CSCM603130: Sistem Cerdas
State Space & Uninformed Search

Fariz Darari, Aruni Yasmin Azizah

Fakultas Ilmu Komputer
Universitas Indonesia

2019/2020 • Semester Ganjil

Outline

1. Problem-solving agent
2. Representasi masalah: state space
3. Pencarian solusi: search
   - State space search
   - Pengulangan state
4. Strategi pencarian
   - Breadth-first
   - Uniform-cost
   - Depth-first
   - Iterative-deepening

Outline

Problem-Solving Agent

- Di kuliah yang lalu kita melihat contoh reflex agent: tidak cocok untuk masalah besar!
- Goal-based agent: memiliki tujuan, memungkinkan evaluasi tindakan dan memilih yang terbaik.
- Kali ini kita membahas satu kemungkinan jenis goal-based agent: problem-solving agent
  - Apa problem-nya? Apa solution-nya?
- Kita mulai dengan membatasi solusi dalam bentuk serangkaian tindakan (fixed sequence of actions) yang diambil untuk mencapai tujuan.

Mekanisme kerja Problem-Solving Agent

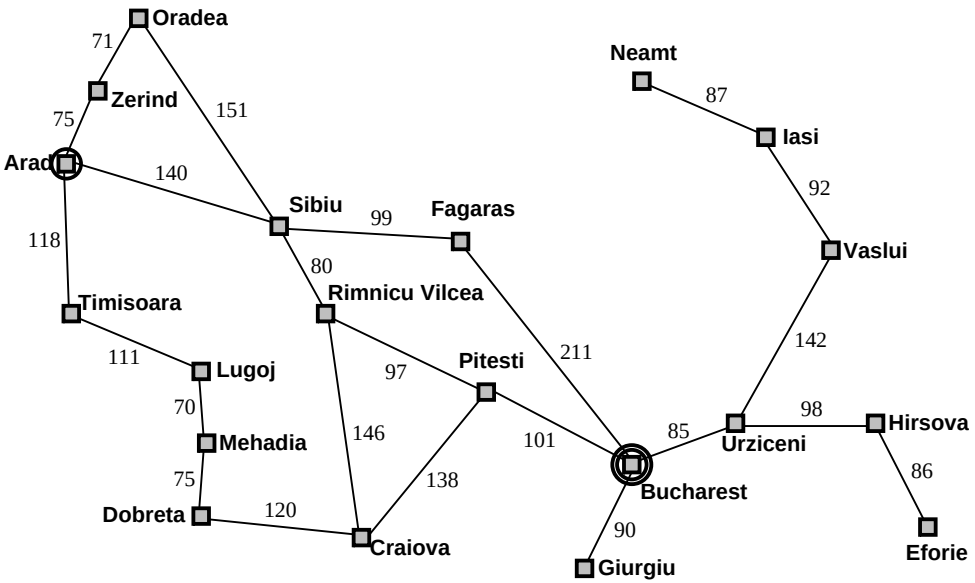
1. Perumusan tujuan (goal formulation): tentukan tujuan yang ingin dicapai
2. Perumusan masalah (problem formulation): tentukan tindakan (action) dan keadaan (state) yang dipertimbangkan dalam mencapai tujuan
   - Untuk saat ini problem solving agent mengasumsikan bahwa environment-nya: fully observable, deterministic, static, discrete
3. Pencarian solusi masalah (searching): tentukan rangkaian tindakan yang perlu diambil untuk mencapai tujuan
4. Pelaksanaan solusi (execution): laksanakan rangkaian tindakan yang sudah ditentukan di tahap sebelumnya
   - Agent ini melaksanakan tindakan dengan "mata tertutup" $\rightarrow$ mengabaikan percept!

Agent program Problem-Solving Agent

function SIMPLEPROBLEMSOLVINGAGENT (*percept*) **returns** *action*

state ← UPDATESTATE(*state*, *percept*)
if *seq* is empty **then**
  goal ← FORMULATEGOAL (*state*)
  problem ← FORMULATEPROBLEM (*state*, *goal*)
  seq ← SEARCH (*problem*)
  if *seq* = *failure* **then return** a null action
action ← FIRST (*seq*)
seq ← REST (*seq*)
return *action*

Tourist Agent berlibur di Rumania

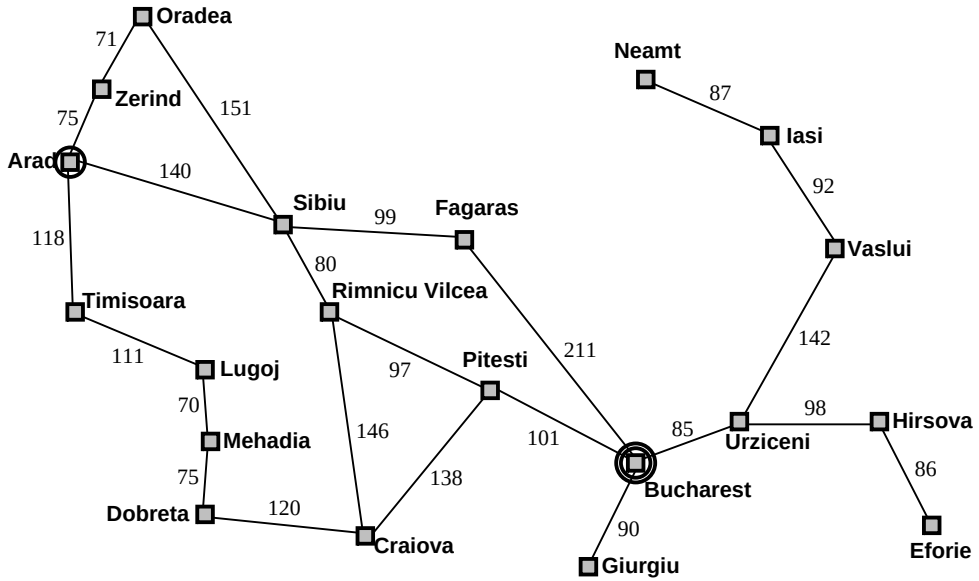


Kini, berada di Arad. Besok, berada di Bucharest.

- Perumusan tujuan: berada di Bucharest
- Perumusan masalah:
  - Tindakan (action): menyetir dari kota ke kota
  - Keadaan (state): berada di kota tertentu yang ada di Rumania
- Pencarian solusi: rangkaian tindakan menyetir dari satu kota ke kota lain menuju Bucharest

