BAB 2
TINJAUAN PUSTAKA

2.1 Kecerdasan Buatan

Kecerdasan buatan atau *Artificial Intelligence* (AI) merupakan ilmu dan rekayasa yang membuat mesin mempunyai intelligensi tertentu khususnya program komputer yang ‘cerdas’. *Artificial Intelligence* (AI) didefinisikan sebagai kecerdasan yang dimiliki oleh suatu entitas buatan, yang dapat berpikir seperti manusia, bertindak seperti manusia, berpikir rasional, atau bertindak rasional. Kecerdasan diciptakan dan dimasukkan ke dalam suatu mesin (komputer) agar dapat melakukan pekerjaan seperti yang dapat dilakukan manusia. Dengan semakin berkembangnya *hardware* dan *software*, saat ini berbagai produk AI telah banyak digunakan dan diaplikasikan kedalam kehidupan sehari-hari.

Implementasi dari *artificial intelligence* saat ini umum ditemui dalam bidang-bidang seperti berikut [6] :

a. *Computer Vision*

   Computer Vision merupakan implementasi *artificial intelligence* yang memungkinkan sebuah sistem komputer mengenali gambar sebagai inputnya. Contohnya adalah mengenali dan membaca tulisan yang ada didalam gambar.

b. *Fuzzy Logic*

   Fuzzy Logic merupakan implementasi *artificial intelligence* yang banyak terdapat pada alat-alat elektronik dan robotika. Dimana alat-alat elektronik atau robotika tersebut mampu berpikir dan bertingkah laku sebagaimana layaknya manusia.

c. *Game*

   Game merupakan implementasi *artificial intelligence* yang berguna untuk meniru cara berpikir seorang manusia dalam bermain *game*. Contohnya adalah program *Perfect Chessmate* yang mampu berpikir setara dengan seorang *grandmaster* catur.

d. *General Problem Solving*

General Problem Solving merupakan implementasi *artificial intelligence* yang berhubungan dengan pemecahan suatu masalah terhadap suatu situasi yang akan diselesaikan oleh komputer. Biasanya permasalahan tersebut akan diselesaikan secara *trial and error* sampai sebuah solusi dari sebuah masalah didapatkan. Contohnya adalah program *Eureka* yang dapat memecahkan *model linier programming*.

e. *Speech Recognition*

Speech Recognition merupakan implementasi *artificial intelligence* yang berguna untuk mengenali suara manusia dengan cara dicocokkan dengan acuan yang telah diprogramkan sebelumnya. Contohnya adalah suara dari user dapat diterjemahkan menjadi sebuah perintah bagi komputer.

f. *Expert System*

Expert System merupakan implementasi *artificial intelligence* yang berguna untuk meniru cara berfikir dan penalaran seorang ahli dalam mengambil keputusan berdasarkan situasi yang ada. Dengan *expert system* seorang user dapat melakukan konsultasi kepada komputer, seolah-olah user tersebut berkonsultasi langsung kepada seorang ahli. Contohnya adalah program aplikasi yang mampu meniru seorang ahli medis dalam mendeteksi demam berdarah berdasarkan keluhan-keluhan pasiennya.

Teknik pemecahan masalah yang ada didalam AI dibagi menjadi empat, yaitu [7] :

a. *Searching*

Searching merupakan teknik menyelesaikan masalah dengan cara merepresentasikan masalah ke dalam *state* dan ruang masalah serta menggunakan strategi pencarian untuk menemukan solusi.

b. *Reasoning*

Reasoning merupakan teknik menyelesaikan masalah dengan cara merepresentasikan masalah ke dalam basis pengetahuan *(knowledge base*) menggunakan *logic* atau bahasa formal.

c. *Planning*

Planning merupakan teknik menyelesaikan masalah dengan cara memecah masalah kedalam sub-sub masalah dan menyelesaikan sub-sub masalah satu demi satu.

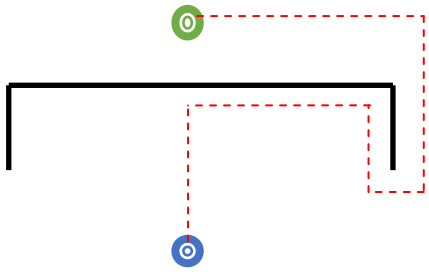
d. *Learning*

Learning merupakan teknik menyelesaikan masalah dengan mengotomatisasi mesin untuk menemukan aturan yang diharapkan bisa berlaku umum untuk data-data yang belum pernah diketahui.

Pemilihan teknik yang digunakan dalam penyelesaian masalah tergantung dari karakteristik permasalahan yang akan diselesaikan. Misalnya implementasi kecerdasan buatan pada *game* yang sering diterapkan yaitu pada *pathfinding*. Penggunaan *pathfinding* paling sering pada *game* adalah untuk mempengaruhi pergerakan dan pengambilan keputusan yang diterapkan pada *non-player character*. Sedangkan teknik untuk pemecahan masalah *pathfinding* dalam *game* menggunakan teknik pencarian atau *searching*.

2.2 Pencarian Jalur / Rute

Pencarian jalur / rute (*pathfinding*) adalah salah satu bidang penerapan yang sering ditangani oleh kecerdasan buatan khususnya dengan menggunakan algoritma pencarian [3]. 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 2.1.



Gambar 2.1 Penentuan rute tanpa *pathfinding*

Keterangan :

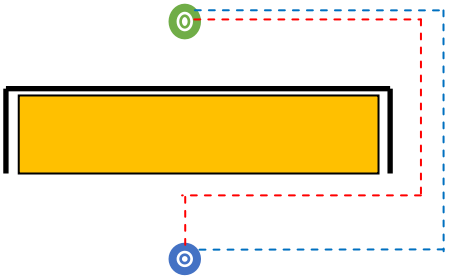
= rintangan / penghalang

= *start*

= *goal*

= jalan yang dilalui

Pada Gambar 2.1, dari *start* menuju *goal*, tanpa algoritma *pathfinding*, unit hanya akan memeriksa lingkungan sekitarnya saja (dilambangkan dengan daerah di dalam bulat biru). Unit tersebut akan maju terus ke atas untuk mencapai tujuan, baru setelah mendekati adanya halangan, lalu berjalan memutar untuk menghindarinya.



Gambar 2.2 Penentuan rute dengan *pathfinding*

Keterangan :

= rintangan / penghalang

= *start*

= *goal*

= jalan yang dilalui

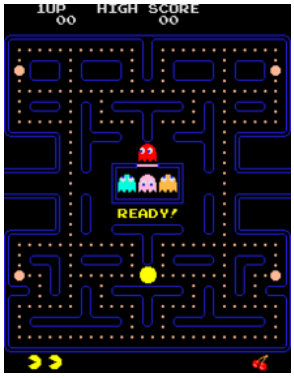
= daerah penghalang

Sebaliknya, penentuan *rule* dengan algoritma *pathfinding* pada Gambar 2.2 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. Oleh karena itu peran algoritma *pathfinding* sangat berguna untuk memecahkan berbagai permasalahan dalam penentuan rute.

2.3 *Game* Labirin

Menurut Heni dan Beckerman (2006,p1), *game* adalah bagian pokok dalam warisan manusia, dimainkan di berbagai kebudayaan dan dinikmati oleh semua golongan umur. *Game* seperti catur telah ada dari zaman dahulu. Konsep *game* komputer sekarang mengikuti ide dari *board game*, cerita meniru dari film dan mensimulasikan kehidupan nyata. Menurut Andang Ismail terdapat dua pengertian *game* (permainan). Pertama, *game* (permainan) adalah sebuah aktifitas bermain yang murni mencari kesenangan tanpa mencari menang atau kalah. Kedua, permainan diartikan sebagai aktifitas bermain yang dilakukan dalam rangka mencari kesenangan dan kepuasan, namun ditandai pencarian menang – kalah.

Game Labirin (Maze *Game*) adalah permainan video deskripsi pertama kali digunakan oleh jurnalis selama tahun 1980-an untuk menggambarkan setiap permainan seluruh lapangan adalah labirin dengan berbagai dinding rintangan di dalamnya. Selama proses penentuan jalur tersebut, jika menemuai jalan buntu maka akan dilakukan proses *backtrack* sampai kembali menemukan jalur yang tepat untuk mencapai tujuan. Dalam *game* labirin dibutuhkan tindakan yang cepat pemain untuk menghindar dari musuh dan menavigasi labirin pada waktu yang terbatas [12]. Saat ini *game* labirin yang berkembang dapat dimainkan dalam bentuk 2D dan 3D. Contoh permainan *genre maze* adalah labirin *Rogue* dimainkan di terminal teks 28x40 pada sistem *Unix* Versi 7 tahun 1980, setiap tingkat terdiri dari 9 kamar, dengan tema penjara. Selain labirin *Rogue*, permainan paling terkenal dari *genre* ini adalah *Pac-Man* dan *Snake*.



Gambar 2.3 *Game Pac-man* [1]

Game pac-man adalah salah satu contoh *game* yang memiliki *gendre maze*. *Game pac-man* memiliki *game play* yang sederhana yaitu pemain harus menggerakkan karakter *pacman* untuk menghabiskan titik-titik yang ada dalam labirin sampai tak tersisa satu pun dan menghindari beberapa musuh yang mengejar karakter *pacman.*

2.4 Metode-metode Pencarian

Salah satu teknik untuk menyelesaikan masalah penggunaan *pathfinding* dalam *game* yaitu menggunakan teknik pencarian atau *searching*. Metode pencarian secara umum dibagi menjadi 2 kelompok, yaitu *blind* atau *uninformed search* (pencarian buta atau tidak dibekali informasi) dan *heuristic* atau *informed search* (pencarian dengan panduan atau berbekal informasi). Setiap metode memiliki karakteristik yang saling membedakan satu sama lain dengan kelebihan dan kekurangan masing-masing, yang bisa diukur performansinya dengan empat kriteria berikut [7] :

a. *Completeness,* yaitu apakah metode pencarian tersebut menjamin ditemukannya solusi jika solusinya memang ada.
b. *Time complexity,* yaitu berapa lama waktu yang diperlukan selama proses pencarian.
c. *Space complexity,* yaitu berapa banyak memori yang diperlukan selama proses pencarian.
d. *Optimality,* yaitu apakah metode pencarian tersebut menjamin ditemukannya solusi yang terbaik jika ada beberapa solusi yang berbeda.

Metode pencarian sangat penting untuk menyelesaikan permasalahan karena setiap *state* atau keadaan menggambarkan langkah-langkah untuk menyelesaikan permasalahan. Dalam sebuah *game*, metode pencarian akan menentukan apa yang harus dilakuan dimana setiap *state* menggambarkan kemungkinan posisi pada saat tertentu.

2.4.1 Pencarian Buta (*Blind Search*)

Dinamakan percarian buta (*Blind Search*) / *Uninformed Search*, karena pada metode ini tidak dibekali informasi awal untuk proses pencarian, dan proses pencarian dilakukan berdasarkan aturan tertentu. Informasi yang ada hanyalah definisi *goal state* itu sendiri sehingga algoritma dapat mengenali *goal state*. Apabila tidak ada informasi maka pencarian buta dalam kerjanya akan memeriksa *node-node* secara tidak terarah dan kurang efisien untuk kebanyakan kasus karena banyaknya *node* yang dikembangkan.

Metode yang tergolong *blind search* diantaranya : *Breadth First Search* (BFS), *Uniform Cost-Search* (UCS), *Depth First Search* (DFS), *Depth Limited Search* (DLS), *Iterative Deepening Search* (IDS), dan *Bidirectional Search* (BDS). Metode *blind search* biasanya digunakan pada pencarian yang memiliki ruang masalah yang kecil. Jika ruang masalahnya sudah besar sebaiknya digunakan metode *informed search* agar pada saat melakukan pencarian dapat lebih terarah dengan informasi-informasi yang diberikan [7].

