BAB 2
LANDASAN TEORI

2.1. Permainan *Word Search Puzzle*

Permainan *word search puzzle* dikenal sebagai *word find game*, yang terkenal karena membantu siswa untuk mengenali kata-kata. Pada permainan ini huruf dari sebuah kata terletak pada grid dan biasanya memiliki bentuk persegi. Untuk memainkan game ini, pemain mencari dan menandai semua kata-kata tersembunyi di dalam grid. Dalam permainan *word search puzzle*, daftar kata tersembunyi disediakan. Biasanya banyak kata yang terkait mudah bagi pemain untuk dicari. Kata terdaftar dapat diatur dalam arah horizontal, vertikal atau diagonal dalam grid. Semakin cepat penyelesaian setiap tingkat, maka skor yang lebih tinggi akan didapatkan. Dalam mencari kata-kata, pengguna membaca dan menghafal kata-kata saat mereka bermain game yang membantu mereka mempelajari kata-kata dan ejaan, huruf demi huruf, dalam teka-teki [5]. Berikut ini adalah gambar dari salah satu permainan *word search puzzle*:



Gambar 2.1 *Word Search Puzzle* [5]

2.2. *Artificial Intelligence*

Sebagian kalangan menerjemahkan *Artificial Intelligence* sebagai kecerdasan buatan, kecerdasan artifisial, intelijensia artifisial, atau intelijensia buatan. Para ahli mendefinisikan AI secara berbeda-beda tergantung pada sudut pandang mereka masing-masing. Ada yang fokus pada logika berpikir manusia saja, tetapi ada juga yang mendefinisikan AI secara lebih luas pada tingkah laku manusia [2]. Menurut Stuart Russel dan Peter Norvig [5] mengelompokkan definisi AI, yang diperoleh dari beberapa *textbook* yang berbeda, ke dalam empat kategori, yaitu:

1. ***Thinking humanly****: the cognitive modeling approach*

   Pendekatan ini dilakukan dengan dua cara sebagai berikut:

   a. Melalui introspeksi: mencoba menangkap pemikiran-pemikiran sendiri pada saat berpikir. Tetapi, seorang psikolog Barat mengatakan: *"how do you know that you understand?"* Bagaimana Anda tahu bahwa Anda mengerti? Karena pada saat Anda menyadari pemikiran Anda, ternyata pemikiran tersebut sudah lewat dan digantikan kesadaran Anda. Sehingga, definisi ini terkesan mengada-ada atau tidak mungkin dilakukan.

   b. Melalui eksperimen psikologi.

2. ***Acting Humanly****: the turing test approach*

   Pada tahun 1950, Alan Turing merancang suatu ujian bagi komputer berintelijensia untuk mengetahui apakah komputer tersebut mampu mengelabuhi seorang manusia yang menginterogasinya melalui *teletype* (komunikasi berbasis teks jarak jauh). Jika *interrogator* tidak dapat membedakan yang diinterogasi adalah manusia atau komputer, maka komputer berintellijensia tersebut lolos dari Turing test. Komputer tersebut perlu memiliki kemampuan: *Natural Language Processing, Knowledge Representation, Automated Reasoning, Machine Learning, Computer Vision, Robotics*. *Turing test* sengaja menghindari interaksi fisik antara *interrogator* dan komputer karena simulasi fisik manusia tidak memerlukan intelijensia.

3. ***Thinking Rationally****: the laws of thought approach*

   Terdapat dua masalah dalam pendekatan ini, yaitu:

   a. Tidak mudah untuk membuat pengetahuan informal dan menyatakan pengetahuan tersebut ke dalam *formal term* yang diperlukan oleh notasi logika, khususnya ketika pengetahuan terseubut memiliki kepastian kurang dari 100%.
   b. Terdapat perbedaan besar antara memecahkan masalah "dalam prinsip" dan memecahkannya "dalam dunia nyata".

4. ***Acting Rationally****: the rational agent approach*

   Membuat inferensi yang logis merupakan bagian dari suatu *rational agent*. Hal ini disebabkan satu-satunya cara untuk melakukan aksi secara rasional adalah menalar secara logis. Dengan menalar secara logis, maka bisa didapatkan kesimpulan bahwa aksi yang diberikan akan mencapai tujuan atau tidak. Jika mencapai tujuan, maka *agent* dapat melakukan aksi berdasarkan kesimpulan tersebut.

Teknik pemecahan masalah yang ada di dalam *Artificial Intelligence* dibagi menjadi 4 yaitu [2]:

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 yaitu mempresentasikan masalah ke dalam basis pengetahuan dan melakukan proses penalaran untuk menemukan solusi.

c. *Planning*

   Planning adalah suatu metode penyelesaian masalah dengan cara memecah masalah ke dalam sub-sub masalah yang lebih kecil, menyelesaikan sub-sub masalah satu demi satu, kemudian menggabungkan solusi-solusi dari sub-sub masalah tersebut menjadi sebuah solusi lengkap dengan tetap mengingat dan menangani interaksi yang terdapat pada sub-sub masalah tersebut.

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.

2.3. Metode-metode Pencarian

Terdapat banyak metode pencarian yang telah diusulkan. Semua metode yang ada dapat dibedakan ke dalam dua jenis: pencarian buta/tanpa informasi (*blind* atau *un-informed search*) dan pencarian heuristik/dengan informasi (*heuristic* atau *informed search*). Setiap metode mempunyai karakteristik yang berbeda-beda dengan kelebihan dan kekurangannya masing-masing.

Untuk mengukur performansi suatu metode pencarian, terdapat empat kriteria yang digunakan, yaitu [2]:

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

2.3.1. Metode Pencarian Buta *(Blind/Un-informed Search)*

Disini digunakan istilah *blind* atau buta karena memang tidak ada informasi awal yang digunakan dalam proses pencarian [2]. Informasi yang ada hanyalah kondisi *start state* dan *goal state* saja. Dengan seperti ini maka pencarian buta akan mencari seluruh kemungkinan yang ada untuk menemukan solusi, sehingga menjadi kurang efisien.

Beberapa metode yang ada pada *blind search* antara lain: *Breadth First Search* (BFS), *Uniform Cost Search* (UCS), *Depth First Search* (DFS), *Depth-Limited Search* (DLS), *Iterative-Deepening Search* (IDS), dan *Bi-directional Search* (BDS).

2.3.2. Metode Pencarian Heuristik *(Informed Search)*

Kata *heuristic* berasal dari sebuah kata kerja bahasa Yunani, *heuriskein*, yang berarti ‘mencari’ atau ‘menemukan’. Dalam dunia pemrograman, sebagian orang menggunakan kata heuristik sebagai lawan kata dari algoritmik, dimana kata heuristik ini diartikan sebagai ‘suatu proses yang mungkin dapat menyelesaikan suatu masalah tetapi tidak ada jaminan bahwa solusi yang dicari selalu dapat ditemukan’. Di dalam mempelajari metode-metode pencarian ini, kata heuristik diartikan sebagai suatu fungsi yang memberikan suatu nilai berupa biaya perkiraan (estimasi dari suatu solusi) [2].

Metode-metode yang termasuk dalam teknik pencarian yang berdasarkan pada fungsi heuristik adalah: *Generate and Test, Hill Climbing (Simple Hill Climbing dan Steepest-Ascent Hill Climbing), Simulated Annealing, Best First Search(Greedy Best First Search* dan A* dengan berbagai varisinya, seperti *Simplified Memory-Bounded A**).

2.4. Algoritma A* (A Bintang)

Algoritma ini merupakan algoritma *Best First Search* yang menggabungkan *Uniform Cost Search* dan *Greedy Best-First Search*. Biaya yang diperhitungkan didapat dari biaya sebenarnya ditambah dengan biaya perkiraan. Dalam notasi matematika dituliskan sebagai: $f(n) = g(n) + h(n)$. Dengan perhitungan biaya seperti ini,algoritma A* adalah *complete* dan optimal [2].

Sama dengan algoritma dasar *Best First Search*, algoritma A* ini juga menggunakan dua senarai *OPEN* dan *CLOSED*. Terdapat tiga kondisi bagi setiap suksesor yang dibangkitkan, yaitu sudah berada di *OPEN*, sudah berada di *CLOSED*, dan tidak berada di *OPEN* maupun *CLOSED*. Pada ketiga kondisi tersebut diberikan penanganan yang berbeda-beda.

Jika suksesor sudah pernah berada di *OPEN*, maka dilakuakn pengecekan apakah perlu pengubahan *parent* atau tidak tergantung pada nilai *g*-nya melalui *parent* lama atau *parent* baru. Jika melalui *parent* baru memberikan nilai *g* yang lebih kecil, maka dilakukan pengubahan *parent*. Jika pengubahan *parent* dilakukan, maka dilakukan pula pembaruan (*update*) nilai *g* dan *f* pada suksesor tersebut.

Dengan perbaruan ini, suksesor tersebut memiliki kesempatan yang lebih besar untuk terpilih sebagai simpul terbaik (*best node*).

Jika suksesor tidak berada di *OPEN* maupun *CLOSED*, maka suksesor tersebut dimasukkan ke dalam *OPEN*. Tambahkan suksesor tersebut sebagai suksesornya *best node*. Hitung biaya suksesor tersebut dengan rumus $f = g + h$.

2.5. Algoritma *Simplified Memory-Bounded A* (SMA*)*

Untuk masalah tertentu, di mana memori komputer terbatas, algoritma A* mungkin tidak mampu menemukan solusi karena sudah tidak tersedia memori untuk menyimpan simpul-simpul yang dibangkitkan. Algoritma IDA* dapat digunakan untuk kondisi seperti ini karena IDA* hanya membutuhkan sedikit memori. Tetapi, satu kelemahan IDA* adalah bahwa pencarian yang dilakukan secara iteratif akan membutuhkan waktu yang lama karena harus membangkitkan simpul berulang kali [2].

Berlawanan dengan IDA* yang hanya mengingat satu *f-limit*, algoritma SMA* mengingat *f-Cost* dari setiap iterasi sampai sejumlah simpul yang ada di dalam memori. Karena batasan memori dalam jumlah tertentu, maka dapat membatasi pencarian hanya sampai pada simpul-simpul yang dapat dicapai dari root sepanjang suatu jalur yang memorinya masih mencukupi. Kemudian mengembalikan suatu rute terbaik diantara rute-rute yang ada dalam batasan jumlah simpul tersebut. Jika memori komputer hanya mampu menyimpan 100 simpul, maka bisa membatasi proses pencarian sampai level 99.

