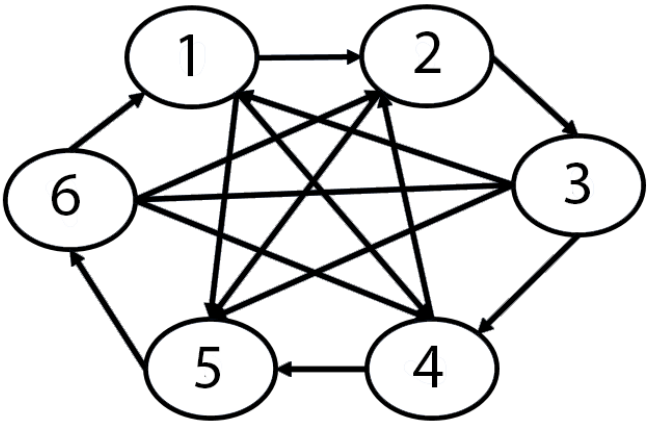
Gambar 5.27 *3-Vertex Cycle* (4)



Gambar 5.28 *3-Vertex Cycle* (5)



Gambar 5.29 *Cycle* (1)



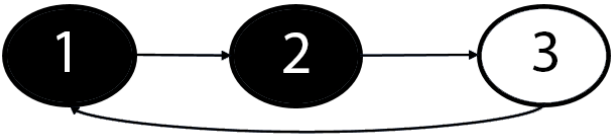
Gambar 5.30 Ilustrasi Input

Hasil dari pengategoriannya adalah sebagai berikut:

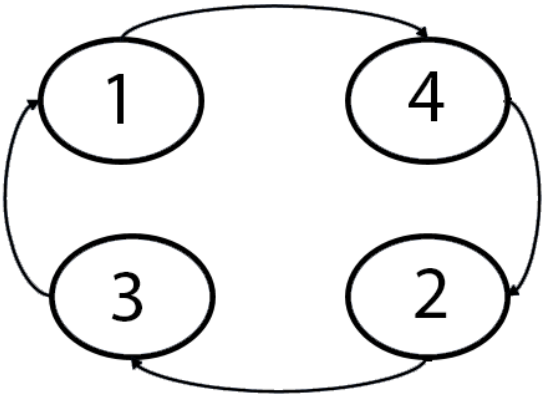
1. Verteks 4 masuk kategori *BothCycle*, karena bisa langsung dituju verteks 1 dan bisa langsung menuju verteks 2
2. Verteks 5 masuk kategori *FromCycle*, karena semua verteks pada *cycle*, bisa langsung menuju verteks 5
3. Verteks 6 masuk kategori *ToCycle*, karena semua bisa langsung menuju semua verteks pada *cycle*

Tabel 5.19 Kategori Verteks (1)

Verteks	Kategori Verteks	Urutan pada Kategori
1	Dalam *cycle*	1
2	Dalam *cycle*	2
3	Dalam *cycle*	3
4	*BothCycle*	1
5	*FromCycle*	1
6	*ToCycle*	1



Gambar 5.31 Memasukkan Verteks 4



Gambar 5.32 *Cycle* (2)

Cycle dengan 3 verteks sudah berhasil didapatkan. Selanjutnya adalah menambahkan isi *cycle* satu verteks terus-menerus sampai *cycle* berisi N verteks, dengan N adalah jumlah total verteks yang ada. Jika ada verteks yang kategori *BothCycle*, verteks tersebut masuk ke dalam *cycle* dengan berada di antara verteks x yang bisa menuju verteks tersebut dan verteks y yaitu verteks setelah x yang bisa dituju verteks tersebut.

Sesusai dengan Gambar 5.30, verteks 4 bisa dituju verteks 1, dan bisa menuju verteks 2. Maka, verteks 4 bisa dimasukkan ke dalam *cycle* di antara 2 verteks tersebut.

