Theoretical Research Methods ... Illustrated in *AI Planning*

Pascal Bercher

Planning & Optimization
Research School of Computer Science
College of Engineering and Computer Science
the Australian National University

April 17, 2020

Motivation

General Questions Covered

- What are important/interesting properties of algorithms?
- What does it mean that one algorithm is better than another?
- How does one *prove* such properties? E.g., how does one show:
  - termination?
  - that one algorithm is better than another?

→ Illustrated with *AI planning* and *planning heuristics*.

Why AI Planning? What is it?

Informal Description

Patrik Haslum

Planning is the art and practice of thinking before acting.

Jörg Hoffmann

Selecting a goal-leading course of action based on a high-level description of the world.

Just a bit more formally...

Planning is the reasoning process required to generate a *plan* – a sequence of action that transforms a given state of a system into a desired one.

Why AI Planning? What is it?

Games, e.g., Solitaire

Why AI Planning? What is it?

Games, e.g., Sliding Tile Puzzle, 15 Puzzle, n^2-1 Puzzle

Initial State

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

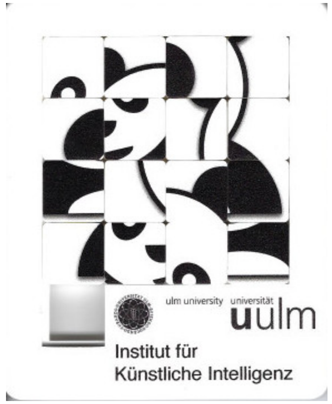
→

Goal State

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Why AI Planning? What is it?

Games, e.g., Sliding Tile Puzzle, 15 Puzzle, n^2-1 Puzzle

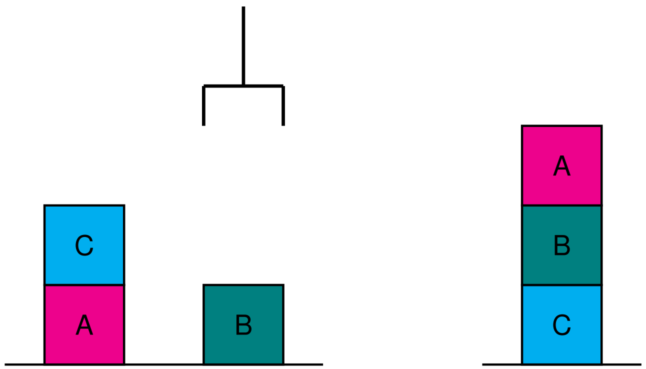


Initial State

Goal State

Why AI Planning? What is it?

Blocksworld



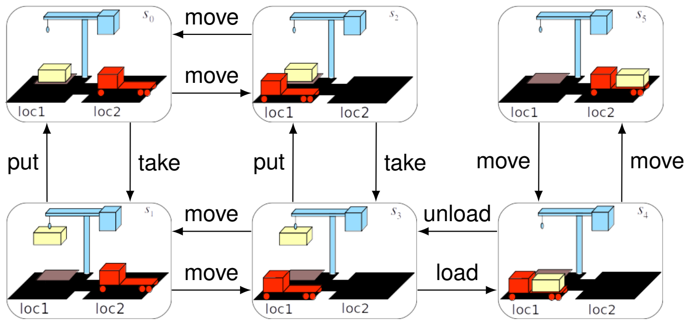
Start Configuration

Desired Configuration

Standard Planning Benchmark in the International Planning Competition (IPC) and *every* planning lecture.

Why AI Planning? What is it?

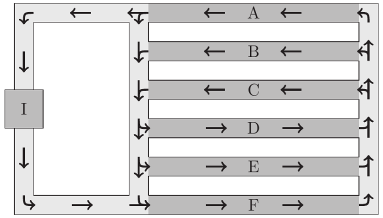
Cranes in a Harbor



Title: Lecture Slides for Automated Planning
Source: `http://www.cs.umd.edu/~nau/planning/slides/chapter01.pdf`

Why AI Planning? What is it?

Greenhouse



Source: `https://www.lemnatec.com/`

Further reading:

- Malte Helmert and Hauke Lasinger. "The Scanalyzer Domain: Greenhouse Logistics as a Planning Problem". In: *Proc. of the 20th Int. Conf. on Automated Planning and Scheduling (ICAPS 2010)*. AAAI Press, 2010, pp. 234–237
- The IPC Scanalizer Domain in PDDL (see paper above).

Problem Definition

What is Classical Planning?

We focus on the "base case" of AI planning: *Classical Planning*

- Discrete (no time).
- Deterministic.
- Fully observable.
- Single-agent.

More formally, a classical planning problem consists of:

- A finite set of (deterministic and discrete) actions.
- A (fully known) initial state.
- A set of (fully known) goal states.

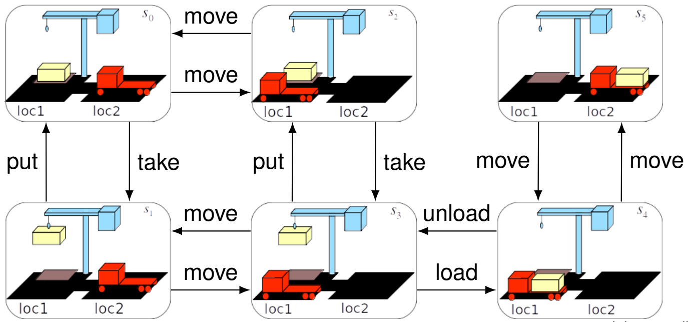
A *solution* (or *plan*) is any sequence of actions transforming the initial state into a goal state.

