Uninformed Search

Bùi Tiến Lên

01/09/2019

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Uninformed search strategies

- No additional information about states beyond that provided in the problem definition
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- Also called **blind search**
- Each strategy is an modified instance of the general **tree/graph search** algorithm

Breadth-first search

Breadth-first search

- **Breadth-first search** (BFS) is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded

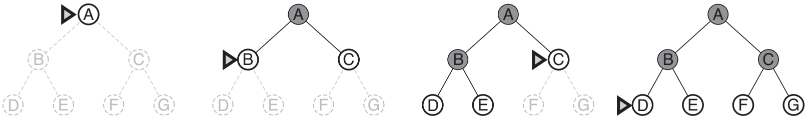


Figure 1: Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Algorithm

- Frontier is a FIFO queue

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← node
  explored ← ∅
  loop do
    if frontier = ∅ then return failure
    node ← POP(frontier)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Properties of breadth-first search

- **Time**: $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- **Space**: $O(b^d)$ ($O(b^{d-1})$ for explored and $O(b^d)$ for frontier)
- **Complete**: Yes (if b is finite)
- **Optimal**: not optimal in general

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-cost search

Uniform-cost search

- **Uniform-cost search** (UCS) expands the node n with the **lowest path cost** $g(n)$
- Implementation: frontier is a priority queue ordered by g
  - Equivalent to Dijkstra's algorithm
- The goal test is applied to a node when it is selected for expansion
- A test is added in case a better path is found to a node currently on the frontier.

Algorithm

- Frontier is a priority queue

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← node # a priority queue ordered by PATH-COST
  explored ← ∅
  loop do
    if frontier = ∅ then return failure
    node ← POP(frontier) # chooses the lowest-cost node in frontier
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in explored with higher PATH-COST then
        replace that frontier node with child
```

Properties of uniform-cost search

- **Time**: $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$ where C^* is the cost of the optimal solution
- **Space**: $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$
- **Complete** Yes, if step cost $\geq \epsilon$ (small positive constant)
- **Optimal** Yes, nodes expanded in increasing order of $g(n)$

Proof

Using contradiction method

- Suppose UCS terminates at a goal state n with a path cost $g(n) = C$.
- If C is not the optimal value then there exists another unexplored goal state n' with $g(n') < C$
- Therefore, there must exist a node n'' on the frontier that is on the optimal path to n' (graph separation property)
- But $g(n'') < g(n') < g(n) \rightarrow n''$ must expand before n, a contradiction.

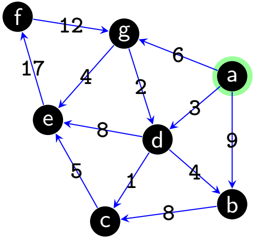
■

Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)

Figure 2: Find a shortest path from a to f

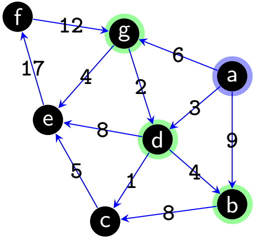


Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)
a(0;null)	b(9;a) d(3;a) g(6;a)

Figure 2: Find a shortest path from a to f



Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)
a(0;null)	b(9;a) d(3;a) g(6;a)
d(3;a)	b(7;d) g(6;a) c(4;d) e(11;d)

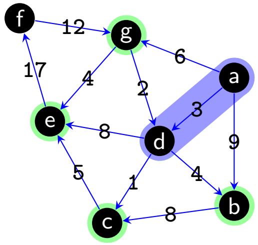


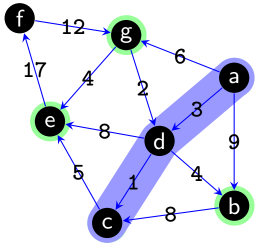
Figure 2: Find a shortest path from a to f

Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)
a(0;null)	b(9;a) d(3;a) g(6;a)
d(3;a)	b(7;d) g(6;a) c(4;d) e(11;d)
c(4;d)	b(7;d) g(6;a) e(9;c)

Figure 2: Find a shortest path from a to f

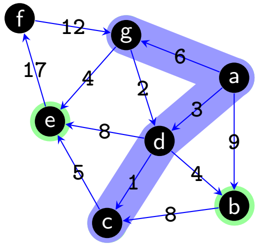


Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)
a(0;null)	b(9;a) d(3;a) g(6;a)
d(3;a)	b(7;d) g(6;a) c(4;d) e(11;d)
c(4;d)	b(7;d) g(6;a) e(9;c)
g(6;a)	b(7;d) e(9;c)

Figure 2: Find a shortest path from a to f

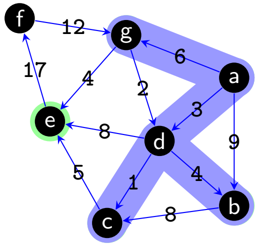


Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)
a(0;null)	b(9;a) d(3;a) g(6;a)
d(3;a)	b(7;d) g(6;a) c(4;d) e(11;d)
c(4;d)	b(7;d) g(6;a) e(9;c)
g(6;a)	b(7;d) e(9;c)
b(7;d)	e(9;c)

Figure 2: Find a shortest path from a to f

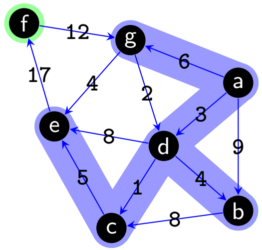


Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)
a(0;null)	b(9;a) d(3;a) g(6;a)
d(3;a)	b(7;d) g(6;a) c(4;d) e(11;d)
c(4;d)	b(7;d) g(6;a) e(9;c)
g(6;a)	b(7;d) e(9;c)
b(7;d)	e(9;c)
e(9;c)	f(26;e)

Figure 2: Find a shortest path from a to f

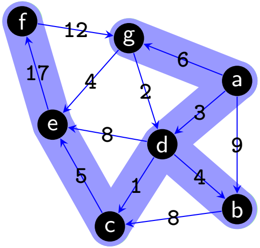


Ilustration

Table 1: Each node has PATH-COST and PARENT

node	frontier
	a(0;null)
a(0;null)	b(9;a) d(3;a) g(6;a)
d(3;a)	b(7;d) g(6;a) c(4;d) e(11;d)
c(4;d)	b(7;d) g(6;a) e(9;c)
g(6;a)	b(7;d) e(9;c)
b(7;d)	e(9;c)
e(9;c)	f(26;e)
f(26;e)	∅

Figure 2: Find a shortest path from a to f



Depth-first search

Depth-first search

- **Depth-first search** (DFS) expands deepest unexpanded node
- Implementation: frontier is a LIFO Stack

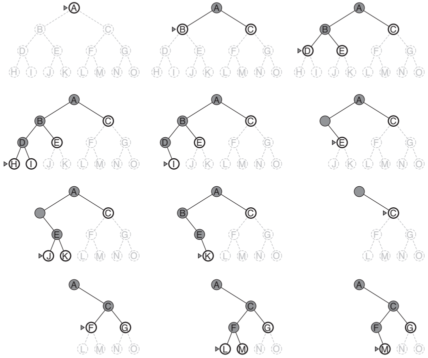


Figure 3: Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Properties of DFS

- **Time**: $O(b^m)$
  - Terrible if m is much larger than d, but if solutions are dense, may be much faster than breadth-first
- **Space**: $O(bm)$, i.e., linear space!
- **Complete**:
  - No in infinite-depth spaces
  - Yes in finite spaces
- **Optimal**: No, the "leftmost" solution, regardless of depth or cost

Depth-limited search

Depth-limited Search

- **Depth-limited Search** (DLS) is a standard DFS with a predetermined depth limit l, i.e., nodes at depth l are treated as if they have no successors
  - infinite problems solved
- Depth limits can be based on knowledge of the problem

Algorithm

- A recursive implementation of depth-limited tree search.

