BAB 3 EKSPERIMEN

Pada bagian eksperimen dijelaskan detail dari eksperimen-eksperimen yang dilakukan. Eksperimen yang dilakukan dapat dibagi menjadi enam bagian, yaitu eksperimen setelah HOL terpasang sampai mengakibatkan sistem HOL harus dihapus dan dipasang kembali; eksperimen formalisasi teori faktor persekutuan terbesar (*greatest common divisor*, atau sering dikenal dengan nama *gcd*); eksperimen membuat struktur-struktur data yang dibutuhkan teori *graph* dan formalisasi sebuah definisi; eksperimen melakukan pembuktian mekanis menggunakan taktik; eksperimen formalisasi definisi-definisi dari teori *graph*; dan eksperimen formalisasi sebuah teorema dari teori *graph*. Pembagian eksperimen-eksperimen tersebut didasarkan pada rentang waktu pengerjaan dan jeda waktu antar eksperimen. Sistem operasi Microsoft Windows XP SP 2 digunakan untuk setiap eksperimen.

3.1. Eksperimen Menggunakan Pustaka HOL Kananaskis-4

Setelah HOL Kananaskis-4 berhasil dipasang di komputer, dilakukan percobaan sederhana yang diambil dari pustaka HOL. Dengan kata lain, sebagian perintah yang terdapat di pustaka HOL dicoba. Jika perintah yang dicoba adalah perintah yang berasal dari pustaka HOL, seharusnya perintah tersebut tidak akan mengalami kesalahan. Akan tetapi, setiap perintah tersebut dijalankan secara interaktif, HOL selalu menampilkan pesan kesalahan yang diikuti dengan nama lengkap direktori di mana HOL berada. Nama lengkap direktori tersebut tidak sepenuhnya benar terutama pada bagian nama yang berisi karakter spasi putih (*white space*). Pada waktu itu, sistem HOL terpasang di dalam direktori dengan nama yang berisi karakter spasi putih. Akhirnya, sistem HOL dihapus dan dipasang kembali di dalam direktori dengan nama yang tidak berisi karakter spasi putih. Sejak itu, perintah-perintah yang dijalankan tidak pernah mengalami kesalahan yang diikuti nama direktori lagi.

3.2. Eksperimen Formalisasi Teori GCD

Pada eksperimen kedua, dilakukan eksperimen mengenai formalisasi teori *gcd*. Teori tersebut sebenarnya sudah ada, sudah diformalisasikan, dan sudah dimasukkan ke dalam pustaka HOL. Akan tetapi, pendekatan definisi *gcd* yang berbeda – dibandingkan dengan definisi *gcd* yang ada di dalam pustaka – diambil. Jika yang terdapat di pustaka menggunakan pengurangan, maka pada eksperimen ini menggunakan sisa hasil bagi (*modulus*) berdasarkan cara Euclid. Dengan demikian, struktur teori *gcd* hasil formalisasi eksperimen ini mempunyai kemiripan dengan yang ada di pustaka. Akibat dari pendekatan definisi yang berbeda adalah teorema yang ada harus dibuktikan dengan cara yang berbeda, walau isi teoremanya mengatakan hal yang sama. Pada akhirnya, eksperimen kecil ini tidak berhasil dan dihentikan karena keterbatasan waktu dan bahan bacaan yang tepat mengenai teori sisa hasil bagi belum dapat ditemukan. Akan tetapi, bentuk formalisasi teori telah berhasil dipelajari melalui eksperimen ini.

3.3. Eksperimen Struktur *Graph* dan Sebuah Definisi dari Teori *Graph*

Pada eksperimen ketiga, dilakukan eksperimen yang menghasilkan bentuk umum hasil formalisasi teori *graph* dan struktur-struktur data yang dibutuhkan teori *graph*. Bentuk umum tersebut adalah sebagai berikut.

```
structure graphTheory =
struct

app load ["pred_setTheory", "stringTheory", "listTheory"];

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

val _ = new_theory "graph";

.
.
.

val _ = export_theory();
end;
```

Perintah `load` di atas hanya digunakan pada mode interaktif HOL dan dapat dibuat menjadi komentar setelah dibuat menjadi *script*. Pustaka-pustaka yang dibuka adalah pustaka standar dan pustaka lainnya yang dibutuhkan dalam formalisasi teori *graph*. Struktur-struktur data, definisi-definisi, dan teorema-teorema diletakkan pada bagian yang ditandai titik tiga secara vertikal.

Terdapat tiga buah struktur data yang dibutuhkan teori *graph*, yaitu verteks (*vertex*), sisi (*edge*), dan *graph*. Mula-mula verteks ingin dibuat dengan tipe data yang umum (*generic*), tetapi ternyata belum berhasil ditemukan caranya sehingga verteks menggunakan tipe data `string`. Sisi dibuat dengan tipe data pasangan yang berisi pasangan verteks dan bobot sisi yang merupakan bilangan bulat menggunakan tipe data `num`. Sisi dibedakan menjadi dua jenis, yaitu berarah dan tidak berarah. Untuk sisi-sisi yang tidak berbobot pada *graph*, semua bobot sisi dapat diisi dengan 1. Struktur data ketiga yaitu *graph* dibuat dengan tipe data pasangan yang berisi himpunan verteks dan daftar sisi. Struktur data *graph* ini sedikit berbeda dengan yang terdapat pada [8]. Struktur ini menggunakan daftar sisi, sedangkan yang terdapat pada [8] menggunakan himpunan sisi. Modifikasi dilakukan karena mengingat adanya tipe *graph* seperti *multigraph* di mana terdapat kemungkinan sisi-sisinya tidak unik dan jika dipaksakan sisi harus unik (karena menggunakan himpunan), maka harus terdapat informasi tambahan dalam struktur data sisi (misal nomor sisi) yang bisa menambah kerumitan formalisasi teori *graph*. Berdasarkan pemikiran tersebut, daftar sisi lebih dipilih dibandingkan dengan himpunan sisi. Ketiga struktur data tersebut dinyatakan dalam HOL sebagai berikut.

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

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

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

Selain struktur data, eksperimen ketiga juga memberikan dua buah fungsi sederhana dan hasil formalisasi dari sebuah definisi dari teori *graph*. Dua buah fungsi sederhana yang dimaksud adalah `getVertices` dan `getEdges`. Fungsi `getVertices` berguna untuk mendapatkan himpunan verteks dari sebuah *graph*. Sedangkan, fungsi `getEdges` berguna untuk mendapatkan daftar sisi dari sebuah *graph*. Dua buah fungsi tersebut dinyatakan dalam HOL sebagai berikut.

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

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

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.

Berdasarkan Definisi 2.1.1, terdapat beberapa hal yang perlu diperhatikan, yaitu himpunan verteks dari *graph* tidak boleh merupakan himpunan kosong, setiap verteks yang digunakan pada semua sisi harus terdapat di dalam himpunan verteks, setiap sisi yang terdapat di dalam daftar sisi dari *graph* tidak boleh memiliki pasangan verteks yang sama, dan untuk setiap sisi tidak boleh berisi pasangan verteks di mana pasangan itu berisi verteks yang sama. Hasil formalisasi Definisi 2.1.1 pada eksperimen ketiga dinyatakan dalam HOL sebagai berikut.

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

val UNDIRECTED_SIMPLE_GRAPH = Define
        `
                Undirected_Simple_Graph G =
                        ~(getVertices G = {})
                        /\
                        (
                                !e1 e2.
                                        (MEM e1 (getEdges G))
```

```
                                        /\
                                        (MEM e2 (getEdges G))
                                        /\
                                        ~(e1 = e2)
                                        ==>
                                        (undirected_simple e1 e2)
                        )
        `;

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

3.4. Eksperimen Pembuktian Mekanis Menggunakan Taktik

Setelah eksperimen ketiga, eksperimen keempat merupakan kumpulan percobaan yang dilakukan mengenai pembuktian menggunakan Taktik. Eksperimen dimulai dengan mengambil contoh yang sangat sederhana, yaitu ingin dibuktikan "1 = 1". Untuk contoh ini, jelas sekali bahwa pernyataan tersebut pasti benar. Pernyataan tersebut (pernyataan yang ingin dibuktikan) dimasukkan secara interaktif ke HOL dengan perintah "`g `1 = 1`;`". Pernyataan tersebut sangat sederhana sehingga bisa langsung dibuktikan menggunakan Taktik `PROVE_TAC[]` dan perintah yang digunakan pada mode interaktif adalah "`e(PROVE_TAC[]);`". Dengan demikian, telah berhasil dibuktikan kebenaran pernyataan tersebut dalam sistem HOL.

