Penerapan Algoritma Runut-balik dalam *Register Allocation* pada *Compiler Design*

David Karel Halomoan - 13520154
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13520154@std.stei.itb.ac.id

***Abstract*— *Compiler* merupakan salahs satu alat yang memepermudah kehidupan manusia pada zaman modern ini. *Compiler* mempermudah kegiatan *programmer* dengan melnerjemahka bahasa tingkat tinggi yang mudah dimengerti ke bahasa mesin yang sulit dimengerti. Salah satu permasalahan yang dihadapi dalam *compiler design* adalah alokasi memori ke dalam register (*register allocation*). Makalah ini membahas penyelesaian *register* allocation melalui pendekatan *graph coloring*. Strategi algoritma Balik-runut digunakan untuk menyelesaikan persoalan *graph coloring* ini. Penulis juga menyediakan *link repository* implementasi kode dalam Bahasa Java.**

Keywords—backtracking, compiler, Java, graph

I. Pendahuluan

Perkembangan ilmu komputer dan kekuatan komputer telah memudahkan kita memecahkan berbagai masalah. Salah satu alat yang digunakan untuk memudahkan penggunaan komputer adalah *compiler*. *Compiler* melakukan translasi bahasa tingkat tinggi menjadi bahasa yang dapat dimengerti dengan lebih mudah oleh mesin. Hal ini tentu memudahkan pembuatan suatu program karena pembuatan program dapat dilakukan dengan bahasa tingkat tinggi (dibandingkan dengan bahasa mesin yang lebih sulit dimengerti).

Salah satu teknik yang digunakan yang digunakan oleh *compiler* adalah *register allocation*. *Register allocation* adalah proses menempatkan (*assigning*) suatu variabel ke register dan mengelola transfer data yang masuk dan keluar register.

Pada makalah ini, penulis akan membahas pengimplemetasian *register allocation* dengan teknik *graph coloring*. Teknik ini akan diimplementasikan dengan algoritma algoritma runut-balik.

II. Landasan Teori

A. Algoritma Runut-balik

Algoritma runut-balik dapat dipandang sebagai *exhaustive search* yang diperbaiki. *Exhaustive search* melakukan eksplorasi semua kemungkinan solusi dan mengevealuasi solusi-solusi tersebut satu per satu. Algoritma runut-balik, pilihan yang dieksplorasi hanya pilihan yang mengarah ke solusi, pilihan yan tidak mengarah ke solusi tidak dipertimbangkan lagi. Algoritma memangkas (*pruning*) simpul-simpul yang tidak mengarah ke solusi.

Algoritma ini pertama kali diperkanalkan oleh D. H. Lemer pada tahun 1950. Uraian umum algoritma ini disajikan oleh R. J Walker, Golomb, dan Baumert.

Algoritma ini juga dapat dipandang sebagai sebuah fase dalam algoritma traversal DFS (*Depth First Search*) atau sebagai sebuah metode pemecahan masalah yang mangkus, tersetruktur, dan sistematis, baik untuk persoalan optimasi maupun non-optimasi.

Properti umum algoritma runut-balik:

1. Solusi persoalan

   Solusi dinyatakan sebagai vektor dengan *n-tuple*:

   $$X = (x_1, x_2, \ldots, x_n), x_i \in S_i$$

   Umumnya

   $$S_1 = S_2 = \cdots = S_n$$

2. Fungsi pembangkit nilai x_k

   Dinyatakan sebagai perdikat *T()*.

   $$T(x[1], x[2], \ldots, x[k-1])$$

   Ekspresi di atas membangkitkan nilai x_k, yang merupakan komponen vektor solusi.

3. Fungsi pembatas (*bounding function*)

   Dinyatakan sebagai predikat $B(x_1, x_2, \ldots, x_n)$.

