OPTIMALISASI PENELUSURAN GRAF MENGGUNAKAN ALGORITMA RUNUT-BALIK

Ricky Gilbert Fernando – 13505077

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl Ganesha 10, Bandung
E-mail: if15077@students.if.itb.ac.id

ABSTRAK

Makal ah ini menjelaskan penggunaan algoritma dalam proses penjelajahan graf. Graf ini dapat merepresentasikan banyak hal dalam kehidupan sehari-hari, seperti proses pengiriman surat elektronik dalam internet. Kita dapat menentukan banyak hal dalam penyelesaian masalah ini. Kita dapat mengambil proses optimal dari sebuah graf berarah. Mungkin saja kita ingin mendapatkan waktu minimum dalam proses pengiriman surat elektronik. Banyak hal yang dapat digambarkan oleh graf ini.

Kata Kunci : pseudo-code, sumber, tujuan, runut-balik, cariRute, cariRute2, *node*, fungsi.

1 PENDAHULUAN

Aplikasi dari masalah pencapaian suatu *node* dalam sebuah graf sangat banyak. Kita dapat membayangkannya sebagai navigasi dalam sebuah labirin satu arah. Atau kita ingin menggambarkan sebuah graf yang mirip atau berdekatan dengan graf yang kita gambar dan begitu juga sebaliknya. Atau kita membutuhkan perencanaan rute melalui suatu network atau pipelines. Atau mungkin kita dapat membayangkannya sebagai proses untuk mengirimkan pesan dari satu *node* ke *node* lain. Kita melihat bahwa semua contoh di atas mengandung suatu bagian yang sama, yaitu graf berarah. Tentu saja terdapat sejumlah *node* dan kumpulan tepi. Bagian tepi ini merepresentasikan suatu jalan penghubung satu arah antara 2 buah *node*. Penggunaan algoritma runut-balik merupakan sebuah solusi dalam memecahkan masalah-masalah di atas.

2 CERITA TENTANG RUNUT-BALIK

Algoritma runut-balik memiliki cerita tersendiri, yaitu benang Ariadne. Benang Ariadne adalah suatu kondisi untuk menyelesaikan masalah dengan menggunakan beberapa proses pendekatan, seperti labirin, logic puzzle yang mengharuskan kita melakukan pencarian untuk semua rute yang tersedia. Berikut adalah ceritanya:

> *Ariadne, dalam mitologi Yunani, adalah putri Minos, raja Crete dan Pasiphae, putri Helios, dewa matahari. Theseus datang ke Crete sebagai 1 dari 14 korban yang dibutuhkan rakyat Athena sebagai persembahan kepada Minotaur, makhluk setengah banteng setengah manusia yang tinggal dalam labirin. Ketika Ariadne melihat Theseus, dia jatuh cinta padanya dan berusaha membantunya jika Theseus berjanji untuk menikahinya. Lalu Ariadne memberikan Theseus sebuah bola benang yang didapat dari Daedalus, pencipta labirin. Theseus mengikat benang pada pintu masuk lalu memanjangkannya selama dia berjalan dalam labirin. Theseus berhasil bertemu dengan Minotaur dan membunuhnya. Dia mampu keluar dari labirin dengan menggulung kembali benang tersebut.* (Diterjemahkan dari Microsoft Encarta 2006)

Kunci dari aplikasi benang Ariadne adalah penciptaan dan penyimpanan dari proses-proses yang telah dilalui dalam menjalani setiap pilihan yang tersedia setiap saat. Kita menyimpannya untuk keperluan proses runut-balik yang menolak keputusan sebelumnya dan mencoba keputusan lain.

3 PROSES PENELUSURAN

3.1 Apakah Algoritma Runut-Balik

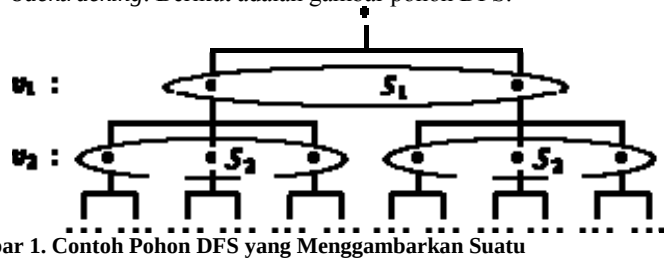
Runut-balik adalah perbaikan dari algoritma brute-force, yang secara sistematik mencari sebuah solusi dari semua pilihan yang tersedia (dari Wikipedia). Proses tersebut dilakukan dengan merepresentasikan semua kemungkinan sebagai sebuah vektor dan dengan penelusuran proses akan