Kemudian, contoh lainnya diambil, yaitu ingin dibuktikan "1 ≠ 2". Untuk contoh ini, jelas juga bahwa pernyataan tersebut pasti benar. Pernyataan tersebut dimasukkan secara interaktif ke HOL dengan perintah "`g `~(1 = 2)`;`". Walau pernyataan ini terlihat sederhana, Taktik `PROVE_TAC[]` gagal membuktikan kebenarannya. Lalu Taktik `RW_TAC[]` digunakan dengan aturan penyederhanaan yang dipilih adalah `arith_ss` (aturan penyederhanaan yang berhubungan dengan aritmatika), sehingga perintah yang digunakan adalah "`e(RW_TAC arith_ss[]);`". Dengan perintah tersebut, pernyataan "1 ≠ 2" telah berhasil dibuktikan.

Sedikit beralih ke *string* dengan mengambil contoh "`"1" = "1"`" sebagai pernyataan yang ingin dibuktikan. Menggunakan cara yang sama (menggunakan

Taktik `PROVE_TAC[]`) dengan cara membuktikan pernyataan "1 = 1", pernyataan "`"1" = "1"`" telah berhasil dibuktikan. Namun, ketika ingin dibuktikan pernyataan "`"1" ≠ "2"`", sampai saat ini belum berhasil ditemukan cara membuktikannya. Padahal, jika mengingat struktur verteks yang menggunakan tipe data *string*, kemungkinan besar ke depannya akan ada pengujian kesamaan atau pun pengujian ketidaksamaan *string*. Hal ini bisa menjadi masalah jika tidak diatasi. Akhirnya, diputuskan untuk mengganti tipe data yang digunakan verteks, dari *string* menjadi bilangan bulat sehingga struktur verteks yang baru dinyatakan dalam HOL sebagai berikut.

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

Contoh lainnya yang ingin dibuktikan yaitu pernyataan "1 ∈ {3, 2, 1}". Untuk bisa menggunakan definisi-definisi dan teorema himpunan, pustaka yang berkaitan dengan himpunan harus dibuka terlebih dahulu. Pustaka yang dimaksud adalah pustaka `pred_setTheory` yang sudah ada di dalam sistem HOL. Operasi anggota himpunan di dalam HOL dinyatakan dengan `IN`. Pada mulanya, digunakan perintah "`g `1 IN {3, 2, 1}`;`" untuk mendaftarkan pernyataan tersebut untuk dibuktikan. Akan tetapi, HOL menampilkan pesan kesalahan yang kurang dapat dipahami. Setelah mencoba cukup lama, akhirnya diketahui bahwa tanda pemisah himpunan di dalam HOL bukan tanda koma, melainkan tanda titik koma. Dengan demikian, perintahnya menjadi "`g `1 IN {3; 2; 1}`;`". Sekarang timbul masalah baru, bagaimana cara membuktikannya? Seperti biasa, awalnya Taktik `PROVE_TAC[]` dan `RW_TAC[]` digunakan, tetapi kedua Taktik ini tidak berhasil melakukan pembuktian. Lalu, kode sumber pustaka yang terkait dengan himpunan dibuka. Awalnya, definisi `IN_DEF` yang terdapat di pustaka digunakan karena terdapat `IN` di dalam pernyataan yang ingin dibuktikan, sehingga perintah yang digunakan adalah "`e(RW_TAC arith_ss[IN_DEF]);`". Usaha ini pun tidak berhasil dan setelah dicoba menggunakan definisi dan teorema lainnya pada awal kode sumber pustaka, pembuktian belum juga berhasil dilakukan.

Lalu, terpikir bahwa mungkin pernyataan itu terlalu sulit dibuktikan sehingga pernyataannya diubah menjadi "1 ∈ {1}". Dengan menggunakan Taktik `RW_TAC[]` dan definisi `IN_DEF`, serta definisi dan teorema awal, pembuktian tetap belum berhasil dilakukan. Terinspirasi dengan metode *brute force*, satu per satu definisi dan teorema yang terdapat di pustaka terkait dari awal dicoba. Akhirnya, ditemukan definisi `IN_INSERT` yang berhasil membuktikan pernyataan tersebut. Perintah yang digunakan adalah "`e(RW_TAC arith_ss[IN_INSERT]);`". Dengan Taktik dan definisi yang sama, berhasil juga dibuktikan pernyataan "1 ∈ {3, 2, 1}". Setelah melihat definisi `IN_INSERT`, yaitu

```
IN_INSERT (!x y s. x IN (y INSERT s) = {y | (x = y) \/ (x IN s)})
```

, definisi itu memang dapat digunakan. Dari pernyataan "1 ∈ {3, 2, 1}" dapat dijabarkan menurut definisi `IN_INSERT` menjadi

```
  1 IN (3 INSERT {2; 1})
= (1 = 3) \/ (1 IN (2 INSERT {1}))
= F       \/ ((1 = 2) \/ (1 IN (1 INSERT {})))
=             F       \/ ((1 = 1) \/ …)
=                         T       \/ …
= T
```

Pada kesempatan lain, ingin dibuktikan juga pernyataan "{1, 2, 3} = {1, 3, 2}". Awalnya, pembuktian dimulai dengan Taktik `PROVE_TAC[]` dan `REWRITE_TAC[]`, tetapi belum berhasil. Lalu, definisi `EXTENSION` yang terdapat di pustaka yang dapat mengektensi himpunan dicoba menggunakan Taktik `REWRITE_TAC[]`. Setelah itu, *subgoal* terlihat mempunyai struktur yang mirip dengan contoh pernyataan "1 ∈ {3, 2, 1}" sehingga definisi `IN_INSERT` dapat digunakan kembali. *Subgoal* yang dihasilkan menjadi panjang dan terlihat tidak cukup sederhana, tetapi dengan pengalaman sebelumnya, taktik `PROVE_TAC[]` dicoba terlebih dahulu dan akhirnya pembuktian berhasil dilakukan. Setelah pembuktian ulang dilakukan, didapatkan bentuk yang lebih singkat, yaitu menggunakan perintah "`e(REWRITE_TAC[EXTENSION, IN_INSERT] THEN PROVE_TAC[]);`". Berikut adalah detail tampilan pada jendela mode interaktif untuk pembuktian pernyataan "{1, 2, 3} = {1, 3, 2}".

```
- g `{1; 2; 3} = {1; 3; 2}`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         {1; 2; 3} = {1; 3; 2}
```

```
     : proofs
- e(REWRITE_TAC[EXTENSION]);
OK..
1 subgoal:
> val it =
    !x. x IN {1; 2; 3} = x IN {1; 3; 2}

     : goalstack
- e(REWRITE_TAC[IN_INSERT]);
OK..
1 subgoal:
> val it =
    !x.
      (x = 1) \/ (x = 2) \/ (x = 3) \/ x IN {} =
      (x = 1) \/ (x = 3) \/ (x = 2) \/ x IN {}

     : goalstack
- e(PROVE_TAC[]);
OK..
Meson search level: ...............................

Goal proved.
|- !x.
     (x = 1) \/ (x = 2) \/ (x = 3) \/ x IN {} =
     (x = 1) \/ (x = 3) \/ (x = 2) \/ x IN {}

Goal proved.
|- !x. x IN {1; 2; 3} = x IN {1; 3; 2}
> val it =
    Initial goal proved.
    |- {1; 2; 3} = {1; 3; 2} : goalstack
```