Pada algoritma SMA* terdapat sebuah senarai yang digunakan untuk memanipulasi antrian simpul yang terurut berdasarkan *f-cost*. Di sini yang dimaksud *f-cost* adalah gabungan biaya sebenarnya dan biaya perkiraan, yang secara matematika dinyatakan seperti pada persamaan 2.1 berikut:

$$f(n) = g(n) + h(n) \qquad (2.1)$$

Dengan:

n = simpul saat ini

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

$h(n)$ = perkiraan cost dari simpul n ke simpul tujuan (nilai *heuristic*)

$f(n)$ = total cost dari simpul n ke simpul tujuan

Berikut ini adalah algoritma *Simplified Memory-Bounded A** [2]:

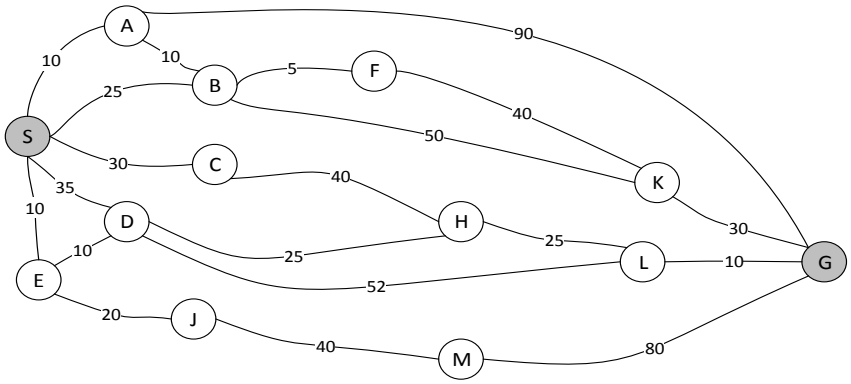
```
function SMA*(masalah) returns solusi
inputs : masalah, sebuah masalah
local variables : Queue, antrian nodes yang terurut berdasarkan
f-cost
Queue ← MAKE-QUEUE({MAKE-SIMPUL(INITIAL-STATE[masalah])})
loop do
      if Queue kosong then return gagal
      n ← simpul di Queue yang memiliki f-cost terkecil dan
level terdalam
      if GOAL-TEST(n) then return sukses
      suk ← NEXT-SUCCESSOR(n)
      if suk bukan goal dan levelnya sudah maksimum then
            f(suk) ← INFINITE
      else
            f(suk) ← MAX(f(n), g(n) + h(n))
      end
      if semua suksesor dari n sudah dibangkitkan then
            Ganti f-cost pada n dengan nilai f(suk) yang
            terkecil. Gantikan nilai f(suk) terkecil ini ke
            semua ancestors dari n(ayah, kakek, dan seterusnya
            keatas) kecuali ancestors yang memiliki f-cost
            lebih kecil daripada f(suk)terkecil itu.
      if semua SUCCESSOR(n) sudah di memori then
            Keluarkan n dari Queue (tetapi tidak dihapus secara
fisik di memori)
      if memori penuh then
            if suk = Goal and f(suk) = f(start) then return
sukses dan exit
            Hapus simpul terburuk di dalam Queue yang memiliki
            f-cost terbesar dan level terdangkal.
            Keluarkan simpul terburuk tersebut dari daftar
            suksesor parent-nya.
            Masukkan parent dari simpul terburuk tersebut ke
            Queue jika parent tersebut tidak ada di Queue.
      end
       insert suk in Queue
```

Gambar 2.2 Algoritma *Simplified Memory-Bounded A***

Pada awalnya, *Queue* berisi *initial state* yang memiliki *f-cost = h(n)* karena *g(n)* = 0 (tidak ada biaya dari *initial state* menuju *initial state*). Selanjutnya, dilakukan iterasi sampai *goal* ditemukan atau sampai *Queue* kosong. Awalnya, *initial state* akan terpilih sebagai *n*. Kemudian, satu per satu, suksesornya dibangkitkan dan dihitung biayanya yang dilambangkan dengan *f(suk)*. Ketika

semua suksesor dari *n* sudah dibangkitkan, maka ganti nilai *f-cost* pada *n* dengan nilai *f-cost* terkecil dari nilai-nilai *f-cost* yang ada pada semua suksesornya. Pergantian ini dilakukan juga untuk semua *ancestor*-nya *n*(ayah, kakek, dst keatas). Tetapi jika *ancestor*-nya *n* tersebut memiliki nilai *f-cost* yang lebih kecil daripada *f-cost* baru pada *n*, maka nilai *f-cost* pada ancestor tersebut tidak perlu diganti. Selanjutnya, jika semua suksesor dari *n* sudah di memori, maka keluarkan *n* dari *Queue*, tetapi tidak dihapus secara fisik dari memori (artinya: biarkan *n* tetap berada di memori). Pengeluaran *n* dari *Queue* ini dilakukan agar *n* tidak terhapus ketika memori *f-cost* terbesar dan level terdangkal (dan jika terdapat lebih dari satu simpul dengan *f-cost* terbesar, maka dipilih simpul yang levelnya terdangkal), maka *n* akan terhapus ketika suksesornya memiliki *f-cost* yang sama dengannya (di mana *f-cost* tersebut adalah yang terbesar di antara semua simpul yang ada di memori). Proses berikutnya adalah penghapusan simpul secara fisik dari memori ketika memori sudah penuh. Simpul yang terpilih akan dihapus dari memori dan perubahan atribut sudah dilakukan pada *parent*-nya, yaitu daftar suksesor yang berisi simpul terhapus tersebut dihilangkan. Kemudian parent dari simpul terhapus tersebut dimasukkan ke dalam *Queue* jika *parent* tersebut tidak ada di *Queue* (artinya: jika *parent* tidak ada di *Queue* berarti *parent* tersebut adalah *n* yang pernah dikeluarkan dari *Queue*).

Misalnya terpadat masalah pencarian rute terpendek seperti pada gambar berikut:

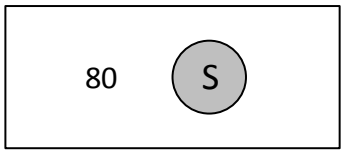


n	S	A	B	C	D	E	F	G	H	J	K	L	M
h(n)	80	80	60	70	85	74	70	0	40	100	30	20	70

Gambar 2.3 Masalah Pencarian Rute

Berikut ini adalah ilustrasi bagaimana langkah-langkah SMA* dalam menyelesaikan masalah yang terdapat pada gambar 2.3 diatas. Misalkan memori komputer hanya mampu menyimpan 6 simpul. Oleh karena itu , level maksimum yang dapat dijangkau oleh SMA* adalah level 5.

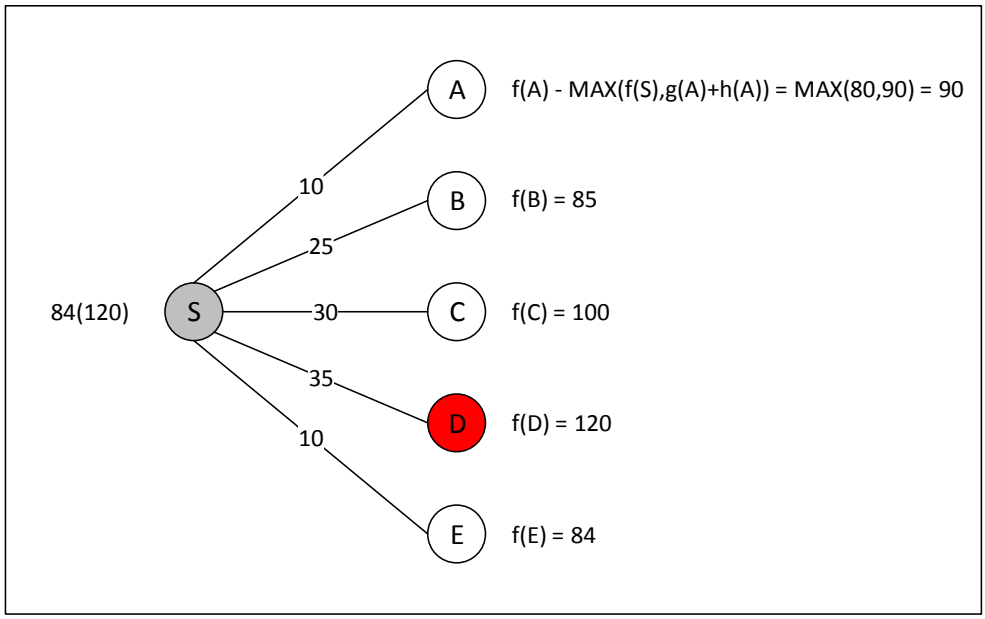
1. Pada langkah pertama kondisi *Queue* = {S}. Pada langkah 1 tersebut seperti pada gambar berikut:



Gambar 2.4 Langkah 1 Pencarian Rute

2. Kemudian langkah selanjutnya seperti berikut:
   a. ***n*** ← S {simpul di *Queue* yang memiliki f-cost terkecil dan level terdalam}
   b. ***suk*** ← A {NEXT-SUCCESSOR(S)}
      hitung f(A) = MAX(f(S), g(A) + h(A)) = MAX(80,90) = 90
      insert A in *Queue*
      maka, *Queue* = {S, A}
   c. ***suk*** ← B {NEXT-SUCCESSOR(S)}
      hitung f(B) = 85
      insert B in *Queue*
      maka, *Queue* = {S, A, B}
   b. ***suk*** ← C {NEXT-SUCCESSOR(S)}
      hitung f(C) = 100
      insert C in *Queue*
      maka, *Queue* = {S, A, B, C}
   c. ***suk*** ← D {NEXT-SUCCESSOR(S)}
      hitung f(D) = 120
      insert D in *Queue*
      maka, *Queue* = {S, A, B, C, D}
   d. ***suk*** ← E { NEXT-SUCCESSOR(S)}
      hitung f(E) = 84

e. Jika semua suksesor dari S sudah dibangkitkan, maka Ganti f-cost pada S dengan nilai f(*suk*) yang terkecil yaitu f(E) = 84. Dengan demikian, f(S) yang tadinya 80 diganti dengan 84.
f. Jika semua SUKSESOR(S) sudah di memori maka, Keluarkan S dari *Queue* (tetapi tidak dihapus secara fisik di memori). Hal ini untuk menghindari terpilihnya S sebagai node yang harus dihapus. Setelah S dikeluarkan dari *Queue*, maka *Queue* = {A, B, C, D}.
g. Jika memori penuh, maka hapus simpul terburuk yang berada **di dalam *Queue***. Di sini terpilih D karena simpul ini adalah yang terburuk di dalam *Queue*, yang memiliki f-cost terbesar dan level terdangkal. Sehingga *Queue* = {A, B, C}. Sebelum simpul D dihapus, simpan nilai f(D) ke dalam simpul parent-nya sebagai history bahwa pernah ada simpul yang dibangkitkan kemudian dihapus. Sehingga pada simpul S terdapat dua nilai yaitu 84(120). Angka yang berada di dalam kurung adalah nilai f(D), nilai f terbaik dari suksesor yang terhapus. Keluarkan simpul terburuk tersebut dari daftar suksesor parent-nya. Dalam hal ini, keluarkan D tersebut dari daftar suksesor S.
h. Masukkan S (parent dari simpul terburuk tersebut) ke *Queue* jika parent tersebut tidak ada di *Queue*. Setelah S dimasukkan ke dalam *Queue*, maka *Queue* = {S, A, B, C}
i. *Insert* E *in Queue*. Sehingga *Queue* = {S, A, B, C, E}
   Untuk langkah 2 tersebut seperti pada gambar berikut:

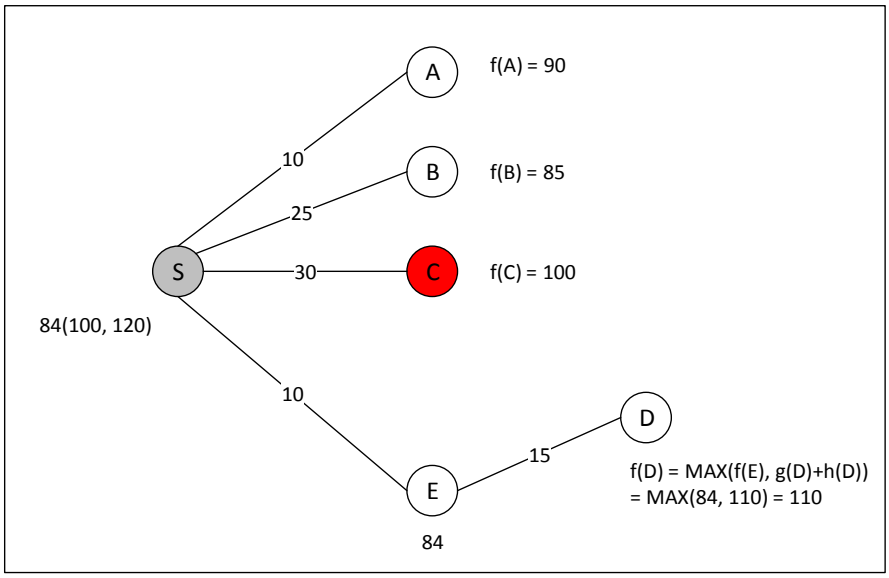


Gambar 2.5 Langkah 2 Pencarian Rute

3. Kemudian langkah 3a seperti berikut:
   a. *n* ← E {simpul di *Queue* yang memiliki f-cost terkecil dan level terdalam}
   b. ***suk*** ← D {NEXT-SUCCESSOR(E)}
   c. Hitung f(D) = 110
   d. Jika suksesor E sudah dibangkitkan (kondisi ini SALAH), sehingga tidak ada aksi yang dilakukan.
   e. Jika semua SUCCESSOR(E) sudah di memori (kondisi ini SALAH), sehingga tidak ada aksi yang dilakukan.
   f. Jika memori penuh (kondisi ini BENAR karena ada 6 simpul: 5 simpul di dalam *Queue* dan simpul D), maka hapus simpul terburuk yang berada **di dalam *Queue***. Di sini terpilih C karena simpul ini adalah yang terburuk di dalam *Queue*, yang memiliki f-cost terbesar dan level terdangkal, yaitu f(C) = 100. Sebelum simpul C dihapus simpan nilai f(C) ke dalam simpul parent-nya sebagai histori. Sehingga pada simpul S terdapat tiga nilai, yaitu 84 (100, 120). Angka yang berada di dalam kurung adalah nilai f(C) dan f(D), dua suksesor yang terhapus. Setelah C dihapus dari *Queue*, maka *Queue* = {S, A, B, E}

g. Keluarkan simpul terburuk tersebut dari daftar suksesor parent-nya. Dalam hal ini, keluarkan C tersebut dari daftar suksesor S. Proses memasukkan parent dari C, yaitu S, ke dalam *Queue* tidak diperlukan karena S sudah berada di dalam *Queue*.

h. *Insert* D *in Queue*. Sehingga *Queue* = {S, A, B, E, D,}
Untuk langkah 3a seperti pada gambar berikut:



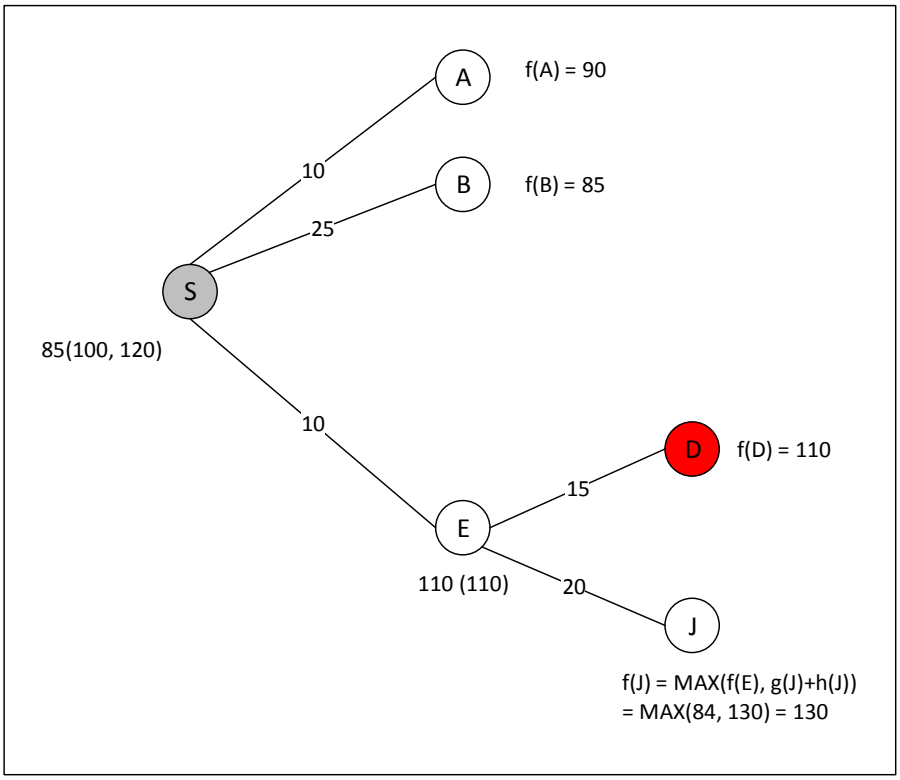
Gambar 2.6 Langkah 3a Pencarian Rute

4. Kemudian langkah 3b seperti berikut
   a. ***suk*** ← J {NEXT-SUCCESSOR(E)}
   b. Hitung f(J) = 130
   c. Jika semua suksesor dari E sudah dibangkitkan (kondisi ini BENAR), maka ganti f-cost pada E dengan nilai f(suk) yang terkecil, yaitu 110. Dengan demikian, f(E) yang awalnya 84 diganti menjadi 110. Nilai f(E) yang baru ini perlu dicek apakah bisa menggantikan nilai parent-nya dan berlaku hingga semua anchestor-nya(ayah, kakek, buyut, dst) dengan cara membandingkan semua sibling (saudara kandung) dari E, yaitu A dan B. Ternyata f(E) lebih besar dibandingkan f(A) = 90 dan f(B) = 85. Sehingga nilai f(E) tidak digunakan untuk mengganti nilai f(S). nilai f(S) yang tadinya

84 digantikan oleh f(B) = 85 yang merupakan nilai terendah diantara semua suksesor S.

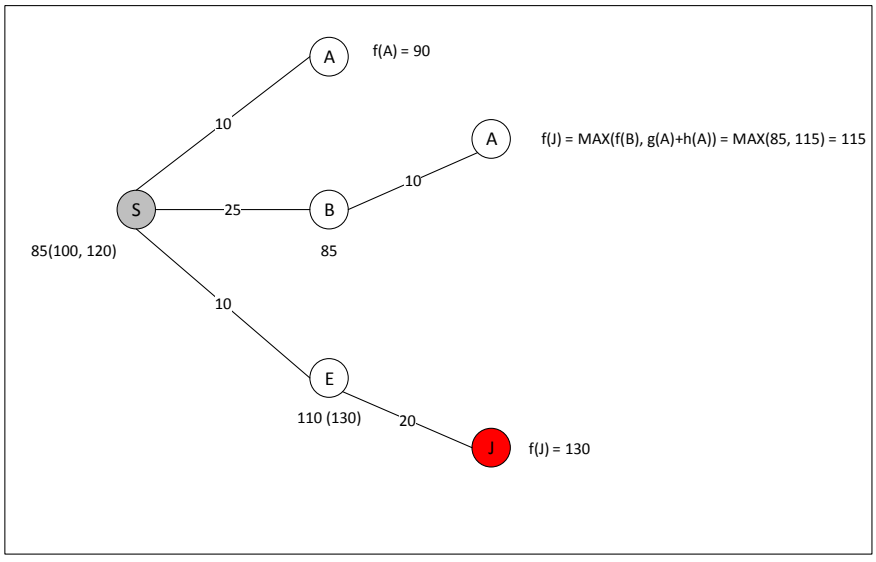
d. Jika semua SUCCESSOR(E) sudah di memori (kondisi ini BENAR), maka Keluarkan E dari *Queue* (tetapi tidak dihapus secara fisik di memori). Setelah E dikeluarkan, maka *Queue* = {S, A, B, D}
e. Jika memori penuh (kondisi ini BENAR karena ada 6 suimpul: 4 di *Queue*, E dan J), maka hapus simpul terburuk yang berada di dalam *Queue*. Di sini terpilih D karena simpul ini adalah yang terburuk di dalam *Queue*, yang memiliki f-cost terbesar dan level terdangkal, yaitu f(D) = 110. Sebelum simpul D dihapus, simpan nilai f(D)ke dalam simpul parent-nya sebagai histori. Sehingga pada simpul E terdapat dua nilai, yaitu 110 (110). Angka yang berada di dalam kurung adalah nilai f(D), suksesor yang terhapus. Setelah D dihapus dari *Queue*, maka *Queue* = {S, A, B}
f. Keluarkan simpul terburuk tersebutdari daftar suksesor parent-nya. Dalam hal ini, keluarkan D tersebutdari daftar suksesor S. Masukkan E ke *Queue*. Sehingga *Queue* = {S, A, B, E}
g. *Insert* J *in Queue*. Sehingga *Queue* = {S, A, B, E, J}

