Teori-teori *graph* di bawah merupakan kumpulan definisi dan teorema berdasarkan subbab 2.2.1 dan 2.2.2.

## A.1. Pengenalan *Graph*

**Definisi 2.1.1**
Sebuah *graph* sederhana $G = (V, E)$ terdiri dari $V$, sebuah himpunan tidak kosong dari verteks-verteks, dan $E$, sebuah daftar pasangan tidak berurut dari elemen-elemen berbeda dari $V$ disebut sisi.

**Definisi 2.1.2**
Sebuah *multigraph* $G = (V, E)$ terdiri dari sebuah himpunan $V$ dari verteks-verteks, sebuah daftar $E$ dari sisi-sisi, dan sebuah fungsi $f$ dari $E$ ke $\{\{u, v\} \mid u, v \in V, u \neq v\}$. Sisi $e_1$ dan $e_2$ disebut sisi paralel jika $f(e_1) = f(e_2)$.

**Definisi 2.1.3**
Sebuah *pseudograph* $G = (V, E)$ terdiri dari sebuah himpunan $V$ dari verteks-verteks, sebuah daftar $E$ dari sisi-sisi, dan sebuah fungsi $f$ dari $E$ ke $\{\{u, v\} \mid u, v \in V\}$. Sebuah sisi disebut sebuah putaran jika $f(e) = \{u, u\} = \{u\}$ untuk sebagian $u \in V$.

**Definisi 2.1.4**
Sebuah *graph* berarah $(V, E)$ terdiri dari sebuah himpunan verteks-verteks $V$ dan sebuah daftar sisi-sisi $E$ yang pasangan-pasangannya berurut dari elemen-elemen dari $V$.

**Definisi 2.1.5**
Sebuah *multigraph* berarah $G = (V, E)$ terdiri dari sebuah himpunan $V$ dari verteks-verteks, sebuah daftar $E$ dari sisi-sisi, dan sebuah fungsi $f$ dari $E$ ke $\{\{u, v\} \mid u, v \in V\}$. Sisi-sisi $e_1$ dan $e_2$ merupakan sisi paralel jika $f(e_1) = f(e_2)$.

## A.2. Terminologi *Graph*

**Definisi 2.2.1**
Dua buah verteks $u$ dan $v$ pada sebuah *graph* tidak berarah $G$ disebut berdekatan (atau bertetangga) di $G$ jika $\{u, v\}$ adalah sebuah sisi dari $G$. Jika $e = \{u, v\}$, sisi $e$ disebut kejadian antara verteks $u$ dan $v$. Sisi $e$ juga dikatakan penghubung $u$ dan $v$. Verteks $u$ dan $v$ disebut titik-titik akhir dari sisi $\{u, v\}$.

**Definisi 2.2.2**
Derajat dari sebuah verteks pada sebuah *graph* tidak berarah adalah banyak sisi-sisi yang terjadi dengannya, kecuali bahwa sebuah putaran pada sebuah verteks memberikan dua kali ke derajat dari verteks itu. Derajat dari verteks $v$ dinotasikan oleh deg($v$).

**Teorema 2.2.1 (Teorema Jabat Tangan)**
Misalkan $G = (V, E)$ merupakan sebuah *graph* tidak berarah dengan $e$ sisi. Maka

$$2e = \sum_{v \in V} \deg(v)$$

**Definisi 2.2.3**
Jika $(u, v)$ adalah sebuah sisi dari *graph* $G$ dengan sisi-sisi berarah, $u$ dikatakan berdekatan ke $v$ dan $v$ dikatakan berdekatan dari $u$. Verteks $u$ dikatakan verteks mula-mula dari $(u, v)$, dan $v$ dikatakan verteks pangkalan atau akhir dari $(u, v)$. Verteks mula-mula dan verteks pangkalan dari sebuah putaran adalah sama.

**Definisi 2.2.4**
Pada sebuah *graph* dengan sisi-sisi berarah, derajat-dalam dari sebuah verteks $v$, dinotasikan oleh $\deg^-(v)$, merupakan banyak sisi dengan $v$ sebagai verteks pangkalannya. Derajat-luar dari $v$, dinotasikan oleh $\deg^+(v)$, merupakan banyak sisi dengan $v$ sebagai verteks mula-mulanya. (Perhatikan bahwa sebuah putaran pada sebuah verteks memberikan 1 ke derajat-dalam dan derajat-luar dari verteks ini.)

