Search Strategies

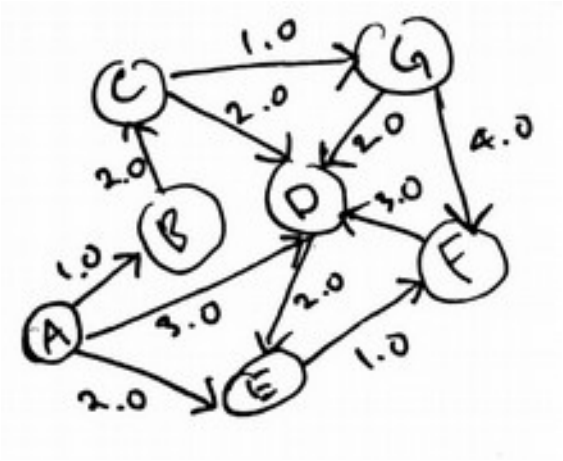
Search strategies adalah metode untuk menyelesaikan masalah.

Untuk melakukan *problem solving* sebelumnya kita harus melakukan dua hal yaitu : *Goal Formulation,* dan *Problem Formulation*.
- *Goal Formulation* untuk menetapkan tujuan yang ingin kita capai.
- *Problem Formulation* untuk mendefinisikan masalah agar lebih mudah untuk dipecahkan.

Dalam *Problem Formulation* ada 5 tahap untuk menyelesaikan masalah,
- *The initial state* : Kondisi awal.
- *Actions* : Hal yang bisa dilakukan untuk menyelesaikan masalah.
- *Transition model* : Deskripsi dari *Actions*
- *Goal test* : Mengkonfirmasi jika hasil dari solusi merupakan *goal*.
- *Path cost* : Biaya dari solusi untuk masalah.

Ada 2 jenis *search strategies* yang dapat digunakan untuk *problem solving*, *Uniformed search,* dan *Informed search*. Untuk *uninformed search* ada 5 strategi, *Breadth-first search (BFS)*, *Uniform-cost search (UCS)*, *Depth-first search (DFS)*, *Depth-limited search (DLS),* dan *Iterative-deepening search (IDS).* Sedangkan untuk *informed search* ada 2 strategi Greedy, dan *A*.*

Contoh soal:



Langkah-langkah BFS :
1. Tentukan *root node*, dan masukkan kepada *queue*
2. Kunjungi *node* pertama di *queue*
3. Cek jika *node* merupakan *goal*
4. Jika iya maka BFS selesai, jika tidak masukkan *child node* kedalam *queue* sesuai dengan urutan abjad. Lalu ulangi dari langkah 2.

Penyelesaian Soal untuk menemukan *node* F sesuai dengan langkah-langkah BFS :

1. Queue = {A}
2. Visit A, Queue = {B^A, D^A, E^A}
3. Visit B^A, Queue = { D^A, E^A, C^B }
4. Visit D^A, Queue = { E^A, C^B , E^D }
5. Visit E^A, Queue = { C^B , E^D, F^E }
6. Visit C^B, Queue = { E^D , F^E, D^C, G^C }
7. Visit E^D, Queue = { F^E , D^C , G^C , F^E }
8. Visit F^E, found *node* F.

Path : A-E-F Visited: A, B, C, D, E, F

Langkah-langkah DFS :

1. Tentukan *root node*, dan masukkan kepada *queue*
2. Kunjungi *node* terakhir di *queue*
3. Cek jika *node* merupakan *goal*
4. Jika iya maka DFS selesai, jika tidak masukkan *child node* kedalam *queue* sesuai dengan urutan abjad terbesar ke yang kecil. Lalu ulangi dari langkah 2.

Penyelesaian Soal untuk menemukan *node* F sesuai dengan langkah-langkah DFS :

1. Queue = {A}
2. Visit A, Queue = { E^A, D^A, B^A }
3. Visit B^A, Queue = { E^A, D^A, C^A }
4. Visit C^A, Queue = { E^A, D^A, G^C, D^C }
5. Visit D^C, Queue = { E^A, D^A, G^C, E^D }
6. Visit E^D, Queue = { E^A, D^A, G^C, F^E }
7. Visit F^E, found *node* F.

Path : A-B-C-D-E-F Visited: A, B, C, D, E, F

Langkah-langkah UCS :

1. Tentukan *root node*, dan masukkan kepada *queue*
2. Kunjungi *node* pertama di *queue*
3. Cek jika *node* merupakan *goal*
4. Jika iya maka UCS selesai, jika tidak masukkan *child node* kedalam *queue* sesuai dengan *total cost* dari *root node* ke *node* yang bisa di *visit*. Urut dari *total cost* terkecil hingga terbesar. Lalu ulangi dari langkah 2. Contoh perhitungan *total cost* missal dari *node* A-E melalui D, *cost* adalah 5 karena A-D memiliki *cost* 3 dan D-E memiliki *cost* 2, sehingga *total cost* = 3 + 2 = 5.

Penyelesaian Soal untuk menemukan *node* F sesuai dengan langkah-langkah UCS:

1. Queue = {A}
2. Visit A, Queue = { B^A, E^A, D^A } //1 ,2 , 3
3. Visit B^A, Queue = { E^A, D^A, C^D } // 2, 3, 3
4. Visit E^A, Queue = { D^A , C^D , F^E } // 3, 3, 3
5. Visit D^A, Queue = { C^D , F^E , E^D } // 3, 3, 5
6. Visit C^D, Queue = { F^E , G^C , E^D , D^C } // 3, 5, 5, 5
7. Visit F^E, found *node* F.

Path : A-E-F Visited: A, B, C, D, E, F

<u>Given *Heuristic Cost* :</u>

h(A) = 3.0 h(E) = 5.0
h(B) = 4.0 h(F) = 1.0
h(C) = 2.0 h(G) = 0.0
h(D) = 1.0

Langkah-langkah *Greedy* :

1. Tentukan *root node*, dan masukkan kepada *queue*
2. Kunjungi *node* pertama di *queue*
3. Cek jika *node* merupakan *goal*
4. Jika iya maka *Greedy* selesai, jika tidak masukkan *child node* kedalam *queue* sesuai dengan *heuristic cost node*. Urut dari *heuristic cost* terkecil hingga terbesar. Lalu ulangi dari langkah 2.

Penyelesaian Soal untuk menemukan *node* F sesuai dengan langkah-langkah *Greedy*:

1. Queue = {A}
2. Visit A, Queue = { D^A, B^A, E^A } //1 ,4 , 5
3. Visit D^A, Queue = { B^A, E^A , E^D } // 4, 5, 5
4. Visit B^A, Queue = { C^B , E^A , E^D } // 2, 5, 5
5. Visit C^B, Queue = { G^C , D^C , E^A , E^D } // 0, 1, 5, 5
6. Visit G^C, Queue = { D^C , F^G , E^A , E^D } // 1, 1, 5, 5
7. Visit D^C, Queue = { F^G , E^A , E^D, E^D } // 1, 5, 5, 5
8. Visit F^G, found *node* F.

