Linköping University

Automated Planning

Planning as Search

Jonas Kvarnström
Department of Computer and Information Science
Linköping University

jonas.kvarnstrom@liu.se – 2019

How to generate a plan?

jonkv@ida

One way of defining planning:

*Using **knowledge** about the world,*
including possible actions and their results,
*to **decide** what to do and when*
*in order to achieve an **objective**,*
***before** you actually start doing it*

→ How?

Is planning a straight-forward process of adding actions in the right order?

```
while (exists unvisited position) {
   pos ← nearest unvisited position
   plan += flyto(pos)
   plan += aim()
   plan += take-picture()
}
```

→

Usually, conditions are too complex; better to **test alternatives – search**

```
Generate some starting point
while (not complete) {
   try some alternatives
   create modified alternatives
   throw away some alternatives…
}
```

Planning as Search

jonkv@ida

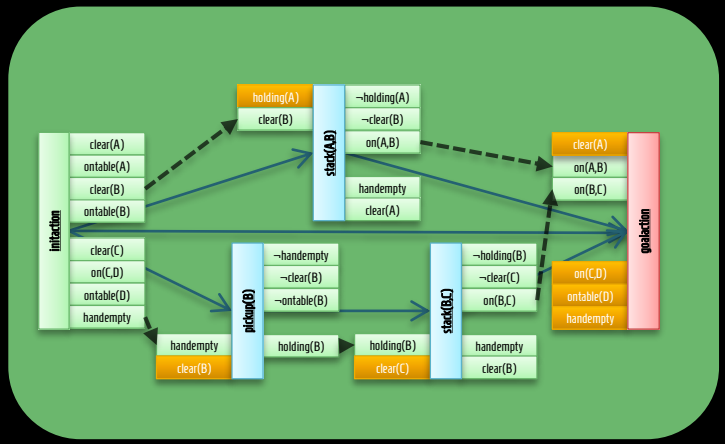
- To generate plans using search, we need:

1) A **node structure** defining what information is in a node

State space search: A node is a **state**

s={fact1,fact2,fact3,…}

Partial Order Causal Link:
A node is a complex **plan structure**



2) A way of creating an **initial node** from a problem instance

Search spaces are generally too large to represent completely

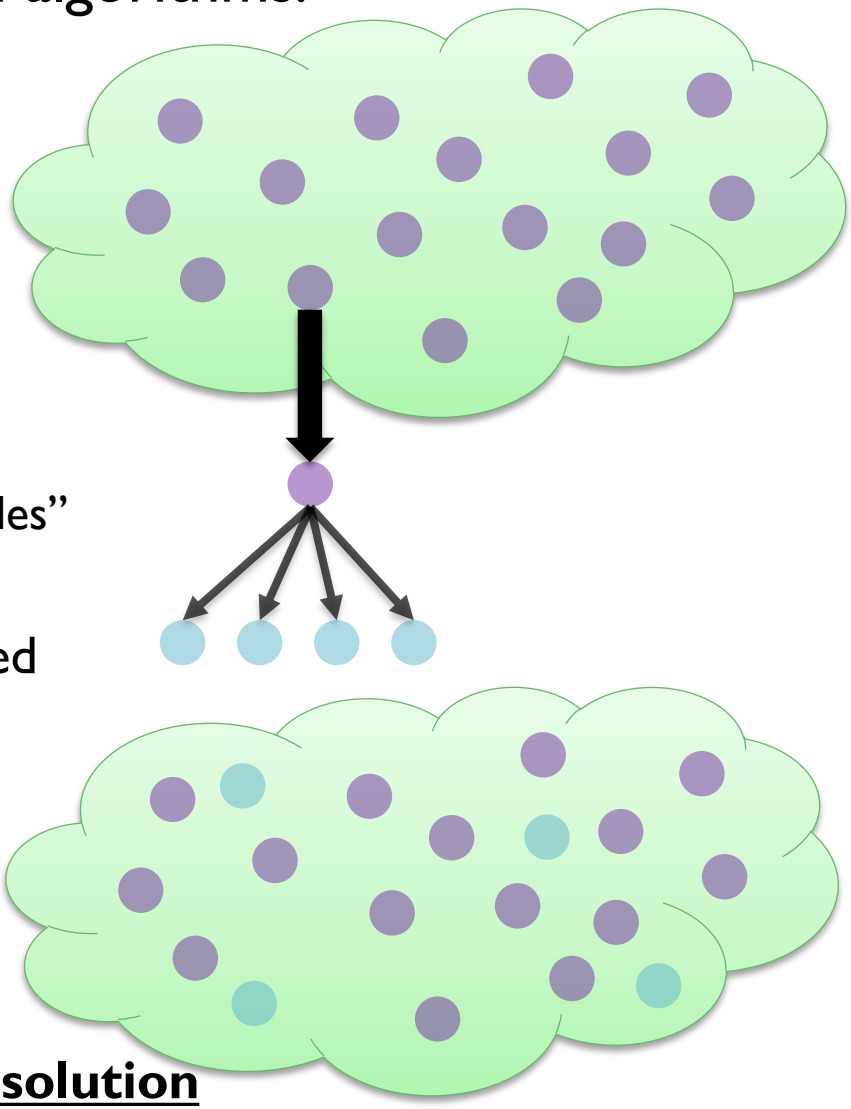
Need to **start somewhere** and then **expand the space incrementally**

