An Overview of AI Planning

Fahiem Bacchus
University of Toronto

Outline

- Planning Domains and Problems
- Representing Actions (the transitions)
  - Formalisms
    - STRIPS
    - ADL (Action Description Language)
    - Situation Calculus with successor state axioms
  - Features
    - Modular updates arising from the frame assumption
    - Parameterized, relational representations

Outline

- Finding Plans
  - Alternate State Spaces
    - Propositional Spaces
      - Direct translation to SAT
      - GraphPlan (adding local constraints)
    - Plan space
      - Partially ordered parameterized plans.
    - Forward Space
      - Distance Heuristics
      - Domain specific search control

Planning Domains

- In AI planning the focus is on relational representations, with predicates and objects.
- For each planning domain we have a specific first-order language containing predicates, and functions useful for describing the domain.
- A planning domain is defined by a set of operators which are a parameterized representation of the transitions available in the domain.

Planning Problems

- Given a planning domain D a classic planning problem
  - Inputs
    - The initial state
    - The goal
    - D
  - State Space
    - All states reachable by applying a sequence of actions to the initial state
    - Actions are operators with their parameters instantiated by constants mentioned in the initial state.
  - Output
    - A sequence of actions that transform the initial state to a state satisfying the goal.

Classical Planning

- Assumptions
  - There is complete knowledge of the initial state.
  - The actions are deterministic, their effects are completely specified, and they do not modify the set of objects in the world, thus they preserve completeness of knowledge.
  - The goal is a property of an individual state.
- Extensions
  - various extensions have been explored but these extensions are quite close to the classical framework.

Representing the Initial state

- Use a relational representation.
- Simplest case. The initial state is a set of ground atomic predicates to which
  - Predicate completion is applied.
    - The set contains all true ground instances of each predicate.
  - Domain closure is implicit.
    - The set of constants mentioned in the set are all the objects in the world. (Thus the model is finite.)
  - The unique names assumption is applied.
    - Distinct constants are not equal.
- Like a database.

Representing the Initial state

- More generally:
  - The initial state can be any logical description from which
    - It is efficient to determine the truth of every ground atomic formula.
    - A domain closure axiom follows

$$\forall x.x = c_1 \vee x = c_2 \vee \cdots \vee x = c_n$$

    - The inequality of all constants (unique names axioms) follows.
  - These properties are required to support various procedures used in planning algorithms.

Representing the Initial state

- This implies that it is possible to efficiently determine in the initial state the truth of any first-order sentence.
- More generally, given any first-order formula (with free variables) it is possible to determine the set of instantiations of these variables (with constants) that satisfy the formula in the initial state.
- It also implies that it is possible to represent the initial state as a set of propositions.

Representing the Initial state

- Notice that the relational representation can be much more compact than a propositional representation.
- E.g., in the standard blocks world with 500 blocks in the initial state, there are about 250,000 possible `on(x,y)` relations.
  - 25K byte bit vector (40,000 states = 1GB)
  - A block can only be on one other block so only 500 possible `on(x,y)` relations in a database.

The Goal

- Representation of the goal
  - A set of ground literals
  - A state satisfies the goal if it satisfies all literals in the goal
  - Other possibilities
    - A more complex condition on a state, specified with a first-order formula.
    - A condition on the sequence of states visited by the plan (a “Temporally Extended Goal”)
    - The difficulty here lies in creating methods for effectively searching for plans satisfying these more complex goals.

Representation of Operators

- Operators specify the possible state transitions for a planning domain.
- For any particular planning problem, it is necessary to use the initial state as well as the operator specification to determine the transitions possible for this particular problem (the actions).
- To be problem independent operators use parameters.

Representation of Operators

- Since operators are domain specific, problem independent
  - Possible to develop methods that compute properties of transitions that apply to all possible problems in the domain.
  - The representation is more compact, independent of the size of the particular planning problem.

STRIPS Representation

- STRIPS is the simplest and the second oldest representation of operators in AI.
- When that the initial state is represented by a database of positive facts, STRIPS can be viewed as being simply a way of specifying an update to this database.

STRIPS Representation

```
(def-strips-operator (pickup ?x)
   (pre (handempty) (clear ?x) (ontable ?x))
   (add (holding ?x))
   (del (handempty) (clear ?x) (ontable ?x)))
```

STRIPS Representation

```
(def-strips-operator (pickup ?x)
   (pre (handempty) (clear ?x) (ontable ?x))
   (add (holding ?x))
   (del (handempty) (clear ?x) (ontable ?x)))
```

operator name and parameters

STRIPS Representation

```
(def-strips-operator (pickup ?x)
    (pre (handempty) (clear ?x) (ontable ?x))
    (add (holding ?x))
    (del (handempty) (clear ?x) (ontable ?x)))
```

List of predicates that must hold in the current state for the action to be applicable

STRIPS Representation

```
(def-strips-operator (pickup ?x)
   (pre (handempty) (clear ?x) (ontable ?x))
   (add (holding ?x))
   List of predicates that must be true
   in the next state
   (del (handempty) (clear ?x) (ontable ?x)))
```

STRIPS Representation

```
(def-strips-operator (pickup ?x)
   (pre (handempty) (clear ?x) (ontable ?x))
   (add (holding ?x))
   (del (handempty) (clear ?x) (ontable ?x)))
```

List of predicates that must be false in the next state

STRIPS Representation

- Given the initial state
  - All instantiations of the parameter `?x` that satisfy the precondition

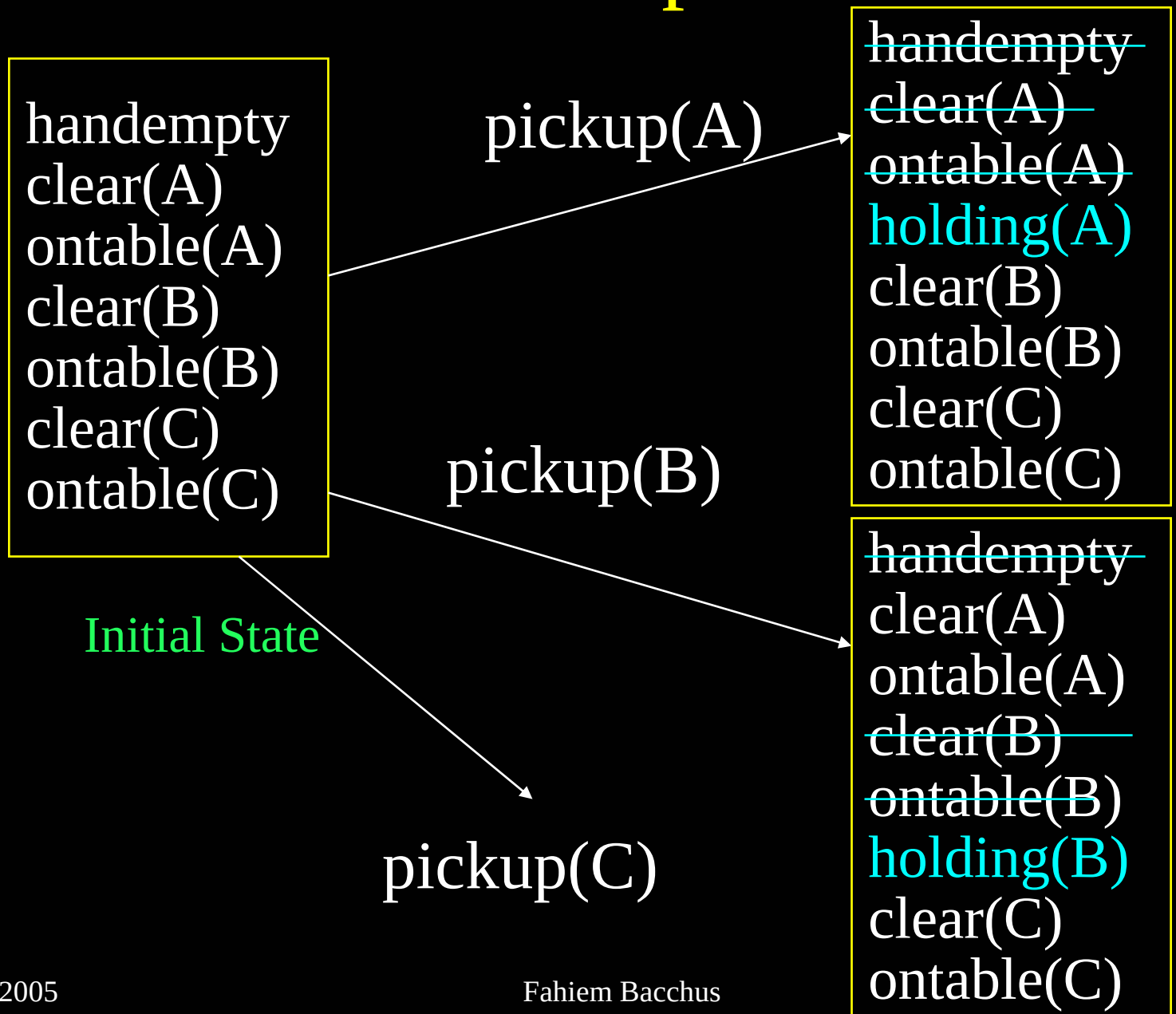
    ```
    (and (handempty) (clear ?x) (ontable ?x))
    ```

    produce a different action (transition) that can be applied to the initial state.
  - Actions whose preconditions are not satisfied are not legal transitions.

STRIPS Representation

- Actions are deterministic:
  - Given a particular instantiation of the parameters, the action specifies a finite collection of ground atomic formulas that must be made true and another collection that must be made false in the successor state.
- Nothing else is changed! (This has many algorithmic consequences).

STRIPS Representation



STRIPS Representation

- The properties of the initial state and the operators imply that from a finite collection of operators it is possible to determine
  - the finite collection of actions that can be applied to the initial state.
  - In each successor state generated by these actions, we can once again evaluate all logical formulas, and thus once again determine the set of all applicable actions.

STRIPS Representation

- Hence, we can incrementally generate the set of all states reachable from the initial state by sequences of actions.
- This is the forward space, and we can search for plans in this space. Later we will examine methods for accomplishing this.

ADL Representation

- Action description language due to Pednault.
- Generalizes STRIPS to allow for
  - Arbitrary first-order preconditions
  - Conditional effects
  - Universal effects
  - Functions

ADL Representation

```
(def-adl-operator (move ?x ?old ?new)
  (pre (and (on ?x ?old) (not (?old = ?new))
            (not (exists (?z) (on ?z ?x)))
            (not (exists (?z) (on ?z ?new)))))
  (add (on ?x ?new))
  (del (on ?x ?old))
  (forall (?z)
     (implies (above ?x ?z) (del (above ?x ?z))))
  (forall (?z)
     (implies (above ?new ?z) (add (above ?x ?z)))))
```

ADL Representation

```
(def-adl-operator (move ?x ?old ?new)
  (pre (and (on ?x ?old) (not (?old = ?new))
            (not (exists (?z) (on ?z ?x)))
            (not (exists (?z) (on ?z ?new)))))
  First-order preconditions.

  (add (on ?x ?new))
  (del (on ?x ?old))
  (forall (?z)
     (implies (above ?x ?z) (del (above ?x ?z))))
  (forall (?z)
     (implies (above ?new ?z) (add (above ?x ?z)))))
```

ADL Representation

```
(def-adl-operator (move ?x ?old ?new)
  (pre (and (on ?x ?old) (not (?old = ?new))
            (not (exists (?z) (on ?z ?x)))
            (not (exists (?z) (on ?z ?new)))))
  (add (on ?x ?new))
  (del (on ?x ?old))
  (forall (?z)
     (implies (above ?x ?z) (del (above ?x ?z))))
  (forall (?z)
     (implies (above ?new ?z) (add (above ?x ?z)))))
```

Conditional effects where quantification can be used to specify the set of atomic updates.

ADL Representation

- As in STRIPS
  - Every action specifies a finite collection of ground atomic formulas that must be made true and another collection that must be made false in the successor state
  - Nothing else changes.
- Given the completeness properties of the initial state
  - it is still possible to compute all applicable actions, and the effects of these actions.
  - All successors states have the same completeness properties.

ADL Representation

- So, it is remains possible to generate and search the forward space with ADL actions.
- ADL actions do pose some additional complexities for alternate search spaces.
  - One approach is to compile all ADL actions into a set of STRIPS actions.
    - Can yield an exponential number of STRIPS actions [Nebel, 2000]
  - An alternative is to develop techniques for dealing directly with ADL actions (perhaps with some restrictions) in these alternate search spaces
    - UCPOP, ADL for searching the space of partially ordered plans. [Penberthy & Weld, 1992]

Situation Calculus

- This is the earliest method in AI for representing actions.
- Abandoned in favor of STRIPS due to the inefficiency of planning with this representation.
- More recently Reiter has demonstrated that with domain specify search control knowledge [Bacchus & Kabanza 1995] powerful planners can be constructed for the situation calculus. These planners are in the same ballpark of efficiency as competitive approaches.

Situation Calculus

- A first-order language in which each predicate and function that can be modified by actions (fluents), takes an extra situation argument

on(A,B,s)	A is on B in situation s
weight(Fred,s)=150	Fred's weight in situation s

Situation Calculus

- Actions are objects: to facilitate quantification over them.
- There is a generic "do" function which applies an action to a situation to yield a new situation

do(pickup(A),s) the situation that arises from applying the action "pickup(A)" to situation s.

weight(Fred,do(DagstuhlSeminar,s))=???

Situation Calculus

- Actions are specified by first-order formulas
  - a precondition formula

$$\forall x, s.Poss(pickup(x), s) \equiv$$
$$\forall z.\neg holding(z, s) \wedge \neg heavy(x) \wedge nextTo(x, s)$$

  - effect axioms

$$\forall x, s.holding(x, do(pickup(x), s))$$

$$\forall x, l_1, l_2, s.\ fuel(x, do(drive(x, l_1, l_2), s))$$
$$= fuel(x, s) - distance(l_1, l_2) \div mpg(x)$$

Situation Calculus

- By making an
  - Explanation Closure assumption (fluents do not change value from situation to situation unless an action occurred that affected them)
  - Action Closure assumption (the described set of actions are the only actions and these actions have no other effects than those described).
- It is possible to automatically convert the set of effect axioms into successor state axioms, one for every fluent.

Situation Calculus

- These are axioms of the form

$$\forall x_1, \ldots, x_n, s.P(x_1, \ldots, x_n, do(a, s)) \equiv \gamma(x_1, \ldots, x_n, s)$$

$$\forall x_1, \ldots, x_n, y, s.f(x_1, \ldots, x_n, do(a, s)) = y \equiv \gamma(x_1, \ldots, x_n, y, s)$$

  - Where γ is a formula that mentions the previous situation only (and the arguments of the fluent).

$$\forall x, s.holding(x, do(a, s)) \equiv$$

$$holding(x, s) \wedge a \neq putdown(x) \vee a = pickup(x)$$

- This approach subsumes both the STRIPS and ADL representations.

Situation Calculus

- We can query any state S that arises from a sequence of actions (applied to the initial state) by asking an equivalent query of the initial state. In particular,
  - We can query S to see if an actions preconditions hold.
  - We can query S to see if the goal holds.
- The successor state axioms can be used to convert these queries to equivalent queries about the initial state.

Situation Calculus

- This conversion process is called regression
  - Given property P and action A, what is the property Q that must hold now so that after executing A, P holds.
  - With successor state axions regression is efficient, it is purely syntactic rewriting.
- Modulo the growth in complexity of the queries due to regression, whether or not we can answer the regressed query in the initial state depends on the properties of the theory describing the initial state (e.g., completeness).

Situation Calculus

- Hence, if we can query the initial state efficiently we can search for plans by examining sequences of actions applied to the initial state.
  - This is the forward search space.
- The restrictions we placed on the initial state are an example of conditions under which querying the initial state is efficient.

Functions

- Functions are allowed in ADL, and it is still possible to search the forward space as long as in the initial state we have complete knowledge of all function values (implicitly or explicitly), for other spaces we must develop special techniques, or make extra restrictions.

```
(def-adl-operator (store-in-room ?x ?r)
   (pre (> (load-capacity (floor-material ?r))
           (weight ?x)))
   (update (storage-capacity ?r)
           (- (storage-capacity ?r)
              (base-area ?x))))
```

Features of Action Representations

- Actions cause modular updates, they affect only a (generally) small set of predicates and a small set objects.
- The frame assumption (that most things are unchanged) is built into these representations.
- Specified in a parameterized manner
  - compact specification of many different actions
  - captures structural commonalties between sets of related actions.
- These features all play a role in the search techniques developed in planning.

Searching for Plans

- We have described the action representations in terms of the transitions they generate.
- One can search in the space of action sequences applied to the initial world.
- The states that arise from such sequences can be computed or queried.
  - achievement of the goal can be tested
  - satisfaction of preconditions can be tested (thus the set of actions applicable to that state can be determined)
- This is the forward search space.

Searching for Plans

- We will return to methods for searching the forward space.
- There are other types of spaces over which plans can be searched for.
- Much of the work in planning has been devoted to developing methods for searching such alternate spaces.
- Now we will describe some of the spaces that can be searched for plans.

Propositional Spaces

- Under domain closure all possible states and all possible actions can be represented by a collection of propositions: each possible instantiation of the predicates and each possible instantiation of the operators.
- In order to represent the search space with propositions, we simply need to impose a fixed bound on plan length.
- Since the length of the plan is unknown, we can incrementally increase the bound on length, at each state doing search in the resulting propositional space.

Propositional Spaces

State at T0	Action at T0	State at T1	
on(A,B,0)	pickup(A,0)	on(A,B,1)	
on(B,A,0)	pickup(B,0)	on(B,A,1)	
onTable(A,0)	putdown(A,0)	on(A,Table,1)	
onTable(B,0)	putdown(B,0)	on(B,Table,1)	
clear(A,0)	stack(A,B,0)	clear(A,1)	
clear(B,0)	stack(B,A,0)	clear(B,1)	...
handempty(0)	unstack(A,B,0)	handempty(1)	
holding(A,0)	unstack(B,A,0)	holding(A,1)	
holding(B,0)		holding(B,1)	
...		...	...

Simplest idea—a set of propositions to specify each of the k-states and k-actions taken in the k-step plan.

Now specify the conditions required to make this a k-step plan.

Propositional Spaces

```
on(A,B,0)
on(B,A,0)
onTable(A,0)
onTable(B,0)
clear(A,0)
clear(B,0)
handempty(0)
holding(A,0)
holding(B,0)
```

```
pickup(A,0)
pickup(B,0)
putdown(A,0)
putdown(B,0)
stack(A,B,0)
stack(B,A,0)
unstack(A,B,0)
unstack(B,A,0)
```

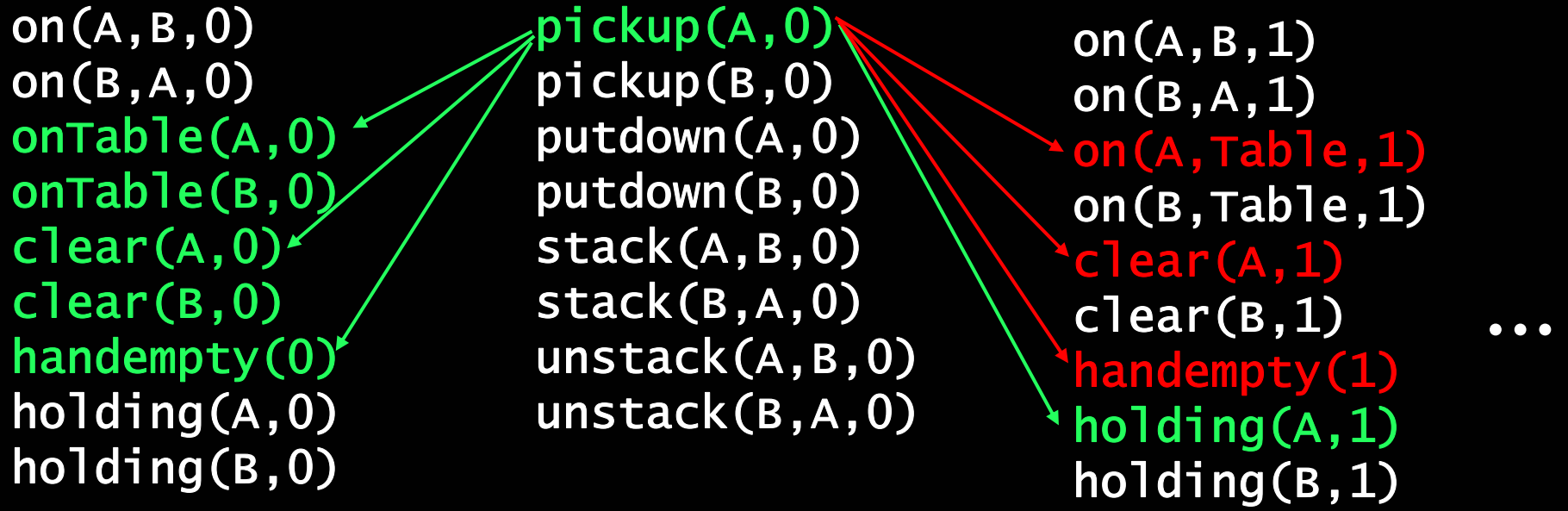
```
on(A,B,1)
on(B,A,1)
on(A,Table,1)
on(B,Table,1)
clear(A,1)
clear(B,1)
handempty(1)
holding(A,1)
holding(B,1)
```

...

State at T0

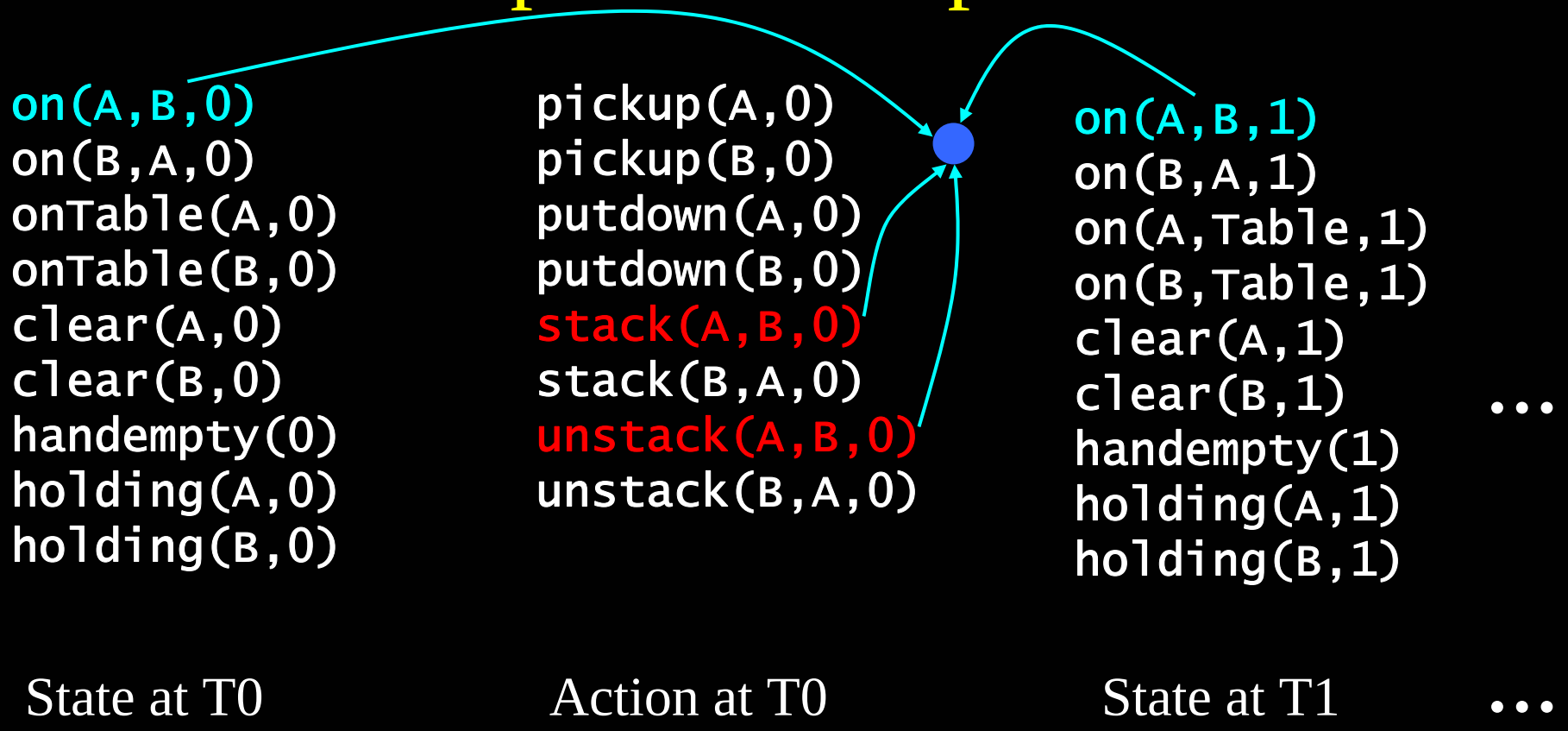
Initial state forces some propositions to be true and the others false. The goal forces some propositions at step k to be true.

Propositional Spaces



If an action is true, its preconditions must be true and its add effects must be true its delete effects must be false. (Easy for STRIPS, harder for ADL)

Propositional Spaces



Since the set of changes caused by an action are specified, a proposition cannot change its value unless it is changed by an action.

Propositional Spaces

State at T0	Action at T0	State at T1	
on(A,B,0)	pickup(A,0)	on(A,B,1)	
on(B,A,0)	pickup(B,0)	on(B,A,1)	
onTable(A,0)	putdown(A,0)	on(A,Table,1)	
onTable(B,0)	putdown(B,0)	on(B,Table,1)	
clear(A,0)	stack(A,B,0)	clear(A,1)	
clear(B,0)	stack(B,A,0)	clear(B,1)	...
handempty(0)	unstack(A,B,0)	handempty(1)	
holding(A,0)	unstack(B,A,0)	holding(A,1)	
holding(B,0)		holding(B,1)	
			...

Only one action can occur at each time.

Propositional Spaces

- These conditions are encoded as clauses.
- Then we solve with standard SAT solvers: the state of the action variables at each time step in the solution specifies the plan.
  - Fast for smaller problems, but the size of the SAT problem grows as a high order polynomial.
  - A = O(|Ops||Dom|$^{\text{Arity(Ops)}}$) --- number of actions
  - need O(nA^2) clauses [Kautz, McAllester, Selman, 1996]
  - E.g. binary actions like *stack(x,y)*, O(|Dom|^4)
  - A 19 block problem in the blocks world generates over 17,000,000 clauses and 40,000 variables.

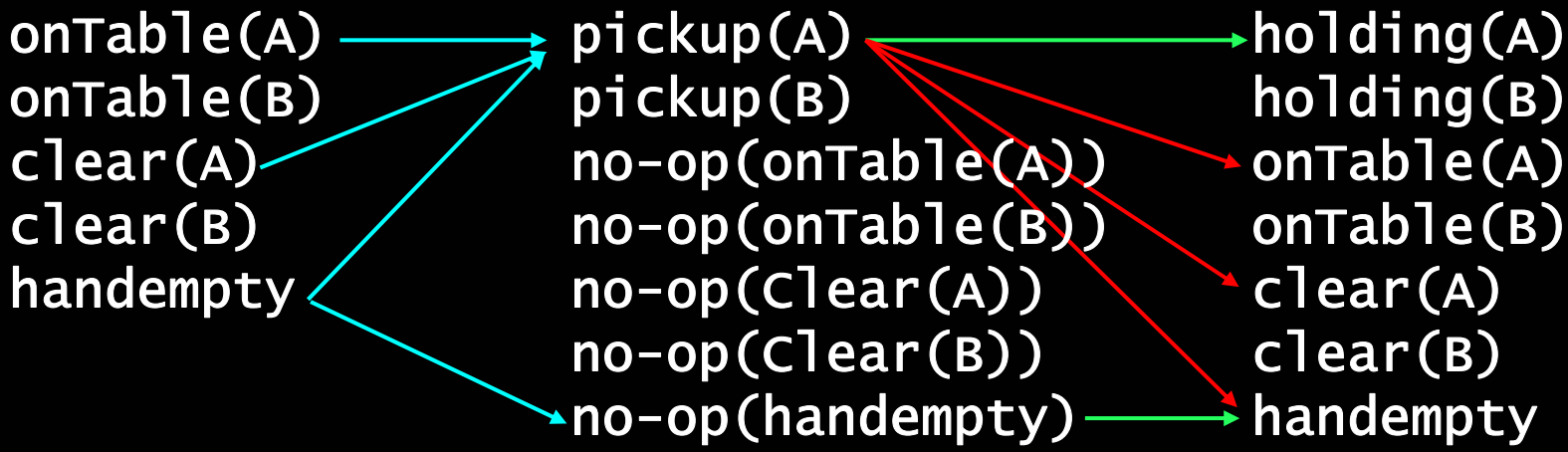
Constrained Propositional Spaces

- A standard technique in Constraint Programming is to make a problem easier to solve by enforcing various types of local consistency.
- This may reduce the size of the problem, or might increase the number of constraints (these can increase the efficiency of backtracking algorithms like Davis Putnam).
- Often insights from the problem are used to determine which local consistency properties are most worthwhile enforcing.

Graphplan

- Graphplan [Blum & Furst, 1995] is an instance of this general approach.
- Utilizes the structure of the updates generated by actions to build a more constrained propositional representation.
- Builds a leveled graph with an alternation between levels containing propositional nodes and levels containing action nodes.
- Three types of edges: precondition-edges, add-edges, and delete-edges.

Graphplan



Initial state

Only the propositions true in the initial state.

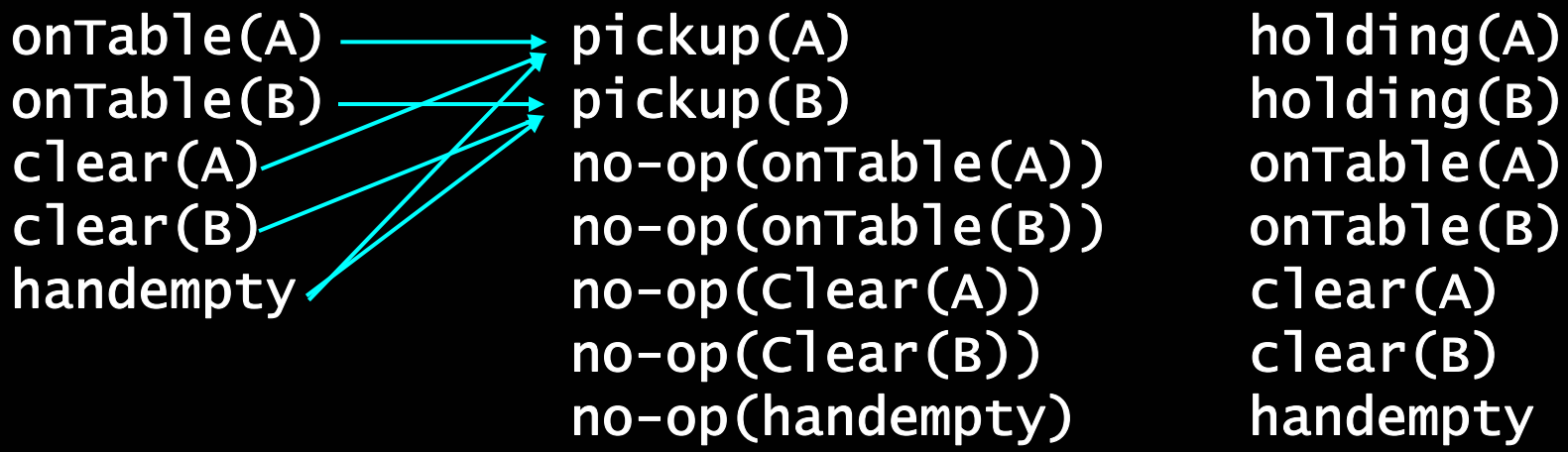
Possible actions

Only the actions whose preconditions are in the previous level.

Also have no-ops for capturing non-changes.

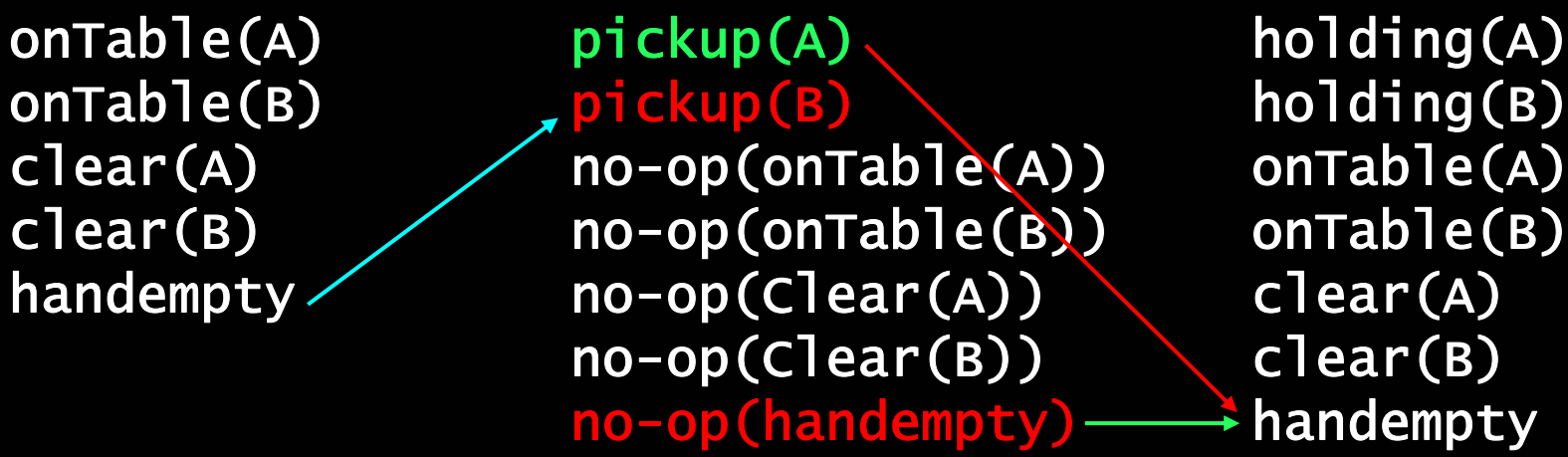
All propositions added by actions in previous level

Graphplan



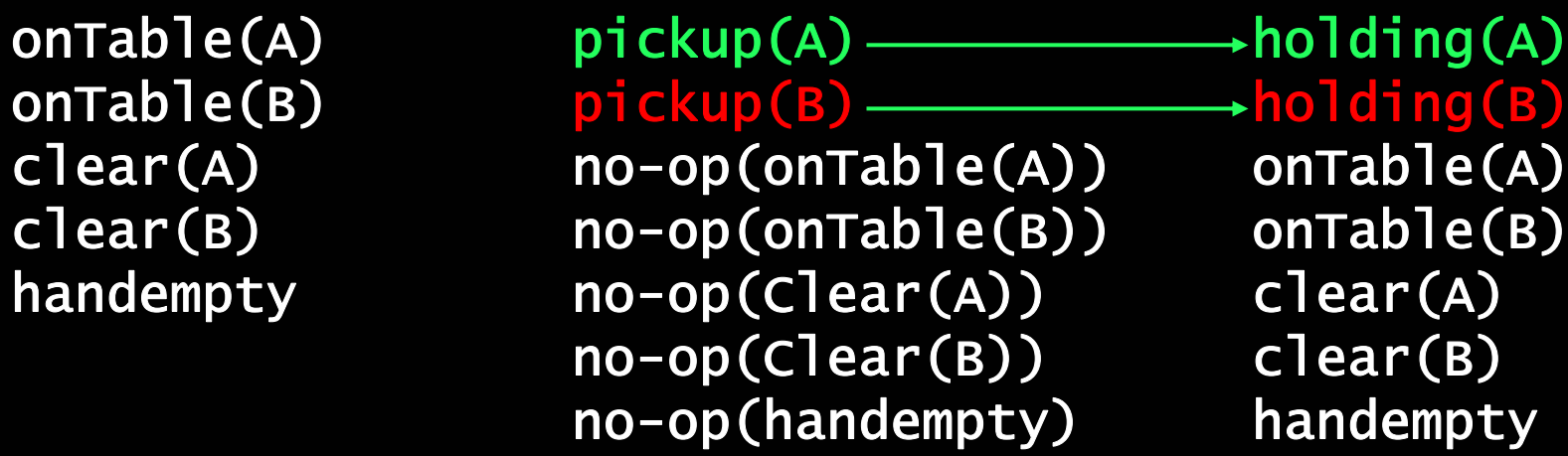
- More than one action is allowed to occur at each level: a limited form of concurrency.
- Enforces a level of local consistency by detecting and propagating mutual exclusivity relations (Mutex).

Graphplan



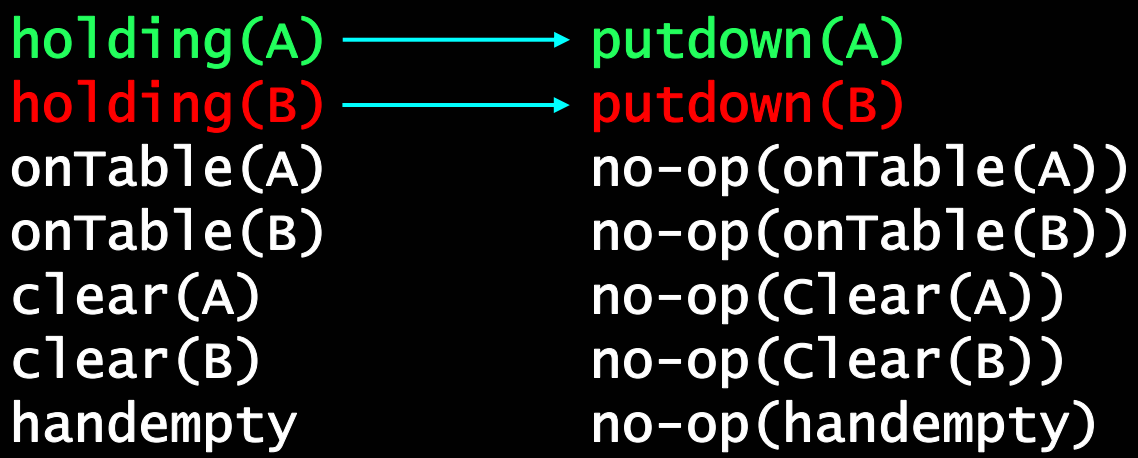
- Two actions are mutex if either action deletes a precondition or add effect of another.

Graphplan



- Two propositions *p* and *q* are mutex if all actions adding *p* are mutex of all actions adding *q*

Graphplan



- Two actions are mutex if two of their preconditions are mutex.

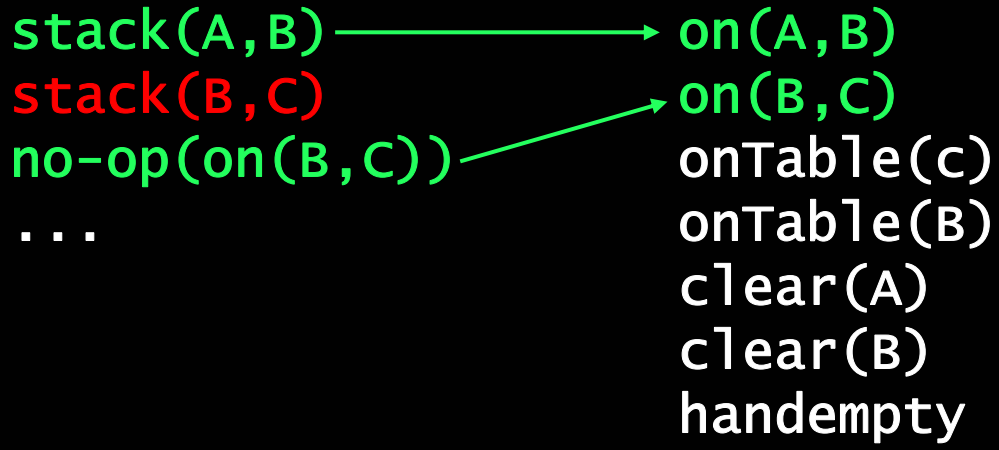
Searching the Graphplan

on(A,B)
on(B,C)
onTable(C)
onTable(B)
clear(A)
clear(B)
handempty

K

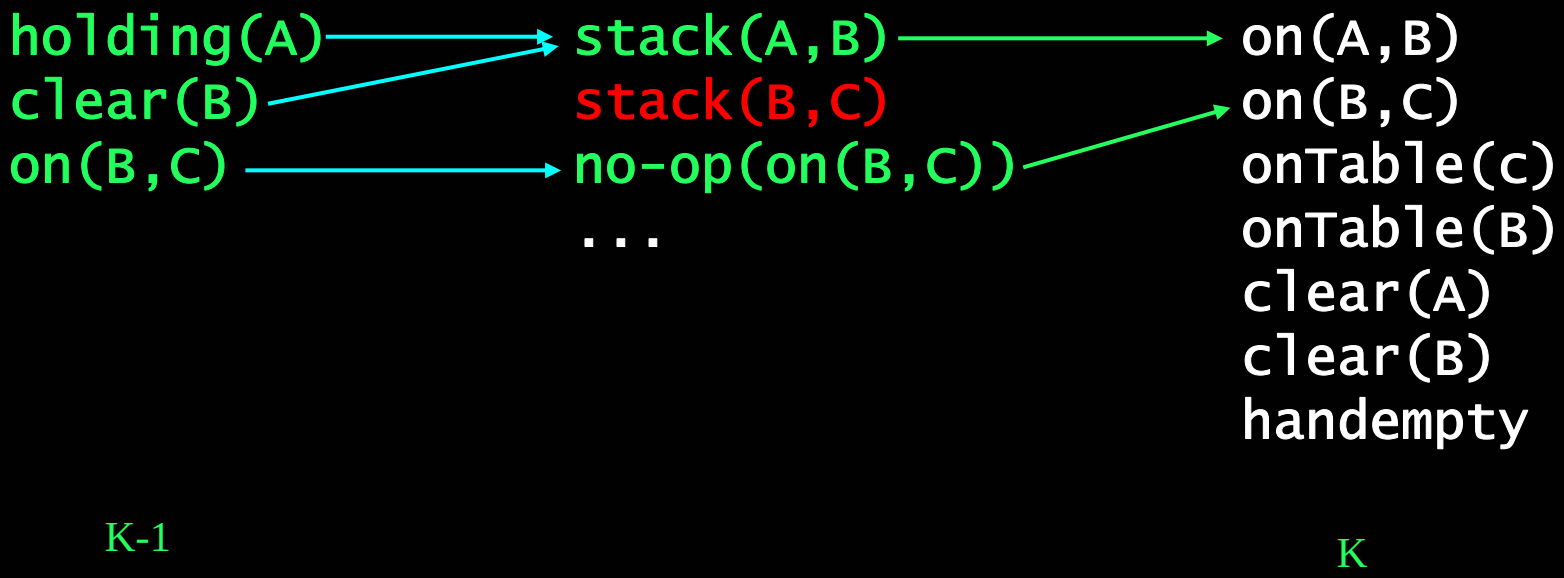
Build the graph to level k, such that every member of the goal is present at level k, and no two are mutex.

Searching the Graphplan



Find a non-mutex collection of actions supporting each goal.

Searching the Graphplan



The preconditions of these actions at level K-1 become the new goal.
We must check that these new goals are all members of the proposition level k-1 and are all non-mutex.

Searching the Graphplan

- This particular search procedure is limited to STRIPS (ADL operators must be translated to STRIPS).
- Current status:
  - Fast for smaller problems.
  - The mutex constraints greatly enhance the efficiency of search.
  - The graph does not grow as fast as the simple sat translation
  - But it is still growing super-linearly, and time to search is grows exponentially in its size.

Enhancements

- Other techniques have been developed for discovering additional mutex information.
- TIM [Fox & Long 1998], and DISCOPLAN [Gerevini & Schubert 2000] are two approaches to discovering additional state constraints, e.g., an object can't be in two locations at the same time.
- Such information can be used to impose additional constraints that can be used during planning
  - e.g., a goal containing at(A,L1) and at(A,L2) is an immediate failure.

“Digression” on CSPs

- It is useful to frame the Graphplan techniques in terms of Constraint programming as this provides a general view of this technique for speeding up search.

CSPs

A Constraint Satisfaction Problem consists of

1. A set of variables $\{V_1,...,V_n\}$
2. A domain of values for each variable $Dom[V_i]$
3. A set of constraints $\{C_1,...,C_m\}$
   - Each constraint is over some set of *k* variables
   - *k* is the arity of the constraint
   - The constraint is a mapping from an assignment to each of the *k* variables it is over to true/false. True means that the assignment satisfies the constraint.

Problem is to find an assignment of values to variables which satisfies all the constraints.

CSPs

- The general problem is NP-hard, but various forms of local constraint propagation are possible in polytime.
- A set of *i* variables $\mathcal{V}$ is *(i,j)-consistent* with respect to another set of *j* variables $\mathcal{U}$ and a set of constraints $\mathcal{C}$ (over the variables in $\mathcal{V}$ and $\mathcal{U}$) if
  - every assignment of the variables in $\mathcal{V}$ that satisfies all the constraints over $\mathcal{V}$ can be extended to a set of assignments over $\mathcal{U}$ that satisfies all of the constraints in $\mathcal{C}$.

CSPs

- E.g.,
  - $\mathcal{V} = \{V_1, V_2\}$, with a constraint $C_1(V_1, V_2)$ between V_1 and V_2.
  - $\mathcal{U} = \{V_3, V_4\}$ and $\mathcal{C} = \{C_2(V_1, V_2, V_3, V_4,)\}$
- Then (2,4) consistency holds if every pair of assignments to V_1 and V_2 that satisfies C_1, can be extended to an assignment to V_3, and V_4 that satisfies C_2.

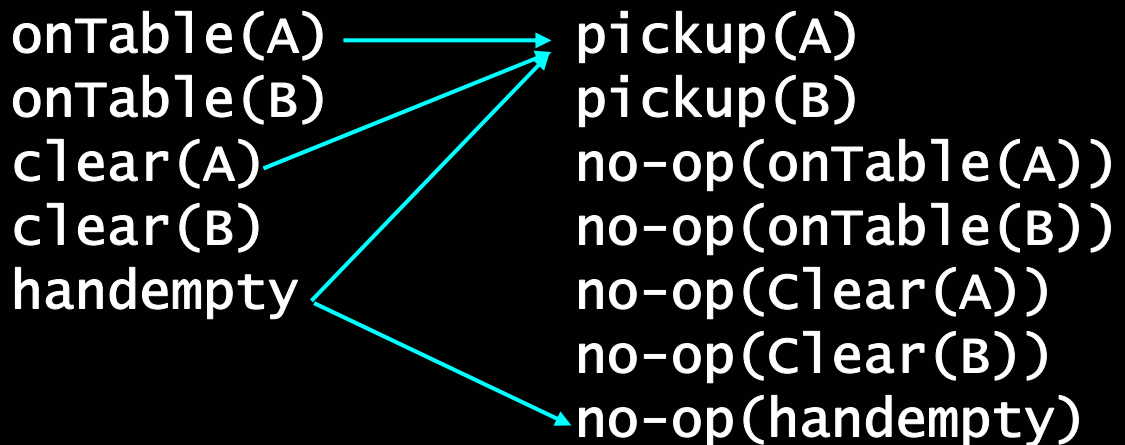
CSPs

- If a CSP is not *(i,j)* consistent (with respect to a particular $\mathcal{V}$, $\mathcal{U}$, and $\mathcal{C}$) then it can be made consistent by imposing a new constraint over $\mathcal{V}$.
- E.g.,
  - *(1,j)* consistency is achieved by deleting values from the domain of the variable in $\mathcal{V}$.
  - *(2,j)* is achieved by imposing a binary constraint over the two variables in $\mathcal{V}$.
- If D is the size of the largest domain of values, any (i,j) consistency can be achieved in $O(D^{i+j})$.

CSPs

- By enforcing various (i,j) consistencies one hopes to make the problem easier to solve (e.g., by allowing search to detect dead ends sooner).
- When modeling a problem as a CSP one uses insights into the problem and empirical data to choose which (i,j) consistencies are worth enforcing.
- Importantly the set of constraints C may have a special structure which allows consistency to be enforced much more efficiently.

CSPs and Graphplan



All propositions are binary variables.

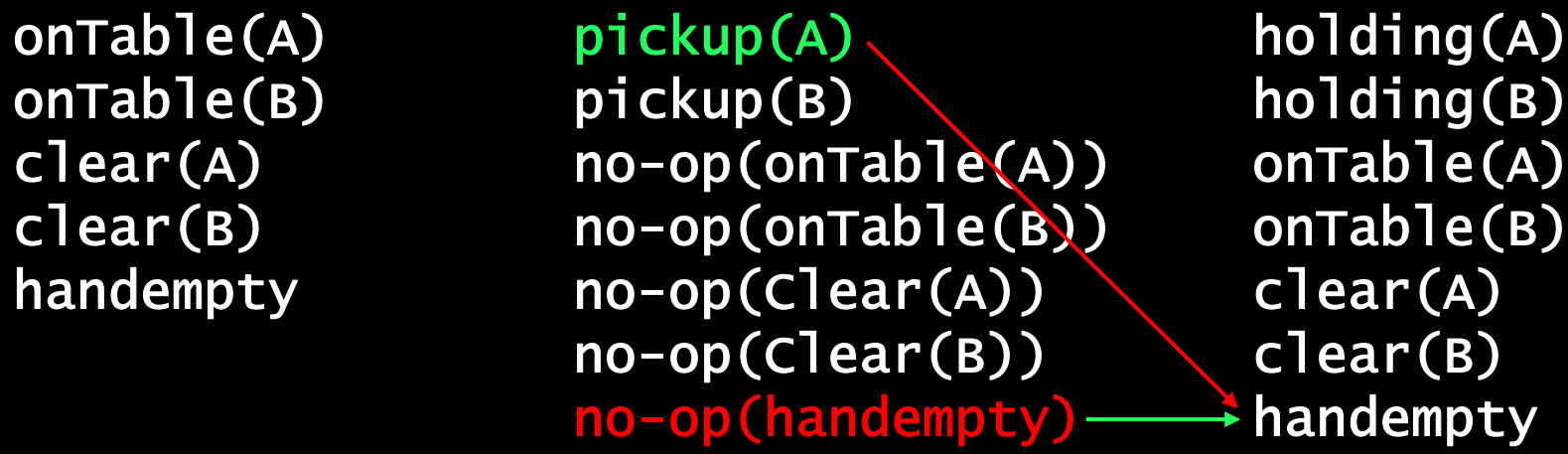
There is a constraint between every action and its set of preconditions.

Possible actions

Only the actions whose preconditions are in the previous level.

This is a (1,k) consistency with the action's preconditions.

CSPs and Graphplan



- An action that deletes the effect of another is mutex.
- This is a (2,1) consistency. For the pair of assignments `pickup(A)=True`, `no-op(handempty)=True` there is no possible assignment to `handempty`.

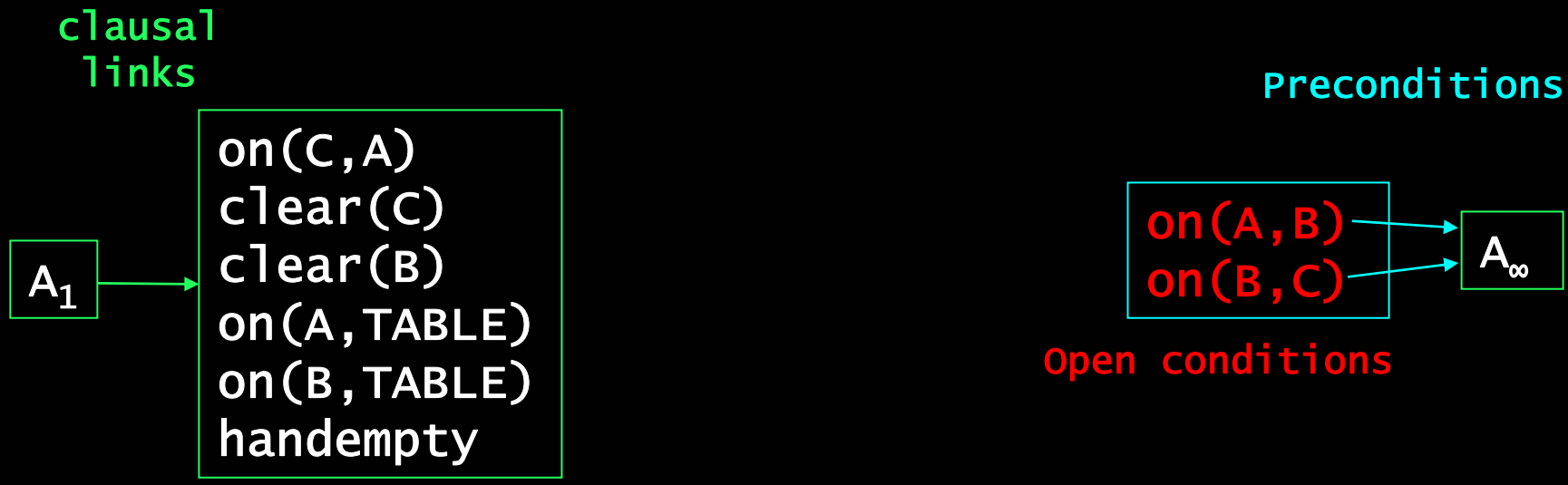
CSPs and Graphplan

- From this point of view it is possible to view the Graphplan construction as a process of constraint propagation.
- And it becomes possible to search for a solution using the much more powerful backtracking search techniques developed in constraint programming (including no-good recording, and intelligent backtracking).
- An approximation to this is [Do & Kambhampati 2000].
- CPLAN [Chen & van Beek] is an approach that uses domain specific constraints (normal in constraint programming).

Plan Space

- Another space in which plans can be searched for is the space of partially ordered plans.
- Again the insight is that the only way a property can be changed is by an action that mentions that property among its effects.
- Thus to transform the initial state to a goal state, every change must be supported by an action, and every action must have all of its preconditions supported.

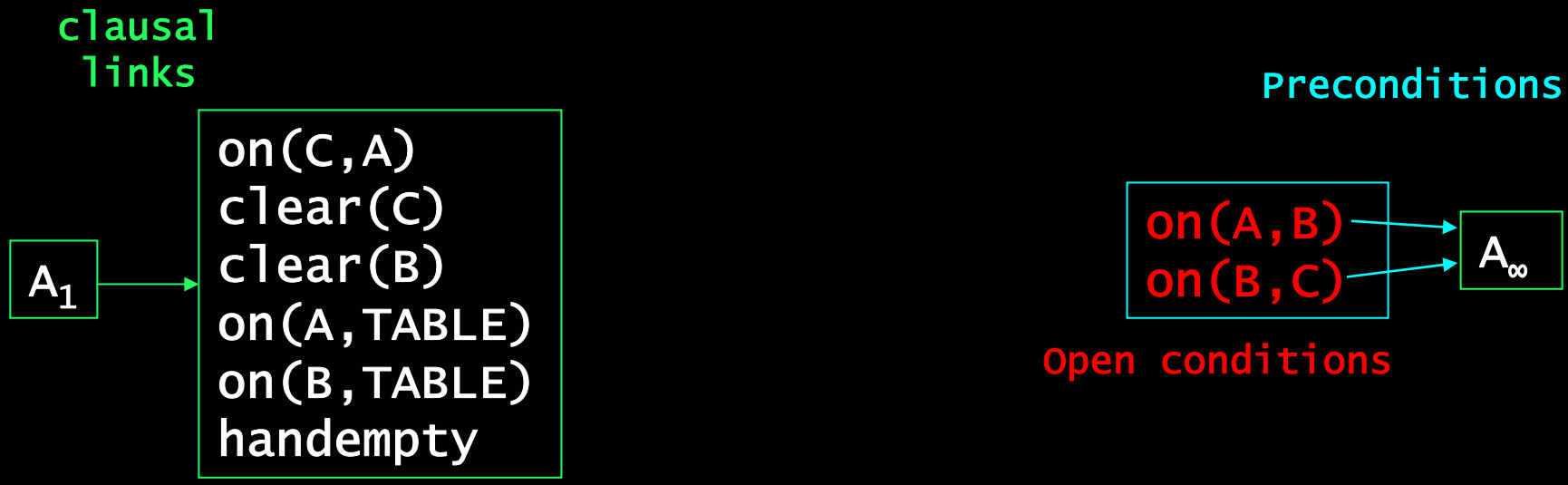
Plan Space



The initial plan has an initial action that "causes" all of the facts in the initial state, and a final action whose preconditions are all of the goals.

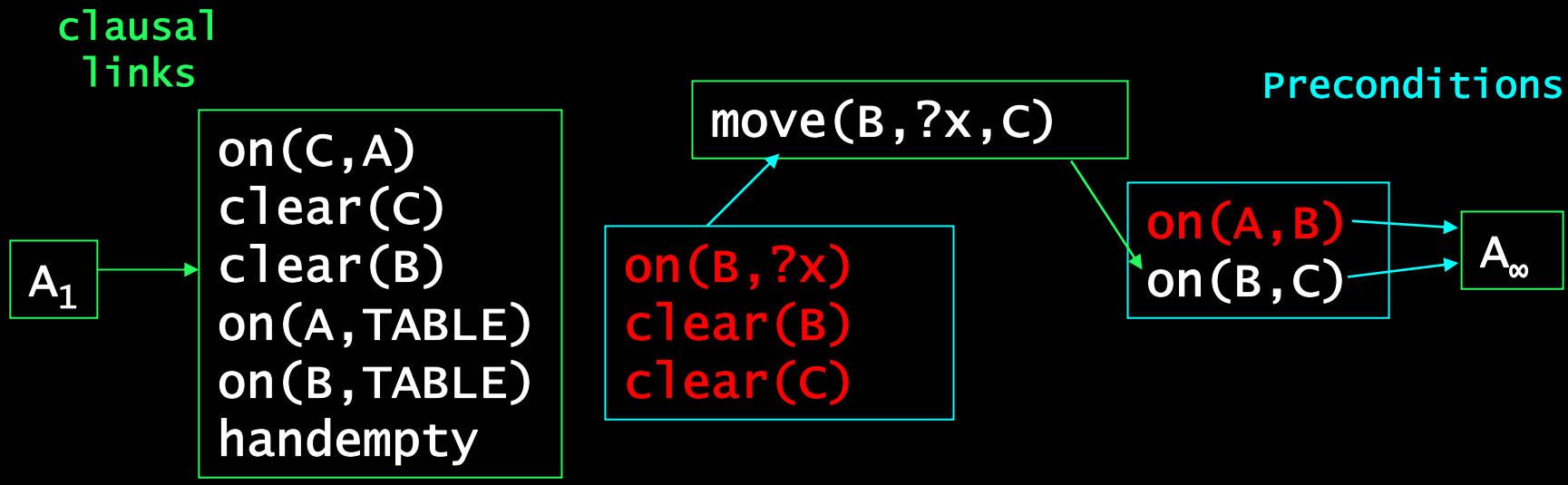
The open conditions are the unsupported facts.

Plan Space



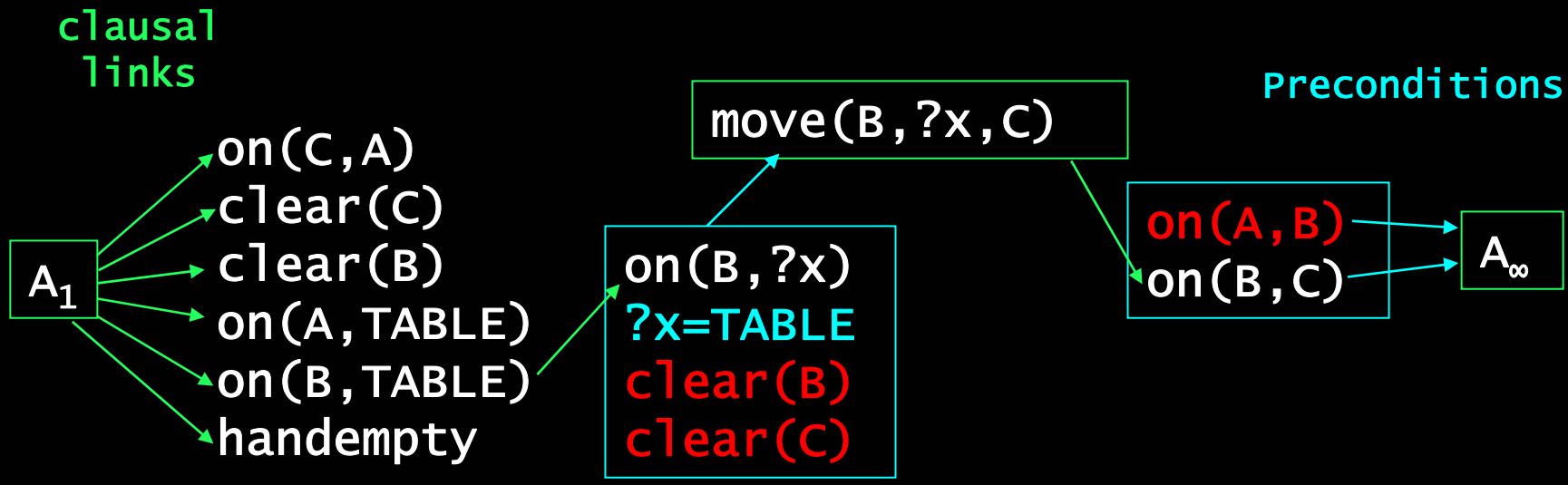
Something must cause every open condition, and search involves searching through the space of all possible ways open conditions can be supported.

Plan Space



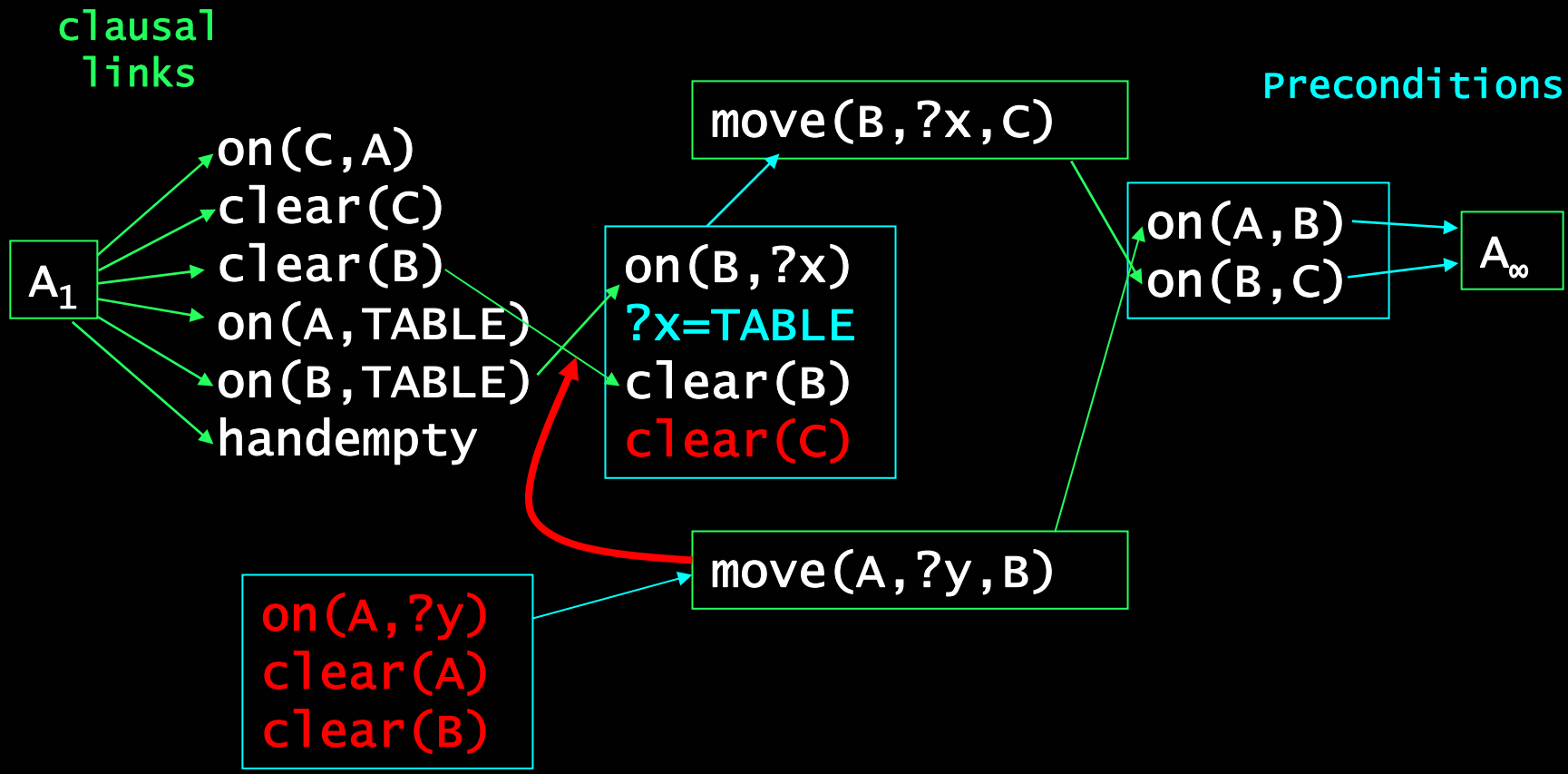
- An open condition can be supported by a new action.
  - note that not all of the operator's parameters need be bound to support the condition.
  - This corresponds to an entire set of possible ground actions.

Plan Space



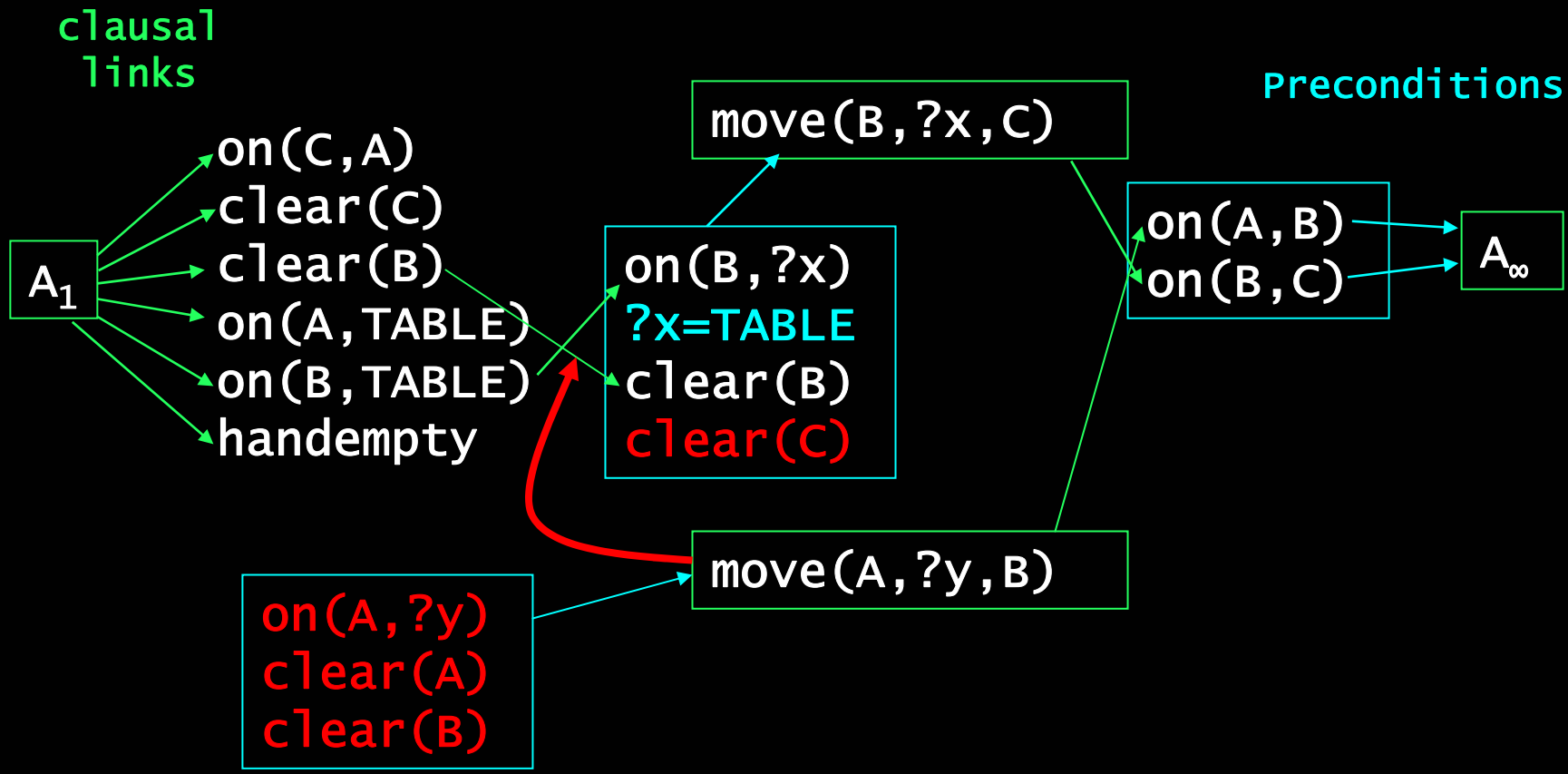
- Or it could be supported by the effect of an existing action.
  - Note that this might require committing to a binding for a variable.

Plan Space



- New actions can be unordered.
- But they might threaten existing causal links!

Plan Space



- Threats are resolved by imposing ordering constraints, or by imposing binding constraints (the threat might exist only under some bindings.

Plan Space

- Each possible way of supporting an open condition is a choice point in the space.
- Each possible way of resolving a threat is a choice point in the space.

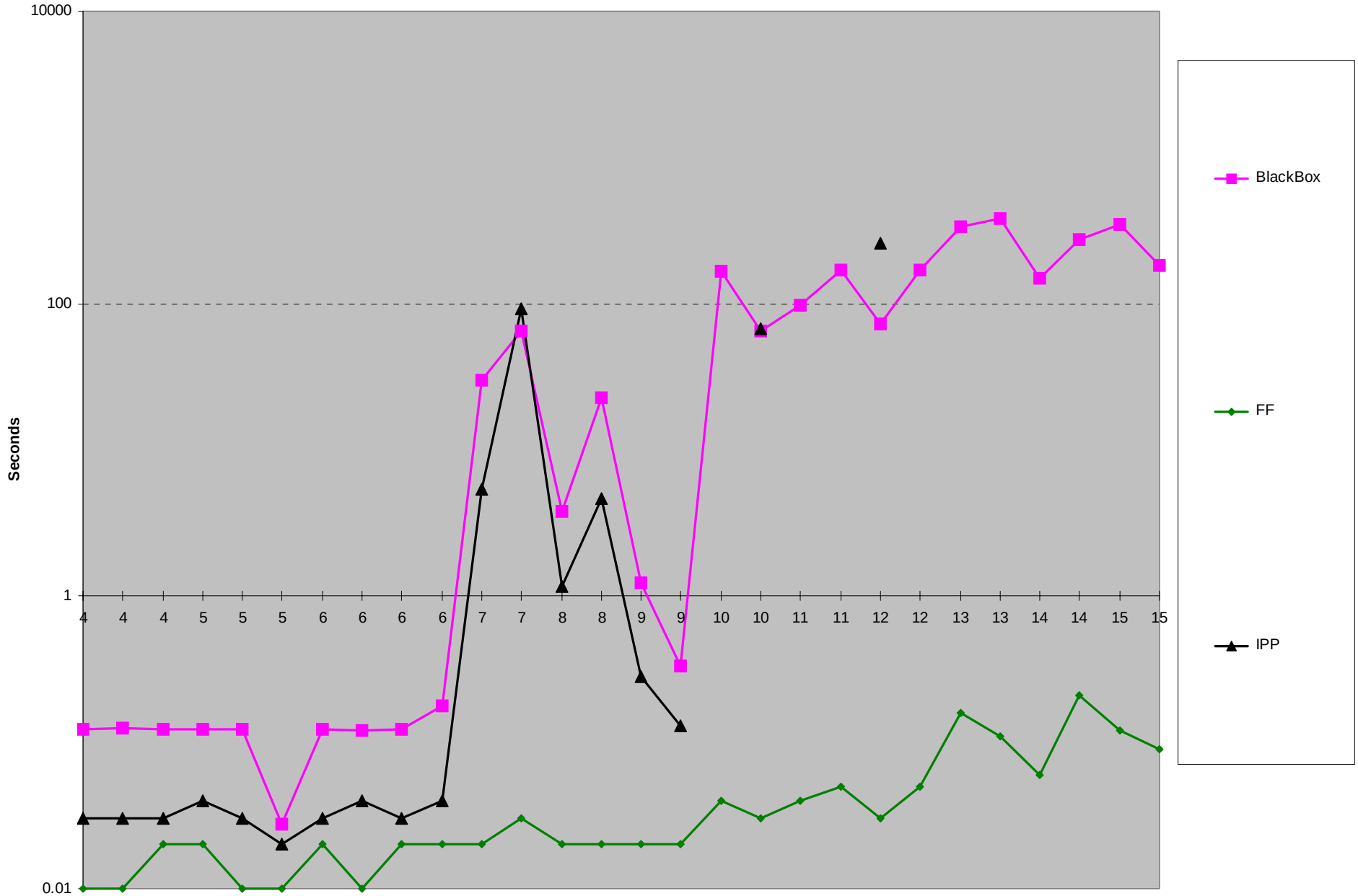
Plan Space

- Search in this space has a history be being quite inefficient.
  - many partial plans have no completions due to unsatisfiable ordering or binding constraints.
  - partial plans can be developed with actions that could never be reached from the initial state.
- Recently [Nguyen & Kambhampati, 2001] have shown that by using more extensive constraint propagation, and good distance heuristics the approach can be as effective as GraphPlan.
- What is very interesting about plan space search is that the points in the search space are partially ordered, parameterized plans.

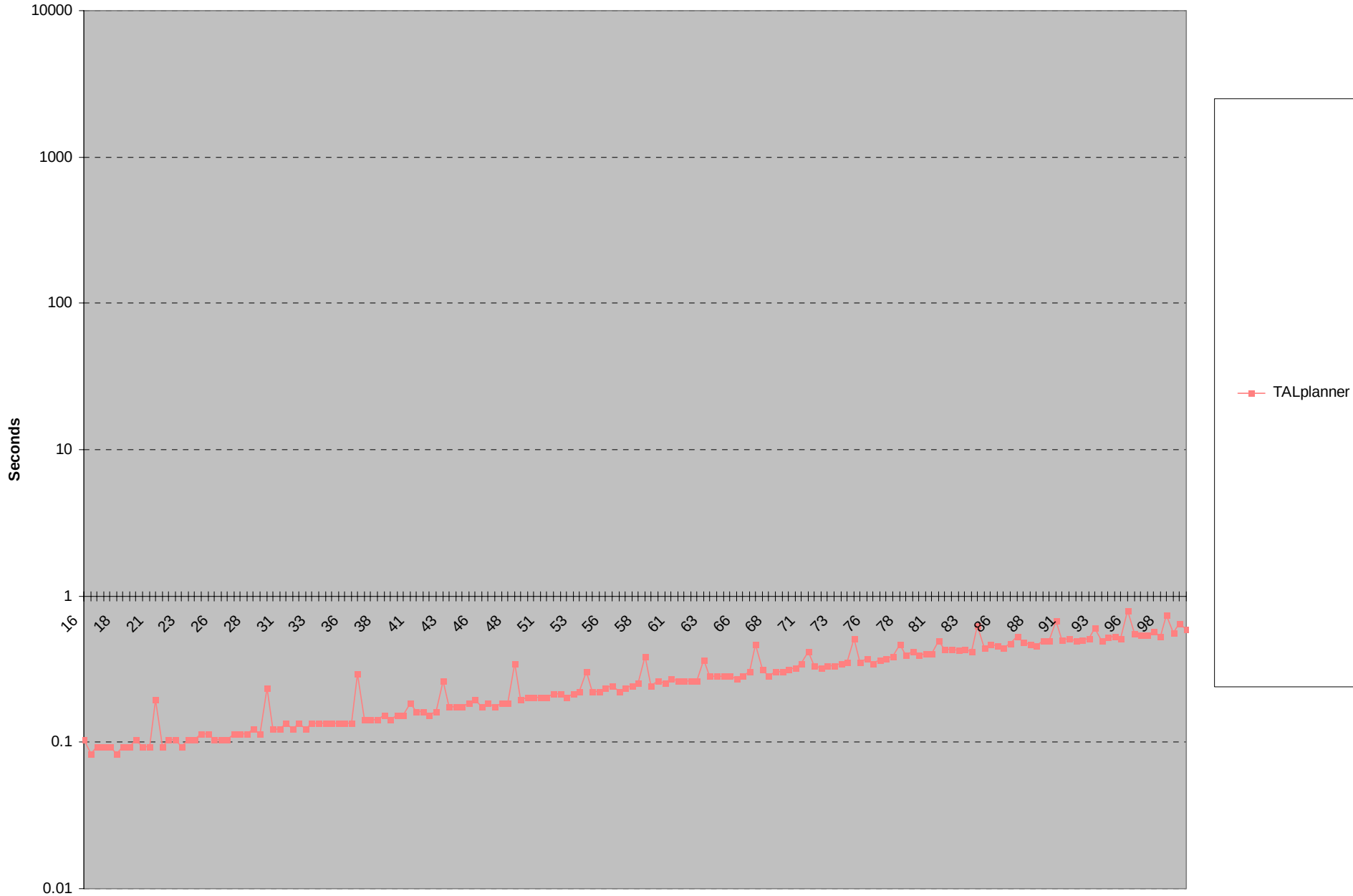
Forward Space

- The conceptually simplest search space is simply to directly apply actions to the initial state, searching for a state that satisfies the goal.
- The difficult has been guiding the search towards goal states.
- There are at least two very powerful ideas for accomplishing this
  - Distance heuristics computed from Graphplan like reachability analysis.
  - Domain specific search control expressed as properties all reasonable plans should have.
- Currently (AIPS2000 planning competition) these two approaches are the fastest and most scalable methods for finding plans.

Fully Automated Logistics Time Comparison



Hand Tailored Logisitic Time Comparison



Distance Heursitics

- A classical technique in AI is to compute a heuristic estimate of the distance to the goal.
- One guides search by first exploring those states that appear to be closer to the goal.
- A standard method for generating heuristics is to consider a relaxed version of the problem, measuring the distance to the goal in this relaxed version.
- A common relaxation in planning is to consider a problem in which the delete effects of all actions are removed.

Distance Heursitics

- E.g., FF (Fast Forward) [Hoffman 2001] evaluates the heuristic merit of a state by constructing a plan to transform that state to a goal state using the relaxed actions. The number of actions in this plan is an estimate of the distance to the goal.
- Give a set of actions without deletes, it becomes possible to find a plan in polynomial time (if one exists).
  - It remains hard (NP-complete) to find a plan with the minimal number of actions.
  - [Hoffman & Nebel 2001]

Distance Heursitics

- With a heuristic ranking of every state, there are various search strategies that can be employed.
- E.g., Fast Forward uses a greedy local search
  - Starting with the initial state always move to a successor state that has lower heuristics merit.
  - If no successor is better, FF uses a systematic search of the neighborhood to find a superior state that is only small number of actions away.
- Empirically FF's particular search strategy proves to very effective on the standard benchmark problems (e.g., those used in the AIPS2000 competition).

Distance Heuristics

- This heuristics is not admissible
  - Is always a lower bound on the actual distance to the goal.

  so it could not be used in a branch and bound search (A^*) to generate optimal solutions.
- Other heuristics including admissible heuristics, have been tried [Geffner & Haslum 2000], [Nguyen & Kambhampi 2000].

Distance Heuristics

- Distance Heuristics are also useful in partial order planning.
- In the propositional search spaces, when encoding the problem as a CSP, distance constraints can be used
  - These are constraints on the distance to various goal conditions, and were used in CPLAN [Chen & van Beek 1999]

Domain Specific Search Control

- In [Bacchus & Kabanza 1995] an approach was developed for using domain specific search control expressed as formulas of first-order linear temporal logic.
- This approach has proved to be very successful, and yields very fast planners including TLplan, and TALPlanner which was the fastest planner in the AIPS2000 competition (domain specific).
- How does this work? And what can it tell us about searching large spaces?
- First a concrete example.

Logistics World

- A simplified logistics problem
- Packages to be transported between locations.
- Two types of vehicles trucks and planes
- Trucks can move inside of a city.
- Planes can fly between airports.
- No resource bounds on how many objects a vehicle can transport at the same time.
- Restrict to the case where we are only concerned with generating plans containing reasonable small numbers of actions.

Logistics World

```
(def… (load ?package ?vehicle ?loc)
  (pre (at ?package ?loc) (at ?vehicle ?loc)
       (package ?package))
  (add (in ?package ?vehicle))
  (del (at ?package ?loc)))

(def… (unload ?package ?vehicle ?loc)
  (pre (in ?package ?vehicle) (at ?vehicle ?loc)
       (package ?package))
  (add (at ?package ?loc))
  (del (in ?package ?vehicle)))

(def… (drive ?truck ?loc1 ?loc2 ?city)
  (pre (at ?truck ?loc) (loc-at ?loc1 ?city)
       (loc-at ?loc2 ?city) (truck ?truck)
       (not (?loc1 = ?loc2))
  (add (at ?truck ?loc1))
  (del (at ?truck ?loc2)))

(def… (fly ?plane ?loc1 ?loc2)
 (pre (at ?plane ?loc1) (plane ?plane)
       (airport ?loc2) (?loc1 $\ne$ ?loc2))
 (add (at ?plane ?loc1))
 (del (at ?plane ?loc2)))
```

Logistics World

```
(def… (load ?package ?vehicle ?loc)
  (pre (at ?package ?loc) (at ?vehicle ?loc)
       (package ?package))
  (add (in ?package ?vehicle))
  (del (at ?package ?loc)))
```

Put an package in a truck or a plane if both are at the same location

```
(def… (unload ?package ?vehicle ?loc)
  (pre (in ?package ?vehicle) (at ?vehicle ?loc)
       (package ?package))
  (add (at ?package ?loc))
  (del (in ?package ?vehicle)))

(def… (drive ?truck ?loc1 ?loc2 ?city)
  (pre (at ?truck ?loc) (loc-at ?loc1 ?city)
       (loc-at ?loc2 ?city) (truck ?truck)
       (not (?loc1 = ?loc2))
  (add (at ?truck ?loc1))
  (del (at ?truck ?loc2)))

(def… (fly ?plane ?loc1 ?loc2)
 (pre (at ?plane ?loc1) (plane ?plane)
       (airport ?loc2) (?loc1 $\ne$ ?loc2))
 (add (at ?plane ?loc1))
 (del (at ?plane ?loc2)))
```

Logistics World

```
(def… (load ?package ?vehicle ?loc)
  (pre (at ?package ?loc) (at ?vehicle ?loc)
       (package ?package))
  (add (in ?package ?vehicle))
  (del (at ?package ?loc)))
```

```
(def… (unload ?package ?vehicle ?loc)
  (pre (in ?package ?vehicle) (at ?vehicle ?loc)
       (package ?package))
  (add (at ?package ?loc))
  (del (in ?package ?vehicle)))
```

Take an package out at the vehicle's current location.

```
(def… (drive ?truck ?loc1 ?loc2 ?city)
  (pre (at ?truck ?loc) (loc-at ?loc1 ?city)
       (loc-at ?loc2 ?city) (truck ?truck)
       (not (?loc1 = ?loc2))
  (add (at ?truck ?loc1))
  (del (at ?truck ?loc2)))

(def… (fly ?plane ?loc1 ?loc2)
 (pre (at ?plane ?loc1) (plane ?plane)
       (airport ?loc2) (?loc1 $\ne$ ?loc2))
 (add (at ?plane ?loc1))
 (del (at ?plane ?loc2)))
```

Logistics World

```
(def… (load ?package ?vehicle ?loc)
  (pre (at ?package ?loc) (at ?vehicle ?loc)
       (package ?package))
  (add (in ?package ?vehicle))
  (del (at ?package ?loc)))

(def… (unload ?package ?vehicle ?loc)
  (pre (in ?package ?vehicle) (at ?vehicle ?loc)
       (package ?package))
  (add (at ?package ?loc))
  (del (in ?package ?vehicle)))

(def… (drive ?truck ?loc1 ?loc2 ?city)
  (pre (at ?truck ?loc) (in-city ?loc1 ?city)
       (in-city ?loc2 ?city) (truck ?truck)
       (not (?loc1 = ?loc2))
  (add (at ?truck ?loc2))
  (del (at ?truck ?loc1)))

(def… (fly ?plane ?loc1 ?loc2)
 (pre (at ?plane ?loc1) (plane ?plane)
       (airport ?loc2) (not (?loc1 = ?loc2))
 (add (at ?plane ?loc1))
 (del (at ?plane ?loc2)))
```

Drive a truck between two locations in the same city.

Logistics World

```
(def… (load ?package ?vehicle ?loc)
  (pre (at ?package ?loc) (at ?vehicle ?loc)
       (package ?package))
  (add (in ?package ?vehicle))
  (del (at ?package ?loc)))

(def… (unload ?package ?vehicle ?loc)
  (pre (in ?package ?vehicle) (at ?vehicle ?loc)
       (package ?package))
  (add (at ?package ?loc))
  (del (in ?package ?vehicle)))

(def… (drive ?truck ?loc1 ?loc2 ?city)
  (pre (at ?truck ?loc) (in-city ?loc1 ?city)
       (in-city ?loc2 ?city) (truck ?truck)
       (not (?loc1 = ?loc2))
  (add (at ?truck ?loc1))
  (del (at ?truck ?loc2)))
```

```
(def… (fly ?plane ?loc1 ?loc2)
 (pre (at ?plane ?loc1) (plane ?plane)
      (airport ?loc2) (not (?loc1 = ?loc2))
 (add (at ?plane ?loc1))
 (del (at ?plane ?loc1)))
```

Fly an airplane to a new airport.

Sub-Spaces

- Sub-Spaces of the full state space can be defined by various abstractions. E.g., we can ignore certain objects, replace a collection of objects by a single “generic” object, remove certain predicates, etc.

Sub-Spaces

- E.g., in the logistic world if we consider generic package `A`, a generic plane `P`, a generic truck `Ti` for each city, all of the locations, and the set of predicates `at(A,?l)` and `in(A,?v)` we will find that there are only five types of non-cyclic paths in the resulting subspace between states where `at(A,L1)` holds and states where `at(A,L4)` holds.
- Call this the Package Sub-Space.

Sub-Spaces

1. Move by truck
   at(A,L1)→in(A,T1)→at(A,L4)
2. Move by plane
   at(A,L1)→in(A,T1)→at(A,L4)
3. Move by truck then by plane
   at(A,L1)→in(A,T1)→at(A,L2)
   →in(A,P)→at(A,L4)
4. Move by plane then by truck
   at(A,L1)→in(A,P)→at(A,L2)
   →in(A,T2)→at(A,L4)
5. Move by truck then plane then truck
   at(A,L1)→in(A,T1)→at(A,L2)
   →in(A,P)→at(A,L3)→in(A,T3)
   →at(A,L4)

Sub-Space interactions

- Sub-spaces can be independent or they can interact.
  - A sub-space state can enable transitions other sub-spaces spaces
    - `at(Truck,L1)` enables `at(A,L1)→in(A,Truck)`
  - Some sub-space transitions are independent of other sub-space states
    - `at(A,L1)→in(A,Truck)` is independent of `at(B,L1)`
  - Some transitions must synchronize with other sub-space transitions
    - `(not on(A,B))→on(A,B)` forces `clear(B)→(not clear(B))`

Sub-Space cycles

- Some sub-space cycles are necessary as they enable transitions in other sub-spaces
  - holding(?block)→handempty →holding(?block)
- But others are not,
  - In the logistics world a package never interacts with another package.
  - So it is never necessary to have a cycle in the logistics package sub-space.

Domain Specific Search Control

- Identify and block useless sub-space cycles in the forward search space.
- Two ways to do it
  - Examine the history of full states to ensure that the sub-space states are not cyclic.
  - Force the relevant sub-spaces to make non-decreasing progress towards their sub-space goal state.
- In [Bacchus & Kabanza 1995,2000] the latter approach is used.
- LTL (linear temporal logic) is used to specify paths in the state space that have non-decreasing progress in various sub-spaces.

LTL

- A logic whose formulas are true or false of sequences of states.
  - i.e., every formula specifies a property of a sequence of states.
- `Next(f)`: `f` must be true in the next state
- `Always(f)`: `f` must be true in all future states
- `Until(f1,f2)`: `f1` must be true in all future states until `f2` is true.
- Want to specify search control for a problem domain, so use first-order LTL.
- When solving any particular planning problem, the formulas become propostional.

Forcing Monotonic Progress

- Logistics world want to force non-decreasing progress in the package subspace.
- For each package, block the transition from `at`$\rightarrow$`in` and from `in`$\rightarrow$`at` unless we need to load or unload into that type of vehicle at that particular type of location to make progress towards the goal location of the package.
- E.g., only need to load a package into an airplane if the package's goal location is in another city. Only need to unload from a truck if the truck is at an airport (and the package must be moved to another city) or if the truck is at the package's final destination.

Example LTL Formula

```
(always
  (forall (?pkg ?vehicle) (in ?pkg ?vehicle)
    (exists (?loc) (at ?vehicle ?loc)
      (implies
        (or
          (and
            (truck ?vehicle)
            (not (need-to-unload-from-truck ?pkg ?loc)))
          (and
            (airplane ?vehicle)
            (not (need-to-unload-from-plane ?pkg ?loc))))
        (next (in ?pkg ?vehicle)))))))))
```

Example LTL Formula

```
(always
  (forall (?pkg ?vehicle) (in ?pkg ?vehicle)
    (exists (?loc) (at ?vehicle ?loc)
      (implies
        (or
          (and
            (truck ?vehicle)
            (not (need-to-unload-from-truck ?pkg ?loc)))
          (and
            (airplane ?vehicle)
            (not (need-to-unload-from-plane ?pkg ?loc))))
        (next (in ?pkg ?vehicle)))))))
```

Bounded Quantification

TLplan Formula

```
(always
  (forall (?pkg ?vehicle) (in ?pkg ?vehicle)
    (exists (?loc) (at ?vehicle ?loc)
      (implies
        (or
          (and
            (truck ?vehicle)
            (not (need-to-unload-from-truck ?pkg ?loc)))
          (and
            (airplane ?vehicle)
            (not (need-to-unload-from-plane ?pkg ?loc))))
        (next (in ?pkg ?vehicle)))))))
```

Abbreviations for various conditions on the current state and on the goal location of the package.

TLplan Formula

```
(always
  (forall (?pkg ?vehicle) (in ?pkg ?vehicle)
    (exists (?loc) (at ?vehicle ?loc)
      (implies
        (or
          (and
            (truck ?vehicle)
            (not (need-to-unload-from-truck ?pkg ?loc)))
          (and
            (airplane ?vehicle)
            (not (need-to-unload-from-plane ?pkg ?loc))))
        (next (in ?pkg ?vehicle)))))))
```

Temporal formula of the form

```
(always (implies F1 (next F2)))
```

Model checking the LTL formula

- As forward chaining explores action sequences (state sequences) we incrementally check the formula against the generated sequence.
- If the sequence falsifies the formula it is pruned: the sequence contains a sub-state cycle.

Model checking the LTL formula

- This can be accomplished in various ways
  - Every LTL modality can be rewritten into a current component and a next state component
    - `(always f1) == (and f1 (next (always f1)))`
  - The formula is progressed through the current state, by computing the next state component and checking the current component.
  - The process seems to be very similar to on the fly automata construction
  - One key difference is that we do not convert the first-order LTL formula into a propositional formula immediately, it would be too large.
  - Rather we propositionalize on the fly.

The reduced search space

- This yields a reduced search space that can then be explored with various kinds of search strategies.
- In planning benchmark domains it is easy to achieve very significant reductions to the search space, as it is easy to identify useless sub-space cycles by inspection or by examining the paths explored by the planner.
- These reductions are so significant that it can often be the case that simple depth-first search will find a good plan in the reduced space without backtracking.

The reduced search space

- E.g., TALPlanner [Kvarnstrom & Doherty] was able to solve 500 block problems in 1.5 sec. in the AIPS2000 competition (2000 step plan).
- The fastest planner not taking advantage of domain specific information took 900 sec. to solve 50 block problems.
- Furthermore, the plans generated by TALPlanner using its search control are provably not more than 2 times longer than the optimal.
- Optimal blocks world planning is NP-complete.

Speedups: precondition control

- On some problems, e.g., large logistics world problems, the branching factor can be 1000 or 2000, and the plan 300 steps long.
- When we have temporal control formulas of the form
  `(always (implies F1 (next F2)))`
  With `F1` and `F2` being atemporal, it is possible to regress `F2` through each operator to determine what conditions need to hold prior to the operator in order that `F2` will hold after.
- Then one can conjoin a new precondition to each operator `o`
  `(implies F1 (regress[F2,o]))`
- Then one can remove the control formula, it has been folded into the successor relation.

Speedups: materialized views

- States are represented in relational form.
- To compute the set of successor states one needs to find all the actions that are applicable to the state. This is the set of all instantiations satisfying the operator preconditions in the state.
- However actions change only a few relations, so in the successor state the same set of applicable actions will be almost the same.
- Instead of recomputing the set of applicable actions, one can update a relation containing all the satisfying instantiations of the operator preconditions.
- This is materialized view from databases.

Speedups Effects

Blocks	Standard Blocks World	Optimized Blocks World
300	108 sec	0.08 sec
350	192 sec	0.14 sec
400	318 sec	0.19 sec
450		0.12 sec
500		0.14 sec

Speedup Effects

- We can also do breadth first search in the reduced space
- Can find optimal solutions to blocks world problems using breadth first search (NP-compete problem), solving 41 block problems in 13sec on average finding plans that average 142 steps.

Conclusions

- AI planning uses
  - Relational representations.
  - Tackles problems that are very large, but with a lot of structure.
- Constraints can be used.
- Relaxed reachability can provide useful heuristic guidance.
- Many planning domains support strong control knowledge.