

Foundations of Artificial Intelligence

B8. State-Space Search: Depth-first Search & Iterative Deepening

Malte Helmert

University of Basel

March 16, 2026

State-Space Search: Overview

Chapter overview: state-space search

- B1–B3. Foundations
- B4–B8. Basic Algorithms
 - B4. Data Structures for Search Algorithms
 - B5. Tree Search and Graph Search
 - B6. Breadth-first Search
 - B7. Uniform Cost Search
 - B8. Depth-first Search and Iterative Deepening
- B9–B15. Heuristic Algorithms

Depth-first Search

Idea of Depth-first Search

depth-first search:

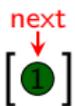
- expands nodes in **opposite order of generation** (LIFO)
- open list implemented as **stack**

↪ **deepest** node expanded first

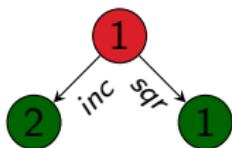
German: Tiefensuche

Depth-first Search Example



open: []

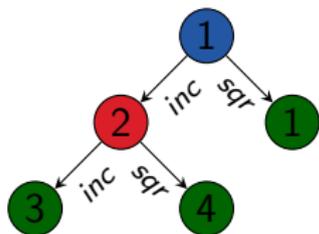
Depth-first Search Example



open: [1 2]

next
↓

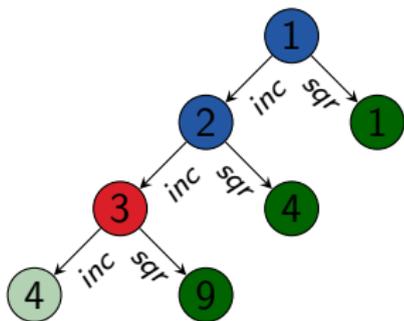
Depth-first Search Example



open: [1 4 3]

next
↓

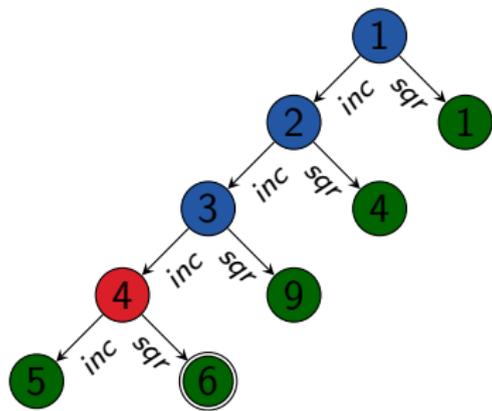
Depth-first Search Example



open: [1 4 9 4]

next
↓

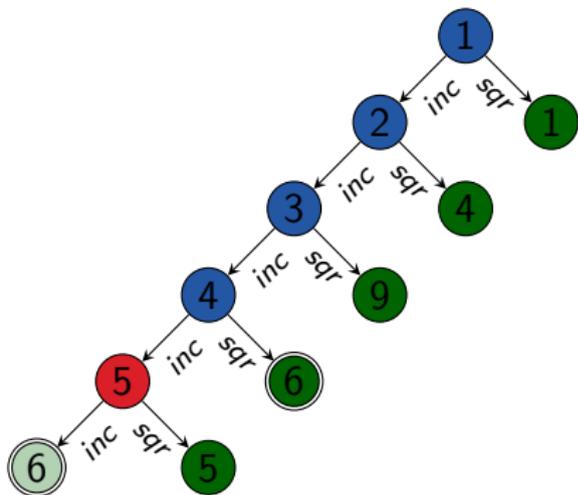
Depth-first Search Example



open: [1 4 9 6 5]

