BAB II

TINJAUAN PUSTAKA

II.1 Kecerdasan Buatan

Artificial Intelligence (AI) atau kecerdasan buatan merupakan cabang dari ilmu komputer yang berhubungan dengan pengautomatisasian tingkah laku cerdas. Pernyataan tersebut juga dapat dijadikan definisi dari AI. Definisi ini menunjukkan bahwa AI adalah bagian dari komputer sehingga harus didasarkan pada *sound theoretical* (teori suara) dan prinsip-prinsip aplikasi dari bidangnya. Prinsip-prinsip ini meliputi struktur data yang digunakan dalam representasi pengetahuan, algoritma yang diperlukan untuk mengaplikasikan pengetahuan tersebut, serta bahasa dan teknik pemrograman yang digunakan dalam mengimplementasikannya.

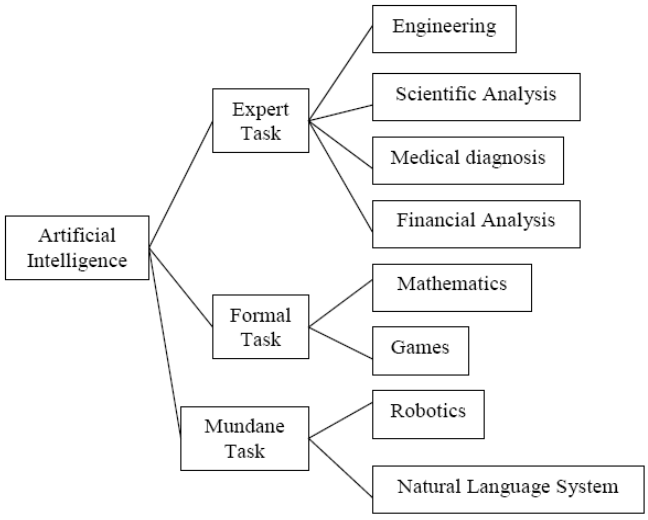
Dari beberapa perspektif, AI dapat dipandang sebagai:

1. Dari perspektif kecerdasan, AI adalah bagaimana membuat mesin yang cerdas dan dapat melakukan hal-hal yang sebelumnya hanya dapat dilakukan manusia.
2. Dari perspektif bisnis, AI adalah sekelompok alat bantu *(tools)* yang berdayaguna dan metodologi yang menggunakan alat-alat bantu tersebut untuk menyelesaikan masalah-masalah bisnis.
3. Dari perspektif pemrograman, AI meliputi studi tentang pemrograman simbolik, pemecahan masalah, dan proses pencarian *(search)*.

4. Dari perspektif penelitian:
    a. Riset tentang AI dimulai pada awal tahun 1960-an, percobaan pertama adalah membuat program permainan catur, membuktikan teori, dan *general problem solving*.
    b. *Artificial intelligence* adalah nama pada akar dari studi area.

Ada dua hal yang sangat mendasar mengenai penelitian-penelitian AI, yaitu *knowledge representation* (representasi pengetahuan) dan *search* (pelacakan). Para peneliti AI terus mengembangkan berbagai jenis teknik baru dalam menangani sejumlah permasalahan yang tergolong ke dalam AI seperti *vision* dan percakapan, pemrosesan bahasa alami, dan permasalahan khusus seperti diagnosa medis.

AI seperti bidang ilmu lainnya juga memiliki sejumlah sub-disiplin ilmu yang sering digunakan untuk pendekatan yang esensial bagi penyelesaian suatu masalah dan dengan aplikasi bidang AI yang berbeda. Gambar II.1 merupakan sejumlah bidang-bidang tugas *(task domains)* dari AI.



Gambar II.1. Bidang-bidang tugas (*task domains*) dari AI

Aplikasi penggunaan AI dapat dibagi ke dalam tiga kelompok, yaitu :

a. *Mundane task*

Secara harfiah, arti *mundane* adalah keduniaan. Di sini, AI digunakan untuk melakukan hal-hal yang sifatnya duniawi atau melakukan kegiatan yang dapat membantu manusia. Contohnya :

1. Persepsi (*vision* & *speech*).
2. Bahasa alami (*understanding, generation & translation*).
3. Pemikiran yang bersifat *commonsense*.
4. *Robot control*.

b. *Formal task*

AI digunakan untuk melakukan tugas-tugas formal yang selama ini manusia biasa lakukan dengan lebih baik. Contohnya :

1. Permainan/*games*.
2. Matematika (geometri, logika, kalkulus, integral, pembuktian).

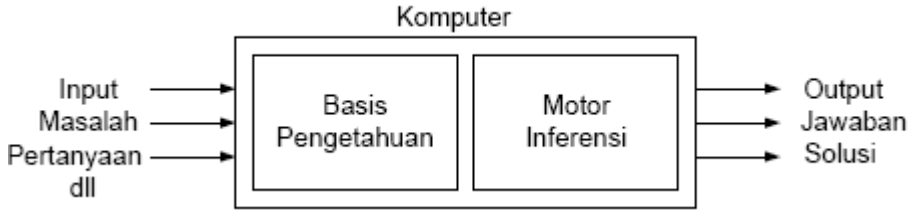
c. *Expert task*

AI dibentuk berdasarkan pengalaman dan pengetahuan yang dimiliki oleh para ahli. Penggunaan ini dapat membantu para ahli untuk menyampaikan ilmu-ilmu yang mereka miliki. Contohnya :

1. Analisis finansial
2. Analisis medikal
3. Analisis ilmu pengetahuan
4. Rekayasa (desain, pencarian, kegagalan, perencanaan, manufaktur)

Aplikasi *Artificial Intelegent* memiliki dua bagian utama, yaitu :

a. Basis Pengetahuan (*Knowledge Base*) : berisi fakta - fakta, teori, pemikiran dan hubungan antara satu dengan lainnya.
b. Motor Inferensi (*Inference Engine*) : kemampuan menarik kesimpulan berdasarkan pengalaman.



Gambar II.2. Penerapan konsep kecerdasan buatan di komputer [8]

II.2 Teknik-Teknik Dasar Pencarian

Pencarian atau pelacakan merupakan salah satu teknik untuk menyelesaikan permasalahan AI. Keberhasilan suatu sistem salah satunya ditentukan oleh kesuksesan dalam pencarian dan pencocokan. Teknik dasar pencarian memberikan suatu kunci bagi banyak sejarah penyelesaian yang penting dalam bidang AI. Ada beberapa aplikasi yang menggunakan teknik pencarian ini, yaitu :

1. Papan *game* dan *puzzle* (tic-tac-toe, catur, menara hanoi).
2. Penjadwalan dan masalah *routing (travelling salesman problem)*.
3. *Parsing* bahasa dan inteprestasinya (pencarian struktur dan arti).
4. Logika pemrograman (pencarian fakta dan implikasinya).
5. *Computer vision* dan pengenalan pola.
6. Sistem pakar bebasis kaidah *(rule based expert system)*.

Pencarian adalah proses mencari solusi dari suatu permasalahan melalui sekumpulan kemungkinan ruang keadaan *(state space)*. Ruang keadaan merupakan suatu ruang yang berisi semua keadaan yang mungkin. Kondisi suatu pencarian meliputi :

1. Keadaan sekarang atau awal.
2. Keadaan tujuan-solusi yang dijangkau dan perlu diperiksa apakah telah mencapai sasaran.
3. Biaya atau nilai yang diperoleh dari solusi.

Solusi merupakan suatu lintasan dari keadaan awal sampai keadaan tujuan. Secara umum, proses pencarian dapat dilakukan seperti berikut :

1. Memeriksa keadaan sekarang atau awal.
2. Mengeksekusi aksi yang dibolehkan untuk memindahkan ke keadaan berikutnya.
3. Memeriksa jika keadaan baru merupakan solusinya. Jika tidak, keadaan baru tersebut menjadi keadaan sekarang dan proses ini diulangi sampai solusi ditemukan atau ruang keadaan habis terpakai. [8]

II.2.1 Masalah Pencarian

Masalah pencarian merupakan proses pencarian solusi yang direncanakan, yang mencari lintasan dari keadaan sekarang sampai keadaan tujuan. Suatu masalah pencarian direpresentasikan sebagai graf berarah. Keadaan direpresentasikan sebagai simpul *(node)*, sedangkan langkah yang dibolehkan atau aksi direpresentasikan dengan busur *(arc)*. Dengan demikian, secara khusus masalah pencarian didefinisikan sebagai berikut :

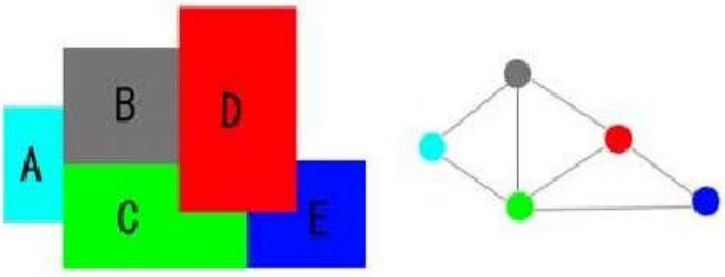
1. *State space* (ruang keadaan).
2. *Start node* (permukaan simpul).
3. Kondisi tujuan dan uji untuk mengecek apakah kondisi tujuan ditemukan atau tidak.
4. Kaidah yang memberikan bagaimana mengubah keadaan.

