Automated Planning & Artificial Intelligence

Uninformed and Informed search in state space

Humbert Fiorino

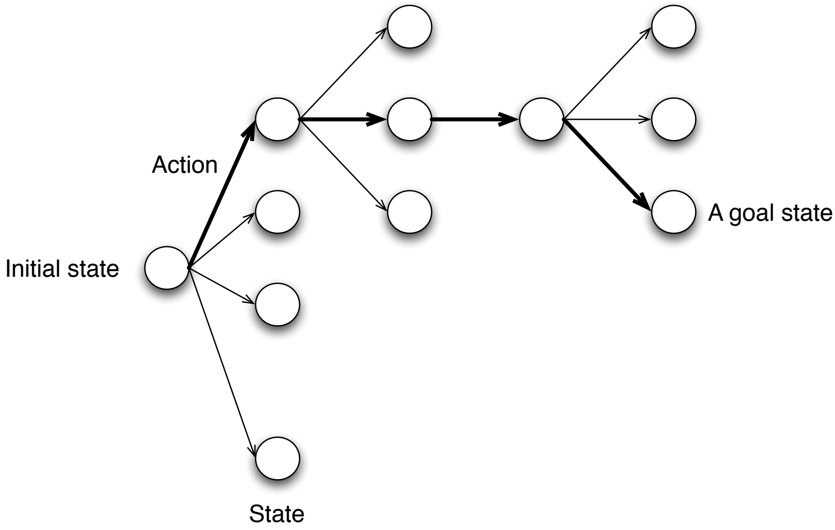
Humbert.Fiorino@imag.fr
http://membres-lig.imag.fr/fiorino

Laboratory of Informatics of Grenoble – MAGMA team

February 2011

Introduction

- There are many planning approaches. *State space planning* is the most obvious :
  - The search space is a subset of the state space
  - Each node corresponds to a state of the world, each arc corresponds to a state transition.
  - A plan is a path in the search space from the initial state to a goal state.


Action
A goal state
Initial state
State

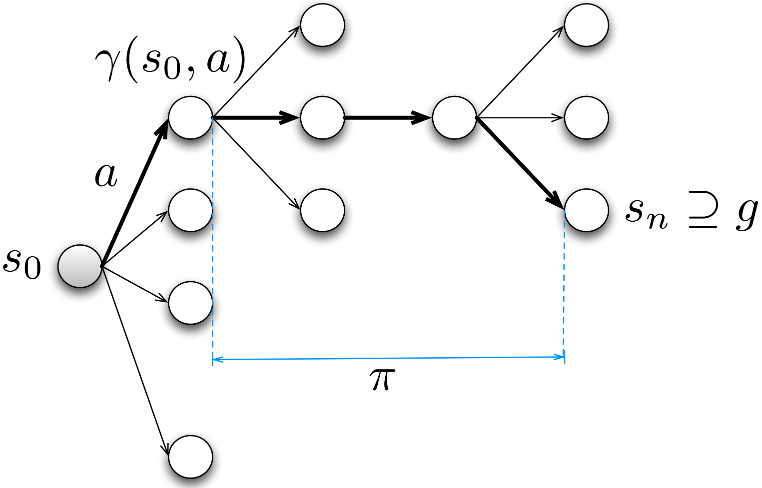
Forward Search Algorithm

- The forward search algorithm takes as input the statement $P = (O, s_0, g)$ of a planning problem P.
- If P is solvable, then Forward-search(O, s_0, g) returns a solution plan, otherwise it returns failure.
- The plan returned by each recursive invocation of the algorithm is called a partial solution because it is part of the final solution returned by the top level invocation.

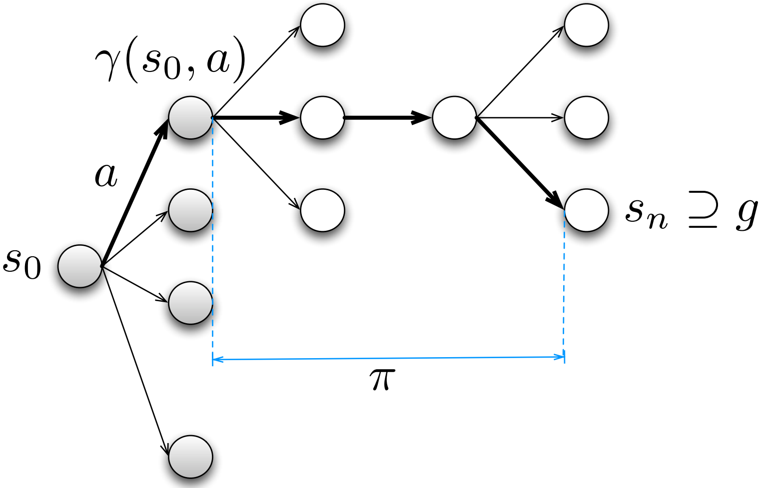
$ForwardSearch(\mathcal{O}, s_0, g)$

```
begin
    if s₀ satisfies g then
        return [];
    else
        applicable ← {a | a is applicable in s₀};
        if applicable = ∅ then
            return ⊥;
        else
            Nondeterministically choose a ∈ applicable;
            π ← ForwardSearch(O, γ(s₀, a), g);
            if π ≠ ⊥ then
                return a · π;
            else
                return ⊥;
end
```