Lalu, juga ingin dibuktikan pernyataan "$\{1, 2\} \neq \{3, 1\}$". Berdasarkan pengalaman banyak definisi yang terdapat di pustaka himpunan yang dilihat, langsung diputuskan untuk menggunakan definisi `NOT_EQUAL_SETS` dengan Taktik-nya `RW_TAC[]` dan aturan penyederhanaannya `arith_ss`. Setelah didapatkan *subgoal*, terlihat bentuknya yang mirip dengan pernyataan "$1 \in \{3, 2, 1\}$" sehingga definisi yang digunakan adalah `IN_INSERT` dengan Taktik yang sama. Seperti biasa, Taktik `PROVE_TAC[]` dicoba, tetapi ternyata belum berhasil. Dengan melihat adanya `IN` di dalam subgoal, definisi `IN_DEF` digunakan dengan Taktik yang sama. Setelah didapatkan *subgoal* baru, himpunan kosong terlihat dan dengan menggunakan definisi EMPTY_DEF dan Taktik yang sama akhirnya pernyataan itu berhasil dibuktikan. Kemudian pembuktian dicoba sekali lagi dan didapatkan cara yang lebih singkat yaitu dengan perintah berikut.

```
e(RW_TAC arith_ss[NOT_EQUAL_SETS, IN_INSERT, IN_DEF, EMPTY_DEF]);
```

Detail pembuktiannya pada jendela mode interaktif dapat dilihat sebagai berikut.

```
- g `~({1; 2} = {3; 1})`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ~({1; 2} = {3; 1})


     : proofs
- e(RW_TAC arith_ss[NOT_EQUAL_SETS]);
OK..
1 subgoal:
> val it =
    ?x. x IN {3; 1} = ~(x IN {1; 2})

     : goalstack
- e(RW_TAC arith_ss[IN_INSERT]);
OK..
1 subgoal:
> val it =
    ?x. (x = 3) \/ (x = 1) \/ x IN {} = ~(x = 1) /\ ~(x = 2) /\ ~(x IN {})

     : goalstack
- e(RW_TAC arith_ss[IN_DEF]);
OK..
1 subgoal:
> val it =
    ?x. (x = 3) \/ (x = 1) \/ {} x = ~(x = 1) /\ ~(x = 2) /\ ~{} x

     : goalstack
- e(RW_TAC arith_ss[EMPTY_DEF]);
OK..

Goal proved.
|- ?x. (x = 3) \/ (x = 1) \/ {} x = ~(x = 1) /\ ~(x = 2) /\ ~{} x

Goal proved.
|- ?x.
     (x = 3) \/ (x = 1) \/ x IN {} = ~(x = 1) /\ ~(x = 2) /\ ~(x IN {})
```

```
Goal proved.
|- ?x. x IN {3; 1} = ~(x IN {1; 2})
> val it =
    Initial goal proved.
    |- ~({1; 2} = {3; 1}) : goalstack
```

Contoh yang ingin dibuktikan sekarang melibatkan Definisi 2.1.1 dalam bentuk formal pada eksperimen ketiga, yaitu apakah sebuah *graph* dengan himpunan verteks {`Vertex_ 1`} dan daftar sisi kosong merupakan sebuah *graph* sederhana. Pernyataan tersebut didaftarkan untuk dibuktikan dengan perintah "`g `Simple_Graph (Graph_ ({Vertex_ 1}, []))`;`". Berdasarkan pengalaman sebelumnya dan Definisi 3.1.1 yang formal, digunakan Taktik `REWRITE_TAC[]` dan definisi-definisi yang digunakan secara bersamaan adalah `SIMPLE_GRAPH`, `UNDIRECTED_SIMPLE_GRAPH`, dan `GET_VERTICES`. Setelah didapatkan *subgoal* baru, terdapat struktur yang mirip dengan pernyataan "{1, 2} ≠ {3, 1}" sehingga Taktik `RW_TAC[]` dapat digunakan dengan definisi-definisi `NOT_EQUAL_SETS`, `IN_INSERT`, `IN_DEF`, dan `EMPTY_DEF`. *Subgoal* baru yang didapat memiliki dua peubah yaitu `e1` dan `e2`. Dilihat dari strukturnya, taktik `Induct_on` dapat digunakan. Oleh karena itu, `Induct_on` digunakan pada peubah `e1`. *Subgoal* baru didapat dan di dalamnya terdapat `MEM` dan `getEdges` sehingga dapat dicoba menggunakan Taktik `REWRITE_TAC[]` dengan definisi `MEM` dan `GET_EDGES`. *Subgoal* baru yang didapat pun bentuknya mirip, maka dapat dilakukan hal yang sama. Pada langkah terakhir ini telah berhasil dilakukan pembuktian terhadap pernyataan awal. Berikut detail pembuktian pada jendela tampilan mode interaktif.

```
.
.
.
> val it =
    Initial goal:

    Simple_Graph (Graph_ ({Vertex_ 1},[]))

     : goalstack
- e(REWRITE_TAC[SIMPLE_GRAPH, UNDIRECTED_SIMPLE_GRAPH, GET_VERTICES]);
OK..
1 subgoal:
> val it =
```

```
    ~({Vertex_ 1} = {}) /\
    !e1 e2.
      MEM e1 (getEdges (Graph_ ({Vertex_ 1},[]))) /\
      MEM e2 (getEdges (Graph_ ({Vertex_ 1},[]))) /\ ~(e1 = e2) ==>
      undirected_simple e1 e2

     : goalstack
- e(RW_TAC arith_ss[NOT_EQUAL_SETS, IN_INSERT, IN_DEF, EMPTY_DEF]);
OK..
1 subgoal:
> val it =
    undirected_simple e1 e2
    ------------------------------------
      0.  MEM e1 (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[])))
      1.  MEM e2 (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[])))
      2.  ~(e1 = e2)
     : goalstack
- e(Induct_on `e1`);
OK..
2 subgoals:
> val it =
    !p.
      MEM (Directed p) (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[]))) ==>
      ~(Directed p = e2) ==>
      undirected_simple (Directed p) e2
    ------------------------------------
      MEM e2 (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[])))

    !p.
      MEM (Undirected p)
        (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[]))) ==>
      ~(Undirected p = e2) ==>
      undirected_simple (Undirected p) e2
    ------------------------------------
      MEM e2 (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[])))
     : goalstack
- e(REWRITE_TAC[MEM, GET_EDGES]);
OK..

Goal proved.
|- !p.
     MEM (Undirected p)
       (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[]))) ==>
     ~(Undirected p = e2) ==>
     undirected_simple (Undirected p) e2

Remaining subgoals:
> val it =
```

```
    !p.
      MEM (Directed p) (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[]))) ==>
      ~(Directed p = e2) ==>
      undirected_simple (Directed p) e2
    ------------------------------------
      MEM e2 (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[])))
     : goalstack
- e(REWRITE_TAC[MEM, GET_EDGES]);
OK..

Goal proved.
|- !p.
     MEM (Directed p)
       (getEdges (Graph_ (Vertex_ 1 INSERT (\x. F),[]))) ==>
     ~(Directed p = e2) ==>
     undirected_simple (Directed p) e2

Goal proved.
 [..] |- undirected_simple e1 e2

Goal proved.
|- ~({Vertex_ 1} = {}) /\
   !e1 e2.
     MEM e1 (getEdges (Graph_ ({Vertex_ 1},[]))) /\
     MEM e2 (getEdges (Graph_ ({Vertex_ 1},[]))) /\ ~(e1 = e2) ==>
     undirected_simple e1 e2
> val it =
    Initial goal proved.
    |- Simple_Graph (Graph_ ({Vertex_ 1},[])) : goalstack
```

Masih dengan melibatkan Definisi 2.1.1 yang formal pada eksperimen ketiga, ingin dibuktikan bahwa sebuah *graph* yang memiliki himpunan verteks {`Vertex_ 1`} dan daftar sisi [`Undirected ((Vertex_ 1, Vertex_ 1), 1)`] adalah bukan *graph* sederhana. Setelah lama mencoba, pembuktiannya tetap belum berhasil ditemukan. Definisi 2.1.1 yang formal pada eksperimen ketiga mulai dicurigai mungkin salah. Oleh karena itu, Definisi 2.1.1 yang informal tersebut perlu dilihat kembali dan ternyata memang definisi formalnya tersebut masih mempunyai banyak cacat. Dengan ditemukannya kekurangan dari definisi formal tersebut, diupayakan pendekatan lain yang dijabarkan pada eksperimen kelima (subbab 3.5).

