What do we mean by search?

- **Search** is a very generic term.
- It is every algorithm that tries out various alternatives or even a way to search.
- Here, we mean **classical search** algorithms.
  - **Search nodes** are expanded to generate successor nodes.
  - **Examples:** breadth-first search, A*, hill-climbing, ...
- To be brief, we just say search in the following (not “classical search”).

Do you know this stuff already?

- We assume prior knowledge of basic search algorithms:
  - uninformed vs. informed
  - systematic vs. local
- There will be a small refresher in the next chapter.
- Background: Russell & Norvig, Artificial Intelligence – A Modern Approach, Ch. 3 (all of it), Ch. 4 (local search)
Search in planning

- **search**: one of the big success stories of AI
- many planning algorithms based on classical AI search (we'll see some other algorithms later, though)
- will be the focus of this and the following chapters (the majority of the course)

Satisficing or optimal planning?

Must carefully distinguish two different problems:
- **satisficing planning**: any solution is OK (although shorter solutions typically preferred)
- **optimal planning**: plans must have shortest possible length

Both are often solved by search, but:
- details are very different
- almost no overlap between good techniques for satisficing planning and good techniques for optimal planning
- many problems that are trivial for satisficing planners are impossibly hard for optimal planners

Planning by search

How to apply search to planning? \(\rightarrow\) many choices to make!

**Choice 1: Search direction**
- **progression**: forward from initial state to goal
- **regression**: backward from goal states to initial state
- bidirectional search

**Choice 2: Search space representation**
- search nodes are associated with **states** (~ state-space search)
- search nodes are associated with **sets of states**
How to apply search to planning? \(\rightarrow\) many choices to make!

**Choice 3: Search algorithm**
- uninformed search:
  - depth-first, breadth-first, iterative depth-first, …
- heuristic search (systematic):
  - greedy best-first, A\(^*\), Weighted A\(^*\), IDA\(^*\), …
- heuristic search (local):
  - hill-climbing, simulated annealing, beam search, …

**Choice 4: Search control**
- heuristics for informed search algorithms
- pruning techniques: invariants, symmetry elimination, partial-order reduction, helpful actions pruning, …

Search-based satisficing planners

FF (Hoffmann & Nebel, 2001)
- search direction: forward search
- search space representation: single states
- search algorithm: enforced hill-climbing (informed local)
- heuristic: FF heuristic (inadmissible)
- pruning technique: helpful actions (incomplete)

\(\rightarrow\) one of the best satisficing planners

Search-based optimal planners

Fast Downward Stone Soup (Helmert et al., 2011)
- search direction: forward search
- search space representation: single states
- search algorithm: A\(^*\) (informed systematic)
- heuristic: multiple admissible heuristics combined into a heuristic portfolio (LM-cut, M&S, blind, …)
- pruning technique: none

\(\rightarrow\) one of the best optimal planners
Our plan for the next lectures

Choices to make:
1. search direction: progression/regression/both
   ⇝ this chapter
2. search space representation: states/sets of states
   ⇝ this chapter
3. search algorithm: uninformed/heuristic; systematic/local
   ⇝ next chapter
4. search control: heuristics, pruning techniques
   ⇝ following chapters


2 Progression

Overview
Example

Planning by forward search: progression

Progression: Computing the successor state \( \text{app}_o(s) \) of a state \( s \) with respect to an operator \( o \).

Progression planners find solutions by forward search:
- start from initial state
- iteratively pick a previously generated state and progress it through an operator, generating a new state
- solution found when a goal state generated

pro: very easy and efficient to implement

October 31st, 2014 B. Nebel, R. Mattmüller – AI Planning 14 / 49

Search space representation in progression planners

Two alternative search spaces for progression planners:
1. search nodes correspond to states
   - when the same state is generated along different paths, it is not considered again (duplicate detection)
   - pro: save time to consider same state again
   - con: memory intensive (must maintain closed list)
2. search nodes correspond to operator sequences
   - different operator sequences may lead to identical states (transpositions); search does not notice this
   - pro: can be very memory-efficient
   - con: much wasted work (often exponentially slower)

