

Foundations of Artificial Intelligence

B6. State-Space Search: Breadth-first Search

Malte Helmert

University of Basel

March 5, 2025

Foundations of Artificial Intelligence

March 5, 2025 — B6. State-Space Search: Breadth-first Search

B6.1 Blind Search

B6.2 Breadth-first Search: Introduction

B6.3 BFS-Tree

B6.4 BFS-Graph

B6.5 Properties of Breadth-first Search

B6.6 Summary

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

B6.1 Blind Search

Blind Search

In Chapters B6–B8 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 B9–B15)

Blind Search Algorithms: Examples

examples of blind search algorithms:

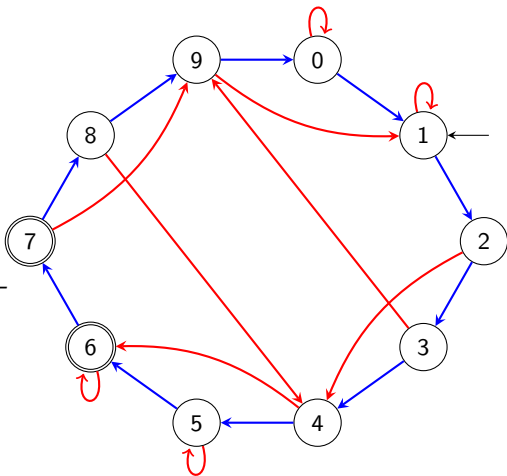
- ▶ **breadth-first search** (↪ this chapter)
- ▶ uniform cost search (↪ Chapter B7)
- ▶ depth-first search (↪ Chapter B8)
- ▶ depth-limited search (↪ Chapter B8)
- ▶ iterative deepening search (↪ Chapter B8)

B6.2 Breadth-first Search: Introduction

Running Example: Reminder

bounded inc-and-square:

- ▶ $S = \{0, 1, \dots, 9\}$
- ▶ $A = \{inc, sqr\}$
- ▶ $cost(inc) = cost(sqr) = 1$
- ▶ T s.t. for $i = 0, \dots, 9$:
 - ▶ $\langle i, inc, (i + 1) \bmod 10 \rangle \in T$
 - ▶ $\langle i, sqr, i^2 \bmod 10 \rangle \in T$
- ▶ $s_1 = 1$
- ▶ $S_G = \{6, 7\}$



Idea


breadth-first search:

- ▶ expand nodes in order of generation (FIFO)
 - ↪ open list is linked list or deque
- ▶ we start with an example using graph search

German: Breitensuche

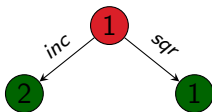
Example: Generic Graph Search with FIFO Expansion



open: []
closed: { }

The diagram shows the state of a search algorithm. The 'open' list contains a single element, a green circle with the number 1, which is the root node. A red arrow labeled 'next' points to this element. The 'closed' list is an empty set, represented by curly braces.

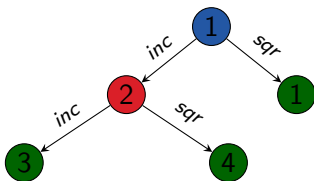
Example: Generic Graph Search with FIFO Expansion



open: $\left[\overset{\text{next}}{\downarrow} \begin{matrix} \bullet \\ \bullet \end{matrix} \right]$

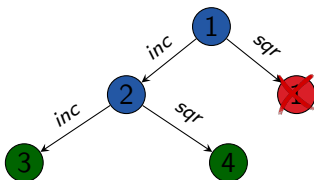
closed: $\{1\}$

Example: Generic Graph Search with FIFO Expansion



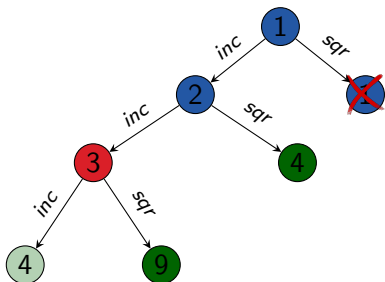
open: $\overset{\text{next}}{\downarrow} [1, 3, 4]$
 closed: $\{1, 2\}$

Example: Generic Graph Search with FIFO Expansion



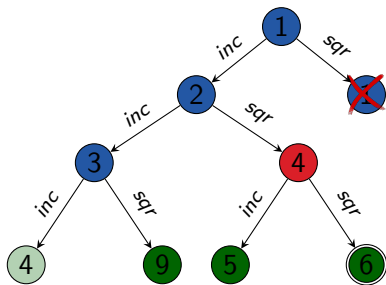
open: $\left[\overset{\text{next}}{\downarrow} \begin{matrix} \text{3} & \text{4} \end{matrix} \right]$
 closed: $\{1, 2\}$