3.5. Eksperimen Memformalisasikan Definisi-Definisi dari Teori *Graph*

Eksperimen kelima memberikan hasil formalisasi dari definisi-definisi yang terdapat di dalam teori *graph*. Hasil formalisasi definisi-definisi dapat dilihat pada Bab 4 dan kode sumber hasil formalisasi definisi-definisi teori *graph* dapat dilihat pada Lampiran B. Kembali dengan Definisi 2.1.1 formal yang belum sempurna, definisi tersebut diperbaiki dengan memperhatikan keempat hal yang perlu diperhatikan dari definisi 2.1.1. Definisi `def211_1`, `def211_2`, `def211_3`, `def211_4`, dan `def211_5` ditambahkan untuk mendukung Definisi 2.1.1. Definisi `def211_1` menyatakan bahwa pasangan verteks yang terdapat di dalam sisi harus merupakan verteks-verteks yang terdapat di dalam himpunan verteks. Definisi `def211_2` menyatakan bahwa pasangan verteks pada sisi tidak boleh merupakan verteks yang sama (tidak boleh ada putaran). Definisi `def211_3`, `def211_4`, dan `def211_5` jika digabungkan menyatakan bahwa untuk sembarang dua sisi berbeda dari *graph* tidak akan memiliki pasangan verteks yang sama.

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 $f(e_1) = f(e_2)$.

Berdasarkan Definisi 2.1.2 tersebut terdapat beberapa hal yang perlu diperhatikan, yaitu pasangan verteks yang terdapat di dalam setiap sisi harus merupakan verteks-verteks yang terdapat di dalam himpunan verteks (hal ini didukung oleh definisi `def212_1`) dan pasangan verteks pada sisi tidak boleh merupakan verteks yang sama (hal ini didukung oleh definisi `def212_2`). Ada juga definisi `def212_3` yang menyatakan bahwa dua buah sisi dikatakan paralel jika dan hanya jika pasangan verteks pada kedua sisi menyatakan pasangan verteks yang sama.

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 $u \in V$.

Berdasarkan Definisi 2.1.3 tersebut terdapat hal yang perlu diperhatikan, yaitu pasangan verteks yang terdapat di dalam setiap sisi harus merupakan verteks-verteks yang terdapat di dalam himpunan verteks (hal ini didukung oleh definisi `def213_1`). Ada juga definisi `def213_2` yang menyatakan bahwa sebuah sisi dikatakan *pseudo* jika dan hanya jika pasangan verteksnya merupakan verteks-verteks yang sama.

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.

Berdasarkan Definisi 2.1.4 tersebut terdapat beberapa hal yang perlu diperhatikan, yaitu pasangan verteks yang terdapat di dalam setiap sisi harus merupakan verteks-verteks yang terdapat di dalam himpunan verteks (hal ini didukung oleh definisi `def214_1`) dan untuk sembarang dua sisi berbeda dari *graph* tidak akan memiliki pasangan verteks yang sama (hal ini didukung oleh definisi `def214_2`, `def214_3`, dan `def214_4`).

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 $f(e_1) = f(e_2)$.

Berdasarkan Definisi 2.1.5 tersebut terdapat hal yang perlu diperhatikan, yaitu pasangan verteks yang terdapat di dalam setiap sisi harus merupakan verteks-verteks yang terdapat di dalam himpunan verteks (hal ini didukung oleh definisi `def215_1`). Ada juga definisi `def215_2` yang menyatakan bahwa dua buah sisi berarah dikatakan paralel jika dan hanya jika pasangan verteks pada kedua sisi menyatakan pasangan verteks yang sama.

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 $\{u, v\}$.

Berdasarkan Definisi 2.2.1 tersebut dapat dibuat tiga buah definisi yang lebih kecil, yaitu definisi `UNDIRECTED_ADJACENT`, `END_POINT`, dan `END_LOOP`. Definisi `UNDIRECTED_ADJACENT` menyatakan bahwa dua buah verteks berdekatan di dalam sebuah *graph* tidak berarah jika dan hanya jika *graph*-nya tidak berarah, kedua verteks berada di dalam *graph*, dan ada sisi yang memiliki pasangan verteks dari kedua verteks tersebut (hal ini didukung oleh definisi `def221_1`). Definisi `END_POINT` menyatakan bahwa sebuah verteks merupakan titik akhir dari sebuah sisi jika dan hanya jika verteks itu terdapat di pasangan verteks dari sisi tersebut. Definisi `END_POINT_LOOP` menyatakan bahwa sebuah verteks merupakan titik akhir sisi putaran jika dan hanya jika pasangan verteks dari sisi itu memiliki verteks-verteks yang sama dan verteks yang ingin diketahui sama dengan verteks dari pasangan verteks tersebut.

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).

Berdasarkan Definisi 2.2.2 tersebut terdapat beberapa hal yang perlu diperhatikan, yaitu *graph*-nya harus tidak berarah dan verteksnya harus berada di dalam himpunan verteks dari *graph*. Definisi `def222_1` dibuat untuk mendukung Definisi 2.2.2. Definisi `def222_1` dinyatakan secara rekursif terhadap daftar sisi.

Jika daftar sisi saat ini kosong, tentu saja derajatnya adalah 0. Jika daftar sisi tidak kosong, maka derajat dihitung berdasarkan kondisi elemen daftar sisi terdepan dengan verteks yang ingin dihitung derajatnya (jika verteks adalah titik akhir putaran dari sisi, maka derajatnya 2; selain itu jika verteks adalah titik akhir dari sisi, maka derajatnya 1; selain itu derajatnya 0), dijumlahkan dengan hasil penghitungan dari derajat verteks pada sisa daftar sisi yang dapat diketahui dengan pemanggilan secara rekursif.

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.

Berdasarkan Definisi 2.2.3 tersebut dapat dibuat enam buah definisi yang lebih kecil, yaitu definisi `DIRECTED_ADJACENT`, `ADJACENT_TO`, `ADJACENT_FROM`, `INITIAL_VERTEX`, `TERMINAL_VERTEX`, dan `SAME_VERTICES_IN_LOOP`. Definisi `DIRECTED_ADJACENT` menyatakan bahwa dua buah verteks berdekatan di dalam sebuah *graph* berarah jika dan hanya jika *graph*-nya berarah, kedua verteks berada di dalam *graph*, dan ada sisi yang memiliki pasangan verteks di mana sesama verteks awal dan sesama verteks akhir sama (hal ini didukung oleh definisi `def223_1`). Definisi `ADJACENT_TO` menyatakan bahwa sebuah verteks dikatakan "berdekatan ke" dari sebuah sisi dari *graph* berarah jika dan hanya jika *graph*-nya berarah, verteksnya terdapat di dalam himpunan verteks dari *graph*, sisinya terdapat di dalam daftar sisi dari *graph*, dan verteks yang ingin diketahui berada pada bagian kiri dari pasangan verteks sisi tersebut (hal ini didukung oleh definisi `def223_2`). Definisi `ADJACENT_FROM` menyatakan bahwa sebuah verteks dikatakan "berdekatan dari" dari sebuah sisi dari *graph* berarah jika dan hanya jika *graph*-nya berarah, verteksnya terdapat di dalam himpunan verteks dari *graph*, sisinya terdapat di dalam daftar sisi dari *graph*, dan verteks yang ingin diketahui berada pada bagian kanan dari pasangan verteks sisi tersebut (hal ini didukung oleh definisi `def223_3`). Definisi `INITIAL_VERTEX` menyatakan bahwa sebuah verteks dikatakan verteks awal dari sebuah sisi berarah jika dan hanya jika verteks itu berada pada bagian kiri dari pasangan verteks sisi tersebut. Definisi `TERMINAL_VERTEX` menyatakan bahwa sebuah verteks verteks dikatakan verteks akhir dari sebuah sisi berarah jika dan hanya jika verteks itu berada pada bagian kanan dari pasangan verteks sisi tersebut. Definisi `SAME_VERTICES_IN_LOOP` menyatakan bahwa sebuah sisi dikatakan sisi putaran jika dan hanya jika verteks-verteksnya 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.