mencari semua kemungkinan jawaban hingga solusi ditemukan. Ketika dijalankan, algoritma mulai dari vektor kosong. Dalam setiap tahapan, proses akan memasukkan sebuah nilai dalam vektor (v_1,...,v_m). Ketika proses mendapatkan fakta bahwa vektor yang sedang dicoba (v_1,…,v_i) tidak mampu mengarah pada vektor solusi maka proses akan melakukan runut-balik dengan menghilangkan nilai terakhir dan mencoba nilai baru yang masih mungkin. Proses ini dilakukan hingga solusi ditemukan atau tidak ada solusi yang mungkin menyelesaikan masalah.
Berikut adalah pseudo-code dari proses runut-balik di atas :

```
ALGORITHM try(v1,...,vi)
   IF (v1,...,vi) adalah
solusi THEN RETURN (v1,...,vi)
   FOR setiap v DO
      IF (v1,...,vi,v)       merupakan
vektor yang mungkin  THEN
        sol = try(v1,...,vi,v)
        IF sol != () THEN RETURN sol
      END
   END
   RETURN ()
```

Jika S_i adalah domain dari v_i, maka S_1 x … x S_m adalah ruang solusi dari masalah. Kriteria validitas yang digunakan dalam pengecekan apakah vektor diterima atau tidak menentukan jumlah ruang yang dibutuhkan untuk dicari dan juga menentukan jumlah *resource* yang digunakan oleh algoritma.

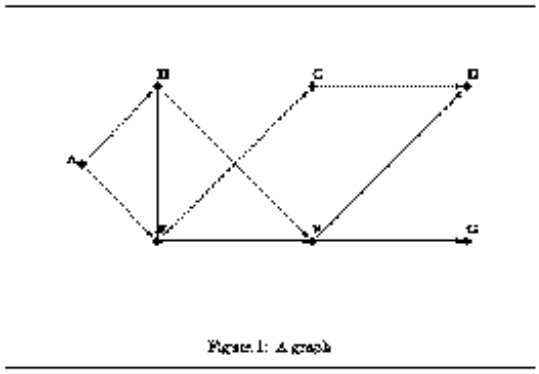
Proses penelusuran ruang solusi bisa direpresentasikan sebagai sebuah pohon DFS. Tidak semua pohon disimpan dalam algoritma, hanya *node* yang mengarah ke solusi saja yang disimpan untuk memungkinkan proses *backtracking*. Berikut adalah gambar pohon DFS:



Gambar 1. Contoh Pohon DFS yang Menggambarkan Suatu Masalah

3.2 Penelusuran Graf

Penelusuran graf merupakan sebuah proses pencapaian dari suatu *node* ke *node* tujuan. Berikut diberikan sebuah contoh :



Gambar 2. Contoh Graf Berarah

Jika kita ingin berangkat dari *node* C ke *node* D, rute yang harus ditempuh mudah untuk ditemukan, sebab terdapat penghubung antara kedua *node*. Lain halnya jika kita ingin bergerak dari *node* E ke *node* D. Terdapat dua pilihan yang mungkin:

1. Kita berangkat dari E ke F lalu menuju D
2. Kita berangkat dari E ke C lalu menuju D

Untuk beberapa kasus, mustahil untuk menghubungkan 2 buah *node*. Dalam contoh yang diberikan, kita tidak mungkin bergerak dari *node* C ke *node* G dengan melihat arah-arahnya. Yang menjadi pertanyaan sekarang adalah bagaimana kita mengetahui apakah 2 buah *node* dapat terhubung atau tidak.

Dalam keadaan riil, graf mempunyai lebih dari 7 *node* dan banyak tepi. Sangat normal jika kita mengembangkan algoritma dalam menemukan rute dalam graf. Kita dapat mencoba menggunakan list sebagai suatu representasi baru. Berikut adalah representasi list dari Gambar 2:

```
(define Graph
  '((A (B E))
    (B (E F))
    (C (D))
    (D ())
    (E (C F))
    (F (D G))
    (G ())))
```

Langkah berikut dalam menelusuri graf dan mencari jalan yang mungkin dalam *node* adalah menentukan fungsi yang akan mencari rute dalam graf:

```
;; cariRute : node node graf -> (list
of node)
```

```
;; untuk mencari jalan dari sumber ke
tujuan dalam graf G
(define (cariRute sumber tujuan G)
        ;; implementasi algoritma
)
```

Berikut adalah penulisan formula jika kita ingin bergerak dari *node* C ke *node* D

```
(find-route 'C 'D Graph)
```

Karena solusi dari masalah di atas adalah *node* C dan D itu sendiri maka jawaban yang diberikan adalah `(list 'C 'D)`.