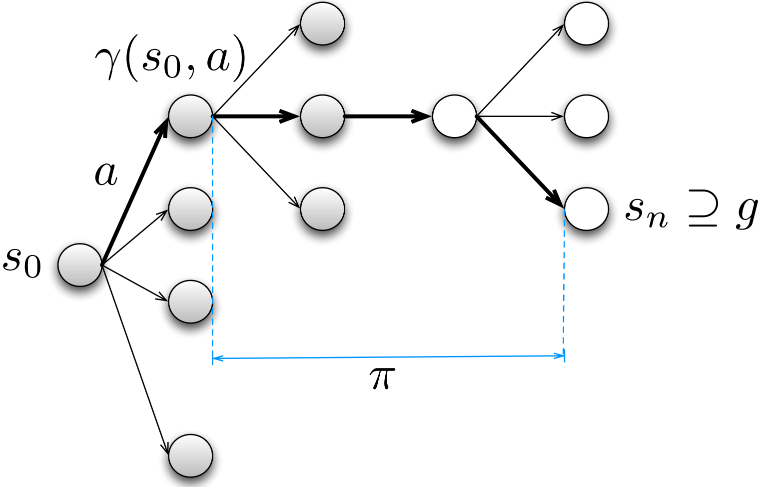
Breadth-first search



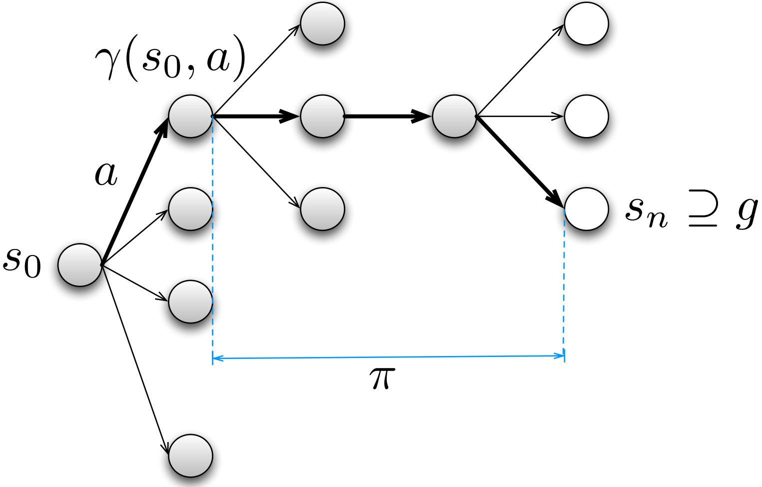
Breadth-first search



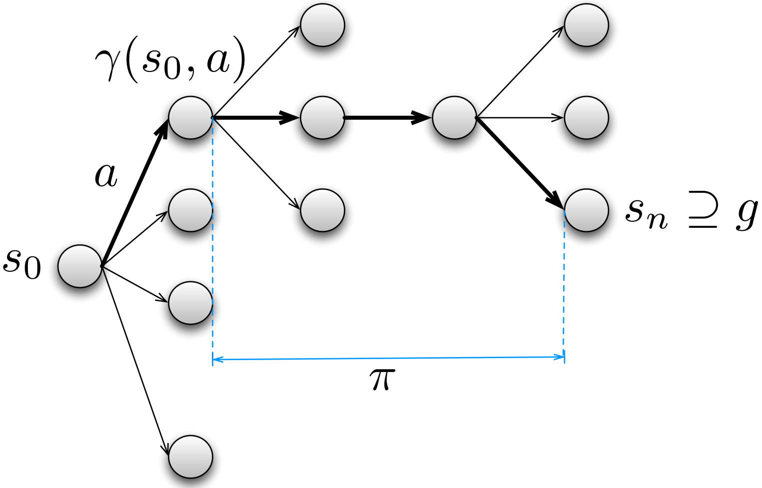
Breadth-first search



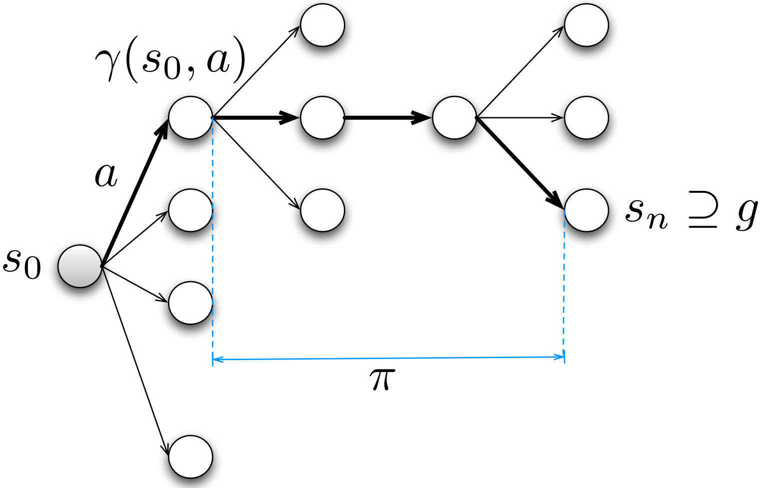
Breadth-first search



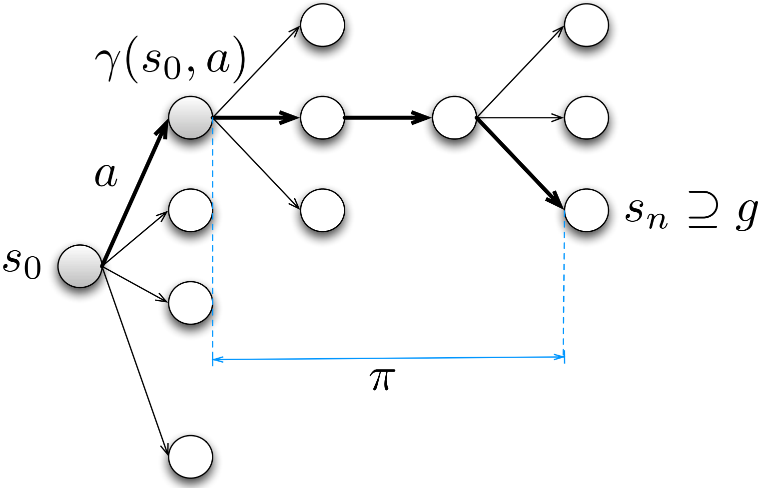
Breadth-first search



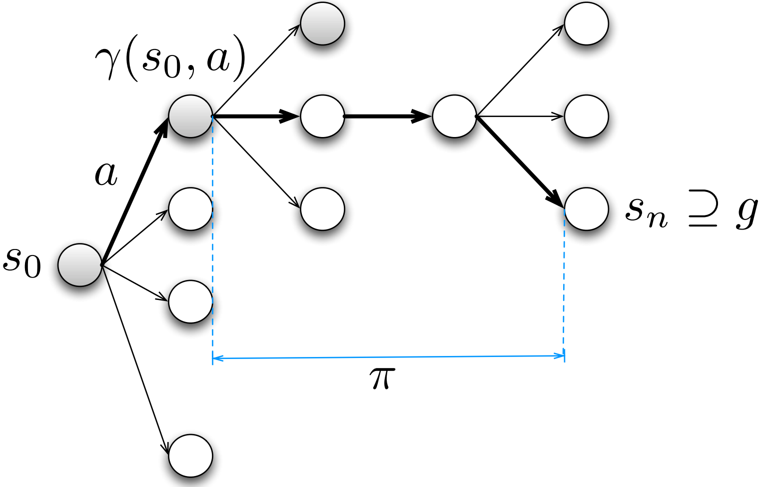
Depth-first search with backtracking



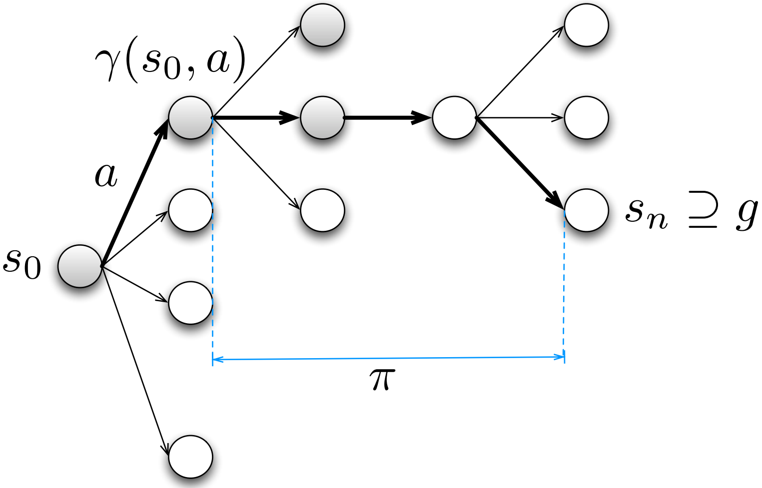
Depth-first search with backtracking



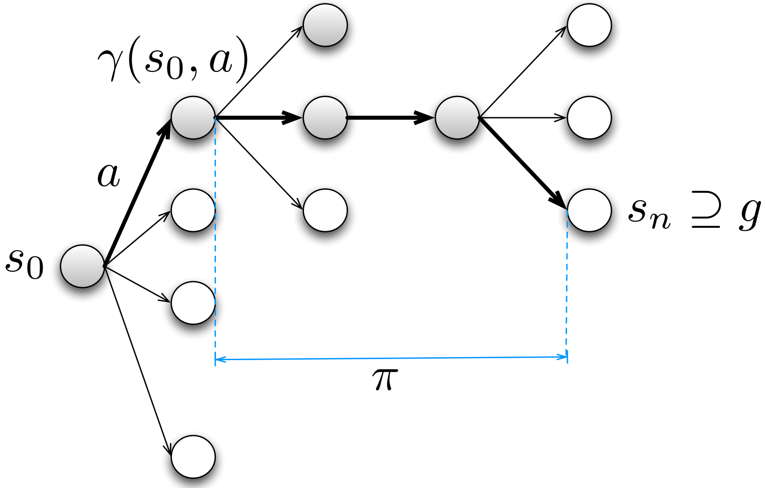
Depth-first search with backtracking



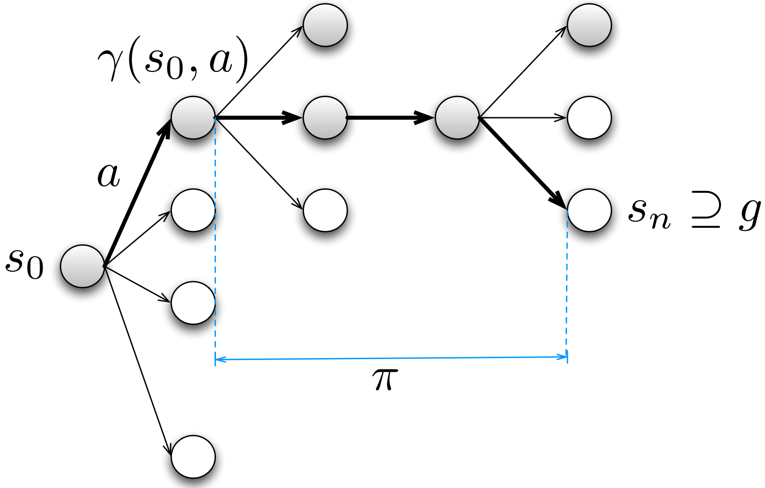
Depth-first search with backtracking



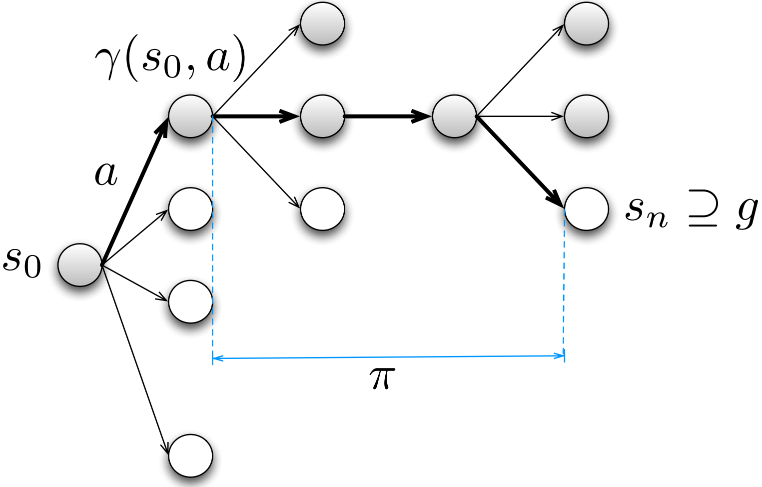
Depth-first search with backtracking



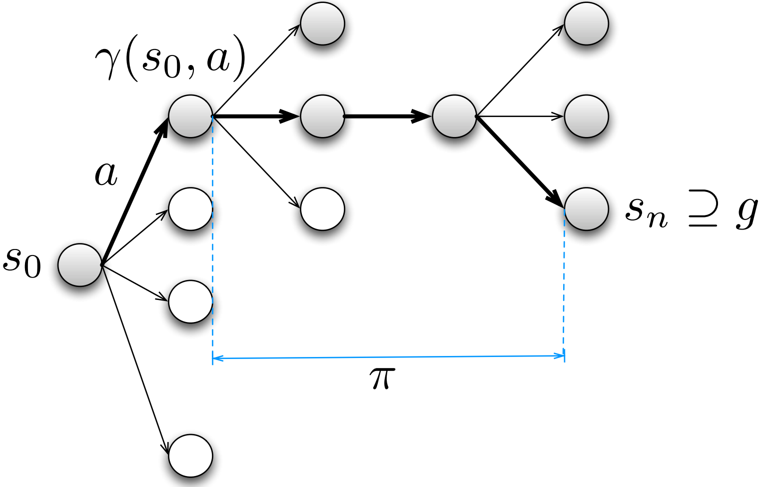
Depth-first search with backtracking



Depth-first search with backtracking



Depth-first search with backtracking

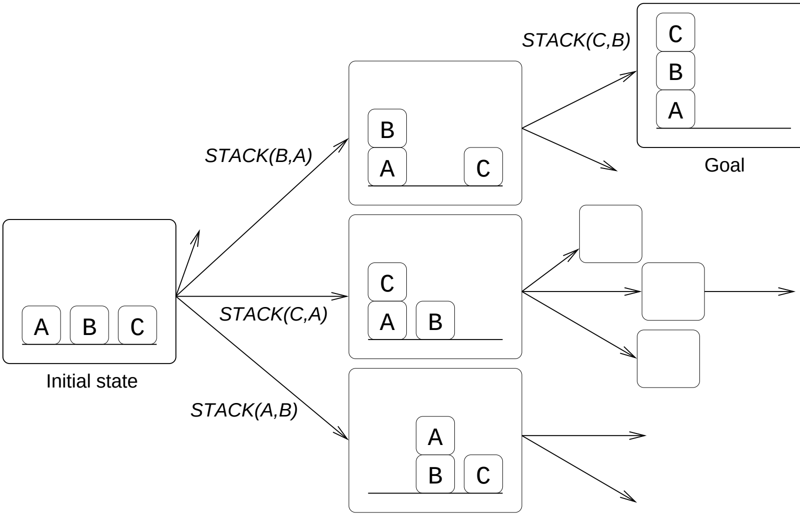