2.4.2 Pencarian Terbimbing (*Heuristic Search*)

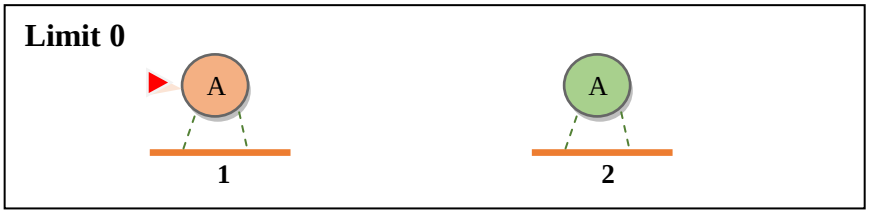
Metode heuristik merupakan metode pencarian yang mempunya informasi tentang biaya untuk mencapai *goal state* dari *current state.* Informasi ini adalah suatu fungsi yang disebut dengan fungsi heuristik. Metode heuristik menggunakan pendekatan nilai perkiraan untuk membantu algoritma dalam melakukan pencarian dan tidak harus *valid* setiap waktu. Metode heuristik dewasa ini banyak berkembang. Telah banyak algoritma yang menerapkan metode heuristik dalam melakukan proses pencarian, metode yang umum dikenal adalah A*. Beberapa contoh algoritma pencarian lain yang menggunakan metode *heuristic search*

adalah *Best First Search, Generate and Test, Hill Climbing,* dan *Greedy Search* [7].

2.5 Algoritma *Iterative Deepening A** (IDA*)

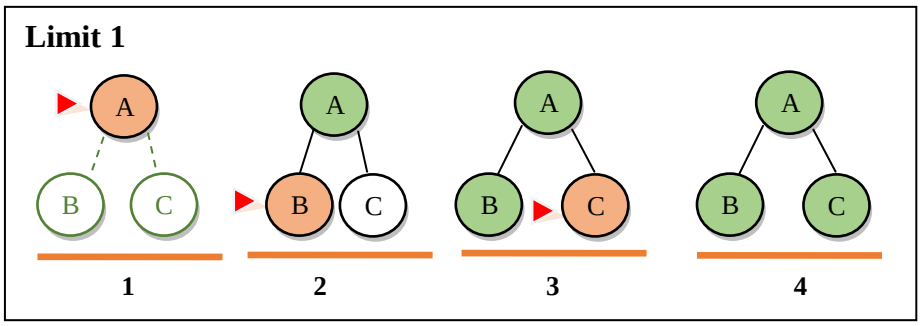
Algoritma IDA* (IDA *star*/ IDA bintang) merupakan varian dari algoritma A* [2]. Prinsip kerja algoritma IDA* menggunakan kelebihan dari algoritma A* dan algoritma *Iterative Deepening Search* (IDS). Kelebihan algoritma A* yaitu biaya yang diperhitungkan didapat dari biaya sebenarnya ditambah dengan biaya perkiraan. Sedangkan kelebihan algoritma IDS menggabungkan kelebihan *Breadth First Search* (*complete* dan *optimal*) dan kelebihan *Depth First Search* (membutuhkan sedikit memori) [9]. IDS merupakan algoritma *Blind Search* yang melakukan pencarian secara iteratif menggunakan penelusuran *Depth Limited Search* (DLS) dimulai dengan batasan limit 0 [8]. Jika belum ditemukan solusi, maka dilakukan iterasi ke-2 dengan batasan limit 1. Demikian seterusnya sampai ditemukan solusi yang paling dangkal. Berikut adalah ilustrasi penelusuran IDS.

1. Misalkan simpul awal adalah A dan simpul tujuan adalah F, maka penelusuran awal pencarian IDS dimulai dari simpul A pada limit 0. Pada Gambar 2.4, lingkaran dengan tanda panah merah menunjukkan simpul yang sedang diperiksa. Sedangkan lingkaran berwarna hijau menunjukkan bahwa pencarian simpul A sudah selesai dilakukan.



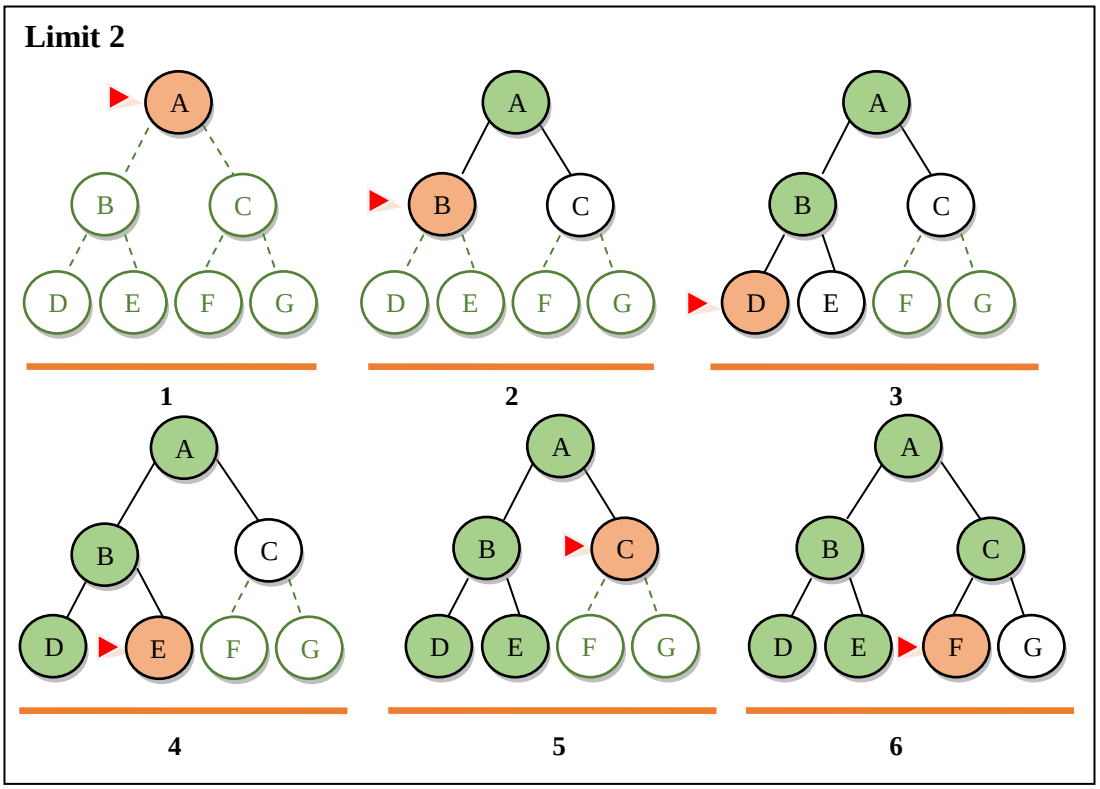
Gambar 2.4 Penelusuran *Iterative Deepening Search* (IDS) Limit 0

2. Namun karena simpul A bukan simpul tujuan, maka ulangi pencarian dari status awal. Sekarang pencarian dimulai dari limit ke-1 dengan mengekspansi suksesor dari simpul A kemudian lakukan pencarian dari simpul anak kiri ke simpul anak kanan. Lihat Gambar 2.5.



Gambar 2.5 Penelusuran *Iterative Deepening Search* (IDS) Limit 1

3. Karena pada limit 1 belum juga menemukan simpul tujuan, maka pencarian diulang kembali dari status awal secara iteratif dan sekarang dimulai dari limit ke-2 dengan simpul A diekspansi, simpul B juga diekspansi dan lakukan penelusuran ke suksesor D dan E, sampai akhirnya simpul C juga diekspansi. Setelah simpul C diekspansi, akhirnya simpul tujuan yaitu simpul F ditemukan. Oleh karena itu pencarian berhenti setelah simpul tujuan ditemukan. Lihat Gambar 2.6.



Gambar 2.6 Penelusuran *Iterative Deepening Search* (IDS) Limit 2

Berdasarkan langkah-langkah pencarian IDS, batasan limit (kedalaman) pada setiap iterasi pencarian IDS inilah yang diterapkan pada algoritma IDA* dengan batasan yang berupa *f-limit*. Jika jalur solusi belum ditemukan, setiap iterasi mengembalikan nilai *f-limit* baru yang akan digunakan sebagai batasan pencarian untuk iterasi berikutnya. Nilai *f-limit* itu sendiri diperoleh dari gabungan biaya sebenarnya dan biaya perkiraan, yang secara matematika dinyatakan sebagai :

$$f(n_i) = g(n) + c(n,n_i) + h(n_i) \quad \ldots (2.1)$$

Dengan :

n = simpul saat ini

n_i = suksesor dari simpul n

$g(n)$ = biaya (*cost*) dari n_a (simpul awal) ke simpul n sepanjang jalur pencarian

$c(n,n_i)$ = biaya dari simpul n ke simpul n_i

$h(n_i)$ = perkiraan biaya minimum dari simpul n_i ke simpul tujuan

$f(n_i)$ = total biaya dari n_a (simpul awal) ke n_t (simpul tujuan) melalui simpul n

Algoritma IDA* adalah algoritma pencarian yang menggunakan fungsi heuristik, yaitu fungsi yang menghitung biaya perkiraan dari simpul tertentu menuju simpul tujuan. IDA* bersifat *complete* (pasti menemukan jalur dari simpul asal ke simpul tujuan) dan *optimal* (jalur yang ditemukan adalah jalur terbaik dengan biaya minimun) asalkan fungsi heuristiknya *admissible* [7]. Salah satu fungsi heuristik *admissible* yang sering dikenal yaitu *Manhattan Distance*. Perhitungan nilai heuristik untuk simpul ke-*n* menggunakan *Manhattan Distance* adalah sebagai berikut :

$$h(n_i) = (\text{abs}|\text{currentX} - \text{targetX}| + \text{abs}|\text{currentY} - \text{targetY}|) \quad \ldots (2.2)$$

Dengan :

$h(n_i)$ = perkiraan biaya minimum dari simpul n ke simpul tujuan

currentX = posisi X simpul awal

currentY = posisi Y simpul awal

targetX = posisi X simpul tujuan

targetY = posisi Y simpul tujuan

Adapun langkah-langkah manual pencarian IDA* adalah sebagai berikut :

1. Tentukan simpul awal (n_a) dan simpul tujuan (n_t).
2. Hitung nilai $f(n_a)$.
3. Jadikan nilai $f(n_a)$ sebagai *f-limit*.
4. Inisialisasi *next-f* = ∞.
5. Ekspansi simpul *n* dengan lakukan perulangan berikut ini untuk setiap suksesor n_i dari simpul *n* :

   5a. Hitung nilai $f(n_i)$

   5b. Apakah $f(n_i)$ adalah simpul tujuan dan $f(n_i) \leq$ *f-limit* ?

      i. Iya, maka pencarian berhenti.

      ii. Tidak, lakukan langkah 5c.

   5c. Apakah $f(n_i) \leq$ *f-limit* ?

      i. Iya, lakukan langkah 5 dari simpul saat ini.

      ii. Tidak, lakukan langkah 5d.

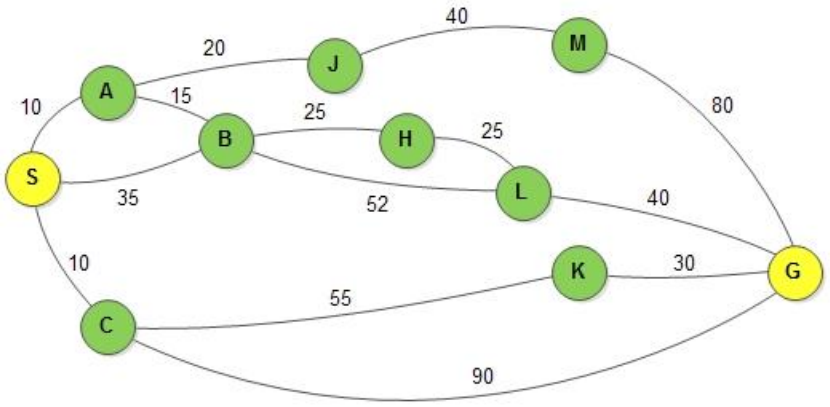
   5d. Apakah $f(n_i) <$ *next-f* ?

      i. Iya, set nilai $f(n_i)$ sebagai *next-f*.

      ii. Tidak, nilai *next-f* tetap.
6. Setelah perulangan selesai, set nilai *next-f* sebagai nilai *f-limit* baru.
7. Inisialisasi kembali nilai *next-f* menjadi ∞.
8. Ulangi pencarian dari status awal (*restart*) mulai dari langkah 5-8 dengan menggunakan nilai batasan *f-limit* baru. Lakukan sampai simpul tujuan ditemukan.

