

# Foundations of Artificial Intelligence

## B7. State-Space Search: Uniform Cost Search

Malte Helmert

University of Basel

March 13, 2024

# State-Space Search: Overview

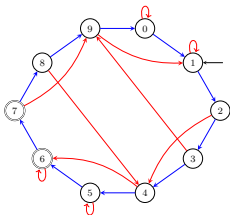
## Chapter overview: state-space search

- 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

# Introduction

# Uniform Cost Search

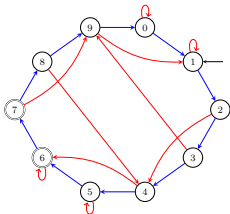
- breadth-first search optimal if all action costs equal
- otherwise no optimality guarantee  $\rightsquigarrow$  [example](#):



- consider bounded inc-and-square problem with  $cost(inc) = 1$ ,  $cost(sqr) = 3$
- solution of breadth-first search still  $\langle inc, sqr, sqr \rangle$  (cost: 7)
- **but:**  $\langle inc, inc, inc, inc, inc \rangle$  (cost: 5) is cheaper!

# Uniform Cost Search

- breadth-first search optimal if all action costs equal
- otherwise no optimality guarantee  $\rightsquigarrow$  **example:**



- consider bounded inc-and-square problem with  $cost(inc) = 1$ ,  $cost(sq) = 3$
- solution of breadth-first search still  $\langle inc, sq, sq \rangle$  (cost: 7)
- **but:**  $\langle inc, inc, inc, inc, inc \rangle$  (cost: 5) is cheaper!

remedy: **uniform cost search**

- always expand a node with **minimal path cost** ( $n.path\_cost$  a.k.a.  $g(n)$ )
- **implementation:** **priority queue** (min-heap) for open list

# Algorithm

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

# Uniform Cost Search

## Uniform Cost Search

```
open := new MinHeap ordered by g
open.insert(make_root_node())
closed := new HashSet
while not open.is_empty():
    n := open.pop_min()
    if n.state  $\notin$  closed:
        closed.insert(n.state)
        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
```



# Uniform Cost Search: Discussion

## Adapting generic graph search to uniform cost search:

- here, early goal tests/early updates of the closed list **not** a good idea. (Why not?)
- as in BFS-Graph, a **set** is sufficient for the closed list
- a tree search variant is possible, but rare:  
has the same disadvantages as BFS-Tree  
and in general **not even semi-complete** (Why not?)

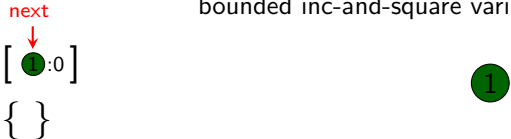
## Remarks:

- identical to **Dijkstra's algorithm** for shortest paths
- for both: variants with/without delayed duplicate elimination

# Example

open: [ 1:0 ]  
closed: { }

next  
↓



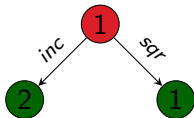
bounded inc-and-square variant:  $\text{cost}(sqr) = 3$

# Example

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

closed: { 1 }

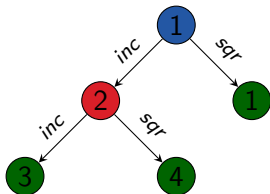
bounded inc-and-square variant:  $\text{cost}(sqr) = 3$



# Example

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

bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$

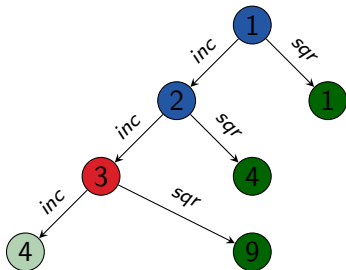


# Example

open: [ **1**:3 ④:3 ④:4 ⑨:5 ]

closed: {1, 2, 3}

bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$

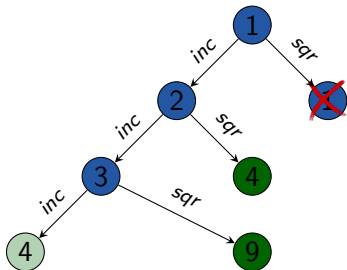


# Example

open: [  $\overset{\text{next}}{\downarrow}$  ④:3 ④:4 ⑨:5 ]

closed: {1, 2, 3}

bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$

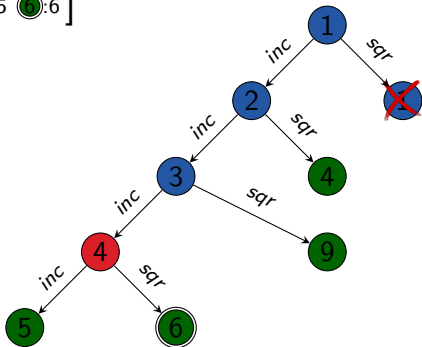


# Example

open: [  $\overset{\text{next}}{\downarrow}$  4:4 5:4 9:5 6:6 ]