Depth-first deterministic implementation

- Loops have to be pruned $\Rightarrow$ store execution trace $(s_0, \ldots, s_k)$ and return $\perp$ each time $s_k = s_i$ with $i < k$.
- This implies :
  1. The algorithm returns $\perp$ for any infinite branch
  2. This property does not make it return $\perp$ on branches leading to a solution.

1. is satisfied by the number of state is finite. Then, any infinite branch necessarily entails $s_k = s_i$ with $i < k$: the algorithm returns $\perp$.
2. To make the algorithm return $\perp$, we need $s_k = s_i$ with $i < k$. Reasoning by contradiction : suppose that there is a finite execution trace $(s_0, \ldots, s_i, \ldots, s_k, \ldots, s_n)$ with $s_k = s_i$ $(i < k)$. Thus, $(s_0, \ldots, s_{i-1}, s_k, \ldots, s_n)$ is also a solution trace and it is not pruned.

Forward Search Algorithm



Soundness

Theorem

Any plan π returned by ForwardSearch$(\mathcal{O}, s_0, g)$ is a solution of $P = (\mathcal{O}, s_0, g)$

- We must show that the algorithm builds up a sequence of applicable actions. The proof is by induction on the call number k.
  - Basis : $k = 1$, s_0 satisfies g, therefore $\pi = []$ is a solution plan
  - Induction step : By induction hypothesis, any plan $\pi \leftarrow \mathit{ForwardSearch}(\mathcal{O}, \gamma(s_0, a), g)$ with $k \geq 1$ is a solution for $(\mathcal{O}, \gamma(s_0, a), g)$. For $k + 1$, a is applicable to s_0. Then, $a \cdot \pi$ is a solution for $P = (\mathcal{O}, s_0, g)$.

Completeness

Let $P = (\mathcal{O}, s_0, g)$ and let Π be the solution set of P. $\forall \pi \in \Pi$, there is at least one deterministic execution trace of $ForwardSearch(\mathcal{O}, s_0, g)$ that returns π.

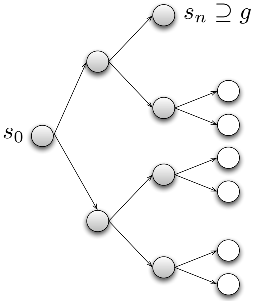
- We must show that, if there is a solution $\pi = [a_1, \ldots, a_k]$, the algorithm necessarily finds it. The prove is by induction on the plan length $k = |\pi|$:
  - Basis : For $k = 0$, $\pi = []$ is returned by line 3.
  - Induction step : In order to return execution trace with length k, necessarily $|ForwardSearch(\mathcal{O}, \gamma(s_0, a), g)| = k - 1$, which is verified by the induction hypothesis. Indeed, $\pi = ForwardSearch(\mathcal{O}, \gamma(s_0, a), g) = [a_2, \ldots, a_k]$ and $|a \cdot \pi| = k$.

Breadth-first search complexity

- Suppose that the branching factor is b and the solution is at depth d. In the worst case,

$$b + b^2 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

nodes are generated and stored ! (time and space complexity are equivalent)



Depth-first search complexity

- It has very modest memory requirements : a single path from the root to a leaf node $= O(m)$, where m is the maximum depth of any node (space complexity).
- In the worst case, all the $O(b^m)$ nodes in the search tree are generated (time complexity).

