IKI 30320: Sistem Cerdas
Kuliah 4: Uninformed Search Strategies (Rev.)

Ruli Manurung

Fakultas Ilmu Komputer
Universitas Indonesia

5 September 2007

Outline

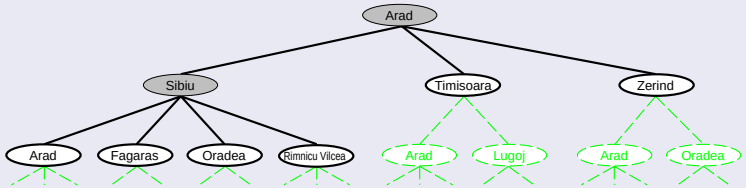
1. Ulasan
2. Breadth-first
3. Uniform-cost
4. Depth-first
5. Iterative-deepening
6. Pengulangan *state*
7. Ringkasan

Problem-solving agent & search

Sebuah problem-solving agent memecahkan sebuah masalah dalam 2 tahap:

- Goal & problem formulation: masalah dinyatakan sebagai sebuah state space, yang sering direpresentasikan dalam bentuk graph.
  Action adalah abstraksi tindakan yang dapat diambil
  State adalah abstraksi keadaan yang dapat terjadi
- Solution search: solusi diperoleh dengan mencari rangkaian tindakan (*action sequence*) yang membawa *agent* ke *goal state*.

Proses pencarian solusi sebagai search tree



Algoritma penelusuran search tree

1. Pada awalnya, *fringe* = himpunan node yang mewakili *initial state*.
2. Pilih satu node dari *fringe* sebagai current node (Kalau *fringe* kosong, selesai dengan gagal).
3. Jika node tsb. lolos *goal test*, selesai dengan sukses!
4. Jika tidak, lakukan node expansion terhadap *current node* tsb. Tambahkan semua node yang dihasilkan ke *fringe*.
5. Ulangi langkah 2.

function TREESEARCH (*problem*, *fringe*) **returns** *solution* or *failure*

```
fringe ← INSERT(MAKENODE(INITIALSTATE(problem)),fringe)
loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVEFIRST(fringe)
    if GOALTEST(problem) applied to STATE(node) succeeds
        then return SOLUTION(node)
    fringe ← INSERTALL(EXPAND(node,problem),fringe)
```

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: jumlah node yang di-generate.
  - space complexity: jumlah maksimum node di dalam memory.
  - optimality: apakah solusi dengan minimum cost pasti ditemukan?
- Time & space complexity diukur berdasarkan
  - b - branching factor dari search tree
  - d - depth (kedalaman) dari solusi optimal
  - m - kedalaman maksimum dari search tree (bisa infinite!)

Uninformed search strategies

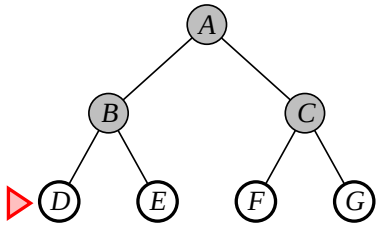
- Uninformed strategy hanya menggunakan informasi dari definisi masalah.
- Bisa diterapkan secara generik terhadap semua jenis masalah yang bisa direpresentasikan dalam sebuah state space.
- Ada beberapa jenis:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited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

Breadth-first search

Prinsip algoritma breadth-first search

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

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



Sifat breadth-first search

- Complete? Ya, jika b terbatas
- Time complexity?
  $b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \rightarrow$ eksponensial dlm. d.
- Space complexity? $O(b^{d+1})$, karena semua node yang di-*generate* harus disimpan.
- Optimal? Ya, jika semua *step cost* sama, tapi pada umumnya tidak optimal.

Masalah utama breadth-first search adalah space

Mis: 1 node memakan 1000 byte, dan $b = 10$
Jika $d = 6$, ada 10^7 node $\approx$ 10 gigabyte.
Jika $d = 12$, ada 10^{13} node $\approx$ 10 petabyte!

Uniform-cost search

Prinsip algoritma uniform-cost search

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

- Implementasi: *fringe* 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*.
- Bandingkan dengan shortest-path algorithm-nya *Dijkstra*!