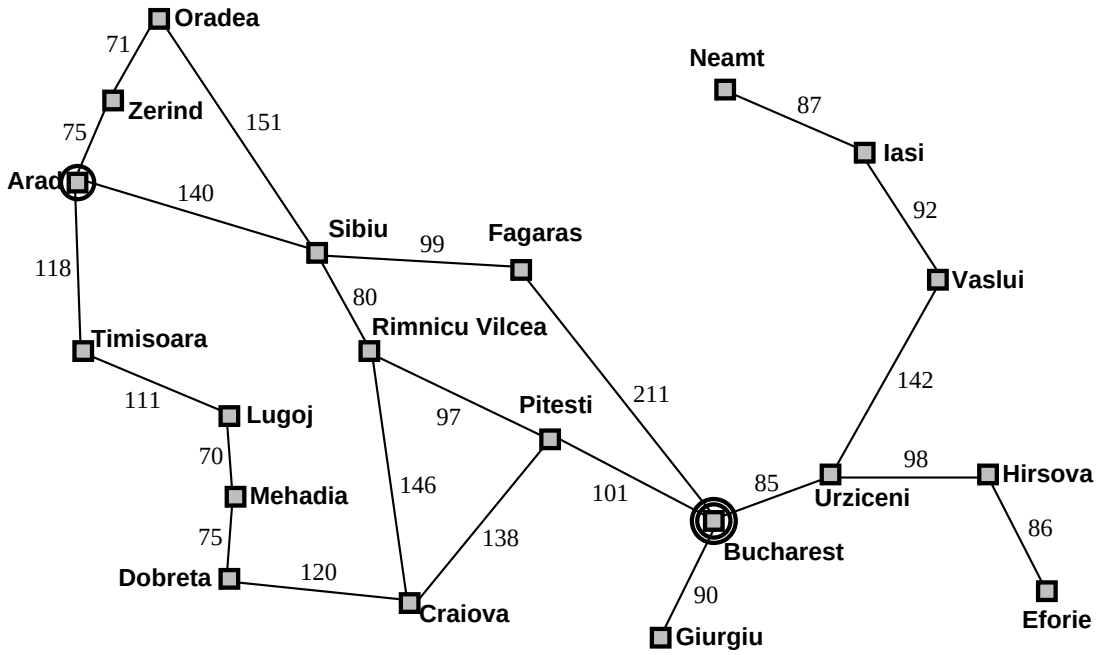
Outline

Perumusan masalah sebagai state space



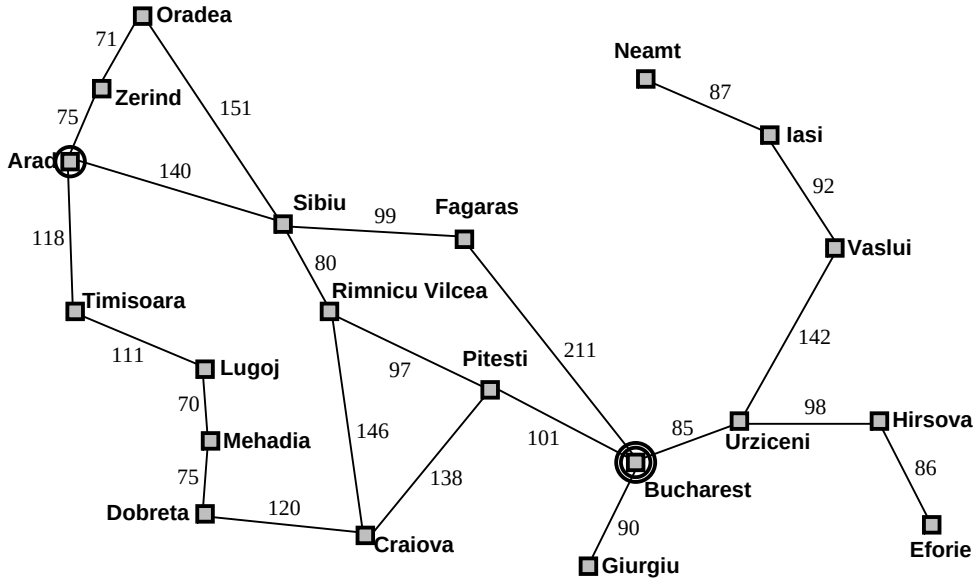
- Initial state: keadaan awal di mana si agent mulai, mis: *BeradaDi*(*Arad*)
- Possible actions: tindakan yang dapat dilakukan si agent pada state *s*, mis: *NyetirKe*(*Zerind*) dapat dilakukan saat agent berada pada state *BeradaDi*(*Arad*).

Perumusan masalah sebagai state space



- Model transisi dispesifikasikan melalui sebuah fungsi *Result*(s, a) yang memberikan state sebagai hasil pelaksanaan tindakan a saat berada di state s. Contoh:
  Result$(BeradaDi(Arad), NyetirKe(Zerind)) = BeradaDi(Zerind)$.

Perumusan masalah sebagai state space



- Initial state, tindakan, dan model transisi mendefinisikan state space: himpunan semua state yang dapat dicapai dari initial state melalui serangkaian tindakan.
  - Dapat direpresentasikan sebagai graph.
  - Path dalam state space adalah serangkaian state (dihubungkan serangkaian action).

Merumuskan state space

- Dunia nyata luar biasa kompleks dan rumit! State space harus merupakan abstraksi masalah supaya bisa dipecahkan.
  - State = representasi "keadaan nyata". Mis: *BeradaDi*(*Arad*)
    – irrelevant: dengan siapa? kondisi cuaca?
  - Action = representasi "tindakan nyata". Mis: *NyetirKe*(*Sibiu*)
    – irrelevant: jalan tikus, isi bensin, istirahat, dll.
- Abstraksi ini membuat masalah yang nyata lebih mudah dipecahkan.

Menelusuri sebuah state space

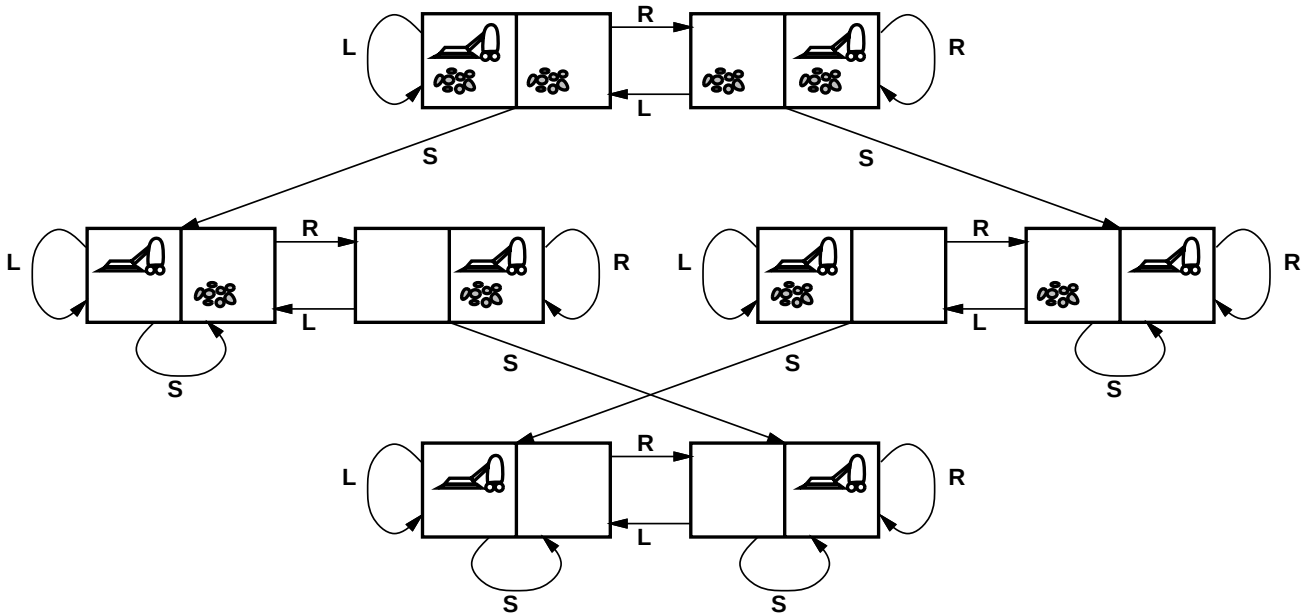
- Goal test: penentuan apakah suatu state adalah tujuan yang ingin dicapai.

  Eksplisit: himpunan goal state, mis: $\{BeradaDi(Bucharest)\}$.

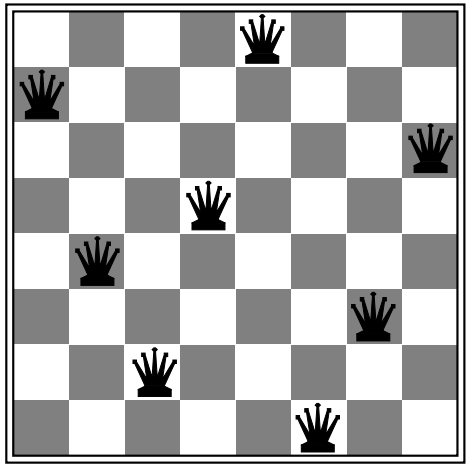
  Implisit: deskripsi tujuan, mis: dalam catur $\rightarrow$ skak mat.
- Path cost function: sebuah fungsi yang memberikan nilai numerik terhadap setiap path. Fungsi ini merefleksikan performance measure si agent.
  - Kita asumsikan path cost function merupakan *jumlah* dari cost individual action sepanjang path.
  - $c(s, a, s')$: step cost melakukan tindakan a di state s untuk mencapai state s'
- Sebuah solusi adalah path dari initial state ke goal state.
- Sebuah solusi optimal adalah solusi dengan nilai path cost minimal.

Contoh: VacuumCleanerWorld

- State: lokasi agent, status debu
- Possible action: *DoKeKiri* (L), *DoKeKanan* (R), *DoSedot* (S)
- Goal test: apakah semua ruangan bebas debu?
- Path cost: jumlah langkah dalam path, step cost = 1
- Model transisi sbb:



Contoh: 8-Queens Problem



Letakkan 8 bidak menteri (*queen*) sedemikian sehingga tidak ada yang saling menyerang (*queen* bisa menyerang dalam satu baris, kolom, diagonal).

- State: Papan catur dengan n buah *queen*, $0 \leq n \leq 8$.
  - Initial state: Papan catur yang kosong.
- Possible action: Letakkan sebuah *queen* di posisi kosong.
- Goal test: 8 *queen* di papan, tidak ada yang saling makan.

Masalah state space... combinatorial explosion!

- Dengan definisi masalah demikian, ada $64 \times 63 \times \ldots \times 57 \approx 1.8 \times 10^{14}$ path!
- Definisi masalah yang lebih baik:
  - State: Papan catur dengan n *queen*, $0 \leq n \leq 8$, satu per kolom di n kolom paling kiri sehingga tidak saling makan.
  - Possible action: Letakkan sebuah *queen* di posisi kosong di kolom paling kiri yang belum ada *queen*-nya sehingga tidak ada yang saling makan.
- State space sekarang ukurannya tinggal 2057, dan mudah dipecahkan.
- Perumusan masalah yang tepat bisa berakibat drastis!
- Meskipun demikian, untuk $n = 100$: 10^{400} vs. 10^{52}...