```
function DEPTH-LIMITED-SEARCH(problem, limit)
returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)
function RECURSIVE-DLS(node, problem, limit)
returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred? ← false
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      result ← RECURSIVE-DLS(child, 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
```

Properties

- **Time**
  - $O(b^l)$
- **Space**
  - $O(bl)$
- **Completeness**
  - Maybe no if $l < d$
- **Optimality**
  - No if $l > d$

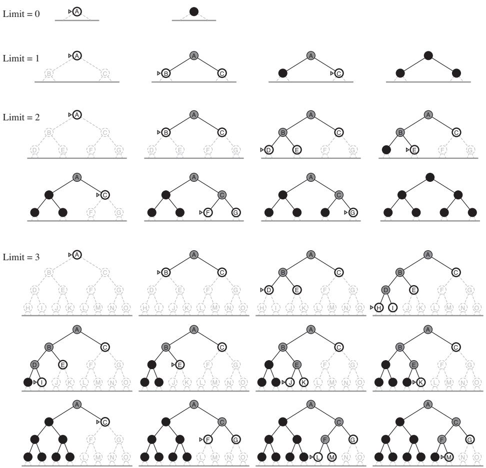
Iterative deepening depth-first search

Iterative deepening depth-first search

- General strategy, often used in combination with depth-first tree search to find the best depth limit
- Gradually increasing the limit until a goal is found
  - The depth limit reaches the depth d of the shallowest goal node.

```
function ITERATIVE-DEEPENING-SEARCH(problem)
returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Illustration



Four iterations of iterative deepening search on a binary tree.

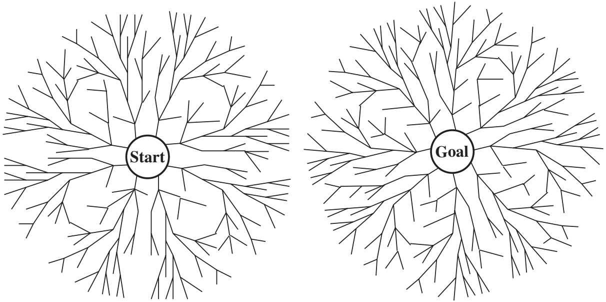
Properties

- **Time complexity**
  - $db^1 + (d-1)b^2 + 1b^d = O(b^d)$
- **Space complexity**
  - $O(bd)$, similar to DFS
- **Completeness**
  - Yes when the branching factor b is finite
- **Optimality**
  - No in general

Bidirectional search

Bidirectional search

- Two simultaneous searches
  - From the **initial state** towards
  - From the **goal state** backwards
- Hoping that two searches meet in the middle



Properties

- **Time and Space complexity**: $O(b^{d/2})$
- **Goal test**: Whether the frontiers of two searches intersect
- **Optimality**: Maybe no
- It sounds attractive, but what is the **tradeoff**?
- Space requirement for the frontiers of at least one search
- Not easy to search backwards (predecessors required)
    - In case there are more than 1 goals
    - Especially if the goal is an abstract description (no queen attacks another queen)

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Itervative Deepen-ing	Bidirec-tional
Complete	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{\frac{d}{2}})$
Space	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{\frac{d}{2}})$
Optimal	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Table 2: Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

References

Goodfellow, I., Bengio, Y., and Courville, A. (2016).
Deep learning.
MIT press.

Nguyen, T. (2018).
Artificial intelligence slides.
Technical report, HCMC University of Sciences.

Russell, S. and Norvig, P. (2016).
Artificial intelligence: a modern approach.
Pearson Education Limited.