Sedangkan untuk kompleksitas ruang dan waktu algoritma IDA* sama dengan kompleksitas ruang dan waktu pada algoritma IDS yaitu kompleksitas ruang sebesar O(db) dan kompleksitas waktu sebesar $O(b^d)$, dimana b adalah faktor percabangan dan d adalah kedalaman pencarian [8]. Ini artinya karena dilakukan secara iteratif, algoritma IDA* membangkitkan simpul-simpul yang sama secara berulang-ulang, sehingga penghematan memori harus dibayar dengan kompleksitas waktu tinggi.

Untuk memperkuat pemahaman algoritma IDA*, perhatikan Gambar 2.7. Gambar 2.7 adalah contoh kasus pencarian rute terpendek dari kota *S* (*initial state*) ke kota G (*goal state*). Terdapat 10 kota yang dinyatakan oleh simpul-simpul dalam suatu *graph* dua arah. Setiap angka pada busur menyatakan jarak sebenarnya antara satu kota dengan kota lainnya. Nilai $h(n_i)$ adalah fungsi heuristik yaitu jarak garis lurus dari simpul *n* ke kota G dalam satuan kilometer. Pada tahap awal dari setiap langkah, simpul S akan dibangkitkan, kemudian dibangkitkan suksesor-suksesornya satu per satu sampai *f-limit* tertentu. Dan pada akhirnya, semua simpul dihapus dari memori.



Jarak Galis Lurus Kota n ke Kota G										
n	S	A	B	C	G	H	J	K	L	M
h(n)	80	74	85	80	0	40	100	30	20	70

Gambar 2.7 Masalah Pencarian Rute Terpendek

Penyelesaikan masalah rute terpendek menggunakan algoritma IDA* adalah sebagai berikut.

1. Tentukan simpul awal dan simpul tujuan.

   Simpul awal (n_a) = *S*

   Simpul tujuan (n_t) = *G*

2. Hitung $f(n_a)$ simpul awal.

   $f(S) = g(S) + c(S,S) + h(S)$

   $= 0 + 0 + 80$

   $= 80$

3. Jadikan nilai $f(n_a)$ sebagai *f-limit*

   f-limit = 80

4. Inisialisasi *next-f* = ∞.

5. Lakukan perulangan untuk setiap suksesor n_i dari simpul *n* (kiri ke kanan). Suksesor adalah simpul anak yang akan dikunjungi berikutnya. Simpul *S* mempunyai 3 simpul suksesor, maka lakukan perulangan sebanyak 3 kali. Lihat Gambar 2.8.

   <u>suksesor 1 dari simpul *S*</u>

   a. $n_i = A$

   b. Hitung $f(n_i)$ dari simpul suksesor.

      $f(A) = g(S) + c(S,A) + h(A)$

      $= 0 + 10 + 74 = 84$

   c. Karena $f(A) > f\text{-}limit$ dan $f(A) < next\text{-}f$, maka $f(A)$ dijadikan sebagai *next-f*.

      $f(A) < \infty$, maka $next\text{-}f = f(A) = 84$

   <u>suksesor 2 dari simpul *S*</u>

   a. $n_i = B$

   b. Hitung $f(n_i)$ dari simpul suksesor.

      $f(B) = g(S) + c(S,B) + h(B)$

      $= 0 + 35 + 85 = 120$

   c. Karena $f(B) > f\text{-}limit$, tetapi $f(B) > next\text{-}f$, maka nilai *next-f* tetap.

      $next\text{-}f = 84$

   <u>suksesor 3 dari simpul *S*</u>

   a. $n_i = C$

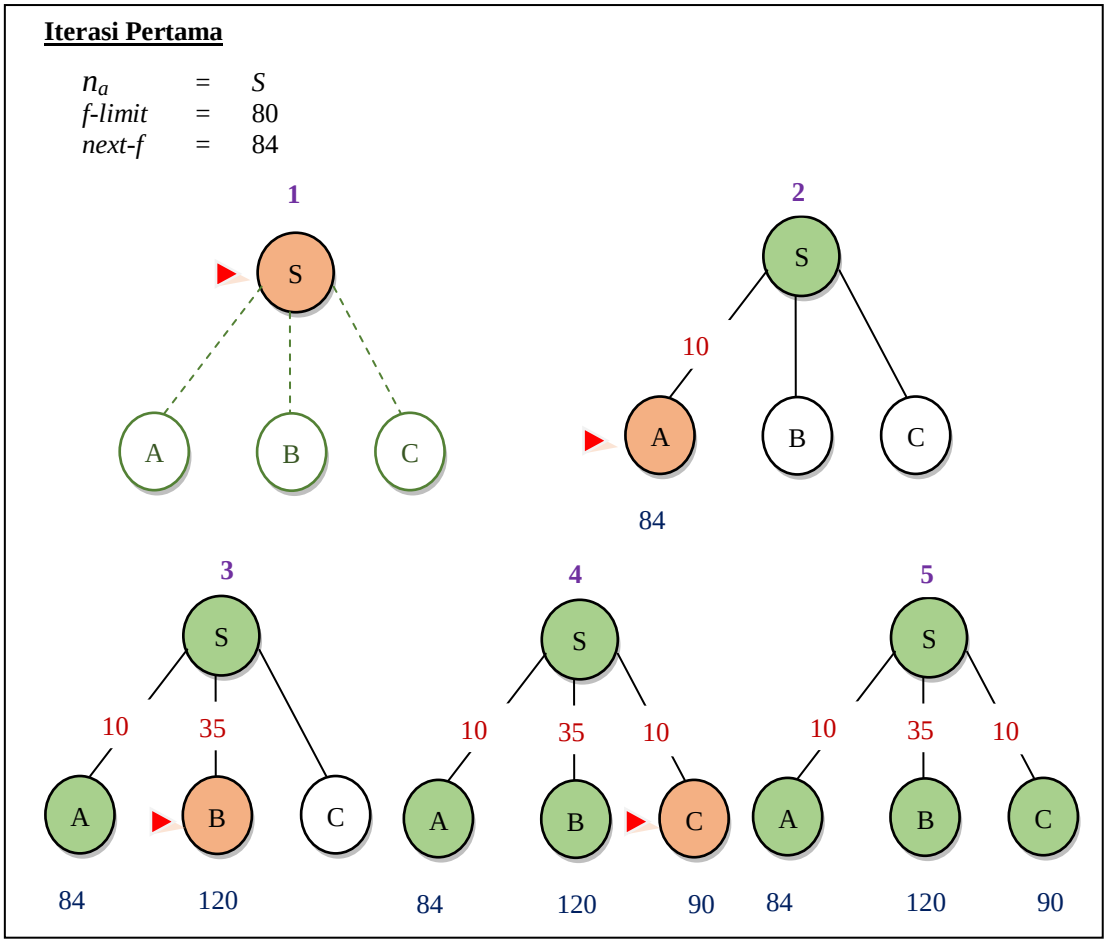
   b. Hitung $f(n_i)$ dari simpul suksesor.

      $f(C) = g(S) + c(S,C) + h(C)$

      $= 0 + 10 + 80 = 90$

   c. Karena $f(C) > f\text{-}limit$, tetapi $f(C) > next\text{-}f$, maka nilai *next-f* tetap.

      $next\text{-}f = 84$



Gambar 2.8 Langkah 5 Penyelesaian IDA*

6. Setelah semua suksesor diekspansi, set *f-limit* = *next-f* dan *next-f* = ∞.
   f-limit = 84 ; *next-f* = ∞.
7. Karena simpul tujuan belum ditemukan maka ulangi pencarian dari status awal. Lakukan perulangan seperti langkah 5 untuk mencari suksesor n_i dari simpul *S* dengan *f-limit* sekarang yaitu 84. Simpul *S* mempunyai 3 simpul suksesor, maka lakukan perulangan sebanyak 3 kali.
   suksesor 1 dari simpul S
   a. $n_i = A$, $f(A) = 84$
   b. Karena $f(A) \leq$ *f-limit*, lakukan perulangan untuk setiap suksesor n_i dari simpul *A*. Simpul *A* mempunyai 2 simpul suksesor, maka lakukan perulangan sebanyak 2 kali. Lihat Gambar 2.9.
      suksesor 1 dari simpul *A*
      a. $n_i = J$

b. Hitung $f(n_i)$ dari simpul suksesor.

$f(J) = g(A) + c(A,J) + h(J)$

$= 10 + 20 + 100 = 130$

c. Karena $f(J) > f\text{-}limit$ dan $f(J) < next\text{-}f$, maka $f(J)$ dijadikan sebagai *next-f*.

$f(J) < \infty$, maka $next\text{-}f = f(J) = 130$

suksesor 2 dari simpul *A*

a. $n_i = B$ (melalui simpul *A*)

b. Hitung $f(n_i)$ dari simpul suksesor.

$f(B) = g(A) + c(A,B) + h(B)$

$= 10 + 15 + 85 = 110$

c. Karena $f(B) > f\text{-}limit$ dan $f(n_i) < next\text{-}f$, maka $f(B)$ dijadikan sebagai *next-f*.

$f(B) < 130$, maka $next\text{-}f = f(B) = 110$

Hasil ekspansi simpul *A* belum menemukan simpul tujuan, oleh karena itu pencarian dilanjutkan arah ke simpul kanan yaitu simpul *B* dan *C*.

suksesor 2 dari simpul *S*

a. $n_i = B$

b. Hitung $f(n_i)$ dari simpul suksesor.

$f(B) = g(S) + c(S,B) + h(B)$

$= 0 + 35 + 85 = 120$

c. Karena $f(B) > f\text{-}limit$, tetapi $f(B) > next\text{-}f$, maka nilai *next-f* tetap.

$next\text{-}f = 110$

suksesor 3 dari simpul *S*

a. $n_i = C$

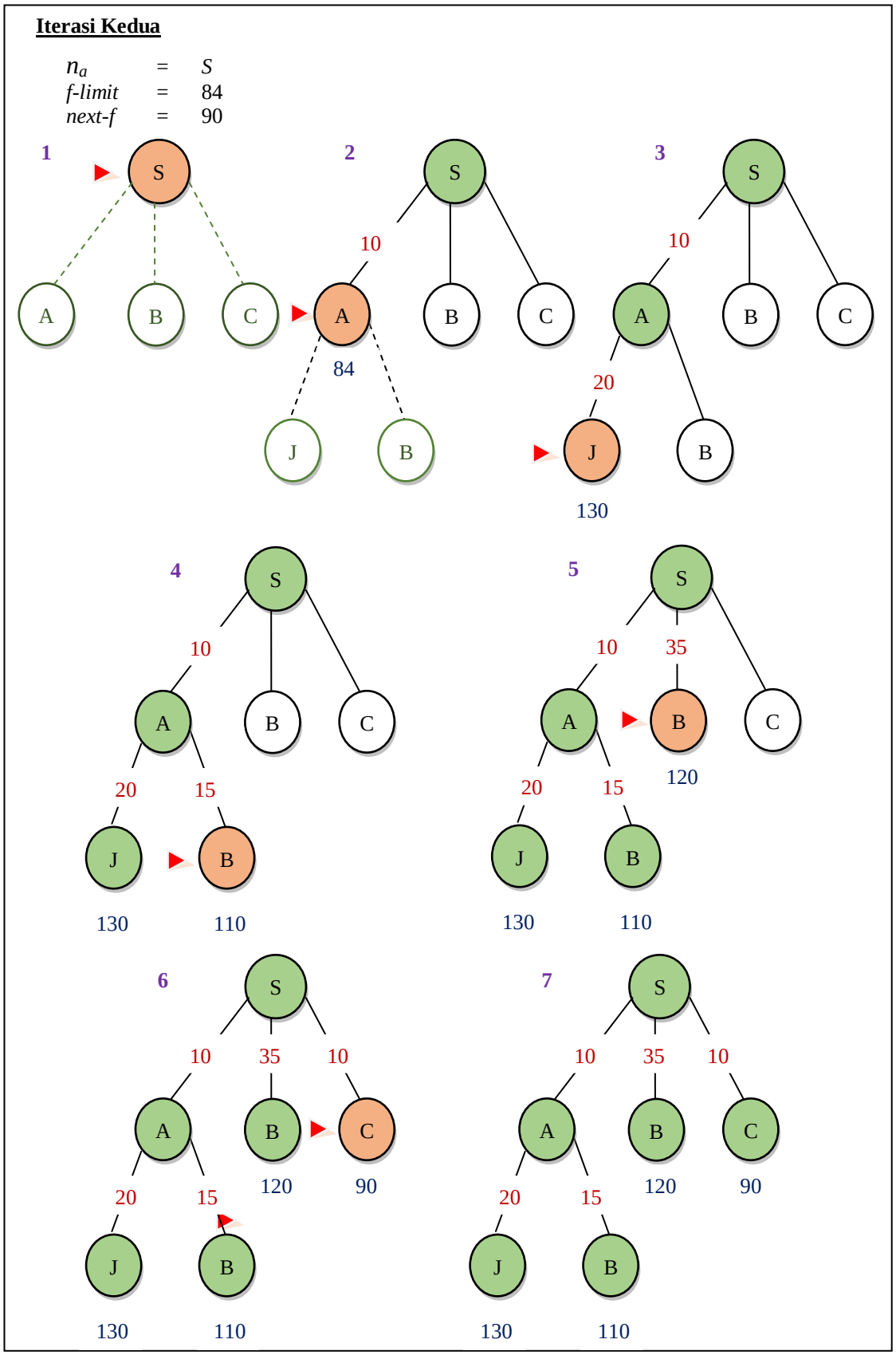
b. Hitung $f(n_i)$ dari simpul suksesor.

