Control Intensive Planning

Planet 2003 Summer School

Fahiem Bacchus

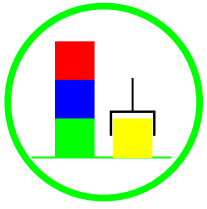
University of Toronto

Why Control Intensive Planning?

- Planning with, e.g., STRIPS operators is computationally hard (PSPACE-complete or worse).
- Hence “domain-independent” control of planning algorithms cannot always be effective.
- Furthermore, we often have knowledge about the planning domain that goes beyond what is expressed in typical domain encodings.
- Such knowledge can yield great gains in efficiency.
- **Why not utilize that knowledge!**

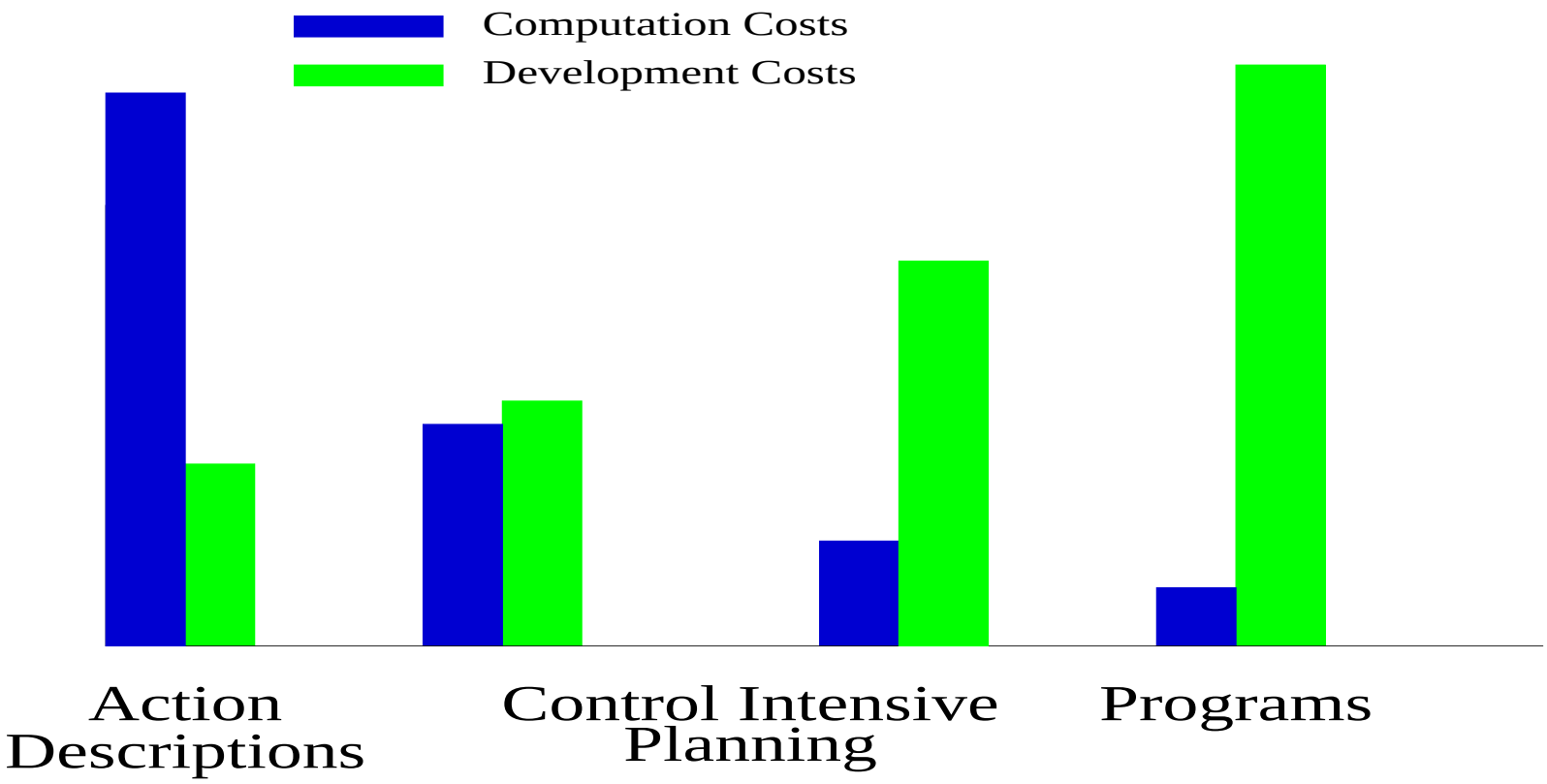
Goals

- **Control Intensive** planning involves finding ways to **build and utilize richer domain models** so as to support more effective problem solving.
- Such an approach is common in many other areas of AI, e.g., Constraint Programming.
- In this lecture I will discuss some ways of accomplishing this within AI planning.
  - I will not attempt to be comprehensive, and I will utilize simple example domains with out regard for their practical significance.



Control Intensive Planning—Costs

- A critical question is how much more effort is required to build better domain models, and is the effort worthwhile?
- Difficult to answer definitively, but the aim is:



Costs...

- The aim is to develop methods for building richer domain models that can support the construction of robust, efficient, and cost effective, domain planners.

- The challenge for the field is to systematize and extend such methods so as to make planning a practical and effective way of solving problems in real applications.

Lecture Outline

- **Background**
  - how to get the most of our of planning representations.
- **Paradigms for Control Intensive Planning**
  - Knowledge of state sequences.
  - Lifted decision making.

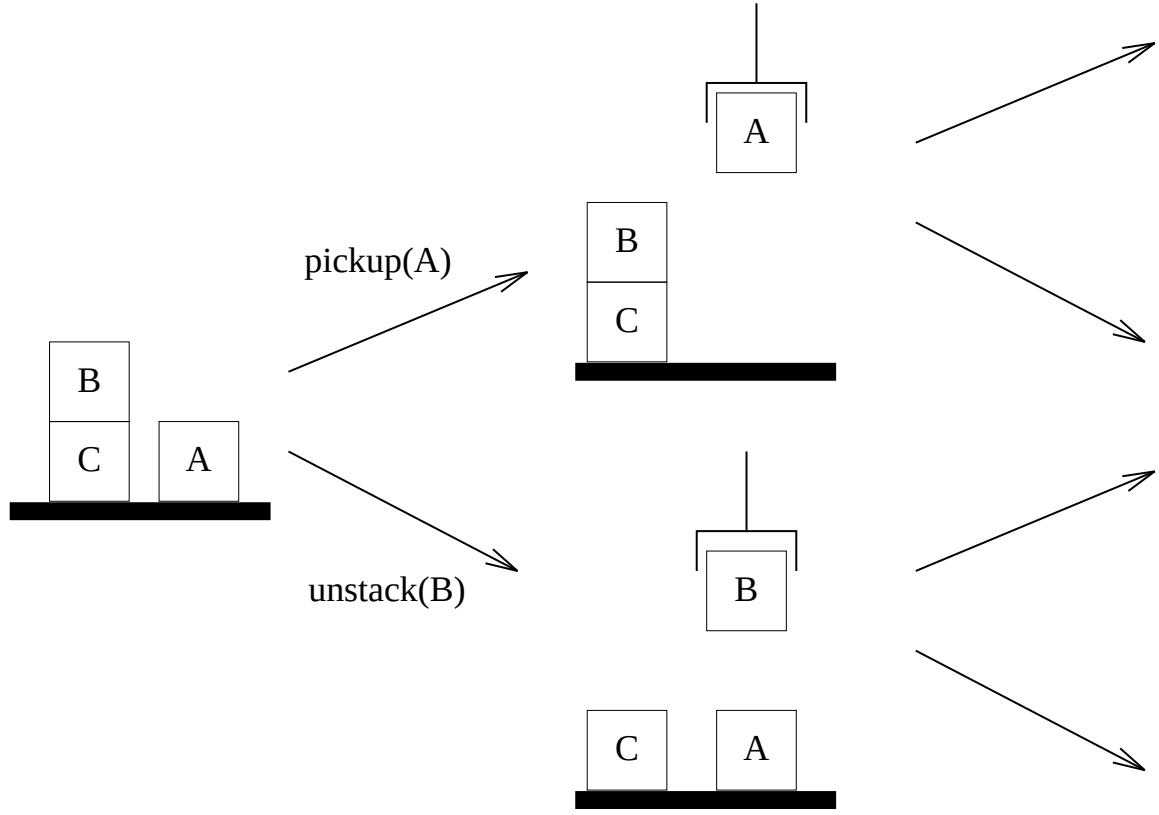
Getting the Most from Planning Representations

Relational Databases

- A key element in control intensive planning is the inherent flexibility and representational power of the STRIPS representation.
- The initial state is represented as a relational database: we have a list of **all** tuples satisfying each relation.
- Actions operate by adding and deleting tuples from relations, thus preserving the state's status as a relational database (its **completeness**).
- Goals are represented as partially specified relations. But it also is **complete** (all relations that **must** be achieved are specified).

Forward Chaining...

- Control Intensive Planners almost exclusively utilize Forward Chaining.



Forward Chaining...

- We start at the initial state and search for a plan in the space of plan prefixes.
- This means that for every candidate plan considered we have a complete description of the world that arises from that plan.
- This information along with information about the goal can be utilized to control planning (search).

Querying the Current State

- The most important thing about relational databases is that they support the efficient evaluation of complex queries.
- This allows us to define **complex concepts** specified as queries against the primitive relations and the goal, and then use these concepts to build richer domain models for controlling planning.
- The query mechanism can also be used to evaluate quantitative measures, e.g., heuristic functions.