Terdapat beberapa cara untuk merepresentasikan ruang keadaan, antara lain : pohon *AND/OR*, graph keadaan, dan pohon pelacakan. [24]

II.2.1.1 Graph Keadaan

Graph terdiri dari *node-node* yang menunjukkan keadaan yaitu keadaan awal dan keadaan baru yang akan dicapai dengan menggunakan operator. *Node-node* dalam graph keadaan saling dihubungkan dengan menggunakan *arc* (busur) yang diberi panah untuk menunjukkan arah dari suatu keadaan ke keadaan berikutnya.

Konsep graph ini pertama diperkenalkan oleh seorang matematikawan dari Swiss, Leonhard Euler pada abad 18. Pada representasi graph, *vertex* digambarkan dengan *node* sedangkan *edge* digambarkan sebagai yang menghubungkan *node-node* tersebut. Contoh representasi graph terhadap suatu peta dapat dilihat pada gambar II.3.



Gambar II.3. Sebuah peta dan representasi graphnya [8]

II.2.1.2 Pohon Pencarian

Pelacakan adalah teknik pencarian sesuatu. Dalam pencarian ada dua kemungkinan hasil yang didapat yaitu menentukan dan tidak menemukan. Sehingga pencarian merupakan teknik yang penting dalam *artificial intellegance.* Hal ini penting dalam menentukan keberhasilan sistem berdasar kecerdasan adalah kesuksesan dalam pencarian dan pencocokan. Keberhasilan dan kualitas pencarian diukur dari empat cara yaitu :

a. Kelengkapan

   Apakah algoritma pencarian menjamin untuk mendapatkan sebuah penyelesaian jika ada penyelesaian?

b. Optimal

   Apakah algoritma pencarian akan mendapatkan penyelesaian optimal (misal : penyelesaian dengan biaya lintasan minimum)

c. Kekompleksan waktu

   Berapa lama waktu yang digunakan untuk penyelesaian permasalahan?

d. Kekompleksan ruang

   Berapa banyak memori yang dibutuhkan untuk melakukan pencarian [24]

Untuk menghindari kemungkinan adanya proses pelacakan suatu node secara berulang, maka digunakan struktur pohon. Struktur pohon digunakan untuk menggambarkan keadaan secara hirarkis. Pohon juga terdiri dari beberapa *node*. *Node* yang terletak pada level-0 disebut juga “akar”. *Node* akar menunjukkan keadaan awal yang biasanya merupakan topik atau objek. *Node* akar ini terletak pada level-0. *Node* akar mempunyai beberapa percabangan yang terdiri atas

beberapa *node successor* yang sering disebut dengan nama “anak” dan merupakan *node-node* perantara.

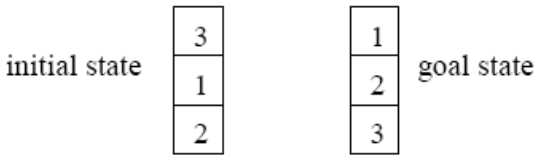
Namun jika dilakukan pencarian mundur, maka dapat dikatakan bahwa *node* tersebut memilki *predecessor*. *Node-node* yang tidak mempunyai anak sering disebut dengan nama *node* “daun” yang menunjukkan akhir dari suatu pencarian, dapat berupa tujuan yang diharapkan *(goal)* atau jalan *(dead end).* [25]

II.2.2 Contoh Pencarian

Misalkan ada tiga kotak 1, 2, 3 pada sebuah papan. Sebuah kotak dapat dipindahkan jika tidak ada kotak lain di atasnya dan hanya ada satu kotak yang boleh dipindahkan. Ada dua kemungkinan pemindahannya, yaitu:

1. Pindahkan sebuah kotak ke atas papan.
2. Pindahkan sebuah kotak ke atas kotak lainnya.

Masalah muncul jika diketahui keadaan awalnya (*initial state* atau *current state*) dan tujuan akhirnya (*goal state* atau *final state*) seperti pada gambar 2.4.

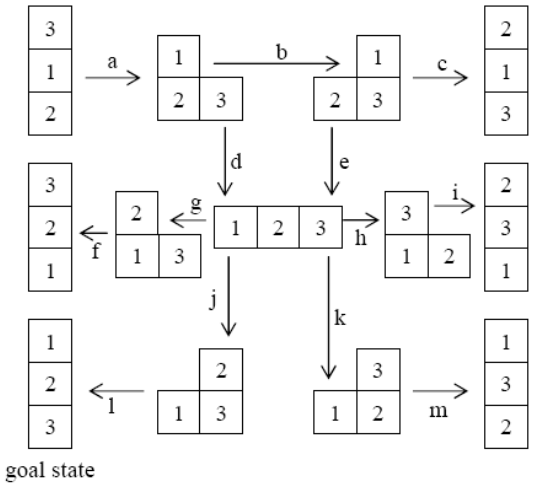


Gambar II.4. Contoh permainan yang merupakan pencarian

Pada gambar II.4 dapat dilihat bahwa ruang keadaan tersebut memiliki 13 elemen atau *node*, dengan perpindahan sebagai berikut:

1. Perpindahan kotak 1 ke atas papan untuk lintasan d dan e.
2. Perpindahan kotak 3 ke atas papan untuk lintasan a.
3. Perpindahan kotak 1 ke atas kotak 2 untuk lintasan l.

4. Perpindahan kotak 1 ke atas kotak 3 untuk lintasan b dan m.
5. Perpindahan kotak 2 ke atas kotak 1 untuk lintasan c dan g.
6. Perpindahan kotak 2 ke atas kotak 3 untuk lintasan i dan j.
7. Perpindahan kotak 3 ke atas kotak 1 untuk lintasan h.
8. Perpindahan kotak 3 ke atas kotak 2 untuk lintasan f dan k.



Gambar II.5. Ruang keadaan

Penyelesaian untuk masalah permainan pada gambar II.5 adalah anggota kumpulan semua lintasan dari keadaan awal hingga tujuan yang lintasannya ditandai dengan huruf a, d, j, dan l.

Secara umum, algoritma pencarian dapat dituliskan seperti berikut :

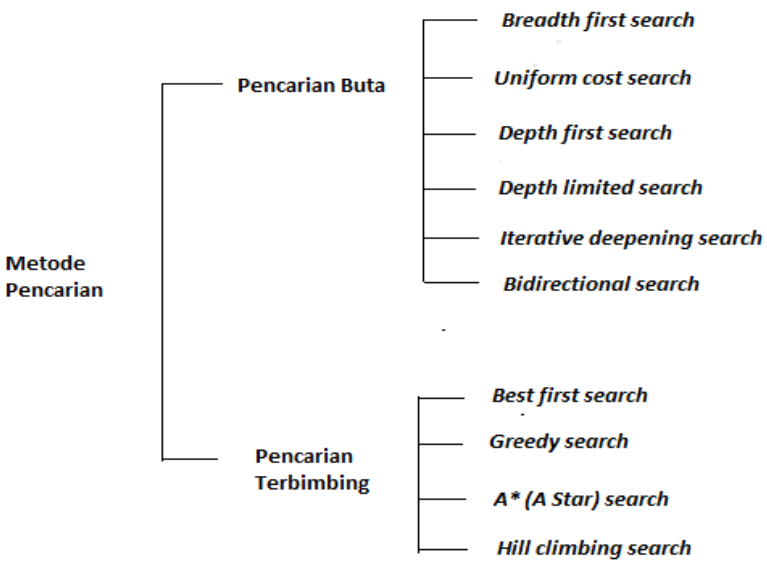
```
Function Gsearch (Problem, QueueingFn): Solution |
Failure
 Var nodes : structure;
 Begin
  While
   Begin
    Node := remove_front_nodes(nodes);
    IF Goal_test(problem,STATE(node)) succeeds then
     Solution := Node
    Else
     Nodes := QueueningFn(node, OPERATOR(problem));
   End;
  EndWhile
 End;
```

II.3 Algoritma Pencarian (*Search Algoritms*)

Permasalahan pencarian dapat diselesaikan dengan beberapa metode yaitu:

1. Metode pencarian yang pertama adalah metode yang sederhana yang hanya berusaha mencari kemungkinan penyelesaian yang disebut juga pencarian buta.
2. Metode yang lebih kompleks yang akan mencari jarak terpendek. Metode ini adalah *British Museum Procedure, Branch and Bound, Dynamic Programming, Best First Search,Greedy Search,* A* *(A Star) Search,* dan *Hill Climbing Search.* Metode-metode ini digunakan pada saat harga perjalanan untuk mencari kemungkinan menjadi perhitungan. Beberapa *procedure*/metode yang kita terapkan saat berhadapan dengan musuh. Prosedur ini adalah *minimax search* dan *alpha-beta pruning.* Metode ini banyak digunakan pada program-program seperti catur dan sebagainya.

