OPTIMASI ALGORITMA POHON MERENTANG MINIMUM KRUSKAL

Karol Danutama / 13508040

Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jalan Selat Bangka IV no 6 Duren Sawit Jakarta Timur
e-mail: karoldanutama@gmail.com

MEMANGKAS KOMPLEKSITAS

Dalam definisi formal algoritma pohon merentang minimum Kruskal, disebutkan bahwa dalam proses penentuan apakah sebuah sisi termasuk ke dalam himpunan G = (V,E) di mana E adalah sisi yang membentuk pohon merentang minimum, harus diperiksa apakah pengambilan sisi tersebut menyebabkan pembentukan sirkuit dari himpunan G = (V,E). Proses pemeriksaan sebuah sirkuit dari sebuah graph memiliki kompleksitas waktu O(V) [1], sehingga algoritma Kruskal yang naif memiliki kompleksitas O(V) . O(E) = O(V E). Kompleksitas ini memiliki banyak masalah, terutama untuk graf lengkap, karena graf lengkap memiliki jumlah sisi mendekati V^2. Dalam makalah ini, akan dibahas langkah – langkah optimasi untuk mengurangi kompleksitas waktu algoritma Kruskal.

Kata kunci: Kompleksitas waktu, Pohon merentang minimum, Sirkuit, Optimasi

1. PENDAHULUAN

Dalam melakukan perancangan rangkaian digital, kita kerap kali harus menghubungkan semua pin – pin yang ada pada papan rangkaian agar sebuah pin dapat berkomunikasi dengan semua pin lain yang ada pada papan tersebut. Untuk menghubungkan n buah pin, maka diperlukan n – 1 buah kabel / koneksi agar persyaratan di atas terpenuhi. Namun, yang menjadi persoalan adalah jumlah kabel yang terbatas padahal antar pin terpisah oleh jarak. Sehingga, kerap kali perancangan yang menggunakan kabel paling sedikit yang akan digunakan dalam perancangan final.

Masalah di atas adalah ilustrasi dari masalah pohon merentang minimum. Cormen, Leiserson dan Rivest mengajukan sebuah model permasalahan di atas sebagai graph terhubung tak berarah G = (V,E), di mana V adalah himpunan pin yang ada pada papan sirkuit dan E adalah himpunan dari kemungkinan koneksi yang terbentuk antara dua buah pin. Untuk setiap sisi (u,v) anggota dari E memiliki bobot w(u,v) yang mendeskripsikan biaya / *cost* untuk menghubungkan pin u dan v. Kita ingin membentuk sebuah graf T yang merupakan sebuah pohon dan T adalah *improper subset* dar E yang menghubungkan semua sisi pada graf G dan total *w* untuk semua sisi yang ada di dalam T adalah minimum. Karena T adalah pohon dan mencakup semua simpul maka pohon T dikatakan **merentang** dan karena bobot *w* yang dilibatkan minimum maka T disebut pohon merentang minimum. Untuk selanjutnya, notasi T dan G akan digunakan untuk mendefinisikan pohon merentang minimum dari graf G.

Saat ini, terdapat dua algoritma pohon merentang minimum yang kerap digunakan, yaitu algoritma Prim dan algoritma Kruskal. Masing – masing algoritma memiliki ciri khas tersendiri tetapi memiliki kesamaan, yaitu menggunakan teknik *greedy* dalam prosesnya. Perbedaan mendasar dari kedua algoritma ini terletak pada proses pengambilan sisi untuk dimasukkan ke dalam pohon T.

Algoritma Kruskal adalah algoritma yang digunakan dalam lingkup kajian teori graf yang berfungsi untuk mencari pohon merentang minimum untuk graf terhubung berbobot G. Artinya, algoritma ini akan mencari himpunan bagian dari sisi yang membentuk graf G di mana himpunan bagian ini akan membentuk sebuah pohon yang melingkupi semua simpul yang terkandung pada graf G dengan jumlah bobot sisi yang ada di dalam himpunan bagian tersebut adalah minimum. Algoritma ini direka cipta oleh Joseph B Kruskal [2].

