Linköping University

Automated Planning

Domain-Configurable Planning, Domain-Configurable Heuristics, Planning with Control Formulas

Jonas Kvarnström
Automated Planning Group
Department of Computer and Information Science
Linköping University

Assumptions

- Recall the fundamental assumption that we **only** specify
  - Structure: Objects and state variables
  - Initial state and goal
  - **Physical** preconditions and **physical** effects of actions

We only specify what *can* be done

The *planner* must decide what *should* be done

But even the most sophisticated heuristics and domain analysis methods lack our intuitions and background knowledge…

Domain-Configurable Planners

Let's see how we can make **a planner** take advantage of what **we** know!

- Planners taking advantage of additional knowledge can be called:
  - **Knowledge-rich**
  - **Domain-configurable**
  - (Sometimes **incorrectly** called "domain-dependent")

Comparisons (1)

More effort

Higher performance

Domain-specific

Must write an entire planner
Can specialize the planner for very high performance

Domain-configurable

High-level (but sometimes complex) domain definition
Can provide more information for high performance

"Domain-independent"

Provide minimal information about actions
Less efficient

Comparisons (2)

More coverage ↑

Domain-configurable

Easier to improve expressivity and efficiency
➔ Often practically useful for a larger set of domains!

"Domain-independent"

Should be useful for a wide range of domains

Domain-specific

Only works in a single domain

Domain-Configurable Heuristics

Introduction

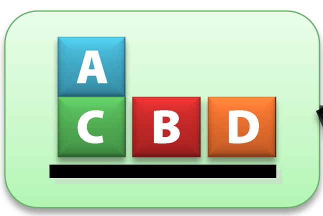
How can **a planner** take advantage of what **we** know?

- First, what we're already used to… **Heuristics**!
  - Given the current state,
    how much will it cost to reach the goal?

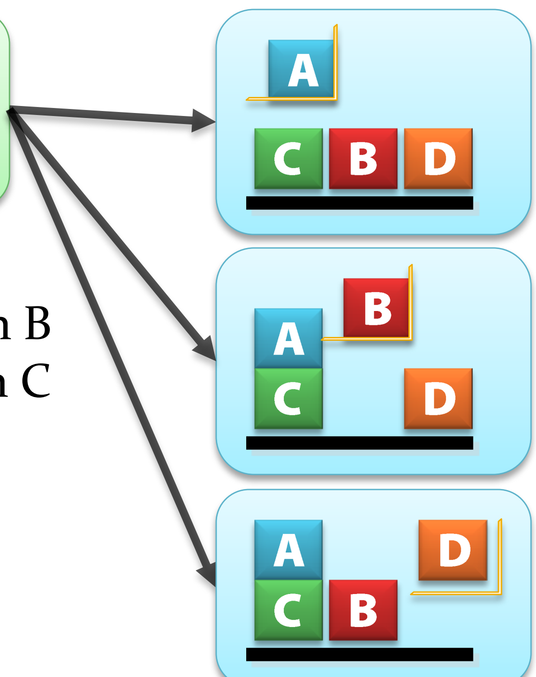
Blocks World Heuristics (1)

- Blocks World, step 1a:

We are **<u>not holding</u>** block A, and it is **<u>misplaced</u>**
➔ we will need one pickup or unstack,
then one putdown or stack



A should be on B
B should be on C
➔ 4 [h*=6]



B should be on C
➔ 2 [h*=5]

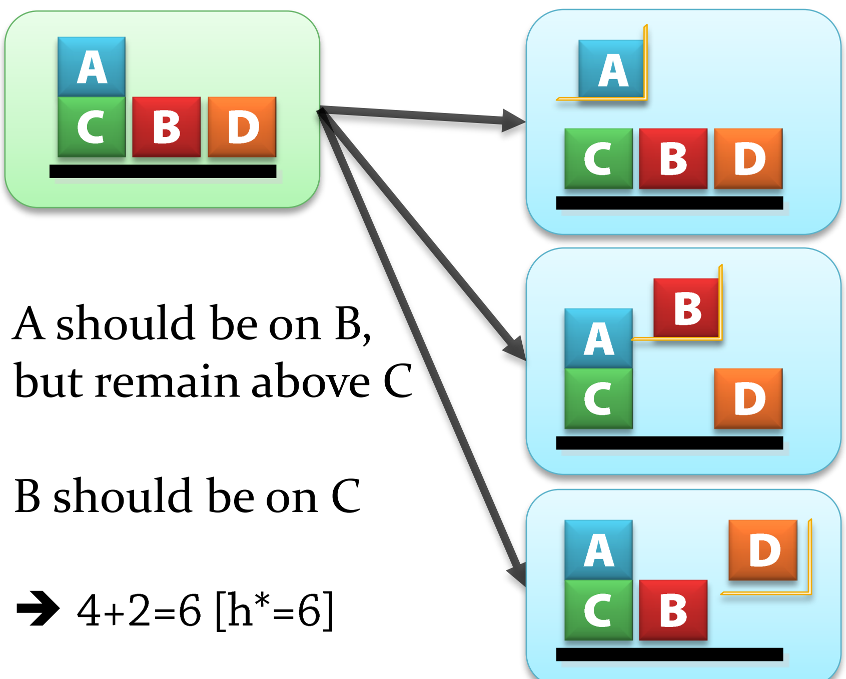
A should be on B
➔ 2 [h*=7]

A should be on B,
B should be on C
➔ 2+2=4 [h*=7]

Blocks World Heuristics (2)

- Blocks World, step 1b:

In addition to the previous condition,
block A **<u>is above</u>** block C, which it should **<u>remain above</u>**
➔ we need to place it somewhere *temporarily*, then *restore* it
(unstack(A), putdown(A), …, pickup(A), stack(B,C))
➔ two more actions



B should be on C
➔ 2 [h*=5]

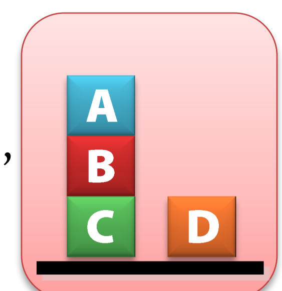
A should be on B / remain above C
➔ 4 [h*=7]

A should be on B / remain above C,
B should be on C
➔ 4+2=6 [h*=7]

A should be on B,
but remain above C

B should be on C

➔ 4+2=6 [h*=6]

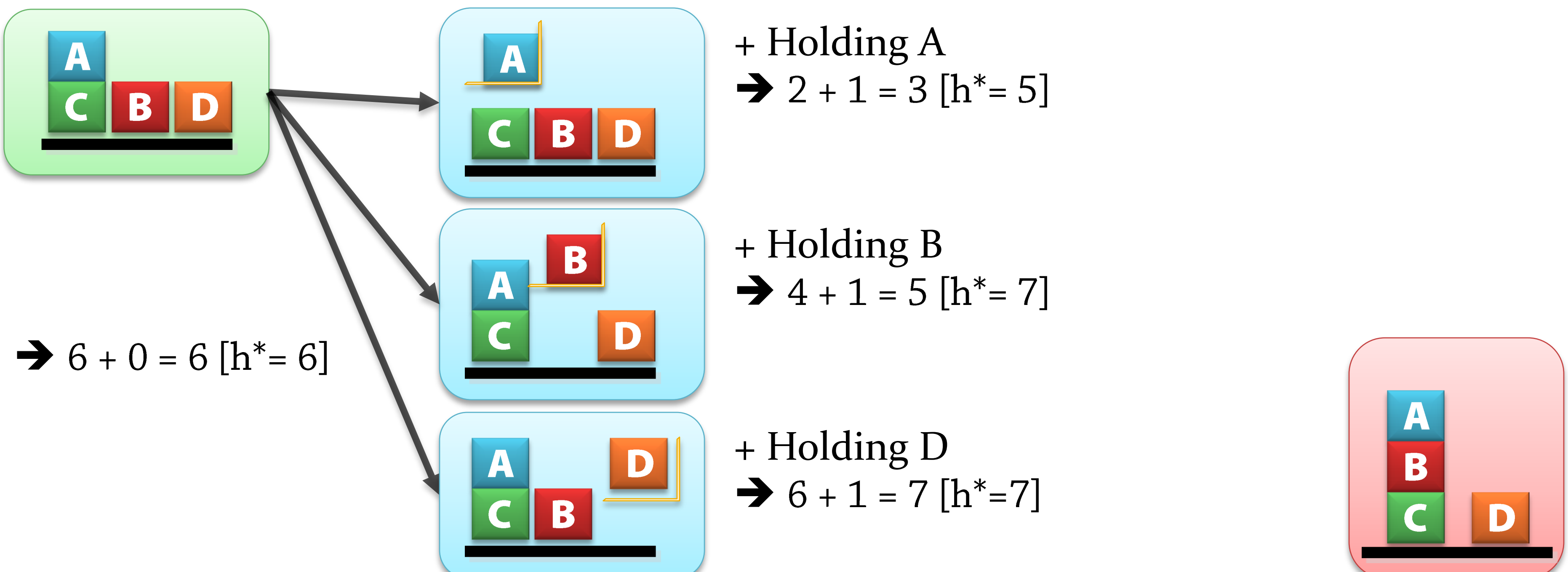


Blocks World Heuristics (3)