Problem Definition

Formalism

A classical planning problem $\mathcal{P} = \langle V, A, s_I, g \rangle$ consists of:

- V is a finite set of *state variables* (also called: *facts* or *propositions*).



copyright: see slide 6

$V = \{$CrateAtLoc1, HoldCrate, TruckAtLoc1, TruckAtLoc2, CrateInTruck$\}$

Problem Definition

Formalism

A classical planning problem $\mathcal{P} = \langle V, A, s_I, g \rangle$ consists of:

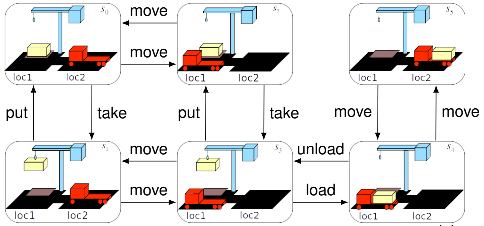
- V is a finite set of *state variables* (also called: *facts* or *propositions*).
  - *States* are collections of state variables.
  - We assume the *closed world assumption*, i.e., all variables not mentioned in a state s do not hold in that state (in contrast to: it's not known whether they hold or not).
  - $S = 2^V$ is called the *state space*.

Problem Definition

Formalism

A classical planning problem $\mathcal{P} = \langle V, A, s_I, g \rangle$ consists of:

- A is a finite set of actions. Each action $a \in A$ is a tuple $(pre, add, del, c) \in 2^V \times 2^V \times 2^V \times \mathbb{R}_0^+$ consisting of a *precondition*, *add and delete list*, and action costs. (We often only give a 3-tuple if there are no action costs.)



copyright: see slide 6

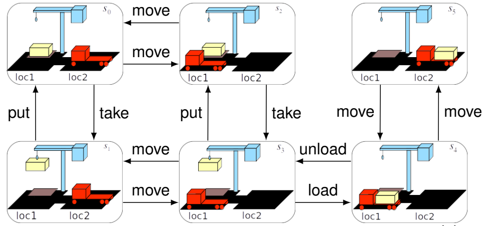
take	pre:	{CrateAtLoc1}	**put**	pre:	{HoldCrate}
	add:	{HoldCrate}		add:	{CrateAtLoc1}
	del:	{CrateAtLoc1}		del:	{HoldCrate}

Problem Definition

Formalism

A classical planning problem $\mathcal{P} = \langle V, A, s_I, g \rangle$ consists of:

- A is a finite set of actions. Each action $a \in A$ is a tuple $(pre, add, del, c) \in 2^V \times 2^V \times 2^V \times \mathbb{R}_0^+$ consisting of a *precondition*, *add and delete list*, and action costs. (We often only give a 3-tuple if there are no action costs.)



copyright: see slide 6

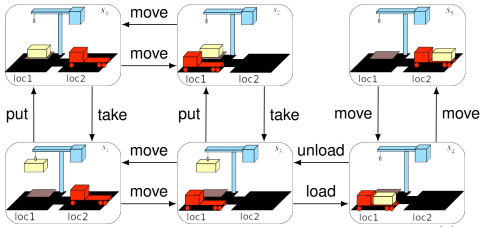
moveLeft	pre:	{TruckAtLoc2}	**moveRight**	pre:	{TruckAtLoc1}
	add:	{TruckAtLoc1}		add:	{TruckAtLoc2}
	del:	{TruckAtLoc2}		del:	{TruckAtLoc1}

Problem Definition

Formalism

A classical planning problem $\mathcal{P} = \langle V, A, s_I, g \rangle$ consists of:

- A is a finite set of actions. Each action $a \in A$ is a tuple $(pre, add, del, c) \in 2^V \times 2^V \times 2^V \times \mathbb{R}_0^+$ consisting of a *precondition*, *add and delete list*, and action costs. (We often only give a 3-tuple if there are no action costs.)



copyright: see slide 6

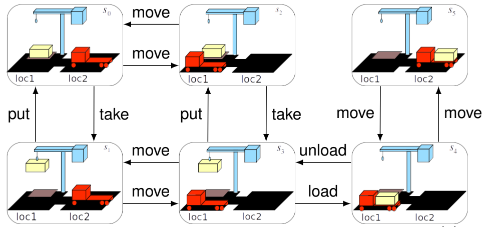
load	pre:	{HoldCrate, TruckAtLoc1}	**unload**	pre:	{CrateInTruck, TruckAtLoc1}
	add:	{CrateInTruck}		add:	{HoldCrate}
	del:	{HoldCrate}		del:	{CrateInTruck}

Problem Definition

Formalism

A classical planning problem $\mathcal{P} = \langle V, A, s_I, g \rangle$ consists of:

- $s_I \in S$ is the initial state (complete state description).
- $g \subseteq V$ is the goal description (encodes a set of goal states).



copyright: see slide 6

$s_I = \{\text{CrateAtLoc1}, \text{TruckAtLoc2}\}$
$g = \{\text{CrateInTruck}, \text{TruckAtLoc2}\}$

Problem Definition

Formalism, cont'd I

Action application:

- An action $a \in A$ is called *applicable* (or executable) in a state $s \in S$ if and only if $pre(a) \subseteq s$. Often, this is given by a function: $\tau(a, s) \Leftrightarrow pre(a) \subseteq s$.
- If $\tau(a, s)$ holds, its application results into the successor state $\gamma(a, s) = (s \setminus del(a)) \cup add(a)$. $\gamma : A \times S \to S$ is called the *state transition function*.

