Foundations of Artificial Intelligence

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

Thomas Keller and Florian Pommerening

University of Basel

March 20, 2023

State-Space Search: Overview

Chapter overview: state-space search

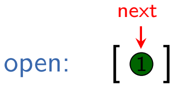
- 5.–7. Foundations
- 8.–12. Basic Algorithms
  - 8. Data Structures for Search Algorithms
  - 9. Tree Search and Graph Search
  - 10. Breadth-first Search
  - 11. Uniform Cost Search
  - 12. Depth-first Search and Iterative Deepening
- 13.–19. Heuristic Algorithms

Depth-first Search

Idea and Example

depth-first (tree) search:

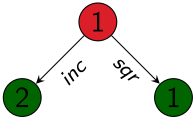
- expands nodes in opposite order of generation (LIFO)
- open list imlemented as stack
- deepest node expanded first



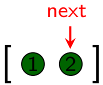
Idea and Example

depth-first (tree) search:

- expands nodes in opposite order of generation (LIFO)
- open list imlemented as stack
- deepest node expanded first



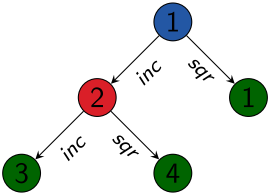
open:



Idea and Example

depth-first (tree) search:

- expands nodes in opposite order of generation (LIFO)
- open list imlemented as stack
- deepest node expanded first



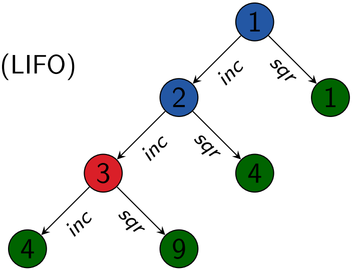
next

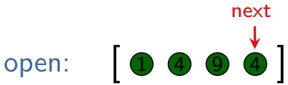
open: [1 4 3]

Idea and Example

depth-first (tree) search:

- expands nodes in opposite order of generation (LIFO)
- open list imlemented as stack
- deepest node expanded first

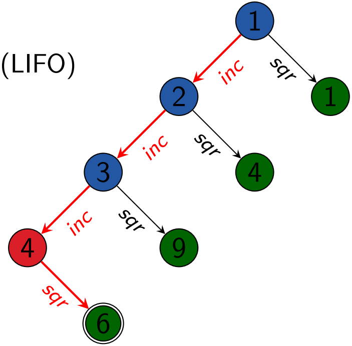




Idea and Example

depth-first (tree) search:

- expands nodes in opposite order of generation (LIFO)
- open list imlemented as stack
- deepest node expanded first
- here: goal test on generation
  (but: depends on implementation)



next

open: [1 4 9]

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 $\langle a, s' \rangle \in$ 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 $\langle a, s' \rangle \in$ succ(n.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

$\rightsquigarrow$ CPU stack as implicit open list

$\rightsquigarrow$ no search node data structure needed

Depth-first Search (Recursive Version)

function depth_first_search(s)

if is_goal(s):
    return $\langle\rangle$
for each $\langle a, s'\rangle \in$ succ(s):
    solution := depth_first_search(s')
    if *solution* $\neq$ **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)

$\rightsquigarrow$ 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 and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

Idea and Example

depth-limited search:

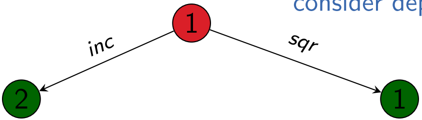
- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

