Penerapan Teori Graf pada Google Map

Akbar Juang Saputra (13511026)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13511026@std.stei.itb.ac.id

***Abstract*—Google Map merupakan peta digital yang dapat mencari jalur terpendek dari tempat berada sampai tujuan awal. Sistem dalam pencarian *shortest path* ini memakai algoritma A*. Algoritma A* memakai algoritma *pathfinding* Dijkstra dan informasi tambahan berupa fungsi heuristik.**

***Index Terms*—Google Map, Algoritma A*, fungsi heuristik.**

I. PENDAHULUAN

Dalam kaitannya, dengan era globalisasi perkembangan teknologi informasi(TI) semakin berkembang pesat.Hampir seluruh orang di dunia,menggunakan teknologi informasi tersebut,untuk mempermudah dan memberikan kenyamanan untuk melakukan segala aktivitas dalam kehidupan sehari-hari.

Seperti contohnya : Handphone sebagai alat komunikasi,internet di gunakan sebagai alat untuk mencari informasi-informasi penting,dan sebagainya. Kemajuan teknologi informasi tidak bisa di hindarkan dari kehidupan manusia karena kemajuan teknologi informasi berjalan seiring dengan kemajuan ilmu pengetahuan. Akan tetapi,perkembangan teknologi informasi ini,juga membawa dampak bagi manusia.

Dalam pencarian tempat pun sekarang sudah memakai peta elektronik yang bernama Google Map. Peta digital yang terdapat pada perangkat smartphone maupun tablet ini lebih mendapatkan sorotan karena sangat berfungsi ketika kita dalam perjalanan. Bukankah Google Map akan sangat berguna jika kita ingin mengatahui letak suatu daerah/tempat ataupun jalan menuju tempat tersebut ketika kita sedang dalam perjalanan? Bahkan aplikasi Maps ini juga dapat memperhitungkan waktu yang kita perlukan untuk mencapai lokasi tersebut.

Di sini kita akan membahas cara kerja Google Map dalam mencari jalur terpendek dari tempat awal ke tempat tujuan.

II. LANDASAN TEORI

2.1 Graf

Graf digunakan untuk merepresentasikan objek-objek diskrit dan hubungan antara objek-objek tersebut. Representasi visual dari graf adalah dengan menyatakan objek sebagai noktah, bulatan, atau titik, sedangkan hubungan antara objek dinyatakan dengan garis. Secara matematis, graf didefinisikan sebagai berikut :

DEFINISI 1.1 Graf *G* didefinisikan sebagai pasangan himpunan (V,E), yang dalam hal ini : *V* = himpunan tidak-kosong dari simpul-simpul(*vertices* atau *node*) = { v1, v2, v3, ... } dan *E* = himpunan sisi (*edges* atau *arcs*) yang menghubungkan sepasang simpul = { e1, e2, e3, ... } atau dapat ditulis singkat notasi *G* = (*V*, *E*).

Definisi 1.1 menyatakan bahwa *V* tidak boleh kosong, sedangkan *E* boleh kosong. Jadi, sebuah graf dimungkinkan tidak mempunyai sisi satu buah pun, tetapi simpulnya harus ada, minimal satu. Simpul pada graf dapat dinomori dengan huruf, seperti *a, b, c, ..., v, w, ...*, dengan bilangan asli 1, 2, 3, ..., atau gabungan keduanya. Sedangkan sisi yang menghubungkan simpul *vi* dengan simpul *vj* dinyatakan dengan pasangan (v*i*, v*j*) atau dengan lambang, seperti *e1, e2, ...* Dengan kata lain, jika *e* adalah sisi yang menghubungkan simpul *vi* dengan simpul *vj*, maka *e* dapat ditulis sebagai *e* = (*vi, vj*).

Graf dapat dikelompokkan menjadi beberapa kategori (jenis) bergantung pada sudut pandang pengelompokannya. Pengelompokan graf dapat dipandang berdasarkan ada tidaknya sisi ganda atau sisi kalang, berdasarkan jumlah simpul, atau berdasarkan orientasi arah pada sisi.

Berdasarkan orientasi arah pada sisi, maka secara umum graf dibedakan menjadi 2 jenis :

1. **Graf tak-berarah** (*undirected graph*)

Graf yang sisinya tidak mempunyai orientasi arah disebut graf tak-berarah. Pada graf tak-berarah, urutan pasangan simpul yang dihubungkan oleh
sisi tidak diperhatikan. Jadi, (vj, vk) = (vk, vj) adalah sisi yang sama.