→ Example: The action

take	pre:	{CrateAtLoc1}
	add:	{HoldCrate}
	del:	{CrateAtLoc1}

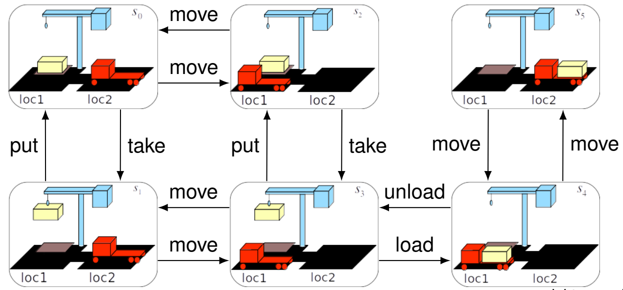
is applicable in state $\{CrateAtLoc1, TruckAtLoc2\}$ resulting into $\{TruckAtLoc2, HoldCrate\}$.

Problem Definition

Formalism, cont'd II

Solution:

- An action sequence $\bar{a}$ consisting of 0 or more actions is called a *plan* or *solution* to a classical planning problem if and only if:
  - $\bar{a}$ is applicable in s_I.
  - $\bar{a}$ results into a goal state, i.e., $\gamma(\bar{a}, s_I) \supseteq g$.



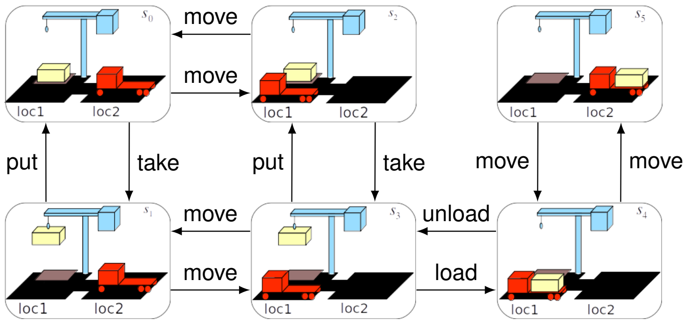
copyright: see slide 6

Solution: **take**, **moveLeft**, **load**, **moveRight**

State Transition Systems

Example

Every classical planning problem is a compact representation of a *state transition system*, i.e., of how states are transformed into each other.



copyright: see slide 6

State Transition Systems

State Transition System

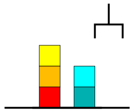
Definition (State Transition System)

A state transition system is a 6-tuple (S, L, c, T, I, G), where

- S is a finite set of states.
- L is a finite set of transition labels.
- $c : L \to \mathbb{R}_0^+$ is a cost function.
- $T \subseteq S \times L \times S$ is the transition relation.
- $I \in S$ is the initial state.
- $G \subseteq S$ is the set of goal states.

State Transition Systems

Size Increase of the State Space in Blocks World



- n blocks, 1 gripper.
- A single action takes a top-most block with the gripper and
  - puts it immediately onto some other top-most block
  - or onto the table.

blocks	states	blocks	states
1	1	10	58941091
2	3	11	824073141
3	13	12	12470162233
4	73	13	202976401213
5	501	14	3535017524403
6	4051	15	65573803186921
7	37633	16	1290434218669921
8	394353	17	26846616451246353
9	4596553	18	588633468315403843

State Transition Systems

Notes

- Planning problems (very compactly!) define state transition systems (cf. Blocks World).
- To solve a planning problem, we construct the underlying state transition system.
- Each node in the search space corresponds to a state and a sequence of actions within the state transition system.

