Foundations of Artificial Intelligence

10. State-Space Search: Breadth-first Search

Malte Helmert and Thomas Keller

University of Basel

March 16, 2020

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

Blind Search

Blind Search

In Chapters 10–12 we consider blind search algorithms:

Blind Search Algorithms

Blind search algorithms use no information
about state spaces apart from the black box interface.

They are also called uninformed search algorithms.

contrast: heuristic search algorithms (Chapters 13–19)

Blind Search Algorithms: Examples

examples of blind search algorithms:

- breadth-first search
- uniform cost search
- depth-first search
- depth-limited search
- iterative deepening search

Blind Search Algorithms: Examples

examples of blind search algorithms:

- breadth-first search (⇝ this chapter)
- uniform cost search
- depth-first search
- depth-limited search
- iterative deepening search

Blind Search Algorithms: Examples

examples of blind search algorithms:

- breadth-first search (⇝ this chapter)
- uniform cost search (⇝ Chapter 11)
- depth-first search (⇝ Chapter 12)
- depth-limited search (⇝ Chapter 12)
- iterative deepening search (⇝ Chapter 12)

Breadth-first Search: Introduction

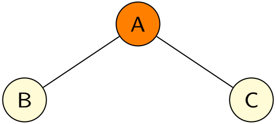
Breadth-first Search

Breadth-first search expands nodes in order of generation (FIFO).
⇝ e.g., open list as linked list or deque

open: A

Breadth-first Search

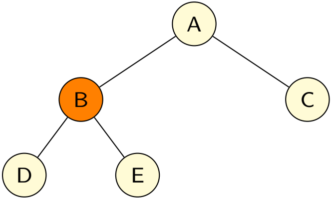
Breadth-first search expands nodes in order of generation (FIFO).
⇝ e.g., open list as linked list or deque



open: B, C

Breadth-first Search

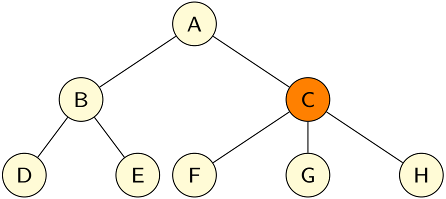
Breadth-first search expands nodes in order of generation (FIFO).
$\rightsquigarrow$ e.g., open list as linked list or deque



open: C, D, E

Breadth-first Search

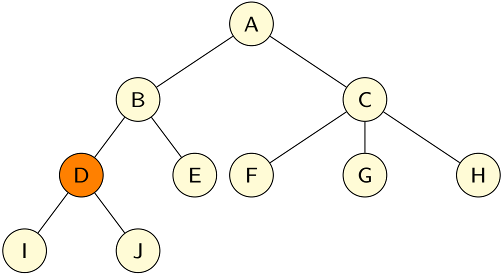
Breadth-first search expands nodes in order of generation (FIFO).
⇝ e.g., open list as linked list or deque



open: D, E, F, G, H

Breadth-first Search

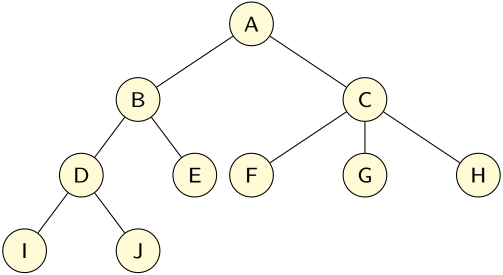
Breadth-first search expands nodes in order of generation (FIFO).
$\rightsquigarrow$ e.g., open list as linked list or deque



open: E, F, G, H, I, J

Breadth-first Search

Breadth-first search expands nodes in order of generation (FIFO).
$\rightsquigarrow$ e.g., open list as linked list or deque



- searches state space layer by layer
- always finds shallowest goal state first

Breadth-first Search: Tree Search or Graph Search?

Breadth-first search can be performed

- without duplicate elimination (as a tree search)
  ⇝ BFS-Tree
- or with duplicate elimination (as a graph search)
  ⇝ BFS-Graph

(BFS = breadth-first search).

⇝ We consider both variants.

German: Breitensuche

BFS-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
```

BFS-Tree (1st Attempt)

breadth-first search without duplicate elimination (1st attempt):

BFS-Tree (1st Attempt)

```
$open$ := new Deque
$open$.push_back(make_root_node())
while not $open$.is_empty():
    $n$ := $open$.pop_front()
    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
```

BFS-Tree (1st Attempt)

breadth-first search without duplicate elimination (1st attempt):

BFS-Tree (1st Attempt)