IKI30320
Kuliah 4
5 Sep 2007
Ruli Manurung
Ulasan
Breadth-first
Uniform-cost
Depth-first
Iterative-deepening
Pengulangan *state*
Ringkasan

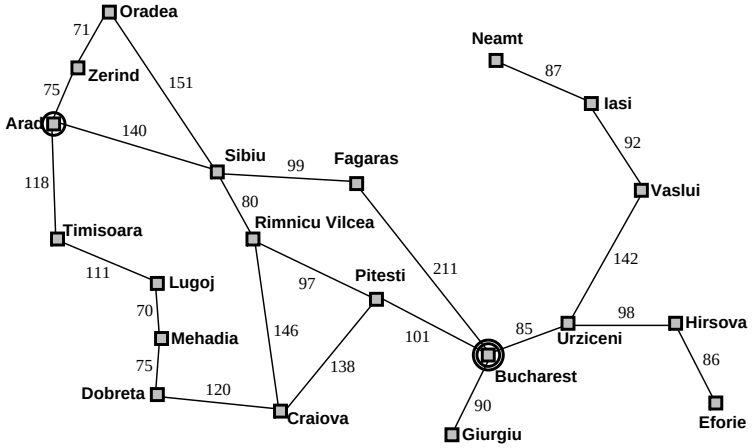
Sifat uniform-cost search

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

IKI30320
Kuliah 4
5 Sep 2007
Ruli Manurung
Ulasan
Breadth-first
Uniform-cost
Depth-first
Iterative-deepening
Pengulangan *state*
Ringkasan

Contoh uniform-cost search

Coba gunakan uniform-cost search untuk mencari optimal solution dari Arad ke Bucharest!



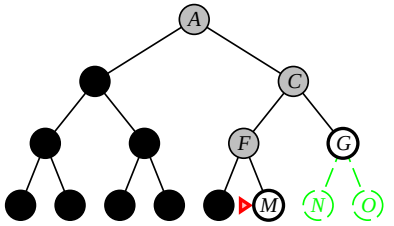
IKI30320
Kuliah 4
5 Sep 2007
Ruli Manurung
Ulasan
Breadth-first
Uniform-cost
Depth-first
Iterative-deepening
Pengulangan *state*
Ringkasan

Depth-first search

Prinsip algoritma depth-first search

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

- Implementasi: *fringe* adalah sebuah *stack*, data struktur LIFO (Last In First Out)
- Hasil *node expansion* ditaruh di depan
- Depth-first search sangat cocok diimplementasikan secara rekursif.



Sifat depth-first search

- Complete? Tidak, *bisa* gagal jika m tak terbatas, atau 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-per-satu. Jika gagal *backtrack* dan coba nilai *successor function* 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* suatu graph).

Implementasi rekursif *depth-limited search*

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

```
cutoff_occurred? ← false
if GOALTEST[problem](STATE[node]) then return SOLUTION(node)
else if DEPTH[node] = limit then return cutoff
else for each successor in EXPAND(node,problem) do
    result ← RECURSIVEDLS(successor,problem,limit)
    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*.

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*.

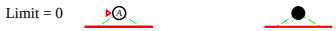
function ITERATIVEDEEPENINGSEARCH (*problem*) **returns** *solution* or *failure*

```
for depth → 0 to ∞ do
    result → DEPTHLIMITEDSEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Contoh iterative-deepening search

Ulasan
Breadth-first
Uniform-cost
Depth-first
Iterative-deepening
Pengulangan *state*
Ringkasan

$\ell = 0$



Contoh iterative-deepening search

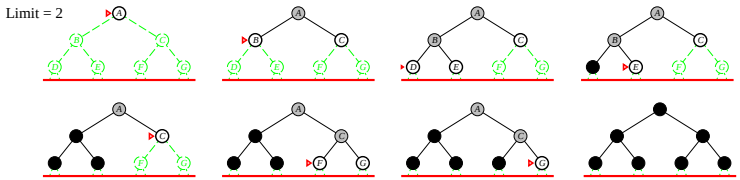
Ulasan
Breadth-first
Uniform-cost
Depth-first
Iterative-deepening
Pengulangan *state*
Ringkasan

$\ell = 1$

Contoh iterative-deepening search

Ulasan
Breadth-first
Uniform-cost
Depth-first
Iterative-deepening
Pengulangan *state*
Ringkasan

