

# Foundations of Artificial Intelligence

## 12. State-Space Search: Depth-first Search & Iterative Deepening

Malte Helmert

University of Basel

March 21, 2022

# State-Space Search: Overview

## Chapter overview: state-space search

- 5.–7. Foundations
- 8.–12. Basic Algorithms
  - 8. Data Structures for Search Algorithms
  - 9. Tree Search and Graph Search
  - 10. Breadth-first Search
  - 11. Uniform Cost Search
  - 12. Depth-first Search and Iterative Deepening
- 13.–19. Heuristic Algorithms

# Depth-first Search

# Depth-first Search

Depth-first search (DFS) expands nodes in opposite order of generation (LIFO).

⇒ deepest node expanded first

⇒ open list implemented as stack

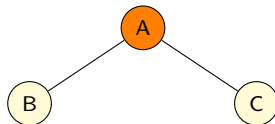
German: Tiefensuche

# Depth-first Search: Example



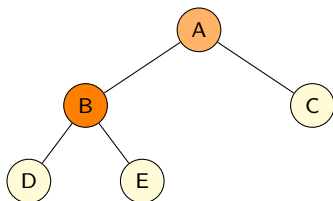
*open:* A

# Depth-first Search: Example



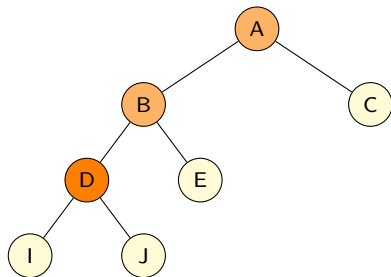
*open:* C, B

## Depth-first Search: Example



*open:* C, E, **D**

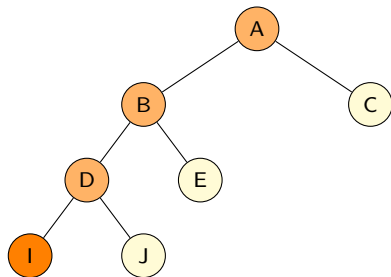
## Depth-first Search: Example



*open:* C, E, J, I

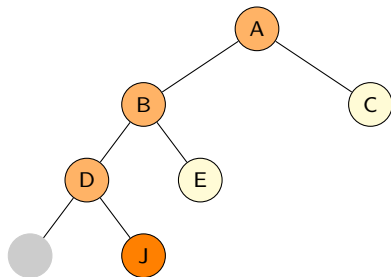


# Depth-first Search: Example



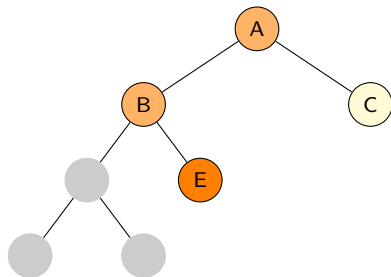
*open:* C, E, J

# Depth-first Search: Example



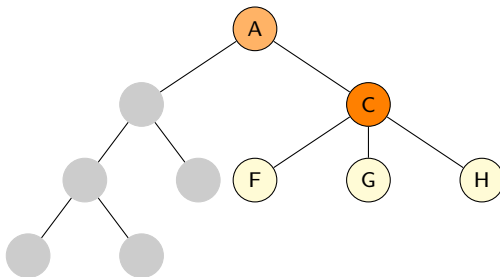
*open:* C, E

# Depth-first Search: Example



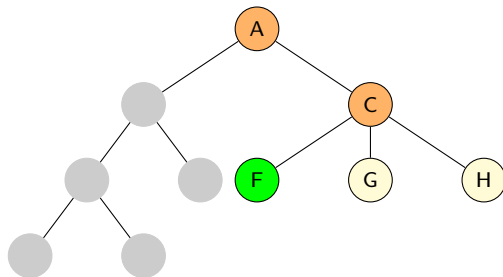
*open:* C

# Depth-first Search: Example



*open:* H, G, **F**

# Depth-first Search: Example



~> solution found!