Berdasarkan Definisi 2.2.4 tersebut dapat dibuat dua buah definisi yang lebih kecil, yaitu definisi `IN_DEGREE` dan `OUT_DEGREE`. Beberapa hal yang perlu diperhatikan dari kedua definisi tersebut adalah *graph*-nya harus berarah dan verteks yang ingin diketahui derajatnya harus terdapat di dalam himpunan verteks dari *graph*. Definisi `def224_1` dibuat untuk mendukung definisi `IN_DEGREE`. Definisi `def224_1` dinyatakan secara rekursif terhadap daftar sisi. Jika daftar sisi saat ini kosong, tentu saja derajatnya adalah 0. Jika daftar sisi tidak kosong, maka derajat dihitung berdasarkan kondisi elemen daftar sisi terdepan dengan verteks yang ingin dihitung derajatnya (jika verteks adalah titik awal dari sisi, maka derajatnya 1; selain itu derajatnya 0), dijumlahkan dengan hasil penghitungan dari derajat dalam verteks pada sisa daftar sisi yang dapat diketahui dengan pemanggilan secara rekursif. Definisi `def224_2` dibuat untuk mendukung definisi `OUT_DEGREE`. Definisi `def224_2` dinyatakan secara rekursif terhadap daftar sisi. Jika daftar sisi saat ini kosong, tentu saja derajatnya adalah 0. Jika daftar sisi tidak kosong, maka derajat dihitung berdasarkan kondisi elemen daftar sisi terdepan dengan verteks yang ingin dihitung derajatnya (jika verteks adalah titik akhir dari sisi, maka derajatnya 1; selain itu derajatnya 0), dijumlahkan dengan hasil penghitungan dari derajat dalam verteks pada sisa daftar sisi yang dapat diketahui dengan pemanggilan secara rekursif.

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).

Berdasarkan Definisi 2.2.5 tersebut terdapat beberapa hal yang perlu diperhatikan, yaitu *graph*-nya harus sederhana, himpunan verteks harus dipartisi menjadi dua himpunan yang saling lepas, dan verteks yang satu dari setiap sisi harus berada di himpunan yang satu dan verteks yang lain dari sisi yang sama harus berada di himpunan yang lain (hal ini didukung oleh definisi `def225_1`).

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.

Berdasarkan Definisi 2.2.6 tersebut terdapat beberapa hal yang perlu diperhatikan, yaitu himpunan verteks dari *subgraph* harus merupakan subhimpunan dari himpunan verteks dari *graph* semula dan daftar sisi dari *subgraph* harus merupakan subdaftar dari daftar sisi dari *graph* semula (hal ini didukung oleh definisi `SUB_LIST`).

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 $G_1 \cup G_2$.

Berdasarkan Definisi 2.2.7 tersebut terdapat beberapa hal yang perlu diperhatikan, yaitu gabungan dari himpunan verteks *subgraph* yang satu dengan *subgraph* yang lain harus sama dengan himpunan verteks dari *graph* gabungan dan penambahan dari daftar sisi *subgraph* yang satu dengan *subgraph* yang lain harus sama dengan daftar sisi dari *graph* gabungan.

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.

Berdasarkan Definisi 2.3.1 tersebut terdapat beberapa hal yang perlu diperhatikan, yaitu kedua *graph* harus merupakan *graph* sederhana, ada sebuah fungsi bijektif dengan daerah asal merupakan himpunan verteks dari *graph* yang satu dan daerah tujuan merupakan himpunan verteks dari *graph* yang lain, dan untuk setiap anggota dari kedua himpunan verteks memenuhi syarat berdekatan di dalam *graph* yang satu jika dan hanya jika hasil fungsi dari anggota himpunan verteks yang satu dengan hasil fungsi dari anggota himpunan verteks yang lain memenuhi syarat berdekatan di dalam *graph* yang lain.

3.6. Eksperimen Memformalisasikan Teorema dari Teori *Graph*

Eksperimen keenam memberikan hasil formalisasi dari teorema yang terdapat di dalam teori *graph*. Hasil formalisasi teorema dapat dilihat pada Bab 4 dan kode sumber hasil formalisasi teorema dari teori *graph* dapat dilihat pada Lampiran B.

Teorema 2.2.1

Misalkan $G = (V, E)$ merupakan sebuah *graph* tidak berarah dengan e sisi, maka $2e = \sum_{v \in V} \deg(v)$.

Pertama, perlu didefinisikan terlebih dahulu dua buah definisi baru yang akan membantu teorema ini, yaitu definisi `number_of_edges` dan `sum_of_degree`. Definisi `number_of_edges` menyatakan banyak sisi dari sebuah *graph* yang menjadi parameternya. Definisi ini dapat dibuat dengan menghitung banyak sisi dari sebuah daftar sisi pada *graph* menggunakan definisi `LENGTH` dari pustaka mengenai daftar (*list*). Hal tersebut dapat dilakukan karena banyak sisi dari sebuah daftar sisi sama dengan panjang dari daftar sisi itu sendiri.

Dengan demikian, definisi `number_of_edges` dinyatakan dalam HOL sebagai berikut.

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

Sementara, definisi `sum_of_degree` menyatakan jumlah derajat dari sebuah daftar sisi yang menjadi parameternya. Definisi ini dapat dibuat secara rekursif terhadap daftar sisi dengan kasus dasarnya adalah daftar sisi yang kosong. Jika daftar sisi kosong, maka tentu saja jumlah derajatnya 0. Jika tidak, maka jumlah derajatnya adalah dua lebih banyak dari jumlah derajat daftar sisi yang telah dibuang elemen terdepan dari daftar sisi semula. Pernyataan "dua lebih banyak" mempunyai arti bahwa sebuah sisi menyumbangkan dua buah derajat kepada jumlah derajat. Dengan demikian, definisi `sum_of_degree` dinyatakan dalam HOL sebagai berikut.

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

Dengan menggunakan dua buah definisi bantuan tersebut, Teorema 2.2.1 dinyatakan dalam HOL sebagai berikut.

```
`!G. Undirected_Graph G ==> ((2 * (number_of_edges G)) = (sum_of_degree (getEdges G)))`
```

Di dalam membuktikan kebenaran teorema di atas, akan digunakan beberapa taktik, definisi, dan teorema yang sesuai. Dilihat dari bentuk terluarnya, ∀-kuantifikasi, dapat diubah dengan aturan instansiasi universal (menghilangkan ∀-kuantifikasi) menggunakan Taktik `STRIP_TAC`. Dengan demikian, bentuk terluarnya sekarang adalah implikasi. Pada bentuk implikasi tersebut terdapat dua bagian, yaitu hipotesis dan konsekuensi. Dalam hal ini, berdasarkan tabel kebenaran implikasi, jika berhasil dibuktikan bahwa konsekuensi benar, maka tidak perlu dibuktikan lagi hipotesisnya. Dengan menggunakan Taktik `STRIP_TAC THEN WEAKEN_TAC funcTrue`, hipotesis berhasil dibuang dan yang tersisa untuk dibuktikan adalah konsekuensinya. Fungsi `funcTrue` adalah sebuah fungsi yang menerima sebuah parameter apa saja dan selalu mengembalikan nilai `true`. Fungsi ini digunakan sebagai parameter taktik

WEAKEN_TAC. Pada saat ini, *goal* berbentuk "2 * number_of_edges G = sum_of_degree (getEdges G)" (tanda kutip hanya untuk kejelasan). Untuk mendapatkan hasil yang lebih sederhana (setidaknya dalam hal definisi number_of_edges dan getEdges dapat digunakan), peubah G perlu dibuka menggunakan Taktik Cases_on `G` sehingga sekarang berbentuk

```
2 * number_of_edges (Graph_ p) = sum_of_degree (getEdges (Graph_ p))
```
.

Selain itu, peubah p juga perlu dibuka menggunakan Taktik Cases_on `p` sehingga sekarang berbentuk

```
2 * number_of_edges (Graph_ (q,r)) = sum_of_degree (getEdges (Graph_ (q,r)))
```
.