Recap: A^* Search

Tree Search and Graph Search

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    if not in the explored set
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
```

An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Recap: A^* Search

Node Selection Strategy

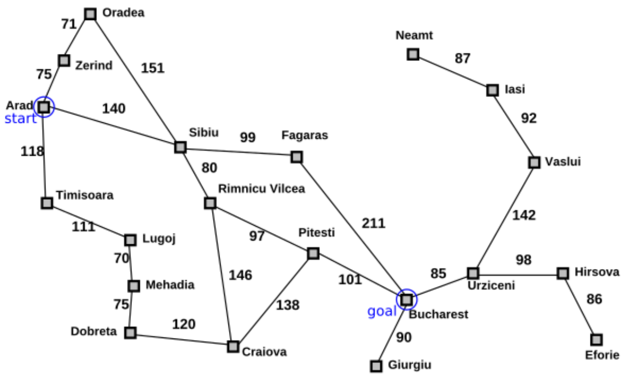
- Tree and graph search can realize various different search strategies such as
  - Uninformed search (like Breadth or Depth First Search, BFS/DFS)
  - Informed search (like A^*)
- We will present the progression planning algorithm as instance of tree search.
- In A^* search, each search node n get's an f value associated:
  - $f(n) = g(n) + h(n)$ with
  - $g(n)$ are the costs of n, e.g., number of actions leading to n
  - $h(n)$ is the heuristic value computed for n. Heuristic means an estimate of the distance from n to a nearest solution.
- Always select a node from the fringe with lowest f value!

→ That's all we need to execute A^* (but we skipped all the theory due to lack of time)

Recap: A^* Search

Example

How to find a(n optimal/good) way from *Arad* to *Bucharest*?

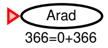


Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Recap: A^* Search

Example, cont'd

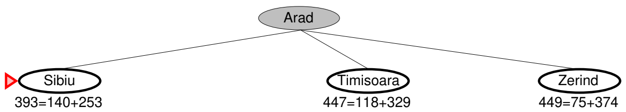


Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance (values see last slide).

Recap: A^* Search

Example, cont'd

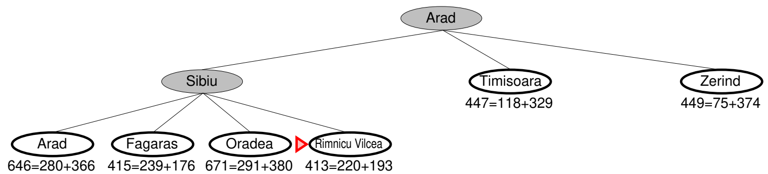


Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance (values see last slide).

Recap: A^* Search

Example, cont'd

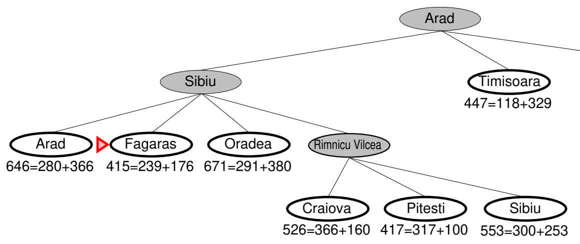


Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance (values see last slide).

Recap: A^* Search

Example, cont'd



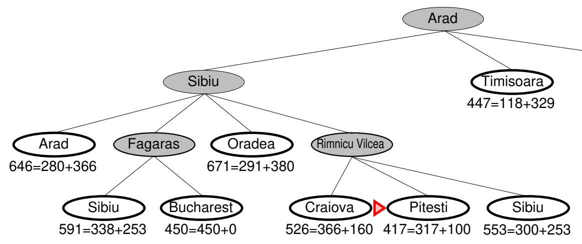
copyright: Title: *Artificial Intelligence: A Modern Approach (Third Edition)*
Url: `https://aima.cs.berkeley.edu/`
Authors: *Stuart Russel* and *Peter Norvig*

Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance (values see last slide).

Recap: A^* Search

Example, cont'd



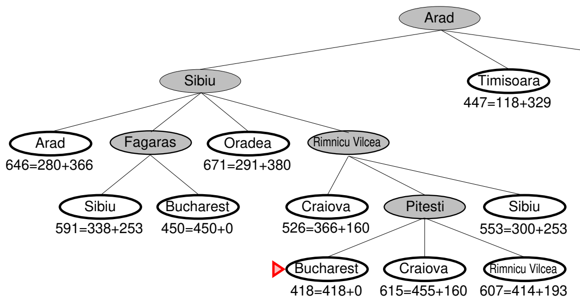
copyright: Title: *Artificial Intelligence: A Modern Approach (Third Edition)*
Url: `https://aima.cs.berkeley.edu/`
Authors: *Stuart Russel* and *Peter Norvig*

Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance (values see last slide).

Recap: A^* Search

Example, cont'd



copyright: Title: *Artificial Intelligence: A Modern Approach (Third Edition)*
Url: `https://aima.cs.berkeley.edu/`
Authors: *Stuart Russel* and *Peter Norvig*

Reminder:

- Always select a node with minimal $f(n) = g(n) + h(n)$.
- Here, h is the linear distance (values see last slide).

Progression Algorithm

Classical Planning as Instance of Tree Search

Algorithm: Progression State-based Search

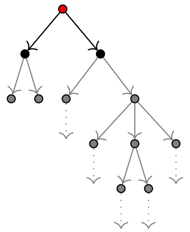
Input: A classical planning problem $\langle V, A, s_I, g \rangle$

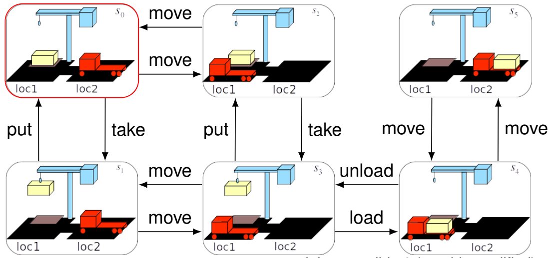