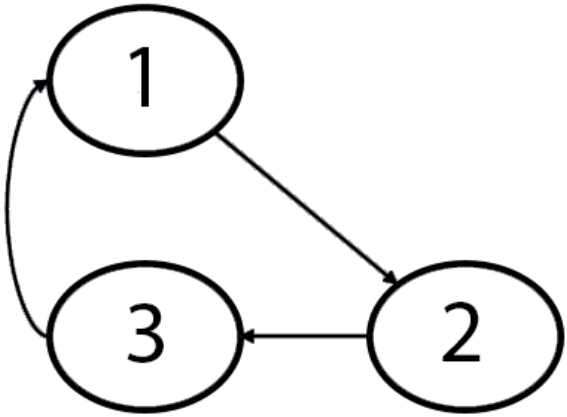
Tabel 5.20 Kategori Verteks (2)

Verteks	Kategori Verteks	Urutan pada Kategori
1	Dalam *cycle*	1
2	Dalam *cycle*	3
3	Dalam *cycle*	4
4	Dalam *cycle*	2
5	*FromCycle*	1
6	*ToCycle*	1

Setelah memasukkan verteks ke dalam *cycle*, dilakukan pengecekan dan perubahan jika ada verteks pada kategori *FromCycle* yang bisa menuju verteks yang baru saja dimasukkan, dan verteks pada kategori *ToCycle* yang bisa dituju verteks yang baru saja dimasukkan.

Verteks yang baru saja masuk *cycle* adalah verteks 4. Sesuai dengan Gambar 5.30, verteks 5 tidak bisa menuju verteks 4, dan verteks 6 tidak bisa dituju oleh verteks 4. Sehingga, tidak ada perubahan pada verteks dengan kategori *FromCycle* dan *ToCycle*.

Karena tidak ada lagi verteks dengan kategori *BothCycle*, maka verteks yang akan ditambahkan ada 2, yaitu dari *FromCycle* dan *ToCycle*. Karena besar verteks hanya boleh bertambah 1, maka dikeluarkan 1 verteks pada *cycle*. Verteks yang dikeluarkan diperbolehkan yang mana saja, kecuali verteks yang kita inginkan berada pada semua *cycle* dan nantinya akan diisi oleh verteks dari *FromCycle* dan *ToCycle*, dimana verteks *FromCycle* dan *ToCycle* bisa berada di sebelah verteks manapun, karena dituju dan menuju semua verteks pada *cycle*. Pada sistem ini, verteks yang dipilih untuk dikeluarkan adalah verteks kedua dalam *cycle*, pada saat ini yaitu verteks 4. Verteks yang dikeluarkan dari *cycle* pasti masuk kategori *BothCycle*, karena awalnya berada diantara verteks pada *cycle*.

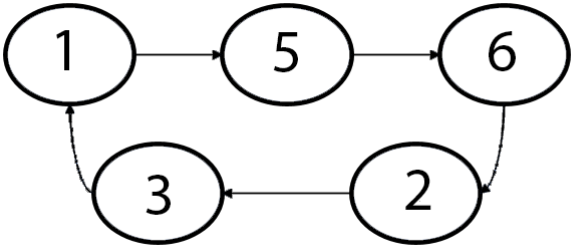


Gambar 5.33 *Cycle* (3)

Tabel 5.21 Kategori Verteks (3)

Verteks	**Kategori Verteks**	**Urutan pada Kategori**
1	Dalam *cycle*	1
2	Dalam *cycle*	2
3	Dalam *cycle*	3
4	*BothCycle*	1
5	*FromCycle*	1
6	*ToCycle*	1

Selanjutnya mencari verteks x pada *FromCycle* yang bisa menuju verteks y pada *ToCycle*. Sesuai dengan Gambar 5.30, verteks 5 bisa langsung menuju verteks 6.



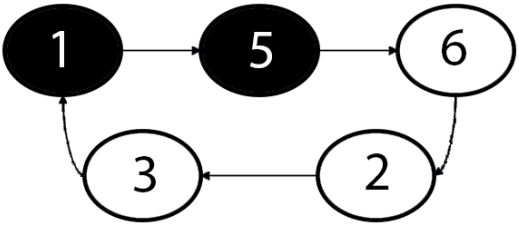
Gambar 5.34 *Cycle* (4)

Tabel 5.22 Kategori Verteks (4)