# 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 9:

## 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  $\langle a, s' \rangle \in \text{succ}(n.state)$ :
        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  $\ell$  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  $\ell$  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

# Depth-bounded Search

## depth-bounded search:

- depth-first search which **prunes** (does not expand) all nodes at a given depth  $d$

↪ not very useful on its own, but important ingredient of more useful algorithms

**German:** tiefenbeschränkte Suche

# Depth-bounded Search: Pseudo-Code

```
function depth_bounded_search(s, depth_bound):  
  if is_goal(s):  
    return ⟨⟩  
  if depth_bound > 0:  
    for each  $\langle a, s' \rangle \in \text{succ}(s)$ :  
      solution := depth_bounded_search(s', depth_bound − 1)  
      if solution ≠ 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-bounded searches with increasing depth bound
- 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_bound  $\in \{0, 1, 2, \dots\}$ :  
    solution := depth_bounded_search(init(), depth_bound)  
    if solution  $\neq$  none:  
        return solution
```

**German:** iterative Tiefensuche



# 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  
     $\rightsquigarrow$  space complexity  $O(bd)$ , where  $d$  minimal solution length
- time complexity only slightly higher than BFS  
    ( $\rightsquigarrow$  analysis soon)

# Iterative Deepening DFS: Example

depth bound: 0



generated in this round: 1

total generated: 1

# Iterative Deepening DFS: Example

depth bound: 1



generated in this round: 1

total generated:  $1 + 1$

# Iterative Deepening DFS: Example

depth bound: 1



generated in this round: 2

total generated: 1 + 2

# Iterative Deepening DFS: Example

depth bound: 1



generated in this round: 3

total generated: 1 + 3

# Iterative Deepening DFS: Example

depth bound: 2



generated in this round: 1

total generated:  $1 + 3 + 1$

# Iterative Deepening DFS: Example

depth bound: 2



generated in this round: 2

total generated:  $1 + 3 + 2$

# Iterative Deepening DFS: Example

depth bound: 2



generated in this round: 3

total generated:  $1 + 3 + 3$



# Iterative Deepening DFS: Example

depth bound: 2

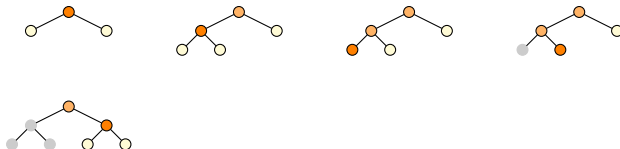


generated in this round: 4

total generated:  $1 + 3 + 4$

# Iterative Deepening DFS: Example

depth bound: 2

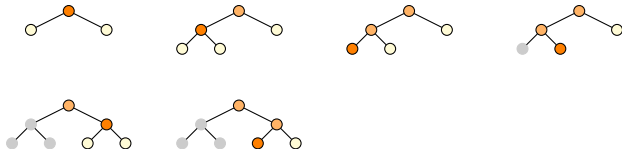


generated in this round: 5

total generated:  $1 + 3 + 5$

# Iterative Deepening DFS: Example

depth bound: 2

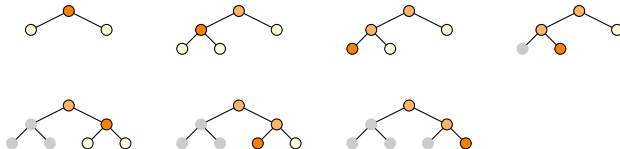


generated in this round: 6

total generated:  $1 + 3 + 6$

# Iterative Deepening DFS: Example

depth bound: 2