Untuk langkah 3b seperti pada gambar berikut:



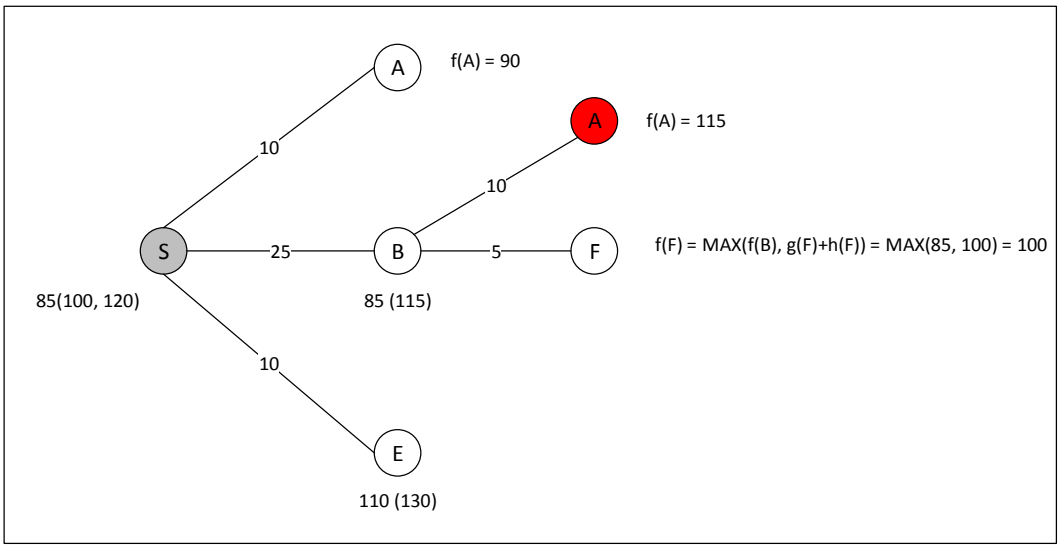
Gambar 2.7 Langkah 3b Pencarian Rute

5. Demikian seterusnya. Langkah 4a, 4b, 4c, hingga langkah 8 memiliki cara yang sama seperti diatas. Untuk gambar dari langkah 4a adalah seperti pada gambar berikut:



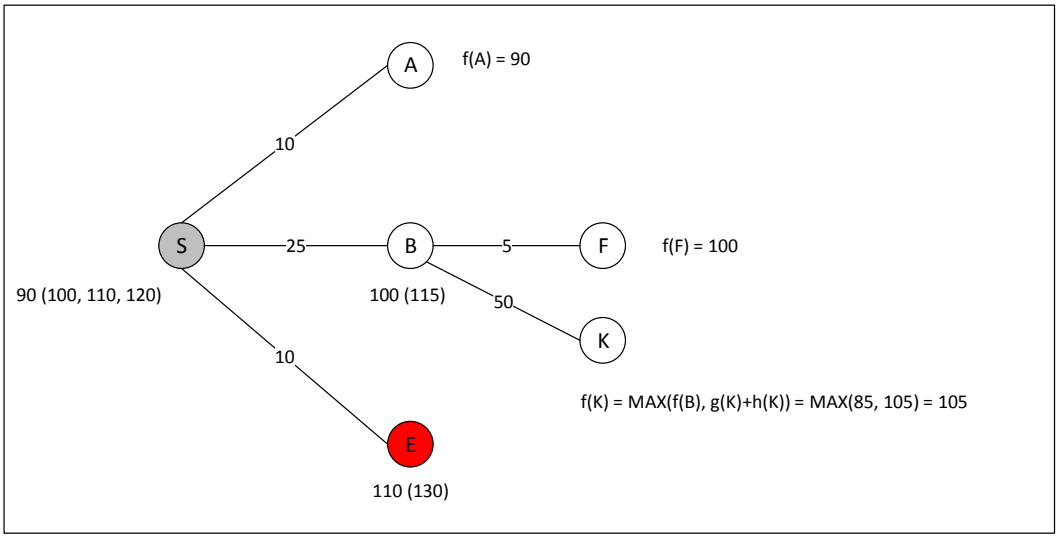
Gambar 2.8 Langkah 4a Pencarian Rute

Langkah 4b adalah seperti pada gambar berikut:



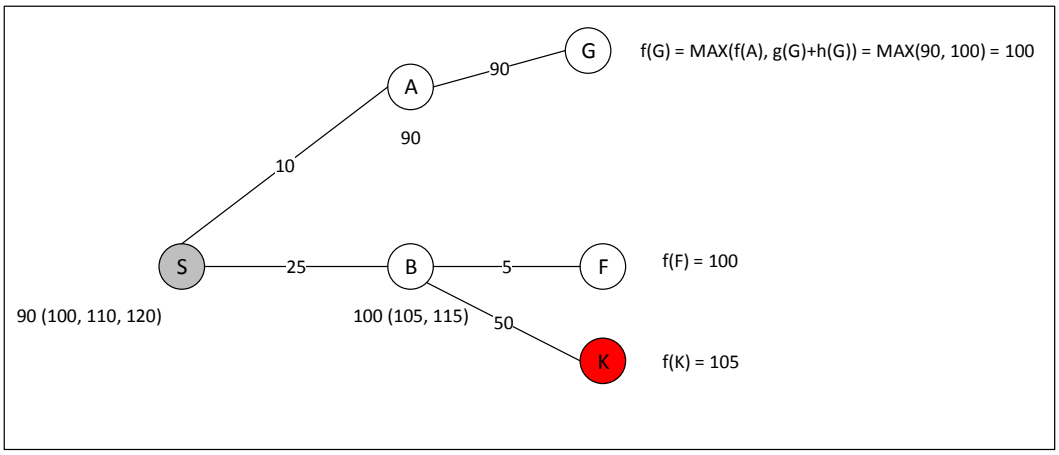
Gambar 2.9 Langkah 4b Pencarian Rute

Langkah 4c adalah seperti pada gambar berikut:



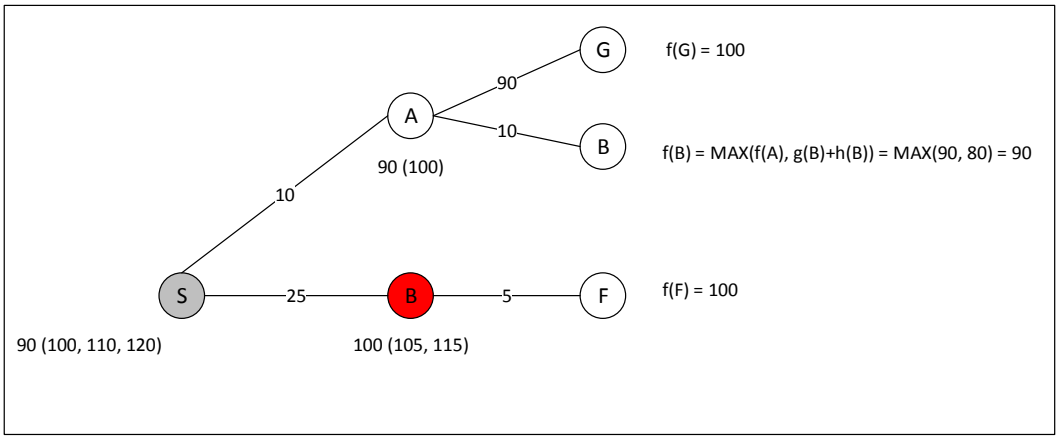
Gambar 2.10 Langkah 4c Pencarian Rute

Langkah 5a adalah seperti pada gambar berikut:



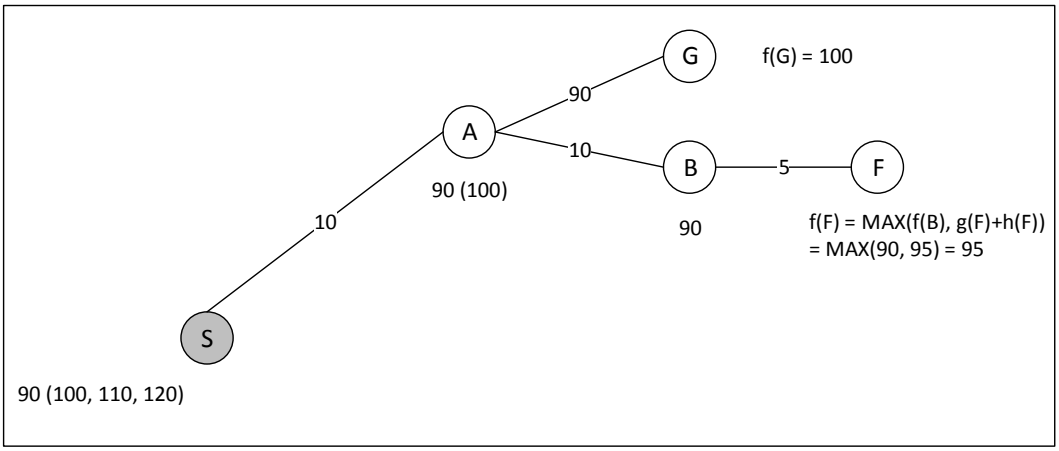
Gambar 2.11 Langkah 5a Pencarian Rute

Langkah 5b adalah seperti pada gambar berikut:



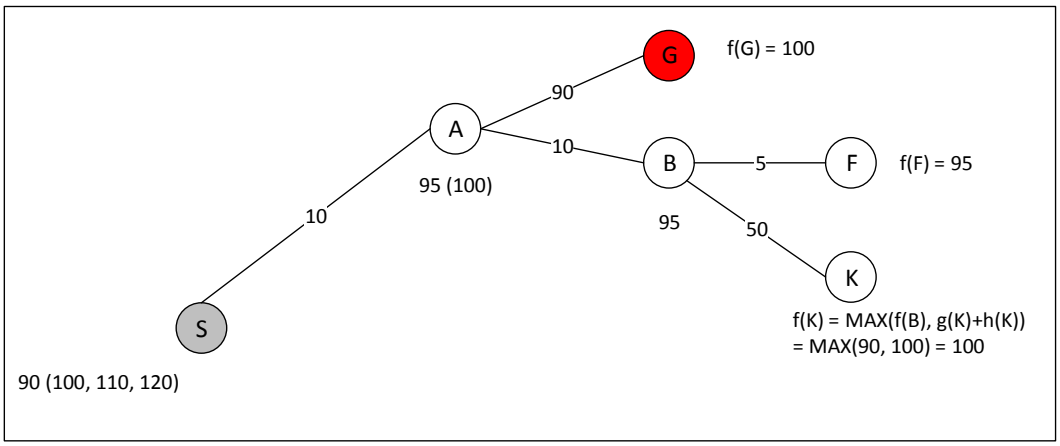
Gambar 2.12 Langkah 5b Pencarian Rute

Langkah 6a adalah seperti pada gambar berikut:



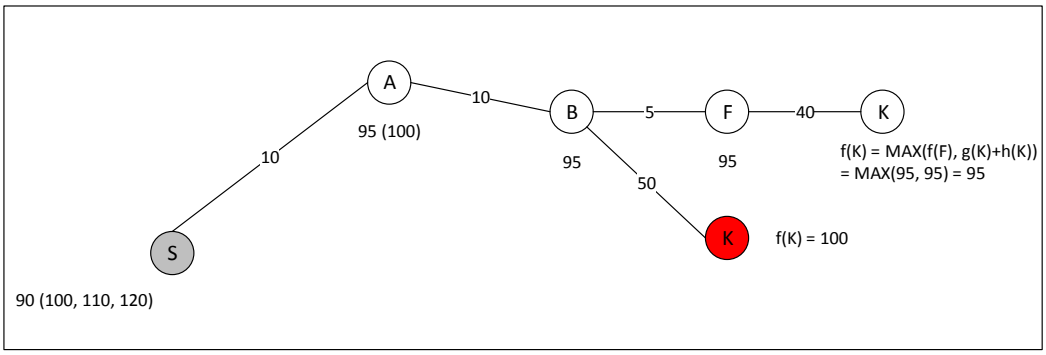
Gambar 2.13 Langkah 6a Pencarian Rute

Langkah 6b adalah seperti pada gambar berikut:



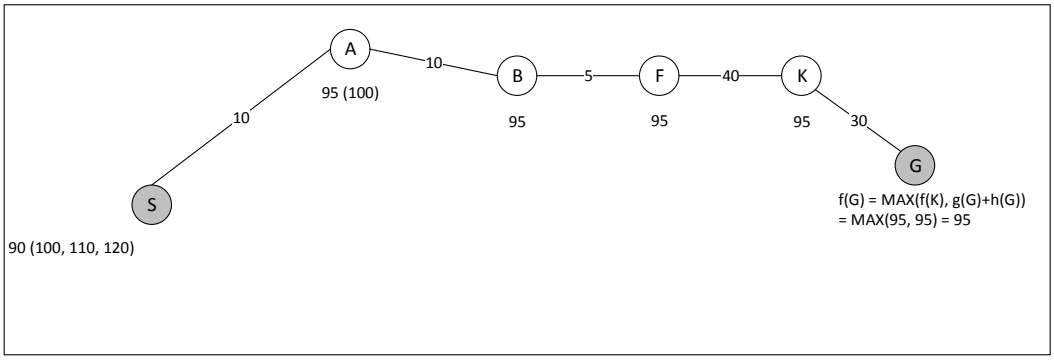
Gambar 2.14 Langkah 6b Pencarian Rute

Langkah 7 adalah seperti pada gambar berikut:



Gambar 2.15 Langkah 7 Pencarian Rute

Langkah 8 adalah seperti pada gambar berikut:



Gambar 2.16 Langkah 8 Pencarian Rute (hasil akhir pencarian)

2.6. Fungsi Heuristik

Di dalam metode yang termasuk heuristic search, fungsi heuristik memainkan peranan yang sangat menetukan. Suatu fungsi dapat diterima sebagai fungsi heuristik jika biaya perkiraan yang dihasilkan tidak melebihi dari biaya sebenarnya. Ketika fungsi heuristik memberikan biaya perkiraan yang melebihi biaya sebenarnya (*overestimate*), maka proses pencarian bisa tersesat dan membuat *heuristic search* menjadi tidak optimal [2].

Suatu fungsi heuristik dikatakan baik jika bisa memberikan biaya perkiraan yang mendekati biaya sebenarnya, fungsi heuristik tersebut semakin baik. Berikut ini adalah contoh fungsi heuristik pada masalah 8-*puzzle* dimana terdapat dua fungsi heuristik yang dapat digunakan, antara lain:

1. *h1* = jumlah kotak yang posisinya salah. Pada masalah tersebut misalnya angka 1, 2, dan 3 sudah berada pada posisi yang benar. Sedangkan lima angka lainnya berada pada posisi yang salah. Jadi *h1* = 5.
2. *h2* = jumlah langkah yang diperlukan masing-masing kotak menuju posisi yang benar di *goal state*. Biasanya disebut *City Block Distance* atau *Manhattan Distance*. Untuk masalah tersebut misalnya angka 1, 2 dan 3 membutuhkan 0 langkah. Angka 4, 5, 7 dan 8 membutuhkan 2 langkah. Sedangkan angka 6 membutuhkan 3 langkah. Sehingga h2 = 0 + 0 + 0 + 2 + 2 + 3 + 2 + 2 = 11.

2.7. Pemrograman Berorientasi Objek

Perkembangan gaya pemrograman mencapai gaya pemrograman orientasi objek setelah era pemrograman terstruktur. Pemrograman berorientasi objek menggantikan pemrograman terstruktur karena mempunyai banyak keunggulan dalam menangani proyek yang luar biasa kompleks. Pemrograman menggunakan bahasa orientasi objek menawarkan fleksibilitas, kegunaulangan dan kemudahan perawatan [7].

Dengan pemrograman berorientasi objek, para pengembang menciptakan blok-blok kode, yang dinamakan objek [7]. Objek-objek ini kemudian digunakan oleh berbagai aplikasi. Suatu saat jika objek mebutuhkan perubahan, para pengembang hanya melakukan perubahan sekali saja dan kemudian ia dengan

mudah akan mewariskannya ke objek-objek lain yang menjadi turunannya. Dengan kepraktisannya, perusahaan-perusahaan perangkat lunak saat ini berusaha mengadopsi teknologi baru ini dan mengintegrasikannya pada aplikasi-aplikasi yang sudah ada. Kenyataannya, kebanyakan aplikasi-aplikasi modern saat ini berorientasi objek.

Paradigma berorientasi objek adalah cara yang berbeda dalam memandang aplikasi-aplikasi [7]. Dengan pendekatan berorientasi objek, para pengembang membagi aplikasi-aplikasi besar menjadi objek-objek, yang mandiri satu terhadap lainnya. Pengembang kemudian mengembangkan aplikasi dengan membuat objek-objek itu saling mengirim pesan (*message*) dan bekerja-sama. Dimbil suatu contoh. Dengan pemrograman terstruktur, katakanlah ingin membuat form dengan *listbox* di dalamnya. Untuk hal ini, maka harus menulis kode-kode yang jumlahnya sangat banyak untuk membuat form dan *listbox* itu sendiri dan kemudian membuat kode-kode untuk tombol OK. Dengan pemrograman berorientasi objek, maka dengan sederhana hanya perlu menggunakan objek-objek yang telah ada sebelumnya: form, *listbox*, dan tombol OK. Maka, alih-alih membuat segalanya dari awal dengan pemrograman terstruktur, dengan pemrograman berorientasi objek tinggal menggunakan objek-objek yang sudah ada dan kemudian berfokus pada ‘apa yang unik’ untuk aplikasi tertentu yang sedang dikembangkan. “*Do not reinventing the wheel!*”, kata banyak ahli. Maka, keunggulan utama mengembangkan objek-objek sekali saja dan menggunakannya lagi (*reusable*) berulang-ulang. Dengan pendekatan berorientasi objek, difokuskan pada informasi dan perilaku yang dimiliki suatu objek sehingga kemudian pengembang dapat mengembangkan sistem/perangkat lunak yang fleksibel dalam menghadapi perubahan-perubahan informasi dan/atau perilaku yang dituntut pengguna.

2.8. UML (*Unified Modelling Language*)

UML (*Unified Modelling Language*) adalah salah satu alat bantu yang sangat handal di dunia pengembangan sistem yang berorientasi objek. Hal ini disebabkan karena UML menyediakan bahasa pemodelan visual yang memungkinkan bagi pengembang untuk membuat cetak biru atas visi mereka dalam bentuk yang baku, mudah untuk dimengerti serta dilengkapi dengan mekanisme

yang efektif untuk berbagi (*sharing*) dan mengkomunikasikan rancangan mereka yang lain [9].