Path : A-B-C-G-F Visited: A, B, C, D, G

Langkah-langkah *A** :

1. Tentukan *root node*, dan masukkan kepada *queue*
2. Kunjungi *node* pertama di *queue*

3. Cek jika *node* merupakan *goal*
4. Jika iya maka *A** selesai, jika tidak masukkan *child node* kedalam *queue* sesuai dengan jumlah *path cost* dengan *heuristic cost node*. Urut dari jumlah terkecil hingga terbesar. Lalu ulangi dari langkah 2.

Penyelesaian Soal untuk menemukan *node* F sesuai dengan langkah-langkah *A**:

1. Queue = {A}
2. Visit A, Queue = { D^A, B^A, E^A } //4 ,5 , 7
3. Visit D^A, Queue = { B^A, E^A, E^D } // 5, 7, 7
4. Visit B^A, Queue = { C^B, E^A, E^D } // 3, 7, 7
5. Visit C^B, Queue = { G^C, D^C, E^A, E^D } // 1, 3, 7, 7
6. Visit G^C, Queue = { D^C, F^G, E^A, E^D } // 3, 5, 7, 7
7. Visit D^C, Queue = { F^G, E^A, E^D, E^D } // 3, 7, 7, 7
8. Visit F^G, found *node* F.

Path : A-B-C-G-F Visited: A, B, C, D, G

Local Search

Local search biasanya menggunakan satu node di satu titik dan untuk mencari goal, dengan kondisi node akan bergerak hanya ke titik tetangga node tersebut. Titik yang dikunjungi oleh node pada saat *search* tidak disimpan.
Local Search memiliki 2 keuntungan,
- membutuhkan sedikit memori
- dapat menemukan solusi walaupun jumlah titik infinit.

Local search algorithm biasa digunakan untuk mencari solusi terbaik, dengan tujuan menemukan *global maximum* dari *objective function*.
Algoritma *local search* bisa dibagi menjadi dua tipe bedasarkan hasil yang dicari,
- algoritma yang sempurna pasti akan mencari hasil jika ada.
- algoritma yang optimal pasti akan mencari global maksimum atau global minimum.

Hill Climbing adalah salah satu cara untuk mencari titik global maksimum. Caranya dimulai dengan node di letakkan di titik secara acak dalam *state space* lalu node bergerak kearah *uphill* untuk menemukan *peak*, *peak* adalah titik dalam *objective function* yang lebih tinggi dari tetangganya. Karena node diletakkan secara acak dalam *state space* bisa saja node langsung menemukan *global maximum* tapi memungkinkan juga node berada di titik terendah. Pada akhirnya titik dengan nilai tertinggi adalah solusi terbaik akan terpilih.

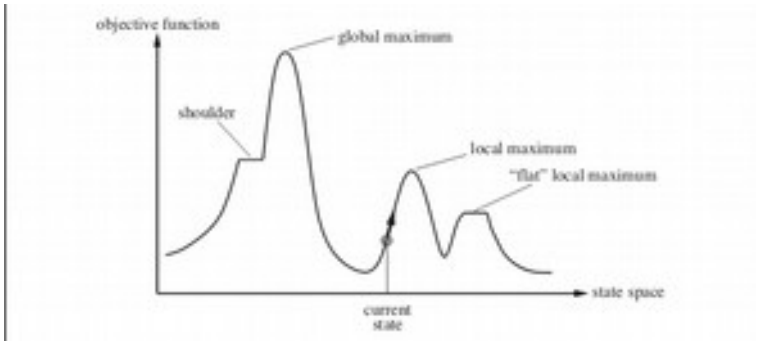
Masalah dalam *Hill Climbing*,
- *Local Maximum, hill climbing* menentukan hasil terbaik bedasarkan *peak* yang ditemukan, jika menemukan *local maximum* algoritma ini akan mengasumsikan bahwa *local maximum* adalah solusi terbaik.
- *Ridge*, sama seperti masalah dengan *local maximum* saat menemukan *ridge* algoritma *hill climbing* akan mengasumsikan bahwa *local maximum* dalam *ridge* adalah solusi terbaik.

Contoh *Ridge*

Sumber : https://slideplayer.com/slide/1512455/5/images/19/Hill-climbing+drawbacks.jpg

- *Plateau / Shoulder*, saat di area datar node tidak bisa bergerak karena tidak ada perbedaan pada titik sekarang, dan titik tetangga node.

Contoh Masalah



Sumber : https://i.stack.imgur.com/HISbC.png

Solusi untuk masalah-masalah *Hill Climbing*,
- *Backtrack* ke node sebelumnya
- Pindah ke bagian lain dari *state space*

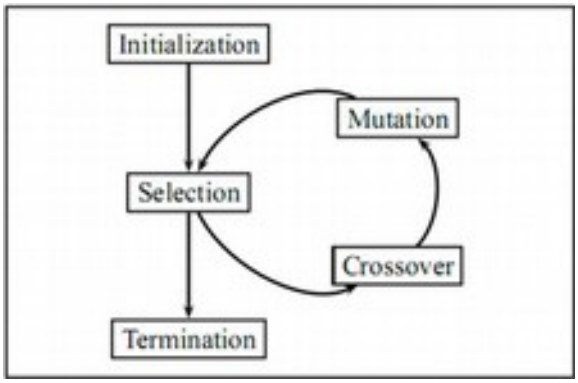
Constraint Satisfaction Problem
Dalam CSP, *local search* adalah metode yang kurang sempurna untuk menemukan solusi. Maka *local search* harus dilakukan berulang-ulang sampai memenuhi aturan yang didefinisikan oleh CSP. Dalam CSP ada beberapa pengertian seperti,
- *State* dapat didefinisikan sebagai *value* dari beberapa atau semua variabel.
- *Goal Test* adalah kumpulan aturan untuk spesifikasikan kombinasi nilai.

Algoritma *Hill Climbing* mewajibkan semua variabel diisi, untuk menerapkan CSP,
- variabel yang tidak diisi harus diperbolehkan.
- memperbolehkan operator menetapkan ulang isi variabel.

Seleksi nilai bedasarkan *heuristic* konflik terkecil, pilih nilai yang melanggar paling sedikit jumlah aturan

Genetic Algorithm, algoritma ini menggabungkan dua genetik orang tua menjadi dua genetik yang baru, dan lebih baik.

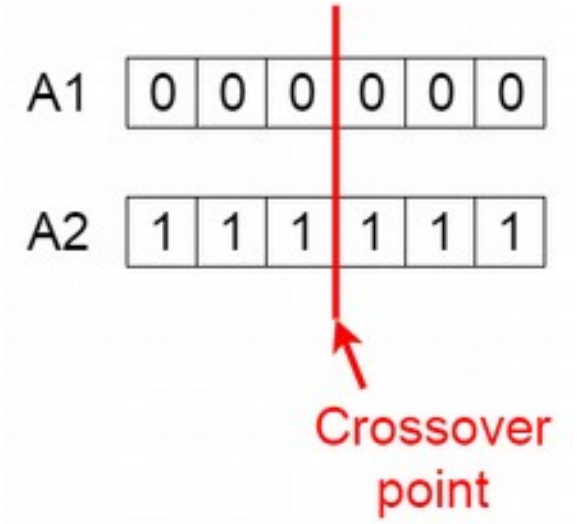
Proses *Genetic Algorithm*



Sumber : https://conversionxl.com/wp-content/uploads/2016/12/Genetic-Algorithm-Tree-Basic-steps-of-GA-selection-crossover-and-mutation-568x374.jpg

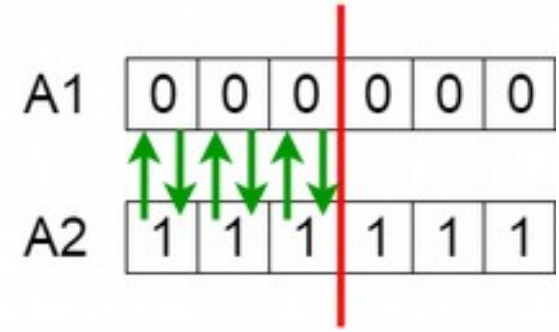
Untuk melakukan *genetic algorithm*, genetik pertama diseleksi bedasarkan *fitness*. Lalu dilakukan *crossover* dengan ditentukan titik crossover, crossover bertujuan untuk menukar genetik orang tua sampai dengan titik crossover.