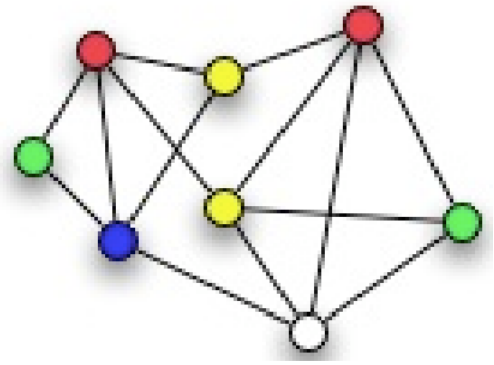
   B bernilai *true* jika $(x_1, x_2, \ldots, x_n)$ mengarah ke solusi. Mengarah ke solusi di sini berarti tidak melanggar kendala (*constraints*).

   Jika B bernilai *true*, pembangkitan nilai untuk x_{k+1} dilanjutkan, tetapi jika *false*, $(x_1, x_2, \ldots, x_k)$ dibuang.

B. Pewarnaan Graf

Pewarnaan graf yang dimaksudkan di sini adalah pewarnaan simpul. Pewarnaan simpul di sini maksudnya

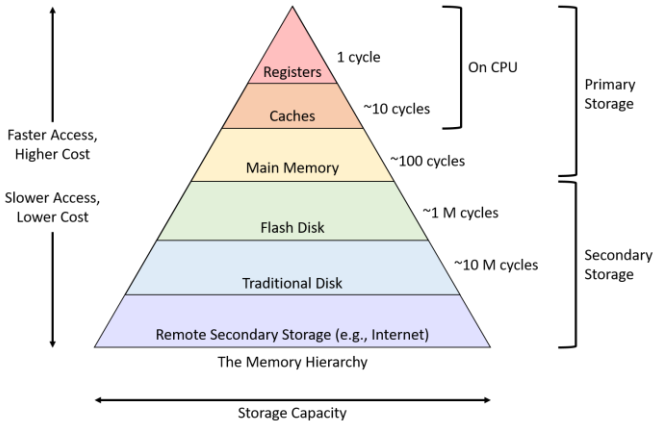
adalah pemberian warna pada simpul-simpul graf sedemikian sehingga dua simpul bertetangga mempunyai warna berbeda.



Gambar 2.1 Contoh pewarnaan pada suatu graf
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian3.pdf

C. *Register Allocation*

Register allocation adalah proses menempatkan (*assigning*) suatu variabel ke register dan mengelola transfer data yang masuk dan keluar register. Hal ini dilakukan karena memori yang relatif cepat memiliki kapasitas yang lebih sedikit. Dalam hal ini, memori yang akan digunakan adalah register. Register adalah memori tercepat dalam hirearki memori. Register adalah suatu penyimpanan lokal pada prosesor yang menyimpan data yang sedang diproses oleh CPU (*Central Processing Unit*). Register biasanya diukur dalam ukuran bit untuk menentukan jumlah data yang dapat disimpan dalam register tesebur. Misal prosesor 32-bit berarti ukuran register pada prosesor tersebut 32 bit dan prosesor 64-bit berarti ukuran register pada prosesor tersebut adalah 64 bit.



Gambar 2.2 Hirearki Memori
Sumber:
https://diveintosystems.org/book/C11-MemHierarchy/mem_hierarchy.html

Seorang *programmer* menulis suatu program dengan tingkat tinggi dan menggunakan variabel (bernama bebas, misal: *a, b, c, x1, y*). Saat *compiler* mengubah bahasa tingkat tinggi tersebut menjadi bahasa mesin, *compiler* akan "menggantikan" variabel-variabel tersebut dengan register. Ini terjadi pada fase generasi kode pada proses kompilasi.

Register allocation biasanya dilakukan oleh pihak yang melakukan *compiler design* dengan bantuan *liveness analysis*. *Liveness analysis* terdiri dari teknik yang diimplementasikan untuk mengoptimisasi alokasi register *space*. Ini perlu dilakukan karena setiap *hardware*/mesin memiliki jumlah register yang terbatas untuk menampung variabel atau data yang sedang digunakanatau dimanipulasi (oleh CPU). Hal ini menyebabkan adanya kebutuhan untuk menyeimbangkan alokasi memori yang efisien untuk mengurangi harga (sebab harga register relatf mahal disbanding memori lain, lihat gambar 2.2) dan juga tetap mempertahankan kekuatan mesin untuk meng-*handle* kode yang kompleks dan jumlah variabel yang banyak pada saat yang bersamaan. Hal ini dilakukan oleh *compiler* saat kompilasi kode masukan (*input*).