$f(C) = g(S) + c(S,C) + h(C)$

$= 0 + 10 + 80 = 90$

c. Karena $f(C) > f\text{-}limit$, tetapi $f(C) < next\text{-}f$, maka $f(C)$ dijadikan sebagai *next-f*.

$f(C) < 110$, maka $next\text{-}f = f(C) = 90$



Gambar 2.9 Langkah 7 Penyelesaian IDA*

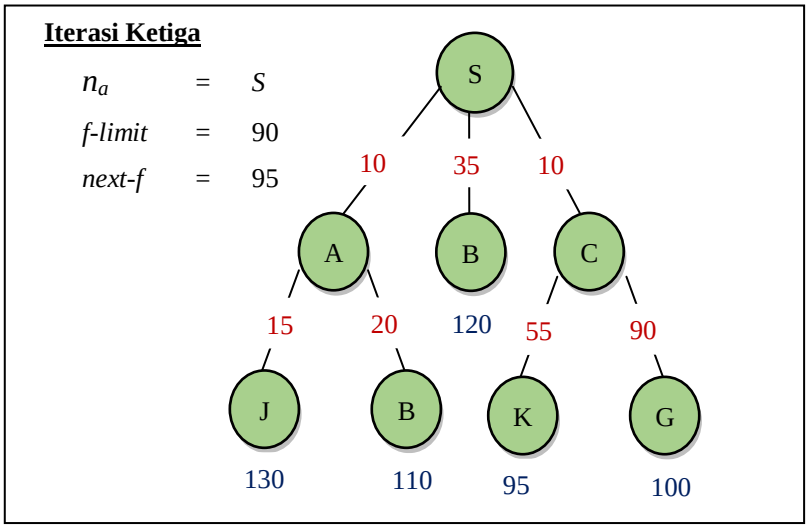
8. Setelah semua suksesor diekspansi, set *f-limit* = *next-f* dan *next-f* = ∞.
   f-limit = 90 ; *next-f* = ∞.
9. Karena simpul tujuan belum ditemukan, ulangi pencarian dari status awal. Lakukan perulangan seperti langkah 7 untuk mencari suksesor n_i dari simpul *S* dengan *f-limit* sekarang yaitu 90. Pada suksesor ke-3 dari simpul *S* yaitu simpul *C* diperoleh nilai $f(C) \leq$ *f-limit* maka lakukan perulangan untuk setiap suksesor n_i dari simpul *C*. Simpul *C* mempunyai 2 simpul suksesor, maka lakukan perulangan sebanyak 2 kali. Lihat Gambar 2.10.

   <u>suksesor 1 dari simpul *C*</u>

   a. $n_i = K$
   b. Hitung $f(n_i)$ dari simpul suksesor.
      $f(K) = g(C) + c(C,K) + h(K)$
      $= 10 + 55 + 30 = 95$
   c. Karena $f(K) >$ *f-limit* dan $f(K) <$ *next-f*, maka $f(K)$ dijadikan sebagai *next-f*.
      $f(K) < \infty$, maka *next-f* = $f(K) = 95$

   <u>suksesor 2 dari simpul *C*</u>

   a. $n_i = G$
   b. Hitung $f(n_i)$ dari simpul suksesor.
      $f(G) = g(C) + c(C,G) + h(G)$
      $= 10 + 90 + 0 = 100$
   c. Simpul *G* merupakan simpul tujuan, tetapi nilai $f(G) >$ *f-limit* dan $f(G) >$ *next-f*, maka nilai *next-f* tetap.
      next-f = 95



Gambar 2.10 Langkah 9 Penyelesaian IDA*

10. Setelah perulangan selesai, set *f-limit* = *next-f* dan *next-f* = ∞.
    f-limit = 95 ; *next-f* = ∞.
11. Karena simpul tujuan belum ditemukan, ulangi pencarian dari status awal. Lakukan perulangan seperti langkah 9 untuk mencari suksesor n_i dari simpul *S* dengan *f-limit* sekarang yaitu 95. Pada suksesor pertama dari simpul *C* yaitu simpul *K* diperoleh nilai $f(K) \leq$ *f-limit* maka lakukan perulangan untuk setiap suksesor n_i dari simpul *n*. Simpul *K* mempunyai 1 simpul suksesor. Lihat Gambar 2.11.

    suksesor 1 dari simpul C

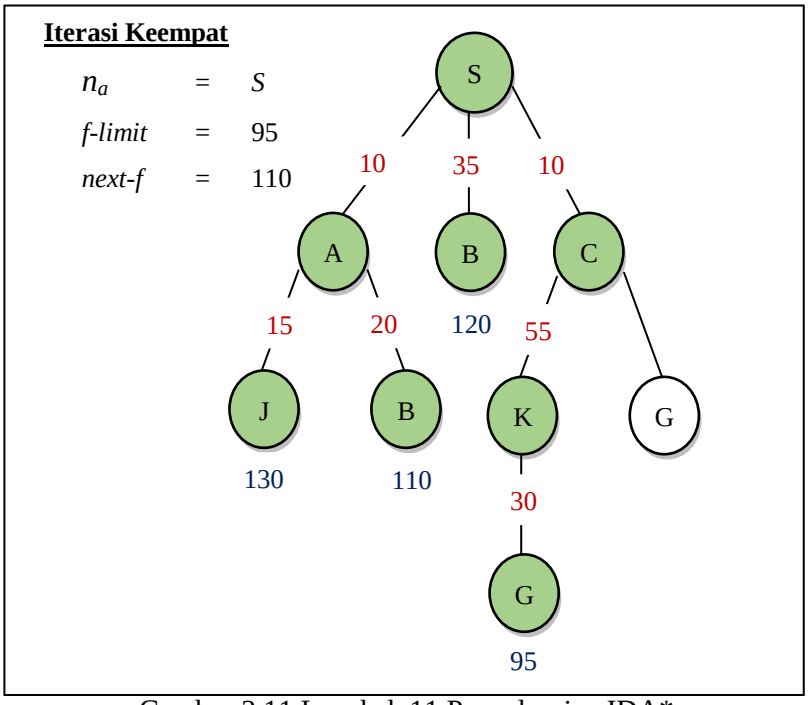
    a. $n_i = G$ (melalui simpul *K*)

    b. Hitung $f(n_i)$ dari simpul suksesor.

    $f(G) = g(K) + c(K,G) + h(G)$

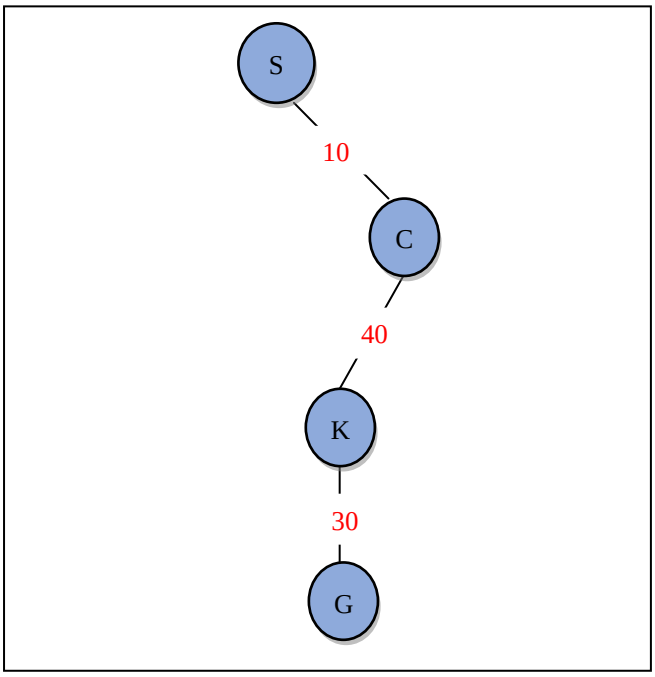
    $= 65 + 30 + 0 = 95$

    c. Simpul *G* merupakan simpul tujuan dan $f(G) \leq$ *f-limit*, oleh karena itu pencarian IDA* berhenti dengan menemukan tujuan yaitu simpul *G*.



Gambar 2.11 Langkah 11 Penyelesaian IDA*

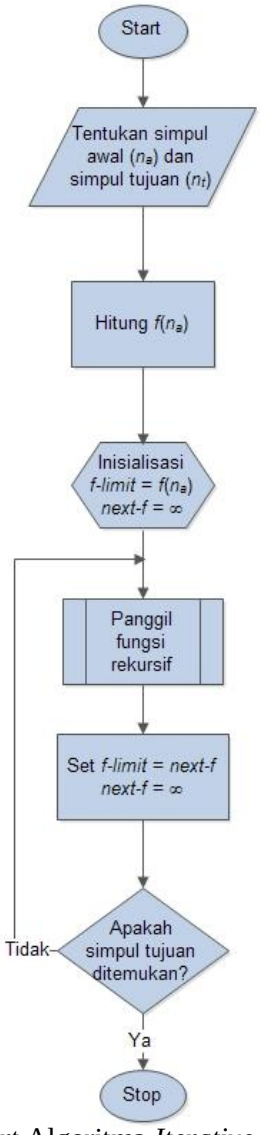
Gambar 2.12 menunjukkan hasil akhir penyelesaian dari algoritma IDA*, diperoleh jalur terpendek yang ditemukan adalah **S-C-K-G** dengan biaya keseluruhan yaitu **95**. Pencarian IDA* akan menghapus jalur yang tidak dipilihnya.