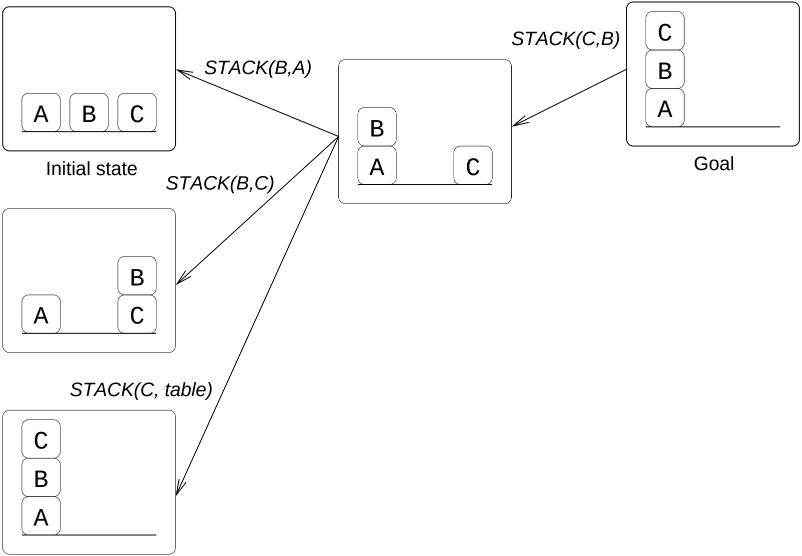
$BackwardSearch(\mathcal{O}, s_0, g)$

Action a is relevant for g iff $g \cap effect^{+}(a) \neq \emptyset$ et $g \cap effect^{-}(a) = \emptyset$:

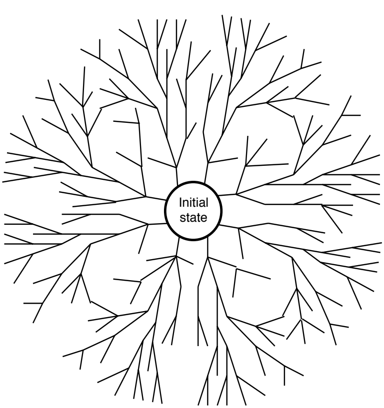
$$\gamma^{-1}(g, a) = (g - effect(a)) \cup precond(a)$$

begin
  if s_0 *satisfies* g **then**
    return [];
  else
    $relevant \leftarrow \{a \mid a \text{ is relevant in } g\}$;
    if $relevant = \emptyset$ **then**
      return $\perp$;
    else
      Nondeterministically choose $a \in relevant$;
      $\pi \leftarrow BackwardSearch(\mathcal{O}, s_0, \gamma^{-1}(g, a))$;
      if $\pi \neq \perp$ **then**
        return $\pi \cdot a$;
      else
        return $\perp$;
end

Backward Search Algorithm



Bidirectional search



Divide and conquer

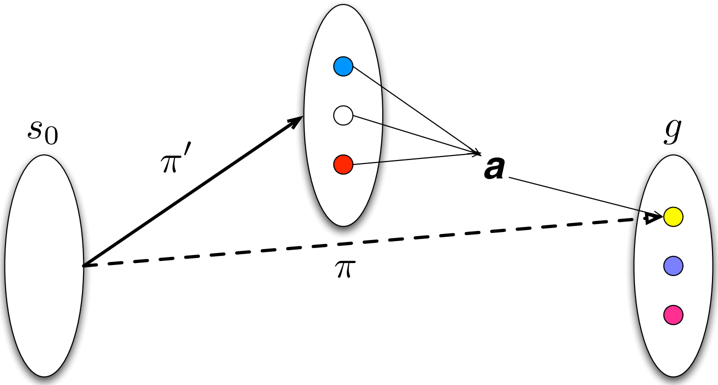
- The biggest issue = how to improve efficiency by reducing the size of the search space ?
- Answer : apply "divide and conquer" strategy :
  - divide the problem to a set of smaller sub-problems, solve each sub-problem independently, combine the results to form the solution
  - In planning we would like to satisfy a set of goals : Divide the planning goals along individual goals, solve (find a plan for) each of them independently, combine the plan solutions in the resulting plan
  - Is it always safe ? No, there can be interacting goals.

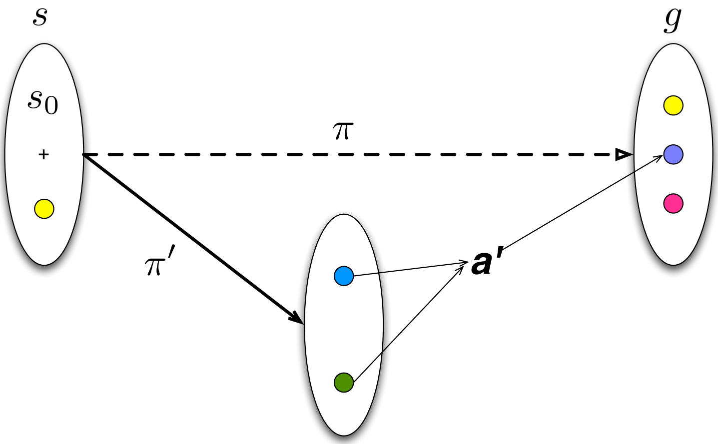
STRIPS algorithm

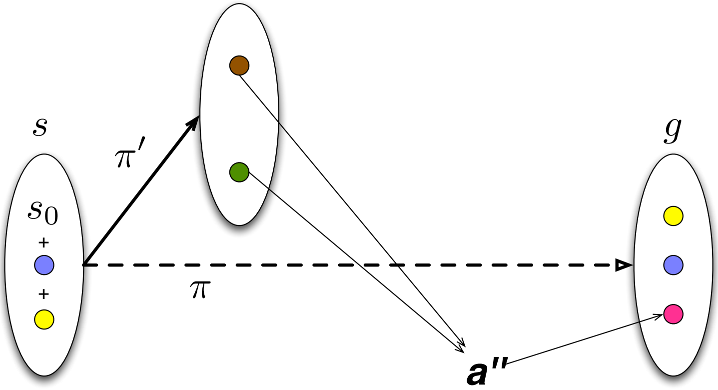
- STRIPS is somewhat similar to the BackwardSearch but differs in the following steps :
  1. In each recursive call of STRIPS, the only subgoals eligible to be worked on are the preconditions of the last operator added to the plan = reduce the branching factor substantially but makes STRIPS **incomplete**.
  2. If the current state satisfies all of an operator's preconditions, STRIPS executes that operator and will not backtrack = prunes the search space but makes STRIPS **incomplete**.

