Computational Complexity and Expressive Power in Classical AI Planning

Part I: Introduction and Basics

Bernhard Nebel
Albert-Ludwigs-Universität Freiburg
Department of Computer Science

Overview

1. Introduction
   (a) Motivation
   (b) Classical Planning: STRIPS & Variants
   (c) Complexity Theory: Basic Notions, NP, PSPACE & EXPSPACE
2. Computational Complexity of Classical Planning
   (a) STRIPS With First-Order Terms and DATALOG-STRIPS
   (b) Propositional STRIPS & Restrictions
   (c) Domain-Dependent Planning
3. Expressive Power of Planning Formalisms
   (a) How to Compare Expressive Power?
   (b) The Compilability Approach
   (c) Conditional Effects & General Preconditions

Motivation: Why Complexity Analyses?

- Planning is so so difficult that it does not make much sense to to try to design algorithms that are efficient in general
- The worst case complexity does not matter – write fast algorithms for the problems that come up

$\rightsquigarrow$ All true, but . . .

- Try to avoid backtracking/search algorithms if problems are simple
- Find out whether methods for NP-complete problems are applicable (local search, backtracking, . . .)
- Determine complexity of supposedly easier sub-problems that can be used in solving the original problem (either as a heuristic or as a sub-problem in backtracking search)

Motivation: Why Analyzing the Expressive Power?

- Expressive power is the motivation for designing new planning languages
- Often there is the question: Syntactic sugar or essential feature?
  - "Compiling away" or change planning algorithm?
- Claims & Conjectures such as
  - STRIPS operators with preconditions in *conjunctive normal form* are more powerful than ordinary STRIPS operators [Bäckström]
  - General Boolean preconditions are *an essential prerequisite* for conditional effects [Anderson et al]

Prerequisites

- Some basic knowledge in Theoretical Computer Science, e.g., **algorithm**, **formal language**, **Turing machine**, **complexity classes**, **polynomial reduction**
- Some basic knowledge in formal logic: **propositional logic**, **conjunctive normal form**, **first-order logic**, **substitution**
- Some basic knowledge in classical STRIPS planning: **operator**, **state**, **fluent**, **goal**

Classical Planning: STRIPS

- $\mathcal{S}$ is a *first-order signature* and $\Sigma_{\mathcal{S}}$ denotes the set of *ground atoms* over the signature (also called **facts** or **fluents**).
- $\Sigma_{\mathcal{S},\mathbf{V}}$ is the set of atoms over $\mathcal{S}$ using variable symbols from the set of variables $\mathbf{V}$.
- A **first-order STRIPS state** S is a subset of $\Sigma_{\mathcal{S}}$ denoting a *complete theory* or *model*.
- A **planning task** (or **planning instance**) is a 4-tuple $\Pi = \langle \mathcal{S}, \mathbf{O}, \mathbf{I}, \mathbf{G} \rangle$, where
  - $\mathbf{O}$ is a set of **operator schemata** (explanation in a minute)
  - $\mathbf{I} \subseteq \Sigma_{\mathcal{S}}$ is the **initial state**
  - $\mathbf{G} \subseteq \Sigma_{\mathcal{S}}$ is the **goal specification**

Operator Schemata, Operators & State Change

- **Operator schemata:**

  $$o = \langle \mathit{para}, \mathit{pre}, \mathit{eff} \rangle,$$

  with $\mathit{para} \subseteq \mathbf{V}$, $\mathit{pre} \subseteq \Sigma_{\mathcal{S},\mathbf{V}}$, $\mathit{eff} \subseteq \Sigma_{\mathcal{S},\mathbf{V}} \cup \neg\Sigma_{\mathcal{S},\mathbf{V}}$ (element-wise negation) and all variables in pre and eff are listed in para.

  Also: $\mathit{pre}(o)$, $\mathit{eff}(o)$.

  eff^{+} = positive effect literals, eff^{-} = negative effect literals
- **Operator** or **action**: Operator schema with empty parameter list (*instantiated schema!*)
- **State change** induced by operator:

$$\mathit{App}(S,o) = \begin{cases} S \cup \mathit{eff}^{+}(o) - \neg\mathit{eff}^{-}(o) & \text{if } \mathit{pre}(o) \subseteq S \text{ and } o \text{ is consistent} \\ \text{undefined} & \text{otherwise} \end{cases}$$

A Small Example

State:

$$S = \left\{ \begin{array}{lll} \text{broken}(\text{b}_1), & \text{in-lamp}(\text{b}_1,\text{l}_1), & \text{at}(\text{l}_1,\text{p}_1) \\ \text{ok}(\text{b}_2), & \text{at}(\text{b}_2,\text{p}_2) & \text{r-at}(\text{p}_3) \end{array} \right\}$$