Gambar 2.12 Hasil Akhir Penyelesaian IDA*

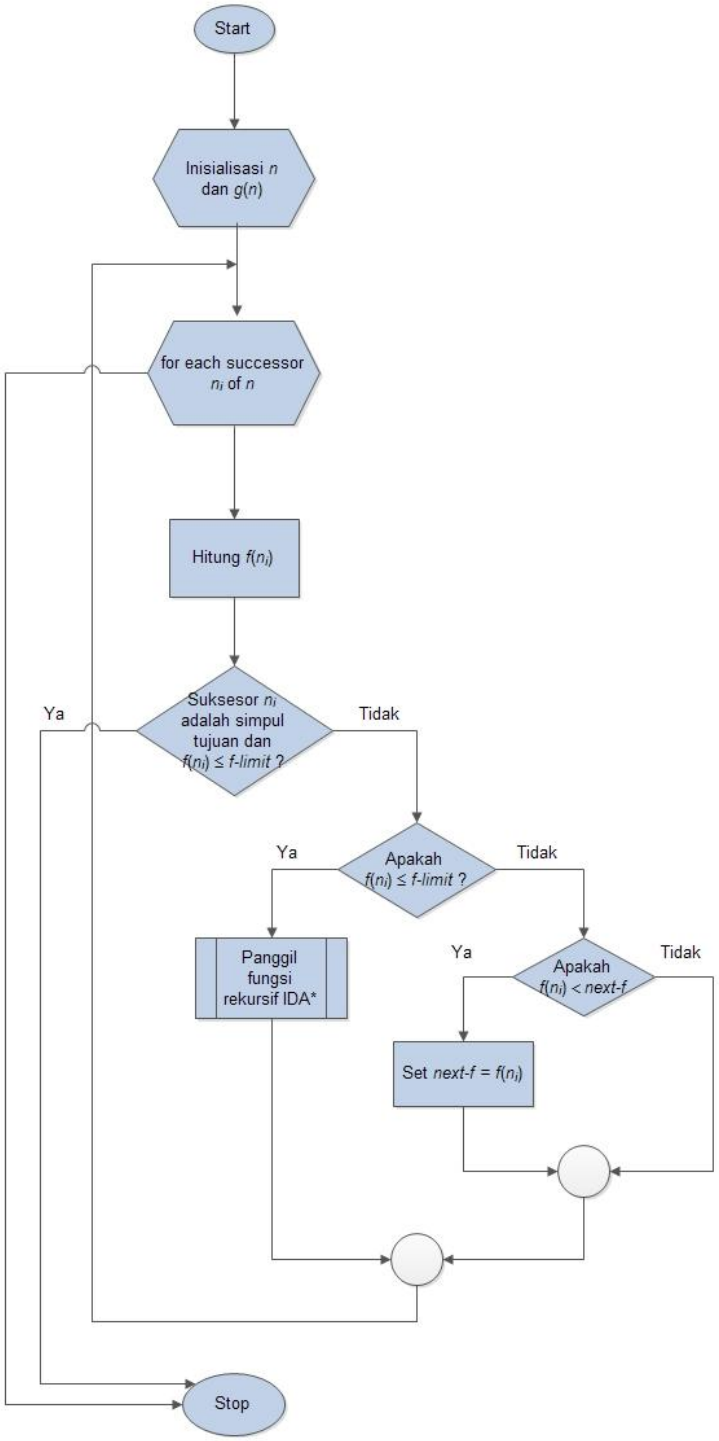
Algoritma IDA* tidak mengingat jalur terpendek saat ini dan biaya simpul-simpul yang pernah dikunjungi, tetapi hanya mengingat satu *path* tunggal dalam satu waktu. Pada akhirmya algoritma IDA* akan menemukan jalur dengan biaya minimum ke simpul tujuan karena hanya meng-*cache* (menyimpan dalam memori) simpul yang berada dalam jalur pencarian. Oleh karena itu jumlah memori yang dibutuhkan menjadi jauh lebih sedikit.

Berdasarkan langkah-langkah pencarian IDA* dari contoh kasus rute terpendek maka dapat dilihat dalam bentuk *flowchart* pada Gambar 2.13. Gambar 2.13 merupakan *flowchart* langkah-langkah pencarian (IDA*), dimana di dalam langkah tersebut memanggil fungsi lain yaitu fungsi rekursif IDA*.



Gambar 2.13 *Flowchart* Algoritma *Iterative Deepening A** (IDA*)

Adapun *flowchart* fungsi rekursif *Iterative Deepening A** (IDA*) dapat dilihat pada Gambar 2.14. Fungsi rekursif IDA* merupakan fungsi yang digunakan untuk memanggil fungsi itu sendiri ketika simpul suksesor akan diekspansi karena nilai $f(n_i)$ lebih kecil sama dengan nilai *f-limit*.



Gambar 2.14 *Flowchart* Fungsi Rekursif Algoritma *Iterative Deepening A** (IDA*)

Pada Gambar 2.15 mengilustrasikan algoritma IDA* secara detail. Dilihat algoritma tersebut terdapat dua fungsi yaitu IDA* dan IDA*_REKURSIF. Fungsi IDA*_REKURSIF adalah fungsi rekursif (fungsi yang memanggil dirinya sendiri) yang secara implisit merepresentasikan *backtracking* (penelurusuran balik). Ketika pembangkitan simpul sudah melampaui batas *f-limit*, maka akan dibangkitkan simpul-simpul pada level sebelumnya.

```
function IDA*(n_a)
   f-limit ← f(n_a)
   next-f ← ∞
   repeat until simpul tujuan ditemukan
     IDA*_REKURSIF(n_a,0)
     f-limit ← next-f
     next-f ← ∞
   end repeat

function IDA*_REKURSIF(n,g(n))
   for setiap suksesor n_i dari simpul n do
     f(n_i) = g(n) + c(n,n_i) + h(n_i)
     if f(n_i) is a goal node and f(n_i) ≤ f-limit then stop
     if f(n_i) ≤ f-limit then
       IDA*-REKURSIF(n_i, g(n) + c(n,n_i))
     else if f(n_i) < next-f then
       next-f ← f(n_i)
     endif
   end do
```

Gambar 2.15 *Pseudo-code* Algoritma IDA* [4]

2.6 Algoritma IDA* dengan *Stochastic Node Caching* (SNC)

Karakteristik algoritma IDA* adalah *complete* dan optimal. Tetapi karena dilakukan secara iteratif, algoritma IDA* mungkin membangkitkan simpul-simpul yang sama secara berulang-ulang, sehingga penghematan memori harus dibayar dengan kompleksitas waktu tinggi yaitu sebesar $O(b^d)$ [8]. Sebagai contoh, terdapat ruang masalah dengan 1000 simpul, masing-masing dengan 2 faktor percabangan. Jika pada pencarian terdapat 20 simpul dalam array solusi, maka algoritma IDA* perlu membangkitkan 2^{20} = 1.048.576 simpul. Untuk membangkitkan simpul sebanyak ini, algoritma IDA* pasti membutuhkan waktu eksekusi yang sangat lama.

Stochastic Node Caching merupakan sebuah teknik yang dapat mengurangi jumlah revisit (kunjungan kembali) simpul-simpul yang dibangkitkan yang terjadi pada algoritma IDA*. Algoritma IDA* SNC seperti algoritma IDA*, memiliki sifat *complete* dan *optimal* [4]. Cara kerja IDA* SNC menggunakan parameter tambahan berupa inputan awal yaitu *M* dan *p* [4]. Nilai *M* merupakan jumlah maksimum simpul yang dapat disimpan dalam memori. IDA* SNC mengalokasikan ruang memori yang dapat menyimpan *M* simpul. Alokasi ruang memori yang digunakan hanya untuk menyimpan nilai heuristik saja, sehingga jumlah memori yang digunakan tidak terlalu signifikan. Karena pada akhirnya IDA* SNC juga sama seperti IDA*, yaitu akan menghapus simpul yang bukan jalurnya. Sedangkan *p* adalah probabilitas yang menentukan bagaimana sebuah simpul dalam *cache* setiap kali diekspansi. Nilai *p* yang diinputkan yaitu dalam rentang nilai 0 sampai 1 sesuai dengan nilai peluang. Untuk menentukan nilai *M* dan *p* yang efektif¸ maka dapat dilakukan dengan percobaan nilai inputannya.

Pemilihan untuk inputan nilai parameter dengan hasil jumlah ekspansi dan waktu pencarian yang lebih sedikit dapat meningkatkan performa eksekusi algoritma tersebut. Sehingga diharapkan kedua parameter tambahan ini dapat mempengaruhi algoritma terhadap cara kerja algoritma IDA* yaitu dalam hal jumlah ekspansi simpul sehingga dapat mengurangi kemungkinan terjadinya kunjungan kembali ke simpul yang sama dan waktu pencarian yang lebih cepat.

Ketika jumlah simpul tertentu (*N*) masih bisa disimpan pada *M* dan nilai random (*RAND* dengan *range* 0-1) lebih kecil sama dengan *p*, maka IDA* SNC akan menyimpan pada *cache* setiap simpul suksesor n_i (anak dari simpul *n*) dan $h(n_i)$ (nilai heuristik) ketika simpul *n* diekspansi. Sehingga pada akhir pencarian, nilai heuristik simpul yang di-*cache* akan di *update* nilainya menjadi $c(n,n_i)$ + $h(n_i)$ kemudian jika pencarian dilakukan kembali dari awal pencarian maka nilai $f(n_i)$ bertambah besar. Oleh karena itu jika pada IDA* simpul tertentu menghabiskan banyak waktu untuk mengekspansi terus-menerus, IDA* SNC tidak mengekspansi simpul tersebut karena nilai $f(n_i)$ akan melebihi nilai *f-limit* nya sehingga waktu pencarian dapat berkurang. Semakin sering sebuah simpul diekspansi, semakin besar kemungkinan simpul tersebut untuk tersimpan di *cache*

jika *M* masih dapat menyimpan simpul. Tetapi jika *M* sudah penuh maka pencarian persis dengan IDA*.

Adapun langkah-langkah manual pencarian IDA* SNC adalah sebagai berikut :

1. Berikan nilai inputan awal untuk jumlah maksimum simpul yang akan disimpan (*M*).
2. Berikan nilai inputan awal untuk probabilitas (*p*) dalam rentang nilai 0 sampai 1.
3. Tentukan simpul awal (n_a) dan simpul tujuan (n_t).
4. Hitung nilai $f(n_a)$.
5. Jadikan nilai $f(n_a)$ sebagai *f-limit*.
6. Inisialisasi *next-f* = ∞.
7. Inisialisasi *cutoff* = ∞.
8. Apakah simpul sudah ada di *cache M* ?
   i. Iya, lanjutkan ke langkah 10.
   ii. Tidak, lanjutkan ke langkah 9.
9. Apakah *cache M* masih bisa menyimpan simpul dan nilai $RAND \leq p$?
   i. Iya, simpan simpul dan nilai heuristik $h(n)$ ke dalam *cache M*.
   ii. Tidak, lanjutkan ke langkah 10.
10. Ekspansi simpul *n* dengan lakukan perulangan berikut ini untuk setiap suksesor n_i dari simpul *n* :

    10a. Hitung nilai $f(n_i)$

    10b. Apakah $f(n_i)$ adalah simpul tujuan dan $f(n_i) \leq$ *f-limit* ?
       i. Iya, maka pencarian berhenti.
       ii. Tidak, lakukan langkah 10c.

    10c. Apakah $f(n_i) \leq$ *f-limit* ?
       i. Iya, lakukan pencarian dari langkah 7 sampai langkah 11 dari simpul saat ini. Kemudian lanjut ke langkah 10e.
       ii. Tidak, lakukan langkah 10d.

    10d. Apakah $f(n_i) <$ *next-f* ?
       i. Iya, set nilai $f(n_i)$ sebagai *next-f*.

ii. Tidak, nilai *next-f* tetap.

10e. Setelah langkah 10c atau 10d dilakukan, maka lanjutkan ke langkah 10f

10f. Apakah nilai $h(n_i) + c(n,n_i) < cutoff$?

i. Iya, set nilai $h(n_i) + c(n,n_i)$ sebagai *cutoff*.

ii. Tidak, nilai *cutoff* tetap.

11. Setelah perulangan selesai, *update* nilai heuristik simpul saat ini / $h(n)$ di dalam *cache M* dengan nilai *cutoff*.
12. Set nilai *next-f* sebagai nilai *f-limit* baru.
13. Inisialisasi kembali nilai *next-f* menjadi ∞.
14. Ulangi pencarian dari status awal (*restart*) mulai dari langkah 7-14 dengan menggunakan nilai batasan *f-limit* baru. Lakukan sampai simpul tujuan ditemukan.

Pemahaman untuk algoritma IDA* SNC akan dijelaskan bedasarkan contoh kasus sebelumnya (lihat Gambar 2.7). Penyelesaikan masalah rute terpendek menggunakan algoritma IDA* SNC adalah sebagai berikut.