Pada bagian selanjutnya, akan dibahas mengenai pendefinisian formal algoritma Kruskal.

2. METODE

2.1 *Growing Forest*

Dasar pembentukan algoritma Kruskal berasal dari analogi *growing forest*. *Growing forest* maksudnya adalah untuk membentuk pohon merentang minimum T dari graf G adalah dengan cara mengambil satu per satu sisi dari graf G dan memasukkannya ke dalam pohon yang telah terbentuk sebelumnya. Seiring dengan berjalannya iterasi untuk setiap sisi, maka *forest* akan memiliki pohon yang

semakin sedikit. Oleh sebab itu, maka analogi ini disebut dengan *growing forest*. Algoritma Kruskal akan terus menambahkan sisi – sisi ke dalam hutan yang sesuai hingga akhirnya tidak akan ada lagi *forest* dengan, melainkan hanyalah sebuah pohon yang merentang minimum. Proses pembentukan pohon merentang minimum dari *growing forest* akan dibahas pada upabab berikutnya.

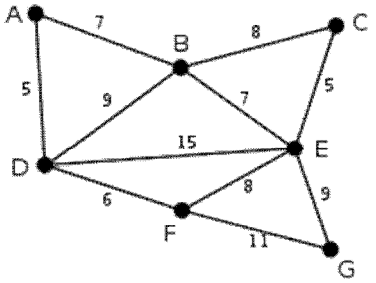
2.2. Pembentukan Pohon Merentang Minimum dari *Growing Forest*

Saat kondisi awal, untuk graf G = (V,E) algoritma Kruskal akan membentuk n buah pohon di mana n adalah jumlah simpul yang ada pada graf G dengan masing – masing pohon mengandung satu buah simpul. Kemudian, akan diperiksa satu per satu sisi dari graf G. Jika, pengambilan sebuah sisi menyebabkan dua buah pohon dapat digabungkan dan membentuk sebuah pohon dengan jumlah simpul adalah jumlah simpul pohon pertama dan kedua dengan jumlah sisi adalah jumlah sisi dari pohon pertama dan pohon kedua ditambahkan dengan 1, maka sisi tersebut boleh diambil. Selain itu, artinya pengambilan sisi tersebut tidak menggabungkan dua buah pohon melainkan membentuk sirkuit dar i sebuah pohon, sisi tersebut tidak boleh diambil. Sehingga, untuk setiap pengambilan sisi yang sukses akan mengurangi jumlah pohon dalam *forest* tersebut hingga akhirnya tidak ada lagi pohon yang bisa digabungkan.

Pembentukan pohon merentang minimum dapat dilakukan dengan cara melakukan manipulasi terhadap pemilihan sisi yang akan diambil. Pengambilan sisi dilakukan dari sisi yang memiliki bobot paling kecil terlebih dahulu secara terurut hingga ke sisi terakhir yang memiliki bobot paling besar.

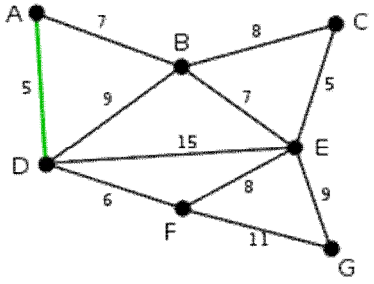
2.3. Ilustrasi Algoritma Kruskal

Pada kondisi awal, asumsikan kita memiliki graf G yang direpresentasikan dengan gambar berikut



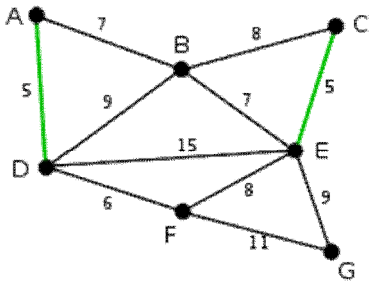
Gambar 1. Kondisi awal

Kemudian, sesuai dengan penjabaran di atas, maka pada kondisi awal dari graf ini akan terdapat 7 buah pohon dengan masing – masing memiliki sebuah simpul, sebutlah pohon a yang memiliki simpul A, pohon b yang memiliki simpul b dan seterusnya. Warna hitam pada sisi menunjukkan bahwa sisi tersebut belum pernah diproses. Pada kondisi awal, semua sisi belum pernah diproses.