Sekarang, definisi number_of_edges dan getEdges dapat digunakan bersama dengan Taktik REWRITE_TAC[] (penggunaannya REWRITE_TAC[NUMBER_OF_EDGES, GET_EDGES]) menghasilkan "2 * LENGTH r = sum_of_degree r". Untuk menyelesaikannya, dapat digunakan induksi matematika dengan Taktik Induct_on `r` yang menghasilkan dua buah *subgoal* berikut.

```
2 * LENGTH [] = sum_of_degree []
2 * LENGTH (h::t) = sum_of_degree (h::t)
```

Ternyata dengan menggunakan dua kali Taktik RW_TAC arith_ss [LENGTH, SUM_OF_DEGREE], kasus dasar dan kasus induksi (dua buah *subgoal* tersebut) sudah berhasil dibuktikan. Dengan demikian, *goal* awal juga telah berhasil dibuktikan. Teorema 2.2.1 dinyatakan dalam HOL sebagai berikut.

```
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]));
```

BAB 4 HASIL FORMALISASI TEORI GRAPH

Dengan mengacu pada semua hal yang perlu diperhatikan (atau pun pembuatan definisi-definisi yang lebih kecil) sesuai dengan isi subbab 3.5 dan 3.6, definisi-definisi dan teorema menjadi formal dan kode sumbernya dapat dilihat pada Lampiran B.

4.1. Rangkuman Hasil

Secara umum, terdapat 13 definisi dan 1 teorema yang setelah dilakukan formalisasi dapat dirangkum sebagai berikut.

- Terdapat tiga struktur data, yaitu verteks (bertipe bilangan bulat), sisi (bertipe pasangan dari pasangan verteks dan bilangan bulat (bobot sisi); dapat berupa sisi berarah atau tidak berarah), dan *graph* (bertipe pasangan dari himpunan verteks dan daftar sisi).
- Terdapat 21 definisi penting yang disesuaikan (tetap maupun dipecah menjadi beberapa definisi kecil) dari 13 definisi awal, yaitu:
    - Definisi 2.1.1: definisi `Simple_Graph` menyatakan definisi untuk *graph* sederhana.
    - Definisi 2.1.2: definisi `Multi_Graph` menyatakan definisi untuk *multigraph*.
    - Definisi 2.1.3: definisi `Pseudo_Graph` menyatakan definisi untuk *pseudograph*.
    - Definisi 2.1.4: definisi `Directed_Simple_Graph` menyatakan definisi untuk *graph* sederhana berarah.
    - Definisi 2.1.5: definisi `Directed_Multi_Graph` menyatakan definisi untuk *multigraph* berarah.
    - Definisi 2.2.1:
        - definisi `Undirected_Adjacent` menyatakan definisi untuk dua buah verteks bertetangga pada *graph* tidak berarah.

  - definisi `end_point` menyatakan definisi untuk sebuah verteks yang merupakan titik akhir dari sisi tidak berarah.
  - definisi `end_point_loop` menyatakan definisi untuk sebuah verteks yang berada pada sisi putaran tidak berarah.
- Definisi 2.2.2: definisi `Degree` menyatakan definisi untuk derajat sebuah verteks pada *graph* tidak berarah.
- Definisi 2.2.3:
  - definisi `Directed_Adjacent` menyatakan definisi untuk dua buah verteks bertetangga pada *graph* berarah.
  - definisi `adjacent_to` menyatakan definisi untuk sebuah verteks merupakan "berdekatan ke" dari sebuah sisi berarah pada *graph* berarah.
  - definisi `adjacent_from` menyatakan definisi untuk sebuah verteks merupakan "berdekatan dari" dari sebuah sisi berarah pada *graph* berarah.
  - definisi `initial_vertex` menyatakan definisi untuk sebuah verteks merupakan verteks awal dari sebuah sisi berarah.
  - definisi `terminal_vertex` menyatakan definisi untuk sebuah verteks merupakan verteks akhir dari sebuah sisi berarah.
  - definisi `same_vertices_in_loop` menyatakan definisi untuk sebuah sisi yang merupakan sisi putaran berarah.
- Definisi 2.2.4:
  - definisi `In_Degree` menyatakan definisi untuk derajat-dalam sebuah verteks pada *graph* berarah.
  - definisi `Out_Degree` menyatakan definisi untuk derajat-luar sebuah verteks pada *graph* berarah.
- Definisi 2.2.5: definisi `Bipartite` menyatakan definisi untuk sebuah *graph* dua-pihak.
- Definisi 2.2.6: definisi `Subset_Graph` menyatakan definisi untuk sebuah *graph* yang merupakan subhimpunan dari sebuah *graph* lainnya.
- Definisi 2.2.7: definisi `Union_Graph` menyatakan definisi untuk sebuah *graph* yang merupakan gabungan dari dua buah *graph*.

  - Definisi 2.3.1: definisi `Isomorphic` menyatakan definisi untuk sebuah *graph* yang merupakan isomorfik dari sebuah *graph* lainnya.
- Terdapat 6 definisi umum (definisi yang dibuat karena sering digunakan oleh definisi lain; definisi ini mempunyai arti jika berdiri sendiri), yaitu:
  - definisi `getVertices` menyatakan definisi untuk himpunan verteks dari sebuah *graph*.
  - definisi `getEdges` menyatakan definisi untuk daftar sisi dari sebuah *graph*.
  - definisi `sublist` menyatakan definisi untuk sebuah daftar yang merupakan subdaftar dari sebuah daftar lainnya.
  - definisi `Undirected_Graph` menyatakan definisi untuk sebuah *graph* tidak berarah.
  - definisi `Directed_Graph` menyatakan definisi untuk sebuah *graph* berarah.
  - definisi `Common_Graph` menyatakan definisi untuk sebuah *graph*.
- Terdapat 28 definisi bantuan (definisi yang mendukung definisi penting dan definisi umum; definisi ini kurang berarti jika berdiri sendiri).
- Terdapat 3 definisi tanggung (definisi yang dibuat untuk mendukung definisi penting dan jika berdiri sendiri masih mempunyai arti), yaitu:
  - Definisi 2.1.1: definisi `Undirected_Simple_Graph` menyatakan definisi untuk sebuah graph sederhana tidak berarah.
  - Definisi 2.1.2: definisi `Undirected_Multi_Graph` menyatakan definisi untuk sebuah *multigraph* tidak berarah.
  - Definisi 2.1.3: definisi `Undirected_Pseudo_Graph` menyatakan definisi untuk sebuah *pseudograph* tidak berarah.
- Terdapat 1 teorema, yaitu:
  - Teorema 2.2.1: teorema `HANDSHAKING_THM` menyatakan bahwa untuk setiap *graph* tidak berarah berlaku dua kali banyak sisi pada *graph* sama dengan total jumlah derajat verteks dari semua verteks pada *graph*.
- Terdapat total 913 baris (termasuk 723 baris bukan merupakan baris kosong) pada kode sumber.
- Kode sumber berukuran 21248 *Byte*.

4.2. Pemetaan Definisi

Pemetaan definisi merupakan pemetaan yang dilakukan dari definisi informal menjadi definisi formal yang diikuti dengan penjelasannya. Pada subbab ini, hanya akan diberikan pemetaan definisi dari tiga buah definisi tanggung, yaitu definisi `Undirected_Simple_Graph`, definisi `Undirected_Multi_Graph`, dan definisi `Undirected_Pseudo_Graph`. Pemetaan definisi lainnya dapat dilihat dengan cara membaca definisi informal, lalu hal-hal yang perlu diperhatikan seperti yang terdapat pada subbab 3.5 dapat digunakan untuk membantu memetakannya menjadi definisi formal.

Definisi `Undirected_Simple_Graph` memiliki definisi informal berdasarkan Definisi 2.1.1 dan Tabel 2.2, yaitu 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. *Graph* sederhana merupakan *graph* yang tidak berarah. Berikut adalah hasil formalisasi definisi tersebut.

```
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))`;
```

Berdasarkan hal-hal yang perlu diperhatikan pada subbab 3.5, definisi informal, dan definisi formal `Undirected_Simple_Graph`, pemetaan definisi informal menjadi definisi formal adalah sebagai berikut.