1. Input nilai M dan p $(0 \leq p \leq 1)$
   Misalkan $M = 2$ dan $p = 0{,}5$
2. Tentukan simpul awal (n_a) dan simpul tujuan (n_t) kemudian hitung $f(n_a)$. Untuk simpul awal, simpul tujuan dan perhitungan nilai $f(n_a)$ pada IDA* SNC sama seperti langkah 2 pada pencarian IDA*.
   $n_a = S$, $n_t = G$, $f(S) = 80$.
3. Inisialisasi *f-limit* = $f(n_a) = 80$ dan *next-f* = ∞.
4. Inisialisasi *cutoff* = ∞.
5. Apakah $N < M$ dan $RAND \leq p$? (*RAND* adalah angka acak yang didapat dari fungsi random, $0 \leq RAND \leq 1$).
   Misalkan angka random yang didapat yaitu $RAND = 0{,}384$.
   $N < M$ dan $RAND \leq p$
   $0 < 2$ dan $0{,}384 \leq 0{,}5$? Benar, maka simpan $h(S)$ ke dalam *cache M*.

	M
0	$h(S) = 80$
1	

6. Lakukan perulangan untuk setiap suksesor n_i dari simpul *n*. Simpul *S* mempunyai 3 simpul suksesor, maka lakukan perulangan sebanyak 3 kali. Lihat Gambar 2.16. Untuk semua perhitungan nilai $f(n_i)$ dan *next-f* pada pencarian IDA* SNC, caranya sama dengan perhitungan pencarian $f(n_i)$ dan *next-f* pada langkah-langkah pencarian IDA*.

   suksesor 1 dari simpul *S*

   a. $n_i = A$, $f(A) = 84$, *next-f* = $f(A) = 84$

   b. $h(A) + c(S,A) < cutoff$?

      $74 + 10 < \infty$? Benar, maka *cutoff* = $h(A) + c(S,A) = 84$

   suksesor 2 dari simpul *S*

   a. $n_i = B$, $f(B) = 120$, *next-f* = 84.

   b. $h(B) + c(S,B) < cutoff$?

      $85 + 35 < 84$? Tidak, maka *cutoff* tetap = 84

   suksesor 3 dari simpul *S*

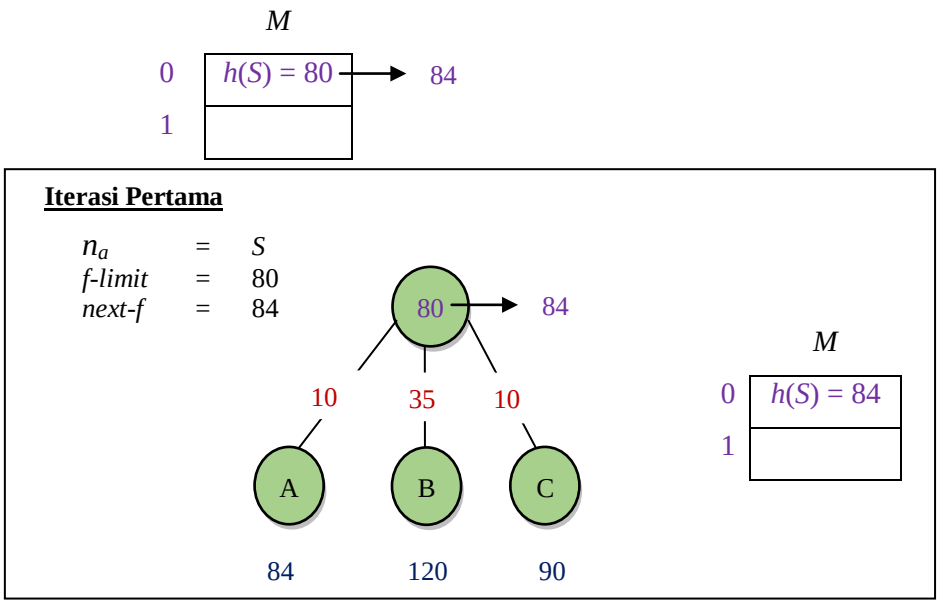
   a. $n_i = C$, $f(C) = 90$, *next-f* = 84

   b. $h(C) + c(S,C) < cutoff$?

      $80 + 10 < 84$? Tidak, maka *cutoff* tetap = 84

7. *Update* nilai $h(S)$ dengan nilai *cutoff*.

   $h(S) = 84$



Gambar 2.16 Langkah 6 – 7 Penyelesaian IDA* SNC

8. Setelah semua suksesor diekspansi, set *f-limit* = *next-f* dan *next-f* = ∞.
   f-limit = 84 ; *next-f* = ∞.
9. Karena simpul tujuan belum ditemukan, ulangi pencarian dari status awal seperti langkah 4 – 7.
10. Inisialisasi *cutoff* = ∞.
11. Karena simpul *S* dan heuristiknya sudah disimpan dalam *cache* maka langsung lakukan untuk setiap suksesor n_i dari simpul *S* dengan *f-limit* sekarang yaitu 84.
12. Simpul *S* mempunyai 3 simpul suksesor, maka lakukan perulangan sebanyak 3 kali.

    suksesor 1 dari simpul S

    a. $n_i = A$, $f(A) = 84$
    b. Karena $f(A) \leq$ *f-limit*, maka lakukan langkah seperti langkah 4 – 7 dengan simpul saat ini adalah simpul *A*. Lihat Gambar 2.17 dan Gambar 2.18.

12a. Inisialisasi *cutoff* = ∞.

12b. Apakah $N < M$ dan $RAND \leq p$?

Misalkan angka acak yang didapat yaitu *RAND* = 0,454.

$N < M$ dan $RAND \leq p$

0 < 2 dan 0,454 ≤ 0,5 ? Benar, maka simpan *h*(*A*) ke dalam *cache M*.

	M
0	*h*(*S*) = 84
1	*h*(*A*) = 74

12c. Simpul *A* mempunyai 2 simpul suksesor, maka lakukan perulangan sebanyak 2 kali. Lihat gambar 2.17.

suksesor 1 dari simpul *A*

a. $n_i = J$, $f(J) = 130$
b. Karena $f(J) >$ *f-limit* dan $f(J) <$ *next-f*, maka $f(J)$ dijadikan sebagai *next-f*.

   $f(J) < \infty$, maka *next-f* = $f(J)$ = 130

Gambar 2.17 Langkah 12 Penyelesaian IDA* SNC

c. $h(J) + c(A,J) < cutoff$?

100 + 20 < ∞ ? Benar, maka $cutoff = h(J) + c(A,J) = 120$

suksesor 2 dari simpul *A*

a. $n_i = B$ (melalui simpul *A*), $f(B) = 110$

b. Karena $f(B) > f\text{-}limit$ dan $f(B) < next\text{-}f$, maka $f(B)$ dijadikan sebagai *next-f*.

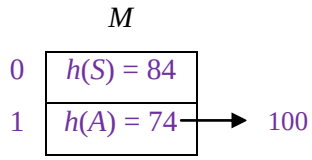
$f(B) < 130$, maka $next\text{-}f = f(B) = 110$

c. $h(B) + c(A,B) < cutoff$?

85 + 15 < 120 ? Benar, maka $cutoff = h(B) + c(A,B) = 100$

12d. *Update* nilai $h(A)$ dengan nilai *cutoff*.

$h(A) = 100$



Gambar 2.18 Langkah 12 Penyelesaian IDA* SNC (2)

c. $h(A) + c(S,A) < cutoff$?

Nilai heurstik simpul *A* yang digunakan adalah heuristik pada *cahe* yaitu 100.

100 + 10 < ∞ ? Benar, maka *cutoff* $h(A) + c(S,A) = 110$

13. Hasil ekspansi simpul *A* belum menemukan simpul tujuan, oleh karena itu pencarian dilanjutkan arah ke simpul kanan yaitu simpul *B* dan *C*. Lihat Gambar 2.19.

suksesor 2 dari simpul *S*

a. $n_i = B$, $f(B) = 120$, $next\text{-}f = 110$

b. $h(B) + c(S,B) < cutoff$?

85 + 35 < 110 ? Tidak, maka *cutoff* tetap = 110

suksesor 3 dari simpul *S*

a. $n_i = C$, $f(C) = 90$

b. Karena $f(C) > f\text{-}limit$, tetapi $f(C) < next\text{-}f$, maka $f(n_i)$ dijadikan sebagai *next-f*.

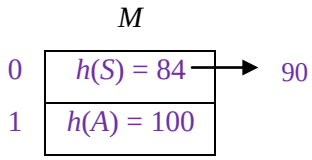
$f(C) < 110$, maka *next-f* = $f(C)$ = 90

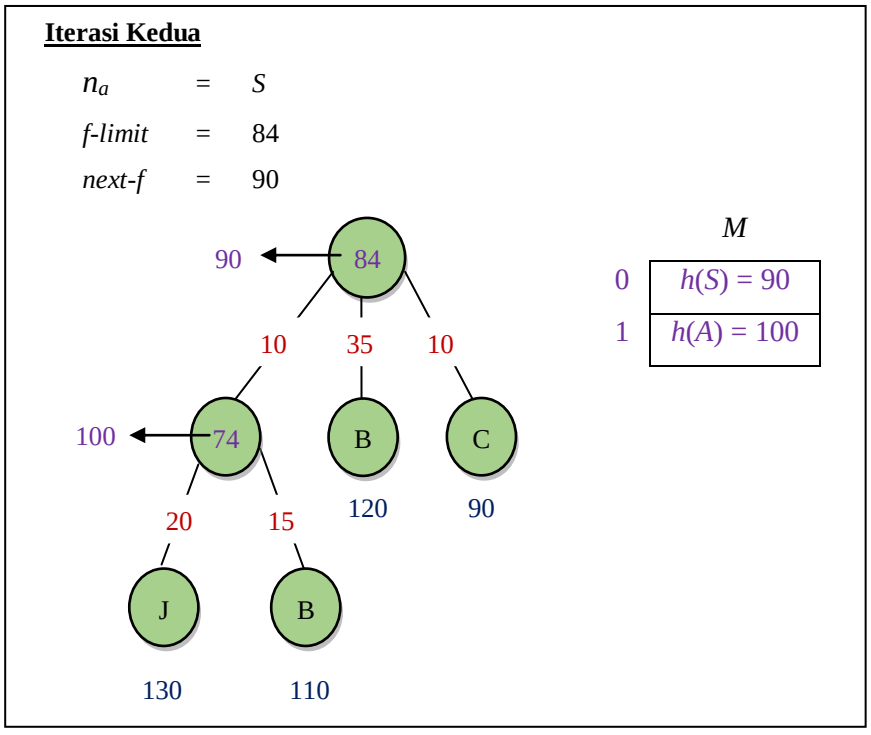
c. $h(C) + c(S,C) < cutoff$?

$80 + 10 < 110$? Benar, maka *cutoff* = $h(C) + c(S,C) = 90$

14. *Update* nilai $h(S)$ dengan nilai *cutoff*..

$h(S) = 90$





Gambar 2.19 Langkah 13 – 14 Penyelesaian IDA* SNC

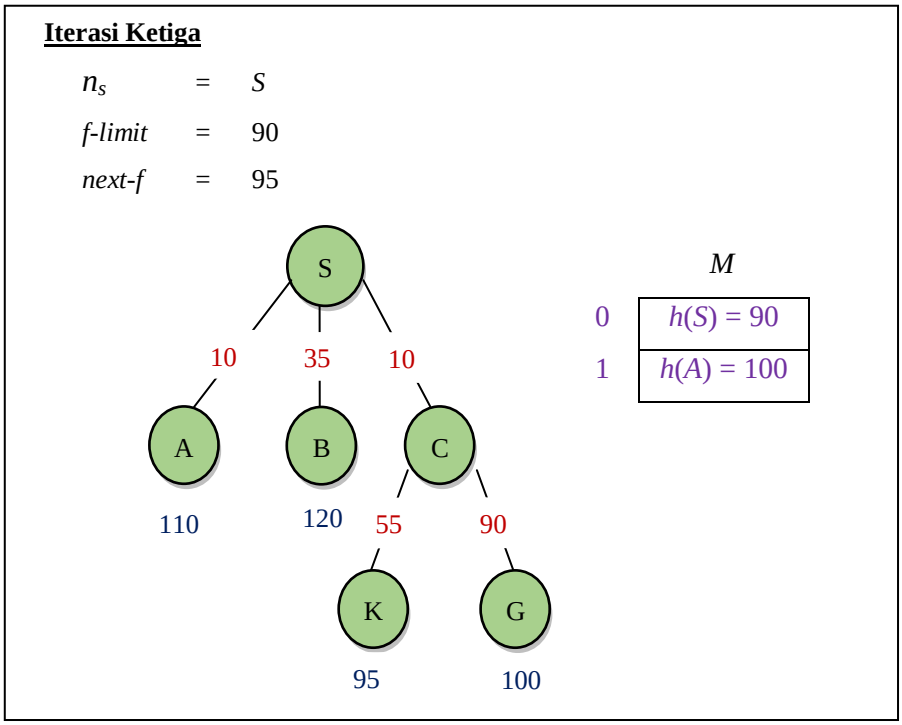
15. Setelah semua suksesor diekspansi, set *f-limit* = *next-f* dan *next-f* = ∞.

f-limit = 90 ; *next-f* = ∞.