UML merupakan kesatuan dari bahasa pemodelan yang dikembangkan oleh Booch, *Object Modelling Technique* (OMT) dan *Object Oriented Software Engineering* (OOSE) [9]. Metode Booch dari Grady Booch sangat terkenal dengan nama metode *Design Object Oriented*. Metode ini menjadikan proses analisis dan design ke dalam empat tahapan iteratif, yaitu identifikasi kelas-kelas dan objek-objek, identifikasi semantik dari hubungan objek dan kelas tersebut, perincian interface dan implementasi. Keunggulan metode Booch adalah pada detail kayanya dengan notasi dan elemen. Pemodelan OMT yang dikembangkan oleh Rumbaugh didasarkan pada analisis terstruktur dan pemodelan *entity-relationship*. Tahapan utama dalam metodologi ini adalah analisis, design sistem, design objek dan implementasi. Keunggulan metode ini adalah dalam penotasian yang mendukung semua konsep OO. Metode OOSE dari Jacobson lebih memberi penekanan pada *use case*. OOSE memiliki tiga tahapan yaitu membuat model *requirement* dan analisis, design dan implementasi, dan model pengujian (*test model*). Keunggulan metode ini adalah mudah dipelajari karena memiliki notasi yang sederhana namun mencakup seluruh tahapan dalam rekayasa perangkat lunak.

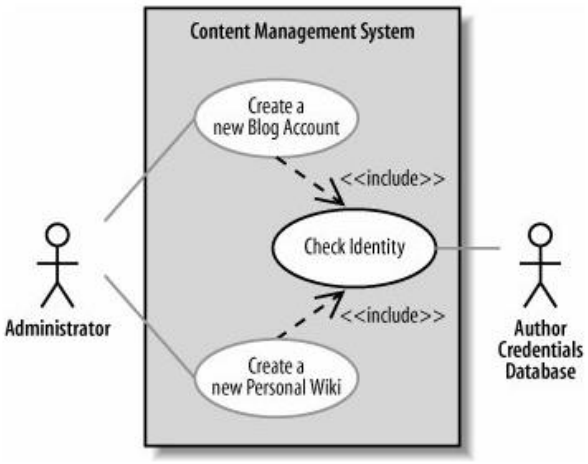
Dengan UML, metode Booch, OMT dan OOSE digabungkan dengan membuang elemen-elemen dari metode lain yang lebih efektif dan elemen-elemen baru yang belum ada pada metode terdahulu sehingga UML lebih ekspresif dan seragam daripada metode lainnya.

2.8.1. *Use Case Diagram*

Use Case adalah deskripsi fungsi dari sebuah sistem dari perspektif pengguna. *Use case* bekerja dengan cara mendeskripsikan tipikal interaksi antara *user* (pengguna) sebuah sistem dengan sistemnya sendiri melalui sebuah cerita bagaimana sebuah sistem dipakai. Urutan langkah-langkah yang menerangkan antara pengguna dan sistem disebut skenario. Setiap skenario mendeskripsikan urutan kejadian. Setiap urutan diinisialisasi oleh orang, sistem lain, perangkat keras atau urutan waktu. Dengan demikian secara singkat biasa dikatakan *use case* adalah

serangkaian skenario yang digabungkan bersama-sama oleh tujuan umum pengguna [9].

Berikut ini adalah contoh *Use Case Diagram*:



Gambar 2.17 *Use Case Diagram* [10]

2.8.2. *Use Case Scenario*

Sebuah diagram yang menunjukkan kasus penggunaan dan aktor dapat menjadi titik awal yang bagus, tetapi tidak memberikan cukup detail untuk desainer sistem untuk benar-benar memahami persis bagaimana kekhawatiran sistem akan dipenuhi. Use Case Skenario adalah cara terbaik untuk mengungkapkan informasi penting ini dalam bentuk penggunaan description setiap kasus berbasis teks harus disertai dengan satu [10].

Berikut ini adalah contoh *Use Case Scenario*:

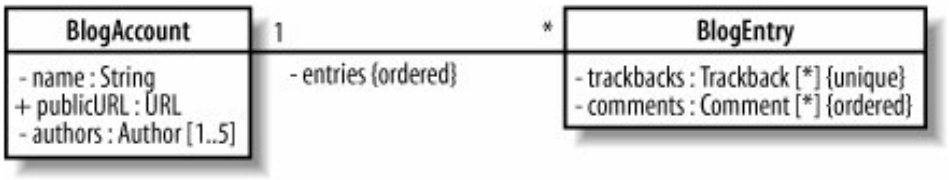
Use case name	**Create a new Blog Account**	
Related Requirements	Requirement A.1.	
Goal In Context	A new or existing author requests a new blog account from the Administrator.	
Preconditions	The system is limited to recognized authors and so the author needs to have appropriate proof of identity.	
Successful End Condition	A new blog account is created for the author.	
Failed End Condition	The application for a new blog account is rejected.	
Primary Actors	Administrator.	
	Secondary Actors	Author Credentials Database.
	Trigger	The Administrator asks the CMS to create a new blog account.
Main Flow	Step	Action
	1	The Administrator asks the system to create a new blog account.
	2	The Administrator selects an account type.
	3	The Administrator enters the author's details.
	4	The author's details are verified using the Author Credentials Database.
	5	The new blog account is created.
	6	A summary of the new blog account's details are emailed to the author.
Extensions	Step	Branching Action
	4.1	The Author Credentials Database does not verify the author's details.
	4.2	The author's new blog account application is rejected.

Gambar 2.18 *Use Case Scenario* [10]

2.8.3. *Class Diagram*

Class Diagram adalah diagram yang digunakan untuk menampilkan beberapa kelas serta paket-paket yang ada dalam sistem/perangkat lunak yang sedang kita kembangkan. Diagram kelas memberikan gambaran/diagram statis tentang sistem/perangkat lunak dan relasi-relasi yang ada di dalamnya [8].

Berikut ini adalah contoh *Class Diagram*:



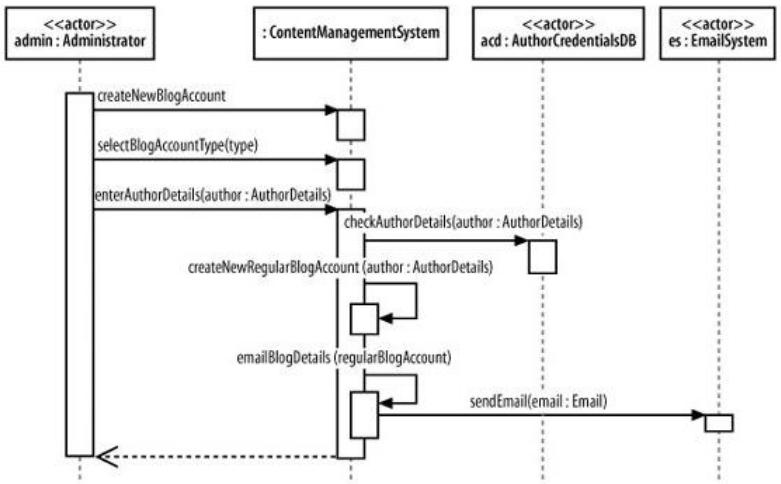
Gambar 2.19 *Class Diagram* [10]

2.8.4. *Sequence Diagram*

Sequence Diagram digunakan untuk menggambarkan perilaku pada sebuah skenario. Diagram ini menunjukkan sejumlah contoh objek dan *message* (pesan) yang diletakkan diantara objek-objek ini dalam *use case*.

Komponen utama *sequence diagram* terdiri atas objek yang dituliskan dengan kotak segiempat bernama. *Message* diwakili oleh garis dengan tanda panah dan waktu yang ditunjukkan dengan *progress* vertikal [9].

Berikut ini adalah contoh *Sequence Diagram*:

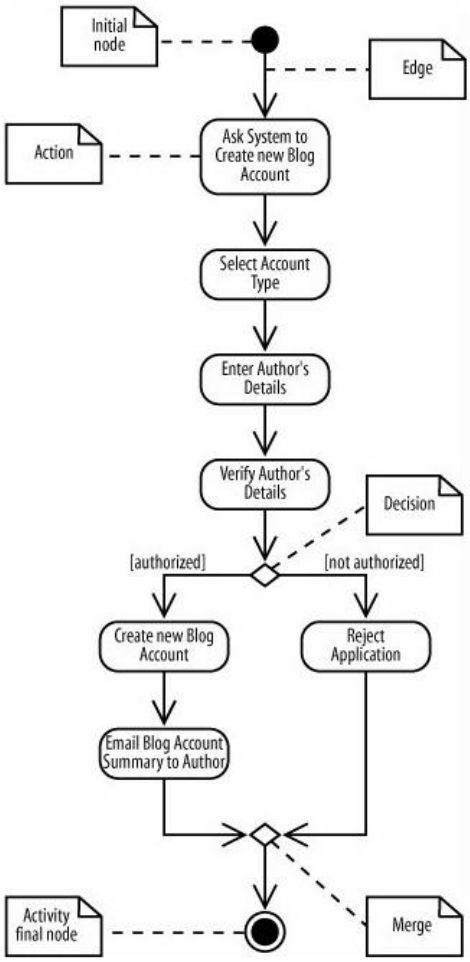


Gambar 2.20 *Sequence Diagram* [10]

2.8.5. *Activity Diagram*

Activity Diagram adalah teknik untuk mendeskripsikan logika prosedural, proses bisnis dan aliran kerja dalam banyak kasus. *Activity Diagram* mempunyai peran seperti halnya *flowchart*, akan tetapi perbedaannya dengan *flowchart* adalah activity diagram mendukung perilaku paralel sedangkan *flowchart* tidak bisa [9].

Berikut ini adalah contoh *Activity Diagram*:



Gambar 2.21 *Activity Diagram* [10]

2.9. Java

Pada 1991, sekelompok insinyur **Sun Microsystem, Inc.**, dipimpin **Patrick Naughton** dan **James Gosling** merancang bahasa untuk perangkat konsumer seperti *cable TV box*. Karena perangkat tidak mempunyai banyak memori, bahasa harus berkuran kecil dan menghasilkan kode yang liat. Karena pabrikan-pabrikan

berbeda memilih pemroses –pemroses berbeda maka bahasa harus netral arsitektur manapun. Proyek diberi nama "**Green**" [6].

Kebutuhan untuk kecil, liat dan netral platform mengantar tim mempelajari implementasi Pascal yang pernah dicoba. Niklaus Wirth, pencipta Pascal telah merancang bahasa kode antara (*intermediate code*) portable untuk mesin hipotesis. Kode antara ini kemudian dapat digunakan di sembarang mesin yang memiliki interpreter. Proyek **Green** menggunakan mesin maya untuk mengatasi isu netral terhadap arsitektur mesin.