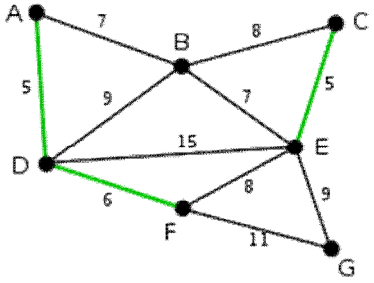
Gambar 2. Kondisi graf setelah penambahan sisi (A,D)

Kemudian, dipilih sisi (A,D) yang menghubungkan pohon a dengan pohon d. Sisi (A,D) dipilih pertama kali sebab sisi ini memiliki bobot terkecil dari antara semua sisi yang belum diproses dan sisi (A,D) menggabungkan dua pohon yang terbentuk. Sehingga, akan terbentuk pohon baru, katakanlah pohon ad yang memiliki simpul anggota A dan D. Warna hijau pada sisi menandakan sisi tersebut telah diproses dan diambil sebagai bagian dari pohon merentang minimum T.



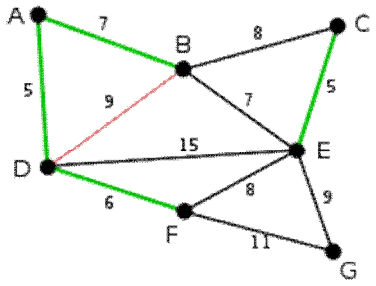
Gambar 3. Kondisi graf setelah penambahan sisi (C,E)

Sama seperti sebelumnya, sisi (C,E) dipilih karena memiliki bobot terkecil dari semua sisi yang belum diproses dan menghubungkan dua buah pohon. Sehingga, terbentuk pohon baru, yaitu pohon ce. Pada kondisi ini, sudah terbentuk lima buah pohon.



Gambar 4. Kondisi graf setelah penambahan sisi (D,F)

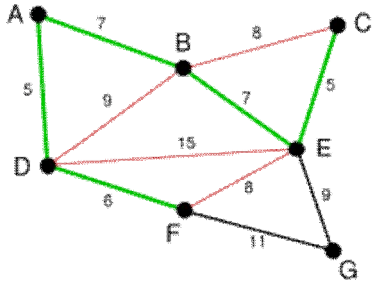
Sisi (D,F) kemudian dipilih karena memiliki kriteria yang sama, terbentuk pohon baru df.



Gambar 5. Kondisi graf setelah penambahan sisi (A,B) dan penolakan penambahan sisi (B,D)

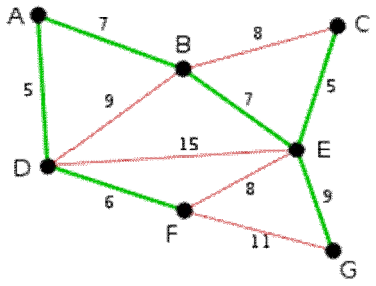
Sisi (A,B) dipilih sehingga membentuk pohon abdf. Saat ini terdapat 3 buah pohon, yaitu pohon abdf, ce dan g

Pada tahap selanjutnya, seharusnya sisi (B,D) diambil karena memiliki bobot terkecil dari semua sisi yang belum diproses. Namun, sisi (B,D) tidak boleh dipilih karena sisi BD tidak menggabungkan dua buah pohon, melainkan membentuk sirkuit dari pohon abdf. Sisi (B,D) bukan merupakan sisi pembentuk pohon merentang minimum.