Metode pencarian dikatakan penting untuk meyelesaikan permasalahan karena setiap *state* (keadaan) menggambarkan langkah-langkah untuk menyelesaikan permasalahan. Metode pencarian dikatakan penting untuk perencanaan karena dalam sebuah permainan akan menentukan apa yang harus dilakukan, dimana setiap *state* menggambarkan kemungkinan posisi pada suatu saat. Metode pencarian adalah bagian dari kesimpulan, dimana setiap *state* menggambarkan hipotesis dalam sebuah rangkaian deduktif. [25]



Gambar II.6. Bagan metode pencarian (*searching*)

Menurut cara algoritma mengembangkan *node* dalam proses pencarian, gambar bagan metode penulusuran dibagi menjadi dua golongan, yakni pencarian buta *(blind search)* dan pencarian terbimbing *(heuristic search)*. [8]

II.3.1 Pencarian Buta (*Blind Search*)

Blind Search adalah pencarian solusi tanpa adanya informasi yang dapat mengarahkan pencarian untuk mencapai *goal state* dari *current state* (keadaan sekarang). Informasi yang ada hanyalah definisi *goal state* itu sendiri, sehingga algoritma dapat mengenali *goal state* bila menjumpainya.

Dengan ketiadaan informasi, maka *blind search* dalam kerjanya memeriksa/mengembangkan *node-node* secara tidak terarah dan kurang efisien untuk kebanyakan kasus karena banyaknya *node* yang dikembangkan.

Beberapa contoh algoritma yang termasuk *blind seacrh* antara lain adalah *Breadth First Search, Uniform Cost Search, Depth First Search, Depth Limited Search, Iterative Deepening Search,* dan *Bidirectional Search.* [1]

II.3.2 Pencarian Terbimbing (*Heuristic Search*)

Berbeda dengan *blind search*, *heuristic search* mempunyai informasi tentang *cost*/biaya untuk mencapai *goal state* dari *current state.* Dengan informasi tersebut, *heuristic search* dapat melakukan pertimbangan untuk mengembangkan atau memeriksa *node-node* yang mengarah ke *goal state.* Misalnya pada pencarian *rute* pada suatu peta, bila kita berangkat dari kota A ke kota tujuan B yang letaknya di Utara kota A, dengan *heuristic search*, pencarian akan lebih difokuskan ke arah Utara (dengan informasi *cost* ke *goal*), sehingga secara umum, *heuristic search* lebih efisien daripada *blind search.*

Heuristic search untuk menghitung (perkiraan) *cost* ke *goal state*, digunakan fungsi *heuristic.* Fungsi *heuristic* berbeda daripada algoritma, dimana *heuristic* lebih merupakan perkiraan untuk membantu algoritma, dan tidak harus valid setiap waktu. Meskipun begitu, semakin bagus fungsi *heuristic* yang dipakai, semakin cepat dan akurat pula solusi yang didapat. Menentukan *heuristic* yang tepat untuk kasus dan implementasi yang ada juga sangat berpengaruh terhadap kinerja algoritma pencarian.

Beberapa contoh algoritma pencarian yang menggunakan metode *heuristic search* adalah : *Best First Search, Greedy Search,* A* *(A Star) Search,* dan *Hill Climbing Search.*

Tabel II.1. Kegunaan *heuristic search*

Algorima Pencarian	Kegunaan	Implementasi
Best First Search	Penelusurannya hanya menggunakan estimasi *cost* / jarak ke node tujuan, h(n) akibatnya pencarian tidak menyeluruh.	Sistem pakar, penjadwalan, pencarian *rute* pada peta geografis dan lain-lain.
Greedy Search	Pada *greedy search* ide utamanya adalah mengembangkan node dengan nilai estimasi biaya ke *goal* yang paling kecil (berarti *node* yang paling dekat ke tujuan).	Masalah penukaran uang, minimisasi waktu di dalam sistem (penjadwalan), memilih beberapa jenis investasi (penanaman modal), memilih jurusan di perguruan tinggi, masalah *knapsnak* dan lainnya.
A* *(A Star) Search*	Menjumlahkan jarak sebenarnya dengan estimasi jaraknya dan pencariannya menyeluruh, tetapi akan memakai memori yang cukup besar untuk menyinpan *node* sebelumnya.	Pencarian jalur terpendek pada peta, pencarian jalur angkutan kota, pencarian jalan pada *game*, menggerakkan karakter *game*, pencarian jalan untuk permainan *Lose Your Marble*, penyelesaian permasalahan 8 *Puzzle* dan lain-lain.
Hill Climbing Search	*Hill climbing* sering digunakan ketika fungsi *heuristic* yang bagus tersedia untuk mengevaluasi *state* tapi ketika tidak ada yang perlu dievaluasi maka fungsi ini tidak ada.	Sistem pakar, pencarian lokasi pada peta, dan lain-lain.

II.4 Algoritma A* (*A Star*)

Algoritma A* merupakan perbaikan dari metode BFS dengan memodifikasi fungsi *heuristic*nya. A* (*A Star*) akan meminimumkan total biaya lintasan. Pada kondisi yang tepat, A* akan memberikan solusi yang terbaik dalam waktu yang optimal. [8]

Pada pencarian *rute* kasus sederhana, dimana tidak terdapat halangan pada peta, A* bekerja secepat dan seefisien BFS. Pada kasus peta dengan halangan, A* dapat menemukan solusi *rute* tanpa ‘terjebak’ oleh halangan yang ada.

Pencarian menggunakan algoritma A* mempunyai prinsip yang sama dengan algoritma BFS, hanya saja dengan dua faktor tambahan.

1. Setiap sisi mempunyai “*cost*” yang berbeda-beda, seberapa besar *cost* untuk pergi dari satu simpul ke simpul yang lain.
2. *Cost* dari setiap simpul ke simpul tujuan bisa diperkirakan. Ini membantu pencarian, sehingga lebih kecil kemungkinan kita mencari ke arah yang salah.

Cost untuk setiap simpul tidak harus berupa jarak. *Cost* bisa saja berupa waktu bila kita ingin mencari jalan dengan waktu tercepat untuk dilalui. Sebagai contoh, bila kita berkendaraan melewati jalan biasa bisa saja merupakan jarak terdekat, tetapi melewati jalan tol biasanya memakan waktu lebih sedikit.

Algoritma A* bekerja dengan prinsip yang hampir sama dengan BFS, kecuali dengan dua perbedaan, yaitu :

1. Simpul-simpul di *list* “terbuka” diurutkan oleh *cost* keseluruhan dari simpul awal ke simpul tujuan, dari *cost* terkecil sampai *cost* terbesar. Dengan kata lain, menggunakan *priority queue* (antrian prioritas). *Cost* keseluruhan dihitung dari *cost* dari simpul awal ke simpul sekarang (*current node*) ditambah *cost* perkiraan menuju simpul tujuan.
2. Simpul di *list* “tertutup” bisa dimasukkan ke list “terbuka” bila jalan terpendek (*cost* lebih kecil) menuju simpul tersebut ditemukan.

Karena *list* "terbuka" diurutkan berdasarkan perkiraan *cost* keseluruhan, algoritma mengecek simpul-simpul yang mempunyai perkiraan *cost* yang paling kecil terlebih dahulu, jadi algoritmanya mencari simpul-simpul yang kemungkinan mengarah ke simpul tujuan. Karena itu, lebih baik perkiraan *cost*-nya, lebih cepat pencariannya. *Cost* dan perkiraannya ditentukan oleh kita sendiri. Bila *cost*-nya adalah jarak, akan menjadi mudah.

Cost antara simpul adalah jaraknya, dan perkiraan *cost* dari suatu simpul ke simpul tujuan adalah penjumlahan jarak dari simpul tersebut ke simpul tujuan. Atau agar lebih mudahnya bisa ditunjukkan seperti berikut ini.

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

dengan :

$f(n)$ = fungsi evaluasi

$g(n)$ = biaya *(cost)* yang sudah dikeluarkan dari keadaan sampai keadaan n

$h(n)$ = estimasi biaya untuk sampai pada suatu tujuan mulai dari n

Perhatikan bahwa algoritma ini hanya bekerja bila *cost* perkiraan tidak lebih besar dari *cost* yang sebenarnya. Bila *cost* perkiraan lebih besar, bisa jadi jalan yang ditemukan bukanlah yang terpendek. [19]

Node dengan nilai terendah merupakan solusi terbaik untuk diperiksa pertama kali pada $g(n) + h(n)$. Dengan fungsi *heuristic* yang memenuhi kondisi tersebut, maka pencarian dengan algoritma A* dapat optimal.