Bagaimana dengan solusi dari persoalan kedua, jika kita ingin bergerak dari *node* E ke *node* D. Satu hal yang harus kita lakukan adalah mencari terlebih dahulu tetangga dari E. Dari Gambar 2, *node* tetangga dari E adalah C dan F. Karena kita akan melakukan pengecekan terlebih dahulu terhadap *node* C, kita bergerak ke *node* C. Kita melihat bahwa *node* tetangga dari C adalah *node* D dan hal ini berarti solusi sudah ditemukan, yaitu `(list 'E 'C 'D)`.

Lalu bagaimana untuk persoalan terakhir? Kita ingin bergerak dari *node* C ke *node* G. Kita tahu bahwa tidak ada rute yang mungkin membawa kita dari C ke G. Salah satu pilihan yang mungkin adalah mengembalikan nilai false yang menyatakan tidak ada rute yang berasal dari C menuju G. Berikut adalah beberapa perubahan yang telah kita lakukan:

```
;; cariRute : node node graph ->
(listof node) or false
;; untuk mencari jalan dari sumber ke
tujuan dalam graf G
;; jika tidak ada jalan yang mungkin,
fungsi mengembalikan nilai false
(define (cariRute sumber tujuan G)
        ;; implementasi algoritma
)
```

Langkah selanjutnya adalah memahami 4 komponen penting fungsi tersebut, yaitu kondisi "*trivial problem*", solusi yang sesuai, pengembangan untuk permasalahan baru dan langkah-langkah kombinasi. Dari contoh-contoh di atas, kita bisa mengambil kesimpulan, pertama komponen sumber dan tujuan mempunyai tipe yang sama, *node*, sedangkan jawaban yang dibutuhkan adalah list of *node*. Poin kedua adalah jika argumennya berbeda. Kita harus mengecek semua *node* tetangga dari sumber dan menentukan rute mana yang kita pilih untuk menuju tujuan. Persoalan akan menjadi rumit ketika jumlah *node* tetangganya banyak. Kita membutuhkan fungsi tambahan. Fungsi ini bertugas untuk menghasilkan list of *node* dan untuk menentukan 1 dari *node* tetangga yang merupakan rute untuk menuju *node* tujuan. Kita akan membuat sebuah fungsi yang berbeda dari cariRute dan kita menamakannya cariRute2. Berikut adalah pseudo-code-nya:

```
;; cariRute2 : (listof node) node
graph -> (listof node) or false
;; untuk mencari jalan dari node
sumberLain ke tujuanLain
;; Jika tidak ada rute yang mungkin,
kembalikan nilai false
(define (cariRute2 sumberLain
tujuanLain G)
        ;; implementasi algoritma
)
```

Sekarang kita akan menggabungkan kedua pseudo-code di atas menjadi satu bagian dalam fungsi cariRute:

```
(define (cariRute sumber tujuan
G)
  (cond
    [(== sumber tujuan) (list
tujuan)]
    [else ... (cariRute2
(tetangga sumber G) tujuan G)
...]))
```

Fungsi tetangga mengembalikan semua list of *node* dari tetangga dari *node* sumber.

Langkah berikutnya adalah kita harus menentukan keluaran dari fungsi cariRute2. Jika ditemukan rute dari tetangganya, fungsi ini mengembalikan rute dari tetangganya ke *node* tujuan. Jika tidak ada satupun tetangganya yang memberikan rute ke tujuan maka fungsi ini mengembalikan nilai false. Kita dapat melihat bahwa keluaran dari fungsi cariRute2 sangat penting dalam proses pencarian rute dalam fungsi cariRute. Oleh karena itu, kita menggunakan ekspresi-cond untuk memudahkan penggunaannya:

```
(define (cariRute sumber tujuan G)
  (cond
    [(== sumber tujuan) (list
tujuan)]
    [else (local ((define tempRute
                    (cariRute2
(tetangga sumber G)

tujuan G)))
            (cond
              [(tempRute) ...]
```

```
              [else ; (cons?
tempRute)
                ...]))]))
```