First-Order Logic as a Query Language

- A relational database is a first-order model.
- Hence, we can evaluate first-order formulas against this model. That is, we can use first-order logic as our query language.
- To facilitate various examples used in the rest of the talk, I will present a particular mechanism for evaluating first-order formulas.
  - The mechanism is simple and easy to implement (used in my own TLPLAN system).
  - It is quite efficient on small databases like those encountered in planning problems.
- **This is not the only query language nor the only evaluation mechanism that can be used.**

Evaluating First-Order formulas

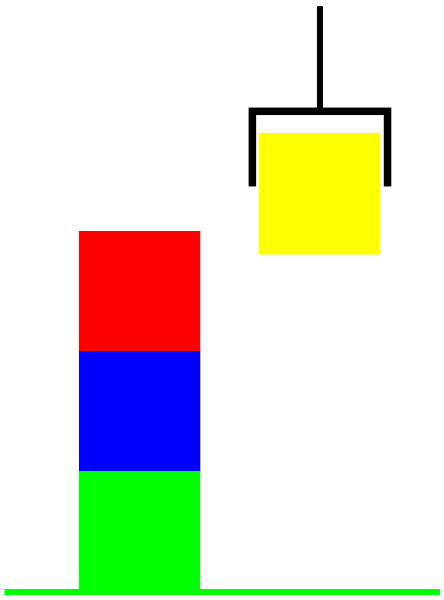
- Say we have a first-order language generated by some finite set of **constant symbols**, $c_1, \ldots, c_n$, **predicate symbols**, $P_1, \ldots, P_m$ (of varying arity), and **function symbols**, $f_1, \ldots, f_o$ (of varying arity).

- In a STRIPS database

  1. The domain of the first-order model consists precisely the set of constant symbols.
  2. For every tuple satisfying a predicate we have that tuple of constants in the predicate's relation.
  3. The value of every function on every tuple of arguments is specified in table.

Example: the Blocks World

- Predicates *on(x,y)*, *ontable(x)*, *clear(x)*, *holding(x)*, and *handempty()*.
- Functions *onTop(x)*.



on(x,y)	*(red,blue)* *(blue,green)*
ontable(x)	*(green)*
clear(x)	*(red)*
holding(x)	*(yellow)*
handempty()	
onTop(x)	*onTop(green) = blue* *onTop(blue) = red* *onTop(red) = nil* *onTop(yellow) = nil*

Eval(ϕ,s,V)

Evaluate a formula ϕ on a state s given some current variable bindings V.

Eval(ϕ,s,V)

ϕ is $P_i(t_1, \ldots, t_n)$	if $P_i($**evalT**$(t_1, s, V), \ldots,$ **evalT**$(t_n, s, V)) \in s$ return(**true**) else return(**false**).
$\phi = \neg\psi$	if **Eval**(ψ,s,V) return(**false**) else return(**true**).
$\phi = \theta \wedge \psi$	if **Eval**(θ, s, V) return(**Eval**(ψ, s, v)) else return(**false**).
$\phi = \forall x.\psi$	for $C = c_1, \ldots, c_n$ V' := $V \cup \{x = C\}$ if **Eval**(ψ,s,V') = **false** return(**false**) return(true)

Eval...

Evaluation is done left to right with **early termination**. This allow recursive queries to be evaluated (as in PROLOG).

Eval(ϕ,s,V)

ϕ is $P_i(t_1, \ldots, t_n)$	if $P_i(\mathbf{evalT}(t_1, s, V), \ldots, \mathbf{evalT}(t_n, s, V)) \in s$ return(**true**) else return(**false**).
$\phi = \neg\psi$	if **Eval**(ψ,s,V) return(**false**) else return(**true**).
$\phi = \theta \wedge \psi$	if **Eval**(θ, s, V) return(**Eval**(ψ, s, v)) **else return(false)**.
$\phi = \forall x.\psi$	for $C = c_1, \ldots, c_n$ V' := $V \cup \{x = C\}$ if **Eval**(ψ,s,V') = **false** **return(false)** return(true)

Eval...

Quantification is replaced by **iteration**.

Eval(ϕ,s,V)

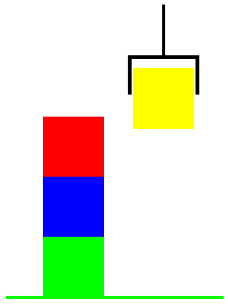
ϕ is $P_i(t_1, \ldots, t_n)$	if $P_i(\mathbf{evalT}(t_1, s, V), \ldots, \mathbf{evalT}(t_n, s, V)) \in s$ return(**true**) else return(**false**).
$\phi = \neg\psi$	if **Eval**(ψ,s,V) return(**false**) else return(**true**).
$\phi = \theta \wedge \psi$	if **Eval**(θ, s, V) return(**Eval**(ψ, s, v)) else return(**false**).
$\phi = \forall x.\psi$	**for** $C = c_1, \ldots, c_n$ V' := $V \cup \{x = C\}$ if **Eval**(ψ,s,V') = **false** return(**false**) return(true)

Eval...

- **evalT** evaluates:
  - Variables are replaced by their binding.
  - Functions are looked up recursively, e.g, $\mathit{onTop}(\mathit{onTop}(green)) = \mathit{onTop}(blue) = red$.
- A bit of notation for bounded quantification:
  - **`(forall (?x) (clear ?x)`** ϕ**`)`** is
    $\forall x.\mathbf{clear}(x) \Rightarrow \phi$
  - **`(exists (?x) (clear ?x)`** ϕ**`)`** is
    $\exists x.\mathbf{clear}(x) \wedge \phi$

Example

```
(def-defined-predicate (above ?x ?y)
   (or (on ?x ?y)
       (exists ?z (on ?z ?y) (above ?x ?z))))
```

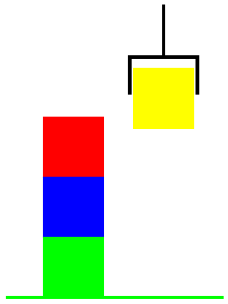


on(x,y)	*(red,blue)* *(blue,green)*

```
==>eval(above red green)
   ==>eval(on red green)
   <==false
   ==>eval(exists (?z) (on ?z ?y) (above ?x ?z))
      ==>eval(on blue green)
      <==true
      ==>eval(above red blue)
         ==>eval(on red blue)
         <==true
      <==true
   <==true
<==true
```

Example

```
(def-defined-predicate (above ?x ?y)
   (or (on ?x ?y)
       (exists ?z (on ?z ?y) (above ?x ?z))))
```

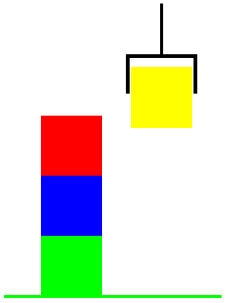


on(x,y)	*(red,blue)* *(blue,green)*

```
==>eval(above red green)
   ==>eval(on red green))
   <==false
   ==>eval(exists (?z) (on ?z ?y) (above ?x ?z))
      ==>eval(on blue green)
      <==true
      ==>eval(above red blue)
         ==>eval(on red blue)
         <==true
      <==true
   <==true
<==true
```

Example

```
(def-defined-predicate (above ?x ?y)
   (or (on ?x ?y)
       (exists ?z (on ?z ?y) (above ?x ?z))))
```

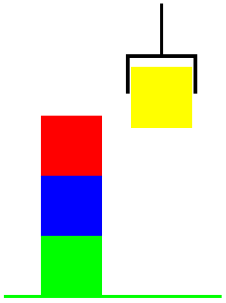


on(x,y)	*(red,blue)* *(blue,green)*

```
==>eval(above red green)
   ==>eval(on red green)
   <==false
   ==>eval(exists (?z) (on ?z ?y) (above ?x ?z)))
      ==>eval(on blue green)
      <==true
      ==>eval(above red blue)
         ==>eval(on red blue)
         <==true
      <==true
   <==true
<==true
```

Example

```
(def-defined-predicate (above ?x ?y)
   (or (on ?x ?y)
       (exists ?z (on ?z ?y) (above ?x ?z))))
```

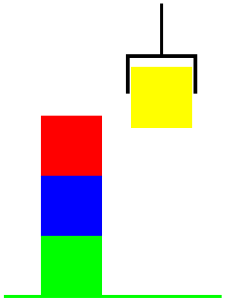


on(x,y)	*(red,blue)* *(blue,green)*

```
==>eval(above red green)
   ==>eval(on red green)
   <==false
   ==>eval(exists (?z) (on ?z ?y) (above ?x ?z))
      ==>eval(on blue green))
      <==true
      ==>eval(above red blue)
         ==>eval(on red blue)
         <==true
      <==true
   <==true
<==true
```

Example

```
(def-defined-predicate (above ?x ?y)
   (or (on ?x ?y)
       (exists ?z (on ?z ?y) (above ?x ?z))))
```

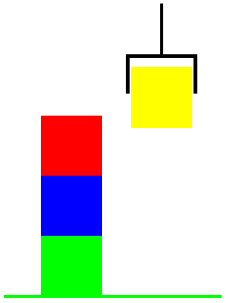


on(x,y)	*(red,blue)* *(blue,green)*

```
==>eval(above red green)
   ==>eval(on red green)
   <==false
   ==>eval(exists (?z) (on ?z ?y) (above ?x ?z))
      ==>eval(on blue green)
      <==true
      ==>eval(above red blue))
         ==>eval(on red blue)
         <==true
      <==true
   <==true
<==true
```

Example

```
(def-defined-predicate (above ?x ?y)
   (or (on ?x ?y)
       (exists ?z (on ?z ?y) (above ?x ?z))))
```



on(x,y)	*(red,blue)* *(blue,green)*

```
==>eval(above red green)
   ==>eval(on red green)
   <==false
   ==>eval(exists (?z) (on ?z ?y) (above ?x ?z))
      ==>eval(on blue green)
      <==true
      ==>eval(above red blue)
         ==>eval(on red blue)
         <==true
      <==true
   <==true
<==true
```

Numerics

- Along with the STRIPS database representing the current state, there is the domain of numbers which is never changed by actions.