- Blocks World, step 2a:

If we are holding a block,
we will need at least **one** *putdown* or **one** *stack* for that block

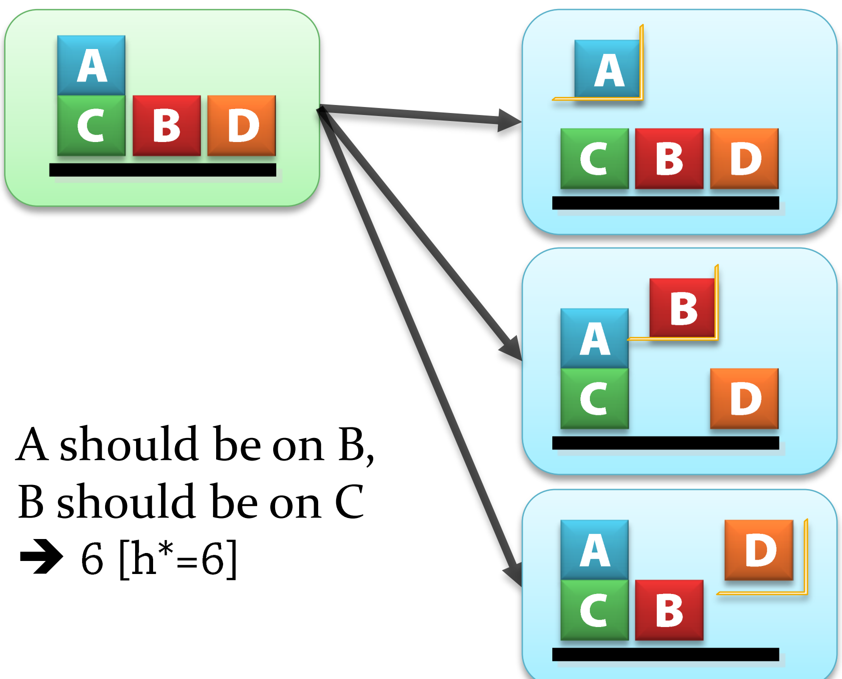
Steps 1/2 never apply for the same block
➔ independent
➔ addition yields admissible heuristic!



Blocks World Heuristics (4)

- Blocks World, step 2b:

We are holding A,
but its destination is B, which **<u>is not ready</u>**
➔ We also need to put it down now, pick it up later
(two more actions)

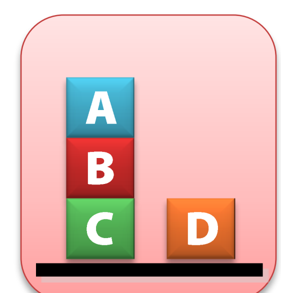


+ Holding A, its destination B is not ready
➔ 2 + 1 + 2 = 5 [h*= 5]

A should be on B,
holding B, its destination D is not ready
➔ 4 + 1 + 2 = 7 [h*= 7]

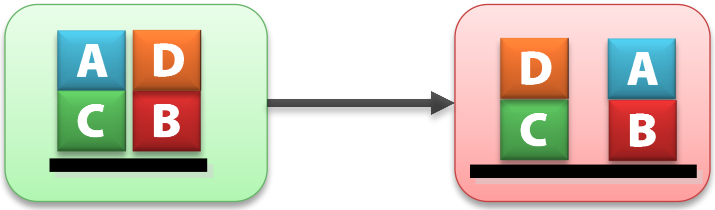
➔ 6 + 1 + 0 = 7 [h*= 7]

A should be on B,
B should be on C
➔ 6 [h*=6]



Blocks World Heuristics (5)

- Does this calculate true costs, h*(s)?
  - No!



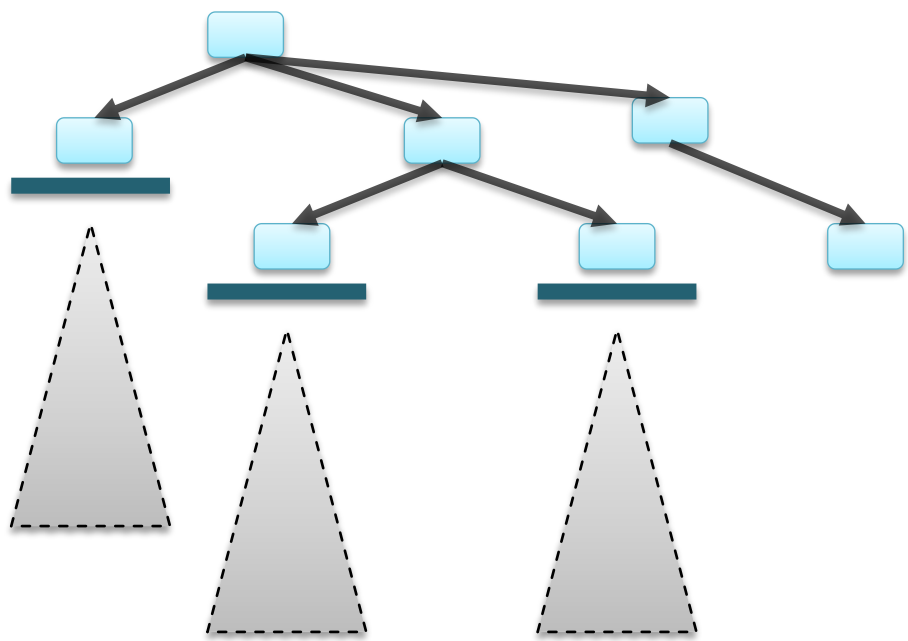
  - A should not be on C, not remain above C ➔ 2
  - D should not be on B, not remain above B ➔ 2
  - **Total estimated cost**: 4
  - **Shortest plan**:
    unstack(A,C); putdown(A);
    unstack(D,B); stack(D,C);
    pickup(A); stack(A,B)

Domain-configurable heuristics:
Feasible, but not so commonly used!

Planning with Control Formulas

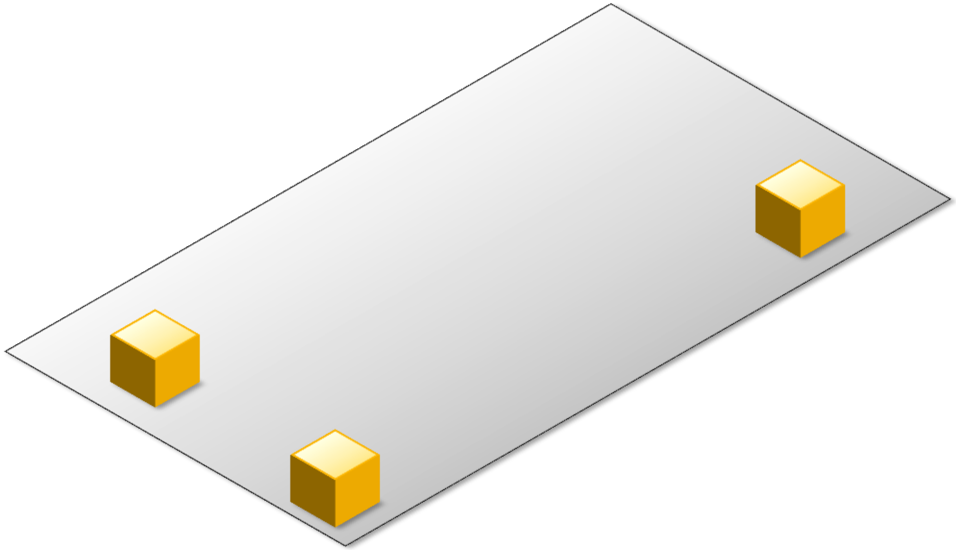
Control Rules

- Heuristics only **prioritize**
  - Good when you are **uncertain** – keep nodes in case they are needed later
- **We** can often find cases where we can **prune** the search tree
  - Prune = beskära = cutting off branches
  - If we “don’t approve” of a search node,
    backtrack and **never consider the node or its descendants again!**