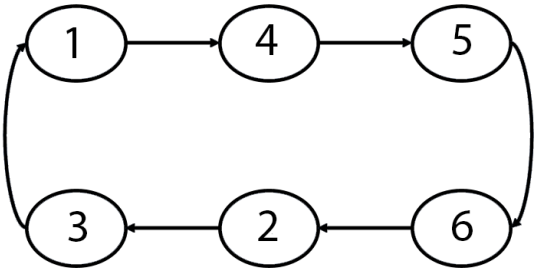
Verteks	Kategori Verteks	Urutan pada Kategori
1	Dalam *cycle*	1
2	Dalam *cycle*	4
3	Dalam *cycle*	5
4	*BothCycle*	1
5	Dalam *cycle*	2
6	Dalam *cycle*	3

Kemudian dilakukan pengecekan dan perubahan pada kategori *FromCycle* yang bisa menuju verteks yang baru dimasukkan, dan verteks pada kategori *ToCycle* yang bisa dituju verteks yang baru dimasukkan. Verteks yang baru masuk *cycle* adalah verteks 5 dan verteks 6. Tapi karena tidak ada verteks dengan kategori FromCycle dan BothCycle, maka tidak ada perubahan pada verteks dengan kategori *FromCycle* dan *ToCycle*.

Karena terdapat verteks dengan kategori *BothCycle*, maka verteks yang akan ditambahkan adalah verteks 4. Sesuai dengan Gambar 5.30, verteks 4 bisa dituju verteks 1, dan bisa menuju verteks 2. Maka, verteks 4 bisa dimasukkan ke dalam *cycle* di antara 2 verteks tersebut.

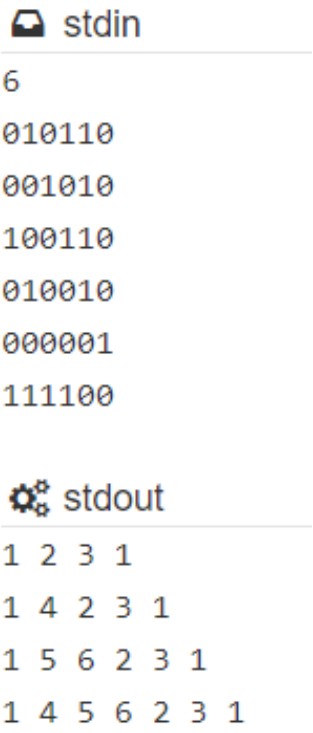


Gambar 5.35 Memasukkan Verteks 4



Gambar 5.36 *Cycle* (5)

Program dijalankan dan diberi masukan sesuai dengan masukan pada Gambar 5.30. Keluaran program telah sesuai dengan permasalahan, tercetak seperti Gambar 5.36. Jadi, program telah dibuat tepat mengeluarkan solusi dari pemasalahan CYCLERUN - Riding in Cycles.



Gambar 5.37 Input Output

BAB VI

KESIMPULAN DAN SARAN

Pada bab ini dijelaskan mengenai kesimpulan dan dari hasil uji coba yang telah dilakukan.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap perancangan desain algoritma untuk menyelesaikan permasalahan klasik SPOJ CYCLERUN - Riding in Cycles dapat diambil kesimpulan sebagai berikut:

1. Konsep *strongly connected* menggunakan Algoritma Kosaraju dan konsep *vertex pancyclic* dapat menyelesaikan permasalahan klasik SPOJ CYCLERUN - Riding in Cycles dengan benar.
2. Kompleksitas waktu sebesar $O(N^2)$ menyelesaikan permasalahan klasik SPOJ CYCLERUN - Riding in Cycles.
3. Waktu yang dibutuhkan program untuk menyelesaikan permasalahan klasik SPOJ CYCLERUN - Riding in Cycles minimum 0,12 detik, maksimum 0,18 detik dan rata-rata 0,1533 detik. Memori yang dibutuhkan adalah sebesar 17MB.

6.2 Saran

Saran yang diberikan adalah mengganti struktur data *list* yang mengimplementasikan *doubly-linked list* menjadi struktur data yang mengimplementasikan *single-linked list*. *Doubly-link list* menggunakan lebih banyak memori karena menyimpan alamat elemen setelahnya dan sebelumnya, dibandingkan dengan *single-linked list* yang hanya menyimpan alamat elemen setelahnya. Sehingga, memori yang akan digunakan akan lebih sedikit.