- Standard numeric predicates and functions can be evaluated by computations over this “numeric domain.”

Other types of Queries

- Queries can be made against the set of predicates listed in the goal:
  - `(goal (on red blue))`
  - `(exists (?z) (goal (on red ?z)))`
- Finally “pseudo-predicates” can be defined that are evaluated for their side-effects only.

```
(forall (?z) (on green ?z)
             (print "~A~%" ?z))
 ==>blue
```

Examples and Demo

```
(forall (?x) (ontable ?x)
        (implies (clear ?x) (print 0 "~A~%" ?x)))

(def-defined-function (depth ?x)
  (and
     (implies (clear ?x)
        (:= depth 0))
     (implies (not (clear ?x))
         (exists (?y) (on ?y ?x)
                 (:= depth (+ 1 (depth ?y)))))))
```

Examples and Demo

```
(forall (?x) (ontable ?x)
        (implies (clear ?x) (print 0 "~A~%" ?x)))

(def-defined-function (depth ?x)
  (and
     (implies (clear ?x)
        (:= depth 0))
     (implies (not (clear ?x))
         (exists (?y) (on ?y ?x)
                 (:= depth (+ 1 (depth ?y)))))))
```

Examples and Demo

```
(def-defined-predicate (prime ?i)
   (not (exists (?j) (is-between ?j 2 (floor (sqrt ?i)))
                (= 0 (mod ?i ?j)))))

(def-defined-function (count-primes ?x ?y)
   (local-vars ?count)
   (and
      (:= ?count 0)
      (forall (?z) (is-between ?z ?x ?y)
               (implies (prime ?z) (:= ?count (+ ?count 1))))
     (:= count-primes ?count)
  ))
```

Conclusions

- The relational representation commonly used in planning supports rich and expressive query languages.
- We can test complex conditions and compute complex functions on the states using mostly **declarative** specifications.
- We can also use these ideas determine all applicable actions in a forward chaining planner (without having to propositionalize prior to planning).

Knowledge of Sequences

Knowledge of Sequences

- We often know about sequences of states that do not make sense.
- For example, it makes no sense for a plan to load a package onto a truck and then immediately unload it.
- Such a sequence of actions would produce a cycle in the plan—the same state will be visited twice. Such cycles can be pruned with standard techniques in search.

Knowledge of Sequences

- However, more elaborate versions persist. Load package A; then package B; then unload package A. The final state is not identical to the start state. Nevertheless, this sequence does not make sense.

- This idea was first identified by Morris & Kibler, in their paper “Don’t be stupid” (IJCAI 81).

Knowledge of Sequences

- Their idea can be generalized so that more knowledge about sequences can be utilized. The generalization consists of
  - Developing a formalism for expressing general knowledge about sequences.
  - Developing an algorithm for utilizing this knowledge.

Linear Temporal Logic

- LTL (Linear temporal logic) is a logic designed to express information about state infinite sequences.
- We can generalize it to the first-order context.
- The resulting language is very useful for expressing domain knowledge.

First-Order Linear Temporal Logic

- Terms
  - **constant symbols**, $c_1, \ldots, c_n$, and **function symbols**, $f_1, \ldots, f_o$ (of varying arity).
  - All of the c_i as well as all variables from an infinite set of variables $\{x, y, z, \ldots\}$, are **terms**.
  - If $t_1, \ldots, t_k$, then $f_i(t_1, \ldots, t_k)$ is a term for any k-ary function symbol f_i.

First-Order Linear Temporal Logic

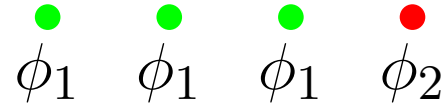
- Formulas
  1. If $t_1, \ldots, t_k$ are terms then for any k-ary predicate symbol P_i, $P_i(t_1, \ldots, t_k)$ is a formula (an atomic formula).
  2. If ϕ and ψ are formulas then so are $\neg\phi$, $\phi \wedge \psi$, and $\phi \vee \psi$.
  3. If ϕ is a formula then so are $\forall x.\phi$ and $\exists x.\phi$.
  4. **Temporal Modalities:** if ϕ and ψ are formulas then so are $\bigcirc\phi$ (**next**), $\Box\phi$ (**always**), $\Diamond\phi$ (**eventually**) and $\phi \,\mathsf{U}\, \psi$ (**until**).
- Note that we are free to nest quantification and the temporal modalities as we wish. Nesting temporal modalities is how LTL gains significant expressive power.

Semantics

- LTL formulas are interpreted over infinite sequences of states $S = \langle s_0, s_1, \ldots \rangle$.
- Each $s \in S$ is a model for the underlying first-order language.
  1. Each s contains the same domain of objects.
  2. Maps each constant symbol to a domain object, each k-ary function symbol to a k-ary function over the domain, and each k-ary predicate to a k-ary relation over the domains.
- An atemporal formula (no temporal modalities) is true in a suffix of S iff that formula is true in the first state s of that suffix. The formula is an arbitrary first-order formula.

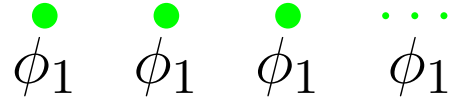
Semantics

- ϕ_1 U ϕ_2 is true in a suffix of S iff ϕ_1 is true **until** ϕ_2 becomes true.

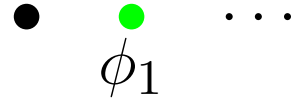


- $\Diamond\phi_1$ is true in a suffix of S iff ϕ_1 **eventually** becomes true.

- $\Box\phi_1$ is true in a suffix of S iff ϕ_1 is **always** true.



- $\bigcirc\phi_1$ is true in a suffix of S iff ϕ_1 is true in the **next** state.



Semantics

- $\forall x.\phi$ is true in a suffix of S iff ϕ_1 is true for all possible assignments of domain values to x.

- $\exists x.\phi$ is true in a suffix of S iff ϕ_1 is true at least one possible assignment of domain values to x.

Examples

- $\bigcirc\bigcirc$*ontable*(red): red is on the table in state s_2.
- $\Box\neg$*holding*$(green)$: *holding*(green) is never true.
- $\Box($*on*$(green, red) \Rightarrow$ *on*$(green, red)$ U *on*$(blue, green)$: whenever green is on red it remains on red until blue is on green.

Examples

- $\Box(\exists x.\mathit{on}(x, red)) \Rightarrow \bigcirc\exists x.\mathit{on}(x, red))$:

  whenever there is a block on red there after there always exists a block on red (but not necessarily the same block).

- $\forall x.\mathit{ontable}(x) \Rightarrow \Box\mathit{ontable}(x)$:

  all objects that are on the table in the **current** state remain on the table in all future states. This time they are the same blocks. (Quantifying into modal contexts).

Notes

- That each state has the same domain of objects is essential for arbitrary nesting of quantification and the modalities.
  - $\exists x.P(x) \wedge \bigcirc(P(x))$
  - If x exists in s but not in s's successor state, there is no easy way to make sense of this formula.

Expressing Control Knowledge

- We express planning knowledge as LTL formulas, and use these formulas to prune the forward chaining search space.
- In the forward chaining search we have a complete plan prefix, yielding a state sequence $\langle s_0, \ldots, s_n \rangle$.
- We can sometimes determine that the LTL formula has been falsified by this sequence: hence all extensions of the sequence will also falsify the LTL formula.
- We prune these falsifying sequences from the search space.

Checking LTL formulas

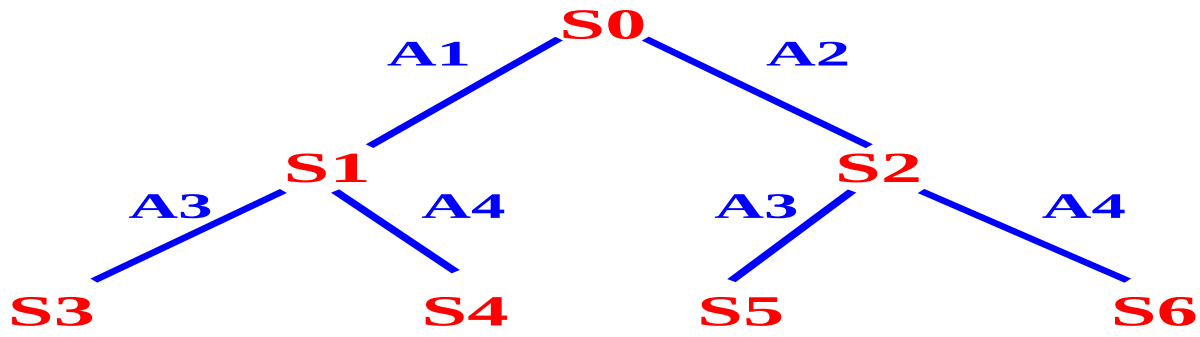
- Given an LTL formula ϕ and a finite sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ we could use the semantic definition, and our query engine for checking the atemporal formulas, to determine whether or not ϕ is falsified.