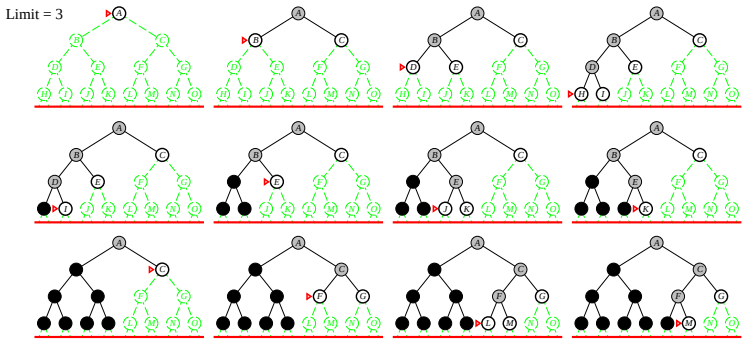
$\ell = 2$



Contoh iterative-deepening search

Ulasan
Breadth-first
Uniform-cost
Depth-first
Iterative-deepening
Pengulangan *state*
Ringkasan

$\ell = 3$



Sifat iterative-deepening search

- Complete? Ya.
- Time complexity? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space complexity? $O(bd)$
- Optimal? Ya, jika semua *step cost* sama. Bisa dimodifikasi spt. *uniform-cost tree*, namanya iterative lengthening search.

Kinerja iterative-deepening search

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

Iterative-deepening search malah lebih cepat dari breadth-first search!

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

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

$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100.$

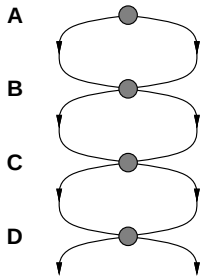
Pada umumnya, *iterative deepening search* adalah *uninformed search strategy* yang terbaik jika *state space* besar dan kedalaman solusi (d) tidak diketahui.

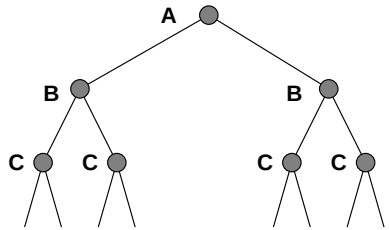
Perbandingan strategi pencarian

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

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 closed list (*fringe* = open list).
- Modifikasi algoritma TREESEARCH dengan *closed list* menjadi GRAPHSEARCH.

Algoritma GRAPHSEARCH

function GRAPHSEARCH (*problem*, *fringe*) **returns** *solution* or *failure*

closed ← {}
fringe ← INSERT(MAKENODE(INITIALSTATE(*problem*)),*fringe*)
loop do
  if EMPTY?(*fringe*) **then return** failure
  node ← REMOVEFIRST(*fringe*)
  if GOALTEST(*problem*) applied to STATE(*node*) succeeds
    then return SOLUTION(*node*)
  if STATE[*node*] $\notin$ *closed* **then**
    add STATE[*node*] to *closed*
    fringe ← INSERTALL(EXPAND(*node*,*problem*),*fringe*)

Sifat GRAPHSEARCH

- Time complexity: sama, jika kita asumsi operasi STATE[*node*] $\notin$ *closed* = $O(1)$ (implementasi dengan hashtable?)
- Space complexity: DFS dan IDS tidak lagi linier!
- GRAPHSEARCH tidak mencatat *path* menuju suatu *state*. Ini mempengaruhi sifat *optimality* suatu strategi:
  - *Uniform-cost* dan *breadth-first search* dengan *step cost* konstanta masih optimal (kenapa?).
  - Untuk variasi *Depth-first* dan *iterative-deepening search*, jika *state* mengulang ditemukan, periksa apakah *path cost*-nya lebih kecil → *update* info *node* dan anak-anaknya!

Ringkasan

- Breadth-first search: *completeness* terjamin, tapi rakus memory.
- Uniform-cost search: mirip BFS, *optimality* terjamin jika *cost path* $\geq \epsilon$ untuk $\epsilon > 0$.
- Depth-first search: *Space complexity* linier, tetapi tidak *complete* (maupun *optimal*).
- Depth-limited search: mirip DFS, tetapi kedalaman search dibatasi sampai ℓ.
- Iterative-deepening search: lakukan DLS secara bertahap dengan $\ell = 0, 1, 2, \ldots$.
- Pengulangan *state* bisa dihindari dengan mencatat *state* yang sudah pernah dicoba.
  TREESEARCH → GRAPHSEARCH.