Example: Generic Graph Search with FIFO Expansion



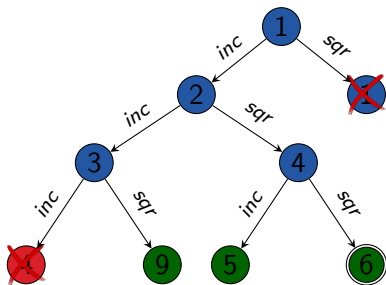
next
 ↓
 open: [4 4 9]
 closed: { 1, 2, 3 }

Example: Generic Graph Search with FIFO Expansion



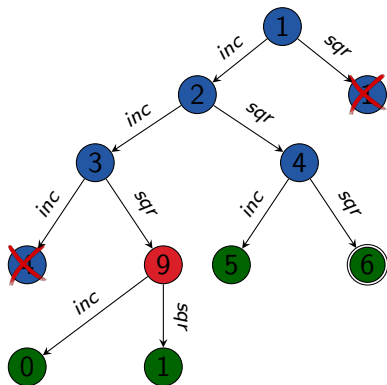
next
 ↓
 open: [4 9 5 6]
 closed: { 1, 2, 3, 4 }

Example: Generic Graph Search with FIFO Expansion



next
 ↓
 open: [7 5 6]
 closed: { 1, 2, 3, 4 }

Example: Generic Graph Search with FIFO Expansion

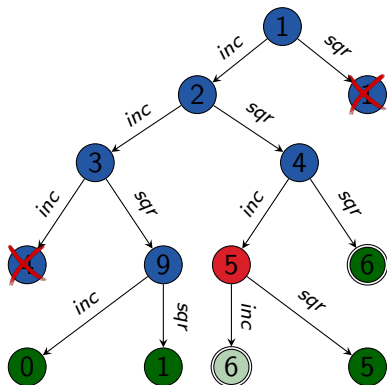


next

open: [5 6 0 1]

closed: { 1, 2, 3, 4, 9 }

Example: Generic Graph Search with FIFO Expansion

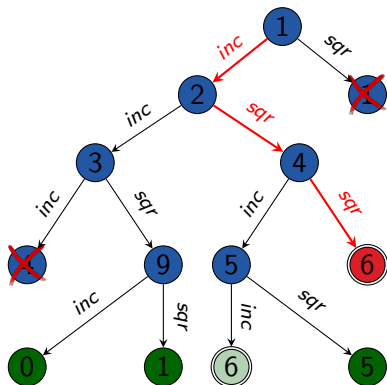


next

open: [6 0 1 6 5]

closed: { 1, 2, 3, 4, 5, 9 }

Example: Generic Graph Search with FIFO Expansion



next

open: [0 1 6 5]

closed: { 1, 2, 3, 4, 5, 6, 9 }

Observations from Example

breadth-first search behaviour:

- ▶ state space is searched **layer by layer**
- ↪ **shallowest** goal node is always found 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.

B6.3 BFS-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  $\langle a, s' \rangle \in \text{succ}(n.\text{state})$ :
        n' := make_node(n, a, s')
        open.insert(n')
return unsolvable
```

BFS-Tree (1st Attempt)

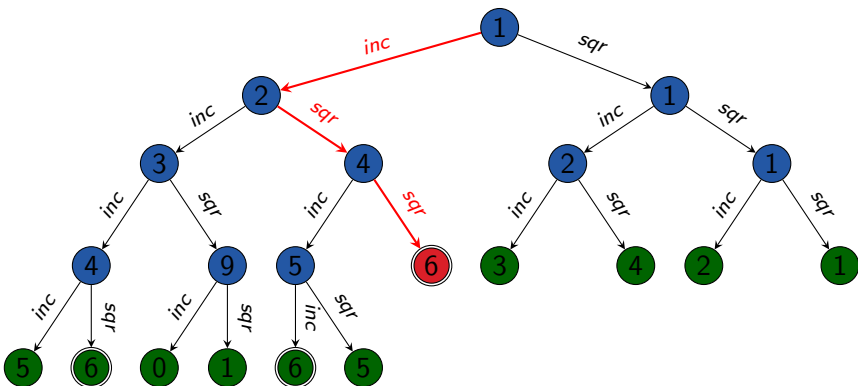
breadth-first search without duplicate elimination (1st attempt):

BFS-Tree (1st Attempt)

```

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

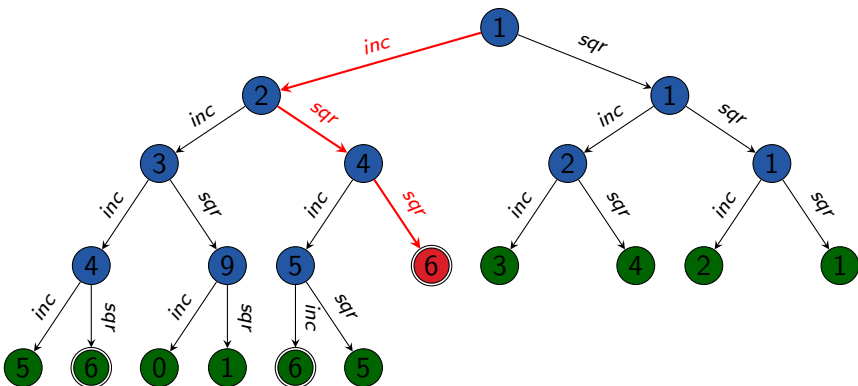

Running Example: BFS-Tree (1st Attempt)