**Definisi 2.2.5**

Sebuah *graph* sederhana $G$ disebut dua-pihak jika himpunan verteks $V$ dapat dipartisi ke dalam dua himpunan saling lepas $V_1$ dan $V_2$ sedemikian sehingga setiap sisi pada *graph* menghubungkan sebuah verteks di $V_1$ dan sebuah verteks di $V_2$ (sehingga tidak ada sisi pada $G$ menghubungkan dua verteks di $V_1$ atau dua verteks di $V_2$).

**Definisi 2.2.6**

Sebuah *graph*-bagian dari sebuah *graph* $G = (V, E)$ adalah *graph* $H = (W, F)$ di mana $W \subseteq V$ dan $F$ subdaftar dari $E$.

**Definisi 2.2.7**

Gabungan dari dua *graph* sederhana $G_1 = (V_1, E_1)$ dan $G_2 = (V_2, E_2)$ adalah *graph* sederhana dengan himpunan verteks $V_1 \cup V_2$ dan daftar sisi $E_1 + E_2$. Gabungan $G_1$ dan $G_2$ dinotasikan oleh $G_1 \cup G_2$.

## A.3. Isomorfisme dari *Graph*

**Definisi 2.3.1**

*Graph* sederhana $G_1 = (V_1, E_1)$ dan $G_2 = (V_2, E_2)$ adalah isomorfik jika terdapat sebuah fungsi $f$ bersifat satu-satu dan pada dari $V_1$ ke $V_2$ dengan sifat bahwa $a$ dan $b$ berdekatan di $G_1$ jika dan hanya jika $f(a)$ dan $f(b)$ berdekatan di $G_2$, untuk setiap $a$ dan $b$ di $V_1$. Fungsi $f$ tersebut disebut isomorfisme.

```
(* ========================================================================== *)
(* Graph Theory                                                               *)
(*  based on "Discrete Mathematics And Its Applications 5ed"                  *)
(*  by Kenneth H. Rosen.                                                      *)
(*                                                                            *)
(* This theory is formalized by Ricky Suryadharma                             *)
(*  using HOL-4 [Kananaskis 4 (built January 02, 2007)].                      *)
(*                                                                            *)
(* Date: December 01, 2008                                                    *)
(*                                                                            *)
(* All definition's and theorem's numbers based on                            *)
(*  "Experiment on Formalization in HOL System with Case Study Graph Theory"  *)
(*  Chapter 2                                                                 *)
(* ========================================================================== *)



structure graphTheory =
struct


(* interactive use
app load ["pred_setTheory", "stringTheory", "listTheory"];
*)

open HolKernel Parse boolLib Prim_rec bossLib numLib
        numTheory pairTheory pred_setTheory prim_recTheory stringTheory listTheory;


val ARW_TAC = RW_TAC arith_ss;
val BRW_TAC = RW_TAC bool_ss;
val SRW_TAC = RW_TAC std_ss;



(* ------------------------------------------------------------------------- *)
(* Create the new theory.                                                    *)
(* ------------------------------------------------------------------------- *)

val _ = new_theory "graph";
```

```
(* ========================================================================== *)
(* Data structure                                                             *)
(* ========================================================================== *)

(* -------------------------------------------------------------------------- *)
(* Vertex                                                                     *)
(* Usage: Vertex_ <vertex_number>                                             *)
(* Example in term: Vertex_ 7                                                 *)
(* -------------------------------------------------------------------------- *)

val VERTEX = Hol_datatype
        `Vertex = Vertex_ of num`;



(* -------------------------------------------------------------------------- *)
(* Edge                                                                       *)
(* Usage:                                                                     *)
(*  Undirected ((<v1>, <v2>), <weight>)                                       *)
(*   or                                                                       *)
(*  Directed ((<init_vertex>, <terminal_vertex>), <weight>)                   *)
(* Example in term: Undirected ((Vertex_ 2, Vertex_ 5), 100)                  *)
(* -------------------------------------------------------------------------- *)

val EDGE = Hol_datatype
        `
                Edge =
                        Undirected of (Vertex # Vertex) # num |
                        Directed of (Vertex # Vertex) # num
        `;


(* -------------------------------------------------------------------------- *)
(* Graph                                                                      *)
(* Example in term: Graph_ (                                                  *)
(*                          {Vertex_ 3; Vertex_ 5; Vertex_ 7},                *)
(*                          [                                                 *)
(*                           Directed ((Vertex_ 3, Vertex_ 5), 132);          *)
(*                           Directed ((Vertex_ 5, Vertex_ 3),  50);          *)
(*                           Directed ((Vertex_ 3, Vertex_ 7), 100)           *)
(*                          ]                                                 *)
(*                         )                                                  *)
(* -------------------------------------------------------------------------- *)

val GRAPH = Hol_datatype
        `Graph = Graph_ of ((Vertex set) # (Edge list))`;
```

```
(* ------------------------------------------------------------------------- *)
(* Useful functions.                                                         *)
(* ------------------------------------------------------------------------- *)

val GET_VERTICES = Define
        `
                getVertices (Graph_ (V, E)) =
                        V
        `;


val GET_EDGES = Define
        `
                getEdges (Graph_ (V, E)) =
                        E
        `;


val SUBLIST = Define
        `
                sublist l1 l2 =
                        (l1 = [])
                        \/
                        (