Keoptimalan dari A* ini cukup langsung untuk dianalisis apabila digunakan dengan *tree search.* Pada kasus ini, A* dinilai optimal jika *h(n)* adalah sebuah *admissible heuristic* yaitu nilai *h(n)* tidak akan memberikan penilaian lebih

pada *cost* untuk mencapai tujuan. Salah satu contoh dari *admissible heuristic* adalah jarak dengan menarik garis lurus karena jarak terdekat dari dua titik adalah dengan menarik garis lurus. Adapun pseudecode dari algoritma A* adalah sebagai berikut :

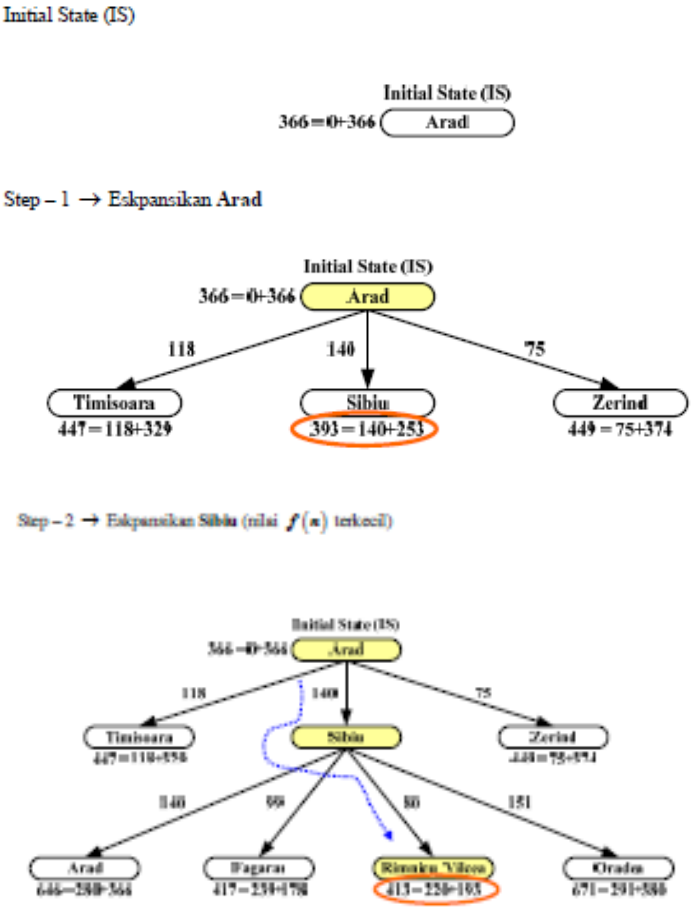
```
function a* (start,goal)
        closedset := the empty set                          % the set of nodes
already evaluated.
        openset := set containing the initial node          % the set of tentaive
nodes to be evaluated.
        g_score[start] := 0                                 % distance from start
along optimal path.
        h_score[start] := heuristic_estimate_of_distance[start, goal]
        f_score[start] := h_score[start]                    % estimated total
distance from start to goal throught y.
        while openset is not empty
                x := the node in openset having the lowest f_score[] value
                if x = goal
                        return reconstruct_path(came_from, goal)
                remove x from openset
                add x to closedset
                foreach y in neighbor_nodes[x]
                        if y in closedset
                                continue
                        tentativie_g_score := g_score[x] + dist_between[x,y]

                        if y not in openset
                                add y to openset

                                tentativie_is_better := true
                        elseif tentative_g_score < g_score[y]
                                tentative_is_better :=true
                        else
                                tentative_is_better := false
                        if tentative_is_better = true
                                came_from[y] := x
                                g_score[y] := tentative_g_score
                                h_score[y] := heuristic_estimate_of_distance[y, goal]
                                f_score[y] := g_score[y] + h_score[y]
        return failure

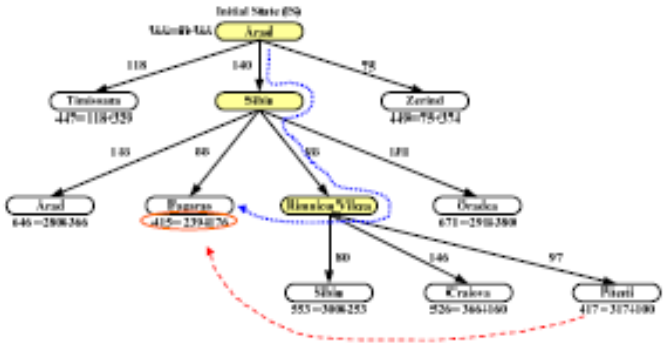
function reconstruct_path(came_from,current_node)
        if came_from[current_node] is set
             p = reconstruct+path(came_from, came_from[current_node])
             return (p+current_node)
        else
             return the empty path
```

Pada persoalan tentang *Romanian Paths* (Arab – Bucharest) yang terdapat pada buku *Russel* & *Norwig* penyelesaiannya dengan algoritma A* dapat dijelaskan pada gambar II.7.

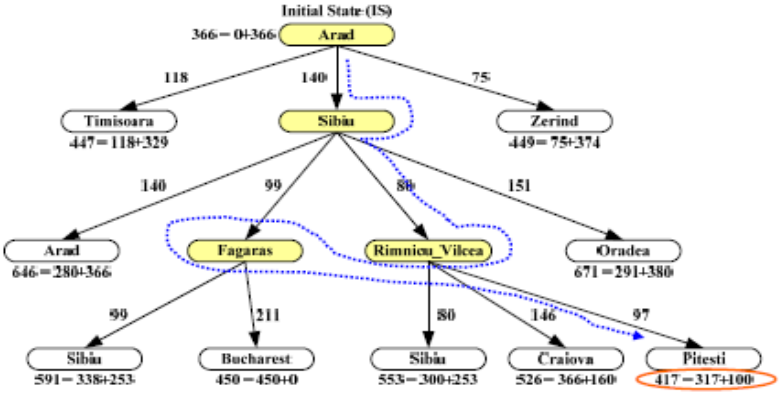


Gambar II.7. Contoh pencarian dengan A* [16, p.95]

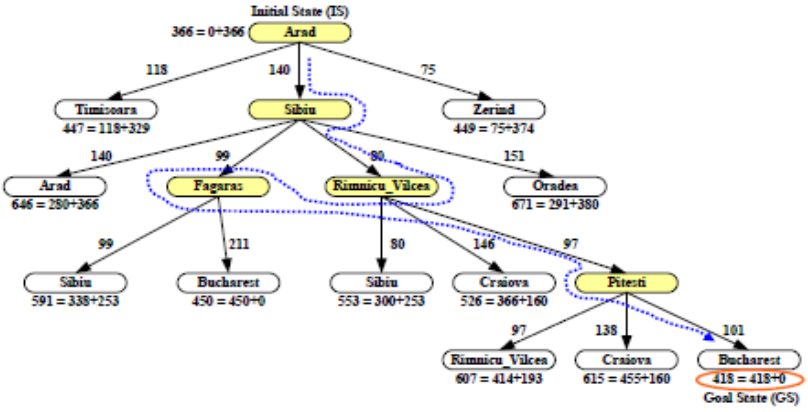
Step – 3 → Ekspansikan **Rimnicu_Vilcea** (nilai $f(n)$ terkecil)



Step – 4 → Ekspansikan **Fagaras** (nilai $f(n)$ terkecil)



Step – 5 → Ekspansikan **Pitesti** (nilai $f(n)$ terkecil)



Gambar II.7. Contoh pencarian dengan A* (lanjutan) [16, p.95]

Urutan pelacakan (*key* : Pilih *node* yang belum diekspansikan yang mempunyai fungsi heuristik $\boldsymbol{f}$ (***n***) terkecil). Kesimpulan berdasarkan penelusuran di atas, *solution path*-nya adalah : Arad – Sibiu – Rimnicu_Vilcea – Fagaras – Pitesti – Bucharest.

Metode A* mirip dengan algoritma pencarian graph yang berpotensial mencari daerah yang luas pada sebuah peta. Metode A* mempunyai fungsi *heuristic* untuk memandu pencarian ke depan sampai tujuan. Metode A* dapat melakukan *backtracking* jika jalur yang ditempuh ternyata salah. Metode A* dapat melakukannya karena menyimpan jejak / *track* yang mungkin sebagai jalur yang optimal. Sebagai contoh, jika kita sedang menuju suatu kota dan sampai pada persimpangan jalan, dan memutuskan untuk belok kiri daripada ke kanan, dan ternyata bila jalan yang dipilih ternyata salah, kita akan kembali ke persimpangan dan mengambil jalan satunya. Itulah yang dilakukan metode A* ini. [23]

Performansi algoritma A* dapat diketahui dengan melihat perbandingan metode dari tabel II.2.

Tabel II.2. Perbedaan ketiga algoritma

No.	Nama algoritma	Metode	Keterangan
1.	*Optimal Search (Djikstra's Algorithm)*	$g(n_i)=g(n)+c(n,n_i)$	*g(n)* adalah *cost* dari IS ke *node n* $c(n,n_i)$ adalah *cost* dari *node n* ke n_i
2.	*Best First Search*	*h(n)*	*h(n)* adalah estimated cost dari jalur terpendek dari node n ke GS
3.	*A**	*f(n)=g(n)+h(n)*	*f(n)* adalah fungsi *heruistic* *g(n)* adalah *cost* dari IS ke *node n* *h(n)* adalah *estimated cost* dari jalur terpendek dari node n ke GS

Dari informasi pada tabel II.2 di atas dapat dibuktikan bahwa bila :

a. $\boldsymbol{g}$ ($\boldsymbol{n}$) = **0** , maka $\boldsymbol{f}$ ($\boldsymbol{n}$) = $\boldsymbol{h}$($\boldsymbol{n}$) sehingga algoritma A* akan bertingkah laku sebagaimana *Best First Search*.

b. $\boldsymbol{h}$($\boldsymbol{n}$) = **0**, maka $\boldsymbol{f}$ ($\boldsymbol{n}$) = $\boldsymbol{g}$ ($\boldsymbol{n}$) sehingga algoritma A* akan bertingkah laku sebagaimana *Optimal Search (Dijkstra's Algorithm)*.

Dengan demikian dapat diambil kesimpulan bahwa algoritma A* mengkombinasikan kelebihan dari algoritma *Optimal Search* dan *Best First Search*. Dari informasi ini, kita dapat menganalisa dan membandingkan *cost* ketiga algoritma tersebut.