consider depth limit $\ell = 2$

Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$



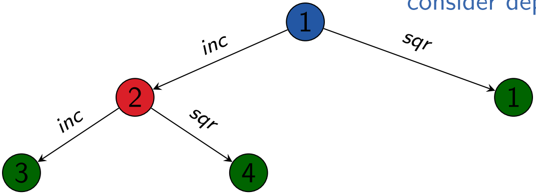
consider depth limit $\ell = 2$

Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

consider depth limit $\ell = 2$

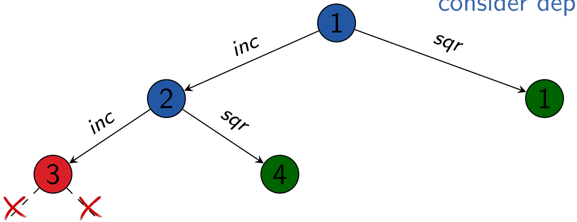


Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

consider depth limit $\ell = 2$

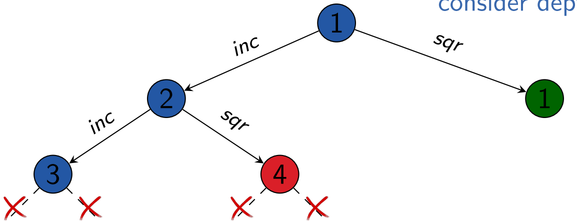


Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

consider depth limit $\ell = 2$

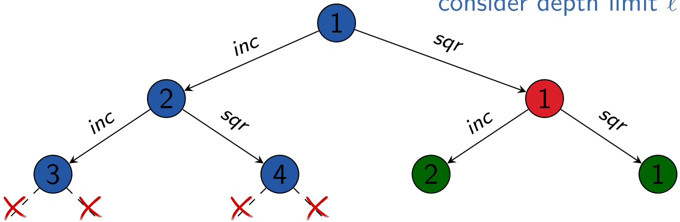


Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

consider depth limit $\ell = 2$

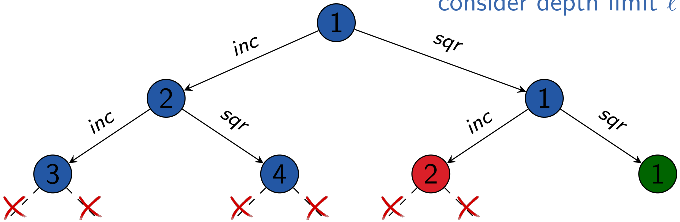


Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

consider depth limit $\ell = 2$

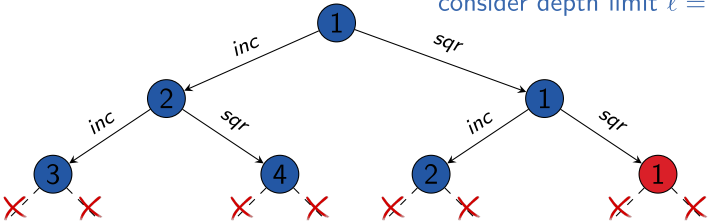


Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$

consider depth limit $\ell = 2$

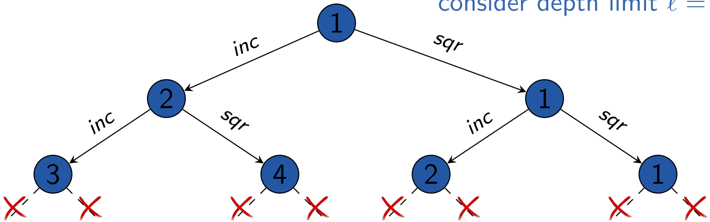


Idea and Example

depth-limited search:

- parametrized with depth limit ℓ
- variant of depth-first search, i.e.,
  - expands nodes in opposite order of generation (LIFO)
  - tree search
- prunes (does not expand) search nodes at depth $d \geq \ell$
- not very useful on its own, but important ingredient of more useful algorithms

consider depth limit $\ell = 2$



Depth-bounded Search: Pseudo-Code

function depth_bounded_search(s, *depth_bound*):

if is_goal(s):
    return $\langle\rangle$
if $\mathit{depth\_bound} > 0$:
    for each $\langle a, s' \rangle \in$ succ(s):
        $\mathit{solution} :=$ depth_bounded_search(s', $\mathit{depth\_bound} - 1$)
        if $\mathit{solution} \neq$ **none**:
            solution.push_front(a)
            return *solution*
return none

Iterative Deepening Depth-first Search

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

- performs a sequence of depth-limited searches
- increases depth limit ℓ in each iteration
- sounds wasteful (each iteration repeats all the useful work of all previous iterations)
- in fact overhead acceptable (analysis follows)

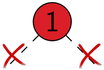
Iterative Deepening DFS

for $\mathit{depth\_bound} \in \{0, 1, 2, \dots\}$:
    $\mathit{solution}$:= depth_bounded_search(init(), $\mathit{depth\_bound}$)
    if $\mathit{solution} \neq$ **none**:
        return $\mathit{solution}$

Example

depth limit: 0

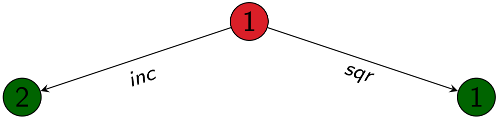
generated nodes: 1



(using a non-recursive version of depth-limited search)