16. Karena simpul tujuan belum ditemukan, ulangi pencarian dari status awal seperti langkah 4 – 7.
17. Inisialisasi *cutoff* = ∞.
18. Karena simpul *S* dan heuristiknya sudah disimpan dalam *cache* dan *M* sudah tidak dapat menyimpan simpul lagi, maka pencarian dilakukan seperti algoritma IDA* dengan *f-limit* sekarang yaitu 90.

19. Simpul *S* mempunyai 3 simpul suksesor, maka lakukan perulangan sebanyak 3 kali. Pencarian dilakukan sama seperti langkah sebelumnya, namun pada iterasi ketiga ini, penggunaan heuristik simpul *A* berpengaruh terhadap ekspansi simpul. Karena nilai heuristik simpul *A* pada *cache M* semakin bertambah, maka nilai *f*(*A*) juga semakin bertambah dan melebihi nilai *f-limit*. Sehingga pencarian dilanjutkan ke suksesor berikutnya yaitu simpul *B* dan *C*. Pada suksesor ketiga yaitu simpul *C*, nilai *f*(*C*) ≤ *f-limit* tetapi *cache M* sudah tidak bisa menyimpan simpul lagi, oleh karena itu pencarian menjadi seperti IDA* kembali. Lihat Gambar 2.20.

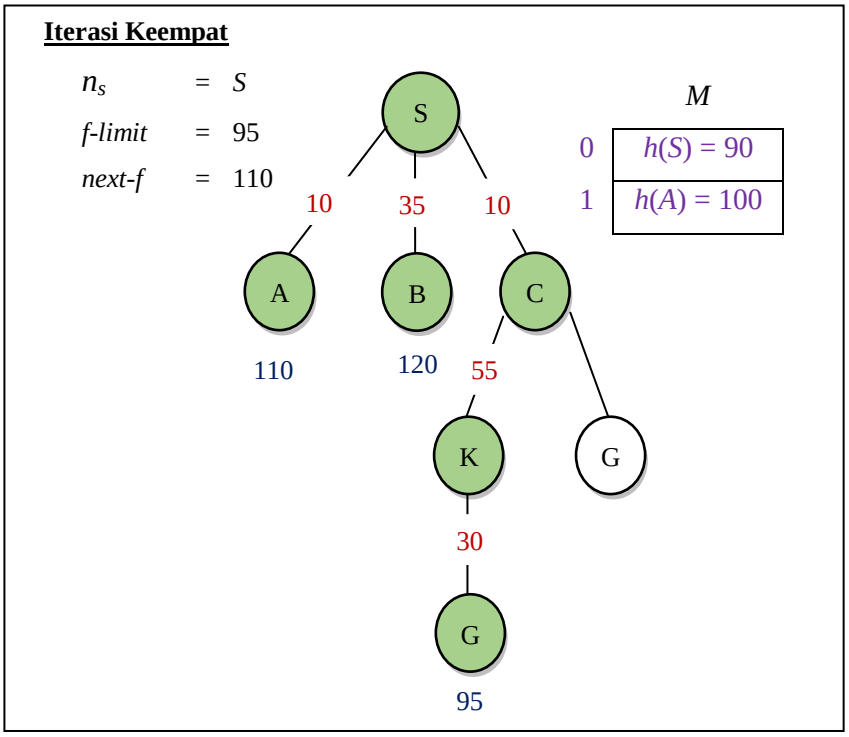
$f(A) = g(S) + c(S,A) + h(A) = 0 + 10 + 100 = 110$



Gambar 2.20 Langkah 19 Penyelesaian IDA* SNC

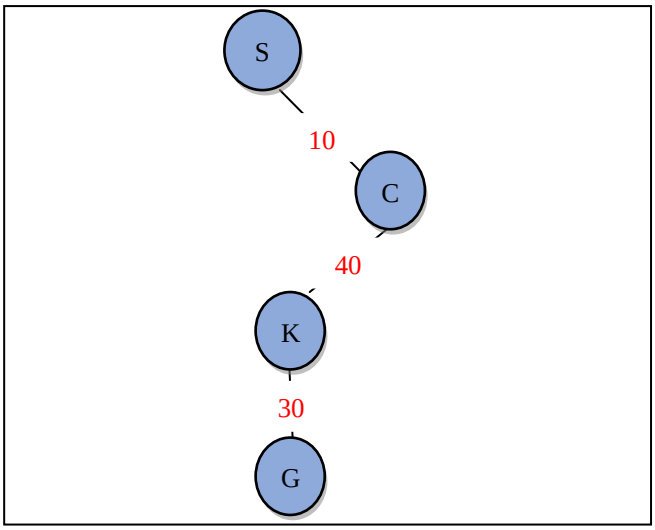
Dari Gambar 2.20 dan Gambar 2.21 terlihat penggunaan algoritma IDA* SNC sangat berpengaruh terhadap ekspansi simpul. Simpul *A* pada kasus dengan menggunakan pencarian IDA* mengekspansi setiap pengulangan iterasi dari awal pencarian, sedangkan IDA* SNC tidak mengekspansi simpul yang sama pada iterasi ketiga dan keempat karena sudah dipastikan jalur

melalui simpul *A* bukanlah simpul tujuan. Oleh karena itu waktu yang dibutuhkan untuk pencarian menjadi lebih sedikit.



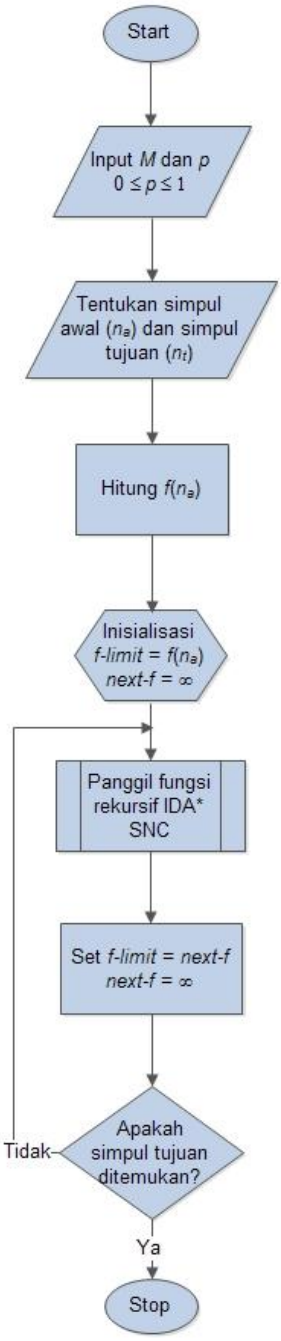
Gambar 2.21 Iterasi Keempat Penyelesaian IDA* SNC

Gambar 2.22 menunjukkan hasil akhir penyelesaian dari algoritma IDA* SNC, diperoleh jalur terpendek yang ditemukan adalah **S-C-K-G** dengan biaya keseluruhan yaitu **95**. Pencarian IDA* SNC juga akan menghapus jalur yang tidak dipilihnya.



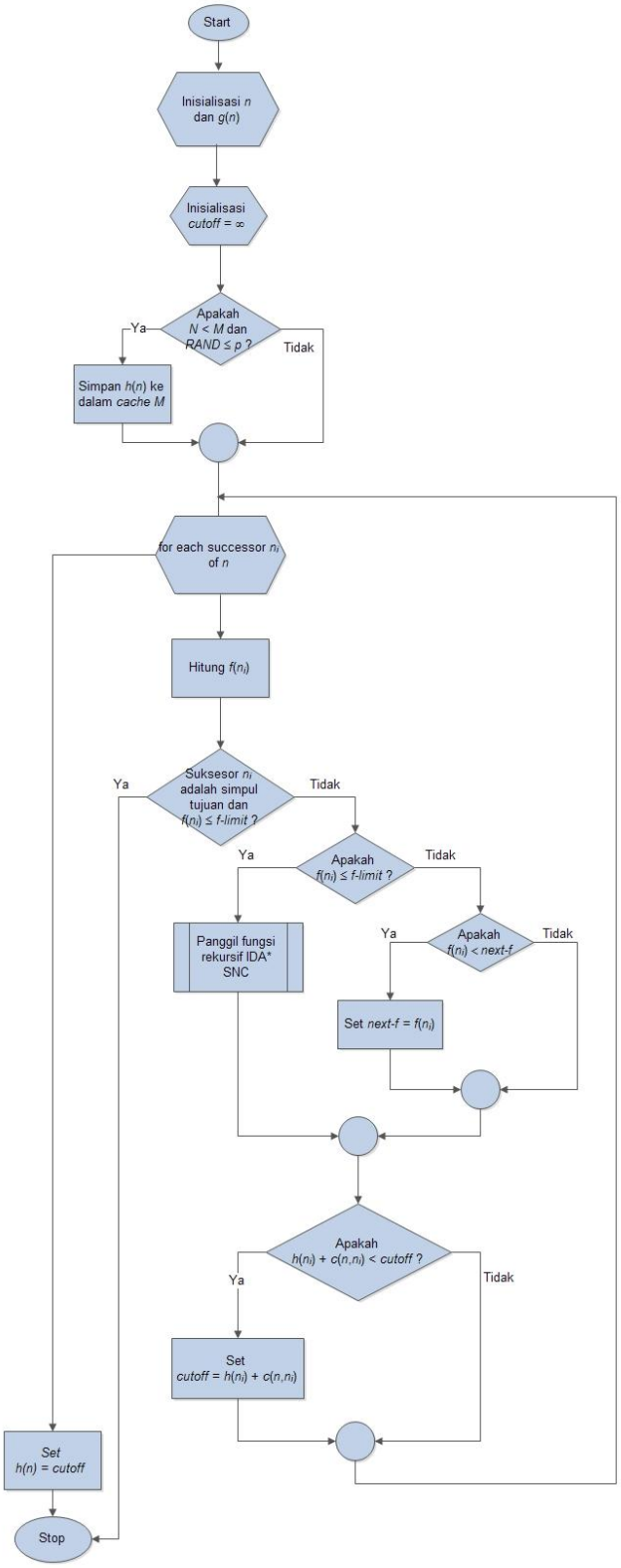
Gambar 2.22 Hasil Akhir Penyelesaian IDA* SNC

Berdasarkan langkah-langkah pencarian IDA* SNC maka dapat dilihat dalam bentuk *flowchart* pada Gambar 2.23. *Flowchart* langkah-langkah IDA* SNC mirip dengan *flowchart* langkah-langkah pada IDA*. Pada IDA* SNC juga memanggil fungsi rekursif IDA* SNC seperti pada IDA* yang memanggil fungsi rekursi IDA*. Perbedaannya yaitu pada fungsi IDA* SNC memiliki parameter tambahan yang diinputkan yaitu nilai *M* dan *p*.



Gambar 2.23 *Flowchart* Algoritma IDA* SNC

Adapun *flowchart* fungsi rekursif IDA* SNC dapat dilihat pada Gambar 2.24.