generated in this round: 7

total generated:  $1 + 3 + 7$

# Iterative Deepening DFS: Example

depth bound: 3



generated in this round: 1

total generated:  $1 + 3 + 7 + 1$

# Iterative Deepening DFS: Example

depth bound: 3



generated in this round: 2

total generated:  $1 + 3 + 7 + 2$

# Iterative Deepening DFS: Example

depth bound: 3



generated in this round: 3

total generated:  $1 + 3 + 7 + 3$

# Iterative Deepening DFS: Example

depth bound: 3

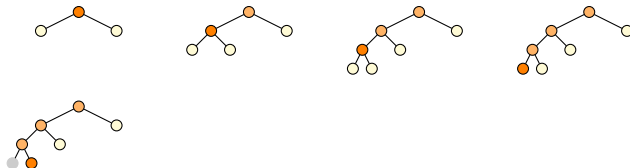


generated in this round: 4

total generated:  $1 + 3 + 7 + 4$



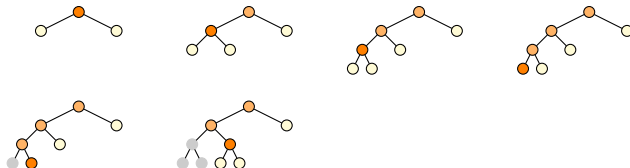
depth bound: 3



total generated:  $1 + 3 + 7 + 5$

# Iterative Deepening DFS: Example

depth bound: 3

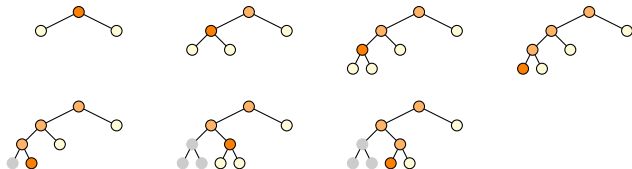


generated in this round: 6

total generated:  $1 + 3 + 7 + 6$

# Iterative Deepening DFS: Example

depth bound: 3

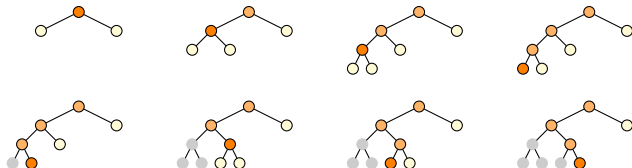


generated in this round: 7

total generated:  $1 + 3 + 7 + 7$

# Iterative Deepening DFS: Example

depth bound: 3

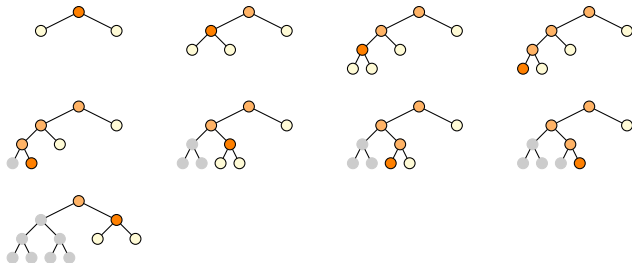


generated in this round: 8

total generated:  $1 + 3 + 7 + 8$

# Iterative Deepening DFS: Example

depth bound: 3

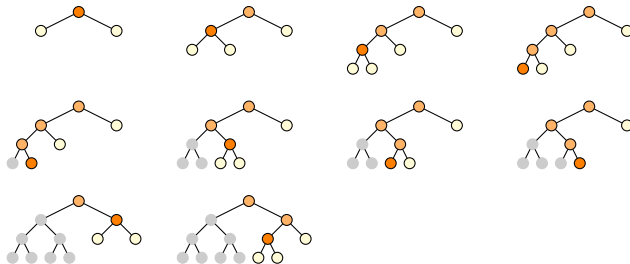


generated in this round: 9

total generated:  $1 + 3 + 7 + 9$

# Iterative Deepening DFS: Example

depth bound: 3

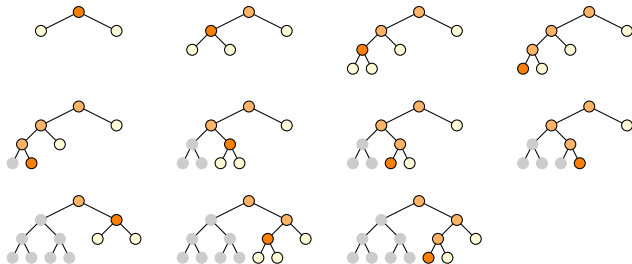


generated in this round: 10

total generated:  $1 + 3 + 7 + 10$

# Iterative Deepening DFS: Example

depth bound: 3

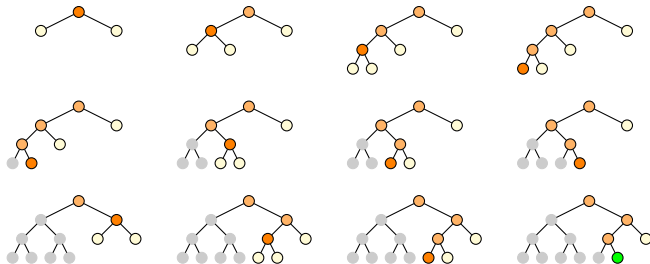


generated in this round: 11

total generated:  $1 + 3 + 7 + 11$

# Iterative Deepening DFS: Example

depth bound: 3



generated in this round: 12

total generated:  $1 + 3 + 7 + 12$

⇒ solution found!



# 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    | $1 + 10 + 100 + 1000 + 10000 + 100000$<br>$= 111111$ |
| iterative deepening DFS | $6 + 50 + 400 + 3000 + 20000 + 100000$<br>$= 123456$ |

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*

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

*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-<br>first | uniform<br>cost                           | depth-<br>first | depth-<br>bounded | iterative<br>deepening |
| complete? | yes <sup>*</sup>  | yes                                       | no              | no                | semi                   |
| optimal?  | yes <sup>**</sup> | yes                                       | no              | no                | yes <sup>**</sup>      |
| 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  
 $\ell$  depth bound  
 $c^*$  optimal solution cost  
 $\epsilon > 0$  minimal action cost

remarks:

<sup>\*</sup> for BFS-Tree: semi-complete

<sup>\*\*</sup> only with uniform action costs