$STRIPS(\mathcal{O}, s, g)$

```
begin
    π ← [];
    repeat
        relevant ← {a | a is relevant for g − s};
        if relevant = ∅ then
            return ⊥;
        else
            Nondeterministically choose a ∈ relevant;
            π′ ← STRIPS(O, s, precond(a));
            if π′ ≠ ⊥ then
                s ← γ(s, π′ · a);
                π ← π · π′ · a;
            else
                return ⊥;
    until s satisfies g ;
    return π;
end
```


s_0
π'
a
g
π


s
s_0
g
π
π'
a'


s
s_0
+
+
π'
π
g
a''

Sussman's Anomaly

Initial state Goal

g = {(on A B)(on B C)(on−table C)(clear A)(arm−empty)}

- Plan for first goal (on A B) :

$$[UNSTACK(C, A), PUTDOWN(C), PICKUP(A), STACK(A, B)]$$

- Now plan for second goal On(B,C) :

$$[UNSTACK(A, B), PUTDOWN(A), PICKUP(B), STACK(B, C)]$$

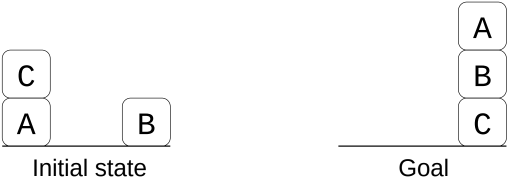
- Plan for second goal (on B C) :

$$[PICKUP(B), STACK(B, C)]$$

- Now plan for first goal (on A B) :

$$[UNSTACK(B, C), PUTDOWN(B), UNSTACK(C, A), PUTDOWN(C), PICKUP(A)STACK(A, B)]$$

Sussman's Anomaly



g = {(on A B)(on B C)(on−table C)(clear A)(arm−empty)}

- Plan for first goal (on A B) :

$$[UNSTACK(C, A), PUTDOWN(C), PICKUP(A), STACK(A, B)]$$

- Now plan for second goal On(B,C) :

$$[UNSTACK(A, B), PUTDOWN(A), PICKUP(B), STACK(B, C)]$$

- Plan for second goal (on B C) :

$$[PICKUP(B), STACK(B, C)]$$

- Now plan for first goal (on A B) :

$$[UNSTACK(B, C), PUTDOWN(B), UNSTACK(C, A), PUTDOWN(C), PICKUP(A)STACK(A, B)]$$

Sussman's Anomaly

g = {(on A B)(on B C)(on−table C)(clear A)(arm−empty)}

- Plan for first goal (on A B) :

$$[UNSTACK(C, A), PUTDOWN(C), PICKUP(A), STACK(A, B)]$$

- Now plan for second goal On(B,C) :

$$[UNSTACK(A, B), PUTDOWN(A), PICKUP(B), STACK(B, C)]$$

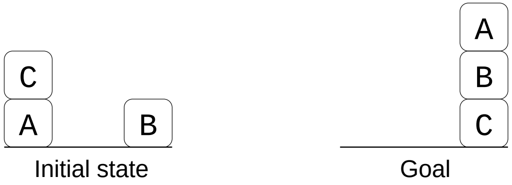
- Plan for second goal (on B C) :

$$[PICKUP(B), STACK(B, C)]$$

- Now plan for first goal (on A B) :

$$[UNSTACK(B, C), PUTDOWN(B), UNSTACK(C, A), PUTDOWN(C), PICKUP(A)STACK(A, B)]$$

Sussman's Anomaly



g = {(on A B)(on B C)(on−table C)(clear A)(arm−empty)}

- Plan for first goal (on A B) :

$$[UNSTACK(C, A), PUTDOWN(C), PICKUP(A), STACK(A, B)]$$

- Now plan for second goal On(B,C) :

$$[UNSTACK(A, B), PUTDOWN(A), PICKUP(B), STACK(B, C)]$$

- Plan for second goal (on B C) :

$$[PICKUP(B), STACK(B, C)]$$

- Now plan for first goal (on A B) :

$$[UNSTACK(B, C), PUTDOWN(B), UNSTACK(C, A), PUTDOWN(C), PICKUP(A)STACK(A, B)]$$

Introduction

- Uninformed search strategies are incredibly inefficient in most cases
- Informed search uses problem-specific knowledge and can find solution more efficiently