Crossover Point Genetic Algorithm



*Sumber : https://cdn-images-1.medium.com/max/800/1*Wi6ou9jyMHdxrF2dgczz7g.png*

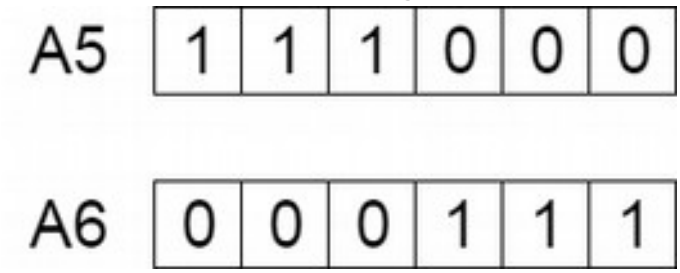
Crossover Genetic Algorithm



*Sumber : https://cdn-images-1.medium.com/max/800/1*eQxFezBtdfdLxHsvSvBNGQ.png*

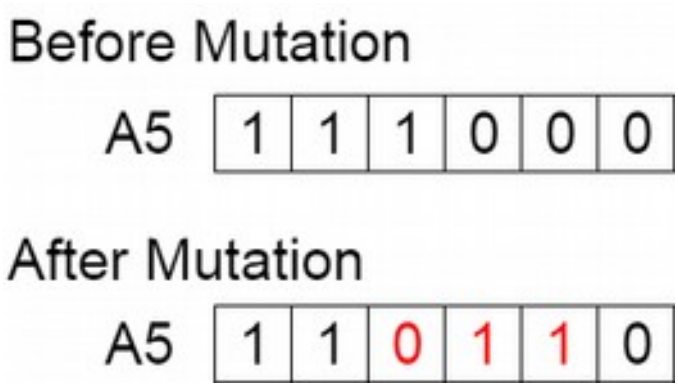
Setelah genetik baru telah dihasilkan, *mutation* dapat terjadi pada hasil genetik yang baru dengan probabilitas yang kecil. Ini bertujuan agar ada diversitas dalam hasil genetika, dan mencegah kesamaan genetika yang terlalu dini.

Hasil *Genetic Algorithm*



*Sumber : https://cdn-images-1.medium.com/max/800/1*_Dl6Hwkay-UU24DJ_oVrLw.png*

Mutation Genetic Algorithm



*Sumber : https://cdn-images-1.medium.com/max/800/1*CGt_UhRqCjIDb7dqycmOAg.png*

Jika hasil genetik sudah menemui kesamaan atau konvergensi maka algoritma akan berhenti, karena sudah menghasilkan satu set hasil-hasil yang berbeda dengan genetik generasi sebelumnya.

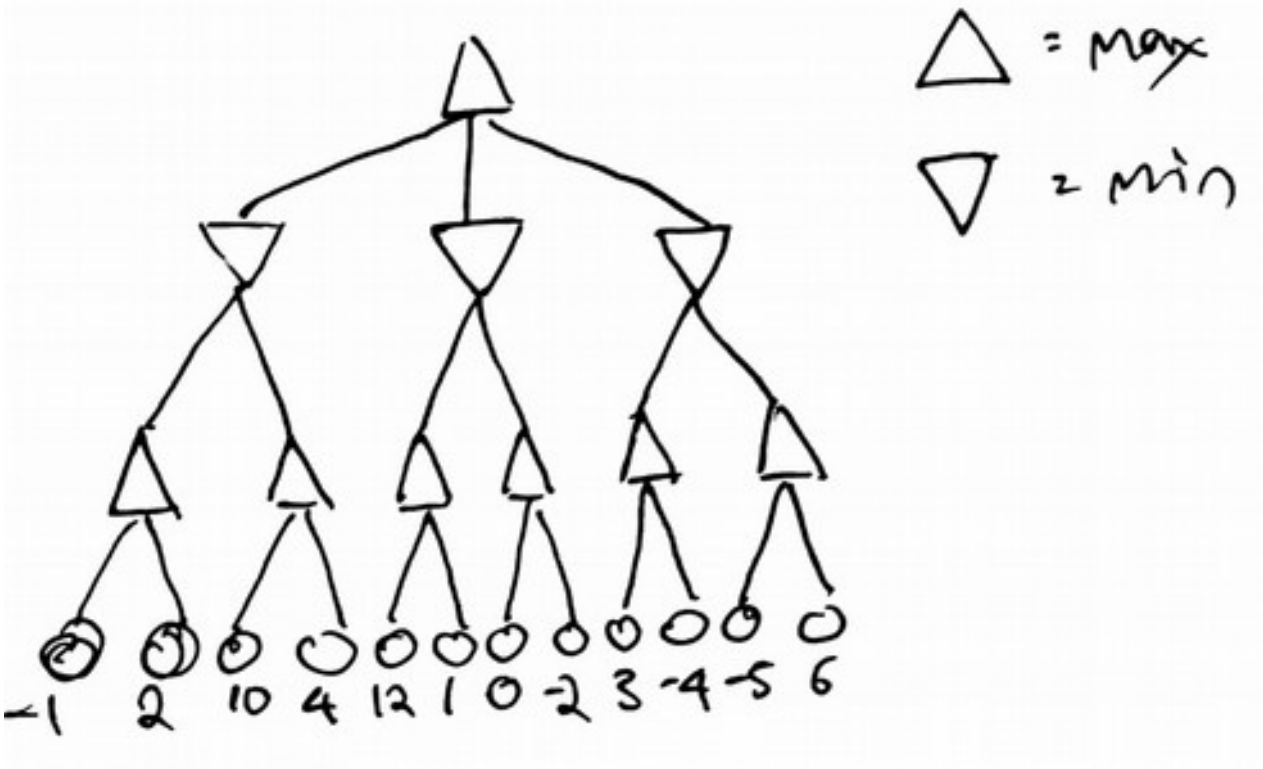
Adversarial Search

Minimax Strategy
Dalam *minimax strategy* cara visualisasinya adalah dengan bentuk *tree*, yang memililki nilai pada *leaf*. Cara bekerja minimax adalah dengan memindahkan nilai dari *leaf* node ke *parent* node sesuai dengan aturan *minimax* strategy. Maka dalam penerapan *minimax game* strategy seleksi langkah yang dilakukan *player* akan terarah kepada node dengan nilai terbesar(terkecil). Tujuan dari *minimax strategy* khususnya dalam *game* adalah untuk memilih langkah dengan nilai yang terbesar, yang merupakan hasil yang terbaik untuk permainan.