Example: Emergency Services Logistics

- Emergency Services Logistics
  - Goal: at(crate1, loc1), at(crate2, loc2), at(crate3, loc3)
  - Now: at(crate1, loc1), at(crate2, loc2)



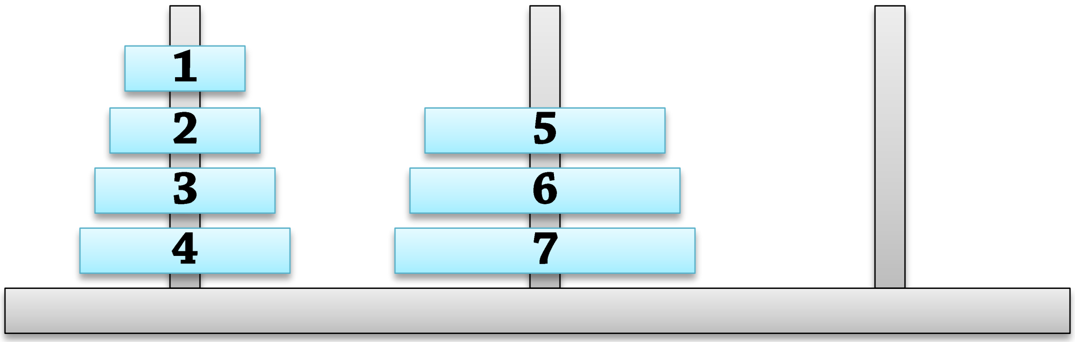
  - Picking up crate1 again is **physically possible**
  - It "destroys" at(crate1, loc1), which is a goal – **obviously stupid!**

The **branch** beginning with pickup(crate1) can be **pruned** from the tree!

Example: Towers of Hanoi

Should we **always** prevent the destruction of achieved goals?

- Goal: on(1,2), on(2,3), on(3,4), on(4,5), on(5,6), on(6,7)
- Now: on(1,2), on(2,3), on(3,4), on(5,6), on(6,7)



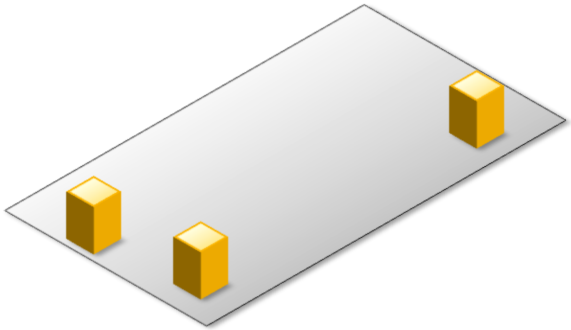
- Moving disk 1 to the third peg is **possible** but "destroys" a goal: on(1,2)
  - Is this also **obviously stupid**?
  - No, it is **necessary**! Disk 1 is blocking us from moving disk 4…

Deciding **which** goals the planner may "destroy"
is one of **many** non-trivial tasks for a planner!

➔ It should benefit from more **control information** from the user!

Precondition Control

Simplest control information: <u>Precondition Control</u>



- **operator** pickup(*robot, crate, location*)
  - **precond**:
    - at(*robot*, *location*), at(*crate*, *location*)
    - handempty(*robot*)
    - ...and **the goal doesn't state that *crate* should end up at *location***!

How to express this???

- Alternative 1: **New predicate** "destination(*crate*, *loc*)"
  - *Duplicates* the information already specified in the goal
  - **precond**: ¬destination(*crate*, *location*)

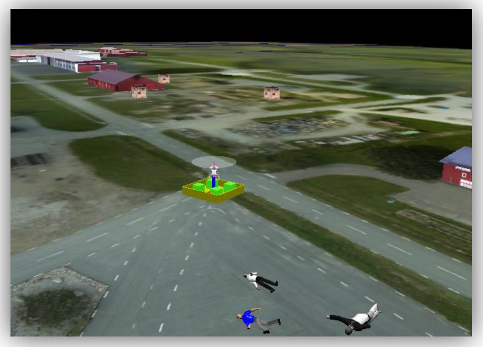
Supported by any planner

- Alternative 2: **New language extension** "goal(ϕ)"
  - Evaluated in the set of goal states, not in the current state
  - **precond**: ¬**goal**(at(*crate*, *location*))

Requires extensions, but more convenient

State Constraints

- A UAV should never be where it **<u>can't reach</u>** a refueling point
  - If this happens in a plan, we can't possibly extend it into a solution satisfying the goal

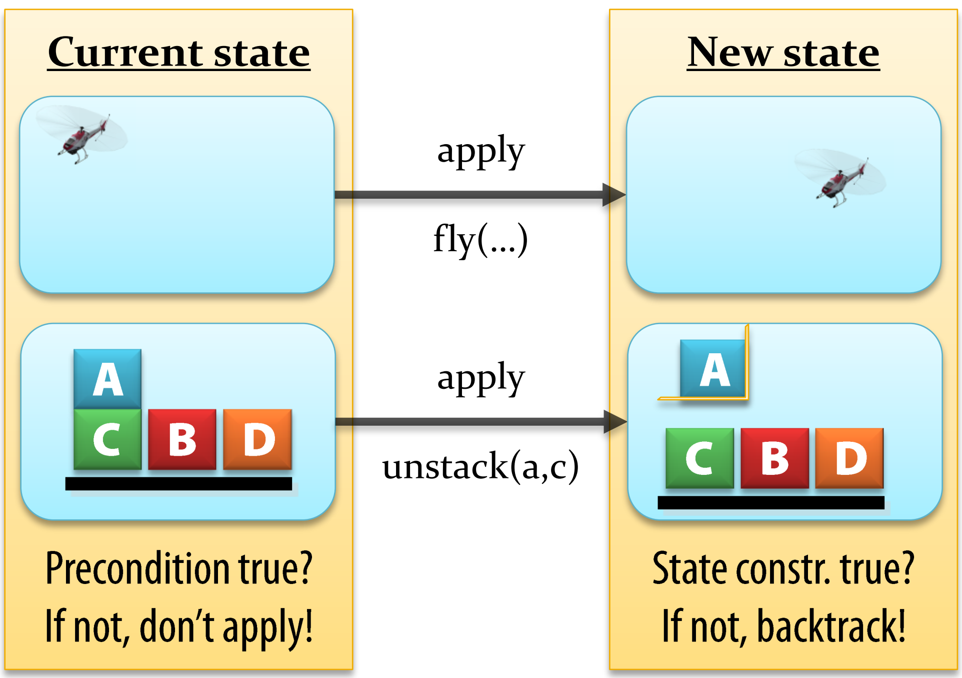


- How to express this?

Using **<u>preconditions</u>** again?	Using **<u>state constraints</u>**?
Must be verified for **<u>every</u>** action: fly, scan-area, take-off, … Must be checked even when the UAV is idle, hovering Inconvenient!	Defined **<u>once</u>**, applied to **<u>every generated state</u>** $\forall u$ (uav(u) $\Rightarrow$ $\exists$rp (refueling-point(rp) $\wedge$ dist(u,rp) * fuel-usage(u) < fuel-avail(u)) Comparatively simple extension!

Testing State Constraints

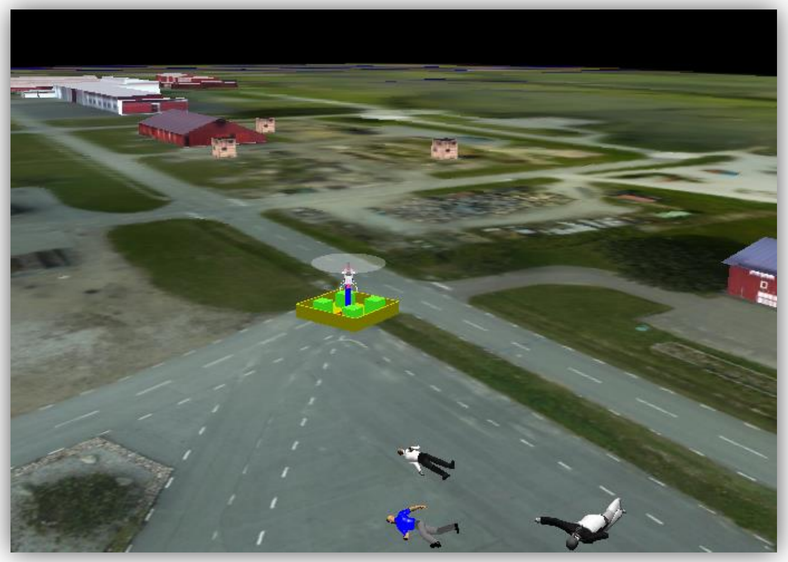
- Testing such **state constraints** is simple
  - When we apply an action, a new state is generated
    - If the formula is not true in that state: Prune!
  - Similar to preconditions
    - But tested in the state **after** an action is applied, not **before**!