Kita melihat bahwa fungsi cariRute2 memberikan 2 tipe kembalian, yaitu list of *node* atau boolean false. Kondisi false didapat ketika tidak ada tetangga dari *node* sumber sehingga tidak mungkin menuju *node* tujuan. Untuk kembalian kedua, cariRute2 mengembalikan sebuah list of *node* dari tetangga sumber menuju tujuan. Sehingga ketika tetangga sumber menuju tujuan, kita menambahkannya ke dalam tempRute. Berikut adalah pseudo-code lengkap dari fungsi cariRute dan cariRute2 :

```
;; cariRute : node node graf ->
(listof node) or false
;; untuk mencari jalan dari sumber ke
tujuan dalam graf G
;; jika tidak ada jalan yang mungkin,
fungsi mengembalikan nilai false
 (define (cariRute sumber tujuan G)
  (cond
    [(== sumber tujuan) (list
tujuan)]
    [else (local ((define tempRute
                    (cariRute2
(tetangga sumber G) tujuan G)))
            (cond
              [(== tempRute false)
false]
              [else (cons sumber
tempRute)]))]))

;; cariRute2 : (listof node) node
graf -> (listof node) or false
;; untuk mencari jalan dari node
sumberLain ke tujuanLain
;; Jika tidak ada rute yang mungkin,
kembalikan nilai false
 (define (cariRute2 tetanggaSumber
tujuan G)
  (cond
    [(isEmpty tetanggaSumber) false]
    [else (local ((define tempRute
(cariRute (car tetanggaSumber) tujuan
G)))
            (cond
              [(== tempRute false)
(cariRute2 (cdr tetanggaSumber)
tujuan G)]
              [else tempRute]))])))
```

3.3 Penjelasan Pseudo-Code

cariRute merupakan fungsi dengan masukan 2 buah *node* dan sebuah graf. *Node* pertama adalah *node* sumber, sedangkan yang lain adalah *node* tujuan. Graf merupakan graf dari problem.

Kembalian dari fungsi cariRute adalah list of *node* yang merupakan deretan *node* rute dari *node* sumber ke *node* tujuan. Pertama-tama fungsi cariRute akan mengecek apakah *node* sumber sama dengan *node* tujuan. Jika ya, bentuk list yang terdiri dari *node* tujuan.

```
    [(== sumber tujuan) (list
tujuan)]
```

Jika tidak, kita membentuk sebuah list of *node* bernama tempRute yang terdiri dari kembalian fungsi cariRute2 dengan masukan tetangga dari *node* sumber ke *node* tujuan dalam graf G.

```
    [else (local ((define tempRute
                    (cariRute2
(tetangga sumber G) tujuan G)))
```

Jika tempRute mengembalikan nilai false maka,kembalikan nilai false, di sini proses runut-balik akan terjadi. Jika tidak, gabungkan *node* sumber dengan tempRute, lalu kembalikan list of *node* tersebut.

```
            (cond
              [(== tempRute false)
false]
              [else (cons sumber
tempRute)]))])))
```

Sekarang kita akan membahas cariRute. Fungsi cariRute2 bertujuan untuk mencari rute dari list of *node* ke *node* tujuan dalam graf G. Kembaliannya adalah rute dari tetangga *node* sumber ke *node* tujuan. Jika tidak ada tetangga dari *node* sumber maka kembalikan false.

```
    [(isEmpty tetanggaSumber) false]
```

Jika memiliki tetangga, program membentuk list of *node* bernama tempRute yang merupakan fungsi cariRute dengan *node* sumbernya adalah 1 tetangga dari *node* sumber. Jika tidak ada tempRute yang menuju *node* tujuan, maka kita mencoba kembali fungsi cariRute2 dengan masukan list of *node*nya adalah sisa dari *node* tetangga yang telah dicoba. Di sini proses runut-balik berlangsung. Jika tempRute memiliki jalan yang mungkin, kembalikan tempRute.

```
    [else (local ((define tempRute
(cariRute (car tetanggaSumber) tujuan
G)))
            (cond
              [(== tempRute false)
(cariRute2 (cdr tetanggaSumber)
tujuan G)]
              [else tempRute]))]))
```

4 KESIMPULAN

Penelusuran graf dengan menggunakan algoritma runut-balik mempunyai keuntungan, yaitu waktu terburuk yang mungkin terjadi adalah Θ(n!), yaitu ketika tidak ada solusi yang mungkin. Beberapa kerugiannya adalah penggunaan *resource* yang cukup besar sebab menggunakan algoritma rekursif.

Inti dari algoritma runut-balik adalah penggunaan memori untuk melihat kembali setiap jalan yang telah dilalui dan ketika menghadapi jalan buntu, kita dapat kembali ke titik pengambilan keputusan dan mencoba keputusan lain yang mungkin menuju solusi. Lebih baik kita merepresentasikan masalah sebagai sebuah graf atau sebuah pohon DFS sehingga kita lebih mudah menganalisis dan mencari solusi dari persoalan yang diberikan.

REFERENSI

[1] Munir, Rinaldi, "Strategi Algoritmik", ITB, 2007.
[2] http://en.wikipedia.org/ Tanggal akses: 16 Mei 2007 pukul: 15.00
[3] http://www.htdp.org/ Tanggal akses: 16 Mei 2007 pukul 15.00