Planning as Search (2)

jonkv@ida

- General way of **formalizing** search algorithms:
  - There are some ”open” nodes, that we:
    - Know how to reach
    - Haven’t explored yet

    At first: The **initial node**!
  - **Pick** / remove one of them
    - Using some *strategy* for picking ”good nodes”
  - Find **neighbor** nodes that can be created in a single step (whatever that step is!)
  - Put **those** back in the set of nodes
    - New options!
  - Repeat until a node **corresponds to a solution**

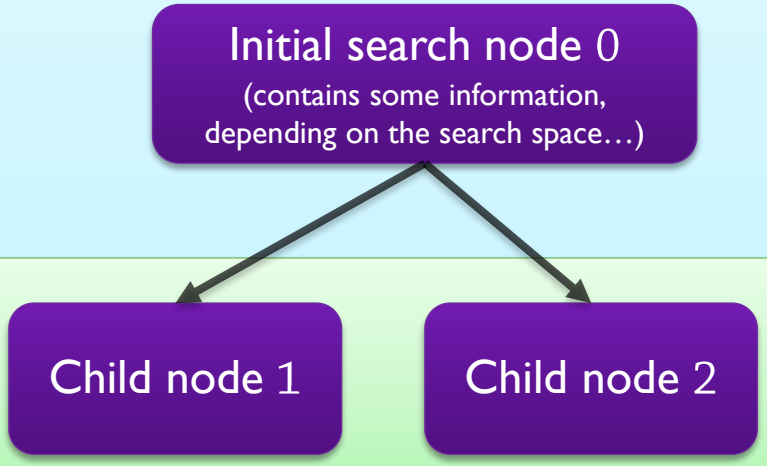


Planning as Search (3)

jonkv@ida

Search space

Classical planning: Finite number of search nodes



Edges could correspond to actions…
or to something completely different (POCL)!

3) A **successor function** / **branching rule** returning all successors (neighbors) of any search node

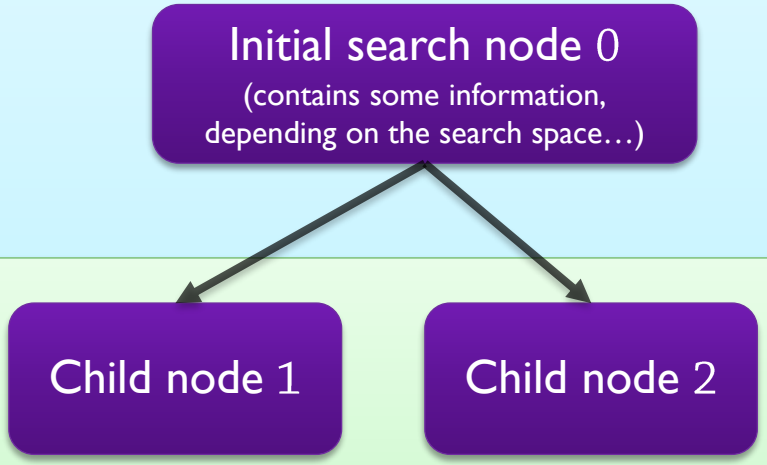
Expand a node = generate its successors

Planning as Search (4)

jonkv@ida

Search space

Classical planning: Finite number of search nodes



To know when we succeeded:

➔ 4) A **solution criterion**, detecting when a node corresponds to a **solution**

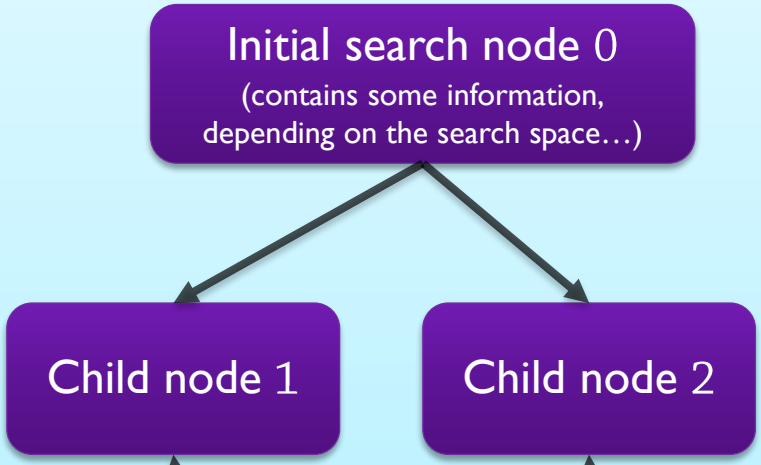
➔ 5) A **plan extractor**, telling us which **plan** a solution node corresponds to

Planning as Search (5)

jonkv@ida

Search space

Classical planning: Finite number of search nodes



Finally, we need to decide **how to search**:
6) A **search strategy**
chooses which node to expand next

Planning as Search (6)

jonkv@ida

- General **Search-Based Planning Algorithm**:

```
search(problem) {
    initial-node ← make-initial-node(problem) // [2]
    open ← { initial-node }
    while (open ≠ ∅) {
        node ← search-strategy-remove-from(open) // [6]
        if is-solution(node) then  // [4]
            return extract-plan-from(node) // [5]

        foreach newnode ∈ successors(node) { // [3]
            add newnode to open
        }
    }
    // Expanded the entire search space without finding a solution
    return failure;
}
```

Expand node

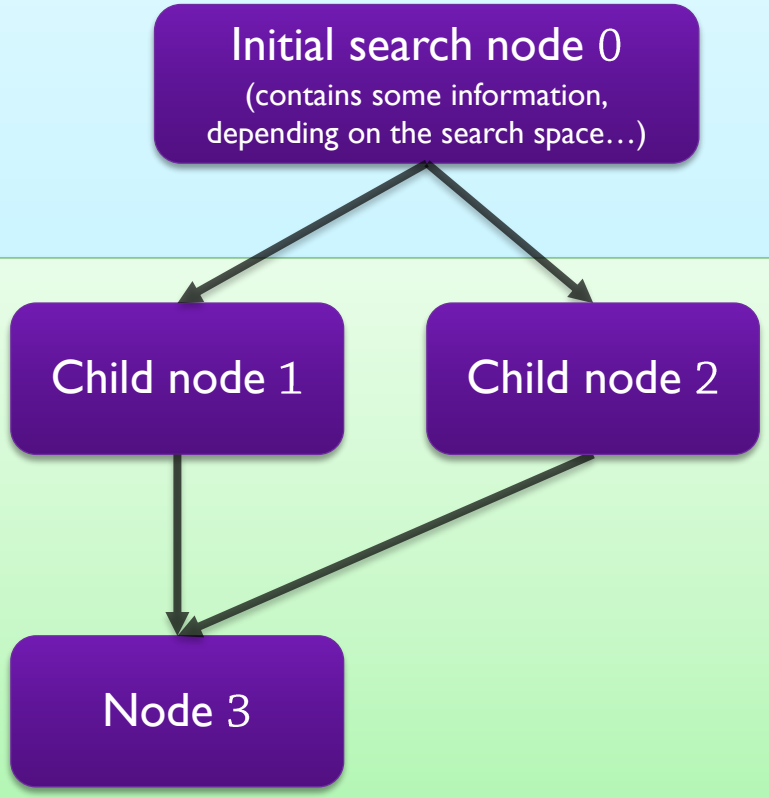
Searching Graphs

Searching Graphs (1)

jonkv@ida

Search space

Classical planning: Finite number of search nodes



In a **graph**, two nodes can share a successor!
Option 1: Keep track of all visited nodes, *detect* when the same successor is generated again

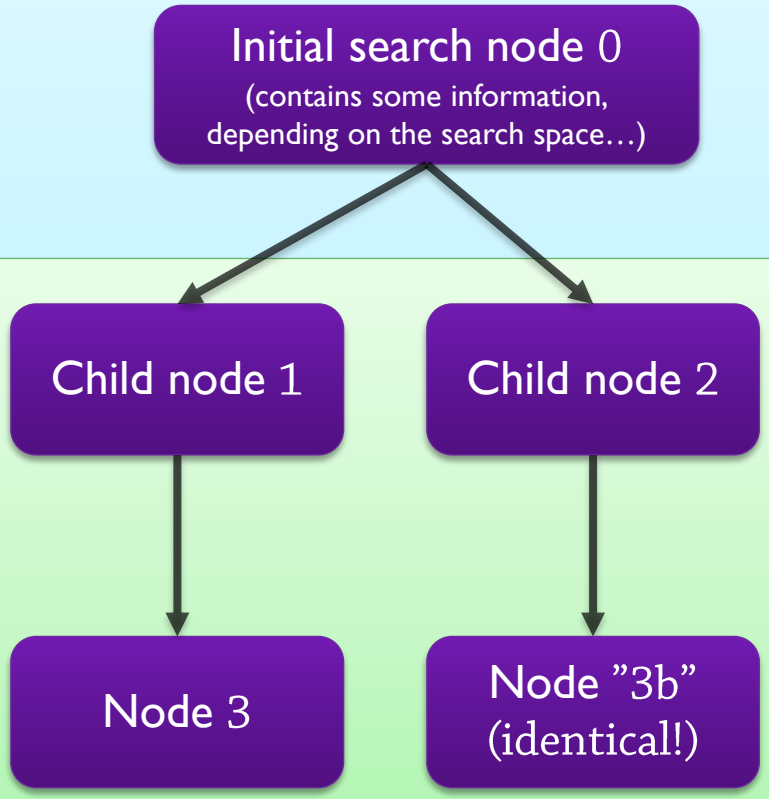
➔ Requires a lot of memory
➔ Only investigate a given node once, second time: don't expand it

Searching Graphs (2)

jonkv@ida

Search space

Classical planning: Finite number of search nodes



Option 2:

Don’t keep track of visited nodes

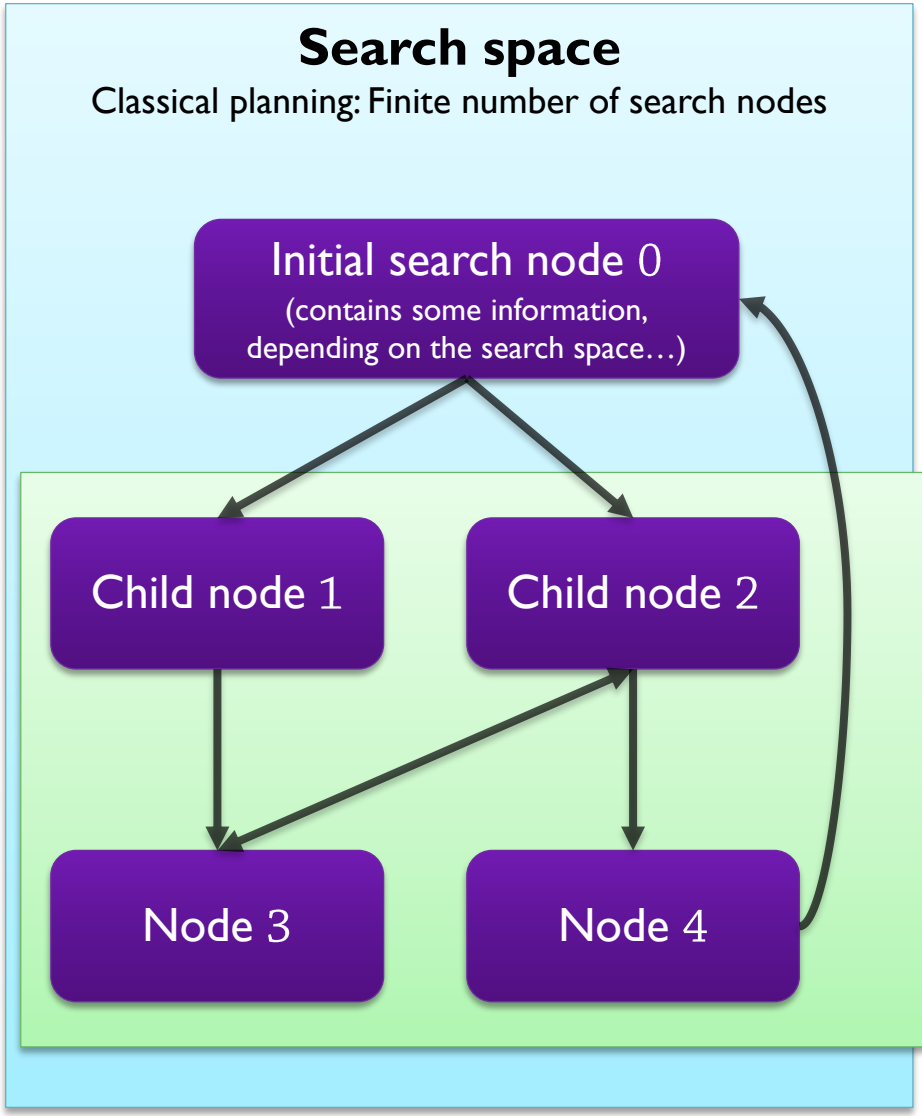
➔ Saves memory

➔ Investigate some subtrees multiple times

➔ The search space is traversed as a *tree*

Searching Graphs (3)

jonkv@ida



An **ancestor** may also be a **successor**
➔ **loops** in the search graph

Depending on the search **strategy**,
it may or may not be necessary
to detect and handle this

Searching Graphs (4)

jonkv@ida

- To avoid searching subgraphs twice (shared successors + loops):
  - **search**(*problem*) {

```
initial-node ← make-initial-node(problem) // [2]
open ← { initial-node }
added ← { initial-node }
while (open ≠ ∅) {
    node ← search-strategy-remove-from(open) // [6]
    if is-solution(node) then // [4]
        return extract-plan-from(node) // [5]

    foreach newnode ∈ successors(node) { // [3]
        if not (newnode ∈ added)
            add node' to open
            add newnode to added
    }
}
// Expanded the entire search space without finding a solution
return failure;
```

}

Nodes are removed from *open*, but not from *added*

Aspects of Search

Defined by the search space

1) Node structure

2) Generating initial search node

3) Branching rule, creating successors

4) Determining if a node is a solution

5) Extracting a plan from a node

Forward State Space, Backward Goal Space, Partial Order Causal Link, …, …, …

Defined by the search strategy

6) Deciding which node to expand next

Uninformed

- Depth first (DFS)
- Breadth first
- Dijkstra's algorithm
- Uniform cost
- Depth limited DFS
- Iterative deepening DFS
- …

Informed

- Greedy Best First
- A*
- Weighted A*
- Iterative deepening A*
- Beam Search
- Hill Climbing
- Enforced Hill Climbing
- Simul. Annealing
- …

↓

Heuristics!

Independence!

Search Spaces
Forward state space
Backward goal space
Partial Order Causal Link
Hierarchical Task Networks

Search Strategies
Hill Climbing
Enforced Hill Climbing
A*
(Repeated) Weighted A*

Heuristics
Goal count
Landmarks
Pattern Databases
Relaxation, Delete Relaxation
Relaxed Planning Graphs

Tweaking Search Spaces
Predicates vs state variables
Lifted search spaces

Tweaking Search Strategies
Helpful actions / Preferred operators
Dual Queues, Boosted Dual Queues
Lazy Search

Meta search strategies
Portfolio planning