Gambar 6. Kondisi graf setelah penambahan sisi (B,E) dan penolakan penambahan sisi (B,C) dan (E,F)

Sisi (B,E) dipilih dengan kriteria yang sama, bobot minimum dan menggabungkan pohon abdf dan ce, sehingga tersisa 2 buah pohon, yaitu pohon abcdef dan g. Sisi (B,C) dan (E,F) tidak dipilih karena membentuk sirkuit pada pohon abcdef.



Gambar 7. Kondisi akhir graf setelah penambahan sisi (E,G)

Terakhir, sisi (E,G) dipilih untuk membentuk sebuah pohon merentang minimum. Sisi (D,E) dengan bobot terbesar yang terakhir diproses tidak dipilih karena membentuk sirkuit.

2.4. Permasalahan Algoritma Kruskal

Seperti yang telah disebutkan pada bagian pendahuluan, algoritma Kruskal memilii kelemahan yang terletak pada saat proses pemeriksaan apakah penambahan sisi akan menyebabkan terbentuknya sirkuit dari pohon yang sudah terbentuk. Secara naif, mungkin memang bisa diperiksa dengan cara melakukan penelusuran dari pohon yang baru terbentuk. Untuk setiap penelusuran ini, menurut Cormen [1] memiliki kompleksitas waktu O(V E) dan penelusuran ini dilakukan setiap kali melakukan penambahan sisi ke dalam pohon. Artinya, kompleksitas totalnya adalah

$$O(V E)\, O(E) = O(V E^2) \qquad (1)$$

Untuk graf yang bersifat lengkap, kompleksitas total bisa menjadi sangat besar. Hal ini dimungkinkan terjadi sebab untuk graf lengkap, jumlah sisi dapat dihitung, dengan rumus:

$$E = \tfrac{1}{2}(V^2 - V) \qquad (2)$$

Sehingga, kompleksitasnya bisa dituliskan menjadi:

$$O(V E) = O(V \tfrac{1}{2}(V^2 - V)) = O(V^3) \qquad (3)$$

Untuk menangani permasalahan besarnya kompleksitas algoritma Kruskal untuk graf lengkap, maka harus dilakukan optimasi terhadap algoritma pemeriksaan sirkuit untuk meningkatkan kemangkusan algoritma Kruskal.

2.5. Optimasi dengan *Disjoint Set*

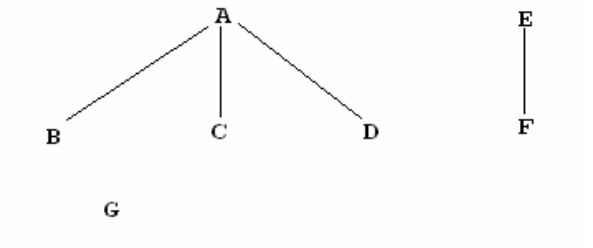
Jika melihat penjabaran algoritma Kruskal, terutama pada bagian pemeriksaan pembentukan sirkuit, kita dapat menarik sebuah lemma, yaitu: *pengambilan sisi akan menyebabkan jika dan hanya jika pengambilan sisi tersebut tidak menggabungkan dua buah pohon.* Dari lemma di atas, maka terdapat alternatif cara untuk memeriksa kesahihan pengambilan sebuah sisi, yaitu dengan memeriksa apakah pengambilan sisi tersebut menggabungkan dua buah pohon atau tidak. Jika pengambilan sisi tersebut menggabungkan dua buah pohon, maka sisi tersebut boleh diambil. Selain itu, pengambilan sisi tidak boleh dilakukan.

Untuk memeriksa apakah sebuah sisi menggabungkan dua buah pohon atau tidak, kita dapat memanfaatkan struktur data *disjoint set*. *Disjoint set* adalah struktur data yang di dalamnya terkandung himpunan – himpunan yang tidak saling beririsan dan segala operasi yang berkaitan dengan penambahan dan pengurangan elemen dari struktur data ini harus menjaga sifatnya yang mengandung himpunan – himpunan yang tidak saling beririsan [1].

Untuk algoritma Kruskal, struktur data ini dapat digunakan sebagai representasi *forest* dengan himpunan – himpunan pohon yang sedang terbentuk dari pengambilan sisi. Dengan ilustrasi di atas, pada kondisi awal struktur data *disjoint set* adalah himpunan {A} , {B} , {C} , {D} , {E} , {F} dan {G}

2.6. *Disjoint Set* dengan Struktur Pohon Berakar

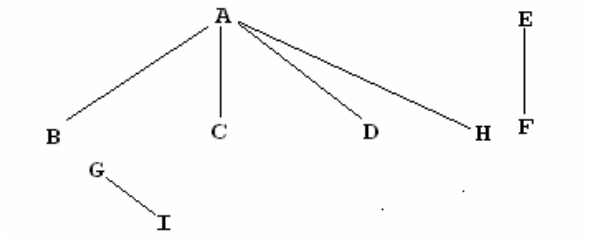
Struktur data *disjoint set* direpresentasikan dengan struktur pohon berakar, artinya adalah setiap himpunan memiliki struktur pohon yang memiliki akar. Misalnya, untuk himpunan {A,B,C,D}, {E,F} dan {G}, salah satu kemungkinan struktur datanya diilustrasikan oleh gambar berikut:



Gambar 8. Ilustrasi struktur data *disjoint set*

Dengan masing – masing himpunan memiliki akar A, E dan G. Untuk selanjutnya, nama akar akan digunakan untuk merepresentasikan nama himpunan.

Penambahan sebuah simpul / *node* dapat dilakukan dengan cara menambahkan simpul tersebut sebagai anak dari akar pohon yang ingin ditambahkan. Sebagai contoh, penambahan simpul H ke himpunan A dan simpul I ke himpunan G, maka struktur datanya adalah menjadi sebagai berikut:



Gambar 9. Ilustrasi penambahan simpul pada struktur data *disjoint set*

Dalam makalah ini, tidak akan dibahas operasi pengurangan simpul karena kurang relevan dengan bahasan algoritma Kruskal.

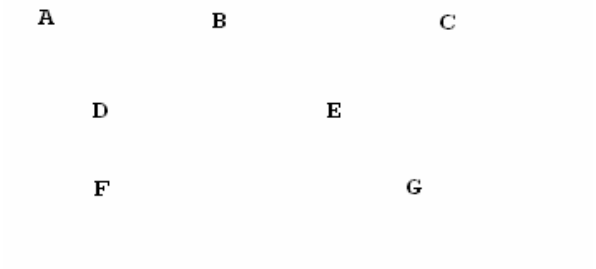
2.7. Penggunaan *Disjoint Set* pada Algoritma Kruskal

Seperti yang telah disebutkan sebelumnya, *disjoint set* digunakan untuk merepresentasikan pohon – pohon yang sedang terbentuk di dalam *growing forest*. Mengacu pada lemma yang telah disampaikan sebelumnya bahwa penambahan sisi yang tidak menyebabkan sirkuit maka penambahan sisi tersebut menggabungkan dua buah pohon, maka dapat disimpulkan bahwa jika sisi tersebut tidak menyebabkan sirkuit, simpul yang bersisian dengan sisi tersebut merupakan dua buah sisi yang berada di dalam dua **himpunan** yang berbeda. Dari lemma kedua ini, maka struktur data *disjoint set* dapat digunakan untuk memeriksa apakah sebuah sisi menggabungkan dua buah pohon atau tidak dengan memeriksa apakah simpul yang bersisian dengan sisi tersebut berada pada himpunan yang berbeda atau tidak. Jika ternyata kedua simpul tersebut berada pada dua buah himpunan yang berbeda, maka sisi tersebut boleh diambil.

2.8. Ilustrasi algoritma Kruskal dengan *Disjoint Set*

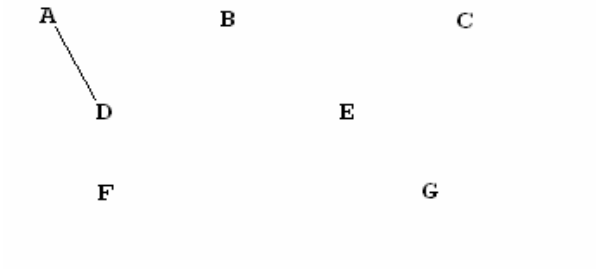
Mengacu pada contoh dalam upabab 2.3, saya akan kembali mencoba mengilustrasikan algoritma Kruskal, kali ini dengan struktur data *disjoint set*.

Pada kondisi awal, semua simpul tidak terhubung, artinya terdapat 7 buah himpunan dengan masing- masing 1 anggota



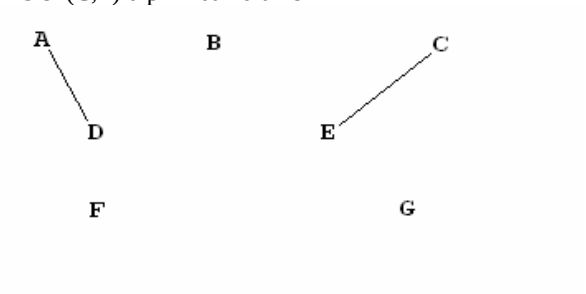
Gambar 10. Kondisi awal struktur *disjoint set*

Kemudian, pada langkah kedua, sisi (A,D) diambil, sehingga struktur data yang ada menjadi:



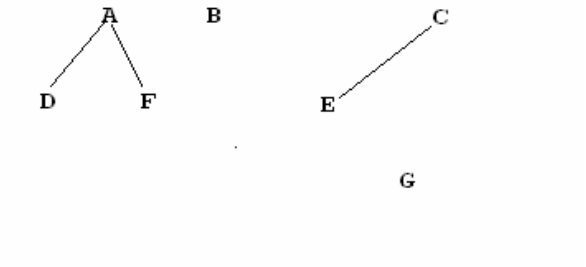
Gambar 11. Kondisi struktur *disjoint set* setelah penambahan sisi (A,D)

Sisi (C,E) dipilih dan diambil



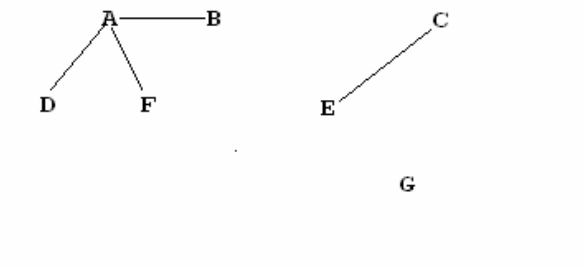
Gambar 12. Kondisi struktur *disjoint set* setelah penambahan sisi (C,E)

Sisi (D,F) diambil, F menjadi anak dari A karena akar dari D adalah A dan akar dari F adalah F



Gambar 13. Kondisi struktur *disjoint set* setelah penambahan sisi (D,F)

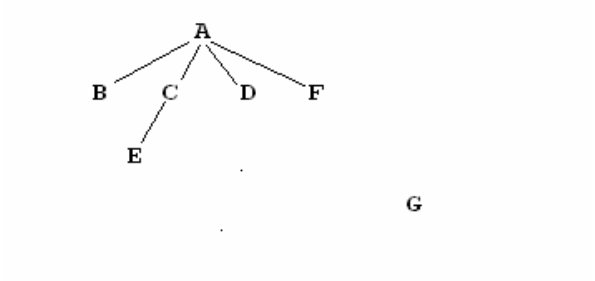
Sisi (A,B) diambil



Gambar 14. Kondisi struktur *disjoint set* setelah penambahan sisi (A,B) dan penolakan penambahan sisi (B,D)

Sisi (B,D) seharusnya diambil, tetapi tidak diperbolehkan sebab sisi B dan D sudah terdapat di dalam himpunan yang sama, yaitu himpunan A

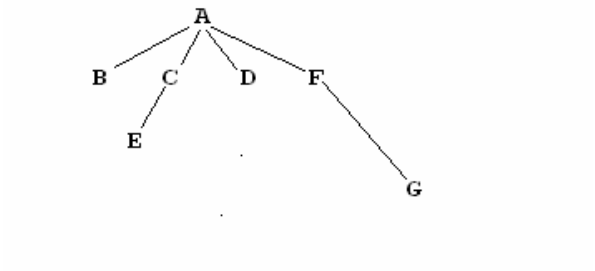
Kemudian sisi (B,E) diambil. Simpul B memiliki akar A dan simpul E memiliki akar C. Simpul C menjadi **anak** dari simpul A. Dalam kasus ini, terdapat alternatif cara penggabungan, yaitu dengan menjadikan A sebagai anak dari C karena kedalaman dari pohon A dan pohon C sama. Namun, jika kedalaman pohon yang akan digabungkan berbeda, maka pohon dengan kedalaman lebih kecil menjadi anak dari akar pohon yang memiliki kedalaman lebih besar.



Gambar 15. Kondisi struktur *disjoint set* setelah penambahan sisi (B,E)

Pengambilan sisi (B,C) dan (E,F) tidak diperbolehkan sebab simpul B,C,E dan F sudah terdapat di dalam himpunan A.

Sisi (E,G) diambil. Akar dari E adalah A dan akar dari G adalah G. Pohon A memiliki kedalaman 2 sedangkan pohon G memiliki kedalaman 0, oleh sebab itu simpul G menjadi anak dari A.



Gambar 16. Kondisi akhir struktur *disjoint set*

Selesai. Penambahan simpul (F,G) dan (D,E) tidak akan terjadi sebab semua simpul telah masuk ke dalam 1 himpunan.

2.9. Kompleksitas Algoritma Kruskal dengan *Disjoint Set*

Menurut Cormen, Leiserson dan Rivest, jika algoritma Kruskal diterapkan dengan struktur data *disjoint set*, kompleksitas yang secara naif dinyatakan sebagai $O(V E^2)$, dapat dipangkas menjadi $O(E\ ^2\log V)$. Kompleksitas $O(E\ ^2\log V)$ merupakan kompleksitas dalam kasus terburuk. Dalam kasus rata – rata, kompleksitas algoritma Kruskal dengan *disjoint set* adalah O(E), karena proses pemeriksaan sirkuit dengan *disjoint set* adalah nilai invers dari fungsi Ackermann di mana fungsi tersebut adalah fungsi yang peningkatan nilainya sangat lambat.

3. KESIMPULAN

Dari penjabaran mengenai algoritma Kruskal di atas, dapat dilihat bahwa pemilihan struktur data yang tepat dapat disimpulkan bahwa pemilihan struktur data yang tepat dapat memangkas kompleksitas sebuah algoritma. Dalam kasus ini, pemilihan struktur data *disjoint set*

daripada dengan struktur data graf biasa dapat meningkatkan kemangkusan algoritma Kruskal. Meskipun demikian, untuk menciptakan sebuah algoritma yang mangkus diperlukan banyak waktu dan usaha yang besar, sebab banyak sekali pembuktian yang harus melibatkan riset dan eksperimen.

Ucapan terima kasih saya sampaikan kepada para ilmuwan dalam bidang sains komputer yang telah menciptakan berbagai macam algoritma yang menakjubkan dan tidak terpikir sebelumnya. Tidak lupa syukur saya panjatkan kepada Tuhan YME atas kesempatannya untuk menyelesaikan makalah kajian ini.

REFERENSI

[1] Cormen, et.al, "Algorithms", The MIT Press, 1994

[2] "Kruskal's Algorithm", http://en.wikipedia.org/wiki/Kruskal%27s_algorithm, diakses pada 16 Desember 2009