Karena orang-orang di proyek **Green** berbasis C++ bukan Pascal maka kebanyakan sintaks diambil dari C++ serta mengadopsi orientasi objek bukan prosedural. Mulanya bahasa yag diciptakan diberi nama "**Oak**", kemudian diganti "**Java**" karena telah ada bahasa yang bernama "Oak".

Produk pertama proyek "**Green**" adalah "*****7", sebuah kendali jauh yang cerdas. Karena pasar belum tertarik dengan prosuk konsumer cerdas maka proyek **Green** harus menemukan pasar lain. Penerapan mengarah menjadi teknologi di *web*. Pada 1995, **Netscape** memutuskan membuat *browser* dilengkapi Java. Setelah itu diikuti **IBM**, **Symantec**, bahkan **Microsoft**.

Sebagai bahasa yang menampung hampir seluruh kemampuan terbaik bahasa pemrograman yang pernah dikembangkan umat manusia, maka bahasa ini memang menjadi tidak mudah, sedikit rumit dan kompleks. Namun, pantas sedikit bersusah agar mampu menguasai salah satu kakas terampuh umat manusia, yaitu bahasa pemrograman *modern*.

Bahasa Java tidak lagi hanya untuk pemanis di web sebagai applet yang membuat **Duke** berdansa. Java adalah kakas, tetap hanya perangkat, bagaimanapun tetap hanya orang hebat yang dapat memberi arti penting kakas seperti dikatakan **James Gosling** , tokoh terpenting di Java:

"*All along, the language was a tool, not the end.*"

Bahasa Java telah disertai berton-ton pustaka kelas, namun tidak perlu mengetahui semuanya. Perlu diketahui sekumpulan yang diperlukan untuk menyelesaikan persoalan serius yang sedang dihadapi.

Bahasa pemrograman orientasi objek telah menjadi aliran utama (mainstream), Java berorientasi objek sejati melebihi C++. Segala sesuatu di Java kecuali sedikit tipe dasar (**int, float, double, char**) adalah objek.

Dibanding bahasa C++, Java lebih memberi kemudahan, antara lain [3]:

1. Perancang Java menghilangkan keperluan dealokasi manual. Java dilengkapi *garbage collector* yang bertugas mendealokasi memori yang tidak diperlukan. Tidak ada lagi upaya pemrogram untuk dispose(). Pemrogram tidak dibebani urusan korupsi dan sampah memori.
2. Java menerapkan *array* sebenarnya, menghilangkan keperluan aritmatika pointer yang berbahaya dan berpeluan besar menyebabkan kesalahan.
3. Menghilangkan keniscayaan operasi penugasan (asignment) sebagai pengujian atas kesamaan di kalimat bersyarat. Di C/C++, pernyataan seperti

   (nPerson = 3) …

   sering menyebabkan masalah, memaksudkan pengujian namun yang terjadi adalah pemberian nilai. Pada kasus ini, kompilator tidak berhak memberi peringatan karena pernyataan itu merupakan pernyataan yang sah.
4. Menghilangkan pewarisan jamak (*multiple inheritance*) diganti fasilitas **interface**. **Interface** memberi banyak manfaat antara lain untuk pewarisan jamak tanpa dibebani kompleksitas yang muncul dari pengelolaan hirarki pewarisan jamak.

2.10. Teknik Pengujian Perangkat Lunak

Ujicoba *software* merupakan elemen yang kritis dari SQA dan merepresentasikan tinjauan ulang yang menyeluruh terhadap spesifikasi,desain dan pengkodean. Ujicoba merepresentasikan ketidaknormalan yang terjadi pada pengembangan *software*. Selama definisi awal dan fase pembangunan, pengembang berusaha untuk membangun *software* dari konsep yang abstrak sampai dengan implementasi yang memungkin [11].

Para pengembang membuat serangkaian uji kasus yang bertujuan untuk ”membongkar” *software* yang mereka bangun. Kenyataannya, ujicoba merupakan salah satu tahapan dalam proses pengembangan *software* yang dapat dilihat (secara psikologi) sebagai destruktif, dari pada sebagai konstruktif.

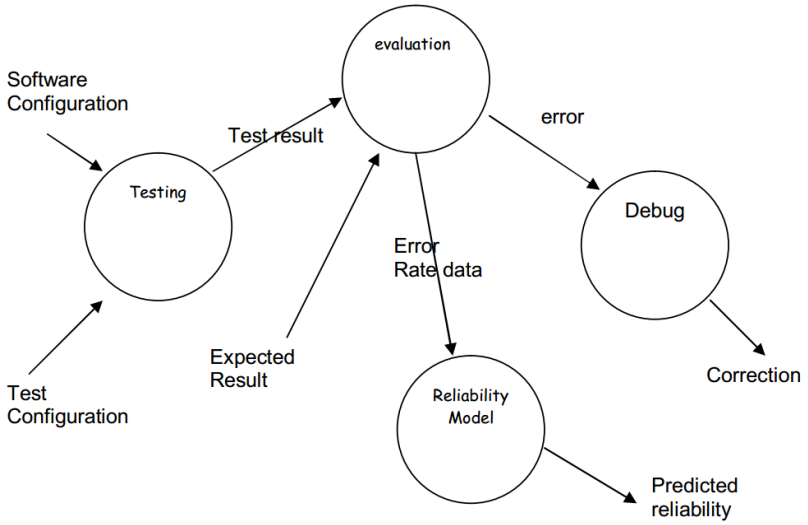
Pengembang *software* secara alami merupakan orang konstruktif. Ujicoba yang diperlukan oleh pengembang adalah untuk melihat kebenaran dari *software* yang dibuat dan konflik yang akan terjadi bila kesalahan tidak ditemukan. Dari sebuah buku, Glen Myers menetapkan beberapa aturan yang dapat dilihat sebagai tujuan dari ujicoba:

1. Ujicoba merupakan proses eksekusi program dengan tujuan untuk menemukan kesalahan
2. Sebuah ujicoba kasus yang baik adalah yang memiliki probabilitas yang tinggi dalam menemukan kesalahan-kesalahan yang belum terungkap
3. Ujicoba yang berhasil adalah yang mengungkap kesalahan yang belum ditemukan

Sehingga tujuan dari ujicoba ini adalah mendesain serangkaian tes yang secara sistematis mengungkap beberapa jenis kesalahan yang berbeda dan melakukannya dalam waktu dan usaha yang minimum. Jika pengujian diselenggarakan dengan sukses, maka akan membongkar kesalahan yang ada didalam perangkat lunak, manfaat lain dari pengujian adalah menunjukkan bahwa fungsi perangkat lunak telah bekerja sesuai dengan spesifikasi, dan kebutuhan fungsi telah tercapai. Sebagai tambahan, data yang dikumpulkan pada saat pengujian dilaksanakan akan menyediakan suatu indikasi keandalan perangkat lunak yang baik dan beberapa indikasi mutu perangkat lunak secara keseluruhan.

2.10.1. Alur Informasi Test (*Test Information Flow*)

Berikut ini adalah gambar alur informasi test (*Test Information Flow*):



Gambar 2.22 *Test Information Flow*

Alur informasi untuk ujicoba mengikuti pola seperti gambar diatas. Dua kategori input yang disediakan untuk proses ujicoba adalah:

1. *Software configuration* yang terdiri dari spesifikasi kebutuhan *software*, spesifikasi desain dan kode sumber;
2. *Test configuration* yang terdiri dari rencana dan prosedur ujicoba, Tools ujicoba apapun yang dapat digunakan, dan kasus ujicoba termasuk hasil yang diharapkan. Pada kenyataannya, konfigurasi ujicoba merupakan subset dari konfigurasi *software*.

2.10.2. Desain Kasus Ujicoba (*Test Case Design*)

Desain ujicoba untuk *software* atau produk teknik lainnya sama sulitnya dengan desain inisial dari produk itu sendiri. Dengan tujuan dari ujicoba itu sendiri yaitu, mendesain ujicoba yang tingkat kemungkinan penemuan kesalahan yang tinggi dengan jumlah waktu dan usaha yang sedikit [11].

Selama beberapa dekade, metode desain ujicoba kasus telah dikembangkan. Metode ini menyediakan pendekatan sistematik untuk ujicoba. Hal yang lebih penting yaitu, metode-metode ini menyediakan mekanisme yang dapat membantu memastikan kelengkapan ujicoba dan menyediakan tingkat kemungkinan yang tinggi dalam penemuan kesalahan pada *software*.

Semua produk yang dikembangkan (engineered) dapat diujicoba dengan salah satu cara dari 2 cara berikut:

1. Mengetahui fungsi-fungsi yang dispesifikasikan pada produk yang didesain untuk melakukannya, ujicoba dapat dilakukan dengan mendemonstrasikan setiap fungsi secara menyeluruh;
2. Mengetahui cara kerja internal dari produk, ujicoba dapat dilakukan untuk memastikan bahwa seluruh operasi internal dari produk dilaksanakan berdasarkan pada spesifikasi dan komponen internal telah digunakan secara tepat.

Pendekatan pertama adalah *black box testing* dan yang kedua adalah *white box testing*. *Black box testing* menyinggung ujicoba yang dilakukan pada interface *software*. Walaupun didesain untuk menemukan kesalahan, ujicoba *black box* digunakan untuk mendemonstrasikan fungsi *software* yang dioperasikan; apakah input diterima dengan benar, dan ouput yang dihasilkan benar; apakah integritas informasi eksternal terpelihara. Ujicoba *black box* memeriksa beberapa aspek sistem, tetapi memeriksa sedikit mengenai struktur logikal internal *software*.

White box testing didasarkan pada pemeriksaan detail prosedural. Alur logikal suatu *software* diujicoba dengan menyediakan kasus ujicoba yang melakukan sekumpulan kondisi dan/atau perulangan tertentu. Status dari program dapat diperiksa pada beberapa titik yang bervariasi untuk menentukan apakah status yang diharapkan atau ditegaskan sesuai dengan status sesungguhnya.