- Example, if $s_0 \models P(a)$ and $s_1 \models \neg Q(a)$, then $\Box(P(a) \Rightarrow \bigcirc Q(a))$ is falsified.

- Falsification by the finite sequence means that there is no way to extend $\langle s_0, s_1 \rangle$ to a model of the formula (an infinite sequence).

Progressing LTL formulas

- We want to check the LTL formula incrementally
- State sequences are being built up incrementally, and each prefix is extended in a number of different ways.



- In the four sequences, the subsequence $\langle S0 \rangle$ is repeated 4 times. We don't want to repeatedly check the formula on $S0$.
- One method is via various caching schemes (Kvarnström). Another method is progression.

Progressing LTL Formulas

- Say we have an LTL formula ϕ and we want to check whether or not it is falsified by a state sequence $\langle s_0, s_1, \ldots \rangle$.

- The idea is to compute a new formula ϕ^+ by **progressing** ϕ through s_0. ϕ^+ has the property that it will be falsified by $\langle s_1, \ldots \rangle$ if and only if ϕ is falsified by $\langle s_0, s_1, \ldots \rangle$.

- That is, progression captures all of the impact of s_0, so that subsequently we can ignore s_0.

Progressing LTL Formulas

Algorithm Prog(ϕ,s)
case:

1.	ϕ is atemporal:	$\phi^+ := \mathbf{Eval}(\phi, s)$
2.	$\phi = \phi_1 \wedge \phi_2$:	$\phi^+ := \mathbf{Prog}(\phi_1, s) \wedge \mathbf{Prog}(\phi_2, s)$
3.	$\phi = \neg\phi_1$:	$\phi^+ := \neg\mathbf{Prog}(\phi_1, s)$
4.	$\phi = \bigcirc\phi_1$:	$\phi^+ := \phi_1$
5.	$\phi = \phi_1 \mathsf{U} \phi_2$:	$\phi^+ := \mathbf{Prog}(\phi_2, s) \vee (\mathbf{Prog}(\phi_1, s) \wedge \phi)$
6.	$\phi = \Diamond\phi_1$:	$\phi^+ := \mathbf{Prog}(\phi_1, s) \vee \phi$
7.	$\phi = \Box\phi_1$:	$\phi^+ := \mathbf{Prog}(\phi_1, s) \wedge \phi$
8.	$\phi = \forall x.\pi(x) \Rightarrow \phi_1$:	$\phi^+ := \bigwedge_{c \in \pi} \mathbf{Prog}(\phi_1(x/c), s)$

Example

- Say our control formula is $\Box \mathit{ontable}(A)$, and the current state s satisfies $\mathit{ontable}(A)$.

$$\begin{aligned} \mathbf{Prog}(\Box \mathit{ontable}(A)) &= \mathbf{Prog}(\mathit{ontable}(A), s) \wedge \Box \mathit{ontable}(A) \\ &= \mathbf{true} \wedge \Box \mathit{ontable}(A) \\ &= \Box \mathit{ontable}(A) \end{aligned}$$

- On the other hand if the current state does not satisfy $\mathit{ontable}(A)$ we obtain

$$\begin{aligned} \mathbf{Prog}(\Box \mathit{ontable}(A)) &= \mathbf{Prog}(\mathit{ontable}(A), s) \wedge \Box \mathit{ontable}(A) \\ &= \mathbf{false} \wedge \Box \mathit{ontable}(A) \\ &= \mathbf{false} \end{aligned}$$

Notes

Theorem 1 *Let $M = \langle s_0, s_1, \ldots \rangle$ be any LTL model, and let s_i be the i-th state in the sequence of states. Then, we have for any LTL formula ϕ, $\langle M, s_i \rangle \models \phi$ if and only if $\langle M, s_{i+1} \rangle \models \textbf{\textit{Prog}}(\phi, s_i)$.*

- This theorem says that we can check if the sequence we generate satisfies an LTL formula by incrementally progressing our formula through the sequence.
- If the LTL formula progresses to **true**, the sequence definitely satisfies the formula, if it progresses to **false**, it definitely falsifies the formula. In the **false** case we can prune this sequence from the search space.
- Hence, we can prune partial sequences of actions.

Notes

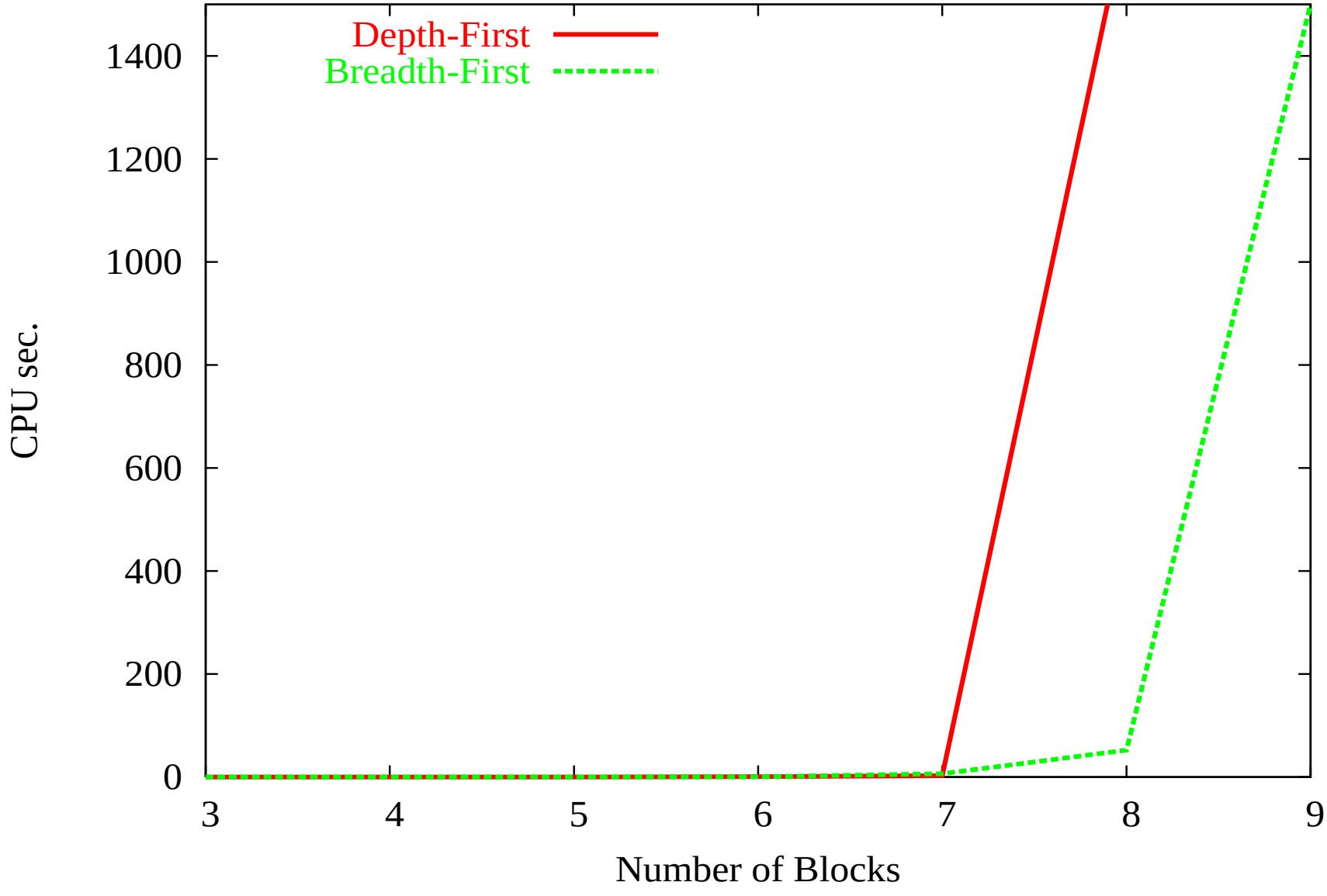
- The progression algorithm uncovers the properties that must be true of the current state, and uses **Eval** to test them.
- Additionally, it computes a representation of the “future” constraints.
- Eventualities offer no pruning power.

Example: the Blocks World

The standard 4 operator description of this world:

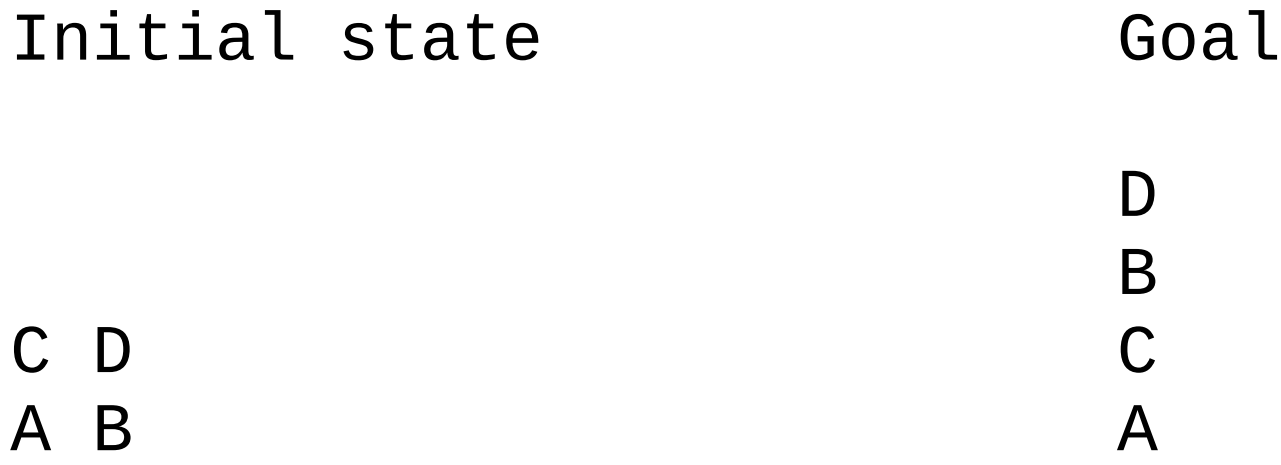
Operator	Preconditions and Deletes	Adds
pickup(x)	*ontable*(x), *clear*(x), *handempty*.	*holding*(x).
putdown(x)	*holding*(x).	*ontable*(x), *clear*(x), *handempty*.
stack(x, y)	*holding*(x), *clear*(y).	*on*(x, y), *clear*(x), *handempty*.
unstack(x, y)	*on*(x, y), *clear*(x), *handempty*.	*holding*(x), *clear*(y).