                                !m.
                                        (MEM m l1)
                                        ==>
                                        (MEM m l2)
                        )
        `;



(* ========================================================================= *)
(* 2.1. Introduction to Graphs                                               *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Definition 2.1.1                                                          *)
(* ------------------------------------------------------------------------- *)

val def211_1 = Define
        `
                (undirected_simple_e_in_setV (Undirected ((v1, v2), _)) V) =
                        (v1 IN V)
                        /\
                        (v2 IN V)
        `;
```

```
val def211_2 = Define
        `
                (undirected_simple_diff_v_in_e (Undirected ((v1, v2), _))) =
                        ~(v1 = v2)
        `;


val def211_3 = Define
        `
                (undirected_simple_diff_e (Undirected ((v1, v2), _)) (Undirected ((v3,
v4), _))) =
                        (
                                ~(v1 = v3)
                                \/
                                ~(v2 = v4)
                        )
                        /\
                        (
                                ~(v1 = v4)
                                \/
                                ~(v2 = v3)
                        )
        `;

val def211_4 = Define
        `
                (undirected_simple_diff_ht_e headE tailE) =
                        !e.
                                (MEM e tailE)
                                ==>
                                (undirected_simple_diff_e headE e)
        `;


val def211_5 = new_recursive_definition
        {
                name = "undirected_simple_diff_allE",
                rec_axiom = list_Axiom,
                def =
                        --`
                                (undirected_simple_diff_allE [] = T)
                                /\
                                (
                                        undirected_simple_diff_allE (h::t) =
                                                (undirected_simple_diff_ht_e h t)
```

```
                                        /\
                                        (undirected_simple_diff_allE t)
                                )
                        `--
        };


val UNDIRECTED_SIMPLE_GRAPH = Define
        `
                Undirected_Simple_Graph G =
                        ~(getVertices G = {})
                        /\
                        (
                                !e.
                                        (MEM e (getEdges G))
                                        ==>
                                        (
                                        (undirected_simple_e_in_setV e
(getVertices G))
                                        /\
                                        (undirected_simple_diff_v_in_e e)
                                        )
                        )
                        /\
                        (undirected_simple_diff_allE (getEdges G))
        `;


val SIMPLE_GRAPH = Define
        `
                Simple_Graph G =
                        (Undirected_Simple_Graph G)
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.1.2                                                          *)
(* ------------------------------------------------------------------------- *)

val def212_1 = Define
        `
                (undirected_multi_e_in_setV (Undirected ((v1, v2), _)) V) =
                        (v1 IN V)
                        /\
                        (v2 IN V)
        `;
```

```
val def212_2 = Define
        `
                (undirected_multi_diff_v_in_e (Undirected ((v1, v2), _))) =
                        ~(v1 = v2)
        `;


val def212_3 = Define
        `
                (undirected_paralel (Undirected ((v1, v2), _)) (Undirected ((v3, v4),
_))) =
                        (
                                ~(v1 = v2)
                                /\
                                ~(v3 = v4)
                        )
                        /\
                        (
                                (
                                        (v1 = v3)
                                        /\
                                        (v2 = v4)
                                )
                                \/
                                (
                                        (v1 = v4)
                                        /\
                                        (v2 = v3)
                                )
                        )
        `;


val UNDIRECTED_MULTI_GRAPH = Define
        `
                Undirected_Multi_Graph G =
                        (
                                !e.
                                        (MEM e (getEdges G))
                                        ==>
                                        (
                                                (undirected_multi_e_in_setV e
(getVertices G))
                                                /\
                                                (undirected_multi_diff_v_in_e e)
                                        )
                        )
```

```
        `;


(*
Cut out because multi-edges allowed means
there might exists multi-edges or might be no multi-edges.

                        /\
                        (
                                ?e1 e2.
                                        (MEM e1 (getEdges G))
                                        /\
                                        (MEM e2 (getEdges G))
                                        /\
                                        (undirected_paralel e1 e2)
                        )
*)


val MULTI_GRAPH = Define
        `
                Multi_Graph G =
                        (Undirected_Multi_Graph G)
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.1.3                                                          *)