Beberapa istilah dalam *liveness analysis*:

- *Live variable* (variabel hidup): Sebuah variabel dikatakan "hidup" pada suatu saat, dalam proses kompilasi program jika nilainya digunakan untuk melakukan proses komputasi dari evaluasi suatu operasi aritmetik saat itu juga atau variabel tersebut mengandung nilai yang akan digunakan pada masa yang akan datang tanpa variabel tersebut didefinisikan kembali (diubah nilainya) sebelum penggunaan tersebut.
- *Live range*: *Live range* sebuah variabel didefinisikan sebagai bagian dari kode saat variabel tersebut "hidup". *Live range* dari variabel mungkin berkelanjutan atau tersebar dalam bagian kode yang berbeda-beda. Ini berarti sebuah variabel mungkin "hidup" pada suatu saat dan "mati" pada saat selanjutnya dan mungkin "hidup" lagi pada bagian tertentu selanjutnya. Definisi *live range* yang lebih sederhana akan diberikan pada bagian selanjutnya dalam makalh ini.

Makalah ini tidak membahas analisis variabel hidup (*liveness analysis*) secara detail. Jika pembaca ingin mengetahui lebih lanjut mengenai *liveness analysis* terutama dalam kaitannya dengan *compiler design*, dianjurkan untuk mencari melalui berbagai sumber yang tersedia di internet.

Contoh *register allocation*:

- Terdapat suatu program:

```
a := c + d
e := a + b
f := e - 1
```

  Diasumsikan variabel a dan e "mati" setelah digunakan.
- Variabel temporer a dapat "digunakan kembali" (telah mati) setelah baris e := a + b.

- Variabel temporer e dapat “digunakan kembali” (telah mati) setelah baris f := e – 1.
- Sehingga, variabel temporer a, e, dan f (f diikutkan karena tidak pernah digunakan) dapat dialokasikan ke dalam satu register (misal r1).
- Kode yang dihasilkan ditunjukkan di bawah dengan variabel dialokasikan ke register.

```
r1 := r2 + r3
r1 := r1 + r4
r1 := r1 + 1
```

- Nilai pada variabel temporer yang telah mati tidak diperlukan lagi untuk komputasi selanjutnya sehingga variabel tersebut dapat “digunakan kembali”.

III. Deskripsi Masalah

Selain ukurannya yang kecil, register juga memiliki harga yang relatif mahal (dibandingkan memori lain) dilihat dari gambar 2.2. Ini menyebabkan jumlah dan kapasitas register pada suatu *hardware* terbatas, dan bahkan sangat sedikit jika dibandingkan memori jenis lainnya. Walaupun *programmer* tidak mempunyai batasan untuk jumlah vaiabel yang dapat digunakan olehnya dalam suatu program, di pihak lain *hardware* memiliki memori yang terbatas. Tantangannya adalah mengatur cara agar register dengan jumlah terbatas pada suatu CPU bisa dialokasikan untuk jumlah variabel tak tentu yang digunakan oleh *programmer*. Register juga harus diatur agar tidak menghilangkan atau memodifikasi suatu nilai secara ilegal saat nilai tersebut sedang dibutuhkan oleh proses yang berjalan di CPU.

Dari penjelasan di atas, ditemukan definisi baru untuk *register* allocation, yaitu adalah proses menentukan nilai yang harus dimasukkan ke suatu register tertentu pada waktu yang tepat saat eksekusi program.

Ada banyak cara untuk mengatasai masalah ini, tetapi yang akan dibahas dalam makalh ini adalah pendekatan *register allocation* dengan *graph coloring*. Pada bagian selanjutnya akan dijelaskan mengenai pendekatan tersebut. Permasalahn *grah coloring* tersebut lalu akan diselesaikan dengan strategi algoritma runut-balik.