2. **Graf berarah** (*directed graph* atau *digraph*)

Graf yang setiap sisinya diberikan orientasi arah
disebut sebagai graf berarah. Kita lebih suka menyebut sisi berarah dengan sebutan busur (*arc*).
Pada graf berarah, (vj, vk) _ (vk, vj). Untuk busur (vj, vk), simpul vj dinamakan **simpul asal** (*initial vertex*) dan simpul v*k* dinamakan **simpul terminal** (*terminal vertex*).

2.2 Algoritma A*

A* merupakan bentuk yang paling dari *Best First*

Search (BFS). A* mengevaluasi node dengan menggabungkan *g(n)*, yaitu *cost* untuk mencapai *node*, dan *h(n)*, yaitu *cost* yang diperlukan dari node untuk mencapai tujuan, sehingga:

$$f(n) = g(n) + h(n)$$

g(n) merupakan *cost* suatu *path* dari *node* awal ke *node n*, dan *h(n)* adalah perkiraan *cost* terendah dari *node n* ke tujuan. *f(n)* adalah perkiraan solusi dengan *cost* termurah melalui *n.*

Dengan demikian, untuk menemukan solusi termurah, hal yang pertama yang dicoba adalah node dengan nilai *g(n)+h(n)* terendah. Strategi ini jelas lebih baik dengan disediakannya fungsi heuristik *h(n)* yang dapat memenuhi kondisi tertentu sehingga A* menjadi optimal.

Perlu disadari bahwa algoritma Dijsktra adalah subset dari algoritma A*. Dalam A*, akan dihitung perkiraan *total cost* dari *node* dengan menambahkan nilai heuristik pada *cost* sejauh ini. A* kemudian memilih sebuah *node* untuk diproses berdasarkan nilai tersebut.

Jika heuristik selalu menghasilkan 0, maka perkiraan *total cost* selalu akan sama dengan *cost* sejauh ini. Saat A* memilih node dengan perkiraan *total cost* terkecil, *node* dengan *cost* sejauh ini terkecil dipilih. Hal ini sangat mirip dengan Dijkstra. A* dengan heuristik adalah versi *pathfinding* dari Dijkstra.

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated.
    openset := {start}    // The set of tentative nodes to be evaluated, initially containing the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

    while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score <= g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure

function reconstruct_path(came_from, current_node)
    if came_from[current_node] in set
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node
```

Gambar 1 Pseudocode Algoritma A*

2.2.1 Sejarah Algoritma A*

Algoritma A* yang menggabungkan cost path saat ini dengan heuristic search, dikembangkan oleh Hart, Nilsson, dan Raphael, dengan beberapa koreksi lanjutan. Dechter dan Pearl mendemonstrasikan efisiensi secar optimal dari A*.

Tulisan mengenai A* yang asli memperkenalkan kondisi konsistensi pada fungsi heuristik. Kondisi onoton diperkenalkan oleh Pohl sbagai pengganti yang lebih sederhana, namun Pearl menunjukkan bahwa keduanya ekuivalen.

Pohl mempelopori pembelajaran tentang hubungan antara galat (*error*) dalam fungsi heuristik dengan kompleksitas waktu A*. Hasil dasar yang diperoleh untuk *tree search* dengan *unit steps costs* dan node tujuan jamak. *"Effective branching factor"* diusulkan oleh Nilsson sebagai pengukuran empiris dan efisiensi; ekuivalen dengan mengasumsikan time cost dari *O((b)^d)*. Karena tree search diaplikasikan pada graf, Kort *et al.* menentang bahwa tie cost lebih baik dimodelkan sebagai O(b^(d-k)), dimana k tergantung pada akurasi heuristik; sehingga analisis ini menghasilkan beberapa kontroversi. Untuk graph search, Helmert dan Roger mencatat beberapa permasalahan yang terkenal yang mengandung banyak node secara eksponen dalam path solution optimal, menyiratkan kompleksitas waktu eksponen untuk A* bahkan dengan galat mutlak konstan pada *h.*

2.2.2 Langkah-Langkah Algoritma A*

Algoritma ini bekerja dengan cara yanag mirip dengan Dijkstra. Daripada mempertimbangkan open node dengan nilai cost sejauh ini terendah, node yang paling mungkin untuk menuju path terpendek secara keseluruhan dipilih yang dikendalikan oleh heuristik. Jika heuristik akurat, maka algorita tersebut akan efisien. Jika heuristik tidak tepat, maka algoritma akan bekerja lebih buruk dari Dijkstra. Berikut merupakan langkah-langkah algoritma A*

a. Meproses *Current Node*

Selama iterasi, A* mempertimbangkan setiap koneksi yang keluar dari *current node*. Untuk setiap koneksi, A* menemukan *end node* dan menyimpan total cost dari path sejauh ini dan koneksi yang sampai ke sana, sam seperti sebelumnya.

Sebagai tambahan, A* menyimpan satu nilai lagi: perkiraan dari cost total untuk sebuah path dari start node menuju current node dan menuju tujuan (selanjutnya disebut perkiraan total cost). Perkiraan ini merupakan jmlah dari dua nilai: cost sejauhini dan seberapa jauh dari node tersebut hingga tujuan. perkiraan ini dihasilkan dari potongan kode yang terpisah dan bukan bagian dari algoritma tersebut.

Perkiraan-perkiraan ini disebut "nilai heuristik" dari suatu node, dan nilainya tidak dapat negatif (karena cost dalam graf tidak negatif, dan tidak masuk akal jika perkiraan negatif). Penggenerasian dari nilai heuristik adalah perhatian utama dalam mengimplementasikan algoritma A*.

b. *Node List*

Seperti sebelumnya, algoritma ini menyimpan daftar dari open node yang telah dikunjungi namun belum diproses dan closed node yang telah diproses. Node dipindahkan ke open list saat mereka ditemukan pada ujung koneksi. Node dipindahkan ke closed list setelah diproses dalam iterasi masing-masing.

Tidak seperti sebelumnya, node dari open list dengan perkiraan total cost terkecil dipilih pada setiap iterasi, yang hampir selalu berbeda dengan cost terkecil sejauh ini.