Blind Search



Final Position Blocks

Consider the situation:



We can solve this problem without unstacking *C* from *A*. *C* and *A* are already in their final position.

- A block is in its **final position** if it and the tower below it do not violate any of the goal's *on* requirements. Neither *D* nor *B* are in their final position: *B* is on the table but it is required to be on *C*, and *D* is on a block that has to be moved.

First-Order Definition

- We can write a recursive first-order formula that defines when a block is in its final position.

```
(def-defined-predicate (in-final-position ?x)
  (or (and (ontable ?x)
           (not (exists (?y) (goal (on ?x ?y)))))

      (exists (?y) (on ?x ?y)
              (and (not (goal (ontable ?x)))
                   (forall (?z) (goal (on ?x ?z))
                           (= ?z ?y))
                   (forall (?z) (goal (on ?z ?y))
                           (= ?z ?x))
                   (in-final-position ?y)))))
```

First-Order Definition

- We can write a recursive first-order formula that defines when a block is in its final position.

```
(def-defined-predicate (in-final-position ?x)
  (or (and (ontable ?x)
           (not (exists (?y) (goal (on ?x ?y)))))

      (exists (?y) (on ?x ?y)
              (and (not (goal (ontable ?x)))
                   (forall (?z) (goal (on ?x ?z))
                           (= ?z ?y))
                   (forall (?z) (goal (on ?z ?y))
                           (= ?z ?x))
                   (in-final-position ?y)))))
```

First-Order Definition

- We can write a recursive first-order formula that defines when a block is in its final position.

```
(def-defined-predicate (in-final-position ?x)
  (or (and (ontable ?x)
           (not (exists (?y) (goal (on ?x ?y)))))

      (exists (?y) (on ?x ?y)
              (and (not (goal (ontable ?x)))
                   (forall (?z) (goal (on ?x ?z))
                           (= ?z ?y))
                   (forall (?z) (goal (on ?z ?y))
                           (= ?z ?x))
                   (in-final-position ?y))))))
```

First-Order Definition

- We can write a recursive first-order formula that defines when a block is in its final position.

```
(def-defined-predicate (in-final-position ?x)
  (or (and (ontable ?x)
           (not (exists (?y) (goal (on ?x ?y)))))

      (exists (?y) (on ?x ?y)
              (and (not (goal (ontable ?x)))
                   (forall (?z) (goal (on ?x ?z))
                            (= ?z ?y))
                   (forall (?z) (goal (on ?z ?y))
                            (= ?z ?x))
                   (in-final-position ?y)))))
```

First-Order Definition

- We can write a recursive first-order formula that defines when a block is in its final position.

```
(def-defined-predicate (in-final-position ?x)
  (or (and (ontable ?x)
           (not (exists (?y) (goal (on ?x ?y)))))

      (exists (?y) (on ?x ?y)
              (and (not (goal (ontable ?x)))
                   (forall (?z) (goal (on ?x ?z))
                           (= ?z ?y))
                   (forall (?z) (goal (on ?z ?y))
                           (= ?z ?x))
                   (in-final-position ?y)))))
```

First-Order Definition

- We can write a recursive first-order formula that defines when a block is in its final position.

```
(def-defined-predicate (in-final-position ?x)
  (or (and (ontable ?x)
           (not (exists (?y) (goal (on ?x ?y)))))

      (exists (?y) (on ?x ?y)
              (and (not (goal (ontable ?x)))
                   (forall (?z) (goal (on ?x ?z))
                           (= ?z ?y))
                   (forall (?z) (goal (on ?z ?y))
                           (= ?z ?x))
                   (in-final-position ?y)))))
```

Preserving GoodTowers

A goodtower is a tower of blocks whose top block is in its final position

```
(always
  (forall (?x) (clear ?x)
    (implies
       (in-final-position ?x)
       (next (and (not (holding ?x))
                  (forall (?y) (on ?y ?x)
                           (in-final-position ?y)
```

This formula says that every good tower should be preserved. Action sequences in which an action destroys a existent good tower should be be pruned.

Bad Towers

What about towers that aren't good towers.

```
(def-defined-predicate (badtower ?x)
  (and (clear ?x) (not (in-final-position ?x))))
```

Clearly these towers need to be dismantled. Hence, it is never useful to stack new blocks on top of such towers. This yields the augmented control formula:

Bad Towers

```
(always
  (forall (?x) (clear ?x)
    (and
      (implies
         (in-final-position ?x)
         (next (and (not (holding ?x))
                    (forall (?y) (on ?y ?x)
                                 (in-final-position ?y)))))
      (implies
         (badtower ?x)
         (next (not (exists (?y) (on ?y ?x))))))))
```

Singleton Towers

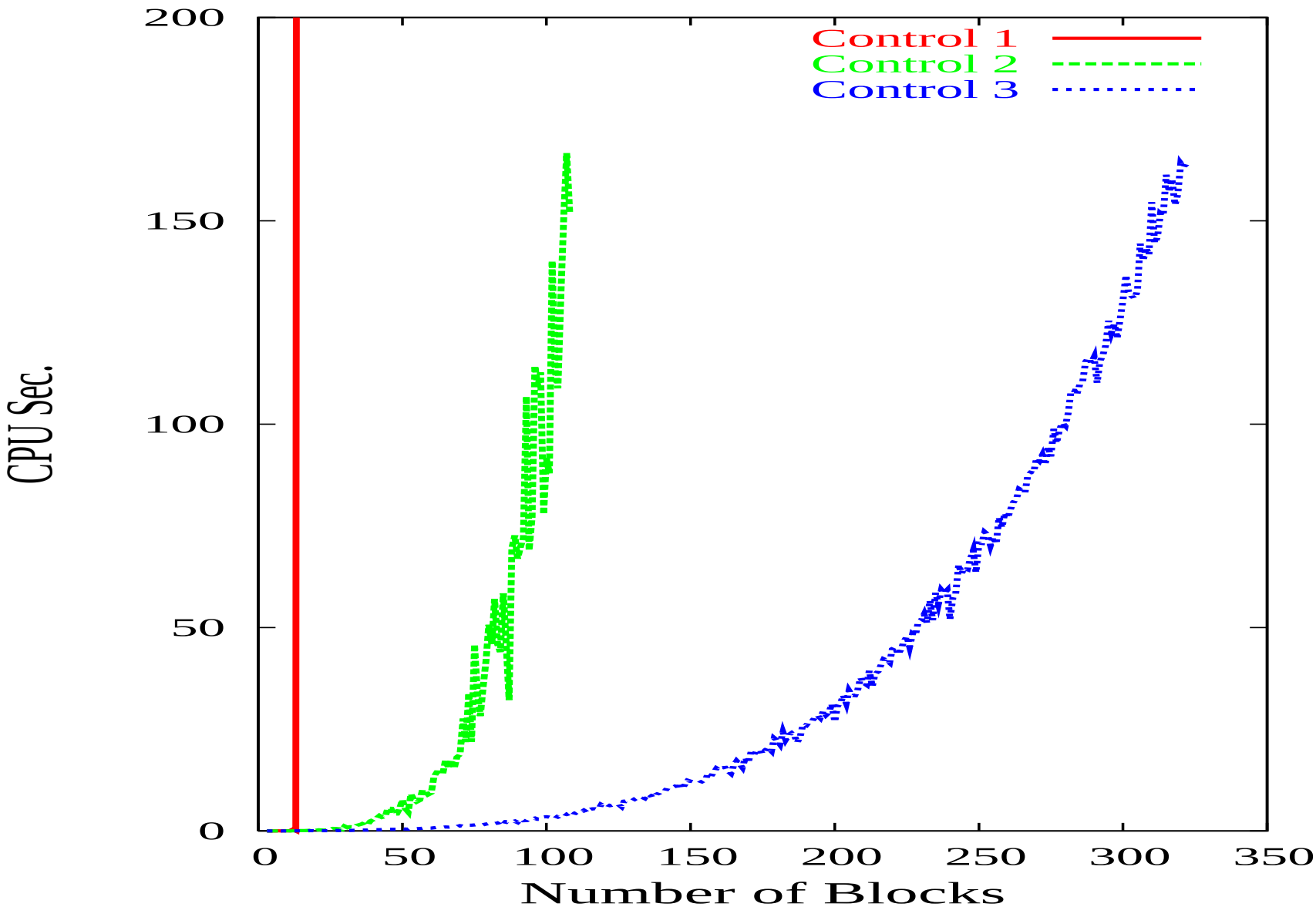
- Finally, a single block that is on the table that is required to be on another block is also a bad tower.