Gambar 2.24 *Flowchart* Fungsi Rekursif Algoritma IDA* SNC

Sedangkan *pseudo-code* dari algoritma IDA* SNC dapat dilihat pada Gambar 2.25.

```
function IDA*SNC(n_a)
   f-limit ← f(n_a)
   next-f ← ∞
   repeat until simpul tujuan ditemukan
     IDA*SNC_REKURSIF(n_a,0)
     f-limit ← next-f
     next-f ← ∞
   end repeat
function IDA*SNC_REKURSIF(n,g(n))
   cutoff ← ∞
   if N < M and RAND ≤ P then
     cache n with h(n)
   endif
   for setiap suksesor n_i dari simpul n do
     f(n_i) = g(n) + c(n,n_i) + h(n_i)
     if f(n_i) is a goal node and f(n_i) ≤ f-limit then stop
     if f(n_i) ≤ f-limit then
       IDA*SNC_REKURSIF(n_i, g(n) + c(n,n_i))
     else if f(n_i) < next-f then
       next-f ← f(n_i)
     endif
     if h(n_i)+ c(n,n_i) < cutoff then
       cutoff ← h(n_i)+ c(n,n_i)
     endif
   end do
   h(n) ← cutoff
```

Gambar 2.25 *Pseudo-code* Algoritma IDA* SNC [4]

2.7 OOP (*Object Oriented Programming*)

Objek adalah kesatuan entitas yang memiliki sifat dan tingkah laku. Dalam kehidupan sehari-hari, objek adalah benda, baik benda berwujud nyata seperti manusia, hewan, mobil, komputer, handphone, pena, ataupun benda yang tidak nyata atau konsep, seperti halnya tabungan bank, sistem antrian, sistem internet banking, dan sebagainya. Jadi pengertian OOP adalah konsep yang membagi program menjadi objek-objek yang saling berinteraksi satu sama lain. Objek adalah benda, baik benda yang berwujud nyata maupun benda yang tidak nyata (konsep). Jika menggunakan OOP maka akan ada enam keuntungan yang dapat diperoleh, yaitu [13]:

a. Alami (*Natural*).
b. Dapat diandalkan (*Reliable*).
c. Dapat digunakan kembali (*Reusable*).
d. Mudah untuk dalam perawatan (*Maintainable*).
e. Dapat diperluas (*Extendable*).
f. Efisiensi waktu.

Berikut ini beberapa bahasa pemrograman yang sudah menggunakan konsep OOP, adalah :

1. *C++*.
2. *Visual C++*.
3. *Visual Basic*.
4. *Java*.

2.8 UML (*Unified Modeling Language*)

UML adalah bahasa spesifikasi standar untuk mendokumentasikan, menspesifikasikan, dan membangun sebuah sistem. UML adalah himpunan struktur dan teknik untuk pemodelan desain program berorientasi objek (OOP) serta aplikasinya. UML adalah metodologi untuk mengembangkan sistem OOP dan sekelompok perangkat *tool* untuk mendukung pengembangan sistem tersebut. UML mulai diperkenalkan oleh *Object Management Group*, sebuah organisasi yang telah mengembangkan model, teknologi, dan standar OOP sejak tahun 1980-an. Sekarang UML sudah mulai banyak digunakan oleh para praktisi OOP. UML merupakan dasar bagi perangkat (*tool*) desain berorientasi objek dari IBM.

UML mendefinisikan diagram-diagram sebagai berikut [14]:

1. *use case diagram*
2. *activity diagram*
3. *sequence diagram*
4. *class diagram*
5. *statechart diagram*
6. *collaboration diagram*
7. *component diagram*

8. *deployment diagram*

2.8.1 *Use Case Diagram*

Use case diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Yang ditekankan adalah "apa" yang diperbuat sistem, dan bukan "bagaimana". Sebuah *use case* merepresentasikan sebuah interaksi antara aktor dengan sistem. *Use case* merupakan sebuah pekerjaan tertentu, misalnya *login* ke sistem, meng-*create* sebuah daftar belanja, dan sebagainya. Seorang atau sebuah aktor adalah sebuah entitas manusia atau mesin yang berinteraksi dengan sistem untuk melakukan pekerjaan-pekerjaan tertentu.

Use case diagram dapat sangat membantu menyusun *requirement* sebuah sistem, mengkomunikasikan rancangan dengan klien, dan merancang *test case* untuk semua *feature* yang ada pada sistem. Sebuah *use case* dapat meng-*include* fungsionalitas *use case* lain sebagai bagian dari proses dalam dirinya. Secara umum diasumsikan bahwa *use case* yang di-*include* akan dipanggil setiap kali *use case* yang meng-*include* dieksekusi secara normal.

Sebuah *use case* dapat di-*include* oleh lebih dari satu *use case* lain, sehingga duplikasi fungsionalitas dapat dihindari dengan cara menarik keluar fungsionalitas yang *common*. Sebuah *use case* juga dapat meng-*extend use case* lain dengan *behaviour*-nya sendiri. Sementara hubungan generalisasi antar *use case* menunjukkan bahwa *use case* yang satu merupakan spesialisasi dari yang lain.

2.8.2 *Activity Diagram*

Activity diagram menggambarkan berbagai alur aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing alur berawal, *decision* yang mungkin terjadi, dan bagaimana mereka berakhir. *Activity diagram* juga dapat menggambarkan proses paralel yang mungkin terjadi pada beberapa eksekusi.

Activity diagram merupakan *state diagram* khusus, dimana sebagian besar *state* adalah *action* dan sebagian besar transisi di-*trigger* oleh selesainya *state* sebelumnya (*internal processing*). Oleh karena itu *activity diagram* tidak

menggambarkan *behaviour* internal sebuah sistem (dan interaksi antar subsistem) secara eksak, tetapi lebih menggambarkan proses-proses dan jalur-jalur aktivitas dari level atas secara umum.

2.8.3 *Sequence Diagram*

Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem (termasuk pemain, *display*, dan sebagainya) berupa *message* yang digambarkan terhadap waktu. *Sequence diagram* terdiri atar dimensi vertikal (waktu) dan dimensi horizontal (objek-objek yang terkait).

Sequence diagram biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respons dari sebuah *event* untuk menghasilkan *output* tertentu. Diawali dari apa yang men-*trigger* aktivitas tersebut, proses dan perubahan apa saja yang terjadi secara internal dan *output* apa yang dihasilkan.

2.8.4 *Class Diagram*

Class adalah sebuah spesifikasi yang jika diinstansiasi akan menghasilkan sebuah objek dan merupakan inti dari pengembangan dan desain berorientasi objek. *Class* menggambarkan keadaan (atribut/properti) suatu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut (metoda/fungsi).

Class diagram menggambarkan struktur dan deskripsi *class, package* dan objek beserta hubungan satu sama lain seperti *containment*, pewarisan, asosiasi, dan lain-lain.

2.8.5 *Statechart Diagram*

Statechart diagram menggambarkan transisi dan perubahan keadaan (dari satu *state* ke *state* lainnya) suatu objek pada sistem sebagai akibat dari *stimuli* yang diterima. Pada umumnya *statechart diagram* menggambarkan *class* tertentu (satu *class* dapat memiliki lebih dari satu *statechart diagram*).

Dalam UML, *state* digambarkan berbentuk segiempat dengan sudut membulat dan memiliki nama sesuai kondisinya saat itu. Transisi antar *state*

umumnya memiliki kondisi *guard* yang merupakan syarat terjadinya transisi yang bersangkutan, dituliskan dalam kurung siku. *Action* yang dilakukan sebagai akibat dari *event* tertentu dituliskan dengan diawali garis miring.

2.8.6 *Collaboration Diagram*

Collaboration diagram juga menggambarkan interaksi antar objek seperti *sequence diagram*, tetapi lebih menekankan pada peran masing-masing objek dan bukan pada waktu penyampaian *message*. Setiap *message* memiliki *sequence number*, di mana *message* dari level tertinggi memiliki nomor 1. *Messages* dari *level* yang sama memiliki *prefiks* yang sama.

2.8.7 *Component Diagram*

Component diagram menggambarkan struktur dan hubungan antar komponen piranti lunak, termasuk ketergantungan (*dependency*) di antaranya. Komponen piranti lunak adalah modul berisi *code*, baik berisi *source code* maupun *binary code*, baik *library* maupun *executable*, baik yang muncul pada *compile time*, *link time*, maupun *run time*.

Umumnya komponen terbentuk dari beberapa *class* dan/atau *package*, tapi dapat juga dari komponen-komponen yang lebih kecil. Komponen dapat juga berupa *interface*, yaitu kumpulan layanan yang disediakan sebuah komponen untuk komponen lain.

2.8.8 *Deployment Diagram*

Deployment/physical diagram menggambarkan detail bagaimana komponen di-*deploy* dalam infrastruktur sistem, di mana komponen akan terletak (pada mesin, *server* atau piranti keras apa), bagaimana kemampuan jaringan pada lokasi tersebut, spesifikasi *server*, dan hal-hal lain yang bersifat fisikal

Sebuah *node* adalah *server*, *workstation*, atau piranti keras lain yang digunakan untuk men-*deploy* komponen dalam lingkungan sebenarnya. Hubungan antar *node* (misalnya TCP/IP) dan *requirement* dapat juga didefinisikan dalam diagram ini.

2.9 Java

Java adalah salah satu bahasa pemrograman berorientasi objek (OOP-*Object Oriented Programming*) [11]. Paradigma OOP menyelesaikan masalah dengan merepresentasikan masalah ke model objek yang bertujuan untuk mempermudah pengembangan program dengan mengikuti model yang sudah ada pada kehidupan sehari-hari. Menurut Schildt (2005,p2-4), Java dibuat oleh James Gosling, Patrick Naughton, Chris Warth, Ed Frank, dan Mike Sheridan di suatu perusahaan perangkat lunak yang bernama Sun Microsystems, pada tahun 1991. Bahasa pemrograman ini mula-mula diinisialisasi dengan nama "Oak", namun pada tahun 1995 diganti namanya menjadi "Java".

Keunggulan *java* dibanding bahasa pemrograman lainnya, adalah sebagai berikut :

1. Java bersifat sederhana dan relatif mudah

   Java dimodelkan sebagian dari bahasa C++, namun dengan memperbaiki beberapa karakteristik dari C++ sendiri, seperti misalnya dalam mengurangi kompleksitas beberapa fitur, lalu penambahan fungsionalitas, serta penghilangan beberapa aspek pemicu ketidakstabilan sistem pada C++. Sebagai contoh, Java menggantikan konsep pewarisan lebih dari satu (*multiple inheritance*) dengan interface, menghilangkan konsep 37 pointer yang sering membingungkan, otomatis sistem alokasi memori, dan sebagainya. Ini membuat Java menjadi relatif sederhana dan mudah untuk dipelajari dibandingkan bahasa pemrograman lainnya.

2. Java berorientasi pada objek (*Object Oriented*)

   Java adalah bahasa pemrograman yang berorientasi objek bukan seperti Pascal, Basic, atau C yang masih berbasis prosedural. Dalam memecahkan masalah Java membagi program menjadi objek-objek, kemudian memodelkan sifat dan tingkah lakunya masing-masing. Selanjutnya Java menentukan dan mengatur interaksi antara objek yang satu dengan lainnya.

3. Java bersifat terdistribusi

   Pada dekade awal perkembangan PC (Personal Computer), komputer hanya bersifat sebagai workstation tunggal, tidak terhubung satu sama lain. Saat

ini sistem komputerisasi cenderung terdisibusi, mulai dari workstation client, e-mail server, database server, web server, proxy server dan sebainya.

4. Java bersifat *Multiplatform*

   Dewasa ini kita mengenal banyak *platform Operating System*, mulai dari *Windows, Apple*, berbagai varian UNIX dan Linux, dan sebagainya. Pada umumnya program yang dibuat dan dikompile di suatu platform hanya bisa dijalankan di *platform* tersebut. Java bersifat multi *platform*, yakni dapat diterjemahkan oleh Java Interpreter pada berbagai sistem operasi.

5. Java bersifat *MultiThread*

   Thread adalah proses yang dapat dikerjakan oleh program dalam suatu waktu. Java bersifat *Multithreading*, artinya dapat mengerjakan beberapa proses dalam waktu yang hampir bersamaan.