IV. Analisis dan Implementasi

A. Pendekatan Graph Coloring pada Register Allocation

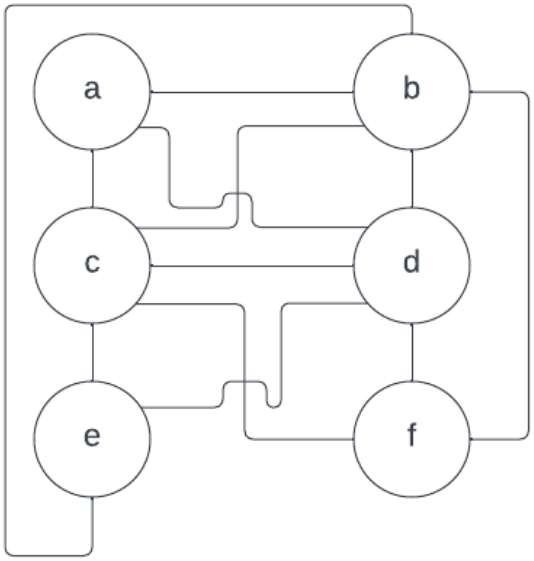
Register allocation adalah masalah yang sudah ada lama sekali. *Register allocation* yang digunakan dalam *compiler* FORTRAN orisinil pada tahun 1950-an menggunakan algoritma yang sangat “kasar”. Terobosan baru dalam pemecahan masalah ini baru dicapai pada tahun 1980 ketika Chaitlin menemukan skema *register allocation* berdasarkan teknik *graph coloring*. Teknik ini cukup sederhana dan bekerja dengan baik dalam prakteknya.

Ketika permasalahan ini diformulasikan sebagai *graph coloring*, setiap simpul (*node*) dalam graf melambangkan *live range* dari sebuah nilai tertentu. *Live range* didefiniskan sebagai rentang waktu dari penulisan (*write*) ke suatu register diikuti oleh semua penggunaan (pembacaan) register tersebut sampai register tersebut ditulis ulang kembali. Secara sederhana, simpul ini dibuat untuk setiap variabel temporer yang ada pada program.

Sisi (*edge*) di antara dua simpul menandakan kedua *live range* (kedua simpul tersebut) saling “mencampuri” (*interfere*) satu sama lain karena *lifetime* mereka tumpang-tindih (*overlap*). Secara sederhana, sisi antara variabel temporer *x* dan *y* berarti kedua variabel temporer tersebut “hidup” secara bersamaan pada suatu titik (*point* atau waktu) dalam eksekusi program.

Graf yang dihasilkan dari teknik ini adalah suatu graf tak berarah dan tak berbobot yang disebut sebagai *interference graph.*

Seperti yang sudah dijelaskan sebelumnya, dua variabel temporer yang hidup secara bersamaan tidak dapat dialokasikan ke dalam register yang sama (karena saat terjadi perubahan pada satu nilai dapat mempengaruhi nilai yang lain). Dari sini, diketahui dua variabel temporer dapat dialokasikan ke register yang sama jika tidak ada sisi yang menghubungkan mereka. Ini tentu sama dengan permasalah pada *graph coloring*, yaitu kedua simpul yang saling bertetangga tidak boleh memiliki warna yang sama. Dalam kasus *register allocation*, warna dilambangkan sebagai register. Warna yang berbeda dilambangkan sebagai register yang berbeda. Dari sini dapat disimpulkan, jika *interference graph* memerlukan *k* warna berbeda pada *graph coloring*, program yang direpresentasikan oleh *interference graph* tersebut memerlukan maksimum *k* register berbeda pada suatu titik dalam eksekusi program. Sebagai contoh, *interference graph* untuk program pada bab II bagian C adalah:

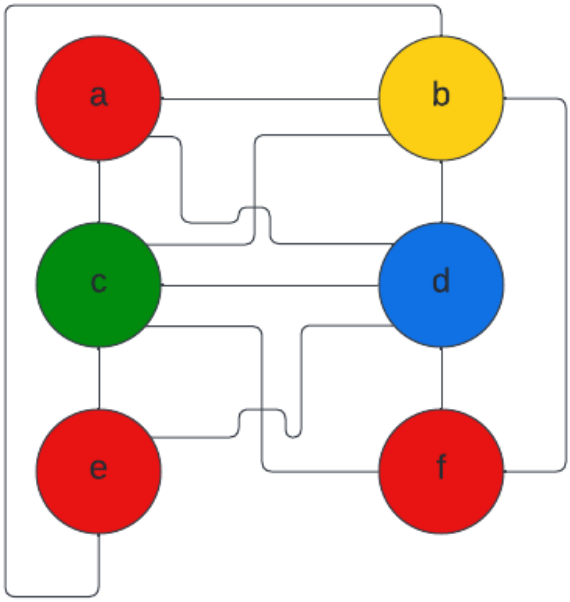


Gambar 4.1 Interference graph dari contoh program pada Bab II Bagian C
Sumber:
Dokumen pribadi penulis

Penjelasan dari gambar 4.1:

- c dan d hidup bersamaan (baris 1: a := c + d)
- a dan b hidup bersamaan (baris 2: e := a + b)
- c dan d dari baris 1 tidak "mati" setelah digunakan sehingga "hidup" bersama a dan b yang "hidup" pada baris 2
- b dari baris 2 tidak "mati" setelah digunakan sehingga "hidup" bersama e yang "hidup" pada baris 3 dan c dan d yang masih belum "mati", a setelah baris 2 "mati" karena telah digunakan (sesuai asumsi)
- f dari baris 3 tidak "mati" setelah digunakan sehingga "hidup" bersama c, d, dan b yang belum "mati", e setelah baris 3 "mati" karena telah digunakan (sesuai asumsi)

Interference graph lalu diberi warna sesuai prinsip *graph coloring* menjadi:



Gambar 4.2 Interference graph pada gambar 4.1 setelah melalui proses graph coloring
Sumber:
Dokumen pribadi penulis

Penjelasan warna pada gambar 4.2:

- Warna merah melambangkan register r1
- Warna hijau melambangkan register r2
- Warna biru melambangkan register r3
- Warna kuning melambangkan register r4

Penjelasan tersebut sesuai dengan penyelesaian contoh program pada Bab II bagian C.

B. Algoritma Runut-balik untuk Menyelesaiakan Persoalan Graph Coloring

Misal jumlah simpul pada graf adalah *n* dan jumlah warna *m*. Properti algoritma runut-balik untuk persoalan *graph coloring*:

1. Solusi persoalan

   $X = (x_1, x_2, ..., x_n), x_i \in S_i$

   $S_i = \{1, 2, ..., m\}$

   $x_i = 1$ *atau* 2 *atau ... atau* m

2. Fungsi pembangkit

   $T(x[1], x[2], ..., x[k-1])$ membangkitkan semua kemungkinan nilai untuk x_k (semua kemungkinan warna untuk simpul k), yang merupakan komponen vektor solusi.

3. Fungsi pembatas

   - B bernilai true jika $(x_1, x_2, x_3, \ldots, x_k)$ mengarah ke solusi. Dalam persoalan *graph coloring*, akan diperiksa x_k (solusi terbaru yang dibangkitkan) memiliki tetangga yang memiliki warna sama atau tidak.

- Jika x_k memiliki tetangga dengan warna sama, solusi (x_1, x_2, x_3, …, x_k) dibuang.
- Jika x_k tidak memiliki tetangga dengan warna sama, pembangkitan nilai untuk x_{k+1} dilanjutkan.