Output: A solution $\bar{a}$ or ***fail*** if none exists

$fringe \leftarrow \{(s_I, \varepsilon)\}$
while $fringe \neq \emptyset$ **do**
  $(s, \bar{a}) \leftarrow nodeSelectAndRemove(fringe)$
  if $s \supseteq g$ **then return** $\bar{a}$
  for $a \in A$ **do**
    if $pre(a) \subseteq s$ **then**
      $s' = (s \setminus del(a)) \cup add(a)$
      $fringe \leftarrow fringe \cup \{(s', \bar{a} \circ a)\}$
return ***fail***

Progression Algorithm

Example





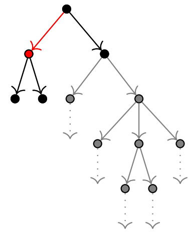
copyright: see slide 6 (graphic modified)

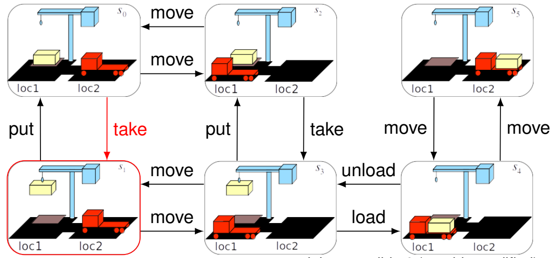
$s_I = \{TruckAtLoc2, CrateAtLoc1\}$

Introduction | AI Planning Problems | AI Search | Planning as Search | Heuristic Planning | Summary

Progression Algorithm

Example





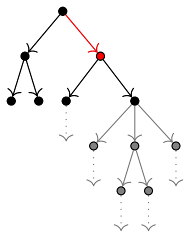
copyright: see slide 6 (graphic modified)

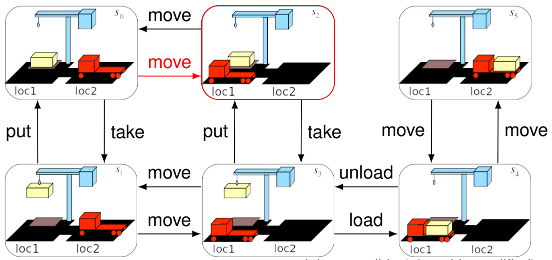
$s_I = \{TruckAtLoc2, CrateAtLoc1\}$ — CrateAtLoc1 → *take* → HoldCrate, ¬CrateAtLoc1 — $\{TruckAtLoc2, HoldCrate\}$

Introduction | AI Planning Problems | AI Search | Planning as Search | Heuristic Planning | Summary

Progression Algorithm

Example





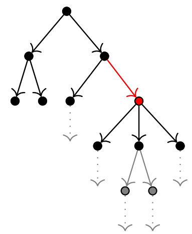
copyright: see slide 6 (graphic modified)

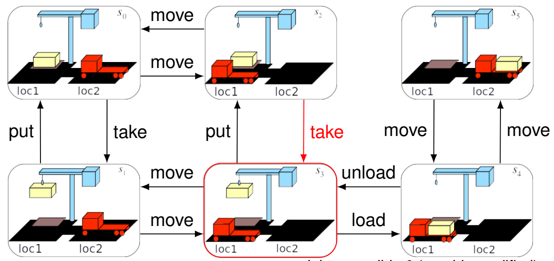
$s_I = \{TruckAtLoc2, CrateAtLoc1\}$ — TruckAtLoc2 → *moveLeft* → TruckAtLoc1, ¬TruckAtLoc2 — $\{TruckAtLoc1, CrateAtLoc1\}$

Introduction | AI Planning Problems | AI Search | Planning as Search | Heuristic Planning | Summary

Progression Algorithm

Example



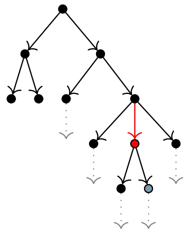


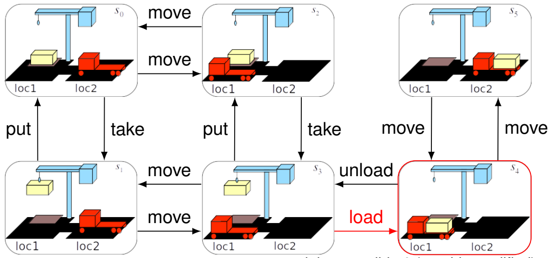
copyright: see slide 6 (graphic modified)

$s_I = \{TruckAtLoc2, CrateAtLoc1\}$ — TruckAtLoc2 → *moveLeft* → TruckAtLoc1, ¬TruckAtLoc2 — $\{TruckAtLoc1, CrateAtLoc1\}$ — CrateAtLoc1 → *take* → HoldCrate, ¬CrateAtLoc1 — $\{TruckAtLoc1, HoldCrate\}$

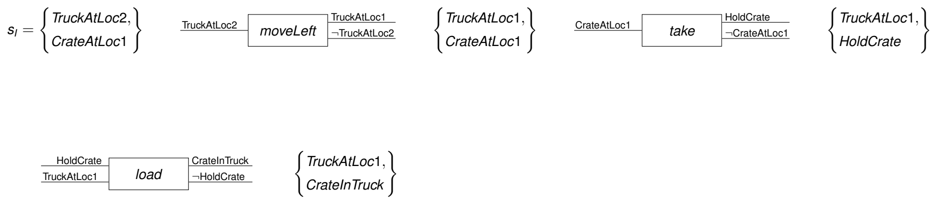
Progression Algorithm

Example



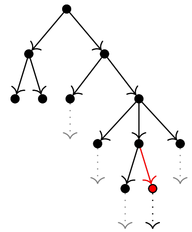


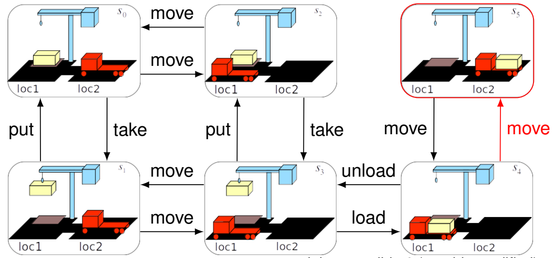
copyright: see slide 6 (graphic modified)



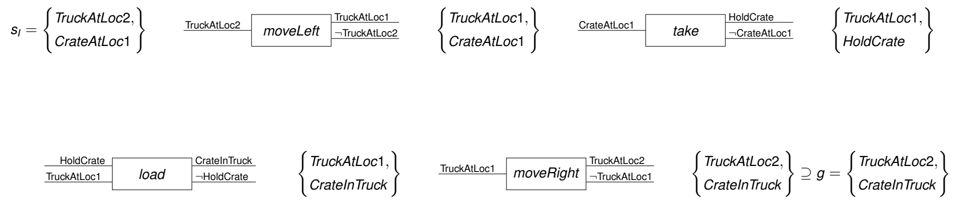
Progression Algorithm

Example





copyright: see slide 6 (graphic modified)



Analyzing the Planning Algorithm

Properties

What are we interested in? Which properties are of interest?

- Does it always *terminate*? If not, can we make it so?
- How can we make the algorithm *more efficient*?
- What's the runtime?
- Is it *correct*, i.e., is every plan it returns an actual solution?
- Is it *complete*, i.e., does it always find a solution if one exists?
- Is it *optimal*, i.e, does it always find the best solution?

Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g\rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

$fringe \leftarrow \{(s_I, \varepsilon)\}$
while $fringe \neq \emptyset$ **do**
  $(s, \bar{a}) \leftarrow nodeSelectAndRemove(fringe)$
  if $s \supseteq g$ **then return** $\bar{a}$
  for $a \in A$ **do**
    if $pre(a) \subseteq s$ **then**
      $s' = (s \setminus del(a)) \cup add(a)$
      $fringe \leftarrow fringe \cup \{(s', \bar{a} \circ a)\}$
return ***fail***

Does it always *terminate*?

- No, due to cycles in state-space.

Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g\rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

$fringe \leftarrow \{(s_I, \varepsilon)\}$
while $fringe \neq \emptyset$ **do**
  $(s, \bar{a}) \leftarrow nodeSelectAndRemove(fringe)$
  if $s \supseteq g$ **then return** $\bar{a}$
  for $a \in A$ **do**
    if $pre(a) \subseteq s$ **then**
      $s' = (s \setminus del(a)) \cup add(a)$
      $fringe \leftarrow fringe \cup \{(s', \bar{a} \circ a)\}$
return ***fail***

How can we make it always terminate?

- Ensure that every search node (state) is explored only *once*.
- Check the current plan length. Discard nodes of a certain length.

Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g\rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

$fringe \leftarrow \{(s_I, \varepsilon)\}$
while $fringe \neq \emptyset$ **do**
  $(s, \bar{a}) \leftarrow nodeSelectAndRemove(fringe)$
  if $s \supseteq g$ **then return** $\bar{a}$
  for $a \in A$ **do**
    if $pre(a) \subseteq s$ **then**
      $s' = (s \setminus del(a)) \cup add(a)$
      $fringe \leftarrow fringe \cup \{(s', \bar{a} \circ a)\}$
return ***fail***

If we made it terminate (by storing visited nodes), what's the runtime?

- Worst case: Until all $|2^V| = 2^{|V|}$ states are generated.

$\rightarrow$ Exponential time and space requirement.

Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g\rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

$fringe \leftarrow \{(s_I, \varepsilon)\}$
while $fringe \neq \emptyset$ **do**
  $(s, \bar{a}) \leftarrow nodeSelectAndRemove(fringe)$
  if $s \supseteq g$ **then return** $\bar{a}$
  for $a \in A$ **do**
    if $pre(a) \subseteq s$ **then**
      $s' = (s \setminus del(a)) \cup add(a)$
      $fringe \leftarrow fringe \cup \{(s', \bar{a} \circ a)\}$
return ***fail***

How can we make an algorithm *more efficient*?

- By including (and studying properties of) heuristics. See later.
- Much more! $\rightarrow$ I offer research projects and PhD theses!

Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g\rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

$fringe \leftarrow \{(s_I, \varepsilon)\}$
while $fringe \neq \emptyset$ **do**
  $(s, \bar{a}) \leftarrow nodeSelectAndRemove(fringe)$
  if $s \supseteq g$ **then return** $\bar{a}$
  for $a \in A$ **do**
    if $pre(a) \subseteq s$ **then**
      $s' = (s \setminus del(a)) \cup add(a)$
      $fringe \leftarrow fringe \cup \{(s', \bar{a} \circ a)\}$
return ***fail***

Is it *correct*, i.e., is every plan it returns an actual solution?

- Yes, which can be proved trivially (show that the properties of returned plans match the solution criteria).

Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g\rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

```
1 fringe ← {(s_I, ε)}
2 while fringe ≠ ∅ do
3     (s, ā) ← nodeSelectAndRemove(fringe)
4     if s ⊇ g then return ā
5     for a ∈ A do
6         if pre(a) ⊆ s then
7             s' = (s \ del(a)) ∪ add(a)
8             fringe ← fringe ∪ {(s', ā ∘ a)}
9 return fail
```

Is it *complete*, i.e., does it always find a solution if one exists?

- This depends on the node selection strategy. And on the fact whether duplicates are considered again.

Analyzing the Planning Algorithm

Properties: Proof Sketches

Algorithm: Progression State-based Search

Input: A classical planning problem $\langle V, A, s_I, g\rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

```
1 fringe ← {(s_I, ε)}
2 while fringe ≠ ∅ do
3     (s, ā) ← nodeSelectAndRemove(fringe)
4     if s ⊇ g then return ā
5     for a ∈ A do
6         if pre(a) ⊆ s then
7             s' = (s \ del(a)) ∪ add(a)
8             fringe ← fringe ∪ {(s', ā ∘ a)}
9 return fail
```

Is it *optimal*, i.e, does it always find the best solution?

- Yes, if used with A^* and an admissible heuristic (see AI lecture/handbook).

Introduction

Search-Guidance in Classical Planning

Problems of progression search:

- Often very huge branching factor (many actions are applicable to a state).
- The search space size increases exponential with search depth (cf. blocks world!)
- Thus, how we implement the node selection (line 3) has a *huge* impact on efficiency! (We rather explore the exact path from the initial state to a goal state rather than the entire search space.)
- Which state to explore next is decided by heuristics!

Introduction

Heuristic Example: Sliding Tile Puzzle

Initial State:

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

→

Goal State:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

How far are we still away?

- Number of misplaced tiles: 13

Heuristic Example: Sliding Tile Puzzle

Initial State:

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

→

Goal State:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

How far are we still away?

- Number of misplaced tiles: 13
- "Distance" (horizontal and vertical distance) per tile to goal position → *Manhatten distance*: 1

Heuristic Example: Sliding Tile Puzzle

Initial State:

2	1	4	8
9	7	11	10
6	5	15	3
13	14	12	

→

Goal State:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

How far are we still away?

- Number of misplaced tiles: 13
- "Distance" (horizontal and vertical distance) per tile to goal position → *Manhatten distance*: 1 + ... + (2+1) + ...

Heuristic Example: Sliding Tile Puzzle

Initial State:

		4	
9			10
	5	15	
13		12	

→

Goal State:

			4
5			
9	10		12
13		15	

How far are we still away?

- Number of misplaced tiles: 13
- "Distance" (horizontal and vertical distance) per tile to goal position → *Manhatten distance*: 1 + ... + (2+1) + ...
- Ignore tiles and solve optimally: 10

Planning Heuristic Construction

How to come up with heuristics in a *domain-independent* way?

- Perform a *problem relaxation*.
- Solve the relaxed problem.
- Use the cost of the solution in the relaxed problem as approximation (i.e., heuristic) of the actual problem.

Example Sliding Tile Puzzle:

- *Number of misplaced tiles.* Relaxation: We can always move tiles to any location, i.e., ignore all preconditions.
- *Manhatten distance.* Relaxation: We can move a tile, even if the neighbor tile is not free, i.e., ignore some preconditions.
- *Ignore tiles.* Some tiles (i.e., state variables) do not exist.

Definitions

Definition (Heuristic)

Given a state transition system $ts = (S, L, c, T, I, G)$, a *heuristic h* is a function $h : S \to \mathbb{R}^+ \cup \{\infty\}$.

Definition (Perfect Heuristic)

A heuristic $h^* : S \to \mathbb{R}^+$ is called *perfect*, if for all states $s \in S$ $h^*(s)$ is the cost of the cheapest transition from s to a goal $s' \in G$. Further, $h^*(s) = \infty$ for all states s for which no goal state can be reached.

Definitions, cont'd I

Definition (Safe Heuristic)

A heuristic h is called *safe*, if for all states $s \in S$ $h(s) = \infty$ implies $h^*(s) = \infty$.

Definition (Goal-aware Heuristic)

A heuristic h is called *goal-aware*, if all goal states, i.e., $s_G \in G$ holds $h(s_G) = 0$.

Definitions, cont'd II

Definition (Admissible Heuristics)

A heuristic h is called *admissible*, if for all states $s \in S$, it holds $h(s) \leq h^*(s)$.

Definition (Dominance)

A heuristic h_1 is said to dominate another heuristic h_2 if for all states $s \in S$, $h_1(s) \geq h_2(s)$.

Analysis of Properties

Why analyzing these properties?

- Because they tell us how "good" (well-informed) they are, and whether one heuristic is better than another.
- The more accurate heuristic estimates, the smaller the explored search space!
- Better-informed heuristics might be harder to compute, so smaller search space does not imply better runtime.
- Every well-informed heuristic should be goal-aware.
- Admissibility guarantees optimality when used with tree search.
- If h_1 and h_2 are admissible, and h_1 dominates h_2, then h_1 is more accurate than h_2 and should create smaller search spaces.

→ We will analyze these properties for heuristics in the assignments.

Simplifying the Planning Problem