- The above control formula allows the planner to pickup such blocks. Clearly, such blocks should not be picked up unless their final position is ready. This gives the final control formula:

BlocksWorld Control

```
(always
  (forall (?x) (clear ?x)
    (and
      (implies
         (in-final-position ?x)
         (next (and (not (holding ?x))
                    (forall (?y) (on ?y ?x)
                                 (in-final-position ?y)))))
      (implies
         (badtower ?x)
         (next (not (exists (?y) (on ?y ?x))))))

      (implies
         (and (ontable ?x)
              (not (exists (?y) (goal (on ?x ?y))
                                (goodtower ?y))))
         (next (not (holding ?x))))))
```

Controlled Search



Further Optimizations

- Random reconfigurations of 300 blocks with the final control rule requires about 135 sec. on a 1GHz machine.
- Note that blocks world is not an easy domain for domain independent planners: none can reliably solve problems even of size 50.
- 7 years ago this would have been great. However, now the following speeds can be achieved on these types of problems:

Achievable Speeds

# Blocks	Unoptimized	Optimized
300	135 sec.	0.08
350	192 sec.	0.14
400		0.19
450		0.12
500		0.14
1,000		0.85
5,000		13.10

Almost 20,000 steps in the 5,000 block plan.

Mechanisms for Speedups

- These speedup can be achieved with general purpose mechanisms.

Materialized Views

- In Databases there is the concept of Views: a view is a set of tuples that satisfies a query, e.g., the set of all blocks that satisfy the query `(and (ontable ?x) (clear ?x))`.
- These are the blocks that can be picked up.
- The observation is that since database transactions only makes local changes to the database, it can be more efficient to **materialize** the view.
- That is, we build the relation containing all satisfying tuples, and when the database is updated we compute the incremental update to the materialized view.

Materialized Views

- It is not that difficult to achieve: there is a way of automatically analyzing the query to compute how it should be updated.
- For example, **`(and (ontable ?x) (clear ?x))`**:
  - If **`ontable`** and **`clear`** are unchanged, then so is the view.
  - If **`(ontable c)`** is added, then check if **`(clear c)`** is true, and if so add **`c`** to the view.
  - If **`(ontable c)`** is deleted, then delete **`c`** from the view (if it is in the view).
- These incremental changes are much more efficient than recomputing the entire query.

Optimizing Precondition Checking

This kind of analysis is easy to do with STRIPS actions.

1. We define a set of new predicates, representing the preconditions of the actions.
2. We create a set of initialization steps that are executed prior to planning. These steps initialize the views.
3. We analyze the action effects and add extra effects to maintain the views.
4. We substitute the views for the preconditions.

The Blocks World

Operator	Preconditions and Deletes	Adds
pickup(x)	*ontable*(x), *clear*(x), *handempty*.	*holding*(x).
putdown(x)	*holding*(x).	*ontable*(x), *clear*(x), *handempty*.
stack(x, y)	*holding*(x), *clear*(y).	*on*(x, y), *clear*(x), *handempty*.
unstack(x, y)	*on*(x, y), *clear*(x), *handempty*.	*holding*(x), *clear*(y).

canUnstack—2. Initialization

```
(canUnstack ?x ?y) = (and (clear ?x) (on ?x ?y))

(set-initialization-sequence

  (forall (?x ?y) (on ?x ?y)
          (implies (clear ?x)
                   (add (canUnstack ?x ?y))))

)
```

canUnstack—3. Maintaining

Operator	Preconditions and Deletes	Adds
pickup(x)	*ontable*(x), *clear*(x), *handempty*.	*holding*(x).

pickup deletes **`(clear x)`** but in the blocks world **`(ontable ?x)`** implies

```
(not (exists (?y) (on ?x ?y)))
```

so **`pickup`** cannot cause any changes to **`canUnstack`**.

canUnstack—3. Maintaining

Operator	Preconditions and Deletes	Adds
putdown(x)	*holding*(x).	*ontable*(x), *clear*(x), *handempty*.

putdown adds **`(clear x)`** but again **`(ontable ?x)`** implies

```
(not (exists (?y) (on ?x ?y)))
```

so **`putdown`** also cannot cause any changes to **`canUnstack`**.

canUnstack—3. Maintaining

Operator	Preconditions and Deletes	Adds
stack(x, y)	*holding*(x), *clear*(y).	*on*(x, y), *clear*(x), *handempty*. ***canunstack(x,y)***

stack adds **`canUnstack`**.

canUnstack—3. Maintaining

Operator	Preconditions and Deletes	Adds
unstack(x, y)	*on*(x, y), *clear*(x), *handempty*. ***canUnstack(x,y)***	*holding*(x), *clear*(y).

unstack deletes **`canUnstack`**.

canUnstack—4. Substituting

Operator	Preconditions and Deletes	Adds
unstack(x, y)	*handempty*. ***canUnstack(x,y)***	*holding*(x), *clear*(y).

unstack can remove other preconditions.

Benefits

- In problems involving 5,000 blocks, materializing the preconditions can yield a tremendous computational savings—each action makes very few changes.
- Even greater computational gain can be achieved by materializing ***in-final-position***.
- Note that materializing preconditions can benefit any forward chaining planner, including domain independent ones. (Graphplan and other backwards chaining planners?)

Open Research Problem

- Can this process be fully automated?

  1. Requires using domain information for full simplification: e.g., realizing that ontable is incompatible with being on another block.

  2. Requires identifying subsets of the preconditions that are worth materializing. E.g., consider materializing

```
(and (handempty) (clear ?x) (on ?x ?y))
```

Removing LTL formulas

- In the blocks world, and many other domains the control is in the form

  (always (implies ϕ (next ψ)))

  where ϕ and ψ are atemporal (no modalities).
- Any operator O takes a state s and maps it into a next state t.
- s must satisfy O's preconditions, and t is a modification of s by O's effects.
- We can ask the question "**under what circumstances will O cause a state satisfying ϕ to falsify ψ.**"

Example

- A simple transformation of the BlocksWorld Control yields the following rule:

```
(always
   (implies
      (and (clear ?x) (in-final-position ?x))
      (next (and (not (holding ?x))
                 (forall (?y) (on ?y ?x)
                         (in-final-position ?y))))))
```

- It is obvious that we should not *pickup* nor *unstack* a block that is in its final position, as both cause *holding* in the next state. *putdown* cannot falsify this rule. And *stack* must be restricted so that if it stacks unto a block in its final position, it must stack the right block.

Example

- A simple transformation of the BlocksWorld Control yields the following rule:

```
(always
   (implies
      (and (clear ?x) (in-final-position ?x))
      (next (and (not (holding ?x))
                 (forall (?y) (on ?y ?x)
                          (in-final-position ?y))))))
```

- **Should not *pickup* nor *unstack* a block that is in its final position**—as both cause *holding* in the next state.

Example

- A simple transformation of the BlocksWorld Control yields the following rule:

```
(always
   (implies
      (and (clear ?x) (in-final-position ?x))
      (next (and (not (holding ?x))
                 (forall (?y) (on ?y ?x)
                          (in-final-position ?y))))))
```

- ***putdown* cannot falsify this rule**—its precondition *holding* is incompatible with **`(and (clear ?x) (in-final-position ?x))`**.

Example

- A simple transformation of the BlocksWorld Control yields the following rule:

```
(always
   (implies
      (and (clear ?x) (in-final-position ?x))
      (next (and (not (holding ?x))
                 (forall (?y) (on ?y ?x)
                          (in-final-position ?y))))))
```

- ***stack* must be restricted**—it creates **`(on ?y ?x)`** so it must ensure that it stacks the right block.

Benefits

- It is not difficult to encode the entire control rule into extra action preconditions.
- These preconditions ensure that the control rule is never violated.
- The benefit is that we no longer have to generate the progressed formula, and instead of generating the world then determining that it is bad, we never generate the world at all.

Open Research Problems

- Can this process be fully automated?

  1. The simplest cases of **(always** ϕ **(next** ψ**))** have been treated (Kvarnström).
  2. Can more general cases be automated?
  3. Can be cast as a theorem proving problem, and again it requires utilizing extra domain information.

Pandora's Box

Preconditions

- We have seen that in some cases we can convert LTL control formulas into action preconditions.
- This can be done automatically, potentially in more general cases.
- But remember that our aim was to develop methods for effectively expressing and using control knowledge.
- In some cases it can be just as easy (easier?) to write domain control knowledge directly as preconditions.

Logistic Domain

```
(def-adl-operator (load ?obj ?v ?loc)
   (pre (?obj ?loc) (at ?obj ?loc)
        (?v)        (at ?v ?loc)
        (and (vehicle ?v) (object ?obj)))
    (add (in ?obj ?v))
    (del (at ?obj ?loc)))

(def-adl-operator (unload ?obj ?v ?loc)
   (pre (?obj ?v) (in ?obj ?v)
        (?loc)    (at ?v ?loc))
   (add (at ?obj ?loc))
   (del (in ?obj ?v)))
```

Logistic Domain

```
(def-adl-operator (load ?obj ?v ?loc)
   (pre (?obj ?loc) (at ?obj ?loc)
        (?v)        (at ?v ?loc)
        (and (vehicle ?v) (object ?obj)))
    (add (in ?obj ?v))
    (del (at ?obj ?loc)))

(def-adl-operator (unload ?obj ?v ?loc)
   (pre (?obj ?v) (in ?obj ?v)
        (?loc)    (at ?v ?loc))
   (add (at ?obj ?loc))
   (del (in ?obj ?v)))
```

Logistic Domain

```
def-adl-operator (drive ?t ?from ?to)
   (pre (?t)    (truck ?t)
        (?from) (at ?t ?from)
        (?city) (in-city ?from ?city)
        (?to)   (in-city ?to ?city)
        (not (= ?from ?to)))
   (add (at ?t ?to))
   (del (at ?t ?from)))

(def-adl-operator (fly ?p ?from ?to)
   (pre  (?p)    (airplane ?p)
         (?from) (at ?p ?from)
         (?to)   (airport ?to)
         (not (= ?from ?to)))
   (add (at ?p ?to))
   (del (at ?p ?from)))
```

Logistic Domain

```
def-adl-operator (drive ?t ?from ?to)
   (pre (?t)    (truck ?t)
        (?from) (at ?t ?from)
        (?city) (in-city ?from ?city)
        (?to)   (in-city ?to ?city)
        (not (= ?from ?to)))
   (add (at ?t ?to))
   (del (at ?t ?from)))

(def-adl-operator (fly ?p ?from ?to)
   (pre  (?p)    (airplane ?p)
         (?from) (at ?p ?from)
         (?to)   (airport ?to)
         (not (= ?from ?to)))
   (add (at ?p ?to))
   (del (at ?p ?from)))
```

Logistic Domain

- There are a collection of objects in various locations in various cities.
- The goal involves changing the locations of these objects to other cities or in the same city.
- To move to another city might involve moving by truck to an airport, then by plane to the other city, then by truck to the final destination.