II.4.1 Kompleksitas Algoritma A* (*A Star*)

Kompleksitas waktu dari algoritma A* tergantung dari *heuristic*nya. Dalam kasus terburuk (*worst case*), jumlah simpul yang diekspansi bisa eskponensial dalam solusinya (jalan tependek). Akan tetapi, kompleksitasnya bisa berupa polinomial bila fungsi heuristik *h* bertemu kondisi berikut:

$$| h(x) - h^{*}(x) | = O(\log h^{*}(x))$$

Dimana *h** adalah heuristik optimal, atau *cost* pasti untuk menuju tujuan dari *x*. Dengan kata lain, kesalahan (*error*) dari *h* tidak boleh tumbuh lebih cepat dari algoritma "*perfect heuristic*" *h** yang mengembalikan jarak sebenarnya dari *x* menuju tujuan. [16]

II.4.2 Efisien Waktu Algoritma A*

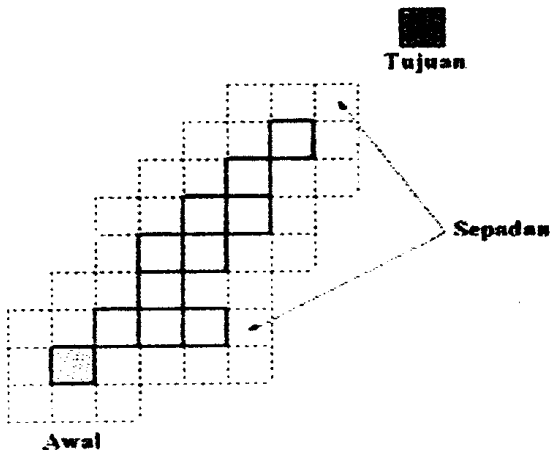
Dengan digunakannya fungsi *heuristic* H(n), algoritma A* dapat memfokuskan pencarian pada *node-node* yang berada pada arah yang mendekati *node* tujuan. Kemudian pencarian diterminasikan pada waktu *node* tujuan

diperiksa. Hal ini dapat meminimalisasikan jumlah *node* yang harus diperiksa dan arena waktu yang diperlukan untuk mendapatkan jalur berbanding lurus dengan jumlah *node* yang diperiksa, maka waktu pencarian dapat diminimalisasikan.

Walaupun jumlah *node* yang diperiksa dapat diminimalisasikan, algoritma A* mempunyai kasus terburuk. Pada kasus ini, sebagian besar ataupun keseluruhan *node* pada jalan diperiksa, sehingga algoritma A* bekerja seperti algoritma dijkstra atau BFS *(Best-First-Search).* Ada dua hal yang dapat menyebabkan keadaan terburuk ini, yaitu keadaan sepadan dan jika jalur yang dicari tidak ditemukan. [10]

II.4.3 Keadaan Sepadan pada Algoritma A*

Jika dua atau lebih *node* yang diperiksa mempunyai harga *f(n)* yang sama, maka keadaan sepadan *(tie)* terjadi. Hal ini dimungkinkan karena *f(n)* bergantung pada dua fungsi, yaitu fungsi *g(n)* dan *h(n).* Hal ini sangat mungkin terjadi antara *node-node* yang letaknya berjauhan, dan kemungkinan besar *node* yang satu terletak dekat *node* tujuan sedangkan yang lainnya terletak jauh dari *node* tujuan.



Gambar II.8. Keadaan sepadan

Karena algoritma A* memberikan prioritas berdasarkan harga *f(n)*, maka jika keadaan sepadan terjadi, terdapat lebih dari satu *node* dengan prioritas sama. Akibatnya adalah *node-node* tersebut akan diperiksa lebih dulu, yang mungkin *node* tersebut terletak berjauhan dengan *node* tujuan. Hal ini berakibat turunnya kinerja algoritma A*. [10]

II.4.4 Fungsi *Heuristic*

BFS dan A* sebagai algoritma pencarian yang menggunakan fungsi *heuristic* untuk 'menuntun' pencarian *rute*, khususnya dalam hal pengembangan dan pemeriksaan *node-node* pada peta. Dalam aplikasi ini, fungsi *heuristic* yang dipakai untuk pencarian *rute* mengisi nilai/notasi h pada algoritma BFS dan A*. Ada beberapa fungsi *heuristic* umum yang bisa dipakai untuk algoritma BFS dan A* ini. Salah satunya adalah yang dikenal dengan istilah '*Manhattan Distance*'. Fungsi *heuristic* ini digunakan untuk kasus dimana pergerakan pada peta hanya lurus (horizontal atau vertikal), tidak diperbolehkan pergerakan diagonal. [15]



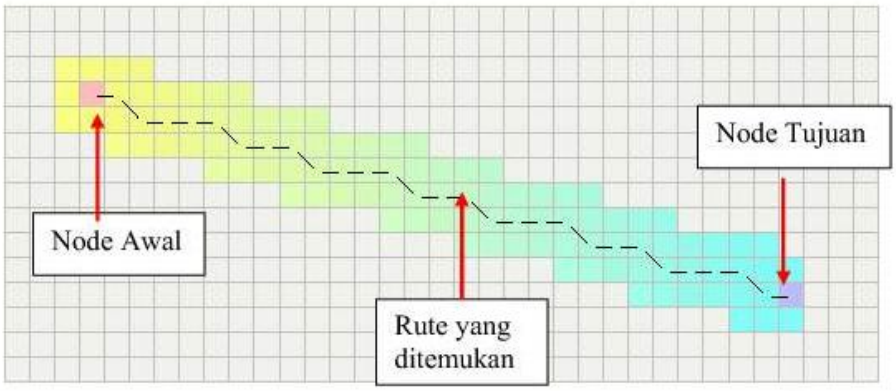
Gambar II.9. *Rute* dengan langkah diagonal tidak diperbolehkan

Perhitungan nilai *heuristic* untuk *node* ke-n menggunakan *Manhattan Distance* adalah sebagai berikut :

```
h(n) = (abs(n.x - goal.x) + abs(n.y - goal.y))
```

Dimana h(n) adalah nilai *heuristic* untuk *node* n, dan *goal* adalah *node* tujuan.

Jika pergerakan diagonal pada peta diperbolehkan, maka digunakan fungsi *heuristic* selain *Manhattan Distance*. Untuk mendekati kenyataan, *cost* untuk perpindahan *node* secara diagonal dan orthogonal dibedakan. *Cost* diagonal adalah 1,4 kali *cost* perpindahan secara orthogonal.



Gambar II.10. *Rute* dengan langkah diagonal diperbolehkan

Maka fungsi *heuristic* yang digunakan adalah sebagai berikut:

```
h_diagonal(n) = min(abs(n.x - goal.x) + abs(n.y – goal.y))
h_orthogonal(n) = (abs(n.x - goal.x) + abs(n.y – goal.y))
h(n) = h_diagonal(n) + (h_orthogonal (n) – (2 * h_diagonal(n)))
```

Dimana h_diagonal(n) adalah banyaknya langkah diagonal yang bisa diambil untuk mencapai goal dari *node* n. h_orthogonal adalah banyaknya langkah lurus yang bisa diambil untuk mencapai *goal* dari *node* n.

Nilai *heuristic* kemudian diperoleh dari h_diagonal(n) ditambah dengan selisih h_orthogonal(n) dengan dua kali h_diagonal(n). Dengan kata lain, jumlah langkah diagonal kali *cost* diagonal ditambah jumlah langkah lurus yang masih bisa diambil dikali *cost* pergerakan lurus. [14]

II.5 *Linked List*

Dikembangkan tahun 1955-1956 oleh Allen Newell, Cliff Shaw dan Herbert Simon di RAND *Corporation* sebagai struktur data utama untuk bahasa *Information Processing Language* (IPL). IPL dibuat untuk mengembangkan program *artificial intelligence*, seperti pembuatan *Chess Solver*. Victor Yngve di *Massachusetts Institute of Technology* (MIT) juga menggunakan *linked list* pada *natural language processing* dan *machine transitions* pada bahasa pemrograman COMMIT.

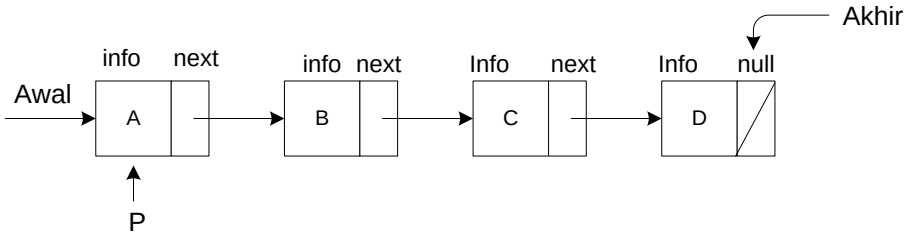
Linked list adalah salah satu bentuk struktur data, berisi kumpulan data (*node*) yang tersusun secara sekuensial, saling sambung menyambung, dinamis dan tidak terbatas. Pada tabel II.3 dapat disimpulkan *linked list* lebih optimal daripada *array*. *Linked list* sering disebut juga senarai berantai dan saling terhubung dengan bantuan variabel pointer. Masing-masing data dalam *linked list* disebut dengan *node* (simpul) yang menempati alokasi memori secara dinamis dan biasanya berupa *struct* yang terdiri dari beberapa *field*.