One way to design a heuristic is to ignore delete effects:

Definition (Delete-free and -relaxed Planning Problems)

Let $\mathcal{P} = \langle V, A, s_I, g \rangle$ be a STRIPS planning problem.

- It is called delete-free if for all $a \in A$, $del(a) = \emptyset$.
- Its delete-relaxation is the (delete-free) problem $\langle V, A', s_I, g \rangle$, where $A' = \{(pre, add, \emptyset, c) \mid (pre, add, del, c) \in A\}$.

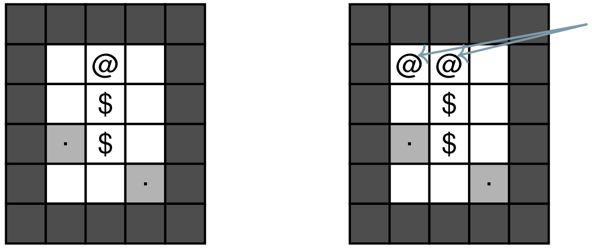
→ $\mathcal{P}^+$ refers to the delete-relaxation of $\mathcal{P}$.

→ h^+ refers to the perfect heuristic (h^*) for $\mathcal{P}^+$.

Definitions, Delete Relaxation

What's the core idea behind delete relaxation?
→ What's true once stays true!

Consider Sokoban after: moving left,



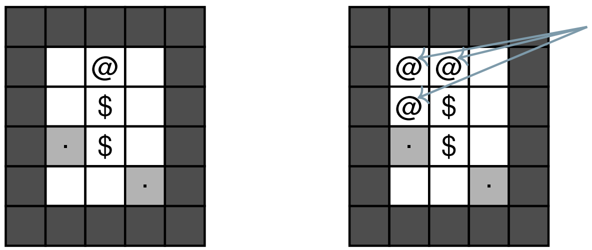
These positions are also *free!* (Since they were free before or have become so.)

@ = the figure $ = a crate · = a goal position

Definitions, Delete Relaxation

What's the core idea behind delete relaxation?
→ What's true once stays true!

Consider Sokoban after: moving left, down,



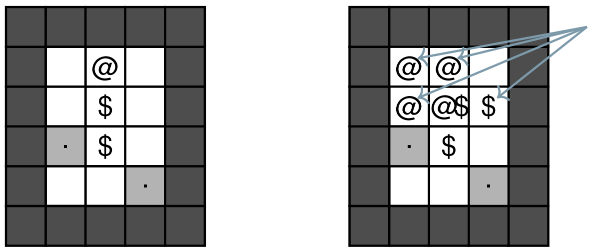
These positions are also *free!* (Since they were free before or have become so.)

@ = the figure $ = a crate · = a goal position

Definitions, Delete Relaxation

What's the core idea behind delete relaxation?
→ What's true once stays true!

Consider Sokoban after: moving left, down, right...



These positions are also *free!* (Since they were free before or have become so.)

@ = the figure $ = a crate · = a goal position

Heuristic(s) based on Delete-Relaxation

Why delete-relaxation and how to exploit it?

- Solving delete-free planning problems can be done in polynomial time!
- (Whereas solving arbitrary planning problems normally requires exponential space and time.)
- Many heuristics are based on delete-relaxation:
  - h^{max} (*shown next!*)
  - h^{Add} (improves h^{max} by incorporating *all* preconditions)
  - h^{FF} (compute a plan for the delete-relaxation)
  - → h^{Add} and h^{FF} might be covered in the assignments.

Relaxed Planning Graph: Definition

Definition (Relaxed Planning Graph)

Let $\langle V, A, s_I, g\rangle$ be a (delete-free) planning problem.

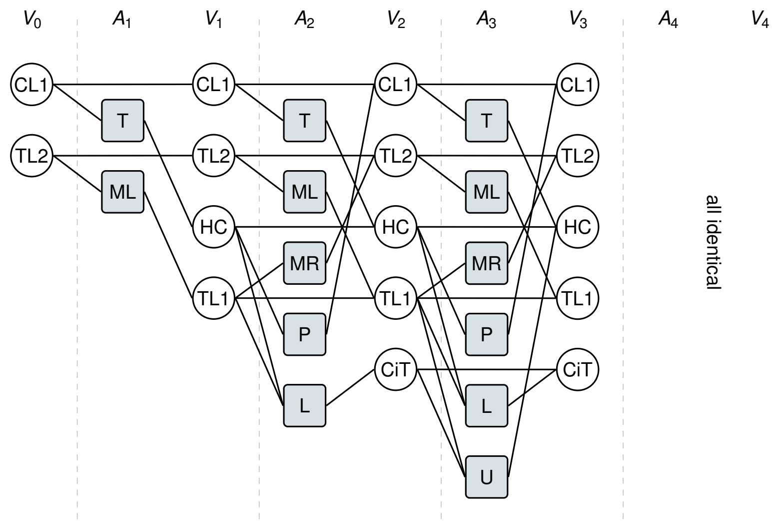
Then, a *relaxed planning graph (rPG)* is a graph $\langle \bar{V}, \bar{A}\rangle$ consisting of:

- $\bar{V} = V^0 \dots V^n$, $V^i \subseteq V$, $0 \leq i \leq n$, a sequence of *variable layers*.
- $\bar{A} = A^1 \dots A^n$, $A^i \subseteq A$, $1 \leq i \leq n$, a sequence of *action layers*.
- $V^0 = s_I$.
- $A^i = \{a \in A \mid pre(a) \subseteq V^{i-1}\}$, $1 \leq i \leq n$.
- $V^i = V^{i-1} \cup \bigcup_{a \in A^i} add(a)$, $1 \leq i \leq n$.
- Choose $n = i$, such that $V^{i-1} = V^i$ holds.

Questions:

- Why is “delete-free” in the problem description put in parentheses?
- Why is n chosen as is? Is there a bound on n?

Relaxed Planning Graph: Example from the Crane in the Harbors Domain



The h^{max} Heuristic

Let $\mathcal{P} = \langle V, A, s_I, g\rangle$ be a classical planning problem and $\mathcal{G} = \langle \bar{V}, \bar{A}\rangle$ its rPG.

- $h^{max}(s)$ returns the first layer number in which all goal variables hold. Meaning: Number of action layers required in $\mathcal{P}^+$ to make the hardest variable in g true (starting in some $s \in S$, e.g., s_I).
- Formally, h^{max} can be calculated as follows:

action vertex The cost of an action vertex $a \in A^i$ is 1 plus the maximum of the predecessor vertex costs.

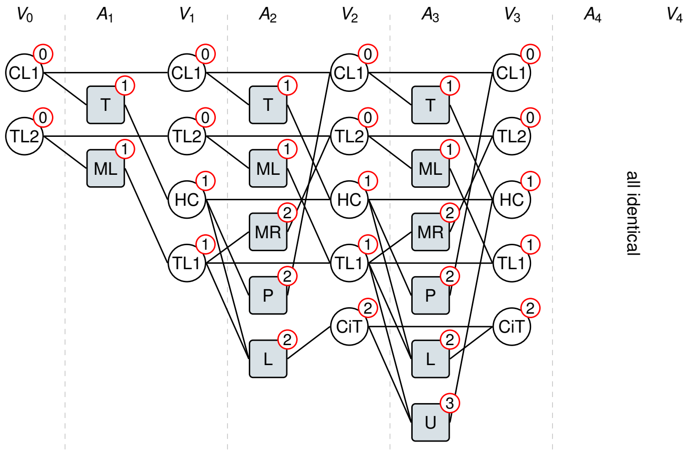
variable vertex
- The cost of a variable vertex v is 0 if $v \in V^0$.
- For all $v \in V^i$, $i > 0$, the cost of v equals the minimum cost of all predecessor vertices (these might be either action or variable vertices).

vertex set For a set of state variables $\bar{v} \subseteq V$, the cost equals the most expensive variable in $\bar{v}$.

heuristic For a state $s \in S$, $h^{max}(s)$ equals the cost of g.

The h^{max} Heuristic: Example

Calculate h^{max} for the Cranes in the Harbor domain.



$s_I = \{CrateAtLoc1, TruckAtLoc2\}$ $g = \{CrateInTruck, TruckAtLoc2\}$

$h^{max}(s_I) = 2$ $h^*(s_I) = 4$ $h^*_{makespan}(s_I) = 3$

The h^{max} Heuristic: Properties

Properties of h^{max} (proofs (trivial) given in lecture talk)

- Perfect?

→ No.

- Safe?

→ Yes.

- Goal-aware?

→ Yes.

- Admissible?

→ Yes.

- “Well-informed”?

→ Not at all. Almost all other heuristics dominate that one.

Theoretical Research Methods ... Illustrated in AI Planning

In this lecture, we ...

- Analyzed (planning) search algorithms, i.e., we ...
  - investigated runtime behavior,
  - investigated space requirements,
  - analyzed heuristics to improve performance, and
  - → learned that heuristics base on special cases that are computationally easier to compute (we aim at poly-computable heuristics, whereas most planning problems – practically – require exponential time and space)
- Possible outlook: computational investigation of (planning) problems and heuristics. We normally investigate
  - complexity of a problem (with/without relaxation)
  - runtime of algorithms/heuristics
  - → Literature and material: see Wattle! (soon)

→ ***Thank you for your attention!***