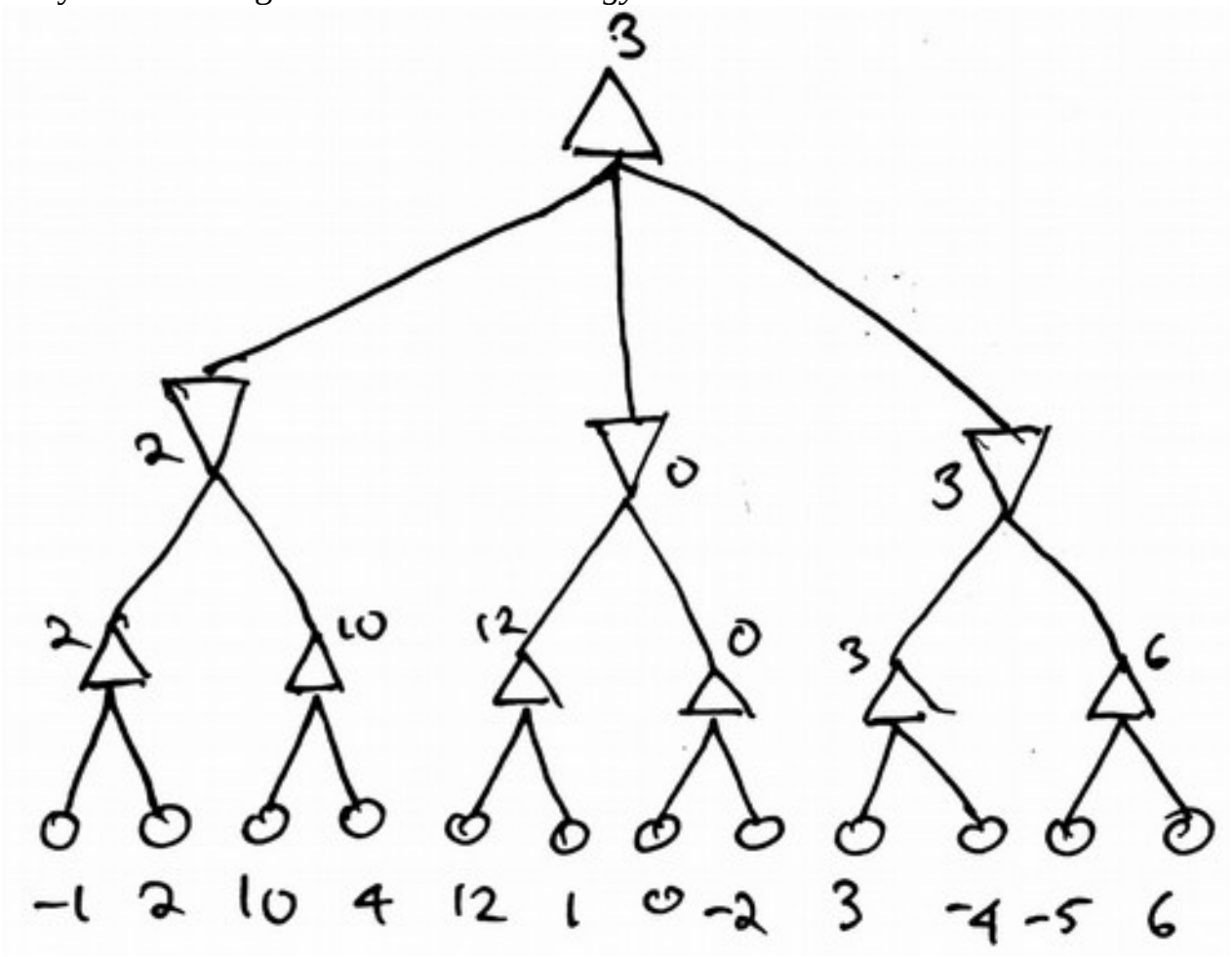
Langkah pertama algoritma *minimax* adalah untuk membuat seluruh *game tree*, lalu menentukan kegunaan setiap node antara node itu merupakan MIN atau MAX. Saat aturan setiap level *tree* sudah ditentukan maka nilai akan dipindahkan dari *child* node ke *parent* node

Alpha-Beta Pruning
Alpha-Beta Pruning adalah cara untuk membantu *minimax strategy* menjadi lebih efisien, tujuannya adalah untuk mengeliminasi cabang *tree* dari tahap pengecekkan karena sudah tidak sesuai kriteria MIN to MAX atau MAX to MIN. Sehingga mengurangi waktu yang dibutuhkan untuk mendapatkan hasil yang sesuai.
alpha = highest-value (MAX) , *beta = lowest-value* (MIN)

Contoh Soal :



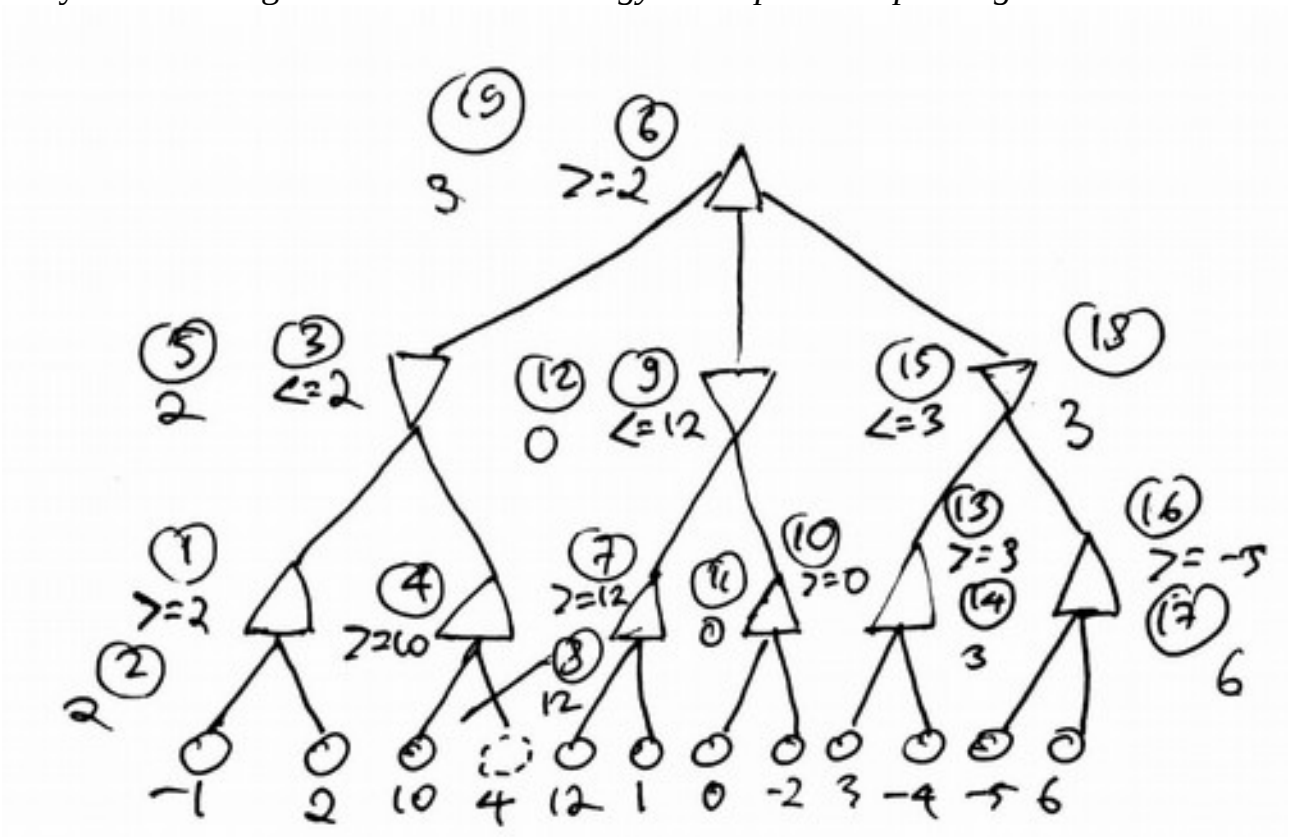
Penyelesaian dengan cara *minimax strategy*.