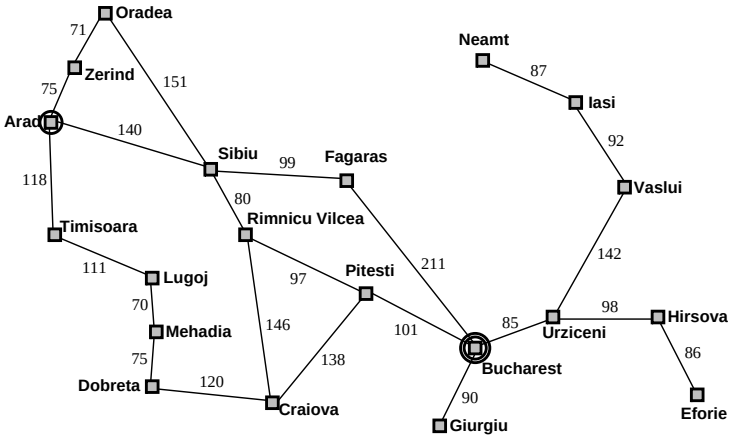
Outline

Outline

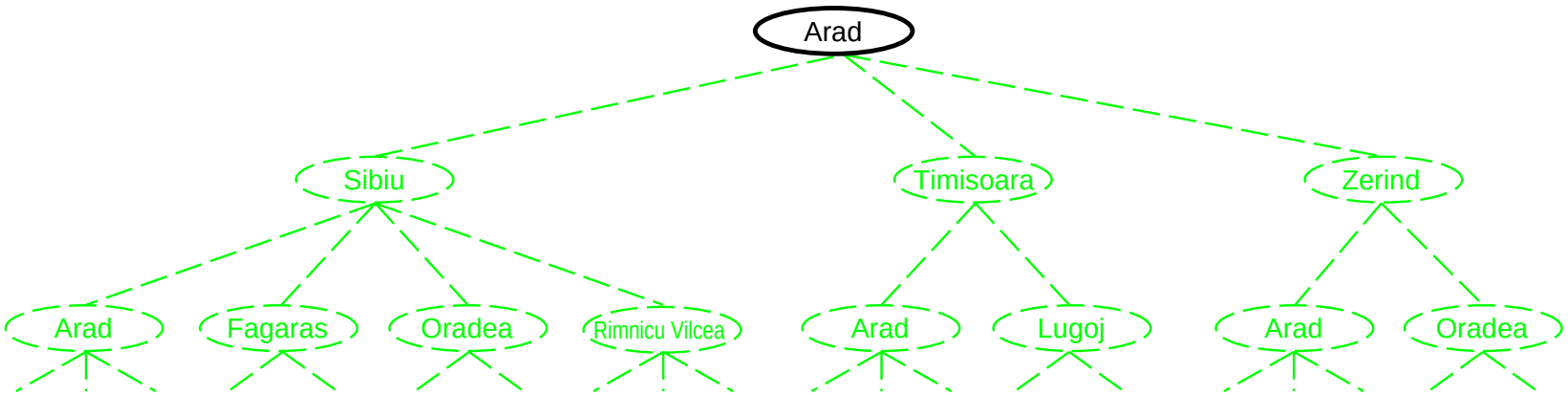
Mencari solusi melalui search tree

- Setelah merumuskan masalah → cari solusinya menggunakan sebuah search algorithm
- Search tree: sequence dari possible actions yang dimulai dari initial state.
- Search tree terdiri dari kumpulan node: merepresentasikan suatu *state* pada suatu *state space*.
- Root node merepresentasikan initial state.
- Node expansion → Penerapan fungsi RESULT: menerapkan action yang ada pada *current state* (yang diwakili *node*) menghasilkan *state* baru (percabangan dari parent node ke child node).
- Kumpulan semua node yang belum di-*expand* disebut frontier sebuah *search tree*.

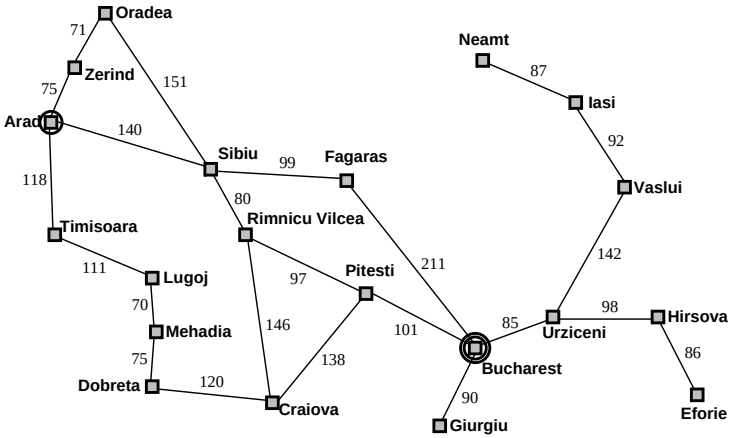
Contoh pembentukan search tree



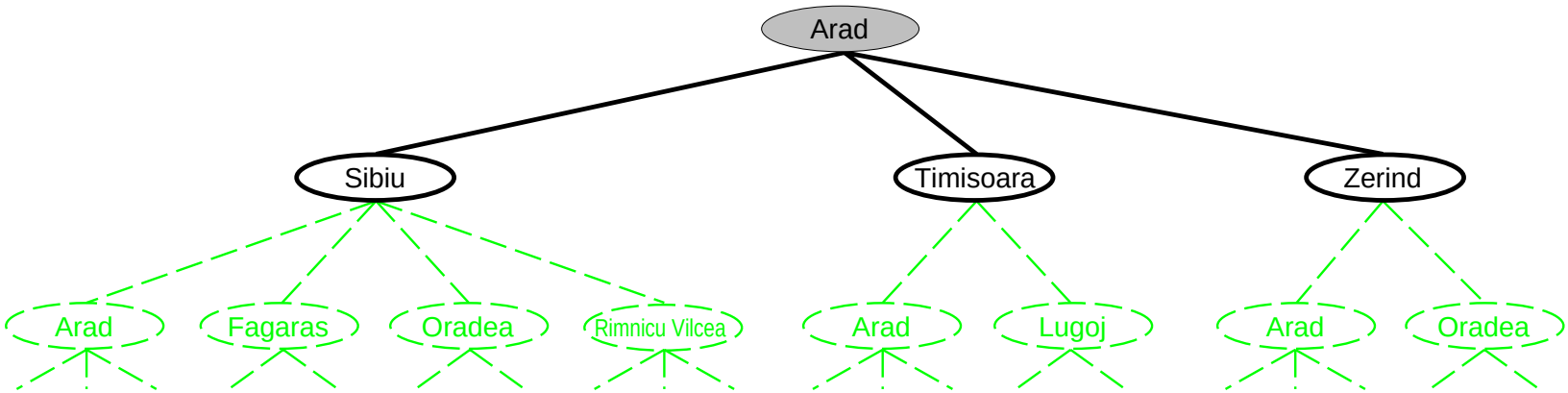
- Mulai dari root node (Arad) sebagai current node.
  - Lakukan goal test.



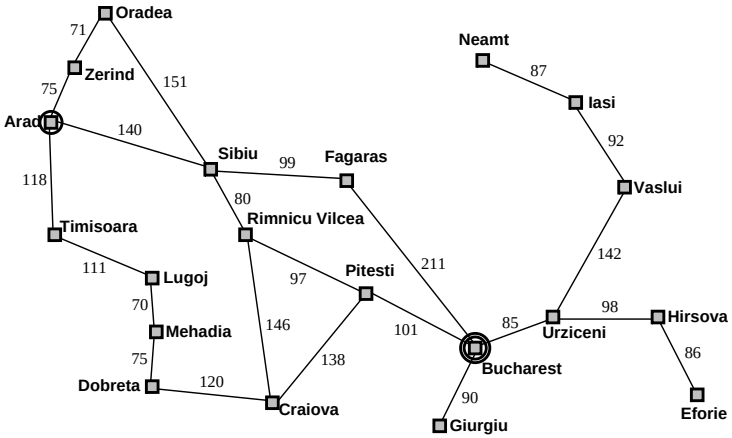
Contoh pembentukan search tree



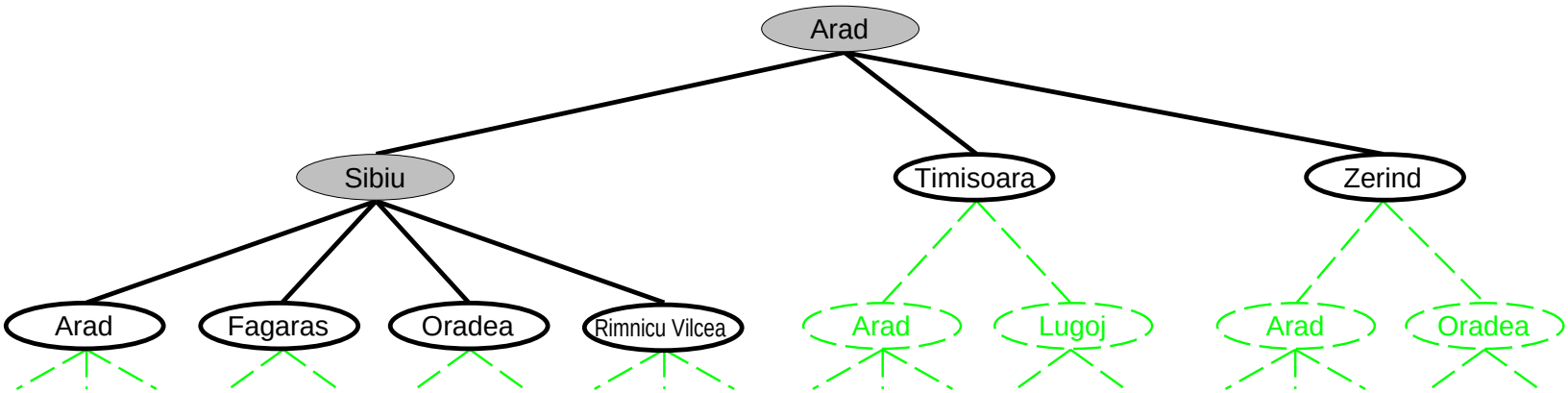
- Lakukan *node expansion* terhadapnya.