next
↓

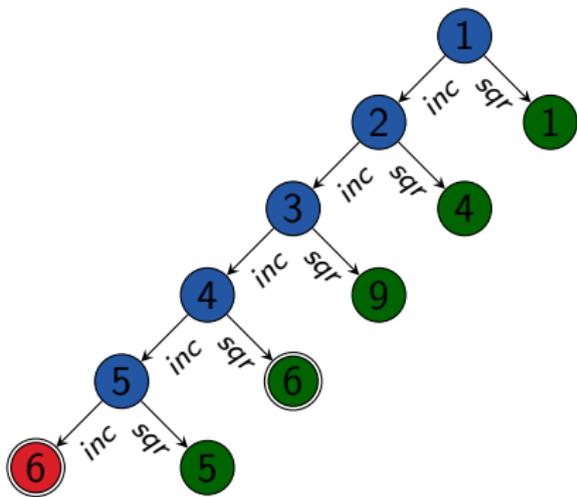
Depth-first Search Example



open: [1 4 9 6 5 6]

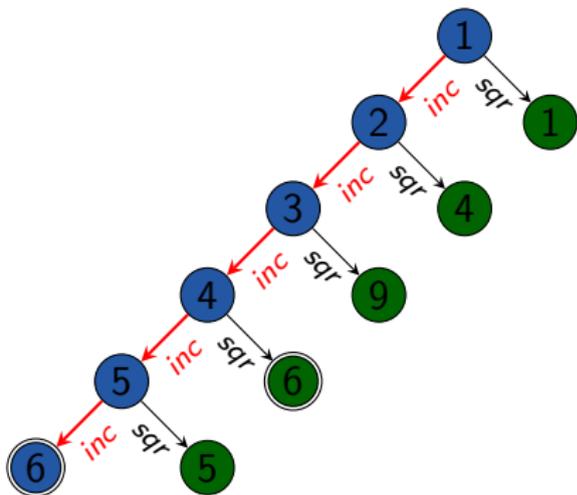
next
↓

Depth-first Search Example



open: [1 4 9 6 5]

Depth-first Search Example



open: [1 4 9 6 5]

Depth-first Search: Some Properties

- almost always implemented as a **tree search** (we will see why)
- **not complete, not semi-complete, not optimal** (Why?)
- complete for **acyclic** state spaces,
e.g., if state space directed tree

Reminder: Generic Tree Search Algorithm

reminder from Chapter B5:

Generic Tree Search

```
open := new OpenList
open.insert(make_root_node())
while not open.is_empty():
    n := open.pop()
    if is_goal(n.state):
        return extract_path(n)
    for each  $\langle a, s' \rangle \in$  succ(n.state):
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable
```

Depth-first Search (Non-recursive Version)

depth-first search (non-recursive version):

Depth-first Search (Non-recursive Version)

```
open := new Stack
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_back()
    if is_goal(n.state):
        return extract_path(n)
    for each  $\langle a, s' \rangle \in \text{succ}(n.\text{state})$ :
        n' := make_node(n, a, s')
        open.push_back(n')
```

return unsolvable

Non-recursive Depth-first Search: Discussion

discussion:

- there isn't much wrong with this pseudo-code
(as long as we ensure to release nodes that are no longer required
when using programming languages without garbage collection)
- however, depth-first search as a **recursive algorithm**
is simpler and more efficient
- ↪ CPU stack as implicit open list
- ↪ no search node data structure needed

Depth-first Search (Recursive Version)

```
function depth_first_search(s)  
if is_goal(s):  
    return ⟨⟩  
for each ⟨a, s'⟩ ∈ succ(s):  
    solution := depth_first_search(s')  
    if solution ≠ none:  
        solution.push_front(a)  
    return solution  
return none
```

main function:

Depth-first Search (Recursive Version)

```
return depth_first_search(init())
```

Depth-first Search: Complexity

time complexity:

- If the state space includes paths of length m , depth-first search can generate $O(b^m)$ nodes, even if much shorter solutions (e.g., of length 1) exist.
- On the other hand: in the **best case**, solutions of length ℓ can be found with $O(b\ell)$ generated nodes. (Why?)
- improvable to $O(\ell)$ with **incremental successor generation**

Depth-first Search: Complexity

time complexity:

- If the state space includes paths of length m , depth-first search can generate $O(b^m)$ nodes, even if much shorter solutions (e.g., of length 1) exist.
- On the other hand: in the **best case**, solutions of length ℓ can be found with $O(b\ell)$ generated nodes. (Why?)
- improvable to $O(\ell)$ with **incremental successor generation**

space complexity:

- only need to store nodes **along currently explored path** (“along”: nodes on path and their children)
- ↪ space complexity $O(bm)$ if m maximal search depth reached
- low memory complexity main reason why depth-first search interesting despite its disadvantages

Iterative Deepening

Idea of Depth-limited Search

depth-limited search:

- parameterized with **depth limit** $\ell \in \mathbb{N}_0$
- behaves like depth-first search, but **prunes** (does not expand) search nodes at depth ℓ
- not very useful on its own, but **important ingredient** of more useful algorithms

German: tiefenbeschränkte Suche

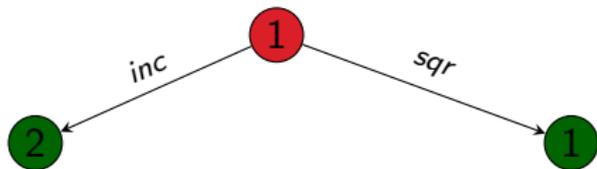
Depth-limited Search Example

Consider depth limit $\ell = 2$.



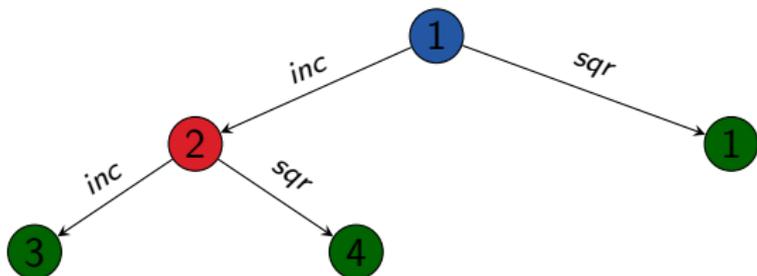
Depth-limited Search Example

Consider depth limit $\ell = 2$.



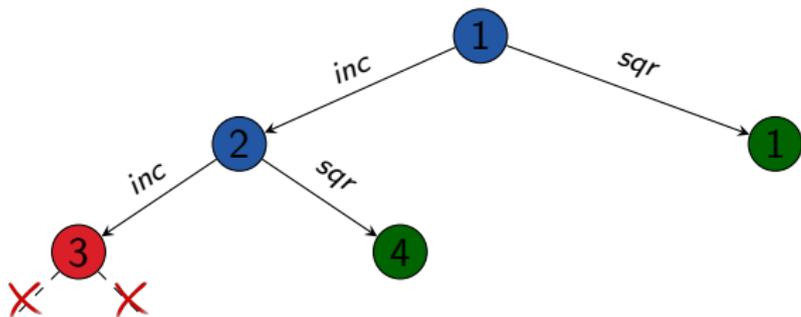
Depth-limited Search Example

Consider depth limit $\ell = 2$.



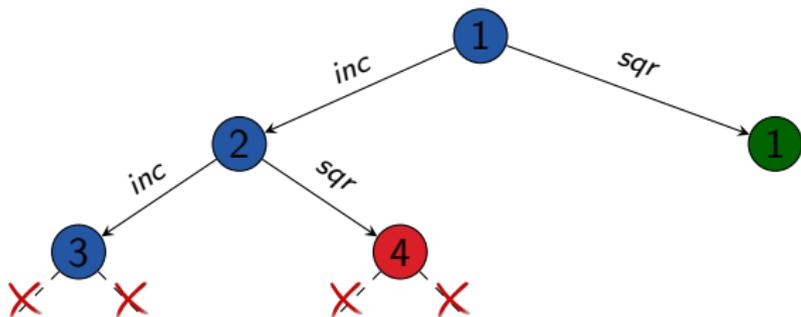
Depth-limited Search Example

Consider depth limit $\ell = 2$.