(* ------------------------------------------------------------------------- *)

val def213_1 = Define
        `
                (undirected_pseudo_e_in_setV (Undirected ((v1, v2), _)) V) =
                        (v1 IN V)
                        /\
                        (v2 IN V)
        `;


val def213_2 = Define
        `
                undirected_pseudo (Undirected ((v1, v2), _)) =
                        (v1 = v2)
        `;


val UNDIRECTED_PSEUDO_GRAPH = Define
```

```
        `
                Undirected_Pseudo_Graph G =
                        (
                                !e.
                                        (MEM e (getEdges G))
                                        ==>
                                        (undirected_pseudo_e_in_setV e (getVertices G))
                        )
        `;


(*
Cut out because loops allowed means
there might exists loops or might be no loop.

                        /\
                        (
                                ?e.
                                        (MEM e (getEdges G))
                                        /\
                                        (undirected_pseudo e)
                        )
*)


val PSEUDO_GRAPH = Define
        `
                Pseudo_Graph G =
                        (Undirected_Pseudo_Graph G)
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.1.4                                                          *)
(* ------------------------------------------------------------------------- *)

val def214_1 = Define
        `
                (directed_simple_e_in_setV (Directed ((v1, v2), _)) V) =
                        (v1 IN V)
                        /\
                        (v2 IN V)
        `;


val def214_2 = Define
        `
```

```
                (directed_simple_diff_e (Directed ((v1, v2), _)) (Directed ((v3, v4),
_))) =
                        (
                                ~(v1 = v3)
                                \/
                                ~(v2 = v4)
                        )
        `;


val def214_3 = Define
        `
                (directed_simple_diff_ht_e headE tailE) =
                        !e.
                                (MEM e tailE)
                                ==>
                                (directed_simple_diff_e headE e)
        `;


val def214_4 = new_recursive_definition
        {
                name = "directed_simple_diff_allE",
                rec_axiom = list_Axiom,
                def =
                        --`
                                (directed_simple_diff_allE [] = T)
                                /\
                                (
                                        directed_simple_diff_allE (h::t) =
                                                (directed_simple_diff_ht_e h t)
                                                /\
                                                (directed_simple_diff_allE t)
                                )
                        `--
        };


val DIRECTED_SIMPLE_GRAPH = Define
        `
                Directed_Simple_Graph G =
                        (
                                !e.
                                        (MEM e (getEdges G))
                                        ==>
                                        (directed_simple_e_in_setV e (getVertices G))
                        )
                        /\
```

```
                        (directed_simple_diff_allE (getEdges G))
        `;



(* ------------------------------------------------------------------------------ *)
(* Definition 2.1.5                                                               *)
(* ------------------------------------------------------------------------------ *)

val def215_1 = Define
        `
                (directed_multi_e_in_setV (Directed ((v1, v2), _)) V) =
                        (v1 IN V)
                        /\
                        (v2 IN V)
        `;


val def215_2 = Define
        `
                (directed_paralel (Directed ((v1, v2), _)) (Directed ((v3, v4), _))) =
                        (
                                (v1 = v3)
                                /\
                                (v2 = v4)
                        )
                        \/
                        (
                                (v1 = v4)
                                /\
                                (v2 = v3)
                        )
        `;


val DIRECTED_MULTI_GRAPH = Define
        `
                Directed_Multi_Graph G =
                        (
                                !e.
                                        (MEM e (getEdges G))
                                        ==>
                                        (directed_multi_e_in_setV e (getVertices G))
                        )
        `;
```

```
(* ------------------------------------------------------------------------- *)
(* The Other Graph's Definitions                                             *)
(* ------------------------------------------------------------------------- *)

val def21Z_1 = Define
        `
                (undirected_e_in_setV (Undirected ((v1, v2), _)) V) =
                        (v1 IN V)
                        /\
                        (v2 IN V)
        `;


val UNDIRECTED_GRAPH = Define
        `
                Undirected_Graph G =
                        !e.
                                (MEM e (getEdges G))
                                ==>
                                (undirected_e_in_setV e (getVertices G))
        `;

val def21Z_2 = Define
        `
                (directed_e_in_setV (Directed ((v1, v2), _)) V) =
                        (v1 IN V)
                        /\
                        (v2 IN V)
        `;


val DIRECTED_GRAPH = Define
        `
                Directed_Graph G =
                        !e.
                                (MEM e (getEdges G))
                                ==>
                                (directed_e_in_setV e (getVertices G))
        `;


val COMMON_GRAPH = Define
        `
                Common_Graph G =
                        (Undirected_Graph G)
                        \/
                        (Directed_Graph G)
```

```
        `;



(* ========================================================================== *)
(* 2.2. Graph Terminology                                                     *)
(* ========================================================================== *)

(* -------------------------------------------------------------------------- *)
(* Definition 2.2.1                                                           *)
(* -------------------------------------------------------------------------- *)

val def221_1 = Define
        `
                (undirected_connect u v (Undirected ((w1, w2), _))) =
                        (
                                (u = w1)
                                /\
                                (v = w2)
                        )
                        \/
                        (
                                (u = w2)
                                /\
                                (v = w1)
                        )
        `;


val UNDIRECTED_ADJACENT = Define
        `
                Undirected_Adjacent u v G =
                        (Undirected_Graph G)
                        /\
                        (u IN (getVertices G))
                        /\
                        (v IN (getVertices G))
                        /\
                        (
                                ?e.
                                        (MEM e (getEdges G))
                                        /\
                                        (undirected_connect u v e)
                        )
        `;


val END_POINT = Define
```

```
        `
                (end_point u (Undirected ((v1, v2), _))) =
                        (u = v1)
                        \/
                        (u = v2)
        `;


val END_POINT_LOOP = Define
        `
                (end_point_loop u (Undirected ((v1, v2), _))) =
                        (v1 = v2)
                        /\
                        (u = v1)
        `;



(* ------------------------------------------------------------------------- *)
(* Theorem 2.2.1                                                             *)
(* ------------------------------------------------------------------------- *)

val NUMBER_OF_EDGES = Define
        `
                number_of_edges G =
                        LENGTH (getEdges G)
        `;


val SUM_OF_DEGREE = Define
        `
                (sum_of_degree [] = 0)
                /\
                (
                        sum_of_degree (h::t) = SUC (SUC (sum_of_degree t))
                )
        `;


fun funcTrue _ = true;


val HANDSHAKING_THM =
        store_thm
        (
                "HANDSHAKING_THM",
                Term `!G. Undirected_Graph G ==> ((2 * (number_of_edges G)) =
(sum_of_degree (getEdges G)))`,
```

```
                REPEAT STRIP_TAC THEN
                WEAKEN_TAC funcTrue THEN
                Cases_on `G` THEN
                Cases_on `p` THEN
                REWRITE_TAC[NUMBER_OF_EDGES, GET_EDGES] THEN
                Induct_on `r` THEN
                REPEAT (ARW_TAC[LENGTH, SUM_OF_DEGREE])
        );



(* ------------------------------------------------------------------------- *)
(* Definition 2.2.2                                                          *)
(* ------------------------------------------------------------------------- *)

val def222_1 = new_recursive_definition
        {
                name = "count_degree",
                rec_axiom = list_Axiom,
                def =
                        --`
                                (count_degree v [] = 0)
                                /\
                                (
                                        count_degree v (h::t) =
                                                if (end_point_loop v h) then 2 +
(count_degree v t)
                                                else if (end_point v h) then 1 +
(count_degree v t)
                                                else (count_degree v t)
                                )
                        `--
        };


val DEGREE = Define
        `
                Degree v G =
                        if ((Undirected_Graph G) /\ (v IN (getVertices G))) then
(count_degree v (getEdges G))
                        else (0 - 1)
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.2.3                                                          *)
(* ------------------------------------------------------------------------- *)
```

```
val def223_1 = Define
        `
                (directed_connect u v (Undirected ((w1, w2), _))) =
                        (u = w1)
                        /\
                        (v = w2)
        `;


val DIRECTED_ADJACENT = Define
        `
                Directed_Adjacent u v G =
                        (Directed_Graph G)
                        /\
                        (u IN (getVertices G))
                        /\
                        (v IN (getVertices G))
                        /\
                        (
                                ?e.
                                        (MEM e (getEdges G))
                                        /\
                                        (directed_connect u v e)
                        )
        `;


val def223_2 = Define
        `
                (adjacent_to_v_e u (Directed ((v1, _), _))) =
                        (u = v1)
        `;


val ADJACENT_TO = Define
        `
                (adjacent_to u e G) =
                        (Directed_Graph G)
                        /\
                        (u IN (getVertices G))
                        /\
                        (MEM e (getEdges G))
                        /\
                        (adjacent_to_v_e u e)
        `;
```

```
val def223_3 = Define
        `
                (adjacent_from_v_e u (Directed ((_, v2), _))) =
                        (u = v2)
        `;


val ADJACENT_FROM = Define
        `
                (adjacent_from u e G) =
                        (Directed_Graph G)
                        /\
                        (u IN (getVertices G))
                        /\
                        (MEM e (getEdges G))
                        /\
                        (adjacent_from_v_e u e)
        `;


val INITIAL_VERTEX = Define
        `
                (initial_vertex u (Directed ((v1, _), _))) =
                        (u = v1)
        `;


val TERMINAL_VERTEX = Define
        `
                (terminal_vertex u (Directed ((_, v2), _))) =
                        (u = v2)
        `;


val SAME_VERTICES_IN_LOOP = Define
        `
                same_vertices_in_loop (Directed ((v1, v2), _)) =
                        (v1 = v2)
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.2.4                                                          *)
(* ------------------------------------------------------------------------- *)

val def224_1 = new_recursive_definition
        {
```

```
                name = "count_indegree",
                rec_axiom = list_Axiom,
                def =
                        --`
                                (count_indegree v [] = 0)
                                /\
                                (
                                        count_indegree v (h::t) =
                                                if (initial_vertex v h) then 1 +
(count_indegree v t)
                                                else (count_indegree v t)
                                )
                        `--
        };


val IN_DEGREE = Define
        `
                In_Degree v G =
                        if ((Directed_Graph G) /\ (v IN (getVertices G))) then
(count_indegree v (getEdges G))
                        else (0 - 1)
        `;


val def224_2 = new_recursive_definition
        {
                name = "count_outdegree",
                rec_axiom = list_Axiom,
                def =
                        --`
                                (count_outdegree v [] = 0)
                                /\
                                (
                                        count_outdegree v (h::t) =
                                                if (terminal_vertex v h) then 1 +
(count_outdegree v t)
                                                else (count_outdegree v t)
                                )
                        `--
        };


val OUT_DEGREE = Define
        `
                Out_Degree v G =
                        if ((Directed_Graph G) /\ (v IN (getVertices G))) then
(count_outdegree v (getEdges G))
```

```
                        else (0 - 1)
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.2.5                                                          *)
(* ------------------------------------------------------------------------- *)

val def225_1 = Define
        `
                (bipartite_e_in_V1_V2 (Undirected ((u1, u2), _)) V1 V2) =
                        (u1 IN V1)
                        /\
                        (u2 IN V2)
        `;


val BIPARTITE = Define
        `
                Bipartite G =
                        (Simple_Graph G)
                        /\
                        (
                                ?V1 V2.
                                        (V1 SUBSET (getVertices G))
                                        /\
                                        (V2 SUBSET (getVertices G))
                                        /\
                                        ((V1 UNION V2) = (getVertices G))
                                        /\
                                        (DISJOINT V1 V2)
                                        /\
                                        (
                                                !e.
                                                        (MEM e (getEdges G))
                                                        ==>
                                                        (bipartite_e_in_V1_V2 e V1 V2)
                                        )
                        )
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.2.6                                                          *)
(* ------------------------------------------------------------------------- *)
```

```
val SUBSET_GRAPH = Define
        `
                Subset_Graph H G =
                        ((getVertices H) SUBSET (getVertices G))
                        /\
                        (sublist (getEdges H) (getEdges G))
        `;



(* ------------------------------------------------------------------------- *)
(* Definition 2.2.7                                                          *)
(* ------------------------------------------------------------------------- *)

val UNION_GRAPH = Define
        `
                Union_Graph G1 G2 G =
                        (((getVertices G1) UNION (getVertices G2)) = (getVertices G))
                        /\
                        ((APPEND (getEdges G1) (getEdges G2)) = (getEdges G))
        `;



(* ========================================================================= *)
(* 2.3. Graph Isomorphism                                                    *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Definition 2.3.1                                                          *)
(* ------------------------------------------------------------------------- *)

val ISOMORPHIC = Define
        `
                Isomorphic G1 G2 =
                        (Simple_Graph G1)
                        /\
                        (Simple_Graph G2)
                        /\
                        (
                                ?f.
                                        (BIJ f (getVertices G1) (getVertices G2))
                                        /\
                                        (
                                                !a b.
                                                        (
                                                                (a IN (getVertices G1))
                                                                /\
```

(lanjutan)

```
                                                        (b IN (getVertices G2))
                                                )
                                                ==>
                                                (
                                                        (Undirected_Adjacent a b
G1)
                                                        =
                                                        (Undirected_Adjacent (f
a) (f b) G2)
                                                )
                                        )
                        )
        `;



val _ = export_theory();

end;
```

# Lampiran C: Instalasi Sistem HOL dan Cara Menjalankan *Script*

Sistem HOL dapat dipasang pada komputer dengan sistem operasi Unix, Linux, Windows NT (misalnya Windows XP) atau sesudahnya, atau pun MacOS X. Sistem HOL dapat diunduh dari http://hol.sourceforge.net. Sistem HOL yang digunakan dalam penelitian ini dan yang akan dijelaskan pada lampiran ini adalah HOL Kananaskis-4. Sebelum melakukan instalasi sistem HOL, Moscow ML versi terbaru (minimal versi 2.01) harus sudah terpasang pada komputer. Moscow ML dapat diunduh dari http://www.dina.kvl.dk/~sestoft/mosml.html. Pada lampiran ini, hanya akan dijelaskan instalasi sistem HOL pada sistem operasi Windows. Untuk sistem operasi Unix dan Linux, instalasi sistem HOL dapat dilakukan dengan melihat [7].

Setelah pemasang (*installer*) sistem HOL sudah diunduh, jalankan program "kananaskis-4-install.exe". Pada jendela pertama yang muncul, tekan tombol "Next >". Pada jendela kedua, masukkan nama direktori di mana sistem HOL akan dipasang. Perlu diperhatikan bahwa nama direktori tersebut tidak boleh berisi karakter spasi putih (*white space*) berdasarkan eksperimen pada subbab 3.1. Setelah itu, tekan tombol "Next >". Pada jendela ketiga, masukkan nama *folder* di mana *shortcut* sistem HOL akan diletakkan di menu Start. Setelah itu, tekan tombol "Next >". Pada jendela keempat, jika sudah yakin, tekan tombol "Install". Setelah itu, sistem HOL akan terpasang pada komputer.

Untuk menjalankan sistem HOL, kunjungi direktori "bin" yang terdapat di dalam direktori di mana sistem HOL terpasang, lalu jalankan "hol.bat". Setelah itu, sistem HOL akan membuka dan memuat pustaka-pustaka standar ke dalam sistemnya. Kode sumber dari pustaka-pustaka sistem HOL terdapat pada direktori "src" yang terdapat di dalam direktori di mana sistem HOL terpasang. Untuk menggunakan kode sumber yang terdapat pada Lampiran B, ketik perintah "`use "graphScript.sml"`" (dengan asumsi nama berkas kode sumber tersebut adalah "graphScript.sml").