Contoh pembentukan search tree



- Pilih salah satu node yang di-expand sebagai current node yang baru. Ulangi langkah sebelumnya.



Algoritma TreeSearch

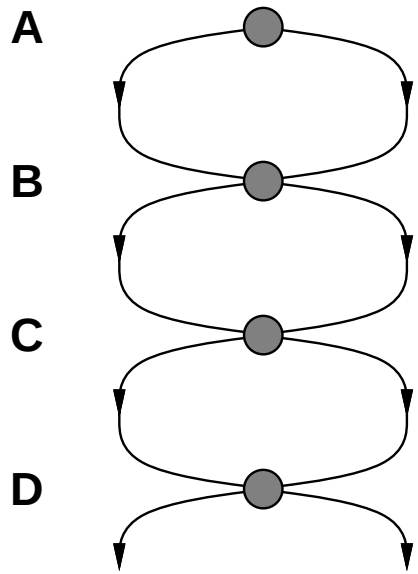
function TreeSearch (*problem*) **returns** *solution* or *failure*

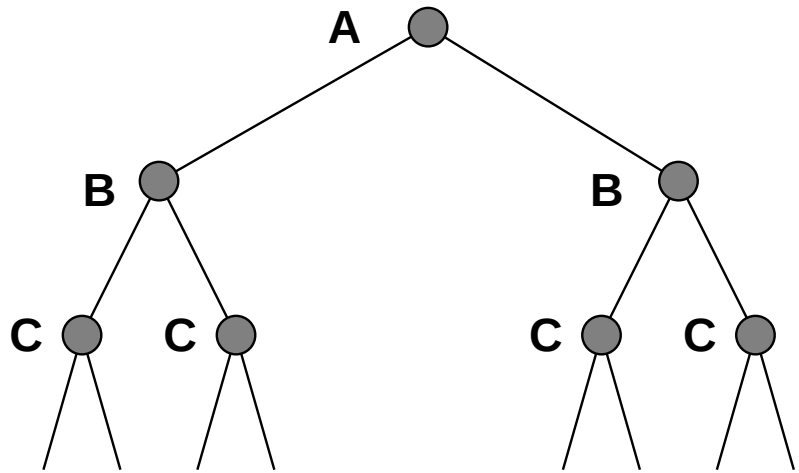
initialize *frontier* using the initial state of *problem*
loop do
  if *the frontier is empty* **then return** failure
  choose a leaf node and remove it from the *frontier*
  if the node contains a goal state
    then return corresponding solution
  expand the chosen node, adding the resulting nodes to the *frontier*

Outline

Masalah: *state* yang mengulang di dalam *search tree*

Kegagalan menangani *state* yang mengulang dapat membuat masalah linier menjadi eksponensial!





Ingat dua variasi definisi masalah 8-queens problem.

Solusi: belajar dari sejarah

- *Algorithms that forget their history are doomed to repeat it...*
- Solusinya adalah untuk mencatat *state* mana yang sudah pernah dicoba. Catatan ini disebut explored set
- Modifikasi algoritma TreeSearch dengan *explored set* menjadi GraphSearch.

Algoritma GraphSearch

function GraphSearch (*problem*) **returns** *solution* or *failure*

initialize *frontier* using the initial state of *problem*
initialize *explored set* **to be empty**
loop do
  if *the frontier is empty* **then return** failure
  choose a leaf node and remove it from the *frontier*
  if the node contains a goal state
    then return corresponding solution
  add the node to the *explored set*
  expand the chosen node, adding the resulting nodes to the *frontier*
    only if not in the *frontier* **or** *explored set*

Outline

Strategi pencarian

- Terdapat berbagai jenis strategi untuk melakukan search.
- Semua strategi ini berbeda dalam satu hal: urutan dari node expansion.
- Search strategy di-evaluasi berdasarkan:
  - completeness: apakah solusi (jika ada) pasti ditemukan?
  - time complexity: berapa lama waktu utk menemukan solusi?
    - Diukur menggunakan jumlah node yang di-expand.
  - space complexity: berapa banyak memory dlm melakukan search?
    - Diukur dengan jumlah maksimum node di dalam memory.
  - optimality: apakah solusi optimal pasti ditemukan?
- Time & space complexity diukur berdasarkan
  - b - branching factor (maks banyaknya successor) dari suatu node
  - d - kedalaman minimal dari suatu node goal
  - m - kedalaman maksimum dari search tree (bisa infinite!)

Uninformed search strategies

- Uninformed search hanya menggunakan informasi dari definisi masalah.
- Bisa diterapkan secara generik terhadap semua jenis masalah yang bisa direpresentasikan dalam sebuah state space.
- Ada beberapa jenis:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative-deepening search

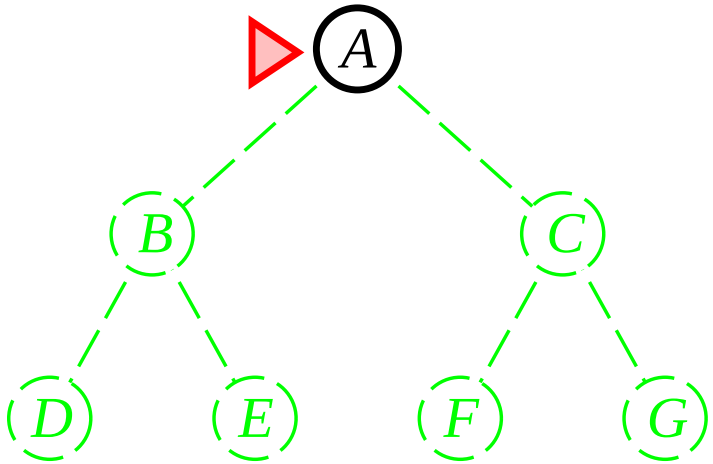
Outline

Breadth-first search

Prinsip algoritma breadth-first search

Lakukan node expansion terhadap node di *frontier* yang paling dekat ke *root*.

- Implementasi: *frontier* adalah sebuah *queue*, struktur data FIFO (First In First Out)
- Hasil *node expansion* (successor function) ditaruh di belakang

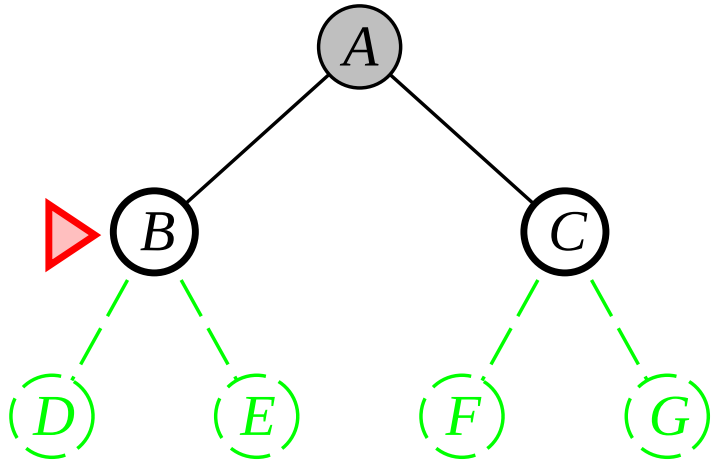


Breadth-first search

Prinsip algoritma breadth-first search

Lakukan node expansion terhadap node di *frontier* yang paling dekat ke *root*.