⇝ first alternative usually preferable in planning (unlike many classical search benchmarks like 15-puzzle)

October 31st, 2014 B. Nebel, R. Mattmüller – AI Planning 15 / 49
Progression planning example (depth-first search)

Example where search nodes correspond to operator sequences
(no duplicate detection)

$s_0 \rightarrow \ldots \rightarrow S^*$

Progression planning example (depth-first search)

Example where search nodes correspond to operator sequences
(no duplicate detection)

$s_0 \rightarrow \ldots \rightarrow S^*$
Progression planning example (depth-first search)

Example where search nodes correspond to operator sequences (no duplicate detection)

$S_0 \rightarrow \ldots \rightarrow S_*$

October 31st, 2014 B. Nebel, R. Mattmüller – AI Planning 16 / 49
Progression planning example (depth-first search)

Example where search nodes correspond to operator sequences (no duplicate detection)

3 Regression

- Overview
- Example
- Regression for STRIPS tasks
- Regression for general planning tasks
- Practical issues

Forward search vs. backward search

Going through a transition graph in forward and backward directions is not symmetric:
- forward search starts from a single initial state;
- backward search starts from a set of goal states.
- when applying an operator $o$ in a state $s$ in forward direction, there is a unique successor state $s'$;
- if we applied operator $o$ to end up in state $s'$, there can be several possible predecessor states $s$.
- most natural representation for backward search in planning associates sets of states with search nodes.
Planning by backward search: regression

**Regression**: Computing the possible predecessor states $\text{regr}_o(G)$ of a set of states $G$ with respect to the last operator $o$ that was applied.

**Regression planners** find solutions by backward search:
- start from set of goal states
- iteratively pick a previously generated state set and regress it through an operator, generating a new state set
- solution found when a generated state set includes the initial state

**Pro**: can handle many states simultaneously  
**Con**: basic operations complicated and expensive

Search space representation in regression planners

identify state sets with logical formulae (again):
- search nodes correspond to state sets
- each state set is represented by a logical formula: $\varphi$ represents $\{ s \in S \mid s \models \varphi \}$
- many basic search operations like detecting duplicates are NP-hard or coNP-hard

Regression planning example (depth-first search)

$$
I \quad \text{γ} \quad \text{ϕ}_1 \quad \text{ϕ}_1 = \text{regr}_\rightarrow (\gamma) \quad \text{ϕ}_2 \quad \text{ϕ}_2 = \text{regr}_\rightarrow (\text{ϕ}_1) \quad \text{ϕ}_3 \quad \text{ϕ}_3 = \text{regr}_\rightarrow (\text{ϕ}_2), \quad I \models \text{ϕ}_3
$$
Regression planning example (depth-first search)

\[ \varphi_1 = \text{regr} \rightarrow (\gamma) \]

\[ \varphi_2 \rightarrow \varphi_1 \rightarrow \gamma \]

October 31st, 2014 B. Nebel, R. Mattmüller – AI Planning

Regression for STRIPS planning tasks

Definition (STRIPS planning task)
A planning task is a STRIPS planning task if all operators are STRIPS operators and the goal is a conjunction of atoms.

Regression for STRIPS planning tasks is very simple:
- Goals are conjunctions of atoms \( a_1 \land \cdots \land a_n \).
- First step: Choose an operator that makes none of \( a_1, \ldots, a_n \) false.
- Second step: Remove goal atoms achieved by the operator (if any) and add its preconditions.

Outcome of regression is again conjunction of atoms.

Optimization: only consider operators making some \( a_i \) true.
**Definition (STRIPS regression)**

Let $\varphi = \varphi_1 \land \cdots \land \varphi_n$ be a conjunction of atoms, and let $\alpha = \langle \chi, e \rangle$ be a STRIPS operator which adds the atoms $a_1, \ldots, a_k$ and deletes the atoms $d_1, \ldots, d_i$.