Definisi Informal	Definisi Formal
Himpunan verteks dari *graph* tidak boleh	`~(getVertices G = {})`

kosong. `G` menyatakan *graph*, notasi `{}` menyatakan himpunan kosong, dan simbol ~ menyatakan negasi.	
Setiap verteks di dalam pasangan verteks dari sisi harus terdapat di dalam himpunan verteks (*E* dari *V*).	`!e. (MEM e (getEdges G)) ==> (undirected_simple_e_in_setV e (getVertices G))`
Setiap sisi terdiri dari elemen-elemen verteks yang berbeda (setiap *E* memiliki elemen *V* berbeda), artinya tidak boleh ada sisi putaran.	`!e. (MEM e (getEdges G)) ==> (undirected_simple_diff_v_in_e e)`
Setiap sisi merupakan pasangan tidak berurut dari verteks dan sisi-sisi tersebut berbeda, artinya tidak ada dua atau lebih sisi tidak berarah yang sama.	`undirected_simple_diff_allE (getEdges G)`

Definisi `Undirected_Multi_Graph` memiliki definisi informal berdasarkan Definisi 2.1.2 dan Tabel 2.2, yaitu 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 $\{\{u, v\} \mid u, v \in V, u \neq v\}$. *Multigraph* merupakan graph yang tidak berarah. Berikut adalah hasil formalisasi definisi tersebut.

```
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)
                                )
                )
        `;
```

Berdasarkan hal-hal yang perlu diperhatikan pada subbab 3.5, definisi informal, dan definisi formal `Undirected_Multi_Graph`, pemetaan definisi informal menjadi definisi formal adalah sebagai berikut.

Definisi Informal	Definisi Formal
Setiap verteks di dalam pasangan verteks dari sisi harus terdapat di dalam himpunan verteks (*E* dari *V*).	`!e. (MEM e (getEdges G)) ==> (undirected_multi_e_in_setV e (getVertices G))`
Setiap sisi terdiri dari elemen-elemen verteks yang berbeda ($u \neq v$), artinya tidak boleh ada sisi putaran.	`!e. (MEM e (getEdges G)) ==> (undirected_multi_diff_v_in_e e)`

Definisi `Undirected_Pseudo_Graph` memiliki definisi informal berdasarkan Definisi 2.1.3 dan Tabel 2.2, yaitu 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 $\{\{u, v\} \mid u, v \in V\}$. *Multigraph* merupakan graph yang tidak berarah. Berikut adalah hasil formalisasi definisi tersebut.

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

Berdasarkan hal-hal yang perlu diperhatikan pada subbab 3.5, definisi informal, dan definisi formal `Undirected_Pseudo_Graph`, pemetaan definisi informal menjadi definisi formal adalah sebagai berikut.

Definisi Informal	Definisi Formal
Setiap verteks di dalam pasangan verteks dari sisi harus terdapat di dalam himpunan verteks (*E* dari *V*).	`!e. (MEM e (getEdges G)) ==> (undirected_pseudo_e_in_setV e (getVertices G))`

4.3. Uraian Formalisasi Definisi Penting dan Umum

Definisi `Simple_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *graph* sederhana. Definisi `Undirected_Simple_Graph` yang mendukung definisi `Simple_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *graph* sederhana yang tidak berarah.

Definisi `Multi_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *multigraph*. Definisi `Undirected_Multi_Graph` yang mendukung definisi `Multi_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *multigraph* yang tidak berarah.

Definisi `Pseudo_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *pseudograph*. Definisi `Undirected_Pseudo_Graph` yang mendukung definisi `Pseudo_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *pseudograph* yang tidak berarah.

Definisi `Directed_Simple_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *graph* sederhana yang berarah. Definisi `Directed_Multi_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya dapat ditentukan apakah merupakan sebuah *multigraph* yang berarah.

Definisi `Undirected_Adjacent` dapat digunakan dengan memberikan tiga buah parameter, yaitu dua buah verteks dan sebuah *graph*. Dalam penggunaannya, kedua buah verteks yang menjadi parameternya dapat ditentukan apakah merupakan tetangga pada *graph* yang menjadi parameternya. Definisi `end_point` dapat digunakan dengan memberikan dua buah parameter, yaitu sebuah verteks dan sebuah sisi tidak berarah. Dalam penggunaannya, verteks yang menjadi parameternya dapat ditentukan apakah merupakan titik akhir dari sisi yang menjadi parameternya. Definisi `end_point_loop` dapat digunakan dengan memberikan dua buah parameter, yaitu sebuah verteks dan sebuah sisi tidak berarah. Dalam penggunaannya, verteks yang menjadi parameternya dapat ditentukan apakah merupakan titik akhir putaran dari sisi yang menjadi parameternya.

Definisi `Degree` dapat digunakan dengan memberikan dua buah parameter, yaitu sebuah verteks dan sebuah *graph*. Dalam penggunaannya, verteks yang menjadi parameternya di dalam *graph* yang menjadi parameternya akan dihitung derajat verteks tersebut.

Definisi `Directed_Adjacent` dapat digunakan dengan memberikan tiga buah parameter, yaitu dua buah verteks dan sebuah *graph*. Dalam penggunaannya, kedua buah verteks yang menjadi parameternya dapat ditentukan apakah merupakan tetangga berarah pada *graph* yang menjadi parameternya. Definisi `adjacent_to` dapat digunakan dengan memberikan tiga buah parameter, yaitu sebuah verteks, sebuah sisi berarah, dan sebuah *graph*. Dalam penggunaannya, verteks yang menjadi parameternya dapat ditentukan apakah merupakan verteks awal dari pasangan verteks di dalam sisi yang menjadi parameternya pada *graph* yang menjadi parameternya. Definisi `adjacent_from` dapat digunakan dengan memberikan tiga buah parameter, yaitu sebuah verteks, sebuah sisi berarah, dan sebuah *graph*. Dalam penggunaannya, verteks yang menjadi parameternya dapat ditentukan apakah merupakan verteks akhir dari pasangan verteks di dalam sisi yang menjadi parameternya pada *graph* yang menjadi parameternya.

Definisi `initial_vertex` dapat digunakan dengan memberikan dua buah parameter, yaitu sebuah verteks dan sebuah sisi berarah. Dalam penggunaannya, verteks yang menjadi parameternya dapat ditentukan apakah merupakan verteks awal dari pasangan verteks di dalam sisi yang menjadi parameternya. Definisi `terminal_vertex` dapat digunakan dengan memberikan dua buah parameter, yaitu sebuah verteks dan sebuah sisi berarah. Dalam penggunaannya, verteks yang menjadi parameternya dapat ditentukan apakah merupakan verteks akhir dari pasangan verteks di dalam sisi yang menjadi parameternya. Definisi `same_vertices_in_loop` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah sisi berarah. Dalam penggunaannya, sisi yang menjadi parameternya dapat ditentukan apakah merupakan sisi putaran.

Definisi `In_Degree` dapat digunakan dengan memberikan dua buah parameter, yaitu sebuah verteks dan sebuah *graph*. Dalam penggunaannya, verteks yang menjadi parameternya di dalam *graph* yang menjadi parameternya akan dihitung derajat-dalam verteks tersebut. Definisi `Out_Degree` dapat digunakan dengan memberikan dua buah parameter, yaitu sebuah verteks dan sebuah *graph*. Dalam penggunaannya, verteks yang menjadi parameternya di dalam *graph* yang menjadi parameternya akan dihitung derajat-luar verteks tersebut.

Definisi `Bipartite` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya akan ditentukan apakah merupakan *graph* dua-pihak.

Definisi `Subset_Graph` dapat digunakan dengan memberikan dua buah parameter, yaitu dua buah *graph*. Dalam penggunaannya, graph yang menjadi parameter pertama akan ditentukan apakah merupakan subhimpunan dari graph yang menjadi parameter kedua. Definisi `Union_Graph` dapat digunakan dengan memberikan tiga buah parameter, yaitu tiga buah *graph*. Dalam penggunaannya, graph yang menjadi parameter ketiga akan ditentukan apakah merupakan gabungan dari graph yang menjadi parameter pertama dan kedua.

Definisi `Isomorphic` dapat digunakan dengan memberikan dua buah parameter, yaitu dua buah *graph*. Dalam penggunaannya, graph yang menjadi parameter pertama akan ditentukan apakah isomorfik dengan graph yang menjadi parameter kedua.