Berikut algoritma runut-balik untuk *graph coloring*:

- Masukan (*input*):
  1. Matriks ketetanggan *G[1..n, 1..n]*

     G[i, j] = true jika ada sisi *(i, j)*

     G[i, j] = false jika tidak ada sisi *(i, j)*
  2. Warna

     Dinyatakan dengan integer *1, 2, ..., m*
- Luaran (*output*)
  1. Tabel *X[1..n]*, yang dalam hal ini, *x[i]* ada;ah warna untuk simpul i.
- Algoritma (dalam notasi algoritmik):
  1. Inisialisasi *x[1..n]* dengan *0* untuk semua nilai pada *x*
  2. Panggil prosedur PewarnaanGraf(1) (parameternya bernilai 1)

```
procedure PewarnaanGraf(input k : integer)
{ Mencari semua solusi solusi pewarnaan graf; algoritma rekursif
  Masukan: k adalah nomor simpul graf.
  Luaran: jika solusi ditemukan, solusi dicetak ke piranti keluaran
}
Deklarasi
  stop : boolean

Algoritma:
  stop ← false
  while not stop do
      WarnaiSimpul(k)      {coba isi x[k] dengan sebuah warna}
      if x[k] = 0 then     {tidak ada warna lagi yang bisa dicoba,  habis}
         stop ← true
      else
          if k = n then                {apakah seluruh simpul sudah diwarnai?}
             write(x[1], x[2], …, x[k])    { cetak solusi }
          else
             PewarnaanGraf(k + 1)      {warnai simpul berikutnya}
      endif
   endif
  endwhile
```

Gambar 4.3 Prosedur PewarnaanGraf
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian2.pdf

```
procedure WarnaiSimpul(input k : integer)
{ Menentukan warna untuk simpul k
 Masukan: simpul ke-k
 Luaran: nilai untuk x[k]
}
Deklarasi
   stop, keluar : boolean
   j : integer

Algoritma:
  stop ← false
  while not stop do
     x[k] ← (x[k]+1) mod (m+1)     { bangkitkan warna untuk simpul ke-k}
     if x[k] = 0 then              {semua warna telah terpakai}
        stop ← true
     else
        {periksa warna simpul-simpul tetangganya}
        …
```

Gambar 4.4 Prosedur WarnaiSimpul bagian 1
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian2.pdf

```
        for j ← 1 to n do
           if (G[k, j])       {jika ada sisi dari simpul k ke simpul j}
              and             {dan}
             (x[k] = x[j])    {warna simpul k = warna simpul j }
           then
              exit loop       {keluar dari kalang}
           endif
        endfor

        if j = n+1 {seluruh simpul tetangga telah diperiksa dan
                    ternyata warnanya berbeda dengan x[k] }
        then
           stop ← true        {x[k] sudah benar, keluar dari kalang}
        endif


     endif
  endwhile
```

Gambar 4.5 Prosedur WarnaiSimpul bagian 2
Sumber:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian2.pdf

C. Penerapan Graph Coloring dengan Algoritma Runut-balik pada Persoalan Register Allocation

Untuk menerapakan algoritma runut-balik pada persoalan *register allocation*, *n* diganti dengan jumlah variabel temporer pada program dan *m* diganti dengan jumlah register yang tersedia pada hardaware. Jika program tidak mengeluarkan hasil, berarti harus dilakukan modifikasi pada program agar variabel temporer dapat dimuat ke dalam register yang ada atau *hardware* di-*upgrade* untuk meningkatkan jumlah register sesuai yang dibutuhkan.

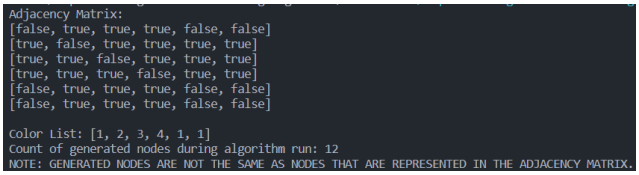
D. *Hasil Implementasi*

Algoritma diimplementasikan dengan program dalam bahasa Java. Prosedur PewarnaanGraf diimplementasikan menjadi sebuah fungsi Bernama ColorGraph yang mengembalikan *list of integer*. *List of integer* tersebut merepresentasikan warna untuk setiap simpul pada graf yang diwarnai. Prosedur WarnaiSimpul diimplemetasikan menjadi prosedur (fungsi dengan *return void* pada bahasa Java) Bernama ColorNode. Parameter fungsi dan prosedur tersebut juga disesuaikan sehingga berbeda dari notasi algoritma yang ada pada gambar 4.3, 4.4, dan 4.5.