The STRIPS regression of $\varphi$ with respect to $\alpha$ is

$$\text{sregr}_\alpha(\varphi) := \begin{cases} \perp & \text{if } a_i = d_j \text{ for some } i, j \\ \perp & \text{if } \varphi_i = d_j \text{ for some } i, j \\ \chi \land (\{\varphi_1, \ldots, \varphi_n\} \setminus \{a_1, \ldots, a_k\}) & \text{otherwise} \end{cases}$$

**Note:** $\text{sregr}_\alpha(\varphi)$ is again a conjunction of atoms, or $\perp$.

---

**Regression for general planning tasks**

- With disjunctions and conditional effects, things become more tricky. How to regress $a \lor (b \land c)$ with respect to $\langle q, d \triangleright b \rangle$?
- The story about goals and subgoals and fulfilling subgoals, as in the STRIPS case, is no longer useful.
- We present a general method for doing regression for any formula and any operator.
- Now we extensively use the idea of representing sets of states as formulae.

---

**Effect preconditions**

**Definition (effect precondition)**

The effect precondition $\text{EPC}_l(e)$ for literal $l$ and effect $e$ is defined as follows:

$$\text{EPC}_l(l) = T$$

$$\text{EPC}_l(l') = \perp \text{ if } l \neq l' \text{ (for literals } l')$$

$$\text{EPC}_l(e_1 \land \cdots \land e_n) = \text{EPC}_l(e_1) \lor \cdots \lor \text{EPC}_l(e_n)$$

$$\text{EPC}_l(\chi \triangleright e) = \text{EPC}_l(e) \land \chi$$

**Intuition:** $\text{EPC}_l(e)$ describes the situations in which effect $e$ causes literal $l$ to become true.
Effect precondition examples

Example

\[ EPC_a(b \land c) = \bot \lor \bot \equiv \bot \]
\[ EPC_a(a \land (b \triangleright a)) = T \lor (T \land b) \equiv T \]
\[ EPC_a((c \triangleright a) \land (b \triangleright a)) = (T \land c) \lor (T \land b) \equiv c \lor b \]

Effect preconditions: connection to change sets

Proof (ctd.)

Inductive case 1, \( e = e_1 \land \cdots \land e_n \):
\[ l \in [e]_s \iff l \in [e_1]_s \cup \cdots \cup [e_n]_s \]  
(Def \( [e_1 \land \cdots \land e_n]_s \))
\[ \quad \text{if } l \in [e'_s] \text{ for some } e' \in \{e_1, \ldots, e_n\} \]
\[ \quad \text{if } s \models EPC_i(e') \text{ for some } e' \in \{e_1, \ldots, e_n\} \]  
(IH)
\[ \quad \text{if } s \models EPC_i(e_1) \lor \cdots \lor EPC_i(e_n) \]  
(Def \( EPC \))

Inductive case 2, \( e = \chi \triangleright e' \):
\[ l \in [\chi \triangleright e']_s \iff l \in [e'_s] \text{ and } s \models \chi \]  
(Def \( [\chi \triangleright e']_s \))
\[ \quad \text{if } s \models EPC_i(e') \text{ and } s \models \chi \]  
(IH)
\[ \quad \text{if } s \models EPC_i(e') \land \chi \]  
(Def \( EPC \))

Remark: \( EPC \) vs. effect normal form

Notice that in terms of \( EPC_a(e) \), any operator \( \langle \chi, e \rangle \) can be expressed in effect normal form as
\[ \left\langle \chi, \bigwedge_{a \in A} \left( (EPC_a(e) \triangleright a) \land (EPC_{\neg a}(e) \triangleright \neg a) \right) \right\rangle, \]

where \( A \) is the set of all state variables.
Regressing state variables

The formula \( \text{EPC}_a(e) \lor (a \land \neg \text{EPC}_{\neg a}(e)) \) expresses the value of state variable \( a \in A \) after applying \( o \) in terms of values of state variables before applying \( o \).

Either:
1. \( a \) became true, or
2. \( a \) was true before and it did not become false.

Regressing state variables: examples

Example

Let \( e = (b \triangleright a) \land (c \triangleright \neg a) \land b \land \neg d \).

<table>
<thead>
<tr>
<th>Variable</th>
<th>( \text{EPC}<em>x(e) \lor (x \land \neg \text{EPC}</em>{\neg x}(e)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( b \lor (a \land \neg c) )</td>
</tr>
<tr>
<td>( b )</td>
<td>( \top \lor (b \land \neg \bot) \equiv \top )</td>
</tr>
<tr>
<td>( c )</td>
<td>( \bot \lor (c \land \neg \bot) \equiv c )</td>
</tr>
<tr>
<td>( d )</td>
<td>( \bot \lor (d \land \neg \top) \equiv \bot )</td>
</tr>
</tbody>
</table>

Regressing state variables: correctness

Lemma (B)

Let \( a \) be a state variable, \( o = \langle \chi, e \rangle \) an operator, \( s \) a state, and \( s' = \text{app}_o(s) \).

Then \( s \models \text{EPC}_a(e) \lor (a \land \neg \text{EPC}_{\neg a}(e)) \) if and only if \( s' \models a \).

Proof.

\((\Rightarrow)\): Assume \( s \models \text{EPC}_a(e) \lor (a \land \neg \text{EPC}_{\neg a}(e)) \).

Do a case analysis on the two disjuncts.

1. Assume that \( s \models \text{EPC}_a(e) \).

   By Lemma A, we have \( a \in [e]_s \) and hence \( s' \models a \).

2. Assume that \( s \models a \land \neg \text{EPC}_{\neg a}(e) \).

   By Lemma A, we have \( \neg a \notin [e]_s \). Hence \( a \) remains true in \( s' \).

\((\Leftarrow)\): Assume \( s' \models a \).

So assume \( s \not\models \text{EPC}_a(e) \lor (a \land \neg \text{EPC}_{\neg a}(e)) \).

Then \( s \not\models \neg \text{EPC}_a(e) \land (\neg a \lor \text{EPC}_{\neg a}(e)) \) (de Morgan).

Case distinction: \( a \) is true or \( a \) is false in \( s' \).

1. Assume that \( s \models a \). Now \( s \models \text{EPC}_{\neg a}(e) \) because \( s \models \neg a \lor \text{EPC}_{\neg a}(e) \).

   Hence by Lemma A \( \neg a \in [e]_s \) and we get \( s' \not\models a \).

2. Assume that \( s \not\models a \). Because \( s \not\models \text{EPC}_a(e) \), by Lemma A we have \( a \notin [e]_s \) and hence \( s' \not\models a \).

Therefore in both cases \( s' \not\models a \).
Regression: general definition

We base the definition of regression on formulae $EPC_i(e)$.

**Definition (general regression)**

Let $\phi$ be a propositional formula and $o = \langle \chi, e \rangle$ an operator. The regression of $\phi$ with respect to $o$ is

$$\text{regr}_o(\phi) = \chi \land \phi \land \kappa$$

where

- $\phi$ is obtained from $\phi$ by replacing each $a \in A$ by $EPC_a(e) \lor (a \land \neg EPC_{-a}(e))$, and
- $\kappa = \bigwedge_{a \in A} \neg (EPC_a(e) \land EPC_{-a}(e))$.

The formula $\kappa$ expresses that operators are only applicable in states where their change sets are consistent.

Regression examples

- $\text{regr}_{(a,b)}(b) \equiv a \land (\top \lor (b \land \neg \bot)) \land \top \equiv a$
- $\text{regr}_{(a,b)}(c \land d) \equiv a \land (\top \lor (b \land \neg \bot)) \land (\bot \lor (c \land \neg \bot)) \land \top \equiv a \land (c \land \bot)$
- $\text{regr}_{(a,c \lor b)}(b) \equiv a \land (c \lor (b \land \neg \bot)) \land \top \equiv a \land (c \lor b)$
- $\text{regr}_{(a,c \lor b),(b \lor \neg b)}(b) \equiv a \land (c \lor (b \land \neg \bot)) \land \neg (c \land \bot) \equiv a \land (c \lor b) \land (c \lor \neg \bot) \land \neg (c \lor \neg \bot) \equiv a \land (c \lor b) \land \neg \bot$

General regression: correctness

**Theorem (correctness of $\text{regr}_o(\phi)$)**

Let $\phi$ be a formula, $o$ an operator and $s$ a state. Then $s \vDash \text{regr}_o(\phi)$ iff $o$ is applicable in $s$ and $\text{app}_o(s) \vDash \phi$.

**Proof.**

Let $o = \langle \chi, e \rangle$. Recall that $\text{regr}_o(\phi) = \chi \land \phi \land \kappa$, where $\phi$ and $\kappa$ are as defined previously.

If $o$ is inapplicable in $s$, then $s \not\vDash \chi \land \kappa$, both sides of the "iff" condition are false, and we are done. Hence, we only further consider states $s$ where $o$ is applicable. Let $s' := \text{app}_o(s)$.

We know that $s \vDash \chi \land \kappa$ (because $o$ is applicable), so the "iff" condition we need to prove simplifies to:

$$s \vDash \phi \iff s' \vDash \phi.$$
The proof is by structural induction on \( \phi \).

To show: for all formulae \( s \models \psi \) iff \( s' \models \psi \), where \( \psi_1 \) is \( \psi \) with every \( a \in A \) replaced by \( EPC_a(e) \lor (a \land \neg EPC_{a}(e)) \).

The proof is by structural induction on \( \psi \).

**Inductive case 1** \( \psi = \neg \psi' \):

\( s \models \psi_1 \) iff \( s \models \neg \psi' \), iff \( s \models \neg \psi' \lor s \models \psi' \), iff \( (IH) s' \models \psi' \land s' \models \psi' \)

**Inductive case 2** \( \psi = \psi' \lor \psi'' \):

\( s \models \psi_1 \) iff \( s \models \psi' \lor \psi'' \), iff \( s \models \psi' \lor \psi'' \land s \models \psi'' \)

**Inductive case 3** \( \psi = \psi' \land \psi'' \):

Very similar to inductive case 2, just with \( \land \) instead of \( \lor \) and “and” instead of “or”.

Both of these problems are \( \text{NP-hard} \).
Restricting formula growth in search trees

Problem: very big formulae obtained by regression
Cause: disjunctivity in the (NNF) formulae
(formulae without disjunctions easily convertible to small formulae $l_1 \land \cdots \land l_n$ where $l_i$ are literals and $n$ is at most the number of state variables.)

Idea: handle disjunctivity when generating search trees

Unrestricted regression: search tree example

Goal $\gamma = a \land b$, initial state $I = \{ a \mapsto \neg 0, b \mapsto \neg 0, c \mapsto 0 \}$.

Unrestricted regression: do not treat disjunctions specially

Goal $\gamma = a \land b$, initial state $I = \{ a \mapsto \neg 0, b \mapsto \neg 0, c \mapsto 0 \}$.

Full splitting: search tree example

Full splitting: always remove all disjunctivity

Goal $\gamma = a \land b$, initial state $I = \{ a \mapsto \neg 0, b \mapsto \neg 0, c \mapsto 0 \}$.

$\neg (\neg c \lor a) \land b$ in DNF: $\neg (c \land b) \lor (a \land b)$

$\leadsto$ split into $\neg c \land b$ and $a \land b$

General splitting strategies

Alternatives:
1. Do nothing (unrestricted regression).
2. Always eliminate all disjunctivity (full splitting).
3. Reduce disjunctivity if formula becomes too big.

Discussion:
- With unrestricted regression the formulae may have size that is exponential in the number of state variables.
- With full splitting search tree can be exponentially bigger than without splitting.
- The third option lies between these two extremes.
(Classical) search is a very important planning approach. Search-based planning algorithms differ along many dimensions, including

- search direction (forward, backward)
- what each search node represents
  (a state, a set of states, an operator sequence)
- Progression search proceeds forwards from the initial state.
  - If we use duplicate detection, each search node corresponds to a unique state.
  - If we do not use duplicate detection, each search node corresponds to a unique operator sequence.

Regression search proceeds backwards from the goal.

- Each search node corresponds to a set of states represented by a formula.
- Regression is simple for STRIPS operators.
- The theory for general regression is more complex.
- When applying regression in practice, additional considerations such as when and how to perform splitting come into play.