Definisi `getVertices` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Definisi ini memberikan himpunan verteks dari graph yang menjadi parameternya. Definisi `getEdges` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Definisi ini memberikan daftar sisi dari graph yang menjadi parameternya. Definisi `sublist` dapat digunakan dengan memberikan dua buah parameter, yaitu dua buah daftar. Dalam penggunaannya, daftar yang menjadi parameter pertama akan ditentukan apakah merupakan subdaftar dari daftar yang menjadi parameter kedua.

Definisi `Undirected_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, graph yang menjadi parameternya akan ditentukan apakah merupakan *graph* tidak berarah. Definisi `Directed_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, graph yang menjadi parameternya akan ditentukan apakah merupakan *graph* berarah. Definisi `Common_Graph` dapat digunakan dengan memberikan sebuah parameter, yaitu sebuah *graph*. Dalam penggunaannya, *graph* yang menjadi parameternya akan ditentukan apakah merupakan *graph*.

4.4. Pemetaan Teorema

Pemetaan teorema merupakan pemetaan yang dilakukan dari teorema informal menjadi teorema formal, penjelasan pembuktian manual dan pembuktian mekanis, dan pemetaan antara pembuktian manual dengan pembuktian mekanis menggunakan Taktik dalam sistem HOL. Teorema 2.2.1 memiliki teorema informal, yaitu misalkan $G = (V, E)$ merupakan sebuah *graph* tidak berarah dengan e sisi, maka $2e = \sum_{v \in V} \deg(v)$. Teorema formalnya dalam HOL dinyatakan sebagai berikut.

```
`!G. Undirected_Graph G ==> ((2 * (number_of_edges G)) = (sum_of_degree (getEdges G)))`
```

Pembuktian manual yang informal untuk Teorema 2.2.1 adalah sebagai berikut. Jumlah derajat dari sebuah *graph* tidak berarah dapat dihitung dengan memperhatikan derajat-derajat yang diberikan oleh setiap sisi dari *graph* itu. Setiap sisi dari sebuah *graph* tidak berarah pasti berhubungan dengan dua buah verteks. Walaupun sisi itu merupakan sisi putaran, menurut definisinya, verteks yang menjadi ujung dari sisi semacam itu memberikan dua derajat ke jumlah derajat dari *graph*. Dengan demikian, setiap sisi dari sebuah *graph* tidak berarah pasti memberikan dua derajat ke jumlah derajat dari *graph*. Dari penjelasan-penjelasan tersebut di atas dapat dikatakan bahwa jumlah derajat dari sebuah *graph* tidak berarah itu sama dengan dua kali banyak sisi yang terdapat di dalam *graph*. Jadi, Teorema 2.2.1 telah terbukti.

Pembuktian manual yang formal untuk teorema yang sama adalah sebagai berikut.

1a	∀*G*. Undirected_Graph(*G*) → ((2 * number_of_edges(G)) = (sum_of_degree(getEdges(*G*)))) Undirected_Graph(*G*) → ((2 * number_of_edges(*G*)) = (sum_of_degree(getEdges(*G*))))	instansiasi universal
2a	Undirected_Graph(*G*) → ((2 * number_of_edges(*G*)) = (sum_of_degree(getEdges(*G*)))) ((2 * number_of_edges(*G*)) = (sum_of_degree(getEdges(*G*)))) { Undirected_Graph(*G*)}	asumsi
3a	((2 * number_of_edges(*G*)) = (sum_of_degree(getEdges(*G*)))) { Undirected_Graph(*G*)} (2 * number_of_edges(*G*)) = (sum_of_degree(getEdges(*G*)))	eliminasi asumsi
4a	(2 * number_of_edges(*G*)) = (sum_of_degree(getEdges(*G*))) (2 * number_of_edges(*Graph*_(*p*))) = (sum_of_degree(getEdges(*Graph*_(*p*))))	*G*=*Graph*_(*p*)
5a	(2 * number_of_edges(*Graph*_(*p*))) = (sum_of_degree(getEdges(*Graph*_(*p*)))) (2 * number_of_edges(*Graph*_(*q*,*r*))) = (sum_of_degree(getEdges(*Graph*_(*q*,*r*))))	*p*=(*q*,*r*)
6a	(2 * number_of_edges(*Graph*_(*q*,*r*))) = (sum_of_degree(getEdges(*Graph*_(*q*,*r*)))) (2 * LENGTH *r*) = (sum_of_degree *r*)	def. number_of_edges, getEdges
7a	(2 * LENGTH *r*) = (sum_of_degree *r*) (2*LENGTH [] = sum_of_degree []) ∧ ∀h. (2*LENGTH (*h*:*r*) = sum_of_degree (*h*:*r*)) {2 * LENGTH *r* = sum_of_degree *r*}	induksi matematika terhadap *r*
8a	(2*LENGTH [] = sum_of_degree []) ∧ ∀h. (2*LENGTH (*h*:*r*) = sum_of_degree (*h*:*r*)) {2 * LENGTH *r* = sum_of_degree *r*} T	def. LENGTH, sum_of_degree; dan aritmatika

Pembuktian mekanis untuk teorema yang sama adalah sebagai berikut.

1b	!G.Undirected_Graph G==>((2*(number_of_edges G))=(sum_of_degree(getEdges G))) Undirected_Graph G==>((2*(number_of_edges G))=(sum_of_degree(getEdges G)))	STRIP_TAC
2b	Undirected_Graph G==>((2*(number_of_edges G))=(sum_of_degree(getEdges G))) ((2*(number_of_edges G))=(sum_of_degree(getEdges G))) --- Undirected_Graph G	STRIP_TAC
3b	((2*(number_of_edges G))=(sum_of_degree(getEdges G))) {Undirected_Graph G} (2*(number_of_edges G))=(sum_of_degree(getEdges G))	WEAKEN_TAC funcTrue
4b	(2*(number_of_edges G))=(sum_of_degree(getEdges G)) (2*(number_of_edges (Graph_ p)))=(sum_of_degree(getEdges (Graph_ p)))	Cases_on \`G\`
5b	(2*(number_of_edges (Graph_ p)))=(sum_of_degree(getEdges (Graph_ p))) (2*(number_of_edges (Graph_ (q,r))))=(sum_of_degree(getEdges (Graph_ (q,r))))	Cases_on \`p\`
6b	(2*(number_of_edges (Graph_ (q,r))))=(sum_of_degree(getEdges (Graph_ (q,r)))) 2 * LENGTH r = sum_of_degree r	REWRITE_TAC[NUMBER_OF_EDGES,GET_EDGES]
7b	2 * LENGTH r = sum_of_degree r !h. 2 * LENGTH (h::r) = sum_of_degree (h::r) -- 2 * LENGTH r = sum_of_degree r 2 * LENGTH [] = sum_of_degree []	Induct_on \`r\`
8b	!h. 2 * LENGTH (h::r) = sum_of_degree (h::r) -- 2 * LENGTH r = sum_of_degree r 2 * LENGTH [] = sum_of_degree [] T	REPEAT (ARW_TAC[LENGTH, SUM_OF_DEGREE])

Pembuktian manual 1a menggunakan instansiasi universal untuk menghilangkan ∀-kuantifikasi (∀*G* dalam Teorema 2.2.1) sehingga pernyataan

∀*G*. Undirected_Graph(*G*) → ((2 * number_of_edges(G)) = (sum_of_degree(getEdges(*G*))))

diubah menjadi pernyataan

Undirected_Graph(*G*) → ((2 * number_of_edges(*G*)) = (sum_of_degree(getEdges(*G*)))).

Pembuktian manual tersebut dipetakan menjadi pembuktian mekanis 1b. Pada pembuktian mekanis 1b, digunakan Taktik `STRIP_TAC` untuk menghilangkan ∀-kuantifikasi (`!G` dalam Teorema 2.2.1) sehingga pernyataan

`!G.Undirected_Graph G==>((2*(number_of_edges G))=(sum_of_degree(getEdges G)))`

diubah menjadi pernyataan

`Undirected_Graph G==>((2*(number_of_edges G))=(sum_of_degree(getEdges G)))`.

Untuk pemetaan pembuktian manual dengan pembuktian mekanis lainnya mengikuti pola yang sama dengan pemetaan 1a menjadi 1b.