Operator Schemata:

$$\begin{array}{rcl}
\text{goto}(x,y) & = & \langle\langle x,y\rangle, \{\text{r-at}(x)\}, \{\text{r-at}(y), \neg\text{r-at}(x)\}\rangle, \\
\text{grab}(x,y) & = & \langle\langle x,y\rangle, \\
 & & \{\text{r-at}(y), \text{at}(x,y)\}, \\
 & & \{\text{holding}(x), \neg\text{at}(x,y)\}\rangle, \\
\text{put-down}(x,y) & = & \langle\langle x,y\rangle, \\
 & & \{\text{at-r}(y), \text{holding}(x)\}, \\
 & & \{\text{at}(x,y), \neg\text{holding}(x)\}\rangle, \\
\text{take-out}(x,y,z) & = & \langle\langle x,y,z\rangle \\
 & & \{\text{r-at}(y), \text{in-lamp}(x,z), \text{at}(z,y)\}, \\
 & & \{\text{holding}(x), \neg\text{in-lamp}(x,z)\}\rangle, \\
\text{put-in}(x,y,z) & = & \langle\langle x,y,z\rangle, \\
 & & \{\text{r-at}(y), \text{lamp-empty}(z), \text{at}(z,y), \text{holding}(x)\}, \\
 & & \{\text{in-lamp}(x,z), \neg\text{holding}(x), \neg\text{lamp-empty}(z)\}\rangle.
\end{array}$$

A Plan for Changing the Light Bulb

State resulting from **executing a plan**:

$$Res(S, \langle\rangle) = S$$

$$Res(S, (o; \Delta)) = \begin{cases} Res(App(S, o), \Delta) & \text{if } App(S, o) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Plan Δ is successful iff $\mathbf{G} \subseteq Res(\mathbf{I}, \Delta)$.

Initial state: $S = \left\{ \begin{array}{lll} \mathsf{broken}(\mathsf{b}_1), & \mathsf{in\text{-}lamp}(\mathsf{b}_1, \mathsf{l}_1), & \mathsf{at}(\mathsf{l}_1, \mathsf{p}_1) \\ \mathsf{ok}(\mathsf{b}_2), & \mathsf{at}(\mathsf{b}_2, \mathsf{p}_2) & \mathsf{r\text{-}at}(\mathsf{p}_3) \end{array} \right\}$

Plan: $\Delta = \langle \mathsf{goto}(\mathsf{p}_3, \mathsf{p}_2), \mathsf{grab}(\mathsf{b}_2, \mathsf{p}_2), \mathsf{goto}(\mathsf{p}_2, \mathsf{p}_1), \mathsf{take\text{-}out}(\mathsf{b}_1, \mathsf{p}_1, \mathsf{l}_1), \mathsf{put\text{-}in}(\mathsf{b}_2, \mathsf{p}_1, \mathsf{l}_1) \rangle$.

Final state:

Simplifications: DATALOG- and Propositional STRIPS

- General STRIPS allows for unrestricted first-order terms, i.e., arbitrarily nested **function terms**

⇝ **Infinite state space**

- Simplification: No function terms (only 0-ary = constants)

⇝ **DATALOG-STRIPS**

- Simplification: No variables in operator schemata

⇝ **Propositional STRIPS**

More "Expressive Power"

STRIPS : plain STRIPS

STRIPS$_N$: negative literals in preconditions

STRIPS$_{Bd}$: precondition in disjunctive normal form

STRIPS$_{Bc}$: precondition in conjunctive normal form

STRIPS$_B$: Boolean expressions as preconditions

STRIPS$_C$: conditional effects (push-accelerator)

STRIPS$_{C,N}$: conditional effects & negative literals

...

Computational Complexity Theory: What is it About?

- Computational complexity theory classifies **problems** according to their (worst-case) resource requirements.

- **Problem** is synonymous to **formal language**, e.g. the **satisfiability problem (SAT)** is the set of formulae that are satisfiable (simplification!)

- Problems are composed of **instances** of varying size. Usually they are specified by a **Given/Question** pair:

  Given: A well-formed Boolean formula φ.

  Question: Is φ satisfiable?

The Complexity Class P

A *problem* is a member of the **complexity class P** iff the problem can be decided by a **deterministic Turing** machine using only **polynomial time**, i.e.,

> the number of computation steps on input w is bounded by $p(|w|)$, where $|w|$ denotes the size of the input and p is a polynomial.

The problems in P are considered as **"efficiently solvable"**

The Complexity Class NP

- There are many problems that are not known to be decidable in polynomial time, but there exists no proof that these problem need necessarily super-polynomial time.
- Use **non-deterministic** machines to characterize these problems
- **NP** is the class of languages that are decided by **nondeterministic Turing** machine using only **polynomial time**, i.e.,

$\leadsto$ $P \subseteq NP$

$\rightarrow$ Conjectured but not proven: $NP \not\subseteq P$

Reductions

We cannot prove that $P \neq NP$. However, we can identify the hardest problems in NP.

Reduction: Given two languages $L_1, L2$. f is called **polynomial many-one reduction** and we write $L_1 \leq_m L_2$ iff:

1. f can be computed in polynomial time and
2. $x \in L_1$ iff $f(x) \in L_2$.

Intuitively, L_1 can be decided using the decision method for L_2 and some polynomial overhead.

$\leq_m$ is a pre-order (reflexive & transitive)

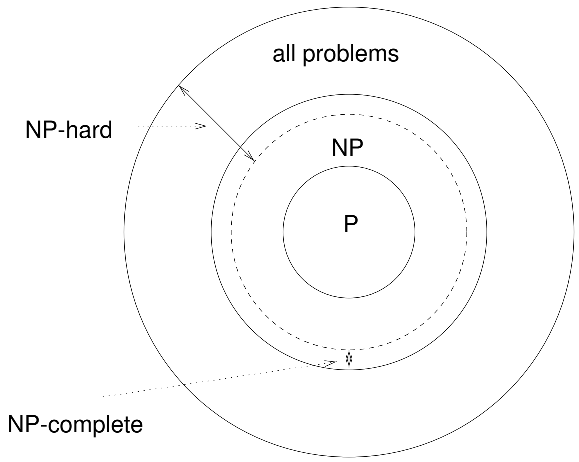
Hard and Complete Problems

- A problem X is called **$\mathcal{C}$-hard** for a complexity class $\mathcal{C}$ (wrt. to poly. many-one reductions) iff for all $L \in \mathcal{C}: L \leq_m X$.
- A problem X is called **$\mathcal{C}$-complete** for a complexity class $\mathcal{C}$ iff it is $\mathcal{C}$-hard and a member of $\mathcal{C}$.
- Intuitively, $\mathcal{C}$-complete probalems are the hardest amongt the $\mathcal{C}$ problems.

NP-complete Problems

- A problem is **NP-complete** iff it is NP-hard and in NP.
- Example: **SAT** – the satisfiability problem for propositinal logic – is NP-complete (Cook/Karp)
- **Membership proofs**: Guess & check algorithm (verification in poly. time = short "proofs")
- **Hardness proofs**: Either **generic reduction** to show $L \leq_m X$ for all $L \in$ NP or use a known NP-complete problem K and show $K \leq_m X$.

Visualization



The Class PSPACE

Definition. PSPACE (NPSPACE) is the class of decision problems that can be decided on deterministic (non-deterministic) Turing machines using only polynomial size.

Some facts about PSPACE:

- PSPACE is closed under complements (as all other deterministic classes)
- PSPACE is identical to NPSPACE (because NDTMs can be simulated on DTMs using only quadratic space)
- NP⊆PSPACE (because in polynomial time one can "visit" only polynomial space, i.e., NP⊆NPSPACE)
- It is unknown whether NP≠PSPACE, but it is believed that this is true.

PSPACE-completeness

A decision problem (or language) is **PSPACE-complete**, if it is in PSPACE and all other problems in PSPACE can be polynomially many-one reduced to it.

Intuitively, PSPACE-complete problems are the "hardest" problems in PSPACE (similar to NP-completeness). They appear to be "harder" than NP-complete problems from a *practical point of view*.

Example for a PSPACE-complete problem:

NDFA equivalence problem:

> **Instance**: Two non-deterministic finite state automata A_1 and A_2.
> **Question**: Are the languages accepted by A_1 and A_2 identical?

Other Complexity Classes ...

- There are complexity classes above PSPACE (EXPTIME, SX-PSPACE, ...),
- there are (infinitely many) classes between NP and PSPACE,
- there are (infinitely many) classes inside P.

Analyzing Planning

Plan existence problem (PLANEX):

Instance: $\Pi = \langle \mathcal{S}, \mathbf{O}, \mathbf{I}, \mathbf{G} \rangle$.
Question: Does there exist a plan $\Delta \in \mathbf{O}^*$ that solves Π, in other words $Res(\mathbf{I}, \Delta) \subseteq \mathbf{G}$?

Bounded plan existence problem (PLANLEN):

Instance: $\Pi = \langle \mathcal{S}, \mathbf{O}, \mathbf{I}, \mathbf{G} \rangle$ and a positive integer n.
Question: Does there exist a plan $\Delta \in \mathbf{O}^*$ of length n or less that solves Π?

From a practical point of view, also PLANGEN (*generating* a plan that solves Π) and PLANLENGEN (*generating* a plan of length n that solves Π) are interesting (but at least as hard as the decision problems).