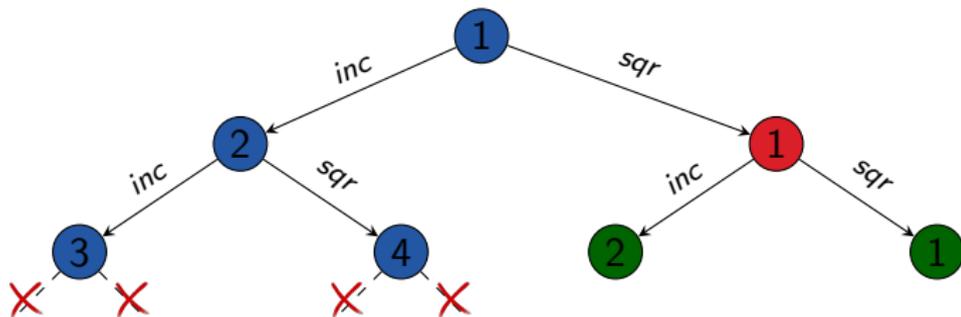
Depth-limited Search Example

Consider depth limit $\ell = 2$.



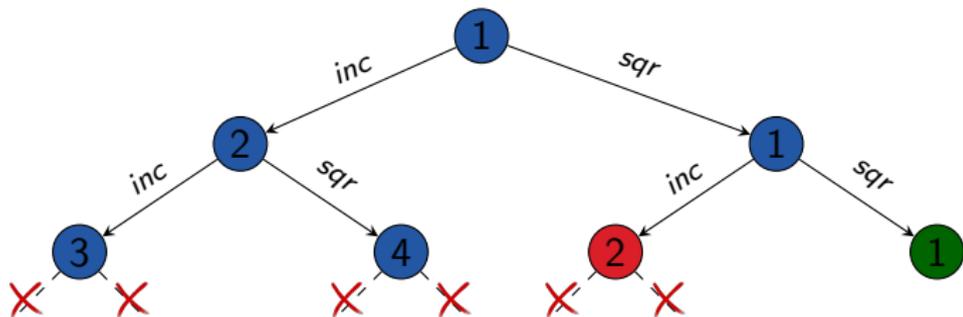
Depth-limited Search Example

Consider depth limit $\ell = 2$.



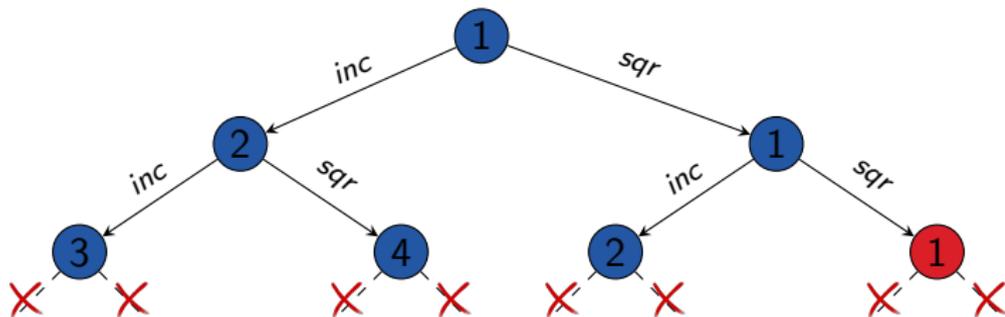
Depth-limited Search Example

Consider depth limit $\ell = 2$.



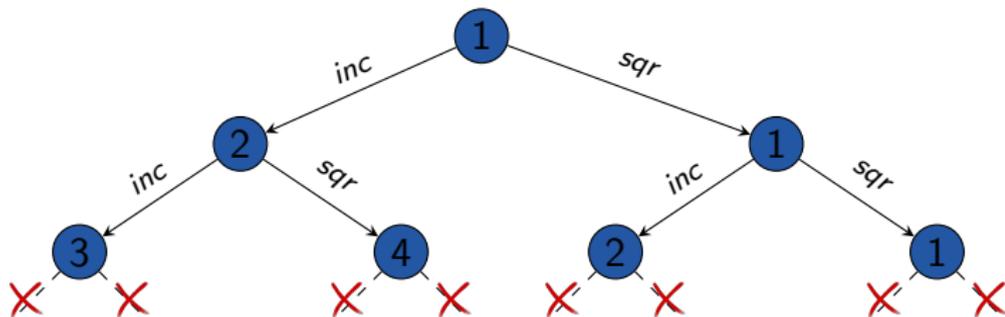
Depth-limited Search Example

Consider depth limit $\ell = 2$.



