

Foundations of Artificial Intelligence

8. State-Space Search: Data Structures for Search Algorithms

Malte Helmert

University of Basel

March 14, 2022

Foundations of Artificial Intelligence

March 14, 2022 — 8. State-Space Search: Data Structures for Search Algorithms

8.1 Introduction

8.2 Search Nodes

8.3 Open Lists

8.4 Closed Lists

8.5 Summary

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

8.1 Introduction

Search Algorithms

- ▶ We now move to **search algorithms**.
- ▶ As everywhere in computer science, suitable **data structures** are a key to good performance.
 - ↪ **common** operations must be **fast**
- ▶ Well-implemented search algorithms process up to $\sim 30,000,000$ states/second on a single CPU core.
 - ↪ bonus materials (Burns et al. paper)

this chapter: some **fundamental data structures** for search

Preview: Search Algorithms

- ▶ **next chapter**: we introduce search algorithms
- ▶ **now**: short **preview** to motivate data structures for search

Example: Search Algorithm

- ▶ Starting with **initial state**,
- ▶ repeatedly **expand** a state by **generating** its **successors**.
- ▶ Stop when a **goal state** is expanded
- ▶ or **all reachable states** have been considered.

German: expandieren, erzeugen

Example: Search Algorithm

- ▶ Starting with **initial state**,
- ▶ repeatedly **expand** a state by **generating** its **successors**.
- ▶ Stop when a **goal state** is expanded
- ▶ or **all reachable states** have been considered.

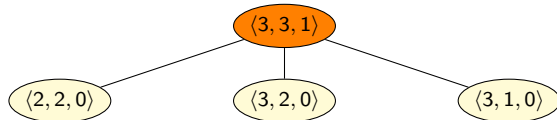
German: expandieren, erzeugen

(3, 3, 1)

Example: Search Algorithm

- ▶ Starting with **initial state**,
- ▶ repeatedly **expand** a state by **generating** its **successors**.
- ▶ Stop when a **goal state** is expanded
- ▶ or **all reachable states** have been considered.

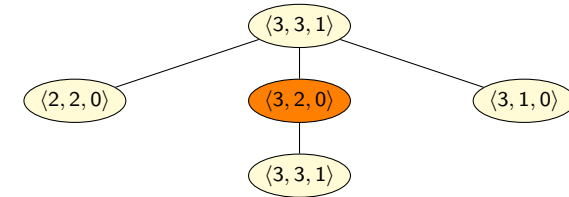
German: expandieren, erzeugen



Example: Search Algorithm

- ▶ Starting with **initial state**,
- ▶ repeatedly **expand** a state by **generating** its **successors**.
- ▶ Stop when a **goal state** is expanded
- ▶ or **all reachable states** have been considered.

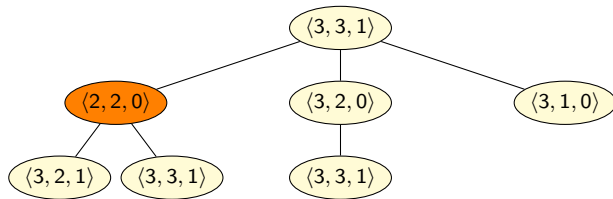
German: expandieren, erzeugen



Example: Search Algorithm

- ▶ Starting with **initial state**,
- ▶ repeatedly **expand** a state by **generating** its **successors**.
- ▶ Stop when a **goal state** is expanded
- ▶ or **all reachable states** have been considered.

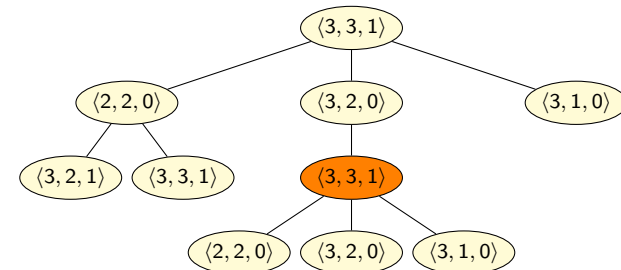
German: expandieren, erzeugen



Example: Search Algorithm

- ▶ Starting with **initial state**,
- ▶ repeatedly **expand** a state by **generating** its **successors**.
- ▶ Stop when a **goal state** is expanded
- ▶ or **all reachable states** have been considered.

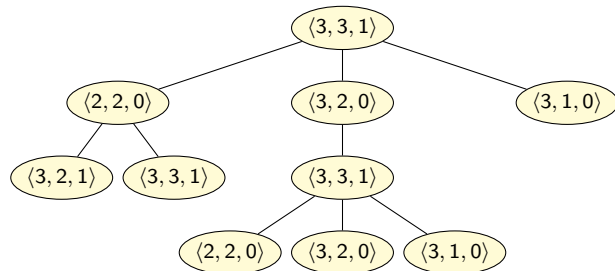
German: expandieren, erzeugen



Example: Search Algorithm

- ▶ Starting with **initial state**,
- ▶ repeatedly **expand** a state by **generating** its **successors**.
- ▶ Stop when a **goal state** is expanded
- ▶ or **all reachable states** have been considered.