Temporal Conditions (1)

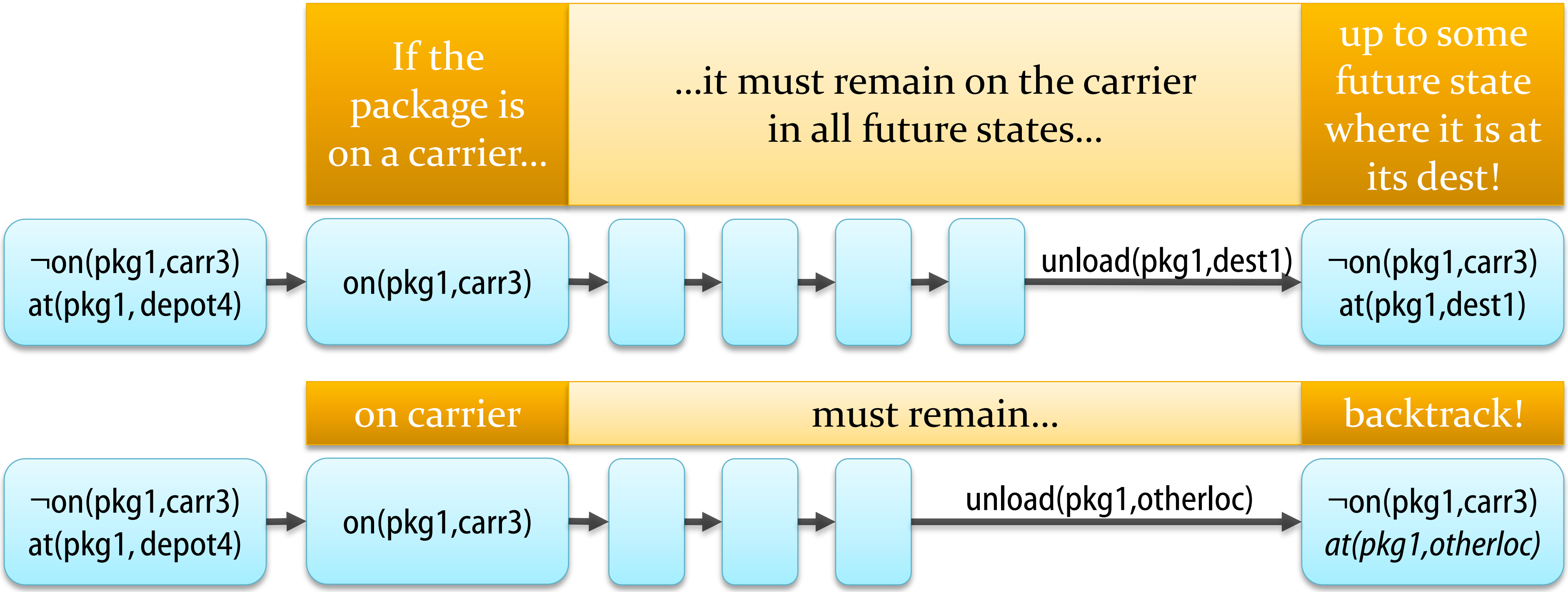
- A package on a carrier should **remain there** until it reaches its destination
  - For any plan where we move it,
    there is another (shorter, more efficient) plan where we don't

How to express this as a single formula?



Temporal Conditions (2)

- "A package on a carrier should **remain** there **until** it reaches its destination"



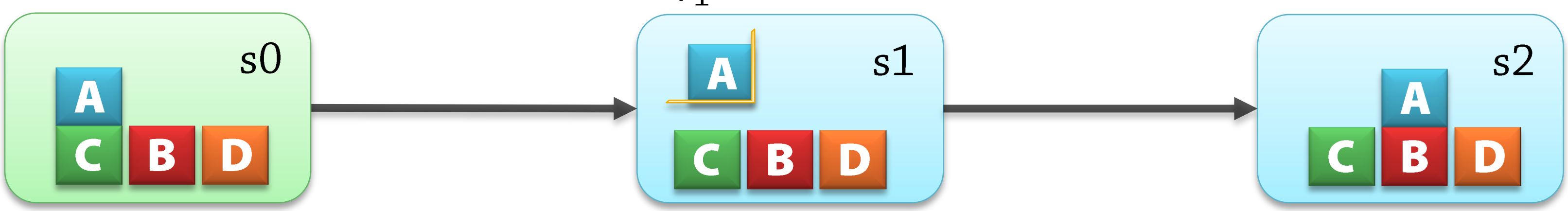
We need a formula constraining an entire **state sequence**, not a single state!

In planning, this is called a **control formula** or **control rule**

Linear Temporal Logic

We need to extend the logical language!

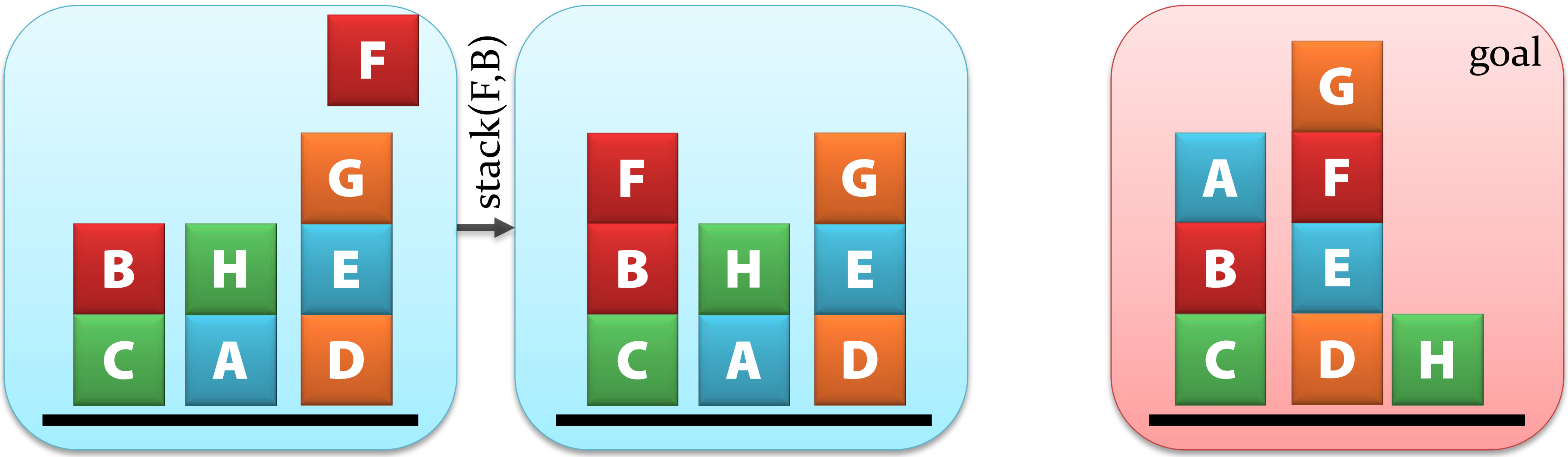
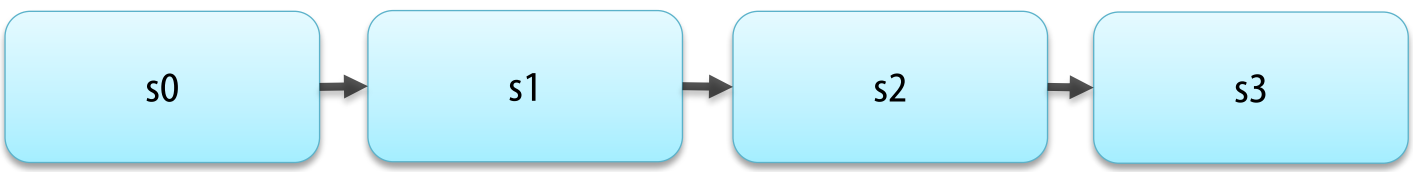
- One possibility: Use **Linear Temporal Logic** (as in TLplan)
  - All formulas evaluated relative to a *state sequence* and a *current state*
  - Assuming that *f* is a formula:
    - $\bigcirc f$ – (**next** *f*) — *f* is true in the next state, e.g.,
    - $\Diamond f$ – (**eventually** *f*) — *f* is true either now **or** in some future state
    - $\Box f$ – (**always** *f*) — *f* is true now **and** in all future states
    - $f_1 \cup f_2$ – (**until** f_1 f_2) — f_2 is true either now or in some future state, and f_1 is true until then