- Implementasi: *frontier* adalah sebuah *queue*, struktur data FIFO (First In First Out)
- Hasil *node expansion* (successor function) ditaruh di belakang

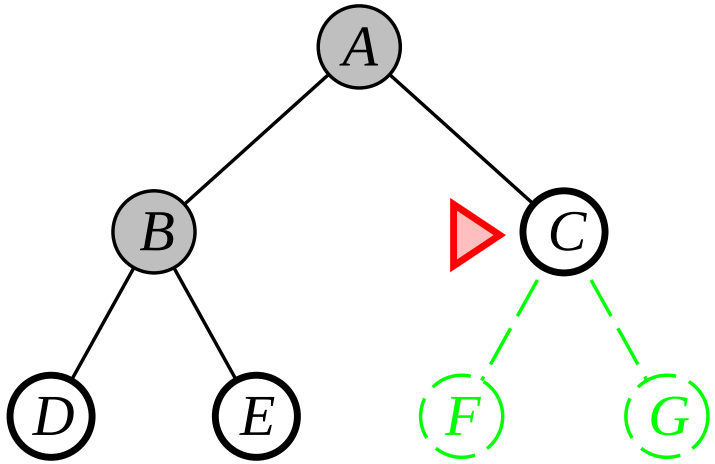


Breadth-first search

Prinsip algoritma breadth-first search

Lakukan node expansion terhadap node di *frontier* yang paling dekat ke *root*.

- Implementasi: *frontier* adalah sebuah *queue*, struktur data FIFO (First In First Out)
- Hasil *node expansion* (successor function) ditaruh di belakang

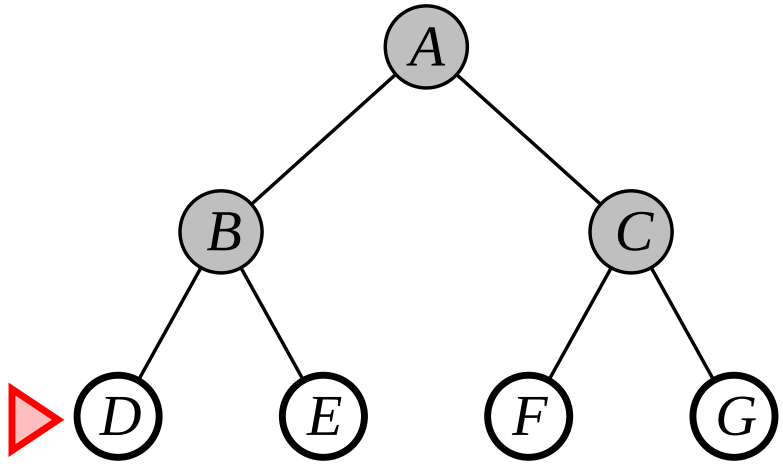


Breadth-first search

Prinsip algoritma breadth-first search

Lakukan node expansion terhadap node di *frontier* yang paling dekat ke *root*.

- Implementasi: *frontier* adalah sebuah *queue*, struktur data FIFO (First In First Out)
- Hasil *node expansion* (successor function) ditaruh di belakang



Sifat breadth-first search

- Complete?

Sifat breadth-first search

- Complete? Ya, jika b terbatas

Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity?

Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \rightarrow$ eksponensial dalam d.

Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \rightarrow$ eksponensial dalam d.
- Space complexity?

Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \rightarrow$ eksponensial dalam d.
- Space complexity? $O(b^d)$, yaitu didominasi oleh banyaknya node pada frontier.

Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \rightarrow$ eksponensial dalam d.
- Space complexity? $O(b^d)$, yaitu didominasi oleh banyaknya node pada frontier.
- Optimal?

Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \rightarrow$ eksponensial dalam d.
- Space complexity? $O(b^d)$, yaitu didominasi oleh banyaknya node pada frontier.
- Optimal? Pada umumnya tidak optimal (optimal, jika semua *step cost* sama).

Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \rightarrow$ eksponensial dalam d.
- Space complexity? $O(b^d)$, yaitu didominasi oleh banyaknya node pada frontier.
- Optimal? Pada umumnya tidak optimal (optimal, jika semua *step cost* sama).

Masalah utama breadth-first search adalah space

Mis: 1 node memakan 1000 byte, dan $b = 10$
Jika $d = 6$, ada 10^6 node $\approx$ 1 gigabyte.
Jika $d = 12$, ada 10^{12} node $\approx$ 1 petabyte!

Outline

Uniform-cost search

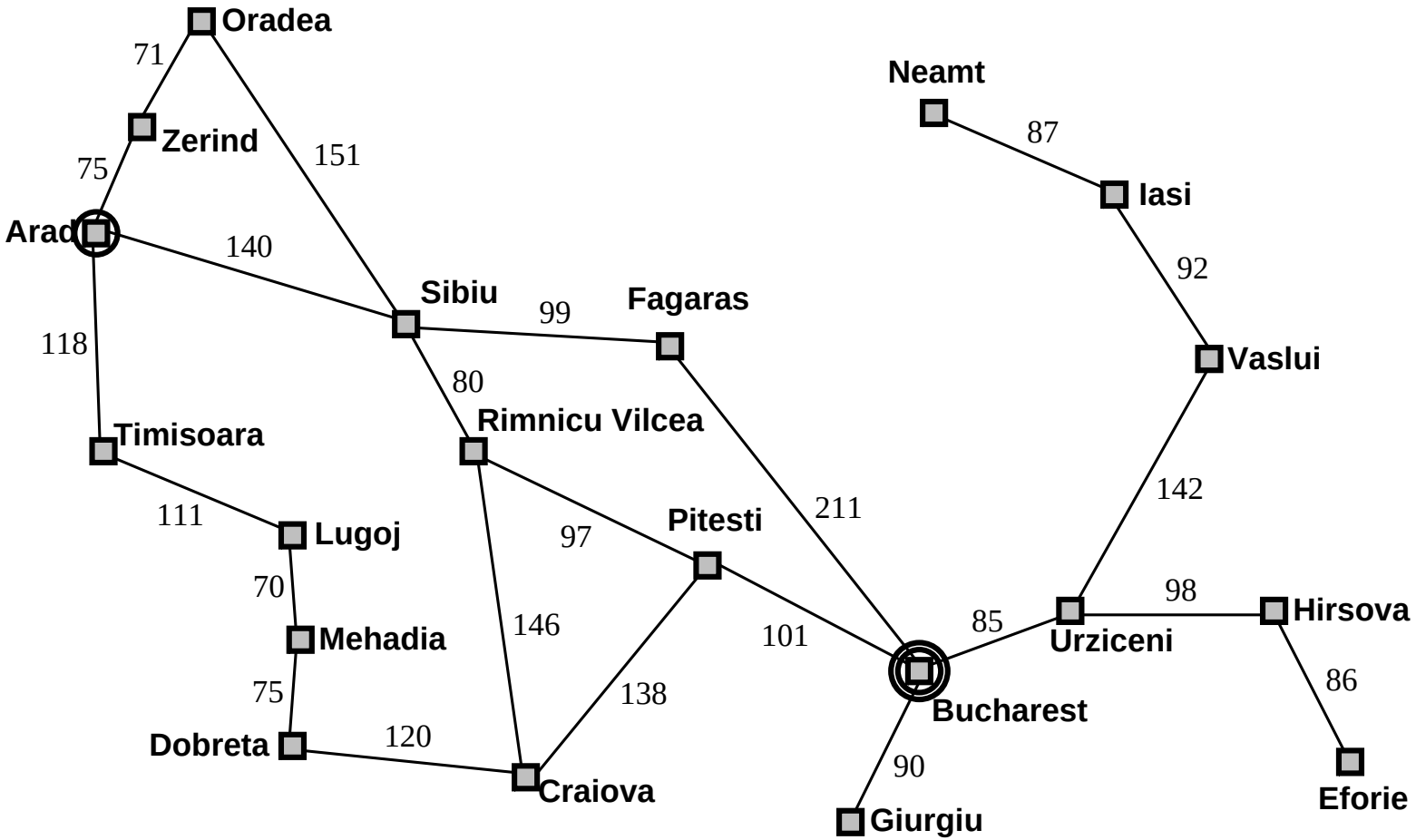
Prinsip algoritma uniform-cost search

Lakukan node expansion terhadap node di *frontier* yang path cost-nya paling kecil.

- Implementasi: *frontier* adalah sebuah *priority queue* di mana node disortir berdasarkan path cost function $g(n)$.
- Jika semua step cost sama, *uniform-cost* sama dengan *breadth-first*.

Contoh uniform-cost search

Coba cari optimal solution dari Arad ke Bucharest!



Sifat uniform-cost search

- Complete?

Sifat uniform-cost search

- Complete? Ya, jika *step cost* $\geq \epsilon$ untuk $\epsilon > 0$

Sifat uniform-cost search

- Complete? Ya, jika *step cost* $\geq \epsilon$ untuk $\epsilon > 0$
- Time complexity?

Sifat uniform-cost search

- Complete? Ya, jika *step cost* $\geq \epsilon$ untuk $\epsilon > 0$
- Time complexity? $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, di mana C^* adalah *cost* dari *optimal solution*

Sifat uniform-cost search

- Complete? Ya, jika *step cost* $\geq \epsilon$ untuk $\epsilon > 0$
- Time complexity? $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, di mana C^* adalah *cost* dari *optimal solution*
- Space complexity?