Example

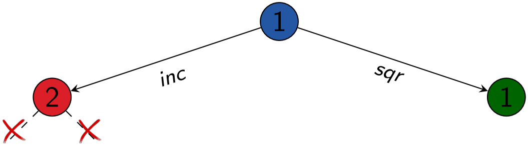
depth limit: 1 generated nodes: 1+3



(using a non-recursive version of depth-limited search)

Example

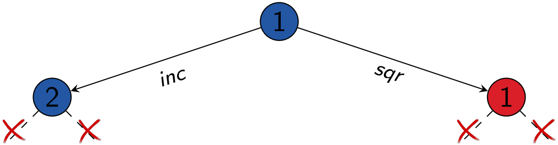
depth limit: 1 generated nodes: 1+3



(using a non-recursive version of depth-limited search)

Example

depth limit: 1 generated nodes: 1+3

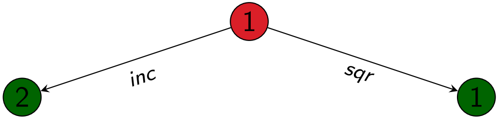


(using a non-recursive version of depth-limited search)

Example

depth limit: 2

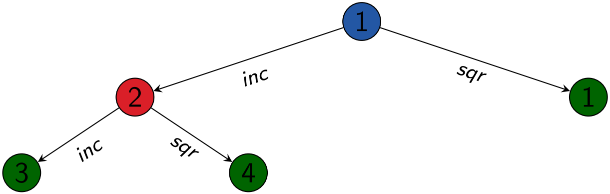
generated nodes: 1+3+3



(using a non-recursive version of depth-limited search)

Example

depth limit: 2 generated nodes: 1+3+5

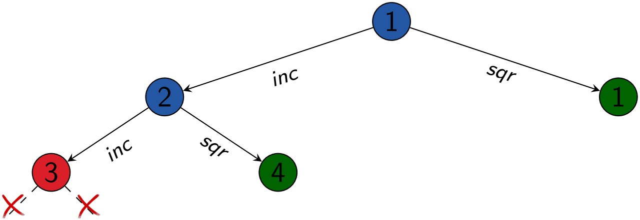


(using a non-recursive version of depth-limited search)

Example

depth limit: 2

generated nodes: 1+3+5

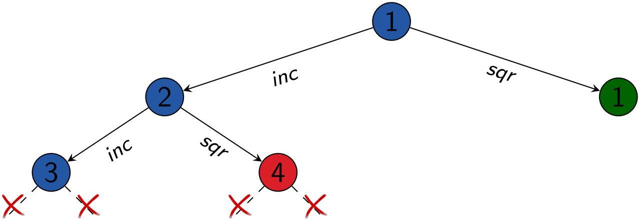


(using a non-recursive version of depth-limited search)

Example

depth limit: 2

generated nodes: 1+3+5

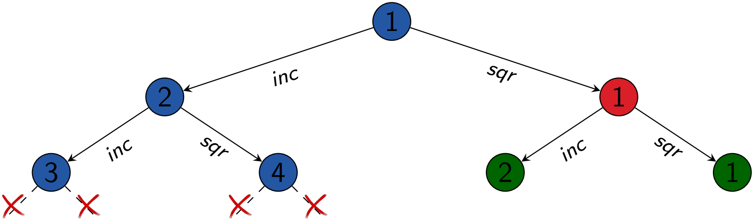


(using a non-recursive version of depth-limited search)

Example

depth limit: 2

generated nodes: 1+3+7

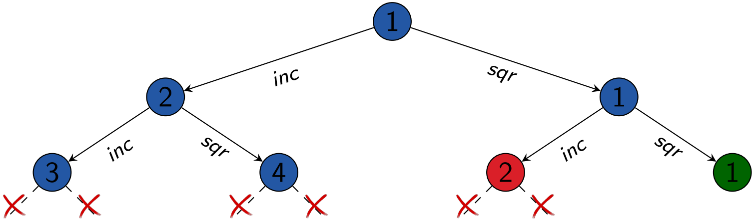


(using a non-recursive version of depth-limited search)

Example

depth limit: 2

generated nodes: 1+3+7

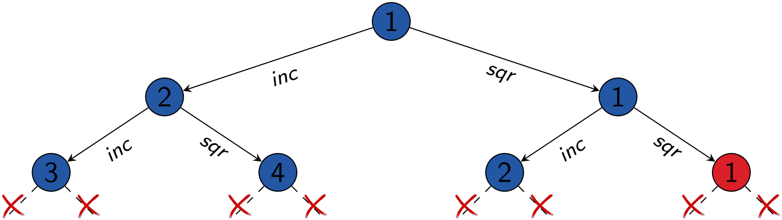


(using a non-recursive version of depth-limited search)