Depth-limited Search Example

Consider depth limit $\ell = 2$.



Depth-limited Search: Pseudo-Code

```
function depth_limited_search(s, depth_limit):
```

```
  if is_goal(s):
```

```
    return  $\langle \rangle$ 
```

```
  if depth_limit > 0:
```

```
    for each  $\langle a, s' \rangle \in \text{succ}(s)$ :
```

```
      solution := depth_limited_search(s', depth_limit - 1)
```

```
      if solution  $\neq$  none:
```

```
        solution.push_front(a)
```

```
        return solution
```

```
return none
```

Iterative Deepening Depth-first Search

iterative deepening depth-first search (iterative deepening DFS):

- **idea:** perform a sequence of depth-limited searches with increasing depth limit
- sounds wasteful (each iteration repeats all the useful work of all previous iterations)
- in fact overhead acceptable (\rightsquigarrow analysis follows)

Iterative Deepening DFS

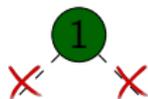
```
for depth_limit  $\in$  {0, 1, 2, ... }:  
    solution := depth_limited_search(init(), depth_limit)  
    if solution  $\neq$  none:  
        return solution
```

German: iterative Tiefensuche

Example

depth limit: 0

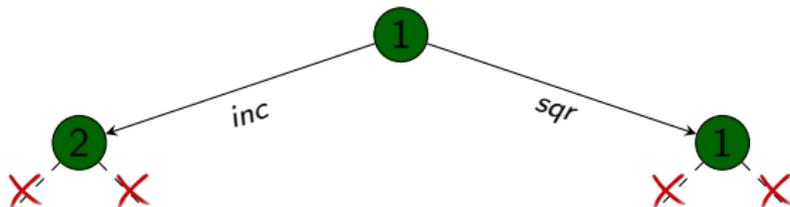
generated nodes: 1



Example

depth limit: 1

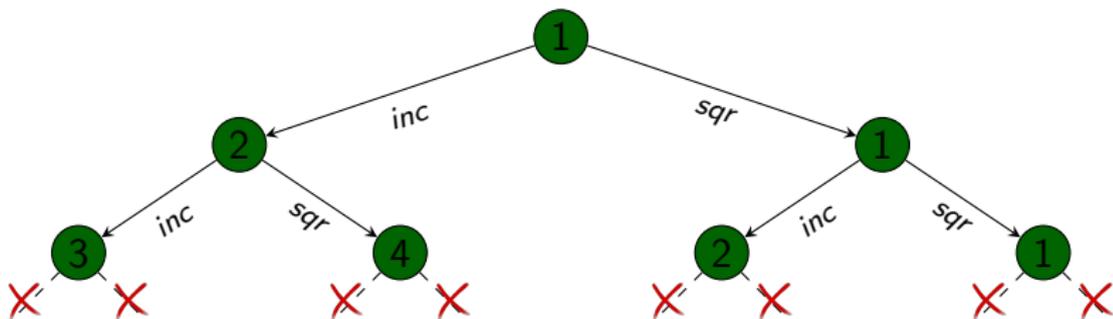
generated nodes: 1+3



Example

depth limit: 2

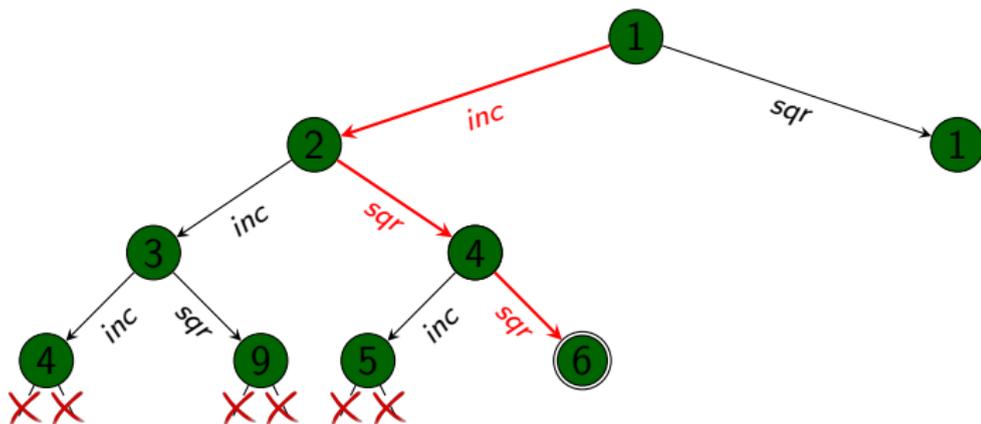
generated nodes: 1+3+7



Example

depth limit: 3

generated nodes: $1+3+7+9=20$



Iterative Deepening DFS: Properties

combines advantages of breadth-first and depth-first search:

- (almost) like BFS: semi-complete (however, not complete)
- like BFS: optimal if all actions have same cost
- like DFS: only need to store nodes along one path
↪ space complexity $O(bd)$, where d minimal solution length
- time complexity only slightly higher than BFS
(↪ analysis soon)

Iterative Deepening DFS: Complexity Example

time complexity (generated nodes):

breadth-first search	$1 + b + b^2 + \dots + b^{d-1} + b^d$