Sifat uniform-cost search

- Complete? Ya, jika *step cost* $\geq \epsilon$ untuk $\epsilon > 0$
- Time complexity? $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, di mana C^* adalah *cost* dari *optimal solution*
- Space complexity? $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- Optimal?

Sifat uniform-cost search

- Complete? Ya, jika *step cost* $\geq \epsilon$ untuk $\epsilon > 0$
- Time complexity? $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, di mana C^* adalah *cost* dari *optimal solution*
- Space complexity? $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- Optimal? Ya, karena urutan *node expansion* dilakukan urut $g(n)$.

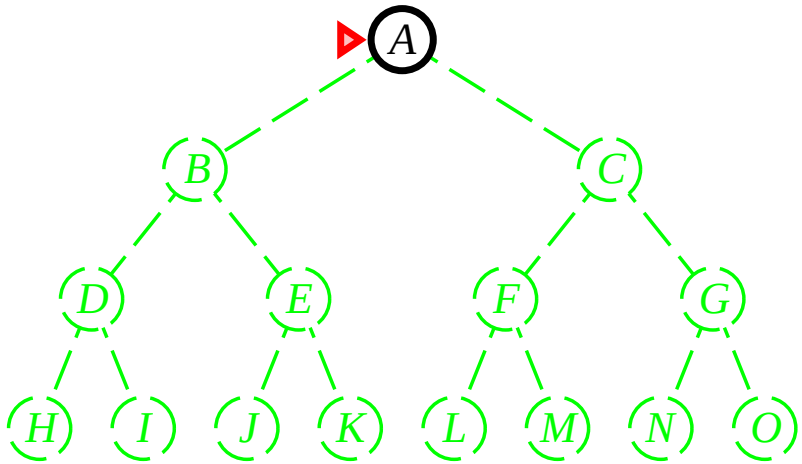
Outline

Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

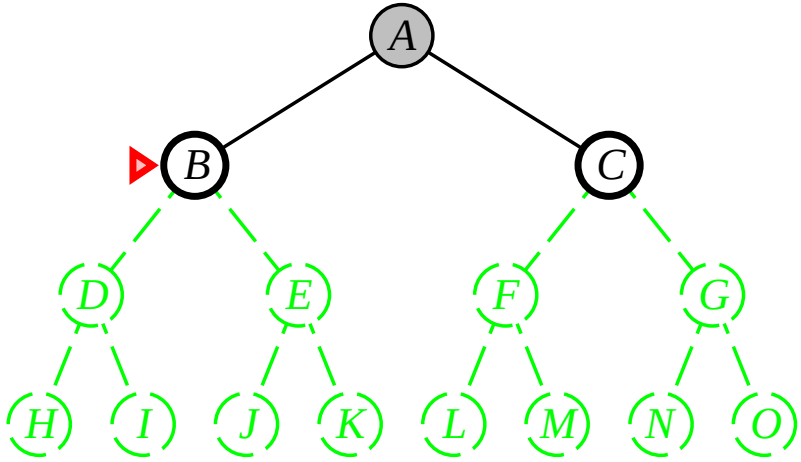


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

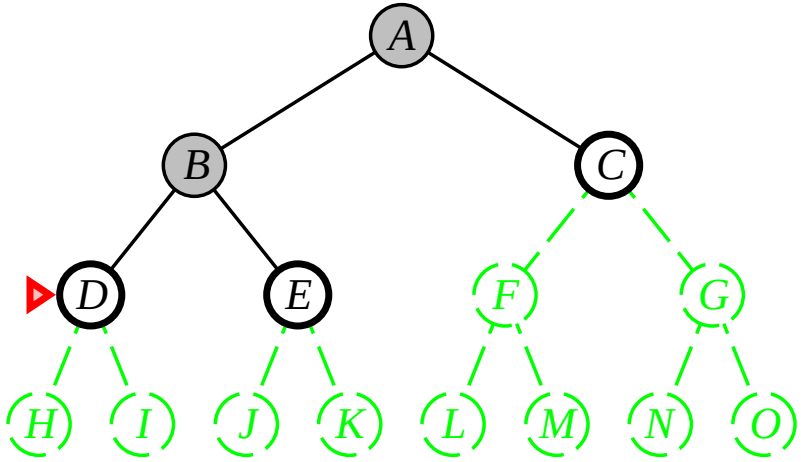


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

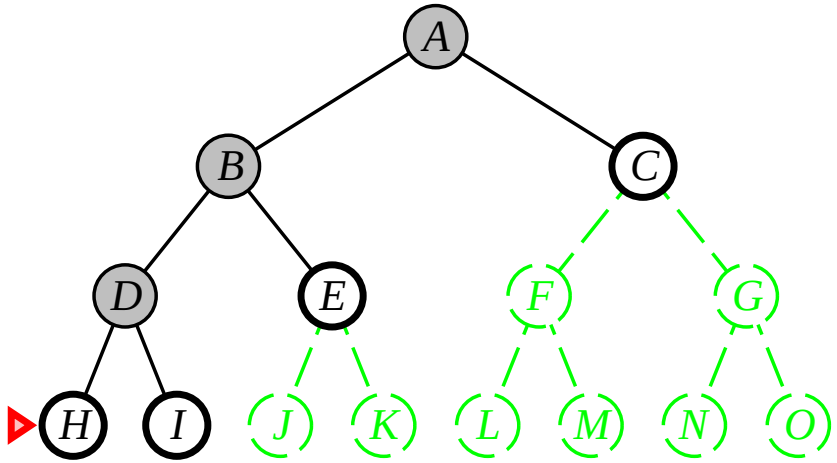


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

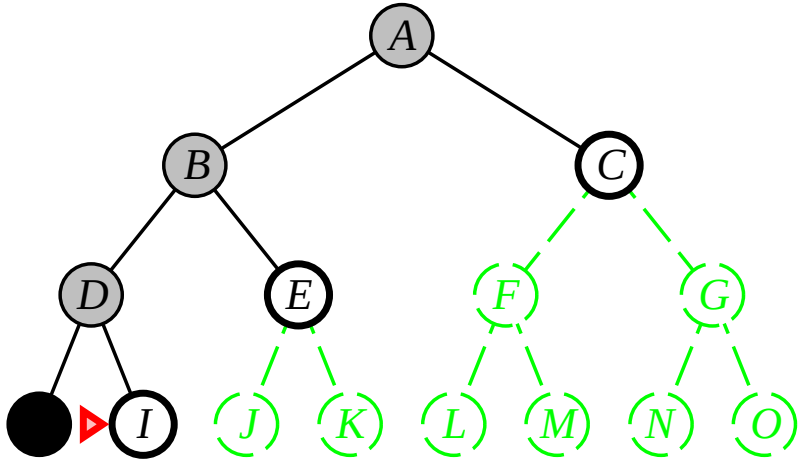


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

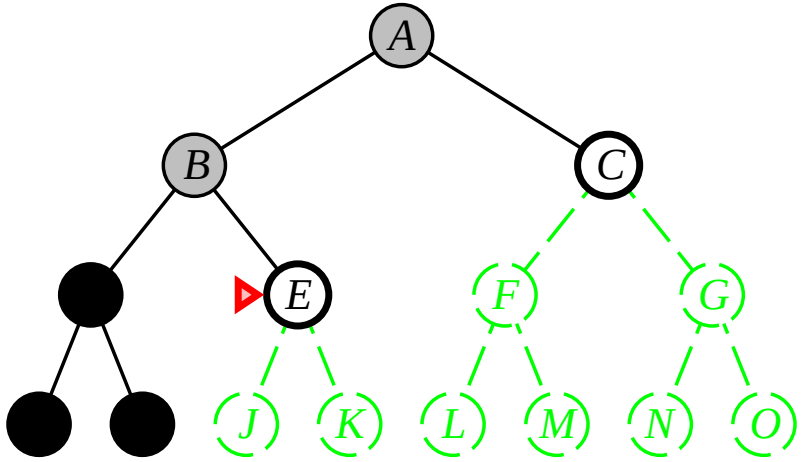


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

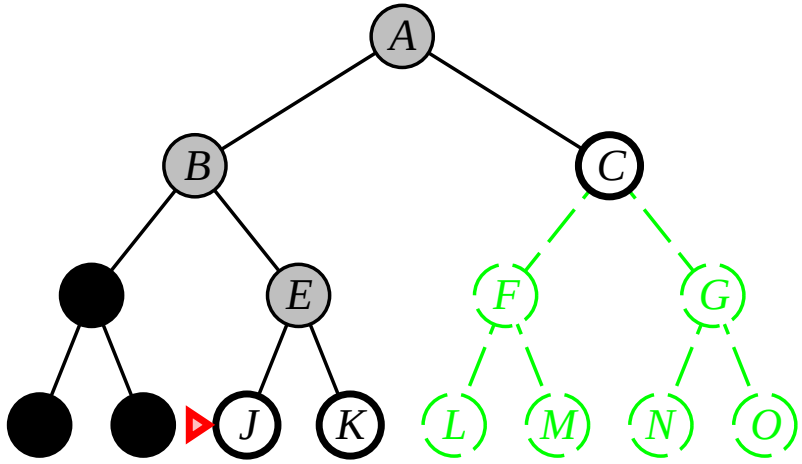


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

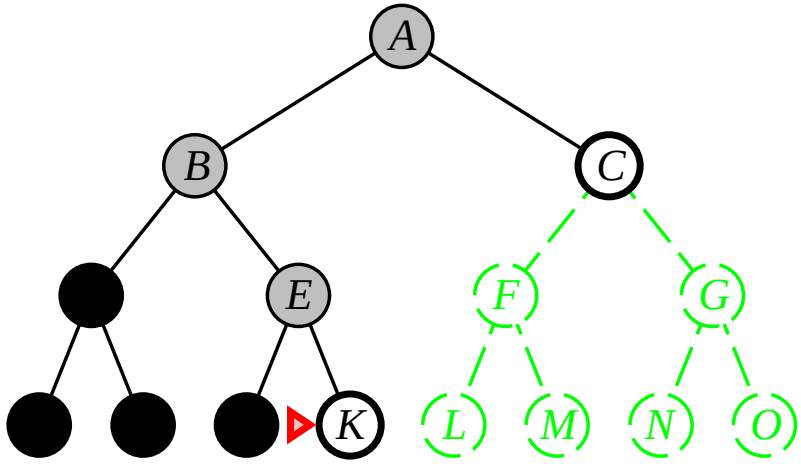


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

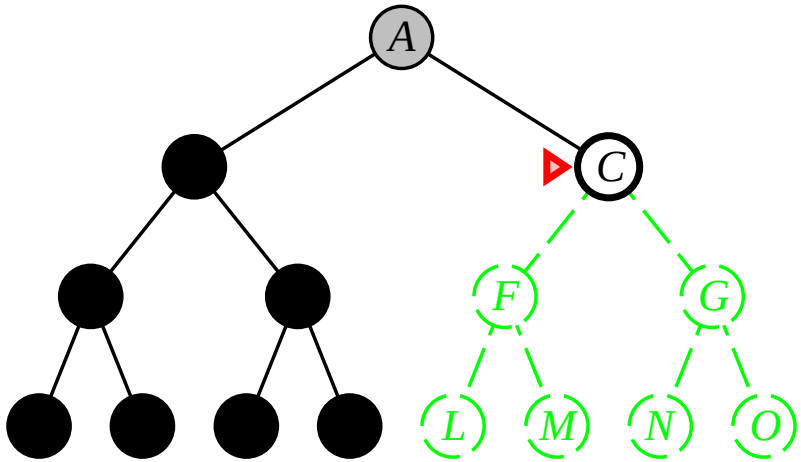


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

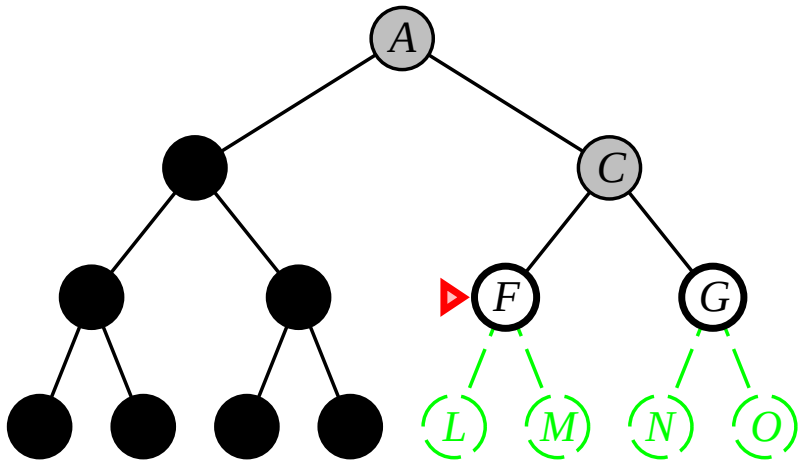


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

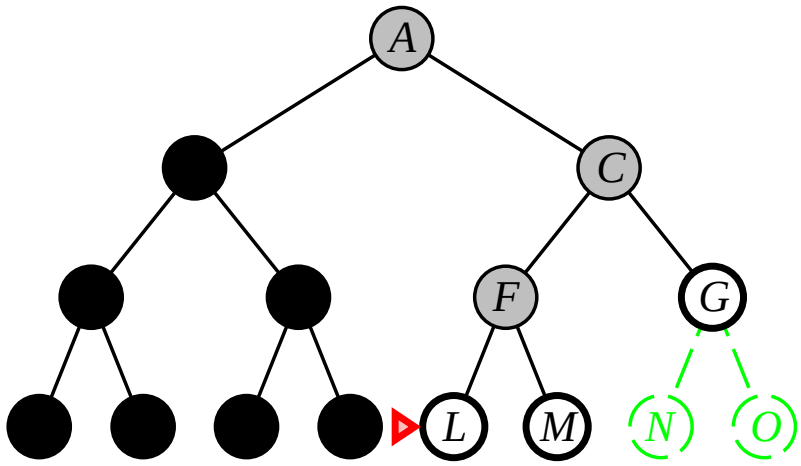


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.

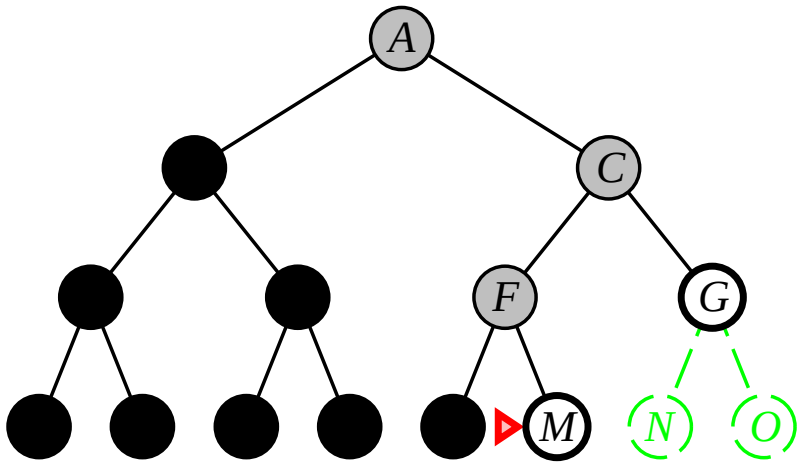


Depth-first search

Prinsip algoritma depth-first search

Lakukan node expansion terhadap node di *frontier* yang paling jauh dari *root*.

- Implementasi: *frontier* adalah *stack* (Last In First Out).
- Hasil *node expansion* ditaruh di depan