closed: { 1, 2, 3, 4 }

bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$

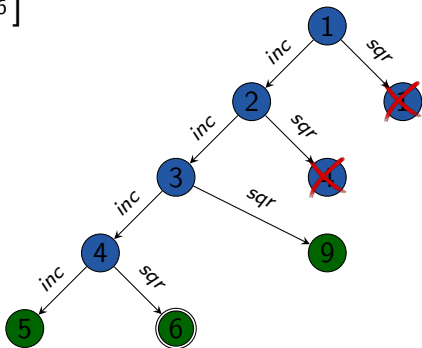


# Example

open: [  $\overset{\text{next}}{\downarrow}$  5:4 9:5 6:6 ]

closed: {1, 2, 3, 4}

bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$

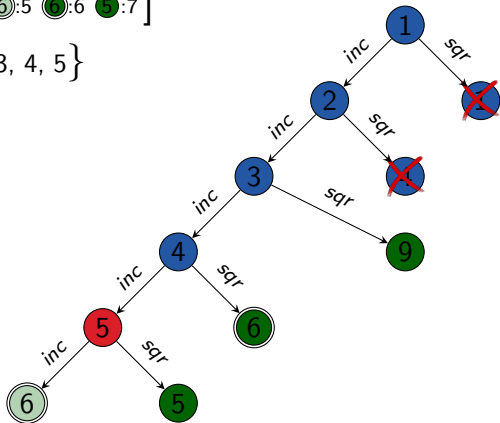




# Example

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

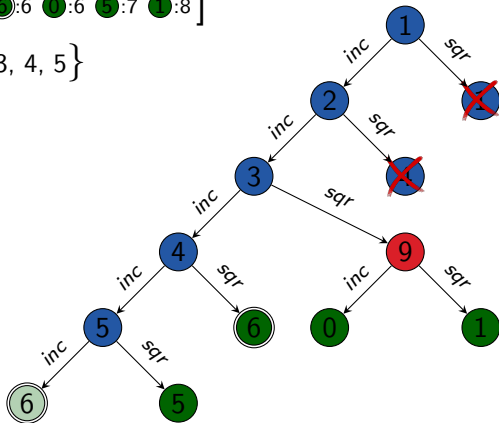
bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$



# Example

next  
↓  
open: [ ⑥:5 ⑤:6 ④:6 ③:7 ②:8 ]  
closed: { 1, 2, 3, 4, 5 }

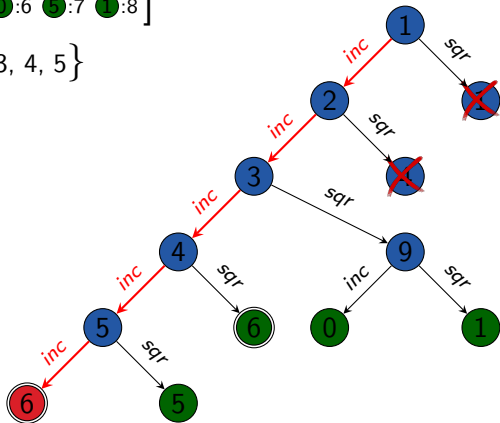
bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$



# Example

next  
↓  
open: [ 6:6 0:6 5:7 1:8 ]  
closed: { 1, 2, 3, 4, 5 }

bounded inc-and-square variant:  $\text{cost}(\text{sqr}) = 3$



# Uniform Cost Search: Improvements

## possible improvements:

- if action costs are small integers, **bucket heaps** often more efficient
- additional early duplicate tests for generated nodes can reduce memory requirements
  - can be beneficial or detrimental for runtime
  - must be careful to keep shorter path to duplicate state

# Properties

# Completeness and Optimality

properties of uniform cost search:

- uniform cost search is **complete** (Why?)
- uniform cost search is **optimal** (Why?)

# Time and Space Complexity

properties of uniform cost search:

- **Time complexity** depends on distribution of action costs (no simple and accurate bounds).
  - Let  $\varepsilon := \min_{a \in A} \text{cost}(a)$  and consider the case  $\varepsilon > 0$ .
  - Let  $c^*$  be the optimal solution cost.
  - Let  $b$  be the branching factor and consider the case  $b \geq 2$ .
  - Then the time complexity is at most  $O(b^{\lfloor c^*/\varepsilon \rfloor + 1})$ . (Why?)
  - often a very weak upper bound
- **space complexity** = time complexity

# Summary



# Summary

**uniform cost search:** expand nodes in order of **ascending path costs**

- usually as a graph search
- then corresponds to Dijkstra's algorithm
- **complete** and **optimal**