iterative deepening DFS	$(d + 1) + db + (d - 1)b^2 + \dots + 2b^{d-1} + 1b^d$

example: $b = 10$, $d = 5$

breadth-first search	$1 + 10 + 100 + 1000 + 10000 + 100000$ $= 111111$
iterative deepening DFS	$6 + 50 + 400 + 3000 + 20000 + 100000$ $= 123456$

for $b = 10$, only 11% more nodes than breadth-first search

Iterative Deepening DFS: Time Complexity

Theorem (time complexity of iterative deepening DFS)

Let b be the branching factor and d be the minimal solution length of the given state space. Let $b \geq 2$.

Then the *time complexity* of iterative deepening DFS is

$$(d + 1) + db + (d - 1)b^2 + (d - 2)b^3 + \dots + 1b^d = O(b^d)$$

and the *memory complexity* is

$$O(bd).$$

Iterative Deepening DFS: Evaluation

Iterative Deepening DFS: Evaluation

Iterative Deepening DFS is often the method of choice if

- **tree search is adequate** (no duplicate elimination necessary),
- all **action costs** are identical, and
- the **solution depth** is **unknown**.

Summary

Summary

depth-first search: expand nodes in **LIFO** order

- usually as a **tree search**
- easy to implement **recursively**
- very **memory-efficient**
- can be combined with **iterative deepening**
to combine many of the good aspects
of breadth-first and depth-first search

Comparison of Blind Search Algorithms

completeness, optimality, time and space complexity

criterion	search algorithm				
	breadth- first	uniform cost	depth- first	depth- limited	iterative deepening
complete?	yes*	yes	no	no	semi
optimal?	yes**	yes	no	no	yes**
time	$O(b^d)$	$O(b^{\lfloor c^*/\epsilon \rfloor + 1})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
space	$O(b^d)$	$O(b^{\lfloor c^*/\epsilon \rfloor + 1})$	$O(bm)$	$O(b\ell)$	$O(bd)$

- $b \geq 2$ branching factor
- d minimal solution depth
- m maximal search depth
- ℓ depth limit
- c^* optimal solution cost
- $\epsilon > 0$ minimal action cost

remarks:

- * for BFS-Tree: semi-complete
- ** only with uniform action costs