- Depth-first search cocok diimplementasikan secara rekursif.



Sifat depth-first search

- Complete?

Sifat depth-first search

- Complete? Tidak, *bisa* gagal jika m tak terbatas, atau (untuk TreeSearch) state space dengan *loop*.
- Time complexity?

Sifat depth-first search

- Complete? Tidak, *bisa* gagal jika m tak terbatas, atau (untuk TreeSearch) state space dengan *loop*.
- Time complexity? $O(b^m) \rightarrow$ jika $m \gg d$, parah!

Sifat depth-first search

- Complete? Tidak, *bisa* gagal jika m tak terbatas, atau (untuk TreeSearch) state space dengan *loop*.
- Time complexity? $O(b^m) \rightarrow$ jika $m \gg d$, parah!
- Space complexity?

Sifat depth-first search

- Complete? Tidak, *bisa* gagal jika m tak terbatas, atau (untuk TreeSearch) state space dengan *loop*.
- Time complexity? $O(b^m) \rightarrow$ jika $m \gg d$, parah!
- Space complexity? $O(bm) \rightarrow$ *linear space*!

Sifat depth-first search

- Complete? Tidak, *bisa* gagal jika m tak terbatas, atau (untuk TreeSearch) state space dengan *loop*.
- Time complexity? $O(b^m) \rightarrow$ jika $m \gg d$, parah!
- Space complexity? $O(bm) \rightarrow$ *linear space*!
- Optimal?

Sifat depth-first search

- Complete? Tidak, *bisa* gagal jika m tak terbatas, atau (untuk TreeSearch) state space dengan *loop*.
- Time complexity? $O(b^m) \rightarrow$ jika $m \gg d$, parah!
- Space complexity? $O(bm) \rightarrow$ *linear space*!
- Optimal? Tidak.

Depth-first search mengatasi masalah space

Mis: 1 node memakan 1000 byte, dan $b = 10$
Jika $d = 12$, space yang dibutuhkan hanya 118 kilobyte ...
bandingkan dengan 10 petabyte!

Variasi depth-first search

- Backtracking search: lakukan *node expansion* satu successor setiap waktu. Jika gagal, *backtrack* dan coba successor yang lain.
- Depth-limited search: Batasi kedalaman maksimal yang dilihat adalah ℓ.
  - Mengatasi masalah untuk *state space* tak terbatas.
  - Sayangnya, ada unsur *incompleteness* baru, jika $\ell < d$.
  - Biasanya d tidak diketahui (tapi bisa ada estimasi, mis. *diameter* state space).