Penjelasan dari jawaban ini adalah

1. Pada lapisan pertama ada 6 *node* MAX, di ikuti dengan 3 *node* MIN, dan lapisan paling atas ada 1 *node* MAX.
2. Dari nilai yang berada di *leaf tree* ini diseleksi nilai-nilai yang memiliki jumlah yang lebih besar dan di masukkan ke *node* MAX.
3. Lalu dari 6 *node* MAX diseleksi lagi ke 3 *node* MIN maka diambil nilai-nilai yang memiliki jumlah yang lebih kecil.
4. Terakhir dari 3 *node* MIN akan diambil 1 nilai yang paling besar dan akan menjadi hasil dari *minimax strategy*, nilai tersebut adalah 3.

Penyelesaian dengan cara *minimax strategy* dan *alpha beta pruning*



Penjelasan dari jawaban ini adalah

1. Pertama nilai -1 dimasukkan kedalam *node* MAX sebagai *alpha*.
2. Lalu setelah di bandingkan dengan nilai 2, karena 2 > -1 maka nilai 2 menggantikan nilai -1 di *node* MAX
3. Lalu nilai 2 dimasukkan ke *node* MIN sebagai *beta*.
4. Saat melihat bahwa ada nilai 10 di *node* lain maka *child* lain *node* tersebut di *prune*, karena nilai *child* pertama sudah lebih besar dari nilai yang ada di *node* MIN. Maka tidak perlu di cek lagi.
5. Nilai 2 menjadi nilai *node* MIN sebagai *beta*.
6. Nilai 2 dimasukkan kedalam *node* MAX yang paling atas sebagai *alpha*.
7. Lalu nilai 12 dimasukkan ke *node* MAX sebagai *alpha*.
8. Setelah perbandingan nilai 12 dan 1, nilai 12 > 1 maka nilai *node* MAX tetap 12
9. Nilai 12 dimasukkan ke *node* MIN sebagai *beta*.
10. Nilai 0 dimasukkan ke *node* MAX sebagai *alpha*.
11. Setelah perbandingan nilai 0 dan -2, nilai 0 > -2 maka nilai *node* MAX tetap 0.
12. Lalu nilai 0 dibandingkan dengan nilai 12 di *node* MIN, karena 0 < 12 maka *node* MIN berganti nilai menjadi 0.
13. *node* MAX dimasukkan nilai 3 sebagai *alpha*.
14. Dilakukan perbandingan dengan nilai -4 dan karena 3 > -4 maka nilai *node* MAX tetap 3.
15. Nilai 3 dimasukkan ke *node* MIN sebagai *beta*.
16. Nilai -5 dimasukkan ke *node* MAX sebagai *alpha.*

17. Dilakukan perbandingan dengan nilai 6 dan karena 6 > -5 maka nilai *node* MAX berganti menjadi 6.
18. Setelah dilakukan perbandingan dengan nilai 6, karena 3 < 6 maka nilai *node* MIN tetap 3.
19. Pada *node* MAX yang teratas, dilakukan perbandingan nilai 2 dan 3, karena nilai 3 > 2 maka nilai *node* MAX terganti menjadi 3.

Logical Agents

Propositional Logic

Propositional logic adalah logika yang paling dasar biasanya yang dipakai dalam koding sehari-hari seperti OR dan AND yang memiliki symbol ˅ untuk OR dan simbol ^ untuk AND. Selain OR dan AND masih ada 3 tipe lagi, dengan total 5 tipe *propositional logic*,

- ¬ (*NOT*), digunakkan untuk negasi sebuah value Boolean, jadi jika sebuah variabel *S* memiliki value *true* dan diberi tanda ¬ dan menjadi ¬*S* value variabel di negasi sehingga menjadi *false*.
- ^ (*AND*), digunakkan untuk membandingkan dua value Boolean *true* dan *false*, dan hanya akan menghasilkan *true* jika dua keadaan yang dibandingkan juga *true*.
- ˅ (*OR*), digunakkan untuk membandingkan dua value Boolean *true* dan *false*, dan hanya akan menghasilkan *true* jika salah satu value yang dibandingkan juga *true*.
- ⇒ (*IMPLICATION*), digunakkan untuk membandingkan dua value Boolean *true* dan *false*, dan hanya akan menghasilkan *false* jika value kedua yang dibandingkan *false*.
- ⇔ (*EQUIVALENT*), digunakkan untuk membandingkan dua value Boolean *true* dan *false*, dan hanya akan menghasilkan *true* jika kedua value yang dibandingkan sama. Contoh variabel *X* = *true* dan *Y* = *false* maka jika dibandingkan akan menghasilkan *false*, tetapi jika variabel *X* = *false* dan *Y* = *false* atau variabel *X* = *true* dan *Y* = *true* maka jika dibandingkan akan menghasilkan *true*.

Tabel *Propositional Logic*

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
False	*False*	*True*	*False*	*False*	*True*	*True*
False	*True*	*True*	*False*	*True*	*True*	*False*
True	*False*	*False*	*False*	*True*	*False*	*False*
True	*True*	*False*	*True*	*True*	*True*	*True*

Sumber : https://www.massey.ac.nz/~mjjohnso/notes/59302/fig06.09.gif

First Order Logic

First-Order Logic

First-Order Logic mengasumsikan dunia terdiri dari objek, relasi, dan fungsi. Berbeda dengan *propositional logic* yang mengasumsi dunia terdiri dari fakta. Perbedaannya bisa dilihat dari penggunaan kalimatnya, kalau *propositional logic* kalimatnya seperti *"Spike is a dong"* sedangkan kalau *first-order logic* kalimat akan menjadi *"there exists X such that X is Spike and X is a dog"* dalam instansi ini *X* adalah variabel.

Jika ada kalimat "Kevin nyetir motor", maka bentuk kalimat ini dalam FOL adalah *nyetir(Kevin, motor)*. Contoh kalimat lain seperti "Kevin jalan ke sekolah dan Nando menggosok gigi", maka bentuk kalimat ini dalam FOL adalah *jalan(Kevin, sekolah)* ^ *menggosok(Nando, gigi)*.