Sepintas seolah-olah *white box* testing akan menghasilkan program yang 100% benar, yang diperlukan hanyalah mendefinisikan alur logikal, membangun kasus uji untuk memeriksa *software* tersebut dan mengevaluasi hasil yang diperoleh. Sayangnya, ujicoba yang menyeluruh ini menghadirkan masalah logikal tertentu. Untuk sebuah program sederhana sekalipun, terdapat banyak alur logikal yang memungkinkan. Sehingga *white box* testing sebaiknya hanya dilakukan pada alur logikal yang penting. Struktur data-struktur data yang penting dapat diujikan dengan uji validitas. Atribut dari black box testing dan *white box* testing dapat dikombinasikan untuk digunakan bersama.

2.10.3. *White-Box Testing*

Ujicoba *white-box* merupakan metode desain uji kasus yang menggunakan struktur kontrol dari desain prosedural untuk menghasilkan kasus-kasus uji. Dengan menggunakan metode ujicoba *white-box*, para pengembang *software* dapat menghasilkan kasus-kasus uji yang [11]:

1. Menjamin bahwa seluruh independent paths dalam modul telah dilakukan sedikitnya satu kali,
2. Melakukan seluruh keputusan logikal baik dari sisi benar maupun salah,
3. Melakukan seluruh perulangan sesuai batasannya dan dalam batasan operasionalnya
4. Menguji struktur data internal untuk memastikan validitasnya

Mengapa menghabiskan banyak waktu dan usaha dengan menguji logikal *software*? Hal ini dikarenakan sifat kerusakan alami dari *software* itu sendiri, yaitu:

1. Kesalahan logika dan kesalahan asumsi secara proposional terbalik dengan kemungkinan bahwa alur program akan dieksekusi. Kesalahan akan selalu ada ketika mendesain dan implementasi fungsi, kondisi atau kontrol yang keluar dari alur utama. Setiap harinya pemrosesan selalu berjalan dengan baik dan dimengerti sampai bertemu ”kasus spesial” yang akan mengarahkannya kepada kehancuran.
2. Sering percaya bahwa alur logikal tidak akan dieksekusi ketika dikenyataannya, mungkin akan dieksekusi dengan basis regular. Alur logika program biasanya berkebalikan dari intuisi, yaitu tanpa disadari asumsi mengenai alur kontrol dan data dapat mengarahkan pada kesalahan desain yang tidak dapat terlihat hanya dengan satu kali ujicoba.
3. Kesalahan typographical (cetakan) bersifat random. Ketika program diterjemahkan kedalam kode sumber bahasa pemrograman, maka akan terjadi kesalahan pengetikan. Banyak yang terdeteksi dengan mekanisme pemeriksaan sintaks, tetapi banyak juga yang tidak terdeteksi sampai dengan dimulainya ujicoba.

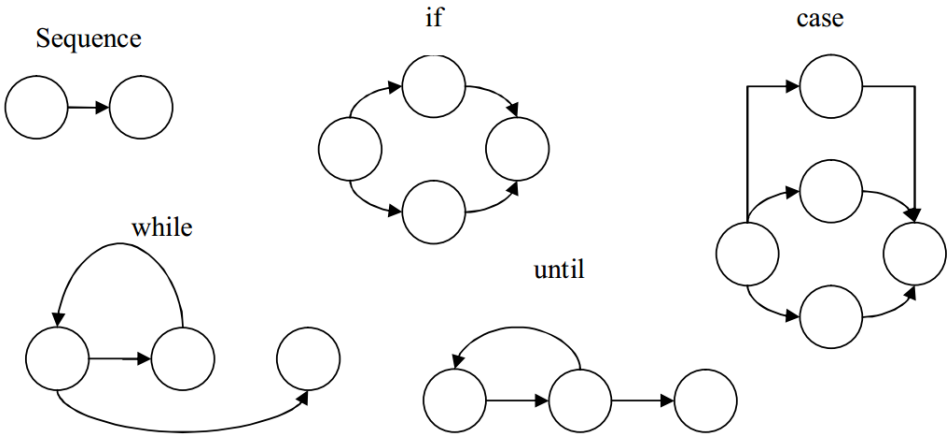
Karena alasan tersebut diatas, maka ujicoba *white box* testing diperlukan selain *black box* testing.

Ujicoba Berbaisi Alur (*Basis Path Testing*)

Ujicoba berbasis alur merupakan teknik ujicoba *white box* pertama yang diusulkan oleh Tom McCabe. Metode berbasis alur memungkinkan perancang kasus uji untuk menghasilkan ukuran kompleksitas logikal dari desain prosedural dan menggunakan ukuran ini untuk mendefinisikan himpunan basis dari alur eksekusi. Kasus uji dihasilkan untuk melakukan sekumpulan basis yang dijamin untuk mengeksekusi setiap perintah dalam program, sedikitnya satu kali selama ujicoba [11].

Notasi graf Alur (*Path Graph Notation*)

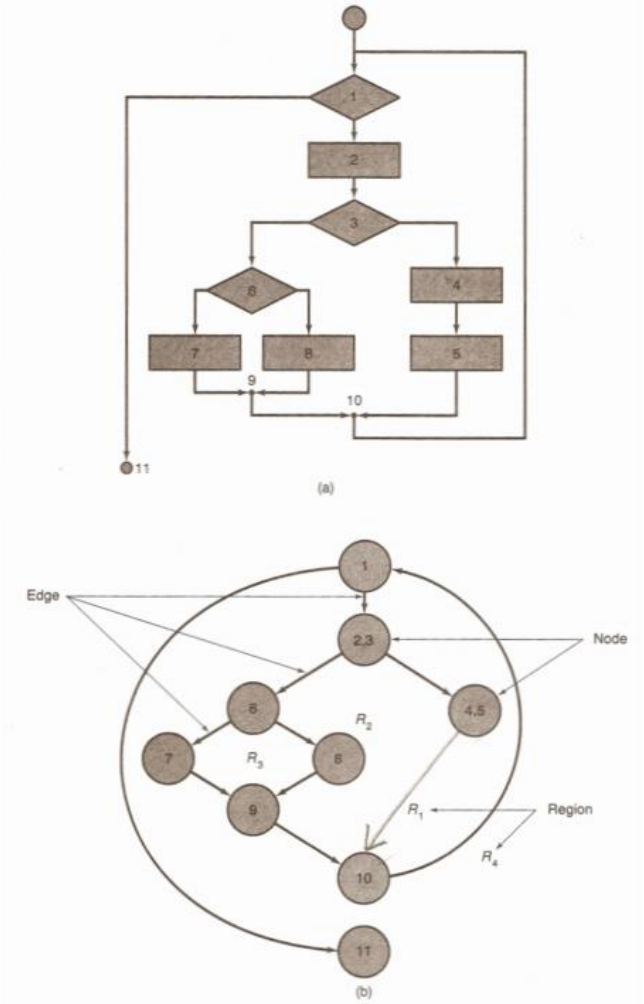
Notasi sederhana untuk merepresentasikan alur kontrol disebut graf alur (*flow graph*), seperti gambar dibawah ini:



Gambar 2.23 *Flow Graph Notation*

Untuk mengilustrasikan kegunaan dari diagram alir dapat dilihat pada gambar dibawah ini. Gambar bagian (a) digunakan untuk menggambarkan struktur kontrol program, sedangkan gambar bagian (b) setiap lingkaran disebut dengan *flow graph node*, merepresentasikan satu atau lebih perintah prosedural. Urutan dari simbol proses dan simbol keputusan dapat digambarkan menjadi sebuah *node*, sedangkan anak panah disebut *edges*, menggambarkan aliran dari kontrol sesuai dengan diagram alir.

Sebuah edge harus berakhir pada sebuah *node* walaupun tidak semua *node* merepresentasikan perintah prosedural. Area yang dibatasi oleh *edge* dan *node* disebut *region*, area diluar *graph* juga dihitung sebagai *region*.



Gambar 2.24 ***Flow Chart(a) dan Flow Graph(b)***

Cyclomatic Complexity

Cyclomatic complexity merupakan *software* metric yang menyediakan ukuran kuantitatif dari komplesitas logikal suatu program. Ketika digunakan dalam konteks metode ujicoba berbasis alur, nilai yang dikomputasi untuk kompleksitas *cyclomatic* mendefinisikan jumlah independent path dalam himpunan basis suatu program dan menyediakan batas atas untuk sejumlah ujicoba yang harus dilakukan untuk memastikan bahwa seluruh perintah telah dieksekusi sedikitnya satu kali.

Independent path adalah alur manapun dalam program yang memperkenalkan sedikitnya satu kumpulan perintah pemrosesan atau kondisi baru. Contoh *independent path* dari gambar flow graph diatas:

Path 1 : 1 – 11

Path 2 : 1 – 2 – 3 – 4 – 5 – 10 – 1 – 11

Path 3 : 1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11

Path 4 : 1 – 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11

Misalkan setip *path* yang baru memunculkan edge yang baru, dengan path:

1 - 2 – 3 – 4 – 5 – 10 - 1 - 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11

path diatas tidak dianggap sebagai independent path karena kombinasi path diatas telah didefinisikan sebelumnya Ketika ditetapkan dalam graf alur, maka independent path harus bergerak sedikitnya 1 *edge* yang belum pernah dilewati sebelumnya. Kompleksitas *cyclomatic* dapat dicari dengan salah satu dari 3 cara berikut :

1. Jumlah *region* dari graf alur mengacu kepada komplesitas cyclomatic
2. Kompleksitas cyclomatic untuk graf alur G didefinisikan:

   V(G) = E – N + 2

   Dimana E = jumlah edge, dan N = jumlah node
3. Kompleksitas cyclomatic untuk graf alur G didefinisikan:

   V(G) = P + 1

   Dimana P = jumlah predicates nodes

Berdasarkan *flow graph* gambar (b) diatas, maka kompleksitas *cyclomatic*-nya dapat di hitung sebagai berikut:

1. *Flow graph* diatas mempunyai 4 *region*
2. V(G) = 11 edges – 9 nodes + 2 = 4
3. V(G) = 3 predicates nodes + 1 = 4

Hasil kompleksitas *cyclomatic* menggambarkan banyaknya *path* dan batas atas sejumlah ujicoba yang harus dirancang dan dieksekusi untuk seluruh perintah dalam program.