Dilakukan pengecekan program terhadap graf *test case* yaitu graf yang berada pada gambar 4.1. Pengecekan dilakukan dengan merepresentasikan graf sebagai matriks ketetanggan / *adjacency matrix*. Berikut representasi simpul pada graf pada gambar 4.1 terhadap indeks pada matriks ketetanggan saat pengecekan:

- a: indeks 0
- b: indeks 1
- c: indeks 2
- d: indeks 3
- e: indeks 4
- f: indeks 5

Hasil eksekusi program terhadap graf tersebut sebagai berikut:



Gambar 4.6 Hasil eksekusi program terhadap graf pada gambar 4.1
Sumber:
Dokumen pribadi penulis

Hasil tersebut sesuai dengan gambar 4.2, yaitu simpul a, e, dan f memiliki warna yang sama (dalam program diberi angka 1 yang dalam gambar 4.2 sebagai warna merah) dan simpul b, c, d memiliki warna yang saling berbeda dan juga berbeda terhadap a, e, dan f (dalam program angka 2 yang dalam gambar 4.2 sebagai warna kuning, angka 3 sebagai warna hijau, angka 4 sebagai warna biru).

V. KESIMPULAN DAN SARAN

Register allocation merupakan salah stau persoalan yang sering dihadapi oleh orang-orang yang melakukan *design compiler*. Algoritma Runut-balik merupakan salah satu strategi algoritma yang memiliki efisiensi jauh lebih baik dari pada *brute force*. Dari analisis ditemukan bahwa persoalan *register allocation* dapat diselesaikan melalui pendekatan *graph coloring*. Persoalan *graph coloring* ini dapat diselesaikan dengan strategi algoritma runut-balik yang memiliki efisiensi yang baik. Strategi ini juga berhasil diimplementasikan ke dalam bahas Java tanpa mengurangi ketepatan algoritma. Diharapkan makalah ini dapat membantu pihan-pihak yang berhubungan dengan bidang *compiler design*. Makalah ini juga dapat menjadi pengaya ilmu bagi pihak-pihak yang tertaik dalam bidang *compiler design* dan strategi algoritma.

Untuk penelitian lebih lanjut, perbandingan dengan strategi algoritma yang lain dapat ditunjukkan (seperi *brute force*).

VI. UCAPAN TERIMA KASIH

Penulis mengucapkan puji syukur kepada Tuhan Yang Maha Esa, karena atas kasih, rahmat, dan karunia-Nya sehingga penulis dapat menyelesaikan makalah ini. Penulis juga mengucapkan terima kasih kepada Ibu Dr. Nur Ulfa Maliadevi, S.T, M.Sc dan Dr. Masayu Leylia Khodra, S.T, M.T., dosen kelas 1 dan 2 mata kuliah Strategi Algoritma IF2211 atas bimbingan dan pengajarannya selama ini. Penulis juga mengucapkan terima kasih kepada para penulis referensi makalah ini karena tanpa karya mereka, makalah ini tidak akan jadi.

VIDEO LINK AT YOUTUBE

https://youtu.be/EbVrQjfaaxc

LINK GITHUB

https://github.com/davidkarelh/Graph-Coloring-With-Backtracking-Algorithm

REFERENSI

[1] http://web.cecs.pdx.edu/~mperkows/temp/register-allocation.pdf

[2] https://www.geeksforgeeks.org/liveliness-analysis-in-compiler-design/

[3] *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian2.pdf*

[4] *https://diveintosystems.org/book/C11-MemHierarchy/mem_hierarchy.html*

[5] *https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian3.pdf*

[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 23 Mei 2022

Ttd
David Karel Halomoan
13520154