Example

depth limit: 2

generated nodes: 1+3+7

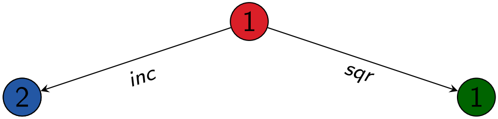


(using a non-recursive version of depth-limited search)

Example

depth limit: 3

generated nodes: 1+3+7+3

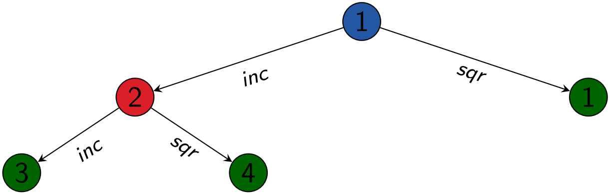


(using a non-recursive version of depth-limited search)

Example

depth limit: 3

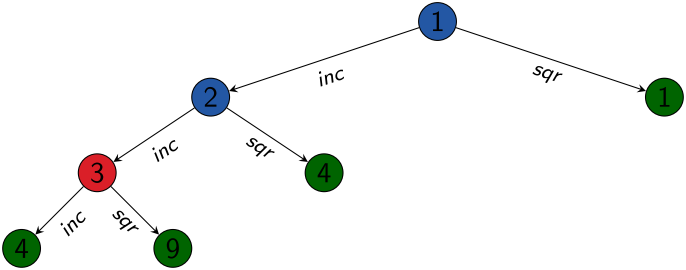
generated nodes: 1+3+7+5



(using a non-recursive version of depth-limited search)

Example

depth limit: 3 generated nodes: 1+3+7+7

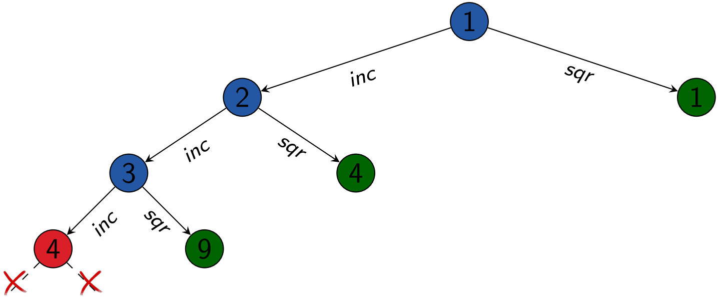


(using a non-recursive version of depth-limited search)

Example

depth limit: 3

generated nodes: 1+3+7+7

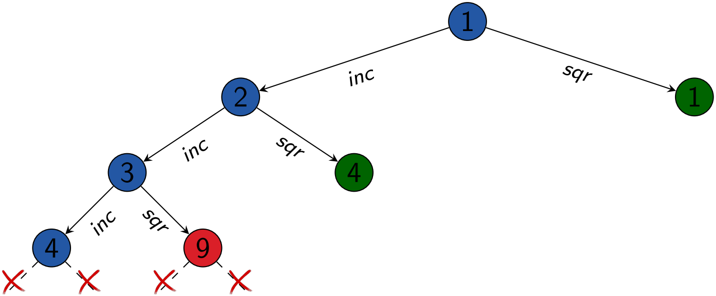


(using a non-recursive version of depth-limited search)

Example

depth limit: 3

generated nodes: 1+3+7+7

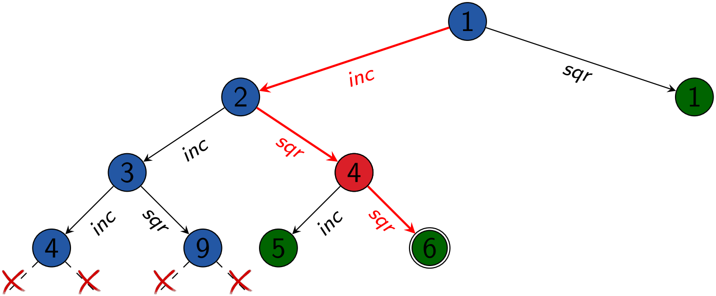


(using a non-recursive version of depth-limited search)

Example

depth limit: 3

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



(using a non-recursive version of depth-limited search)

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
  ($\rightsquigarrow$ next slide)

Iterative Deepening DFS: Complexity Example

time complexity (generated nodes):

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

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 complextive 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

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

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

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