Control Knowledge

There are a number of obvious controls for this domain.

- Complete all loads and unloads of a vehicle prior to moving it.
- Don’t move a vehicle to a irrelevant location.
- Don’t load or unload objects from a vehicle unless we need to.

Useful Definitions

```
(def-defined-predicate (vehicle ?v)
   (or (truck ?v) (airplane ?v)))

(def-predicate (wrong-city ?obj ?c-loc)
  ;;?obj at ?c-loc is in the wrong city
  (exists (?g-loc) (goal (at ?obj ?g-loc))
          (?city)  (in-city ?c-loc ?city)
          (not (in-city ?g-loc ?city))))
```

Useful Definitions

```
(def-predicate (move-by-truck ?obj ?c-loc)
  ;;?obj at ?c-loc needs to be moved by a truck
  (or
    (and (wrong-city ?obj ?c-loc)
         (not (airport ?c-loc))
    (and (not (wrong-city ?obj ?c-loc))
         (exists (?g-loc) (goal (at ?obj ?g-loc)
                  (not (= ?g-loc ?c-loc)))))

(def-predicate (move-by-plane ?obj ?c-loc)
  ;;?obj at ?c-loc needs to be moved by a plane
  (wrong-city ?obj ?c-loc))
```

Useful Definitions

```
(def-predicate (unload-from-truck ?obj ?c-lo
  ;;?obj needs to be unloaded at ?c-loc
  (or
   (goal (at ?obj ?c-loc))
   (and (wrong-city ?obj ?c-loc)
        (airport ?c-loc))))

(def-predicate (unload-from-plane ?obj ?c-lo
  ;;?obj needs to be unloaded at ?c-loc
  (not (wrong-city ?obj ?c-loc)))
```

Precondition Control

```
(def-adl-operator (load ?obj ?v ?loc)
 (pre (?obj ?loc) (at ?obj ?loc)
      (?v)        (at ?v ?loc)
      (and (vehicle ?v) (object ?obj)
           (implies (truck ?v)
                    (move-by-truck ?obj ?loc))
           (implies (airplane ?v)
                    (move-by-plane ?obj ?loc))))
 (add (in ?obj ?v))
 (del (at ?obj ?loc)))
```

Precondition Control

```
(def-adl-operator (unload ?obj ?v ?loc)
    (pre (?obj ?v) (in ?obj ?v)
         (?loc)    (at ?v ?loc)
    (and (implies (truck ?v)
              (unload-from-truck ?obj ?loc))
         (implies (airplane ?v))
              (unload-from-plane ?obj ?loc))))
   (add (at ?obj ?loc))
   (del (in ?obj ?v)))
```

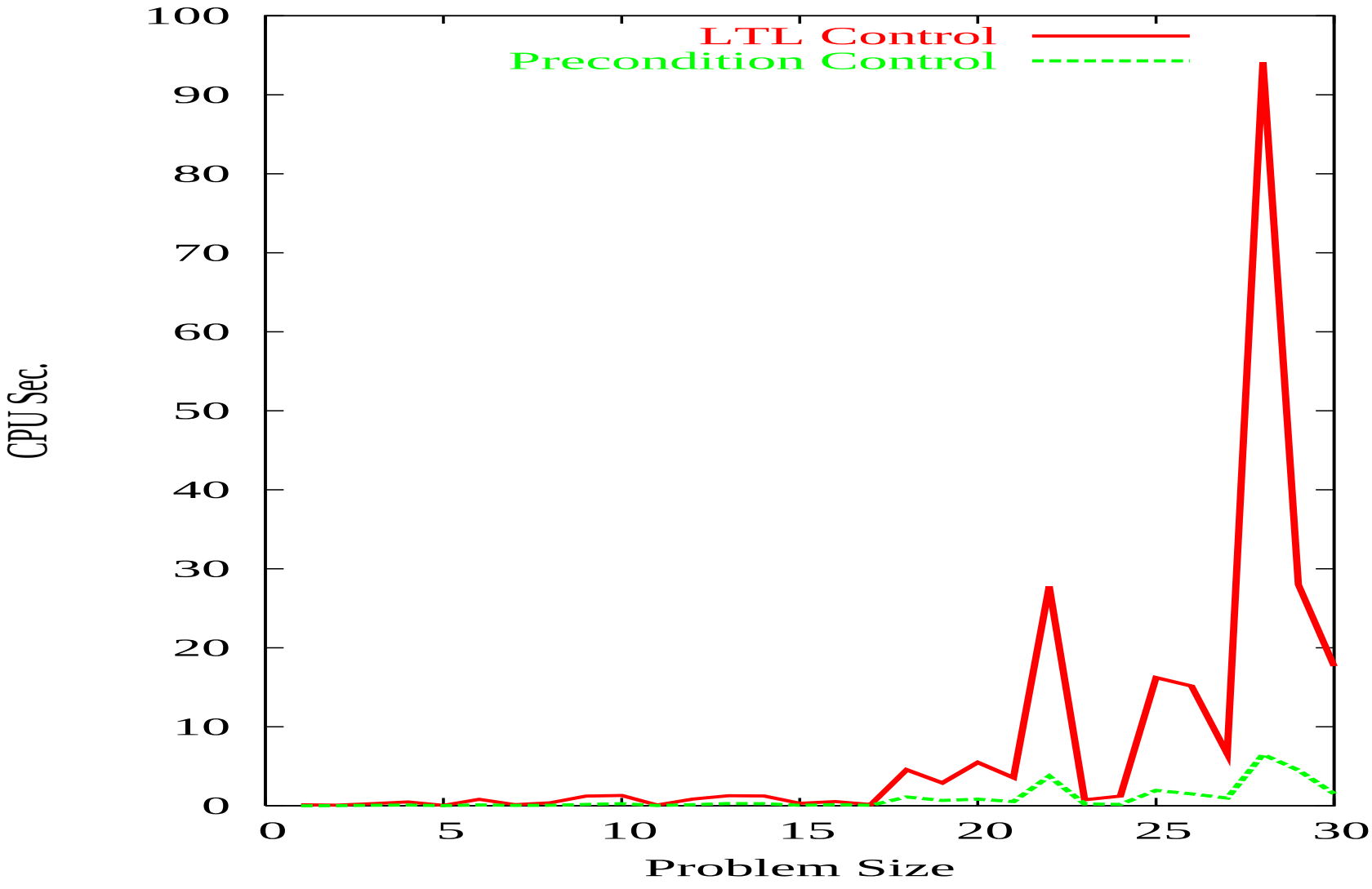
Precondition Control

```
def-adl-operator (drive ?t ?from ?to)
   (pre (?t)    (truck ?t)
        (?from) (at ?t ?from)
        (?city) (in-city ?from ?city)
        (?to)   (in-city ?to ?city)
        (and (not (= ?from ?to))
             (not (exists (?obj) (at ?obj ?from)
                    (move-by-truck ?obj ?from)))
             (not (exists (?obj) (in ?obj ?t)
                    (unload-from-truck ?obj ?from)))
             (or (goal (at ?t ?to))
                 (exists (?obj) (at ?obj ?to)
                   (move-by-truck ?obj ?to))
                 (exists (?obj) (in ?obj ?t)
                   (unload-from-truck ?obj ?to)))))

   (add (at ?t ?to))
   (del (at ?t ?from)))
```

Precondition Control

```
(def-adl-operator (fly ?p ?from ?to)
   (pre  (?p)    (airplane ?p)
         (?from) (at ?p ?from)
         (?to)   (airport ?to)
         (and (not (= ?from ?to))
              (not (exists (?obj) (at ?obj ?from)
                     (move-by-plane ?obj ?from)))
              (not (exists (?obj) (in ?obj ?p)
                     (unload-from-plane ?obj ?from)))
              (or (goal (at ?p ?from))
                  (exists (?obj) (at ?obj ?to)
                    (move-by-plane ?obj ?to))
                  (exists (?obj) (in ?obj ?p)
                    (unload-from-plane ?obj ?to)))))

   (add (at ?p ?to))
   (del (at ?p ?from)))
```

Performance



Notes

- Is any harder to write precondition control? For many domains it is simpler.
- Nevertheless, the issue of automatically generating/computing control is an important research topic.
- LTL is probably a better target for automatically generated control.

Lifting Decision Making

Introduction

- The notion of lifted decision making is less well understood and it is currently harder to see how it can be made systematic or automated.
- Nevertheless, it can make a tremendous difference to the ease of writing control knowledge.
- I will introduce the idea mainly through an example.