Perubahan ini mengijinkan algoritma tersebut memerikasa node yang lebih menjanjikan terlebih dahulu. Jika node memilik perkiraan total cost yang kecil, maka node tersebut pasti memiliki cost sejauh ini yang relatif pula. Jika perkiraanny akurat, maka nodes yang lebih dekat ke tujuan akan dipertimbangkan terlebih dahulu, mempersempit pencarian ke area yang yang paling menguntungkan.

c. Menghitung Jarak Sejauh Ini untuk *Open* dan *Closed Nodes*

Seperti sebelumnya, setelah sampai pada open dan closed node selama iterasi, dan nilai yang terekam harus direvisi. Nilai cost sejauh ini terhitung wajar, dan jika nilai baru lebih rendah dari nilai untuk node yang sudah ada, maka nilainya perlu di-*update*. Dilakukan perbandingan ini secara ketat berdasarkan nilai cosr sejauh ini (hanya nilai yang dapat diandalkan, karena tidak mengandung nilai perkiraan apapun), bukan perkiraan total cost.

Tidak seperti Dijkstra, algoritma A* dapat menemukan rute yang lebih baik menuju node yang sudah ada pada closed list. Jika nilai sebelumnya sangat optimis, maka node mungkin telah diproses untuk berpikir bahwa itu adalah pilihan terbaik, yang pada kenyataannya bukan.

Hal ini mengakibatkan *knock-on problem*. Jika node yang meragukan telah diprose dan dimasukkan dalam closed list, maka berarti semua koneksinya telah dipertimbangkan. Mungkin saja seluruh himpunan node memiliki cost sejauh ini berdasarkan node sejauh ini dari node yang diragukan. Meng-update nilai dari node yang meragukan tidaklah cukup. Seluruh koneksinya juga harus diperiksa kembali untuk mendistribusikan nilai baru tersebut.

Dalam kasus perevisian node dalam open list, hal ini tidak diperukan, karena diketahui bahwa koneki dari sebuah node dalam open list belum diproses. Namun ada cara sederhana untuk memaksa algoritma untuk menghitung kembali dan menyebarkan nilai baru tersebut. Node dari closed list dapat dipindahkan kembali ke open list, yang kemudian akan menunggu sampai tertutup dan koneksinya telah dipertimbangkan kembali. Node yang bergantung pada nilainya kemudian akan diproses sekali lagi.

Jadi, closed node nilainya telah direvisi dikeluarkan dari closed list dan ditempatkan ke open list. Open nodes yang memiliki nilai terevisi tetap pada open list, seperti sebelumnya.

d. Mengakhiri Algoritma

Dalam banyak implementasi, A* berakhir saat node tujuan adalah node terkecil pada open list. Namun node yang memiliki nilai perkiraan total cost terkecil (dan oleh karena itu akan diproses pada iterasi berikutnya dan dimasukkan dalam closed list) mungkin saja nilainya harus direvisi nanti. Tidak ada lagi jaminan bahwa hanya karena node tersebut paling kecil di open list, maka shortest path akan didapatkan di sana. Maka mengakhiri A* saat node tujuan merupakan terkecil di open list tidak akan menjamin bahwa rute terpendek telah ditemukan.

Gambar 2 Contoh Algoritma A* bekerja

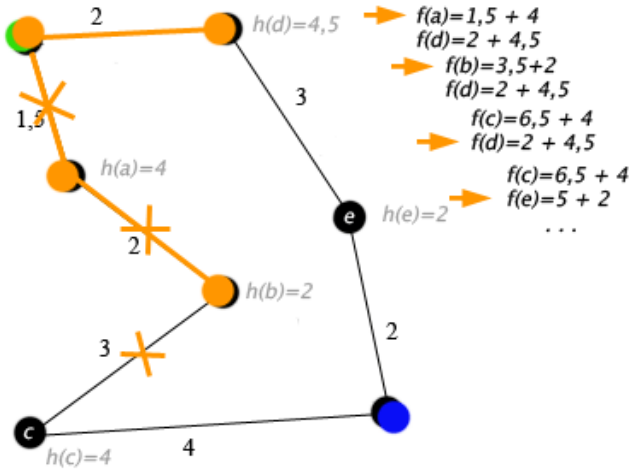
Oleh karena itu, wajar untuk bertanya apakah A* dapat dijalankan sedikit lebih lama untk menghasilkan hasil yang terjamin optimal. Hal ini dapat dilakukan dengan memerintahkan algoritma tersebut untuk berakhir hanya jika node di open list dengan cost sejauh ini (bukan perkiraan total cost) terkecil memiliki nilai cost sejauh ini lebih besar dari cost path yang ditemui menuju tujuan. Dengan cara itu, dan hanya dengan cara itu dapat terjamin bahwa tidak ada path masa depan akan ditemukan yang membentuk shortcut.