(Halaman ini sengaja dikosongkan)

DAFTAR PUSTAKA

[1] "SPOJ.com - Problem CYCLERUN." [Daring]. Tersedia pada: http://www.spoj.com/problems/CYCLERUN/. [Diakses: 11 Juli 2017].

[2] A. Levitin, *The Design & Analysis of Algorithms*, 3rd ed. Pearson Education, Inc., 2012.

[3] D. Paul Van, *Graph Theory and Applications*. Dublin: Self Published, 2009.

[4] "list - C++ Reference." [Daring]. Tersedia pada: http://www.cplusplus.com/reference/list/list/. [Diakses: 11 Juli 2017].

[5] "stack - C++ Reference." [Daring]. Tersedia pada: http://www.cplusplus.com/reference/stack/stack/. [Diakses: 11 Juli 2017].

[6] S. Halim dan F. Halim, *Competitive Programming, The New Lower Bound of Programming Contest*, 3rd ed. Singapore: Self Published, 2013.

[7] J. Xu, *Theory and Application of Graphs*. Springer Science & Business Media, Jul. 2003, google-Books-ID: pSnp4n0Nv3UC.

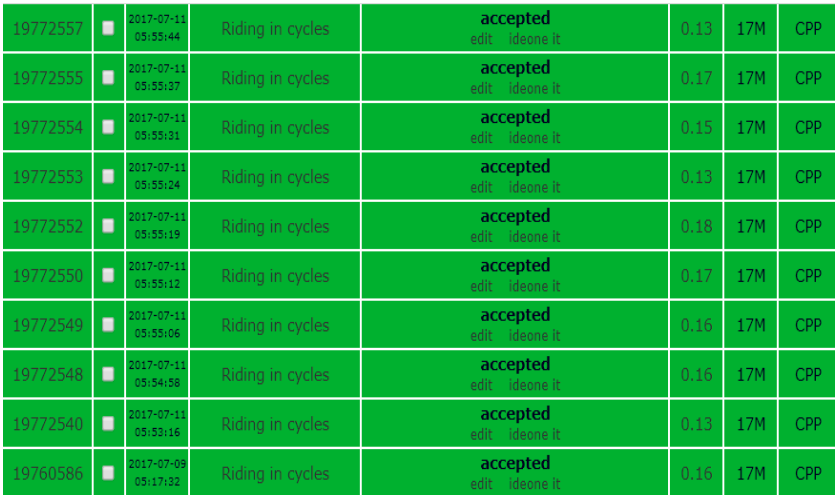
(Halaman ini sengaja dikosongkan)

LAMPIRAN A

UJI COBA

19772599		2017-07-11 06:02:58	Riding in cycles	accepted edit ideone it	0.12	17M	CPP
19772595		2017-07-11 06:02:05	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772594		2017-07-11 06:01:58	Riding in cycles	accepted edit ideone it	0.18	17M	CPP
19772593		2017-07-11 06:01:53	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772592		2017-07-11 06:01:46	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772590		2017-07-11 06:01:40	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772587		2017-07-11 06:01:34	Riding in cycles	accepted edit ideone it	0.15	17M	CPP
19772584		2017-07-11 05:59:58	Riding in cycles	accepted edit ideone it	0.12	17M	CPP
19772583		2017-07-11 05:59:53	Riding in cycles	accepted edit ideone it	0.16	17M	CPP
19772582		2017-07-11 05:59:47	Riding in cycles	accepted edit ideone it	0.13	17M	CPP
19772580		2017-07-11 05:59:40	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772579		2017-07-11 05:59:34	Riding in cycles	accepted edit ideone it	0.12	17M	CPP
19772578		2017-07-11 05:59:27	Riding in cycles	accepted edit ideone it	0.14	17M	CPP
19772571		2017-07-11 05:57:53	Riding in cycles	accepted edit ideone it	0.14	17M	CPP
19772570		2017-07-11 05:57:47	Riding in cycles	accepted edit ideone it	0.18	17M	CPP
19772569		2017-07-11 05:57:41	Riding in cycles	accepted edit ideone it	0.16	17M	CPP
19772568		2017-07-11 05:57:33	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772565		2017-07-11 05:57:28	Riding in cycles	accepted edit ideone it	0.14	17M	CPP
19772564		2017-07-11 05:57:22	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772563		2017-07-11 05:57:16	Riding in cycles	accepted edit ideone it	0.13	17M	CPP