Implementasi rekursif *depth-limited search*

function RECURSIVEDLS (*node*, *problem*, *limit*) **returns** *solution* or *failure*/*cutoff*

```
if GOALTEST[problem](STATE[node]) then return SOLUTION(node)
else if limit = 0 then return cutoff
else
    cutoff_occurred? ← false
    for each successor in EXPAND(node,problem) do
        result ← RECURSIVEDLS(successor,problem,limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

function DEPTHLIMITEDSEARCH (*problem*, *limit*) **returns** *solution* or *failure*/*cutoff*

return RECURSIVEDLS(MAKENODE(INITIALSTATE[*problem*]), *problem*, *limit*)

Perhatikan perbedaan antara *cutoff* dan *failure*.

Outline

Iterative-deepening search

Prinsip algoritma iterative-deepening search

Lakukan depth-limited search secara bertahap dengan nilai ℓ yang *incremental*.

- Strategi ini menggabungkan manfaat *depth* dan *breadth* first: *space complexity* linier dan *completeness* terjamin!
- Lakukan depth-limited search dengan $\ell = 0, 1, 2, \ldots$ sampai tidak *cutoff*.

function ITERATIVEDEEPENINGSEARCH (*problem*) **returns** *solution* or *failure*

for *depth* $\rightarrow$ 0 **to** ∞ **do**
  result $\rightarrow$ DEPTHLIMITEDSEARCH(*problem*, *depth*)
  if *result* $\neq$ *cutoff* **then return** *result*

Contoh iterative-deepening search

$\ell = 0$

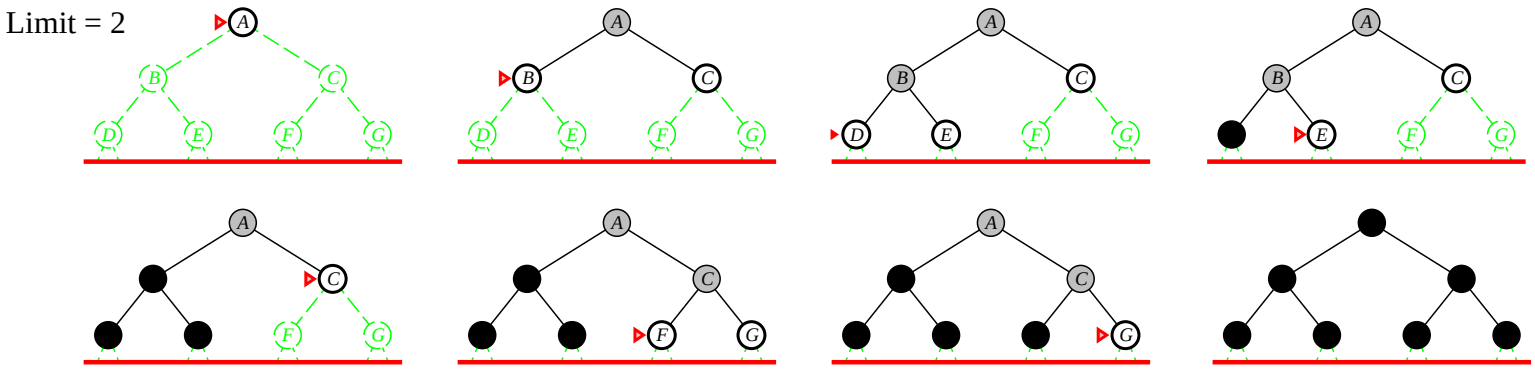
Limit = 0

Contoh iterative-deepening search

$\ell = 1$

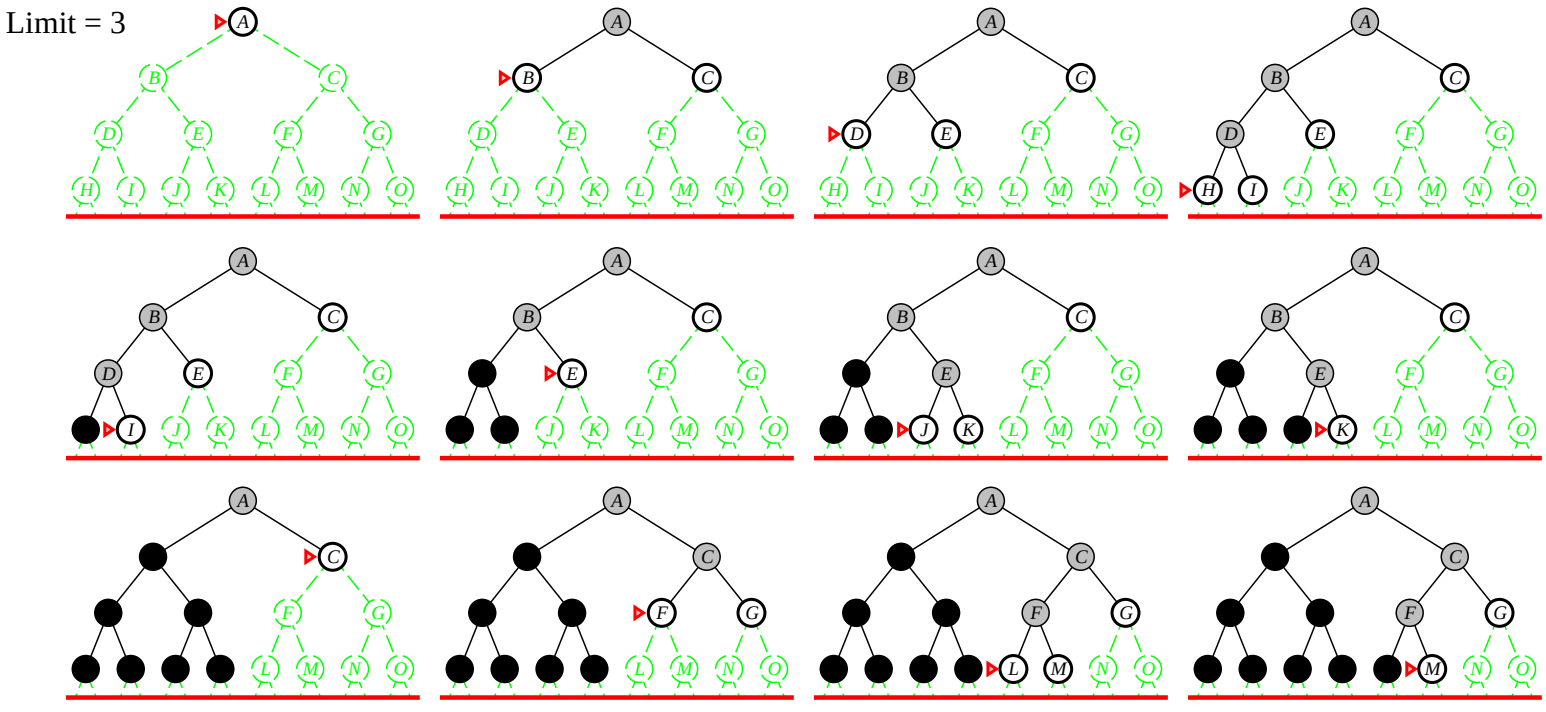
Contoh iterative-deepening search

$\ell = 2$



Contoh iterative-deepening search

$\ell = 3$



Sifat iterative-deepening search

- Complete?

Sifat iterative-deepening search

- Complete? Ya, jika branching factor finite.

Sifat iterative-deepening search

- Complete? Ya, jika branching factor finite.
- Time complexity?

Sifat iterative-deepening search

- Complete? Ya, jika branching factor finite.
- Time complexity?
  $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space complexity?

Sifat iterative-deepening search

- Complete? Ya, jika branching factor finite.
- Time complexity?
  $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space complexity? $O(bd)$

Sifat iterative-deepening search

- Complete? Ya, jika branching factor finite.
- Time complexity?
  $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space complexity? $O(bd)$
- Optimal?

Sifat iterative-deepening search

- Complete? Ya, jika branching factor finite.
- Time complexity?
  $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space complexity? $O(bd)$
- Optimal? Ya, jika semua *step cost* sama.

Kinerja iterative-deepening search

Secara sekilas, strategi ini kelihatan tidak efisien, atau boros: banyak usaha terulang!

Keborosan pada Iterative-deepening search secara umum tidaklah terlalu parah.

$N(IDS) = db + (d-1)b^2 + \ldots + (1)b^d$
$N(BFS) = b + b^2 + \ldots + b^d$

Untuk $b = 10$ dan $d = 5$:

$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$

Pada umumnya, *iterative deepening search* adalah *uninformed search strategy* yang terbaik jika *state space* besar dan kedalaman solusi (d) tidak diketahui.

Perbandingan strategi pencarian

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Ya*	Ya*	Tidak	Tidak	Ya*
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$
Optimal?	Tidak*	Ya	Tidak	Tidak	Ya*