Contoh Soal :

Misalkan ada kalimat seperti :

1. Kevin adalah seorang IT Profesional
2. Steven tinggal di Jepang
3. Semua orang yang tinggal di Jepang bahagia ketika ada penemuan IT Baru
4. Kevin menemukan hal IT Baru

Hasil FOL:

1. Profesional(Kevin, IT)
2. Tinggal(Steven, Jepang)
3. ∀X, ∃Y, ∃Z: Tinggal (X, Y) ^ Profesional (Z, Y) ^ Menemukan(Z) -> Bahagia(X)
4. Menemukan(Kevin)

FOL to CNF Conversion

Untuk konversi menjadi CNF, premis di FOL tidak boleh menggunakan kuantor, implikasi, dan biimplikasi. Lalu pernyataan dianggap salah dan dijadikan premis.

CNF dari kalimat FOL diatas:

1. Profesional(Kevin, IT)
2. Tinggal(Steven, Jepang)
3. ¬Tinggal(X, Y) ˅ ¬Profesional(Z, Y) ˅ ¬Menemukan(Z) ˅ Bahagia(X)
4. ¬ Bahagia(X)

Proof by Resolution

Pembuktian suatu kondisi bisa dilakukan dengan melihat premis-premis yang berhubungan dengan *statement* yang ingin di proof.

Buktikan : Steven bahagia Kevin menemukan hal IT baru
1. Semua orang yang tinggal di Jepang bahagia jika ada penemuan IT baru.
2. Kevin adalah IT Profesional.
3. Kevin membuat penemuan IT baru.
4. Steven tinggal di Jepang.

Maka Steven bahagia

Pembuktian menggunakan FOL :
1. ∀X, ∃Y, ∃Z: Tinggal (X, Y) ^ ITProfesional (Z) ^ Menemukan(Z) -> Bahagia(X)
2. Profesional(Kevin, IT)
3. Menemukan(Kevin)
4. Tinggal(Steven, Jepang)

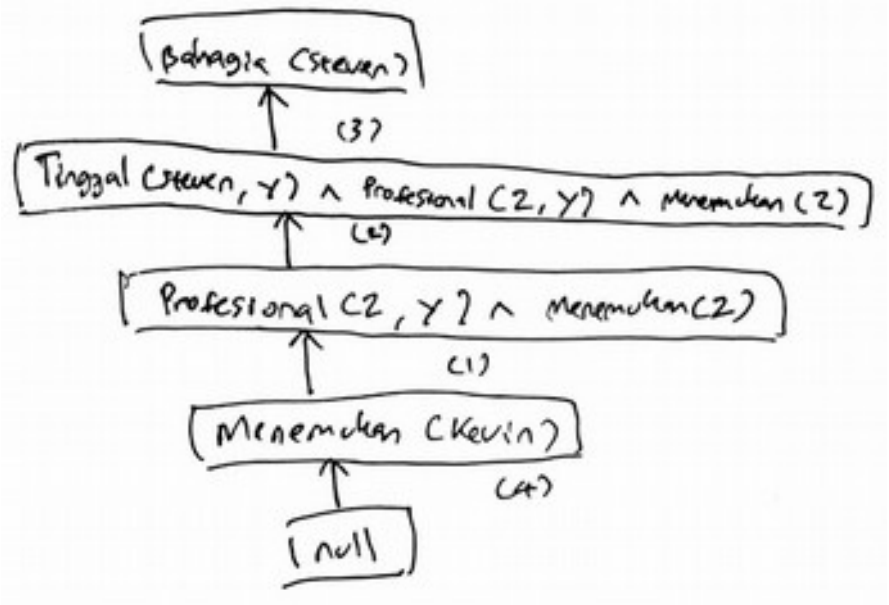
Bahagia(Steven)

Pembuktian dengan FOL yang dikonversikan ke CNF :
1. Profesional(Kevin, IT)
2. Menemukan(Kevin)
3. Tinggal(Steven, Jepang)
4. ¬Tinggal(X, Y) ˅ ¬ITProfesional(Z) ˅ ¬Menemukan(Z) ˅ Bahagia(X)

Bahagia(Steven)

Terbukti bahwa Steven bahagia saat Kevin membuat penemuan IT baru.

Backward Chaining
Cara lain untuk melakukan *proof by resolution* adalah dengan *Backward Chaining*. Kita melakukan pengecekan dengan menggunakan pernyataan terakhir sebagai awal chain. Lalu menambahkan premis-premis setelahnya hingga menghasilkan nilai *null.*



Quantifying Uncertainty

Basic Probability Notation

Probabilitas adalah kemungkinan terjadinya suatu *event* dalam eksperimen yang acak. Dalam probabilitas 0 berarti tidak ada kemungkinan *event* terjadi sedangkan 1 kebalikkannya. Satu dadu memiliki 6 sisi sehingga jika dua dadu di lempar akan ada 1 dari 6^2 kemungkinan bahwa dadu tersebut akan menghasilkan salah satu dari notasi ini (*dadu pertama, dadu kedua*) : (1, 1) , (1, 2) , (1, 3) , (1, 4), (1, 5) …. (6, 6).

Probabilitas bisa dikalkulasikan secara bersih, jika satu kemungkinan tidak melebihi kemungkinan yang lain. Tetapi ada juga probabilitas yang sudah dikondisikan, seperti jika ada dua dadu yang di lempar.

Andaikan dadu pertama selalu memunculkan angka 1, dan dadu dua akan acak secara random, maka hasil probabilitas dadu akan menjadi 1/6 dibandingkan jika tidak ada kondisi maka memiliki probabilitas 1/36.

Ada juga memiliki rumus/notasi untuk probabilitas, misalkan sebuah kondisi b, maka notasi akan seperti ini $P(a \mid b) = P(a \wedge b) / P(b)$. Ini bisa digunakan untuk menghitung probabilitas yang dikondisikan seperti contoh dadu diatas. Jadi bisa tulis menjadi seperti ini : $P(D_1D_2 \mid D_1 = 1) = P(D_1D_2 \mid D_1 = 1) / P(D_1 = 1)$.

Exercise

1. Buktikan bahwa: Tio suka makan daging saat National Meat Day
   a) Premis 1: Semua orang di Indonesia makan daging saat National Meat Day
   b) Premis 2: Orang yang merayakan National Meat Day suka makan daging
   c) Premis 3: Tio tinggal di Indonesia dan merayakan National Meat Day

   Semua orang di Indonesia makan daging saat National Meat Day.
   Semua orang yang merayakan National Meat Day suka makan daging.
   Tio tinggal di Indonesia dan merayakan National Meat Day.