Gambar A.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali(1)



19772557		2017-07-11 05:55:44	Riding in cycles	accepted edit ideone it	0.13	17M	CPP
19772555		2017-07-11 05:55:37	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772554		2017-07-11 05:55:31	Riding in cycles	accepted edit ideone it	0.15	17M	CPP
19772553		2017-07-11 05:55:24	Riding in cycles	accepted edit ideone it	0.13	17M	CPP
19772552		2017-07-11 05:55:19	Riding in cycles	accepted edit ideone it	0.18	17M	CPP
19772550		2017-07-11 05:55:12	Riding in cycles	accepted edit ideone it	0.17	17M	CPP
19772549		2017-07-11 05:55:06	Riding in cycles	accepted edit ideone it	0.16	17M	CPP
19772548		2017-07-11 05:54:58	Riding in cycles	accepted edit ideone it	0.16	17M	CPP
19772540		2017-07-11 05:53:16	Riding in cycles	accepted edit ideone it	0.13	17M	CPP
19760586		2017-07-09 05:17:32	Riding in cycles	accepted edit ideone it	0.16	17M	CPP

Gambar A.2 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali(2)

Tabel A.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali

No	Hasil	Waktu	Memori
1	*Accepted*	0,12 detik	17 MB
2	*Accepted*	0,17 detik	17 MB
3	*Accepted*	0,18 detik	17 MB
4	*Accepted*	0,17 detik	17 MB
5	*Accepted*	0,17 detik	17 MB
6	*Accepted*	0,17 detik	17 MB
7	*Accepted*	0,15 detik	17 MB
8	*Accepted*	0,12 detik	17 MB
9	*Accepted*	0,16 detik	17 MB
10	*Accepted*	0,13 detik	17 MB
11	*Accepted*	0,17 detik	17 MB
12	*Accepted*	0,12 detik	17 MB
13	*Accepted*	0,14 detik	17 MB

Tabel A.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 30 kali

No	Hasil	Waktu	Memori
14	*Accepted*	0,14 detik	17 MB
15	*Accepted*	0,18 detik	17 MB
16	*Accepted*	0,16 detik	17 MB
17	*Accepted*	0,17 detik	17 MB
18	*Accepted*	0,14 detik	17 MB
19	*Accepted*	0,17 detik	17 MB
20	*Accepted*	0,13 detik	17 MB
21	*Accepted*	0,13 detik	17 MB
22	*Accepted*	0,17 detik	17 MB
23	*Accepted*	0,15 detik	17 MB
24	*Accepted*	0,13 detik	17 MB
25	*Accepted*	0,18 detik	17 MB
26	*Accepted*	0,17 detik	17 MB
27	*Accepted*	0,16 detik	17 MB
28	*Accepted*	0,16 detik	17 MB
29	*Accepted*	0,13 detik	17 MB
30	*Accepted*	0,16 detik	17 MB

(Halaman ini sengaja dikosongkan)

BIODATA PENULIS

Daniel Bintar Doharto Sagala Raja Sijabat, akrab dipanggil Daniel lahir di Bandung pada tanggal 31 Desember 1994. Penulis adalah anak pertama dari empat bersaudara. Penulis menempuh pendidikan formal di SD Santa Maria Cimahi, SMPN 1 Cimahi, SMAN 9 Bandung dan S1 Teknik Informatika FTIF ITS.

Penulis terlibat aktif dalam organisasi kemahasiswaan sebagai staff Riset dan Teknologi di Himpunan Mahasiswa Teknik Computer-Informatika (HMTC) ITS. Penulis juga aktif dalam kegiatan kepanitiaan Schematics sebagai staff National Progamming Contest (NPC) Schematics 2014. Penulis dapat dihubungi melalui surat elektronik daniel.bintar@gmail.com.