German: expandieren, erzeugen



... and so on (expansion order depends on search algorithm used)

Fundamental Data Structures for Search

We consider three abstract data structures for search:

- ▶ **search node**: stores a state that has been reached, how it was reached, and at which cost
 - ↪ nodes of the example search tree
- ▶ **open list**: efficiently organizes leaves of search tree
 - ↪ set of leaves of example search tree
- ▶ **closed list**: remembers expanded states to avoid duplicated expansions of the same state
 - ↪ inner nodes of a search tree

German: Suchknoten, Open-Liste, Closed-Liste

Not all algorithms use all three data structures, and they are sometimes implicit (e.g., in the CPU stack)

8.2 Search Nodes

Search Nodes

Search Node

A **search node** (**node** for short) stores a state that has been reached, how it was reached, and at which cost.

Collectively they form the so-called **search tree** (**Suchbaum**).

Attributes of a Search Node

Attributes of a Search Node n

- $n.state$ state associated with this node
- $n.parent$ search node that generated this node (**none** for the root node)
- $n.action$ action leading from $n.parent$ to n (**none** for the root node)
- $n.path_cost$ cost of path from initial state to $n.state$ that result from following the parent references (traditionally denoted by $g(n)$)

... and sometimes additional attributes (e.g., **depth** in tree)

Search Nodes: Java

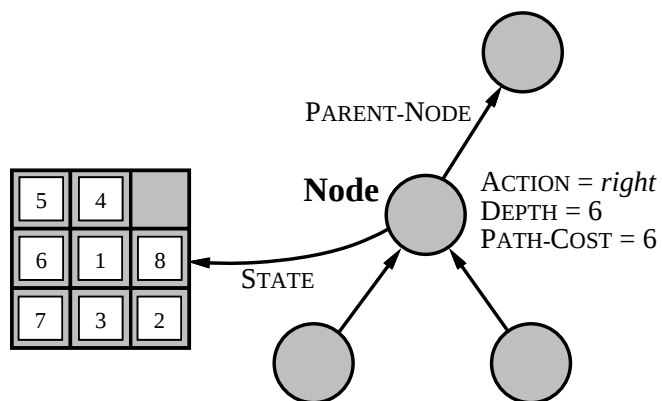
Search Nodes (Java Syntax)

```
public interface State {
}

public interface Action {
}

public class SearchNode {
    State state;
    SearchNode parent;
    Action action;
    int pathCost;
}
```

Node in a Search Tree



Implementing Search Nodes

- ▶ **reasonable implementation** of search nodes is easy
- ▶ **advanced aspects:**
 - ▶ Do we need explicit nodes at all?
 - ▶ Can we use lazy evaluation?
 - ▶ Should we manually manage memory?
 - ▶ Can we compress information?

Operations on Search Nodes: make_root_node

Generate root node of a search tree:

```
function make_root_node()  
  node := new SearchNode  
  node.state := init()  
  node.parent := none  
  node.action := none  
  node.path_cost := 0  
  return node
```

Operations on Search Nodes: make_node

Generate child node of a search node:

```
function make_node(parent, action, state)  
  node := new SearchNode  
  node.state := state  
  node.parent := parent  
  node.action := action  
  node.path_cost := parent.path_cost + cost(action)  
  return node
```

Operations on Search Nodes: extract_path

Extract the path to a search node:

```
function extract_path(node)  
  path :=  $\langle \rangle$   
  while node.parent  $\neq$  none:  
    path.append(node.action)  
    node := node.parent  
  path.reverse()  
  return path
```

8.3 Open Lists

Open Lists

Open List

The **open list** (also: **frontier**) organizes the leaves of a search tree.

It must support two operations efficiently:

- ▶ determine and remove the next node to expand
- ▶ insert a new node that is a candidate node for expansion

Remark: despite the name, it is usually a very bad idea to implement open lists as simple **lists**.

Open Lists: Modify Entries

- ▶ Some implementations support **modifying** an open list entry when a shorter path to the corresponding state is found.
- ▶ This complicates the implementation.
- ↪ We do not consider such modifications and instead use **delayed duplicate elimination** (↪ later)

Interface of Open Lists

Methods of an Open List *open*

open.is_empty() test if the open list is empty

open.pop() removes and returns the next node to expand

open.insert(*n*) inserts node *n* into the open list

- ▶ Different search algorithm use different strategies for the decision which node to return in ***open.pop***.
- ▶ The choice of a suitable data structure depends on this strategy (e.g., stack, deque, min-heap).

8.4 Closed Lists

Closed Lists

Closed List

The **closed list** remembers expanded states to avoid duplicated expansions of the same state.

It must support two operations efficiently:

- ▶ insert a node whose state is not yet in the closed list
- ▶ test if a node with a given state is in the closed list; if yes, return it

Remark: despite the name, it is usually a very bad idea to implement closed lists as simple **lists**. (Why?)

Interface and Implementation of Closed Lists

Methods of a Closed List *closed*

closed.insert(*n*) insert node *n* into *closed*;
if a node with this state already exists in *closed*,
replace it

closed.lookup(*s*) test if a node with state *s* exists in the closed list;
if yes, return it; otherwise, return **none**

- ▶ Hash tables with states as keys can serve as efficient implementations of closed lists.

8.5 Summary

Summary

- ▶ **search node:**
represents states reached during search and associated information
- ▶ **node expansion:**
generate successor nodes of a node by applying all actions applicable in the state belonging to the node
- ▶ **open list** or **frontier:**
set of nodes that are currently candidates for expansion
- ▶ **closed list:**
set of already expanded nodes (and their states)