Formulas true in <**s0**,s1,s2>:
(on A C)
(**next** (holding A))
(**next** (**next** (on A B))
(**until** (clear B) (on A B))

Formulas true in <**s1**,s2>:
(ontable B)
(holding A)
(**next** (on A B)
(**until** (clear B) (on A B))

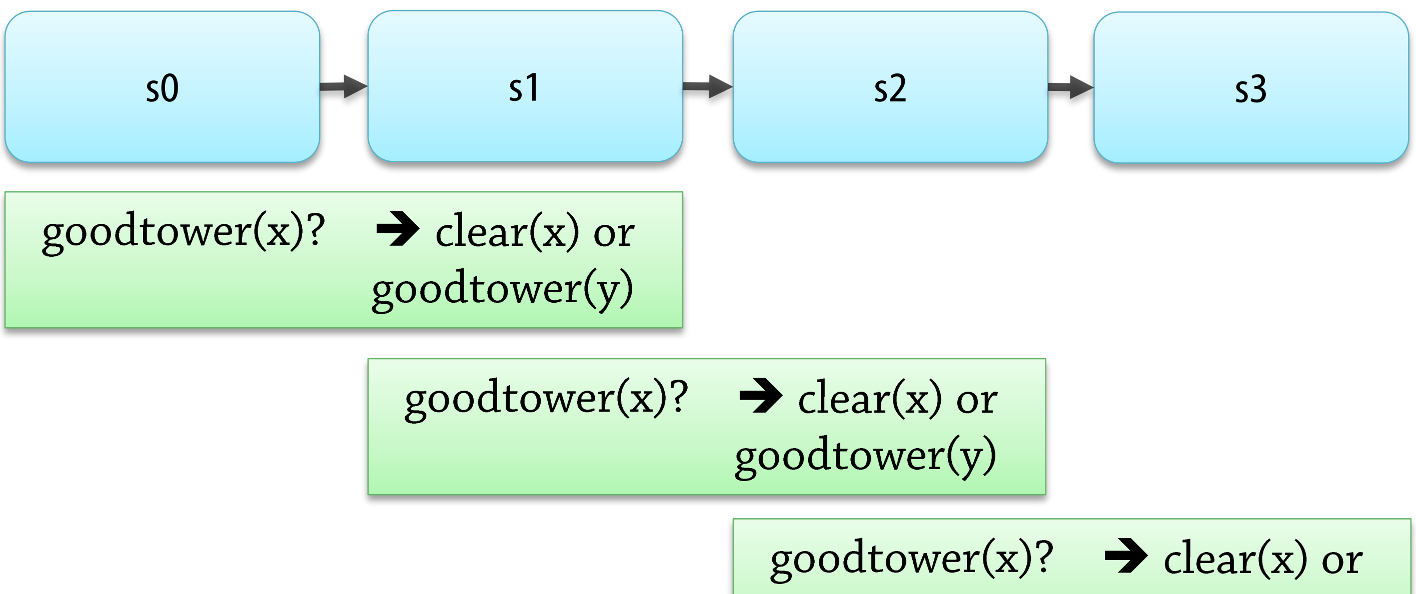
Control Formula

- "A package on a carrier should **remain** there **until** it reaches its destination"
  - ```
    (always
          (forall (?c) (carrier ?c)            ;; For all carriers
              (forall (?p) (package ?p)        ;; For all packages
                  (implies
                      (on-carrier ?p ?c)       ;; If the package is on the carrier
                      (until  (on-carrier ?p ?c)        ;; ...then it remains on the carrier
                              (exists (?loc)            ;; until there exists a location
                                  (at ?p ?loc)          ;; where it is, and
                                  (goal (at ?p ?loc)))) ;; where the goal says it should be
    ))))
    ```
    ```

Finding Control Formulas

Blocks World

- How do we come up with **good control rules**?
  - Good starting point: "**Don't be stupid!**"
  - **Trace** the search process – suppose the planner tries this:



  - Placing **F on top of B** is stupid, because we'll have to remove it later
    - Would have been better to put F on the table!
  - Conclusion: Should not **extend** a **good tower** the wrong way
    - *Good tower*: a tower of blocks that will never need to be moved

Blocks World Example (continued)

- Rule 1: Every goodtower must always **remain a goodtower**
  - ```
    (forall (?x) (clear ?x)   ;; For all blocks that are clear (at the top of a tower)
       (implies
          (goodtower ?x)      ;; If the tower is good (no need to move any blocks)
          (next (or           ;; ...then in the next state, either:
             (clear ?x)                          ;; ?x remains clear (didn't extend the tower)
             (exists (?y) (on ?y ?x)             ;; or there is a block ?y which is on ?x
                         (goodtower ?y))         ;; which is a goodtower
    )))
    ```



goodtower(x)? → clear(x) or goodtower(y)

What about the rest?
    ```

Blocks World Example (continued)

- Rule 1, second attempt:
  - (**always**

```
        (forall (?x) (clear ?x)  ;; For all blocks that are clear (at the top of a tower)
        (implies
            (goodtower ?x)       ;; If the tower is good (no need to move any blocks)
            (next (or            ;; ...then in the next state, either:
                (clear ?x)                        ;; ?x remains clear (didn't extend the tower)
                (exists (?y) (on ?y ?x)           ;; or there is a block ?y which is on ?x
                             (goodtower ?y))      ;; which is a goodtower
))))
```



Supporting Predicates

- Some planners allow us to **<u>define</u>** a predicate recursively
  - **<u>goodtowerbelow</u>**(x) means we **<u>will not have to move</u>** x
    - $goodtowerbelow(x) \Leftrightarrow$

$$[ontable(x) \wedge \neg\exists[y{:}\mathbf{GOAL}(on(x,y))]]$$

$$\vee$$

$$\exists[y{:}on(x,y)]\ \{$$

$$\neg\mathbf{GOAL}(ontable(x)) \wedge$$

$$\neg\mathbf{GOAL}(holding(y)) \wedge$$

$$\neg\mathbf{GOAL}(clear(y)) \wedge$$

$$\forall[z{:}\mathbf{GOAL}(on(x,z))]\ (z = y) \wedge$$

$$\forall[z{:}\mathbf{GOAL}(on(z,y))]\ (z = x) \wedge$$

$$goodtowerbelow(y)$$

$$\}$$

X is on the table, and shouldn't be on anything else

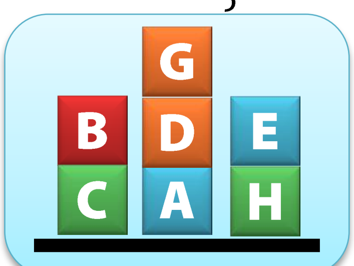
X is on something else

Shouldn't be on the table, shouldn't be holding it, shouldn't be clear

If x should be on z, then it *is* (z is y)

If z should be on y, then it *is* (z is x)