Best-first search

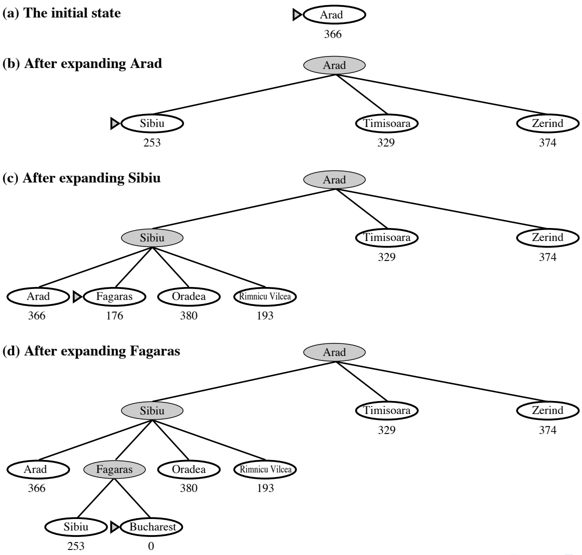
- In best-first search, a node is selected for expension based on an **evaluation function**, $f(n)$
- Traditionnally, the node with the lowest evaluation because the evaluation measures the distance to the goal
- "Best-first search" is inaccurate : if we could really expand the best node first, it would not be a search at all !
- A key component, the **heurisitic function**, $h(n)$, that estimates the cost of the cheapest path from node n to a goal node
- If n is the goal, then $h(n) = 0$

Greedy search

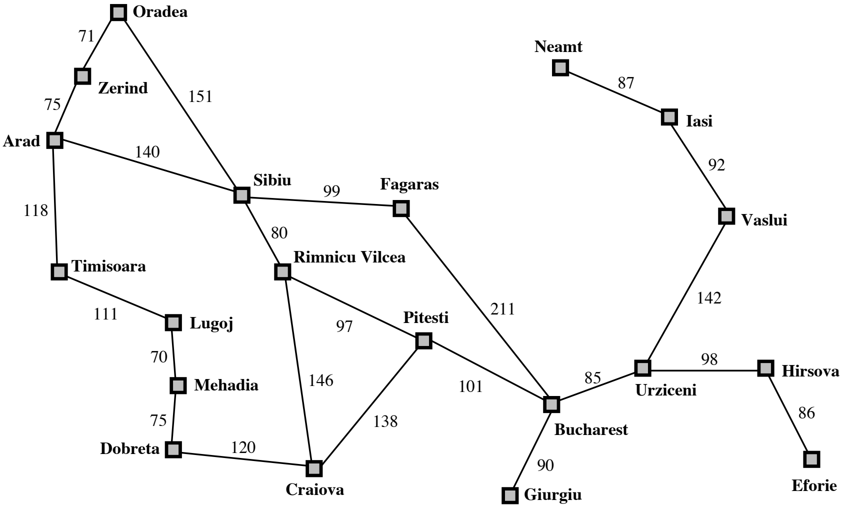
- Expand the node that is closest to the goal on the grounds that is likely to lead to a solution quickly : $f(n) = h(n)$
- For instance, using the straight-line distance to Bucarest...

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

From Arad to Bucarest



From Arad to Bucarest



A^* search

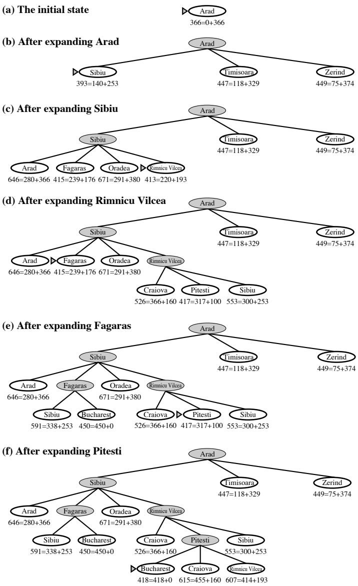
- It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the estimated cost to get from the node to the goal :

$$f(n) = g(n) + h(n)$$

- $f(n) =$ estimated cost of the cheapest solution through n.
- A^* is optimal if $h(n)$ is an **admissible heuristic** i.e. $h(n)$ never overestimates the cost to reach the goal.

The A^* algorithm

1. Put the start node s on OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN and place on CLOSED a node n for which f is minimum.
4. if n is a goal node, exit successfully with the solution obtained by tracing back the pointer from n to s.
5. Otherwise expand n, generating all its sucessors, and attach to them pointers back to n. For every successor n' of n :
   (a) If n' is not already on OPEN or CLOSED, estimate $h(n')$ and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$.
   (b) If n' is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(n')$.
   (c) If n' required pointer adjustement and was found on CLOSED, reopen it.
6. Go to step 2.


(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Sibiu
591=338+253
Bucharest
450=450+0
(f) After expanding Pitesti
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

- Informed search is more efficient than uninformed search but automated planning needs **domain-independent** heuristics !
- For instance, to solve the 8-puzzle problem, we will use the Manhattan distance etc.