   Maka Tio suka makan daging saat National Meat Day

   Pembuktian menggunakan FOL:
   1. ∀X, ∃Y, ∃Z: Tinggal(X, Y) ^ MakanDaging(X, Y, Z) ^ Merayakan(X, Z) -> SukaMakanDagingSaatNationalMeatDay(X)
   2. Tinggal(Tio, Indonesia)
   3. Merayakan(Tio, NationalMeatDay)
   4. MakanDaging(Tio, Indonesia, NationalMeatDay)

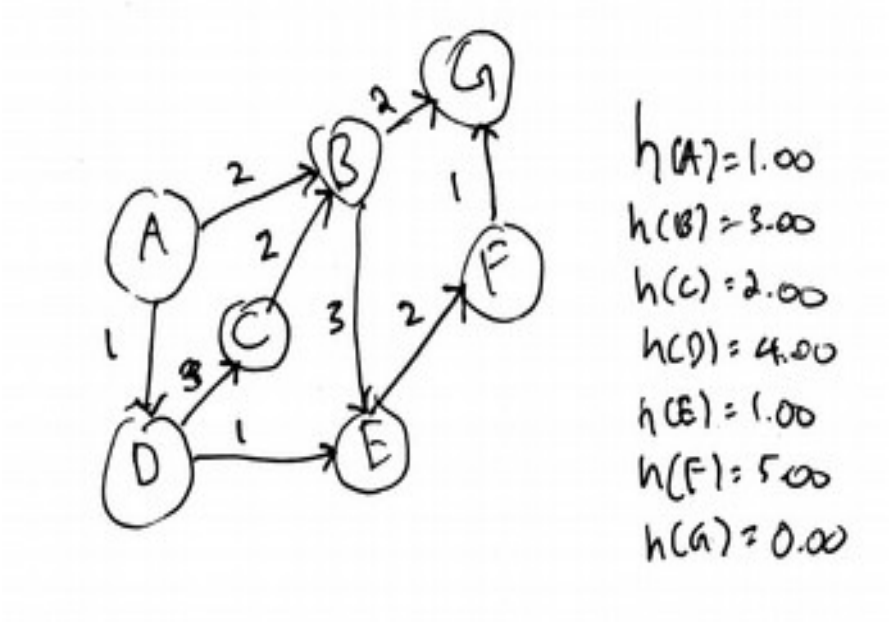
   SukaMakanDagingSaatNationalMeatDay(Tio)

   Pembuktian menggunakan CNF:
   1. Tinggal(Tio, Indonesia)
   2. Merayakan(Tio, NationalMeatDay)
   3. MakanDaging(Tio, Indonesia, NationalMeatDay)
   4. ¬Tinggal(X, Y) ˅ ¬MakanDaging(X, Y, Z) ˅ ¬Merayakan(X, Z) ˅ SukaMakanDagingSaatNationalMeatDay(X)

   SukaMakanDagingSaatNationalMeatDay(Tio)

   Terbukti bahwa Tio suka makan daging saat National Meat Day

2. Tentukan solusi BFS, DFS, UCS, *Greedy,* dan A* dari *graph* berikut:



Penyelesaian Soal untuk menemukan *node G* dengan BFS :

1. Queue = {A}
2. Visit A, Queue = {B^A, D^A}
3. Visit B^A, Queue = { D^A, E^B , G^B }
4. Visit D^A, Queue = { E^B , G^B , C^D , E^D}
5. Visit E^B, Queue = { G^B , C^D , E^D, F^E }
6. Visit G^B, found *node* G.

Path : A-B-G Visited: A, B, D, E, G

Penyelesaian Soal untuk menemukan *node* G dengan DFS :

1. Queue = {A}
2. Visit A, Queue = { D^A, B^A }
3. Visit B^A, Queue = { D^A, G^B, E^B }
4. Visit E^B, Queue = { D^A, G^B, F^E }
5. Visit F^E, Queue = { D^A, G^B, G^F }
6. Visit G^F, found *node* G.

Path : A-B-E-F-G Visited: A, B, E, F, G

Penyelesaian Soal untuk menemukan *node* G dengan UCS:

1. Queue = {A}
2. Visit A, Queue = { D^A, B^A } //1 ,2
3. Visit D^A, Queue = { B^A, E^D , C^D } // 2, 2, 5
4. Visit B^A, Queue = { E^D , C^D , G^B } // 2, 4, 4
5. Visit E^D, Queue = { C^D , G^B , F^E } // 4, 4, 4

6. Visit C^D, Queue = { G^B , F^E , B^C } // 4, 4, 6
7. Visit G^B, found *node* G.

Path : A-B-G Visited: A, B, C, D, E, G

Penyelesaian Soal untuk menemukan *node* G dengan *Greedy*:

1. Queue = {A}
2. Visit A, Queue = { B^A, D^A } //3 ,4
3. Visit B^A, Queue = { G^B, D^A } // 0, 4
4. Visit G^B, found *node* G.

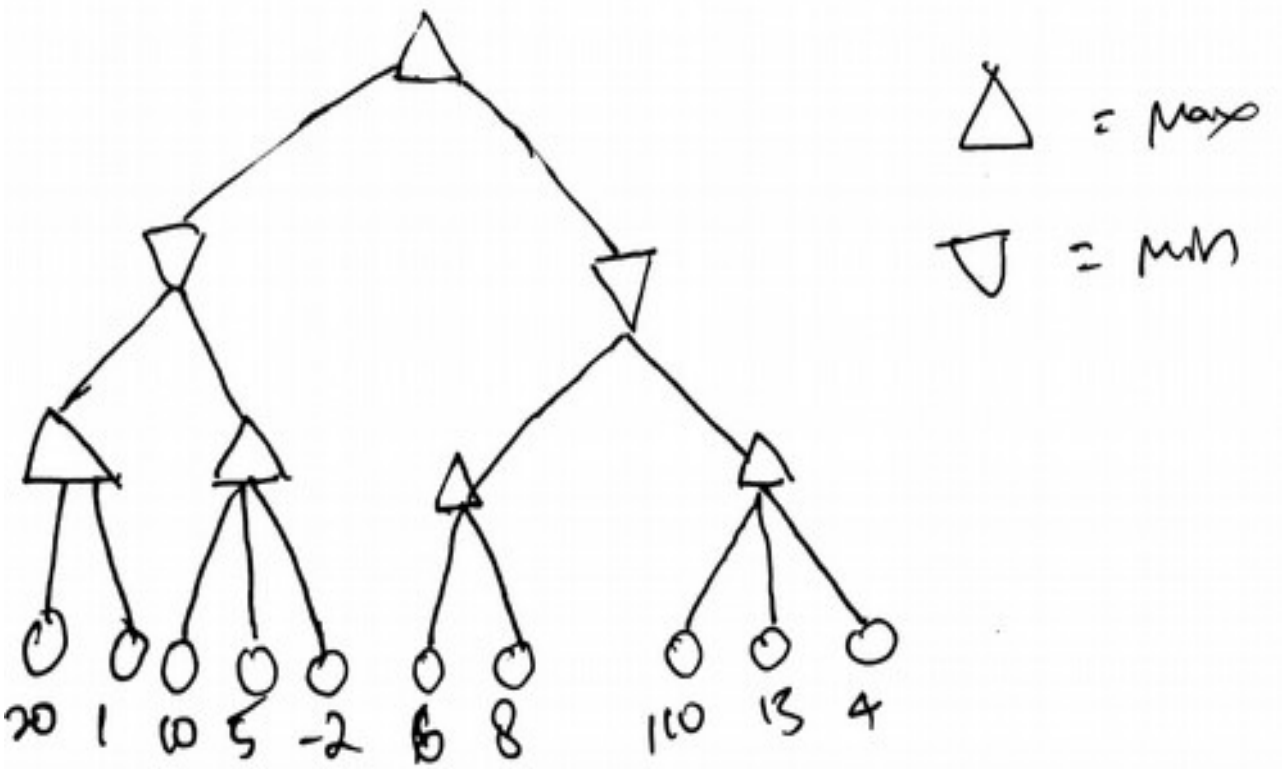
Path : A-B-G Visited: A, B, G

Penyelesaian Soal untuk menemukan *node* G dengan *A**:

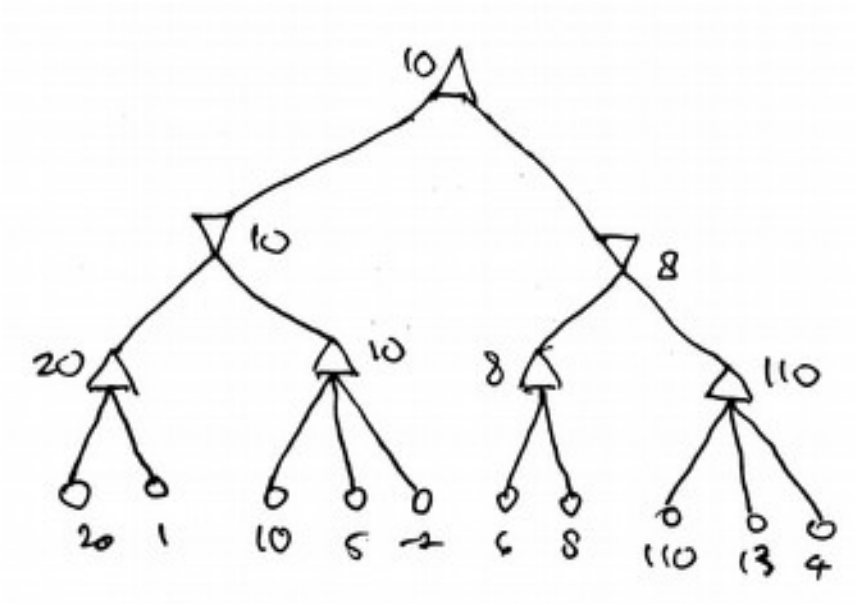
1. Queue = {A}
2. Visit A, Queue = { B^A, D^A } // 5, 5
3. Visit B^A, Queue = { G^B, D^A , E^D } // 2, 5
4. Visit G^B, found *node* G.

Path : A-B-G Visited: A, B, G

3. Selesaikan permasalahan *minimax tree* dibawah ini menggunakan cara *minimax strategy* dan *alpha-beta pruning* :



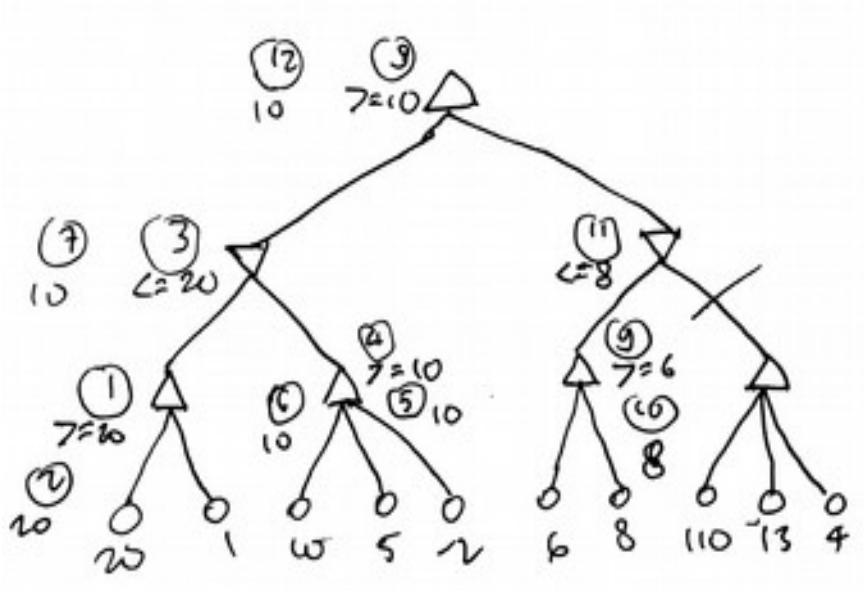
Penyelesaian menggunakan cara *minimax strategy*.



Penjelasan cara *minimax strategy*:

1. Langkah pertama *nilai* yang tersimpan dalam *leaf node* dibandingkan satu sama lain, dan karena diatas *leaf node* adalah MAX *node* maka nilai yang paling besar akan diambil
2. Lalu nilai yang sudah dinaikkan dibandingkan dengan pasangan lainnya, karena diatas MAX *node* ada MIN *node* maka akan diambil nilai terkecil untuk dinaikkan.
3. Saat nilai sudah masuk ke MIN *node* akan ada perbandingan terakhir sebelum masuk MAX *node* teratas, akan diambil nilai terbesar dari kedua *node* untuk dimasukkan ke dalam MAX *node* teratas, dan nilai tersebut adalah 10.

Penyelesaian menggunakan cara *minimax strategy* dan *alpha-beta pruning*.



Penjelasan *alpha-beta pruning*:

1. Nilai yang pertama di *leaf* node terujung kanan dinaikkan sebagai *alpha*.
2. Lalu dibandingkan dengan nilai kedua, karena yang dicari adalah nilai dengan jumlah terbesar, diambil tetap nilai 20 dan dinaikkan ke MAX *node*.
3. Nilai yang muncul dari MAX *node* terujung kanan dinaikkan ke MIN *node* sebagai *beta.*
4. Nilai pada *leaf node* selanjutnya dimasukkan kepada MAX *node* sebagai *alpha.*
5. Lalu dibandingkan dengan nilai kedua *child* dari MAX *node* dan nilai tidak berubah karena dicari nilai terbesar.
6. Lalu dibandingkan dengan nilai ketiga *child* dari MAX *node* dan nilai tidak berubah karena dicari nilai terbesar.
7. Nilai tetap dari MAX *node* lalu dibandingkan dengan nilai *beta* pada MIN *node* diatasnya, dan karena nilainya lebih kecil (10 < 20) maka menggantikan nilai sebelumnya.
8. Nilai 10 lalu dinaikkan ke MAX *node* teratas sebagai *alpha*.
9. Pada *leaf node* selanjutnya nilai pertama dimasukkan ke MAX *node* sebagai *alpha*.
10. Lalu dibandingkan dengan nilai kedua, tetapi karena nilai kedua lebih besar (8 > 6) maka nilai kedua menggantikan nilai pertama di MAX *node*.
11. Nilai kedua lalu dinaikkan ke MIN *node* sebagai *beta.* Tetapi karena nilai MIN *node* sudah lebih kecil dari MAX *node* teratas (8 < 10) maka tidak perlu dilakukan pengecekkan kepada *leaf node* selanjutnya, karena sudah pasti tidak akan terpilih nilai yang lebih besar dari nilai MAX *node* teratas.
12. Nilai pada MAX *node* teratas ditetapkan, dan nilai tersebut adalah 10.