The remainder of the tower is also good



goodtowerbelow: B, C, H

Supporting Predicates

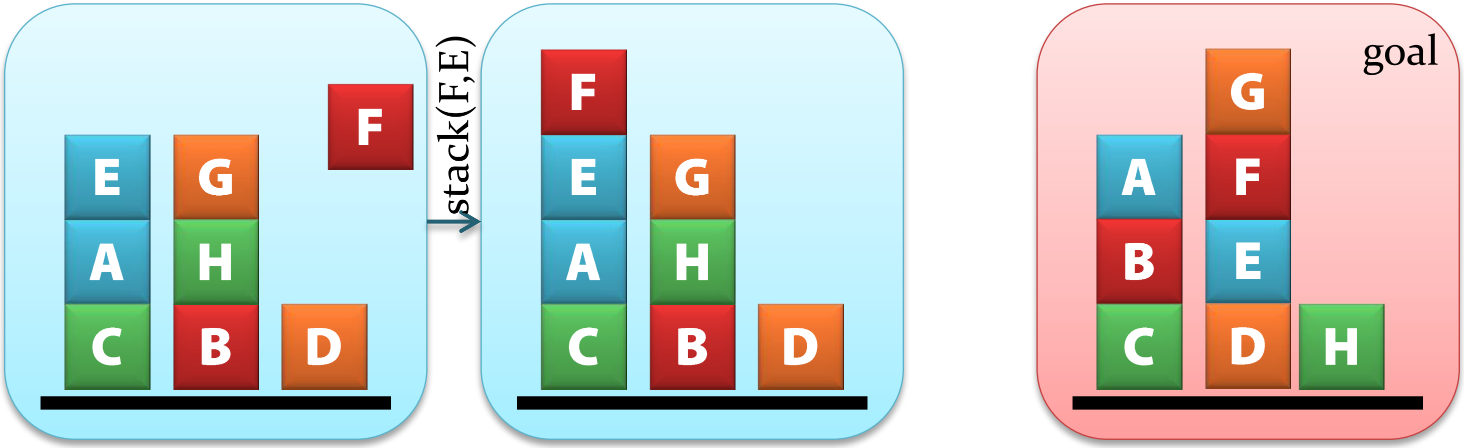
- **goodtower**(x) means x is the block at the top of a good tower
  - *goodtower*(x) $\Leftrightarrow$ *clear*(x) $\wedge \neg$ GOAL(*holding*(x)) $\wedge$ *goodtowerbelow*(x)

- **badtower**(x) means x is the top of a tower that isn't good
  - *badtower*(x) $\Leftrightarrow$ *clear*(x) $\wedge \neg$*goodtower*(x)

goodtower: B
goodtowerbelow: B, C, H
badtower: G, E
(neither: D, A)

Blocks World

- Step 2: Is this stupid?

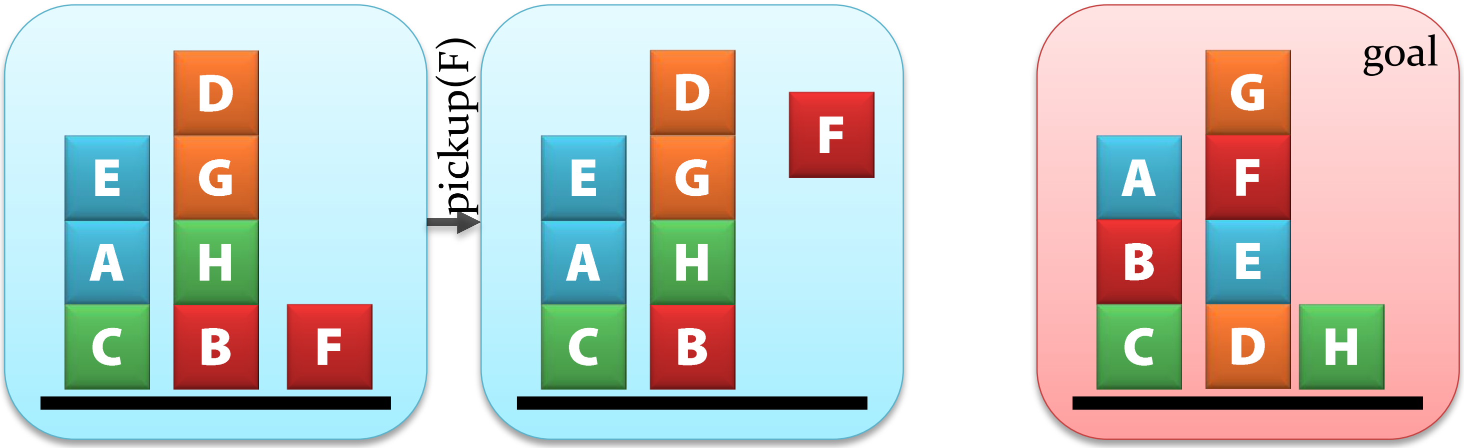


- Placing **<u>F on top of E</u>** is stupid, because we have to move E later…
  - Would have been better to put F on the table!
  - But E was not a goodtower, so the previous rule didn't detect the problem
- Never put anything on a badtower!
  - (**always**

```
(forall (?x) (clear ?x)      ;; For all blocks at the top of a tower
    (implies
        (badtower ?x)        ;; If the tower is bad (must be dismantled)
        (next (not (exists (?y) (on ?y ?x)))))))  ;; Don't extend it!
```

Blocks World

- Step 3: Is this stupid?



  - <u>Picking up F</u> is stupid!
    - It is on the table, so we can wait until its destination is ready:

    -
```
(always
      (forall (?x) (clear ?x)      ;; For all blocks at the top of a tower
          (implies
              (and    (ontable ?x)
                      (exists (?y) (goal (on ?x ?y)) (not (goodtower ?y)))
              (next (not (holding ?x)))))))
```

Pruning using Control Formulas

Pruning using Control Formulas

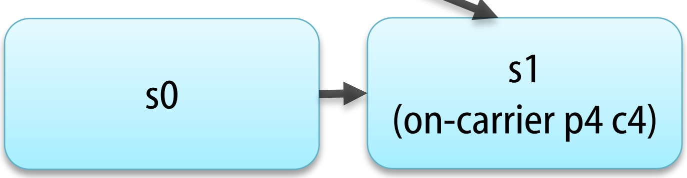
- How do we decide **when to prune** the search tree?
  - Obvious idea:
    - Take the **state sequence** corresponding to the **current action sequence**
    - **Evaluate** the formula over that sequence
    - If it is false: Prune / backtrack!

Evaluation 1

- Problem:
  - ```
    (always
        (forall (?c) (carrier ?c)              ;; For all carriers
            (forall (?p) (package ?c)          ;; For all packages
                (implies
                    (on-carrier ?p ?c)         ;; If the package is on the carrier
                    (until (on-carrier ?p ?c)  ;; ...then it remains on the carrier
                        (exists (?loc)         ;; until there exists a location
                            (at ?p ?loc)       ;; where it is, and
                            (goal (at ?p ?loc)))) ;; where the goal says it should be
    ))))
    ```

No package on a carrier in the initial state: Everything is OK

"Every boat I own is a billion-dollar yacht (because I own no boats)"

s0
    ```

Evaluation 2

- Problem:
  - ```
    (always
        (forall (?c) (carrier ?c)                  ;; For all carriers
            (forall (?p) (package ?c)              ;; For all packages
                (implies
                    (on-carrier ?p ?c)             ;; If the package is on the carrier
                    (until  (on-carrier ?p ?c)           ;; ...then it remains on the carrier
                            (exists (?loc)               ;; until there exists a location
                                (at ?p ?loc)             ;; where it is, and
                                (goal (at ?p ?loc))))    ;; where the goal says it should be
    ))))
    ```

When we add an action placing a package on a carrier...

...there is no future state where the package is at its destination!



The formula is violated,
but only because the solution is not *complete* yet!
We must be allowed to continue,
generating new states...
    ```

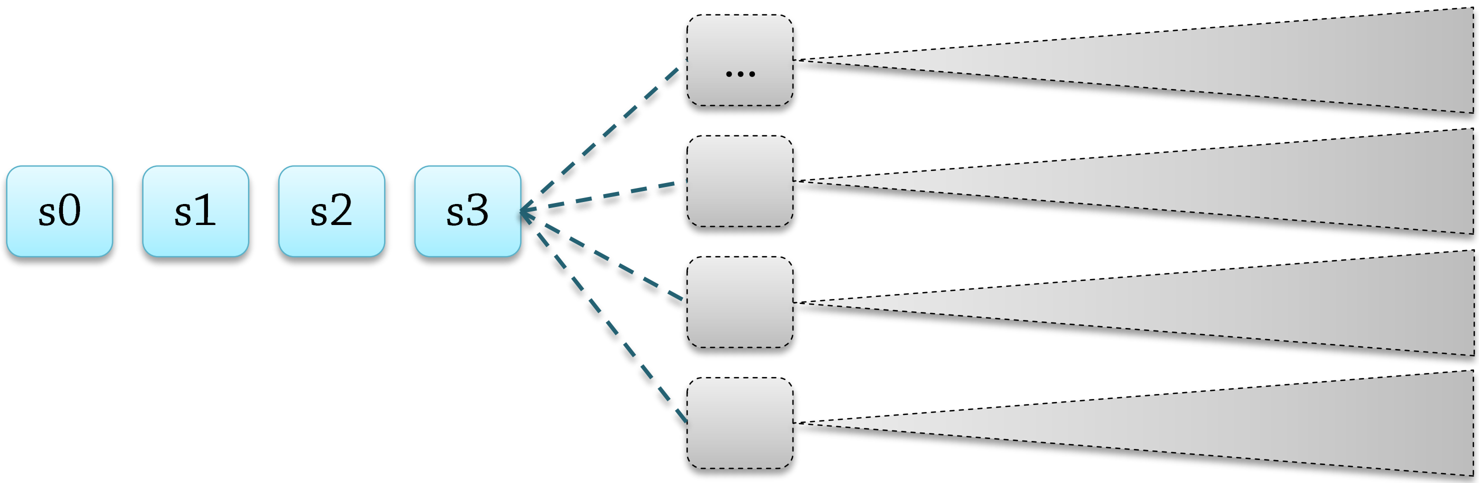
Evaluation 3: What's Wrong?

- We had an **obvious** idea:
  - Take the state sequence corresponding to the current plan
  - Evaluate the formula over that sequence
  - If it is false: Prune / backtrack!

- This is actually **wrong**!
  - Formulas should hold in the state sequence of the **solution**
  - But they don't have to hold in every **intermediate** action sequence…

Analysis

- Analysis:



We have applied some actions, yielding a sequence of states

We intend to generate additional actions and states, but right now we don't know which ones

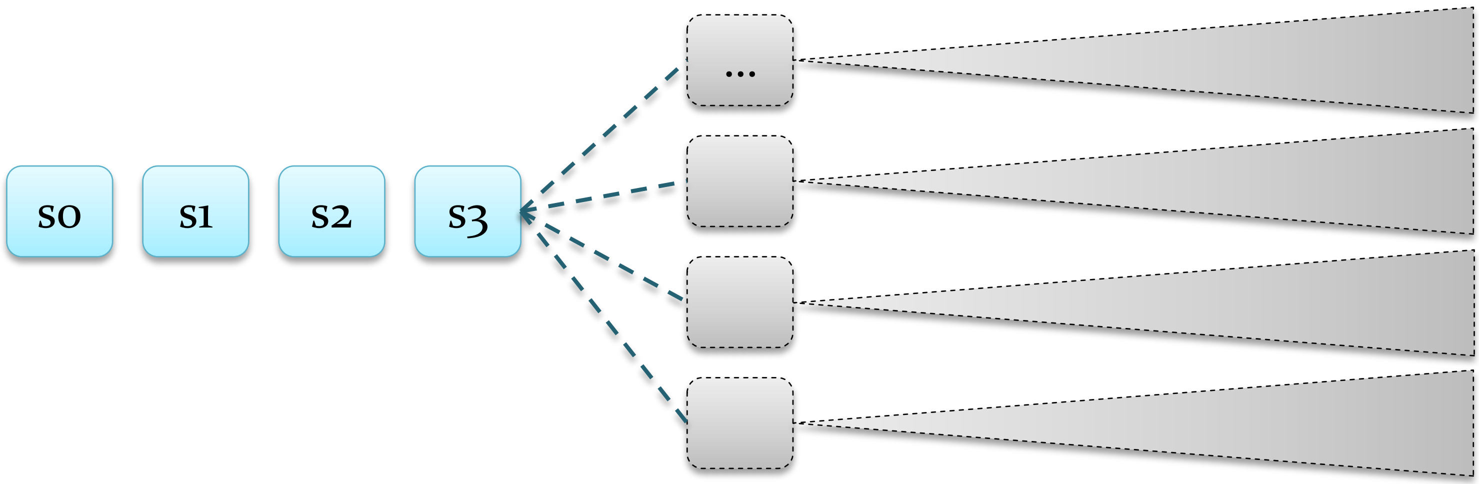
The control formula should be satisfied by the **entire** state sequence corresponding to a solution

We only know **some** of those states

Should only backtrack if we can prove that you **can't** find **additional** states so that the control formula becomes true

Analysis 2

- Analysis 2:



The control formula should be satisfied
by the **entire** state sequence corresponding to a solution

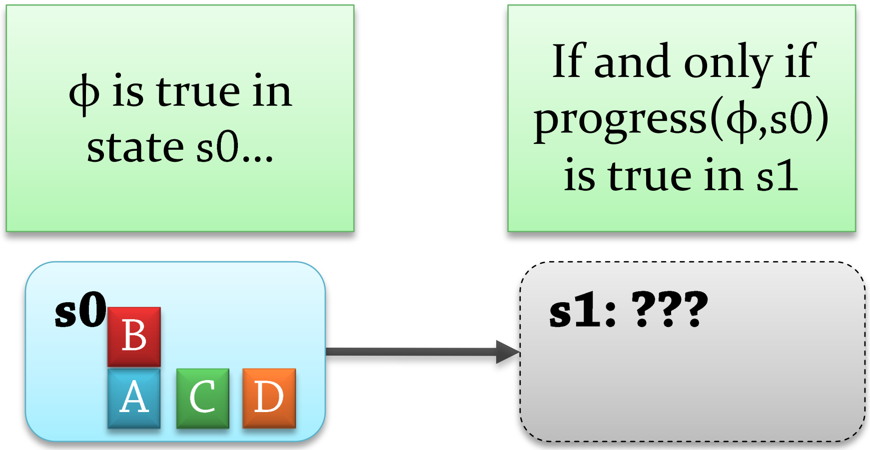
Evaluate those **parts** of the formula that refer to known states

Leave other parts of the formula to be evaluated later

If the result can be proven to be FALSE, then backtrack

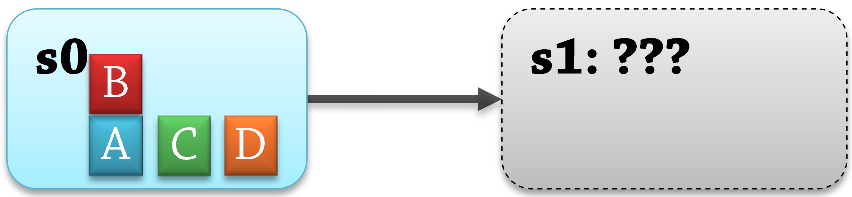
Progressing Temporal Formulas (1)

- We use **formula progression**
  - We **progress** a formula Φ through a single **state** *s* at a time
    - First the initial state, then each state generated by adding an action
  - The result is a **new formula**
    - Containing conditions that we must "postpone", evaluate starting in the **next** state



Progressing Temporal Formulas (2)

- Base case: Formulas **without** temporal operators ("on(A,B) → on(C,D)")
  - progress(Φ, s) = TRUE if Φ holds in *s* (we already know how to test this)
  - progress(Φ, s) = FALSE otherwise



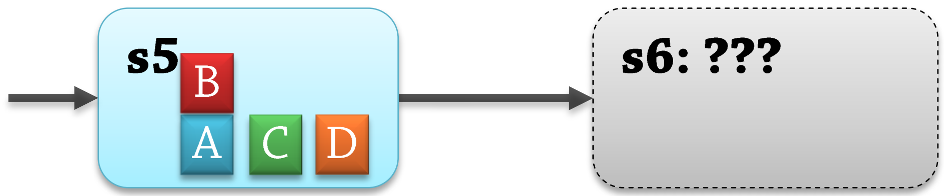
If "f → g" is true in state s0…	Then progress returns TRUE, which is true in s1
If "f → g" is false in state s0…	Then progress returns FALSE, which is false in s1

Progressing Temporal Formulas (3)

- Simple case: <u>**next**</u>
  - progress(next *f*, s) = f
    - Because ”next f” is true in this state iff f is true in the next state
    - This is by definition what progress() should return!