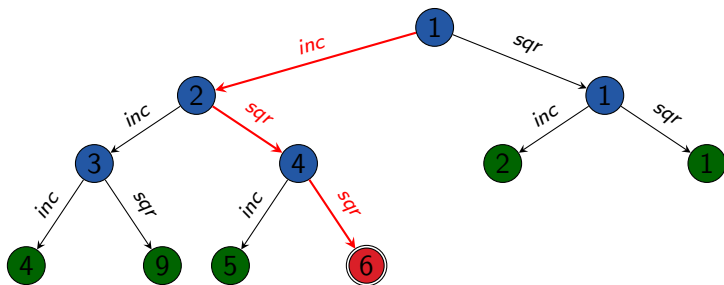
Opportunities for Improvement

- ▶ In a BFS, the first generated goal node is always the first expanded goal node. (Why?)
- ↪ It is more efficient to perform the goal test upon **generating** a node (rather than upon **expanding** it).
- ↪ How much effort does this save?

BFS-Tree without Early Goal Tests



BFS-Tree with Early Goal Tests



BFS-Tree (2nd Attempt)

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

BFS-Tree (2nd Attempt)

```

open := new Queue
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  $\langle a, s' \rangle \in \text{succ}(n.state)$ :
        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  $\langle \rangle$   
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
```

B6.4 BFS-Graph

Reminder: Generic Graph Search Algorithm

reminder from Chapter B5:

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 tests**)
- ▶ 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  $\langle a, s' \rangle \in \text{succ}(n.\text{state})$ :
        n' := make_node(n, a, s')
        if is_goal(s'):
            return extract_path(n')
        if  $s' \notin \text{closed}$ :
            closed.insert(s')
            open.push_back(n')
```

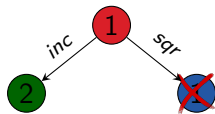
return unsolvable

BFS-Graph: Example



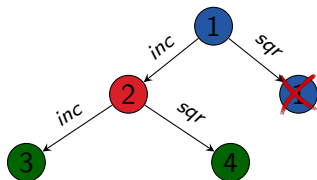
open: $\overset{\text{next}}{\downarrow} [\textcircled{1}]$
closed: $\{1\}$

BFS-Graph: Example



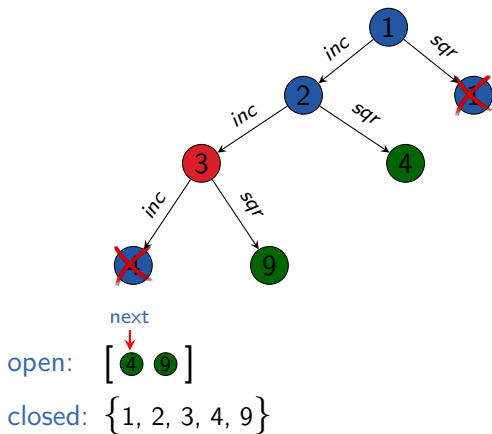
open: $\left[\overset{\text{next}}{\downarrow} \bullet \right]$
closed: $\{1, 2\}$

BFS-Graph: Example

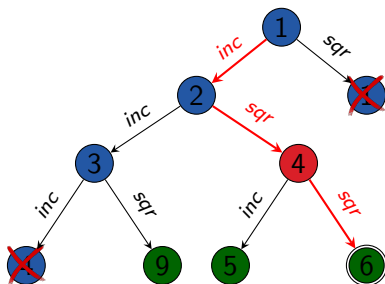


next
 ↓
 open: [3 4]
 closed: { 1, 2, 3, 4 }

BFS-Graph: Example



BFS-Graph: Example



next
 ↓
 open: [9 5]
 closed: { 1, 2, 3, 4, 5, 6, 9 }

B6.5 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*

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

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: $b = 13$; 100 000 nodes/second; 32 bytes/node



Rubik's cube:

- ▶ branching factor: ≈ 13
- ▶ typical solution length: 18

d	nodes	time	memory
4	30 940	0.3 s	966 KiB
6	$5.2 \cdot 10^6$	52 s	159 MiB
8	$8.8 \cdot 10^8$	147 min	26 GiB
10	10^{11}	17 days	4.3 TiB
12	10^{13}	8 years	734 TiB
14	10^{15}	1 352 years	121 PiB
16	10^{17}	$2.2 \cdot 10^5$ years	20 EiB
18	10^{20}	$38 \cdot 10^6$ years	3.3 ZiB

BFS-Tree or BFS-Graph?

Which 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.

B6.6 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**