```
open := new deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    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
```

BFS-Tree (1st Attempt): Discussion

This is almost a usable algorithm, but it wastes some effort:

- In a breadth-first search, the first generated goal node is always the first expanded goal node. (Why?)
- Hence it is more efficient to already perform the goal test upon generating a node (rather than upon expanding it).

$\rightsquigarrow$ How much effort does this save?

BFS-Tree (2nd Attempt)

breadth-first search without duplicate elimination (2nd attempt):

BFS-Tree (2nd Attempt)

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

BFS-Tree (2nd Attempt)

breadth-first search without duplicate elimination (2nd attempt):

BFS-Tree (2nd Attempt)

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

BFS-Tree (2nd Attempt): Discussion

Where is the bug?

BFS-Tree (Final Version)

breadth-first search without duplicate elimination (final version):

BFS-Tree

```
if is_goal(init()):
    return ⟨⟩
open := new Deque
open.push_back(make_root_node())
while not open.is_empty():
    n := open.pop_front()
    for each ⟨a, s′⟩ ∈ succ(n.state):
        n′ := make_node(n, a, s′)
        if is_goal(s′):
            return extract_path(n′)
        open.push_back(n′)
return unsolvable
```

BFS-Tree (Final Version)

breadth-first search without duplicate elimination (final version):

BFS-Tree

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

BFS-Graph

Reminder: Generic Graph Search Algorithm

reminder from Chapter 9:

Generic Graph Search

```
$open$ := new OpenList
$open$.insert(make_root_node())
$closed$ := new ClosedList
while not $open$.is_empty():
    $n$ := $open$.pop()
    if $closed$.lookup($n$.state) = none:
        $closed$.insert($n$)
        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
```

Adapting Generic Graph Search to Breadth-First Search

Adapting the generic algorithm to breadth-first search:

- similar adaptations to BFS-Tree (deque as open list, early goal test)
- as closed list does not need to manage node information, a set data structure suffices
- for the same reasons why early goal tests are a good idea, we should perform duplicate tests against the closed list and updates of the closed lists as early as possible

BFS-Graph (Breadth-First Search with Duplicate Elim.)

BFS-Graph

```
if is_goal(init()):
    return $\langle\rangle$
$open$ := new Deque
$open$.push_back(make_root_node())
$closed$ := new HashSet
$closed$.insert(init())
while not $open$.is_empty():
    $n$ := $open$.pop_front()
    for each $\langle a, s'\rangle \in$ succ($n$.state):
        $n'$ := make_node($n, a, s'$)
        if is_goal($s'$):
            return extract_path($n'$)
        if $s' \notin closed$:
            $closed$.insert($s'$)
            $open$.push_back($n'$)
return unsolvable
```

Properties of Breadth-first Search

Properties of Breadth-first Search

Properties of Breadth-first Search:

- BFS-Tree is semi-complete, but not complete. (Why?)
- BFS-Graph is complete. (Why?)
- BFS (both variants) is optimal
  if all actions have the same cost (Why?),
  but not in general (Why not?).
- complexity: next slides

Breadth-first Search: Complexity

The following result applies to both BFS variants:

Theorem (time complexity of breadth-first search)

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 breadth-first search is

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

Reminder: we measure time complexity in generated nodes.

It follows that the space complexity of both BFS variants also is $O(b^d)$ (if $b \geq 2$). (Why?)

Breadth-first Search: Example of Complexity

example: $b = 10$; 100 000 nodes/second; 32 bytes/node

d	nodes	time	memory
3	1 111	0.01 s	35 KiB
5	111 111	1 s	3.4 MiB
7	10^7	2 min	339 MiB
9	10^9	3 h	33 GiB
11	10^{11}	13 days	3.2 TiB
13	10^{13}	3.5 years	323 TiB
15	10^{15}	350 years	32 PiB

BFS-Tree or BFS-Graph?

What is better, BFS-Tree or BFS-Graph?

BFS-Tree or BFS-Graph?

What is better, BFS-Tree or BFS-Graph?

advantages of BFS-Graph:

- complete
- much (!) more efficient if there are many duplicates

BFS-Tree or BFS-Graph?

What is better, BFS-Tree or BFS-Graph?

advantages of BFS-Graph:

- complete
- much (!) more efficient if there are many duplicates

advantages of BFS-Tree:

- simpler
- less overhead (time/space) if there are few duplicates

BFS-Tree or BFS-Graph?

What is better, BFS-Tree or BFS-Graph?

advantages of BFS-Graph:

- complete
- much (!) more efficient if there are many duplicates

advantages of BFS-Tree:

- simpler
- less overhead (time/space) if there are few duplicates

Conclusion

BFS-Graph is usually preferable, unless we know that there is a negligible number of duplicates in the given state space.

Summary

Summary

- blind search algorithm: use no information except black box interface of state space
- breadth-first search: expand nodes in order of generation
  - search state space layer by layer
  - can be tree search or graph search
  - complexity $O(b^d)$ with branching factor b, minimal solution length d (if $b \geq 2$)
  - complete as a graph search; semi-complete as a tree search
  - optimal with uniform action costs