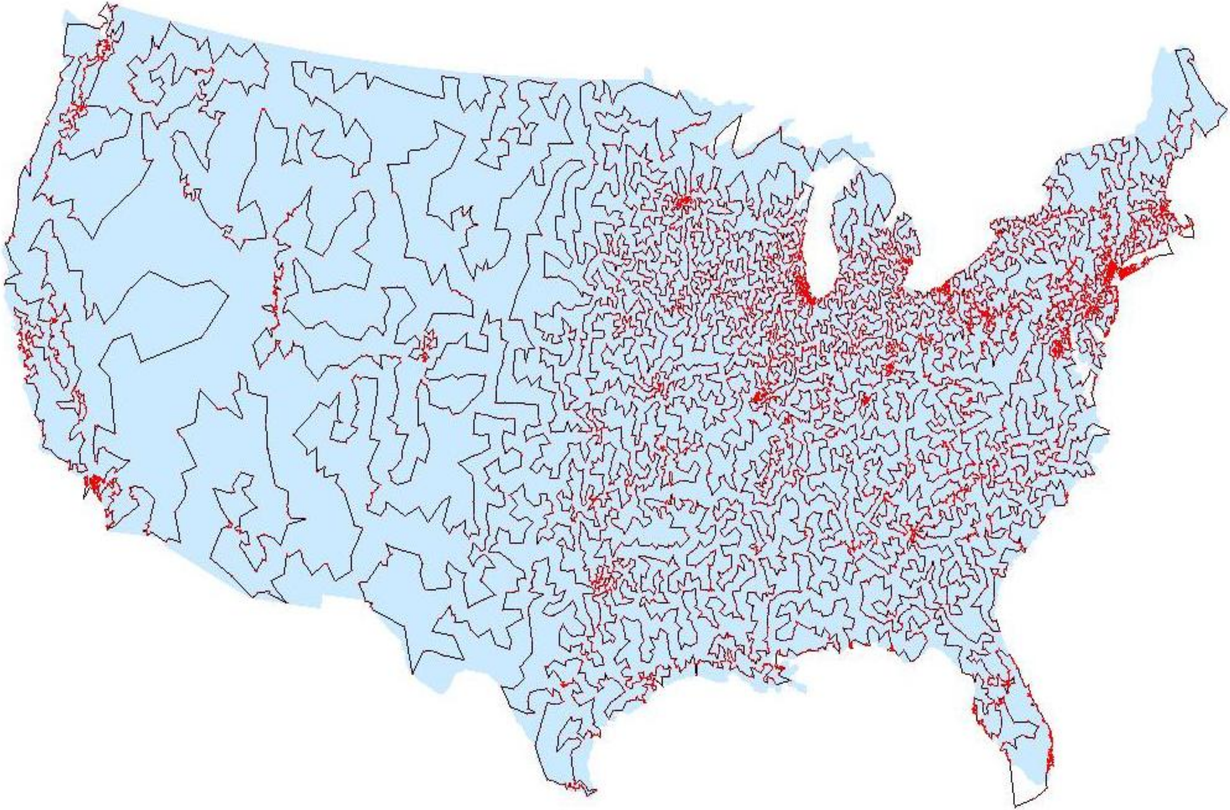
Sirkuit Hamilton, Pohon Pencarian, dan Algoritma dalam Penyelesaian *Travelling Salesman Problem*

Denny Nugrahadi – NIM : 13504054

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if14054@students.if.itb.ac.id



Abstrak

Makalah ini membahas tentang studi terhadap pohon dan graf serta implementasinya dalam pohon pencarian solusi untuk menyelesaikan *Travelling Salesman Problem*.

Travelling Salesman Problem merupakan salah satu permasalahan yang penting dalam dunia matematika dan informatika. Kenyataan bahwa sampai sekarang algoritma yang dipakai semuanya masih merupakan algoritma yang bersifat eksponensial adalah tanda bahwa solusi yang efektif masih harus dicari; minimal harus ditemukan algoritma yang bisa berada pada waktu asimptotik yang polinomial, karena sampai ditemukan solusi yang lebih baik, maka *Travelling Salesman Problem* akan sulit diselesaikan untuk kota tujuan yang banyak.

Prosedur untuk menyelesaikan sebuah *Travelling Salesman Problem* sangat berarti dalam berbagai bidang terapan; suatu contoh yang mendasar adalah dalam manufaktur *printed circuit* : bagaimana menjadwalkan rutedari mesin bor untuk mengebor lubang pada sebuah PCB. Pada permesinan dengan robot seperti ini, yang menjadi “kota” adalah bagian dari mesin atau kakas untuk dib or, dan “biaya perjalanan” adalah waktu yang dibutuhkan untuk melakukan *retooling* pada robot.

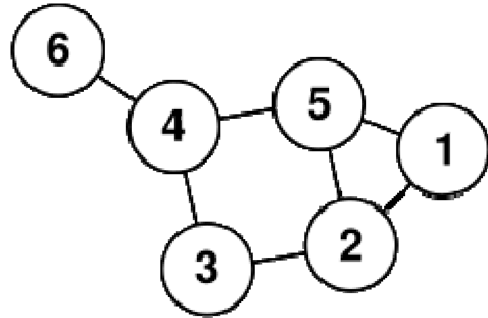
1. Pendahuluan

Sebelum memulai pembahasan lebih lanjut, pertama-tama haruslah dijelaskan terlebih dahulu mengenai apa yang dimaksud dengan *Travelling Salesman Problem*(TSP), atau dalam bahasa Indonesia disebut sebagai “Persoalan Pedagang Keliling”. Persoalan ini mouncul dari suatu pertanyaan: Bila diketahui sejumlah kota dan biaya-biaya untuk berpindah dari satu kota ke kota lainnya, bagaimana cara agar kita bisa mengunjungi tiap kota tersebut tepat sekali kunjungan dan kembali ke kota tempat kita berangkat, dengan biaya serendah mungkin?

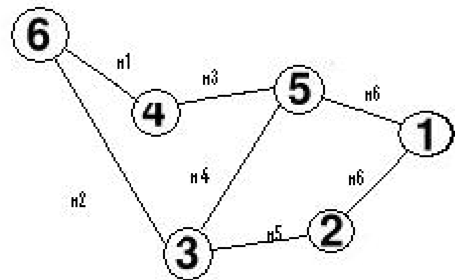
Persoalan ini bisa dideskripsikan ke dalam teorema graf sebagai berikut: Bila diketahui suatu graf berbobot penuh (di mana tiap simpul merepresentasikan kota, tiap sisi merepresentasikan tiap jalan dari satu kota ke kota lainnya, dan tiap bobot pada tiap sisi merepresentasikan biaya yang dibutuhkan untuk menempuh jalan dari satu kota ke kota lainnya), carilah Sirkuit Hamiton yang bobot totalnya paling kecil.

Jika begitu, apa yang dimaksud dengan Graf, dan apa itu Sirkuit Hamilton?

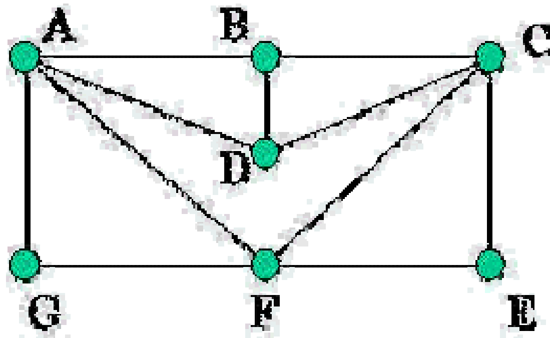
Secara gamblangnya, bisa dikatakan bahwa suatu graf adalah sekumpulan objek yang disebut simpul(*node* atau *vertice* dalam bahasa Inggris) yang mana dihubungkan oleh sambungan-sambungan yang disebut sebagai sisi(dalam bahasa Inggris: *lines* atau *vertices*). Dalam suatu graf(bila tidak didefinisikan, berarti yang dimaksud adalah graf tidak berarah, yaitu graf yang sisi-sisinya tidak mempunyai orientasi arah), suatu garis dari titik A ke titik B diasumsikan sama dengan garis dari titik B ke titik A. Pada graf berarah, kedua arah-arah tadi dihitung sebagai jalur yang berlainan. Umumnya, suatu graf digambarkan pada diagram sebagai sekumpulan titik(yang mewakili simpul) dihubungkan oleh garis ataupun kurva(yang mewakili sisi).



Ada suatu atribut tambahan yang kerap melengkapi suatu graf, yaitu bobot. Bobot merupakan label dari tiap sisi pada suatu graf. Bobot biasanya merupakan bilangan *real*. Kadang, tergantung pemakaian, bobot akan dibatasi hanya untuk bilangan rasional ataupun bilangan integer. Beberapa algoritma tertentu memerlukan batasan-batasan tambahan dalam nilai bobot; sebagai suatu contoh, algoritma *Djikstra* hanya bisa berfungsi secara benar bila bobot dari graf yang diproses bernilai positif. Bobot dari sebuah Alur pada sebuah graf berbobot adalah total dari bobot pada tiap sisi-sisinya. Bila sebuah graf disebut tanpa keterangan tambahan, biasanya graf tersebut tidak berbobot. Pada pembahasan-pembahasan matematis tertentu, istilah *network* (jaringan) bisa dianggap sebagai sinonim dari graf berbobot. Suatu *network* bisa memiliki arah atau tidak.



Suatu Alur Hamilton adalah suatu alur yang mengunjungi simpul masing-masing persisnya sekali. Suatu Siklus Hamilton, Sirkuit Hamilton, Perjalanan Atau Siklus Graf adalah suatu siklus yang mengunjungi puncak masing-masing tepat sekali (kecuali simpul yang merupakan tujuan awal maupun akhir, sehingga dikunjungi 2 kali). Suatu graf yang berisi suatu Siklus Hamilton disebut sebagai Graf Hamilton.



2. Kendala dalam menentukan penyelesaian *Travelling Salesman Problem* yang efisien

   Kendala yang paling utama adalah pada kenyataan bahwa algoritma-algoritma yang digunakan saat ini untuk menyelesaikan TSP mempunyai kompleksitas algoritma yang bisa digolongkan sebagai algoritma eksponensial, yaitu yang memiliki waktu asimptotik O(n!) ataupun $O(2^n)$. Sampai sekarang belum ditemukan algoritma yang berorde polinomial. Sebagai contoh, cara penyelesaian paling dasar, yaitu algoritma *brute force*, di mana kita mencoba mengenumerasikan semua kemungkinan yang ada dan memeriksa tap-tiap bobotnya, program harus memeriksa (n - 1)!/2 buah sirkuit Hamilton untuk tiap n buah simpul, sehingga untuk n yang bernilai 15 saja diperlukan kira-kira 4.358×10^{10} pengecekan, sungguh bukan suatu angka yang kecil.

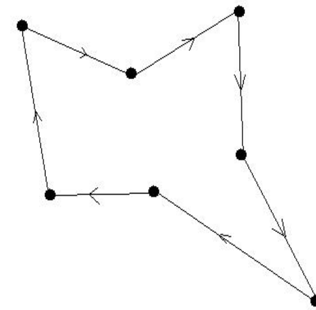
3. Solusi-solusi dalam pemecahan *Travelling Salesman Problem*

   - **Algoritma *Branch and Bound Insertion***

     Algoritma *branch and bound* didasarkan atas algoritma *insertion* yang harus diperhatikan dahulu. Algoritma ini didasarkan atas proses yang menurunkan perjalanan r-kota dari suatu perjalanan (r – 1)-kota. Dimulai dari permasalahan untuk 2 buah kota, proses ini diaplikasikan secara rekursif untuk menghasilkan solusi untuk n-kota secara komplit. Untuk mendapatkan solusi yang benar-benar optimum, *insertion algorithm* yang dipakai harus dimodifikasi agar lebih dari satu perjalanan r-kota yang diperiksa untuk tiap perjalanan (r - 1)-kota; untuk membatasi pencarian ini digunakan strategi *brach and bound*.

Untuk memahami cara pendekatannya maka pertama kita harus melihat cara untuk melakukan rekursi terbalik; yaitu, suatu prosedur yang berlangsung dari perjalanan r-kota ke perjalanan (r - 1)-kota.

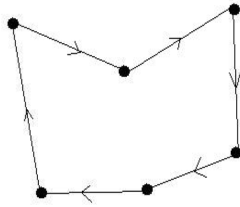
Misalkan kita adalah seorang pedagang keliling dan kita sedang melalui suatu rute perjalanan yang “bagus”, tidak harus optimal, untuk mengujungi r buah kota; kita sebut jalur ini sebagai Tour(r). Bila kita sekarang memutuskan bahwa kita tidak lagi merasa berkepentingan untuk melalui salah satu dari kota-kota tersebut - katakan saja kota B- bagaimana cara kita menyesuaikan rute kita?



Misalkan kalau dengan begitu kita cukup untuk tidak mengunjungi B, sambil tidak mengubah rute yang lain. Maksudnya, bila A adalah kota yang akan dikunjungi sebelum B dan C adalah kota yang akan dikunjungi setelah kota B, kita cukup menghilangkan garis A – B dan garis B – C dan kemudian mengambil rute dari A ke C yang paling pendek, sementara rute yang lain tidak diubah. Sebagai pedoman, kita namai rute ini T(r - 1) dan kita sebut prosedur untuk menurunkan T(r - 1) dari T(r) sebagai *removal algorithm*. Selintas, kita akan beranggapan bahwa jika T(r) adalah Tour yang “baik”, yaitu, Tour yang mendekati optimal, maka kita akan berasumsi bahwa T(r - 1) jugalah “baik”. Keadaan ini sebenarnya cukup meragukan, dan kita sepatutnya mempertanyakan apakah *removal algorithm* ini benar-benar bagus. Bila kita lanjutkan, kita bisa menyatakan:

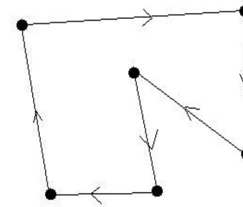
Proposisi P : Bila T(r) dalah Tour yang optimal maka T(r - 1) juga

merupakan Tour yang optimal.



P masih hanyalah sebuah proposisi, kita belumlah membuktikan bahwa pernyataan ini benar atau tidak. Bila P memang benar maka kita akan mendapat cara yang bagus untuk menyelesaikan sebuah TSP. Bila kita mempunyai T(r - 1) yang optimal maka kita akan mengetahui bahwa T(r) yang optmal dapatlah dibentuk dengan mengikutsertakan B ke dalam T(r - 1). Jika memang begitu, yang kita lakukan hanyalah memasukkan B ke tiap sisi r – 1 di dalam T(r - 1) dan pilih sisi mana yang akan memberikan rute perjalanan yang terbaik. Dengan mengaplikasikan prosedur di atas secar rekursif maka kita bisa membuat perjalanan n-kota dimulai dengan kasus trivial, kasus 2 kota. Kita sebut cara ini sebagai *insertion algorithm*.

Teori ini akan benar jika P memang benar. Tapi benarkah P memang benar? Ada kalanya di mana kita bisa memperbaiki *removal algorithm* tadi. Karena *removal algorithm* tidaklah selalu optimal, maka *insertion algorithm* juga tidaklah selalu benar. Namun, kedua algoritma masih memberikan hasil yang **nyaris** optimal dalam banyak kasus.



TSP bisa bersifat simetris ataupun tidak – dan masih banyak kriteria-kriteria lainnya yang akan bermanfaat dalam mengklasifikasikan kasus-kasus TSP. Bila ada suatu tipe TSP di mana P adalah benar maka *insertion algorithm* akan menjadi algoritma yang sangat cepat dan optimal untuk TSP yang memenuhi kriteria tersebut.

Pertanyaannya : Adakah suatu tipe TSP di mana P bernilai benar?

Pada kenyataannya *insertion algorithm* cenderung memberikan kita rute yang berbedatergantung dari urutan di mana kita menambahkan kota-kota yang bersangkutan . Oleh karena itu, pengurutan dari kota-kota tersebut merupakan ide yang cukup baik untuk dilakukan. Hasil yang keluar tidaklah benar-benar menjadi luar biasa, tapi bukan berarti tidak ada pengaruhnya.

Kalau benar begitu, adakah suatu urutan masukan di mana P akan bernilai benar? Bila tidak, seberapa besar performa dari *insertion sort* bisa ditingkatkan dengan pengurutan(*sorting*)?

Kelemahan terbesar dari *insertion algorithm* ini, layaknya pendekatan heuristik lainnya, adalah tidak ada kesimpulan pasti mengenai seberapa optimal pendekatan ini. Tapi algoritma yang bersangkutan dapat memberikan hasil yang “baik.”

Insertion algorithm didasarkan pada rekursif berikut ini, yang kita sebut sebagai Insert(r,T(r - 1)), yang berlangsung pada sub-Tour T(r - 1) pada kota (r - 1) pertama.

Insert(r,T(r - 1)) :
if (r = n) **then** selesai;

else cari r-Tour terbaik, yang bisa didapat dari meng-*insert* kota r ke Toru(r-1) dan melakukan Insert(r + 1, T)(r)

Algoritma *brach and bound insertion* memodifikasi rekursi ini untuk melakukan pencarian yang di *truncate* dari semua ruang solusi untu kmendapatkan Tour yang benar-benar optimal. Adalah proses *truncate* yang membuatnya efisien, memungkinkan kita menyingkirkan sebagian besar kemungkinan solusi tanpa termasuk solusi perjalanan yang optimal. Proses *truncate* berlangsung dengan cara mempertahankan batas paling atas untuk suatu panjang perjalanan – diambil dari panjang Tour terbaik yang didapat sampai saat ini – dan menyingkirkan sub-Tour yang melebihi batasan tersebut.

Algoritma *branch and bound insertion* didasarkan pada rekursi Search(r,T(r - 1)) yang berlangsung pada T(r - 1) tertentu, sebagaimana berikut:

Search(r,T(r - 1)) :
if (r = n) **then** simpan detail Tour dan reset batas B untuk panjang Tour
else susun r-Tour yang didapat dari meng-*insert* kota r di T(r - 1) mengurut sesuai panjang;
periksa tour-tour tersebut sesuai urutan dan untuk tiap T(r) dimana panjang tour kecil dari B, lakukan Search(r+1,T(r))

Dengan begini, tour-tour baru bisa ditemukan, di man nilai B akan berubah sepanjang pengecekan.

Strategi *bound* mengasumsikan bahwa T(r) selalu paling tidak sama panjang dengan T(r - 1). Untuk memahami maksud dari asumsi ini, kita harus mempertimbangkan akibat dari memasukkan kota j antara kota I dan k; ini akan menambahkan suatu "biaya *insertion*"(*insertion cost/IC*) yang kita simbolkan sebagai IC(I,j,k). Ini memiliki relasi yang asederhana dengan matriks jarak D :

IC(i,j,k) = D[I,j] + D[j,k] – D[I,k]

Strategi *bound* akan bisa dipakai jika semuaIC bernilai tidak negatif. Ini penting agar matriks jarak akan diperoleh dari jarak garis antara dua kota, karena D[i,k] merepresentasikan garis antara I dan k, yang paling tidak harus sama pendek dengan rute tidak langsung yang direpresentasikan oleh D[i,j] + D[j,k].

Kita bisa membayangkan proses pencarian sebagai menelusuri suatu pohon yang terdiri dari beberapa *node*(sub-Tour) dan *brach*(*insertion*). Terdapat sampai (r – 1) buah *insertion*, tergantung pengaruh dari *bound*, mengikuti tiap T(r - 1), masing-masing mengarah kepada *node* r-Tour dari mana bisa muncul sampai r buah *branch*. Apa yang dilakukan *bound* adalah mengacuhkan sebagian *branch*, dan ini akan memperkecil ukuran *tree* yang harus diperiksa.

Kita bisa mempertajam kriteria untuk *bound* jika kita siap untuk menerima solusi yang panjangnya berada pada beberapa persen P dari optimum yang sebenarnya. Selanjutnya kita bisa *truncate* pencarian dari tiap *branch* yang mana panjang perjalanannya melebihi (1 – p/100)*B. Ini akan mengurangi waktu pencarian solusi, tapi solusi yang didapat belum tentu optimal.

Variasi yang bisa dipertimbangkan untuk diikuti antara lian adalah mengurutkan urutan di mana kota di-*insert* sebagai bagian dari memperbaiki hasil dari *insertion algorithm*. Idenya adalah untuk membuat sub-Tour sepanjang mungkin selagi n-Tour dibangun, dengan gagasan meningkatkan jumlah *node* pohon yang melanggar batas *bound*, sehingga jumlah pohon yang harus diperiksa berkurang. Algoritma pengurutan yang telah dicoba termasuk:

- Melakukan *pass* pada *insertion algorithm*, pada tiap pemilihan tahap, kota yang selanjutnya di-*insert* adalah yang *insertion cost* minimumnya paling besar
- Sama seperti di atas, tapi kota yang dipilih adalah yang *insertion cost* **kedua terkecil**nya paling besar
- Pilih untuk tiga kota pertama, kota-kota yang membentuk segitiga terpanjang dari tiap tiga kota, dan abaikan kota-kota linnya.

Semua modifikasi-modifikasi ini memberikan perbaikan signifikan, tapi tidak ada yang lebih baik dari yang lainnya.

- **Pendekatan heuristic dengan *TSP Improvement Algorithms***

Perbaikan progresif adalah strategi yang umum dalam mencari solusi minimal (atau maximal) untuk suatu permasalahan *continuous* kompleks n-dimensi. Pada pengertian umum, kita mulai dari solusi **manapun**, diwakilkan pada titik di ruang-n yang akan kita sebut sebagai S1. Selanjutnya, kita mencari arah di dalam ruang-n, mulai dari S1, yang akan membuat lebih baik fungsi yang diminimalisir. Selanjutnya kita ikuti arah ini sejauh kita bisa – yaitu, sampai fungsinya berhenti mengecil atau kita mulai memasuki daerah di ruang-n yang tidak lagi mengandung solusi yang valid.

Ini akan memberikan titik solusi kedua, S2, yang lebih baik daripada S1.Selanjutnya, kita ulangi lagi proses dari S2 dan terus menerus diulang sampai tidak ada lagi peningkatan yang bisa dilakukan. Bila kita beruntung, kita telah mendapatkan solusi minimum global dan tugas telah terselesaikan. Namun, ada kalanya di mana yang kida dapatkan adalah solusi minimum local yang lebih baik dari solusi sekitarnya tapi tidak lebih baik dari solusi yang di dapat dari titk lain. Pada banyak kasus, kita bisa membuktikan bahwa suatu masalah tidak mempunyai minimum local, yang mana pada kasus ini strategi dasar cukup untuk menyelesaikannya.

Jika tidak, kita harus merancang suatu prosedur yang memungkinkan kita mundur dari solusi minimum local dan melihat ruang solusi pada skala lebih besar dengan suatu cara, dan dengan begitu menyimpulkan pada arah mana solusi minimum global – atau paling tidak, solusi minimum local yang lebih baik – mungkin berada.

Apakah pendekatan progresif ini berguna dalam menyelesaikan TSP? Mereka yang telah menggunakannya akan menghargai kenyataan bahwa ini merupakan persoalan yang diskrit dan bukan kontinyu, dan bahwa penyelesaian yang tidak optimal bisa sangat berbeda dari jalur perjalanan yang lebih pendek. Karena itu strategi peningkatan manapun harus bisa 'melihat' berbagai solusi yang dekat dengan solusi sekarang.

Ada beberapa cara untuk menemukan solusi optimum untuk sub-ruang tertentu dari TSP secara keseluruhan. Perbaikan cara ini bekerja sebagai berikut; mulailah dari Toour lama yang manapun, pilih sub-ruang yang mana di dalamnya terkandung Tour ini, dan mulai optimasi dari sub-ruang tersebut. Pada kemungkinan terburuk, kita akan kembali pada solusi yang akan kita pilih untuk memulai, yang mana pada kasus ini kita akan mencoba sub-ruang lain yang mengadung Tour yang sekarang. Pada kemungkinan terbaik kita akan menemukan Tour yang terbaik, dan kita bisa mengoptimasikan dari Tour tersebut. Kita akan terus mencari peningkatan sampai proses menggapai Tour yang mungkin optimum global atau yang optimum lokal sesuai algoritma.

Dynamic programming adalah salah satu cara yang simpel tapi efektif digunakan untuk menyelesaikan berbagai macam persoalan optimasi yang melingkupi membuat keputusan pada tiap bagian langkah.

Suatu langkah yang umum adalah dengan melibatkan *insertion* kota r ke dalam Tour yang terdiri dari kota 1 ke (r - 1). Pada kasus ini, kita hanya memperbolehkan kota r di-*insert* ke sebelah kota (r – 1). Ini hanya menyisakan 2 kemungkinan untuk meng-*insert* kota r – dia bisa di-*insert* sebelum kota r – 1 atau setelahnya. Dengan memberikan batasan-batasan ini, kita bisa dengan cepat mencari ruteterpendek yang bisa didapatkan.

Berapa rute yang sesuai dengan batasan ini? Pada TSP yang asimetris titik mulainya adalah rute yang terdiri dari kota 1 dan kota 2, dan ada 2^(n - 2) buah jumlah rute yang berbeda yang bisa dipilih dengan cara ini. Untuk TSP yang simetris kita bisa mulai dari rute untuk 3 buah kota – sehingga ada 3*2^(n - 3) buah rute yang bisa dipilih. Ada jumlah yang besar untuk nilai n yang besar, tapi gantikan hanya subset yang kecil dari ruang penyelesaian total yang mengandung Tour sebanyak (n - 1)! Pada kasus asimetris atau setengahnya untuk kasus simetris. Kita sebut subset ini sebagai *search subset*.

Algoritmanya bekerja melalui serangkaian langkah selagi kita mempertimbangkan *insertion* untuk tiap kota. Sebelum mepertimbangkan *insertion* untuk kota r kita harus sudah membangkitkan suatu daftar – sebut sebagai list(r - 1) – dari panjang Tour sesuai dengan Tour dari 1 ke r – 1:

- Panjang dari rute terpendek di mana i mendahului r – 1 – dan karenanya r – 2

datang setelah r – 1 – untuk i = 1 sampai r – 2

- Panjang dari rute terpendek di mana i dating setelah r – 1 – dan karenanya r – 2 mendahului r – 1 – untuk i = 1 sampai r – 2

Selagi kita mempertimbangkan kemungkinan untuk meng-*insert* kota r kita akan membangkitkan list dari r. Kita akan mengulangu prosedur ini terus-menerus sampai kita telah membangiktkan list-n, di mana pada tahap ini kita akan mengetahui panjang dari rute terpendek untuk n buah kota pada *search subset*. Selanjutnya kita memerlukan suatu prosedur *backtracking* untuk menghasilkan rute yang terpendek.

Perlu kita perhatikan bahwa terdapat 2(r - 1) masukan pada list-r, bila kita jumlahkan ini pada r = 3 sampai n kita akan melihat bahwa terdapat (n - 2)(n - 3) masukan list secara keseluruhan. Berhubung tiap masukan memerlukan jumlah komputasi yang sama tak peduli ukuran permasalahan, waktu penyelesaian bervariasi antara n^2 untuk n yang besar.

Pada kasus TSP yang simetris jumlah dari masukan menjadi setengahnya karena kita hanya memerlukan satu buah masukan – “kota i bersebelahan dengan kota r - 1” – untuk tiap i. Pada kedua kasus, karenanya, algoritma akan menemukan rute terbaik untuk *search subset* pada waktu yang berkisar di n^2. Kita bisa sebut ini sebagai ***dynamic programming subset algorithm*** *(DPSA).*

Untuk melihat bagaimana DPSA bekerja mari kita lihat pada contoh 6 kota yang mana matriks jaraknya ditampilkan berikut.

	1	2	3	4	5	6
1		44	35	18	28	23
2	44		38	28	27	42
3	35	38		26	14	14
4	18	28	26		14	20
5	28	27	14	14		15
6	23	42	14	20	15	

Titik awal kita adalah 3 Tour terdiri atas kota 1 ke 3. Tour yang ada hanyalah 1 - 2 – 3 atau kebalikannya 3 -2 – 1; keduanya memiliki panjang D[1,2] + D[2,3] + D[3,1] = 44 + 38 + 35 = 117.

Selanjutnya kita bangun 4-list yaitu list dari panjang dari rute terpendek di mana kota 4 bersebelahan dengan kota i untuk i = 1, 2 dan 3. Pada kasus TSP yang simetris kita bisa memperlakukan kota 4 sebagai kasus khusus yang tidak harus bersebelahan dengan kota 3, memperbesah *seach subset*.

Maka ada 3 *insertion* yang mungkin, dengan *insertion cost*:

IC(1,4,2) = D[1,4] + D[4,2] - D[1,2] = 18+28-44 = 2

IC(2,4,3) = D[2,4] + D[4,3] - D[2,3] = 28+26-38 = 16

IC(3,4,1) = D[3,4] + D[4,1] - D[3,1] = 26+18-35 = 9

	4
1	119
2	119
3	126
4	
5	

Sebuah rute dibentuk dengan meng-*insert* kota 4 ke sebelah kota 1 akan melibatkan IC sebesar 2 jika itu adalah antara 1 dan 2 atau 9 jika itu antara kota 3 dan kota 1. Karenanya rute terdekat di mana kota 4 bersebelahan dengan kota 1 adalah 1 - 4 - 2 – 3 yang mana memiliki panjang 117 + 2 = 119. Ini memberikan masukan pertama ke list. Argumen yang sama akan memberikan panjang 119 dan 126 untuk dua masukan lainnya.

Selanjutnya kita pertimbangkan *insertion* untuk 5 kota. Menurut prosedur tiap rute di dalam *search subset* di mana kota 1 bersebelahan dengan kota 5 harus dibentuk dengan meng-*insert* kota 5 antra kota 1 dan kota 4. IC(1,5,4) = 24.

	4	5
1	119→	143
2	119→	132
3	126→	128
4		128
5		

Jelas bahwa rute terpendek di mana kota 1 bersebelahan dengan kota 5 harus dibentuk dengan memasukkan kota 5 ke Tour 4-kota

terpendek di mana kota 4 bersebelahan dengan kota 1. Panjang dari Tour 4-kota tadi adalah 119. Karenanya masukan pertama dari 5-list adalah 119 + 24 = 143. Argumen yang sama menghasilkan nilai 132 dan 128 untuk masukan selanjutnya di dalam list.

Manakah yang merupakan 5-Tour terpendek di mana kota 5 bersebelahan dengan kota 4? Karena menurut aturan subset kota 5 selalu bersebelahan dengan kota 4, ini pastilah rute terpendek yang didapat dari masukan sebelumnya – dalam hal ini, 128. Kita harus mengingat yang mana yang terpendek, maka kita mencatatnya. Sebagai gambaran kita lakukan ini dengan meletakkan kota ke dalam masukan; pada program komputer kita akan mencatat index dari posisinya.

	4	5	6
1	119	143	153
2	119	132	162
3	126	128	143
4		128	149
5			143

Untuk 6-list(dan 7-list, 8-list…dan lain-lainnya untuk persoalan yang lebih besar) dibangkitkan dengan cara yang sama dengan 5-list.

Setelah kita membangkitkan n-list maka kita tahu rute terpendek di dalam *search subset* – diperoleh dari masukan terakhir di dalam n-list – 143 contohnya. Sekarang kita harus membangkitkan rute itu sendiri. Ini kita lakukan dengan melakukan *backtracking* sepanjang list.

Dari lokasi masukan yang dikotakkan pada 6-list kita lihat bahwa rute yang terpendek dibentuk dengan menyisipkan kota 6 antara kota 3 dan 5. Untuk itu kota 5 harus disisipkan antara kota 4 dan 3 yang, sebagai gantinya, mengharuskan kota 4 disisipkan di sebelah kota 3. Dengan membandingkan IC(2,4,3) dan IC(3,4,1) kita lihat bahwakota 4 harus disisipkan antara kota 3 dan 1 dan bukan kota 2 dan kota 3.

Mengaplikasikan *insertion* ini selanjutnya memungkinkan kita menbangun Tour yang optimal:

1-2-3
1-2-3-4
1-2-3-5-4
1-2-3-6-5-4

Sebagai pembuktian, panjang dari Tour ini adalah:

D[1,2]+D[2,3]+D[3,6]+D[6,5]+D[5,4]+D[4,1] = 44+38+14+15+14+18 = **143**

Maka cara ini terbukti.

Perhatikan bahwa ini adalah peningkatan yang pasti dari Tour di mana kita mulai, dengan 1-2-3-4-5-6 yang memiliki panjang 160.

DPSA bisa digunakan sebagai algoritma peningkatan sebagai berikut:

1. Pilih sebuah Tour secara acak menggunakan fungsi untuk membangkitkan nilai acak dan jadikan itu sebagai Tour saat ini
2. Atur urutan *insertion* sebagai urutan kota yang dikunjungi pada Tour yang diikuti dimulai dari kota yang pertama
3. Selesaikan dengan DPSA
4. Jika ini memberikan Tour yang lebih pendek maka jadikan ini Tour yang baru dan kembali ke tahap 3, jika tidak lanjut ke tahap 5
5. Atur urutan *insertion* sebagai urutan kota yang dikunjungi pada Tour yang diikuti dimulai dari kotaselanjutnya dari Tour yang sekarang, dan kembali ke tahap 3
6. Jika tidak ada peningkatan yang bisa diraih menggunakan urutan *insertion* Tour dengan semua titik awal n maka catat Tour yang sekarang sebagai kemungkinan yang optimum
7. Ulangi tahap 1 sampai 5 untuk jangka waktu tertentu

Jelas sekali sebuah DPSA dengan *insertion* di dalam urutan Tour melingkupi Tour itu sendiri.

Dengan menggunakan kota yang berbeda sebagai titik awal menghasilkan sub-ruang yang berbeda yang juga termasuk Tour yang sekarang; variasi ini telah terbukti efektif pada prakteknya. Beberapa variasi lain telah diujikan tanpa penigkatan yang signifikan, diantaranya:

- Membalikkan urutan Tour

- Memperkenalkan gangguan urutan yang acak yang tetap memakai Tour yang sekarang sebagai subset

4. Perangkat Lunak yang ada
   - GATSS, oleh Thomas Pederson, suatu penyelesaian berbasis *Genetic Algorithm* untuk *Traveling Salesman problem* ditulis dalam GNU C++.
   - GAlib, A C++ Genetic Algorithms Library, lihat khususnya contoh yang disajikan, yaitu TSP.
   - Maugis TSP solver, suatu program yang dikembangkan dengan ansi-C, untuk Symmetric Euclidean TSPs, oleh Lionnel Maugis.
   - tsp_solve, sekumpulan heuristik and algoritma optimal untuk TSP, oleh Chad Hurwitz. Dikembangkan dengan *software* GNU C sehingga bisa dijalankan dalam hamper semua *unix*.
   - An ATSP code, oleh Glenn Dicus, Bart Jaworski dan Joseph Ou-Yang.
   - A TSP program in Parlog, oleh Steve Gregory.
   - The Travelling Spider Problem, oleh Moshe Sniedovich.
   - Traveling Salesman Problem solving program (TSPSolver), oleh Victor V. Miagkikh. TSPSolver dikembangkan dalam Borland C++ dan Borland Delphi untuk MS Windows.
   - GRIN, sebuah *software package* untuk graf oleh Vitali Petchenkine.
   - Traveling Salesman Java Applet, oleh Martin Hagerup. Pengarangnya berpendapat bahwa appletnya telah dipilih untuk ditampilkan dalam buku `Java Programming for Dummies'' Membutuhkan Netscape 2.0 di 32 bits atau *Java-browser* lainnya.
   - Simple Closed Paths, dari buku: *Introduction to Programming with Mathematica, 2nd Edition* , 1995, *TELOS*/Springer-Verlag.
   - A Java TSP demo program nenggunakan formulasi network neural Kohonen.
   - A Simple TSP-Solver: An ABACUS Tutorial, oleh Stefan Thienel, 1996.
   - A Guided Local Search demo for TSP, oleh Christos Voudouris, (membutuhkan Microsoft Windows 3.1.).
   - Another Guided Local search demo, membutuhkan SunOS (sudah ada untuk SunOS 5.3) dan XView, oleh Christos Voudouris. Versi baru untuk Windows XP/NT telah tersedia.
   - A heuristic and a brute force method, Java programs oleh Aaron Passey.
   - BOB: Branch-and-bound Optimization liBrary, suatu pustaka algoritma Branch-and-Bound parallel multi fungsidikembangkan di PRiSM Laboratory of University of Versailles-Saint Quentin en Yvelines. Tersedia contoh untuk QAP, TSP Dan VCP.
   - Concorde, oleh David Applegate, Robert Bixby, William Cook, dan V. Chvatal.
   - David Neto's Lin-Kernighan ``cluster aware'' heuristic, oleh David Neto, sebuah program yang dibuat menggunakan CWEB toolset.
   - Traveller, sebuah kode java yang dikembangkan oleh Kent Paul Dolan.

5. Kesimpulan

Sampai saat ini belum ditemukan algoritma yang benar – benar efisien dalam menemukan solusi dari *Travelling Saleman Problem*. Pendekatan heuristic yang dicobakan sekarang inipun belum ada yang mampu untuk menyelesaikannya dengan benar-benar cepat tanpa memakan waktu secara factorial. Walaupun begitu, pendekatan-pendekatan yang ada dapat menjadi referensi tentang cara-cara apa yang pernah dicoba dan mungkin bisa menjadi inspirasi bagi ditemukannya algoritma yang diharapkan. Sampai algoritma itu ditemukan, maka pendekatan-pendekatan yang ada dapat dipakai

sebagai solusi untuk kasus-kasus khusu sesuai kebituhan dalam tingkat sempit.

6. Daftar Pustaka

1. Munir, Rinaldi. 2004. Diktat Kuliah IF2151 Matematika Diskrit Edisi Keempat. Departemen Teknik Informatika : Institut Kuliah Bandung.
2. TSPBIB. 2006. TSPBIB Home Page. *http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html*. Tanggal Akses : 1 Januari 2007.
3. Wikipedia. 2006. Heuristic - Wikipedia, the Free Encyclopedia. *http://en.wikipedia.org/wiki/Heuristic.*Tanggal Akses : 1 Januari 2007.
4. Wikipedia. 2006. Travelling_salesman_problem - Wikipedia, the Free Encyclopedia. *http://en.wikipedia.org/wiki/Travelling_salesman_problem.*Tanggal Akses : 1 Januari 2007.
5. Dakin, Robert. 2002. The Travelling_salesman_problem. *http://members.pcug.org.au/~dakin/tsp.htm.* Tanggal Akses : 1 Januari 2007.