Timed Satellite World

- The Satellite domain was created as part of the 2002 International Planning Competition.
- There are various versions of the domain, I will talk about the timed version.
  1. Action durations vary.
  2. No resources other than time.

Timed Satellite World

- The domain contains a collection of satellites, each with some group of instruments. Each instrument supports some set of image collection modes.
- The instruments must be powered up, and calibrated prior to use. Once calibrated it instrument can be used in any of its modes.
- The satellite must be slewed to a particular direction in order to calibrate an instrument or to take an image.
- The goal is to collect some set of images, each of a particular mode in a particular direction.

Actions

```
(turn_to satellite new_dir old_dir
        ...
```

- The satellite starts off pointing at `old_dir` and after `(slew_time old_dir new_dir)` time ends up pointing at `new_dir`

Actions

```
(switch_on instrument satellite
           ...
```

- If the satellite has power available it powers on an on board instrument in two time units.
- The satellite no longer has power available (only one on board instrument can be on at a time).
- The instrument is no longer calibrated.

Actions

```
(switch_off instrument satellite
      ...
```

- If instrument is on, it powers it off in one time unit.
- The satellite then has power available.

Actions

```
(calibrate satellite instrument
           direction
      ...
```

- The instrument must have power on, the satellite must be pointing in direction, and direction must be a calibration target for the instrument
- After time `(calibration_time instrument direction)` the instrument is calibrated.
- The instrument must stay powered on for the entire duration of the action.

Actions

```
(take_image satellite direction
            instrument mode
    ...
```

- The instrument must be calibrated during the duration of the action. The satellite must be pointing in direction, and the instrument must support images of mode.
- After time 7 time units we `(have_image direction mode)`.

Control

- The domain is very hard for domain-independent planners. Such planners have problems computing good heuristics since many different satellites can acquire the same data.

- Writing control using the previously presented ideas is also hard.

Lifted Decision Making

- The fundamental decisions to be made in this domain is the choice of which satellite is to acquire what data and in what sequence.
- Once we know that satellite A is to acquire data item D, the rest of the control is relatively easy:
  1. Power up the instrument.
  2. Calibrate the instrument by first slewing the satellite to point at a calibration target.
  3. Slew the satellite to the data direction.
  4. Take the image.

Lifted Decision Making

- Concurrency is relatively easy:
  1. Don’t choose more than one satellite to acquire the same data.
  2. Finish acquiring the data before using the satellite to acquire more data.

Achieving the Lifted Control

- Introduce a new action `(allocate-instrument ?sat ?inst)`
- Pick an unallocated satellite, an instrument on board, a direction, and a set of data items that can be acquire by the instrument in that direction.
- The choice between the different allocate actions yields a search space in which all acquisition sequences can be explored.

Achieving the Lifted Control

- `allocate-instrument` adds to the state space two new predicates:
  1. `(sat-control ?sat ?dir ?inst)` to indicate that the satellite is being controlled to acquire data with `?inst` in direction `?dir`.
  2. A collection of `(to-be-collected ?sat ?dir ?mode)` assertions, each one indicating that the satellite is going to acquire an image of `?mode` in direction `?dir`

Power on/off Control

- We only `switch-off` an instrument on a satellite if the satellite is being controlled to acquire data with a different instrument.
- We only `switch-on` an instrument on a satellite if the satellite is being controlled to acquire data with that instrument.

Calibration Control

- We only slew the satellite to a calibration target if the instrument is not yet calibrated and it is to be used to acquire data.
- No other slew directions are allowed until the instrument is powered up and calibrated.
- We only `calibrate` an instrument if the instrument is not yet calibrated and the satellite is being controlled to acquire data with that instrument.

Take Image Control

- We only slew the satellite to the control direction if the instrument is calibrated and powered up.
- No other slew directions are allowed.
- We only allow `(to-be-collected)` images to be taken.

Short Duration Plans

- There are two types of slew operations needed in a plan, one to slew a satellite to a calibration target, and another to slew the satellite to take an image.
- It turns out that the slew times are non-Euclidean, i.e., a direct slew from A to B might take more time than a slew from A to C then from C to B.
- To get short duration plans, we need to control the satellite so that it slews by using a series of slew steps over the shortest path.
- Although unnatural for this domain, “shortest path” sub-problems are common in many planning domains.

Short Duration Plans

- The set of directions form the vertices of a graph, and the slew time between each pair of direction form the edge costs. (In this case every pair of directions is connected).
- There are well know algorithms for finding shortest paths in finite graphs, it would be wasteful for the planner to engage in search to try to find short paths.

Controlling for Shortest Paths

- First we initialize two binary functions (`Mslew_times ?x ?y`) and (`Mslew_next ?x ?y`) by using a Floyd-Warshall all pairs shortest path algorithm.

```
(set-initialization-sequence
   (forall (?d1 ?d2 ?cost ?next)
           (all-pairs-shortest-path
              direction   ;;vertex predicate
              slew_time   ;;edge cost function
              ?d1 ?d2     ;;bind to all pairs
              ?cost       ;;bind to cost
              ?next)      ;;bind to next edge on
                          ;; shortest path
         (and
           (add (= (Mslew_times ?d1 ?d2) ?cost))
           (add (= (Mslew_next ?d1 ?d2) ?next)))))
```

Controlling for Shortest Paths

- Then force the planner to use the shortest plan we add two new actions

```
(Mturn_to_calibrate ?sat ?d_new ?d_prev)
(Mturn_to_takeimage ?sat ?d_new ?d_prev)
```

- The first action requires that an instrument needs to be calibrated, and `?d_new` is the closest calibration target enroute to the final image direction.
- The second requires that the instrument is calibrated and `?d_new` is the final image direction.
- These actions add the predicate `(Expand_turn_to ?sat ?d_new ?d_prev)`.

Controlling for Shortest Paths

- Finally, the primitive
  `(turn_to ?sat ?d_next ?d_prev)`
  requires
  `(Expand_turn_to ?sat ?d_new ?d_prev)`
  as a precondition and that `?d_prev` be the satellite's current direction and that `?d_new` be equal to
  `(Mslew_next ?d_prev ?d_new)`,
  i.e., the next step on the shortest path to the final destination.

Greedy Search for Plans

- Greedy heuristics are used to determine which allocations to explore first—in the competition Tlplan planner only reported the first greedy discovered plan.

Conclusions

- We need not be restricted to “physical” actions and predicates when model domains for planning.
- Instead actions can represent “meta-level” decisions, or reasons for action choices.
- The issue of utilizing know efficient algorithms like shortest path to solve subproblems is an important one—how to integrate the use of such algorithms in a clean way?
- The notion of lifted decision making is very related to HTN planners (Hierarchical Task Networks).
- HTN planners use hierarchical decompositions of the domain, but they have some difficulty interfacing with goals specified as collections of predicates.

References

Slide 2 There has been a lot of work on the complexity of planning, subject to various restrictions, an example of this kind of work is K. Erol, D. Nau, and V. S. Subrahmanian. **Complexity, decidability and undecidability results for domain-independent planning**. Artificial Intelligence, 76(1–2):75–88, 1995.

Slides 7–30 The first order query language described in these slides is specified in more detail in *Using Temporal Logics to Express Search Control Knowledge for Planning*, F. Bacchus and F. Kabanza, Artificial Intelligence volume 16, pages 123–191, 2000. Further demonstration of the power of the query language, e.g., for specifying various types of search algorithms, is presented in the manuscript (available from my home page `http://www.cs.toronto.edu/~fbacchus`
Evaluating First Order Formulas—the foundation for a general Search Engine, F. Bacchus and M. Ady, 1999.

Slides 31–68 The details of using LTL to express search control are presented in *Using Temporal Logics to Express Search Control Knowledge for Planning*, F. Bacchus and F. Kabanza, Artificial Intelligence volume 16, pages 123–191, 2000.

Slides 31–68 If you want to experiment with this way of representing search control you can download the TLPLAN system. A Linux executable is available from *The* TLPLAN *homepage*. (Look near the bottom of the page). All the domains we implemented for the 2002 planning competition are also available.

Slide 33 “Don’t be Stupid,” D. Kibler and P. Morris, IJCAI 1981, pages 345–347.

Slide 81 There is a large literature on materialized views from the database community. A google search will find many leads.

References

Slide 88 Kvarnström work on automatically converting temporal control to precondition control is described in Jonas Kvarnström and Patrick Doherty. **TALplanner: A temporal logic based forward chaining planner**. Annals of Mathematics and Artificial Intelligence, 30:119-169, 2000. And Jonas Kvarnström. **Applying Domain Analysis Techniques for Domain-Dependent Control in TALplanner**. In Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling, AAAI Press, 2002, pages 101-110. Both are available from Kvarnström's home page. *http://www.ida.liu.se/~jonkv/*

Slide 88 Rintanen has done some other work on converting temporal control into preconditions: J. Rintanen. **Incorporation of temporal logic control into plan operators**, ECAI 2000. Proceedings of the 14th European Conference on Artificial Intelligence, Werner Horn, ed., pages 526-530, IOS Press, Amsterdam, 2000. The paper is available from his homepage *http://www.informatik.uni-freiburg.de/~rintanen/*

Slides 90–105 Precondition control is explained in more detail in the manuscript *Precondition Control*, F. Bacchus and M. Ady, 1999.

Slide 129 HTN planning is well described in various publications by *Dana Nau* and his group. A good starting place is *SHOP: Simple Hierarchical Ordered Planner*, In IJCAI-99, pp. 968-973, 1999. D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila.