What do we mean by search?

- **Search** is a very generic term.
  - Every algorithm that tries out various alternatives can be said to “search” in some way.
- 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? — 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

---

**Planning by search**

How to apply search to planning? — many choices to make!

**Choice 2: Search space representation**
- search nodes are associated with states (~state-space search)
- search nodes are associated with sets of states
Speech: Planning by search

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

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

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

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, \ldots)
- 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

Progression

Planning by forward search: progression

Progression: Computing the successor state $app_o(s)$ 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

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)
Progression planning example (depth-first search)

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

October 27th, 2017  B. Nebel, R. Mattmüller – AI Planning 14 / 46
Progression planning example (depth-first search)

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

$s_0$
Progression planning example (depth-first search)

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

\[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \]

October 27th, 2017 B. Nebel, R. Mattmüller – AI Planning 14 / 46

Regression

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**.
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

Regression planning example (depth-first search)

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

$$\varphi_1 = \text{regr} \rightarrow (\gamma)$$

$$\varphi_1 \rightarrow \gamma$$

Regression planning example (depth-first search)

$$\varphi_1 = \text{regr} \rightarrow (\gamma)$$
$$\varphi_2 = \text{regr} \rightarrow (\varphi_1)$$

$$\varphi_2 \rightarrow \varphi_1 \rightarrow \gamma$$

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 $$a_1 \wedge \cdots \wedge a_n$$.
- 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
**STRIPS regression**

**Definition (STRIPS regression)**

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

The STRIPS regression of \( \varphi \) with respect to \( o \) is

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

**Note:** \( \text{sregr}_o(\varphi) \) is again a conjunction of atoms, or \( \bot \).

---

**Regession 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 \( EPC_l(e) \) for literal \( l \) and effect \( e \) is defined as follows:

\[
\begin{align*}
EPC_l(l) &= \top \\
EPC_l(l') &= \bot \text{ if } l \neq l' \text{ (for literals } l') \\
EPC_l(e_1 \land \cdots \land e_n) &= EPC_l(e_1) \lor \cdots \lor EPC_l(e_n) \\
EPC_l(\chi \triangleright e) &= EPC_l(e) \land \chi
\end{align*}
\]

**Intuition:** \( EPC_l(e) \) describes the situations in which effect \( e \) causes literal \( l \) to become true.
Effect precondition examples

Example

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

Effect preconditions: connection to change sets

Lemma (A)

Let \( s \) be a state, \( l \) a literal and \( e \) an effect. Then \( l \in [e]_s \) if and only if \( s \models EPC_l(e) \).

Proof.

Induction on the structure of the effect \( e \).
Base case 1, \( e = l \): \( l \in [l]_s \) by definition, and
\( s \models EPC_l(l) = T \) by definition. Both sides of the equivalence are true.
Base case 2, \( e = l' \) for some literal \( l' \neq l \): \( l' \notin [l']_s \) by definition, and \( s \not\models EPC_l(l') = \bot \) by definition. Both sides are false.

Proof (ctd.)

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

Inductive case 2, \( e = \chi \lor e' \):
\[ l \in [\chi \lor e']_s \iff l \in [\chi]_s \text{ and } s \models \chi \quad \text{(Def [\chi \lor e']_s)} \]
if \( s \models EPC_l(e') \) and \( s \models \chi \) \quad \text{(IH)}
if \( s \models EPC_l(e') \land \chi \)
if \( s \not\models EPC_l(\chi \lor e') \). \quad \text{(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
\[ \langle \chi, \bigwedge_{a \in A} ((EPC_a(e) \lor a) \land (EPC_{-a}(e) \lor \neg a)) \rangle, \]
where \( A \) is the set of all state variables.
Regressing state variables

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

Either:
- $a$ became true, or
- $a$ 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>$EPC_x(e) \lor (x \land \neg EPC_{\neg x}(e))$</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, $o = (\chi, e)$ an operator, $s$ a state, and $s' = \text{app}_o(s)$.

Then $s \models EPC_a(e) \lor (a \land \neg EPC_{\neg a}(e))$ if and only if $s' \models a$.

Proof.

($\Rightarrow$): Assume $s \models EPC_a(e) \lor (a \land \neg EPC_{\neg a}(e))$.

Do a case analysis on the two disjuncts.

1. Assume that $s \not\models EPC_a(e)$. By Lemma A, we have $a \not\in [e]_s$ and hence $s' \not\models a$.
2. Assume that $s \models a \land \neg EPC_{\neg a}(e)$. By Lemma A, we have $\neg a \not\in [e]_s$. Hence $a$ remains true in $s'$.

($\Leftarrow$): We showed that if the formula is true in $s$, then $a$ is true in $s'$. For the second part, we show that if the formula is false in $s$, then $a$ is false in $s'$.

- So assume $s \not\models EPC_a(e) \lor (a \land \neg EPC_{\neg a}(e))$.
- Then $s \not\models \neg EPC_a(e) \land (\neg a \lor EPC_{\neg a}(e))$ (de Morgan).
- Case distinction: $a$ is true or $a$ is false in $s$.
  1. Assume that $s \models a$. Now $s \models EPC_{\neg a}(e)$ because $s \models \neg a \lor EPC_{\neg a}(e)$.
     Hence by Lemma A $\neg a \not\in [e]_s$ and we get $s' \not\models a$.
  2. Assume that $s \not\models a$. Because $s \not\models EPC_a(e)$, by Lemma A we get $a \not\in [e]_s$ 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 \( EPC_t(e) \).

**Definition (general regression)**

Let \( \varphi \) be a propositional formula and \( o = (\chi, e) \) an operator. The regression of \( \varphi \) with respect to \( o \) is

\[
\text{regr}_o(\varphi) = \chi \land \varphi_t \land \kappa
\]

where

- \( \varphi_t \) is obtained from \( \varphi \) by replacing each \( a \in A \) by \( EPC_a(e) \lor (a \land \neg EPC_{-a}(e)) \), and

- \( \kappa = \land_{a \in A} \neg(EPC_a(e) \land EPC_{-a}(e)) \).

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

Regression examples

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

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

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

**Proof.**

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

If \( o \) is inapplicable in \( s \), then \( s \not\models \chi \land \kappa \), 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 \( s \models \chi \land \kappa \) (because \( o \) is applicable), so the "iff" condition we need to prove simplifies to:

\[
s \models \varphi_t \iff s' \models \varphi.
\]
The proof is by structural induction on $\psi$ with every $a \in A$ replaced by $EPC_a(e) \lor (a \land \neg EPC_{\neg a}(e))$.

The proof is by structural induction on $\psi$.

Induction hypothesis $s \models \psi_i$ if and only if $s' \models \psi$.

Base cases 1 & 2 $\psi = \top$ or $\psi = \bot$: trivial, as $\psi_i = \psi$.

Base case 3 $\psi = a$ for some $a \in A$:
Then $\psi_i = EPC_a(e) \lor (a \land \neg EPC_{\neg a}(e))$.

By Lemma B, $s \models \psi_i$ iff $s' \models \psi$.

Both of these problems are NP-hard.

Formula growth

The formula $\text{regr}_{o_n}(\text{regr}_{o_{n-1}}(\cdots \text{regr}_{o_2}(\text{regr}_{o_1}(\varphi))))$ may have size $O(|\varphi||o_1||o_2|\cdots||o_{n-1}||o_n|)$, i.e., the product of the sizes of $\varphi$ and the operators.

$\sim$ worst-case exponential size $O(m^n)$

Logical simplifications

- $\bot \land \varphi \equiv \bot$, $\top \land \varphi \equiv \varphi$, $\bot \lor \varphi \equiv \top$, $\varphi \lor \bot \equiv \varphi$
- $\varphi \lor \bot \equiv \top$, $\neg a \lor \varphi \equiv \neg a \lor \varphi[^T/a]$, $\neg a \lor \varphi \equiv \neg a \lor \varphi[^T/a]$
- $\neg a \lor \varphi \equiv \neg a \lor \varphi[^T/a]$
- idempotency, absorption, commutativity, associativity, \ldots

The following two tests are useful when performing regression searches to avoid exploring unpromising branches:

- Test that $\text{regr}_{o_1}(\varphi)$ does not represent the empty set (which would mean that search is in a dead end).
  For example, $\text{regr}_{o_1}(a \land \bot \equiv \bot)$.
- Test that $\text{regr}_{o_2}(\varphi)$ does not represent a subset of $\varphi$ (which would make the problem harder than before).
  For example, $\text{regr}_{o_2}(a \land b \equiv a \land b)$.
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

Unrestricted regression: do not treat disjunctions specially
Goal: \( \gamma = a \land b \), initial state \( I = \{ a \mapsto → 0, b \mapsto → 0, c \mapsto → 0 \} \).

Full splitting: search tree example

Full splitting: always remove all disjunctivity
Goal: \( \gamma = a \land b \), initial state \( I = \{ a \mapsto → 0, b \mapsto → 0, c \mapsto → 0 \} \).

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.
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.