Ini merupakan kondisi pengakhiran efektif yang dapat dilihat pada Dijkstra, dan dapat ditunjukkan bahwa menerapkan kondisi akan menghasilkan jumlah fill yang sama layaknya menggunakan algoritma *pathfinding* Dijkstra. Node mungkin saja dicari dengan urutan yang berbeda, dan mungkin saja ada sedikit perbedaan dalam himpunan node di open list, namun tingkat fill rata-rata sama. Dengan kata lain, Dijkstra mengambil seluruh kelebihan performa dari A* dan membuatnya tidak bernilai.

Implementasi A* seluruhnya bergantung pada kenyataan bahwa A* dapat menghasilkan hasil yang tidak optimal secara teoritis. Untungnya, hal ini dapat dikendalikan dengan fungsi heuristik. Tergantung pada pemilihan fungsi heuristik, hasil optimal akan terjamin atau hasil sub-optimal yang memberikan eksekusi yang lebih cepat dapat dengan sengaja diijinkan.

Karena A* sering berhubungan dengan hasil sub-optimal, implementasi A* dalam jumlah besar malah berakhir saat node tujuan dikunjungi pertama anpa menunggunya untuk menjadi terkecil di open list. Kelebihan performa A* tidak sebesar melakukan hal yang sama pada Dijkstra, namun banyak pengembang merasa setiap bit berharga, terutama saat algoritma tidak dibutuhkan untuk menjadi optimal.

e. Mengambil Path



Path terahir didapat dengan cara yang benar-benar sama seperti sebelumnya: dengan memulai dari tujuan dan mengakumulasi koneksi saat bergerak mundur ke start node. Koneksi-koneksi tersebut kemudian dibalikkan kembali untuk membentuk path yang benar.

2.2.3 Memilih Fungsi Heuristik

Semakin akurat heuristik yag digunakan, semakin sedikit fill yang akan dialami oleh A*, dan semakin cepat dijalankan. Jika heuristik yang sempurna digunakan (yang selalu mengembalikan jarak path minimi=um antaa 2 node yang sangat tepat), A* akan langsung mengarah pada jawaban yang tepat: algoritma akan menjadi O(p), dimana p adalah jumlah langkah dalam path.

Sayangnya, untuk menemukan jarak yang tepat antara dua nodes, biasanya rute terpendek di antara kedua node tersebut harus ditemukan, yang mungkin berarti menyelesaikan pemasalahan pathfinding.

Untuk heuristik yang tidak sempuna, A* berlaku sedikit berbeda tergantung pada apakah heuristik tersebut terlalu rendah atu terlalu tinggi, berikut adalah penjelasannya:

a. Heuristik yang Terlalu Rendah

Jika heuristik terlalu rendah, maka A* memperkirakan terlalu rendah panjang path aktual, A* membutuhkan waktu lebih lama untuk dijalankan. Perkiraan total cost akan codong ke arah cost sejauh ini (karena nilai heuristik terlalu kecil dari kenyataan). Jadi A* lebih memilih untuk memeriksa node yang lebih dekat ke start node, daripada yang lebih dekat ke tujuan. ini akan meningkatkan waktu yang diperlukan untuk menemukan rute menuju tujuan.

Jika heuristik terlalu rendah, maka hasil yang dihasilkan akan menjadi path terbai yang memungkinkan, dan akan menjadi path yang benar-benar sama dengan yang dihasilkan oleh Dijkstra.

b. Heuristik yang terlalu Tinggi

Jika heuristik terlalu tinggi, maka diperkirakan panjang path aktual terlalu panjang. A* mungkin tidak menghasilkan path terbaik. A* akan cenderung menghasilkan path dengan nodes yang lebih sedikit, bahkan jika koneksi di antara nodes terlalu tinggi.

Nilai perkiraan total cost akan membias kepada heuristik. Algoritma A* akan lebih tidak memperhatikan secara proporsional cost sejauh ini dan akan cenderung lebih menyukai nodes yang memiliki sisa jarak yang lebih kecil. Hal ini akan menggeser fokus pencarian ke tujuan lebih cepat, namun dengan resiko kehilangan rute terbaik.

Ini berarti bahwa panjang total dari path mungkin lebih besar dari panjang total dari path terbaik. Untungnya, tidak berarti bahwa akan langsung didapat path yang buruk. Dapt ditunjukkan bahwa jika heuristik memandang terlalu tinggi oleh sebagian besar x (yakni x adalah pandangan terlalu tinggi terbesar untuk setiap node dalam graf) dan final path tidak akan lebih besar dari x.

2.3 Google Map

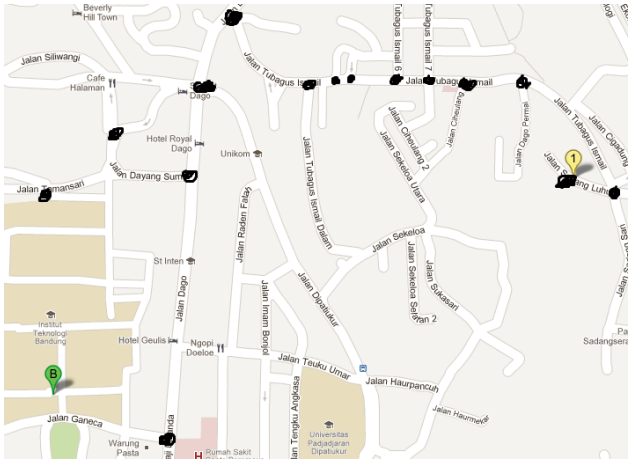
Google Maps adalah sebuah jasa peta globe virtual gratis dan online disediakan oleh Google dapat ditemukan di http://maps.google.com. Ia menawarkan peta yang dapat diseret dan gambar satelit untuk seluruh dunia dan baru-baru ini, Bulan, dan juga menawarkan perencana rute dan pencari letak bisnis di U.S., Kanada, Jepang, HongKong, Cina, UK, Irlandia (hanya pusat kota) dan beberapa bagian Eropa. Google Maps masih berada dalam tahap beta.

III. ISI

Google map bekerja menggunakan algoritma A* dalam mencari jalur paling pendek dari start node ke final node. Cara kerja algoritma ini pada Google Map adalah sebagai berikut:

Start node di sini adalah Jalan Sadang Luhur dan tujuannya adalah gerbang depan ITB.

1. Anggap setiap tempat dan persimpangan sebagai sebuah node, dan jalan sebagai jalur pada node.
2. Anggap panjang jalan pada tiap node sebagai bobot.

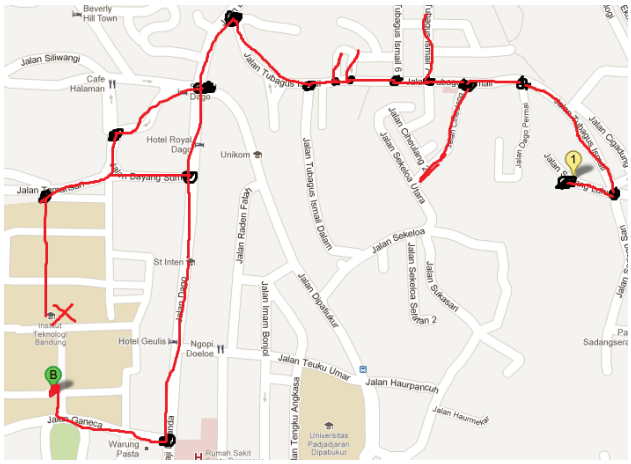


Gambar 3 Peta pada Google Map yang telah Diberi Node pada tiap tempat

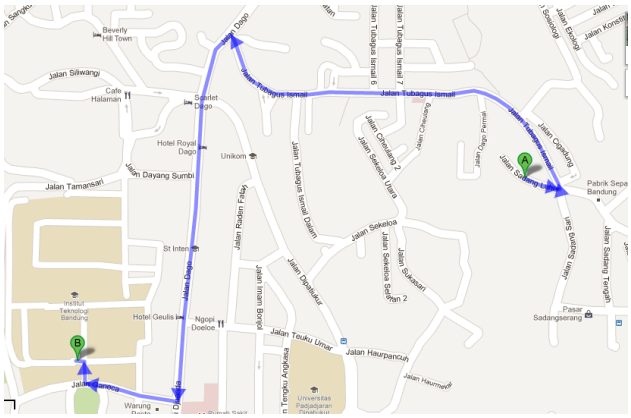
3. Telusuri node terdekat dan simpan semua node yang mungkin ke dalam *open set.* Dan masukkan

start node ke dalam *closed set.*

4. Hapus simpul dari open set dan menambahkannya ke closed set.
5. Periksa semua simpul yang berdekatan. Abaikan simpul yang termasuk closed set, tambahkan simpul ke open set jika belum berada pada open set.
6. Jika simpul yang berdekatan sudah ada pada open set, periksa apakah jalan tersebut merupakan yang lebih baik.
7. Ulangi langkah 4, 5, dan 6 sampai simpul tujuan termasuk ke dalam closed set.



Gambar 4 Hasil langkah 4, 5, dan 6 yang dilakukan berulang-ulang



Gambar 5 Path Solusi telah Ditemukan

IV. KESIMPULAN

1. Algoritma A* merupakan salah satu algoritma Branch & Bound yang selalu memberikan solusi yang optimal.
2. Algoritma A* menggunakan informasi tambahan (nilai heuristik) dalam pencarian solusi. Oleh karena itu, solusi yang optimal sangat tergantung kepada fungsi heuristik yang diterapkan.
3. Algoritma A* dapat diterapkan dalam menentukan shortest path pada Google Map dengan menelusuri tiap nodenya.

REFERENSI

[1] http://digilib.ittelkom.ac.id/index.php?option=com_content&view=article&id=1010:algoritma-a-&catid=21:itp-informatika-teori-dan-pemograman&Itemid=14, diakses tanggal 17 Desember 2012.

[2] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics SSC4* **4** (2): 100–10

[3] Russell, S. J.; Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, N.J.: Prentice Hall. pp. 97–104

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, 29 April 2010

ttd

Akbar Juang Saputra/13511026