Tabel II.3. Perbandingan *array* dan *linked list*

Array	***Linked list***
Statis	Dinamis
Penambahan/penghapusan data terbatas	Penambahan/penghapusan data tidak terbatas
Random access	*Sequential access*
Penghapusan *array* tidak mungkin	Penghapusan *linked list* mudah

Linked list terdiri dari tiga jenis yaitu *single linked list, double linked list* dan *circular linked list*. Salah satu yang akan dijelaskan yaitu tentang *single linked list*.

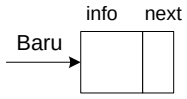
Single linked list adalah *linked list* dengan simpul berisi satu *link*/pointer yang mengacu ke simpul berikutnya, ilustrasinya dijelaskan pada gambar II.11.



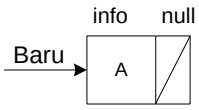
Gambar II.11. Ilustrasi *single linked list*

Setiap *node* pada *linked list* mempunyai *field* yang berisi pointer ke *node* berikutnya, dan juga memiliki *field* yang berisi data. Pada akhir *linked list*, *node* terakhir akan menunjuk ke NULL yang akan digunakan sebagai kondisi berhenti pada saat pembacaan isi *linked list*.

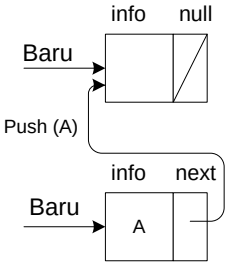
Penambahan *node* baru memiliki tiga cara penyisipan yaitu sisip didepan, sisip ditengah dan sisip diakhir. Pada pembahasan ini akan lebih dijelaskan tentang sisip didepan, penambahan data didepan akan dikaitan di *node* paling awal, namun pada saat pertama kali (data masih kosong), maka penambahan data dilakukan dengan cara ‘Awal’ ditunjukkan ke *node* baru tersebut. Salah satu contoh penambahan data baru yaitu ‘A’ dan ‘B’ dengan penyisipan didepan sebagai berikut :



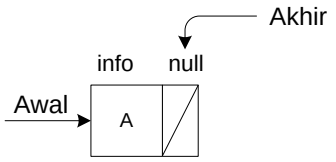
data masih kosong



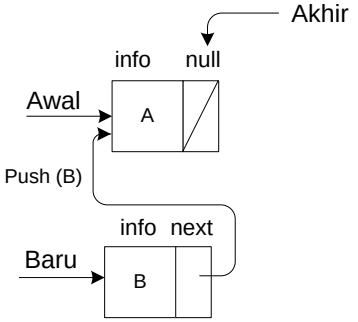
data baru yang akan ditambahkan

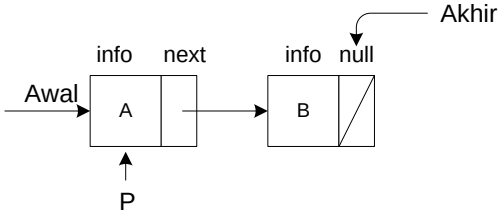


proses penambahan data baru



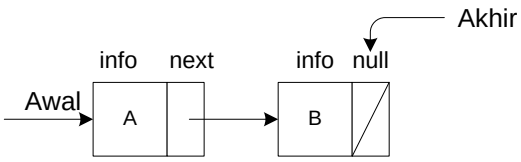
hasil akhir setelah data ditambahkan



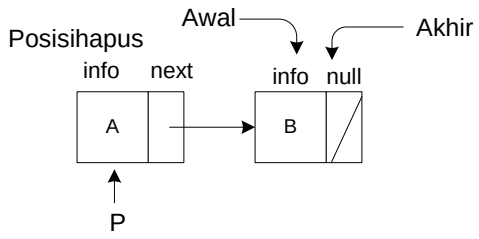


Penambahan data (B)

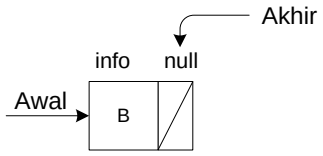
Penghapusan data *node* memiliki tiga cara yaitu hapus didepan, hapus ditengah dan hapus diakhir. Pada pembahasan ini akan dijelaskan tentang hapus didepan.



kondisi *linked list* memiliki data awal lebih dari 1 data, kemudian akan dihapus data ‘A’ yang terletak pada posisi paling depan.



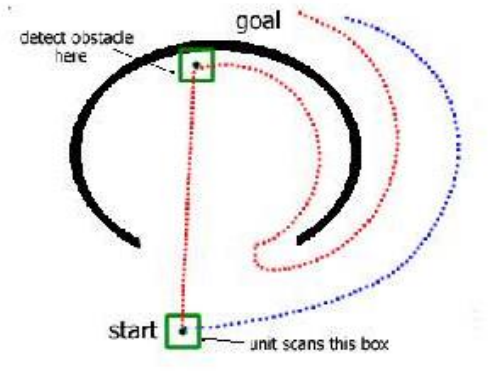
Setelah ‘Awal’ dipindahkan ke data berikutnya maka hapus/hancurkan data di Posisihapus, sehingga *linked list* menjadi seperti di bawah ini.



II.6 *Pathfinding*

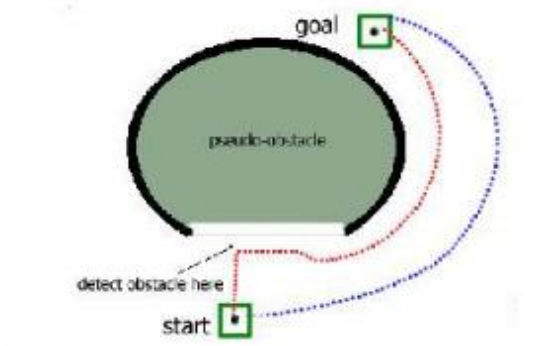
Pathfinding (pencarian jalan/*rute*) adalah salah satu bidang penerapan yang sering ditangani oleh kecerdasan buatan khususnya dengan menggunakan algoritma pencarian. Penerapan yang dapat dilakukan dengan *pathfinding* antara lain adalah pencarian *rute* dalam suatu game dan pencarian jalan/*rute* pada suatu peta. Algoritma pencarian yang dipakai harus dapat mengenali jalan dan elemen peta yang tidak dapat dilewati.

Sebuah algoritma *pathfinding* yang baik dapat bermanfaat untuk mendeteksi halangan/rintangan yang ada pada medan dan menemukan jalan menghindarinya, sehingga jalan yang ditempuh lebih pendek daripada yang seharusnya bila tidak menggunakan algoritma *pathfinding*. Lihat ilustrasi pada gambar II.12.



Gambar II.12. Penentuan *rute* tanpa *pathfinding*

Pada gambar II.12, dari *start* menuju *goal*, tanpa algoritma *pathfinding*, unit hanya akan memeriksa lingkungan sekitarnya saja (dilambangkan dengan daerah di dalam kotak hijau). Unit tersebut akan maju terus ke atas untuk mencapai tujuan, baru setelah mendekati adanya halangan, lalu berjalan memutar untuk menghindarinya. Sebaliknya, penentuan *rule* dengan algoritma *pathfinding* pada gambar II.13 akan memprediksi ke depan mencari jalan yang lebih pendek menghindari halangan (dilambangkan garis biru) untuk mencapai tujuan, tanpa pernah mengirim unit ke dalam 'perangkap' halangan berbentuk U. Karena itu peran algoritma *pathfinding* sangat berguna untuk memecahkan berbagai permasalahan dalam penentuan *rute*.



Gambar II.13. Penentuan *rute* dengan *pathfinding*

II.7 Perkembangan *Game* Komputer

Permainan komputer (*game*) adalah program komputer yang terdiri dari dunia maya yang dikendalikan oleh sebuah komputer di mana pemainnya bisa berinteraksi untuk mencapai sejumlah tujuan (*goal*).

Permainan komputer juga dapat digolongkan ke dalam beberapa aliran/*genre*:

a. Laga (*Action*)

b. Petualangan (*Adventure*)

c. Bermain Peran (*Role-Playing* (RPG))

d. Simulasi (*Simulation*)

e. Olahraga (*Sports*)

Game pathfinding sendiri dapat digolongkan ke dalam *genre* simulasi (*simulation*) yang berkaitan erat dengan bentuk kotak. Permainan ini akan memiliki fungsi *game engine* yaitu *artificial intellegence* dalam proses pencariannya.

Game Engine disebut juga komponen inti dari sebuah *game* atau aplikasi interaktif lainnya yang disajikan secara *real time*. *Game engine* memberikan teknologi dasar, mempermudah pengembangan sebuah *game* dan bahkan memberikan teknologi yang dapat dijalankan dengan *platform* yang berdeda-beda, seperti *game console*, Operasi sistem berbasis desktop *Linux, Max OS,* dan *Microsoft Windows*. Fungsi-fungsi utama yang biasanya disediakan oleh *game engine* termasuk didalamnya *rendering engine* (renderer) untuk grafik 2D atau 3D, *physic engine* atau deteksi benturan (*collation detection*), suara, *scripting*, *animation*, AI (*Artificial Intelligence*), *networking* dan susunan adegan/kejadian/film pendek (*scene graph*).

AI (*Artificial Intelligence*) bisa dikatakan cukup berperan pada sebuah *game*. AI pada *game* biasa berinteraksi dengan *player* dalam berbagai hal, mulai

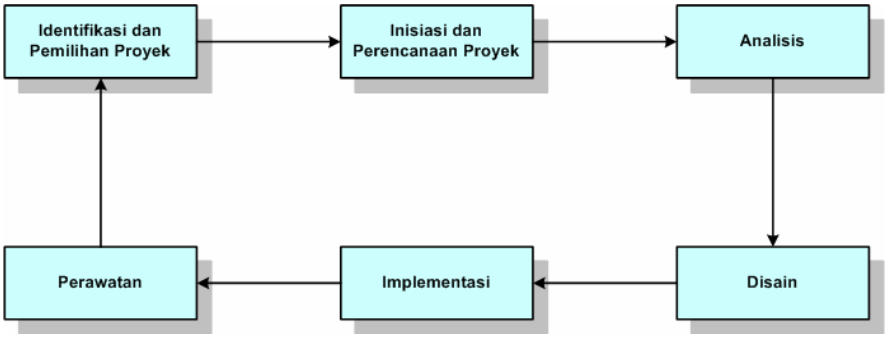
dari bertarung, hingga berjalan. Khusus proses berjalan, algoritma *path finding* adalah algoritma yang dapat dimanfaatkan untuk membantu aplikasi menemukan alur jalannya. *Game* dengan contoh menemukan jalan yaitu *game maze* (labirin), *game* perang, *game* sokoban dan *game* strategi. [20]

II.8 Algoritma Penerapan A* pada *Game*

Setiap permainan memiliki aturan main. Hal ini mempermudah upaya menghasilkan ruang pencarian dan memberikan kebebasan pada para peneliti dari bermacam-macam ambisi dan kompleksitas sifat serta kurangnya struktur permasalahan. Papan konfigurasi yang digunakan untuk memainkan permainan ini mudah direpresentasikan pada komputer dan tidak memerlukan bentuk yang kompleks. Permainan dapat menghasilkan sejumlah besar pencarian ruang. Hal ini cukup besar dan kompleks sehingga membutuhkan suatu teknik yang tangguh untuk menentukan alternatif pengeksplorasian ruang permasalahan. Teknik ini dikenal dengan nama *heuristic* dan merupakan area utama dari penelitian tentang AI. Banyak hal yang biasanya dikenal sebagai kecerdasan tampaknya berada dalam *heuristic* yang digunakan oleh manusia untuk menyelesaikan permasalahannya. [18]

II.9 Metode Pengembangan Perangkat Lunak

Pada rekayasa perangkat lunak, banyak model yang telah dikembangkan untuk membantu proses pengembangan perangkat lunak. Model-model ini pada umumnya mengacu pada model proses pengembangan sistem yang disebut *System Development Life Cycle* (SDLC) seperti terlihat pada gambar II.14.



Gambar II.14. *System Development Life Cycle* (SDLC)

Setiap model yang dikembangkan mempunyai karakteristik sendiri-sendiri. Namun secara umum ada persamaan dari model-model ini, yaitu:

a. Kebutuhan terhadap definisi masalah yang jelas. *Input* utama dari setiap model pengembangan perangkat lunak adalah pendefinisian masalah yang jelas. Semakin jelas akan semakin baik karena akan memudahkan dalam penyelesaian masalah.

b. Tahapan-tahapan pengembangan yang teratur. Meskipun model-model pengembangan perangkat lunak memiliki pola yang berbeda-beda, biasanya model-model tersebut mengikuti pola umum *analysis – design – coding – testing - maintenance.*

c. *Stakeholder* berperan sangat penting dalam keseluruhan tahapan pengembangan. *Stakeholder* dalam rekayasa perangkat lunak dapat berupa pengguna, pemilik, pengembang, pemrogram dan orang-orang yang terlibat dalam rekayasa perangkat lunak tersebut.

d. Dokumentasi merupakan bagian penting dari pengembangan perangkat lunak. Masing-masing tahapan dalam model biasanya menghasilkan sejumlah

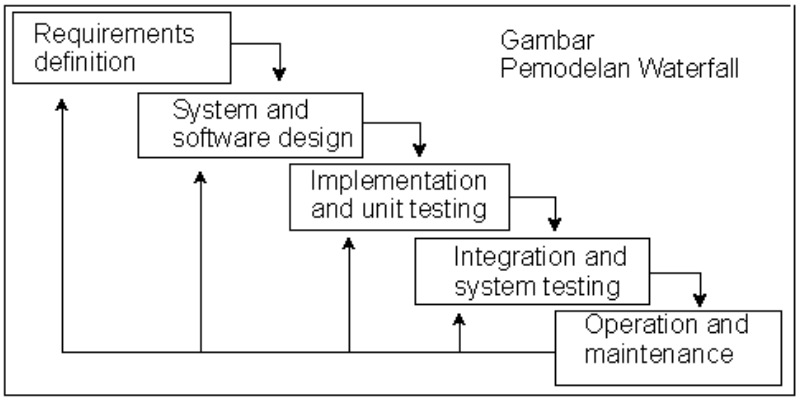
tulisan, diagram, gambar atau bentuk-bentuk lain yang harus didokumentasi dan merupakan bagian tak terpisahkan dari perangkat lunak yang dihasilkan.

e. Keluaran dari proses pengembangan perangkat lunak harus bernilai ekonomis. Nilai dari sebuah perangkat lunak sebenarnya agak susah dirupiahkan. Namun efek dari penggunaan perangkat lunak yang telah dikembangkan haruslah memberi nilai tambah bagi organisasi. Hal ini dapat peningkatan keuntungan organisasi, peningkatan "*image*" organisasi dan lain-lain.

Ada banyak model pengembangan perangkat lunak, antara lain *The Waterfall Model*, *Joint Application Development* (JAD), *Information Engineering* (IE), *Rapid Application Development* (RAD) termasuk di dalamnya *Prototyping*, *Unified Process* (UP), *Structural Analysis and Design* (SAD) dan *Framework for the Application of System Thinking* (FAST). Salah satu yang akan dijelaskan yaitu tentang model *waterfall*.[17]

Model *waterfall* sebenarnya adalah "*linear sequential model*". Model ini sering disebut dengan "*classic life cycle*" atau model *waterfall*. Model ini adalah model yang muncul pertama kali yaitu sekitar tahun 1970 sehingga sering dianggap kuno, tetapi merupakan model yang paling banyak dipakai didalam *Software Engineering* (SE). Model ini melakukan pendekatan secara sistematis dan urut mulai dari level kebutuhan sistem lalu menuju ke tahap analisis, desain, *coding*, *testing / verification*, dan *maintenance*. Disebut dengan *waterfall* karena tahap demi tahap yang dilalui harus menunggu selesainya tahap sebelumnya dan

berjalan berurutan. Sebagai contoh tahap desain harus menunggu selesainya tahap sebelumnya yaitu tahap *requirement*.



Gambar II.15. Model *waterfall*

Gambar II.15 adalah tahapan umum dari proses model *waterfall*, akan tetapi *Roger S. Pressman* memecah model ini menjadi enam tahapan meskipun secara garis besar sama dengan tahapan-tahapan model *waterfall* pada umumnya. Berikut adalah penjelasan dari tahap-tahap yang dilakukan di dalam model ini menurut *Pressman*:

1. Sistem / *information engineering and modeling*. Permodelan ini diawali dengan mencari kebutuhan dari keseluruhan sistem yang akan diaplikasikan ke dalam bentuk *software*. Hal ini sangat penting, mengingat *software* harus dapat berinteraksi dengan elemen-elemen yang lain seperti *hardware*, *database* dan sebagainya. Tahap ini sering disebut dengan *Project Definition*.
2. *Software Requirements Analysis*. Proses pencarian kebutuhan diintensifkan dan difokuskan pada *software*. Untuk mengetahui sifat dari program yang akan dibuat, maka para *software engineer* harus mengerti tentang domain informasi dari *software*, misalnya fungsi yang dibutuhkan, *user interface* dan

sebagainya. Dari dua aktivitas tersebut (pencarian kebutuhan sistem dan *softwar*e) harus didokumentasikan dan ditunjukkan kepada pelanggan.

3. *Design*, proses ini digunakan untuk mengubah kebutuhan-kebutuhan di atas menjadi representasi ke dalam bentuk "*blueprint*" *software* sebelum *coding* dimulai. Desain harus dapat mengimplementasikan kebutuhan yang telah disebutkan pada tahap sebelumnya. Seperti dua aktivitas sebelumnya, maka proses ini juga harus didokumentasikan sebagai konfigurasi dari *software*.
4. *Coding*, untuk dapat dimengerti oleh mesin, dalam hal ini adalah komputer, maka desain tadi harus diubah bentuknya menjadi bentuk yang dapat dimengerti oleh mesin, yaitu ke dalam bahasa pemrograman melalui proses *coding*. Tahap ini merupakan implementasi dari tahap *design* yang secara teknis nantinya dikerjakan oleh *programmer*.
5. *Testing / verification*, sesuatu yang dibuat haruslah diujicobakan. Demikian juga dengan *softwar*e. Semua fungsi-fungsi *software* harus diujicobakan, agar *software* bebas dari *error*, dan hasilnya harus benar-benar sesuai dengan kebutuhan yang sudah didefinisikan sebelumnya.
6. *Maintenance*, pemeliharaan suatu *software* diperlukan, termasuk di dalamnya adalah pengembangan, karena *software* yang dibuat tidak selamanya hanya seperti itu. Ketika dijalankan mungkin saja masih ada *errors* kecil yang tidak ditemukan sebelumnya, atau ada penambahan fitur-fitur yang belum ada pada *software* tersebut. Pengembangan diperlukan ketika adanya perubahan dari eksternal perusahaan seperti ketika ada pergantian sistem operasi, atau perangkat lainnya. [11]

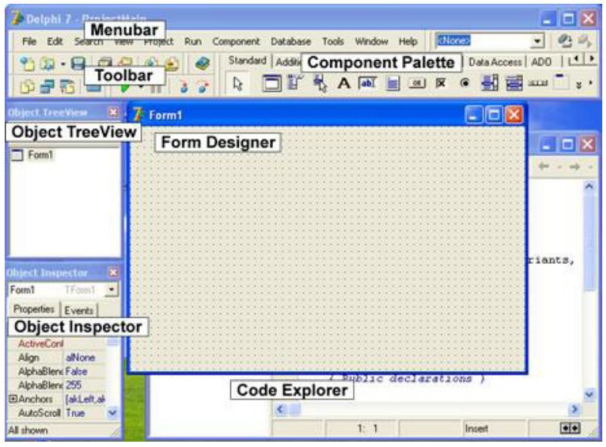
II.10 Borland Delphi

Borland Delphi adalah sebuah alat pengembangan aplikasi-aplikasi untuk sistem operasi *Microsoft Windows*. Delphi sangat berguna dan mudah digunakan untuk membuat suatu program berbasis GUI (*Graphical user interface*) atau *console* (mode teks).

Semua *user interface* seperti *form*, tombol (*button*), dan objek *list-list* telah disertakan dalam Delphi dalam bentuk komponen atau *control*. Pengembang dapat dengan mudah menempatkan komponen-komponen tersebut ke dalam *form*. Pengembang dapat juga menempatkan *control ActiveX* pada *form* untuk membuat program-program khusus seperti *Browser Web* dalam waktu yang cepat. Delphi memungkinkan pengembang untuk merancang keseluruhan *interface* secara visual, dan dengan cepat dapat diimplementasikan sebuah kode perintah berbasis *event* (*event driven*) dengan mengklik mouse. Dengan IDE Delphi, pengembang perangkat lunak dapat membuat program *windows* dengan lebih cepat dan lebih mudah dari sebelumnya.

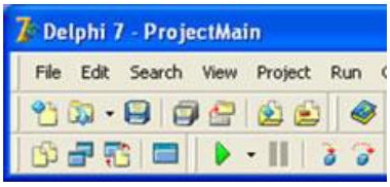
IDE adalah sebuah singkatan dari *Integrated Development Environment* yaitu sebuah lingkungan pengembangan yang terintegrasi, istilah IDE popular untuk menyebut *software* bahasa pemrograman dimana proses pengembangan programmnya mulai dari *coding*, *designing* dan *debugging* dilakukan pada satu *framework* atau pada satu aplikasi yang terintegrasi.

IDE delphi tersebut terbagi menjadi tujuh bagian utama, yaitu : *menubar*, *toolbar, component palette, form designer, code explorer, object treeview dan object inspector.*



Gambar II.16. IDE Delphi

Menubar dan *toolbar* merupakan dua bagian yang biasanya terdapat pada aplikasi-aplikasi *windows* lain. Fungsi dari *menubar* dan *toolbar* ini relatif hampir sama dengan aplikasi *windows* lain, melainkan hanya *menubar* yang biasa digunakan dalam pembuatan program.



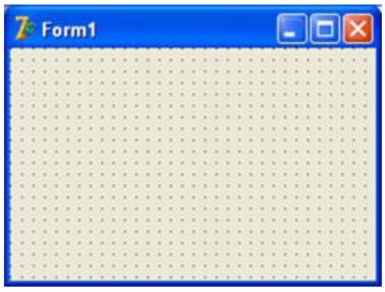
Gambar II.17. *Menu bar* dan *tool bar*

Component Palette berisi kumpulan VCL (*Visual Component Library*) yang berguna dalam desain aplikasi. VCL merupakan pustaka untuk komponen visual, dimana dalam *component palette* dilambangkan dengan ikon yang merepresentasikan komponen tersebut. Komponen-komponen VCL pada *component palette* dikelompokkan ke dalam *tab*, sesuai dengan fungsinya, dengan maksud untuk memudahkan *programmer* dalam memilih komponen yang diinginkannya. Disamping VCL ada juga CLX (*Component Library for Cross Platform*) dalam desain aplikasi pemrograman Delphi.



Gambar II.18. *Component pallete*

Setiap aplikasi biasanya memiliki jendela atau *background interface*, yang dalam bahasa pemrograman Delphi atau bahkan dalam bahasa pemrograman lain yang berbasis visual, biasa disebut dengan *form*. *Form Designer* berfungsi sebagai tempat untuk mendesain *form* untuk aplikasi yang akan dibuat, dan juga sebagai tempat untuk meletakkan komponen-komponen yang kita ambil dari *component palette*.

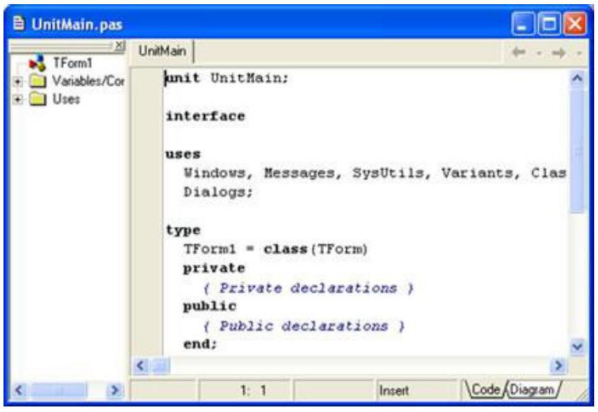


Gambar II.19. *Form designer*

Code Explorer merupakan area di mana kita menuliskan kode program, posisinya secara default terletak dibelakang *form*. Untuk menampilkan *code explorer* di depan form, bisa digunakan tombol F12 pada *keyboard*. Pada *code explorer* tersebut, akan terlihat kode-kode dalam bahasa pemrograman Delphi yang secara otomatis di'*generate'* oleh Delphi, hal ini jelas akan memudahkan atau mempercepat kita dalam menulis program. Pada *code explorer* Delphi, ada sebuah fitur yang disebut dengan *code completion*. Sesuai dengan namanya, *code completion* berfungsi melengkapi kode yang kita tulis dalam bentuk pilihan/*list*

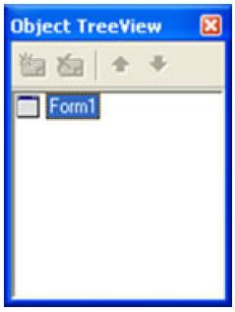
dari *code-code* yang bisa kita gunakan, hal ini akan sangat membantu apabila kita lupa terhadap kode tertentu.

Code completion ini secara otomatis akan muncul ketika anda menekan tombol titik pada *keyboard*, selain itu untuk menampilkan *code completion* ini, anda juga bisa melakukannya dengan menekan kombinasi tombol Ctrl + *space* secara bersamaan. Disamping *code completion*, *code explorer* pada Delphi juga dilengkapi dengan *hint* (*layer* berwarna kuning yang muncul ketika mouse didekatkan pada komponen tertentu). *Hint* ini, muncul ketika kita menggunakan *procedure* atau fungsi dan menekan tombol pada *keyboard*, pada *hint* tersebut akan muncul nama dan tipe data dari parameter yang digunakan atau informasi lain yang diperlukan.



Gambar II.20. *Code Explorer*

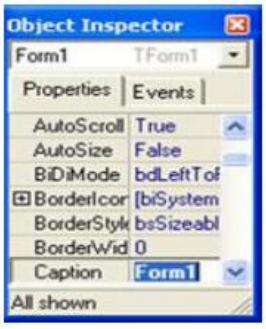
Object TreeView adalah bagian yang berisi daftar komponen yang digunakan dalam *form designer*. Dengan model tampilan bercabang, akan memudahkan kita dalam menunjuk komponen tertentu terutama jika aplikasi yang dibuat menggunakan banyak komponen.



Gambar II.21. *Object treeview*

Object Inspector adalah bagian yang digunakan untuk memanipulasi sifat atau karakteristik dan *event* dari komponen yang kita gunakan dalam *form designer*. Jendela *object inspector* terbagi menjadi dua bagian tab, yaitu *tab property* dan *tab event*. Tab *property* digunakan untuk memanipulasi *property* yang dimiliki oleh komponen tertentu, misalkan ukuran, warna dan *caption* dari komponen, sedangkan *tab event* digunakan untuk menangani pemasukan kode pada kejadian tertentu dari suatu komponen, misalnya : kejadian ketika komponen *button* (tombol) diklik atau onClick.

Sebagaimana yang telah dijelaskan di atas, salah satu fungsi dari *object inspector* ini adalah untuk mengatur *property* dari komponen, pengaturan tersebut tidak akan disimpan pada *file* unit, melainkan akan disimpan pada *file form* (*.dfm).[4]



Gambar II.22. *Object inspector*