”next f” is true in state s5…

If and only if ”f” is true in state s6



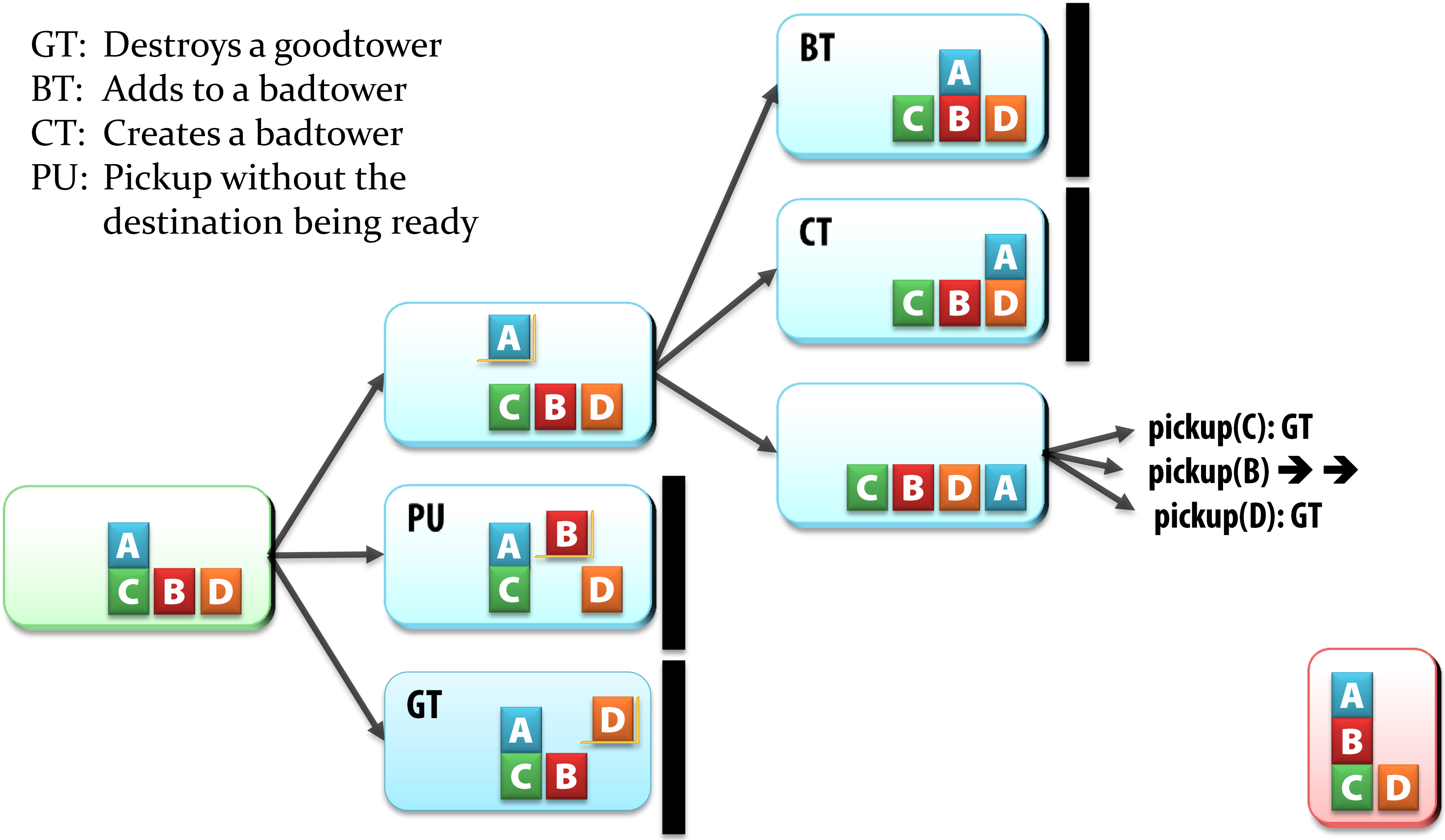
”next holding(B)” is true in s5…

iff holding(B) is true in s6

Additional cases are discussed in the book (always, eventually, until, …)

DFS with Pruning

GT: Destroys a goodtower
BT: Adds to a badtower
CT: Creates a badtower
PU: Pickup without the destination being ready



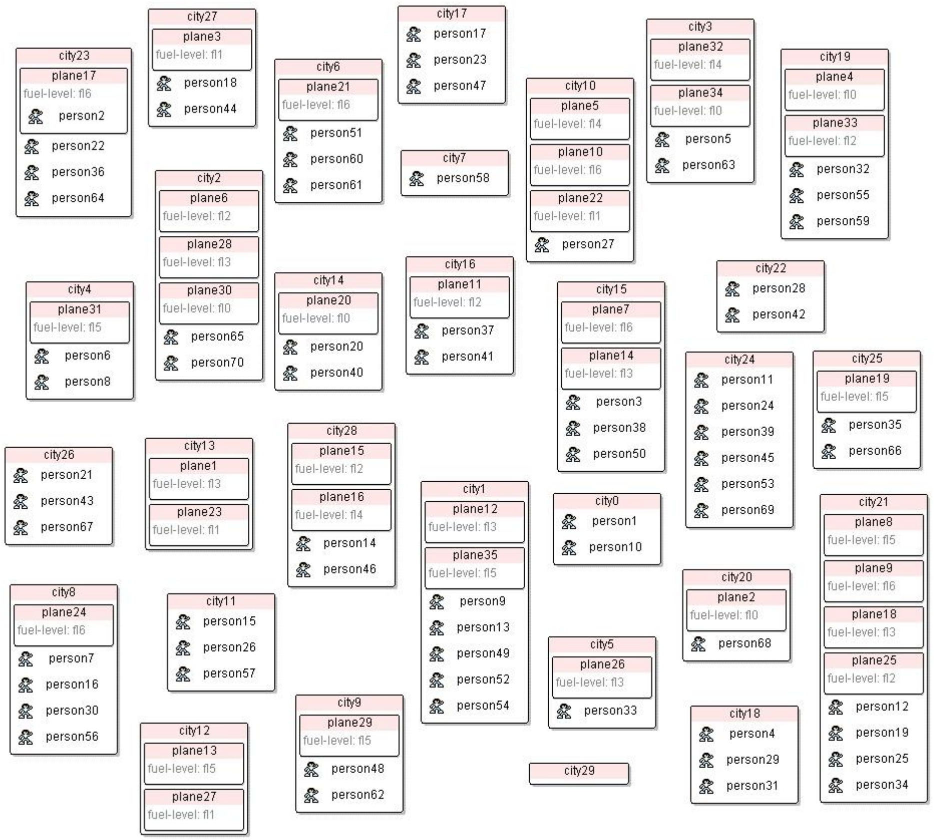
Performance

- 2000 International Planning Competition
  - **TALplanner** received the top award
    for a "hand-tailored" (i.e., domain-configurable) planner
- 2002 International Planning Competition
  - **TLplan** won the same award
- Both of them (as well as SHOP, an HTN planner):
  - Ran **several orders of magnitude** faster
    than the "fully automated" (i.e., not domain-configurable) planners
    - especially on large problems
  - Solved problems on which other planners ran out of time/memory

TALplanner: A demonstration

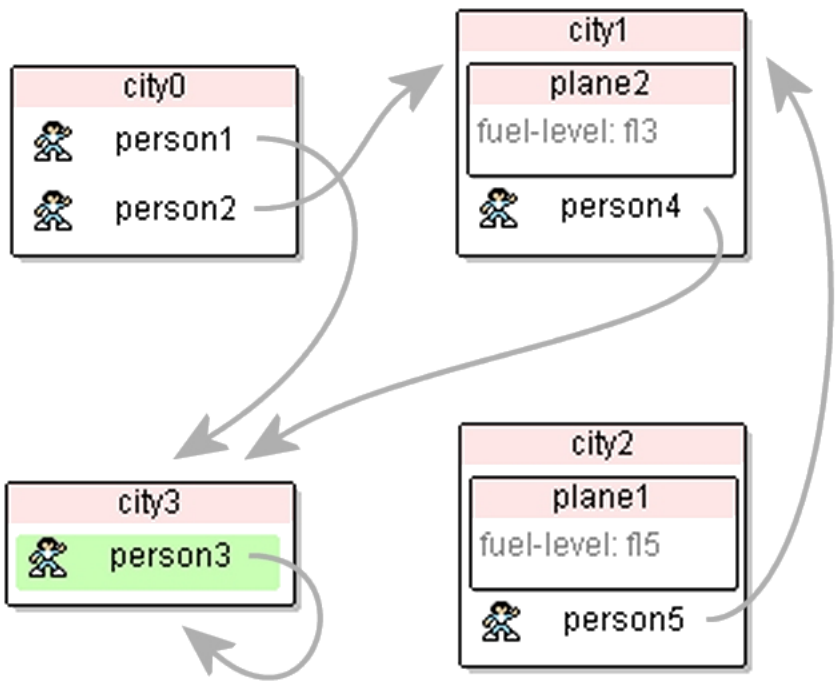
TALplanner Example Domain

- Example Domain: ZenoTravel
  - Planes move people between cities (**board**, **debark**, **fly**)
  - Planes have limited fuel level; must **refuel**
  - Example instance:
    - 70 people
    - 35 planes
    - 30 cities



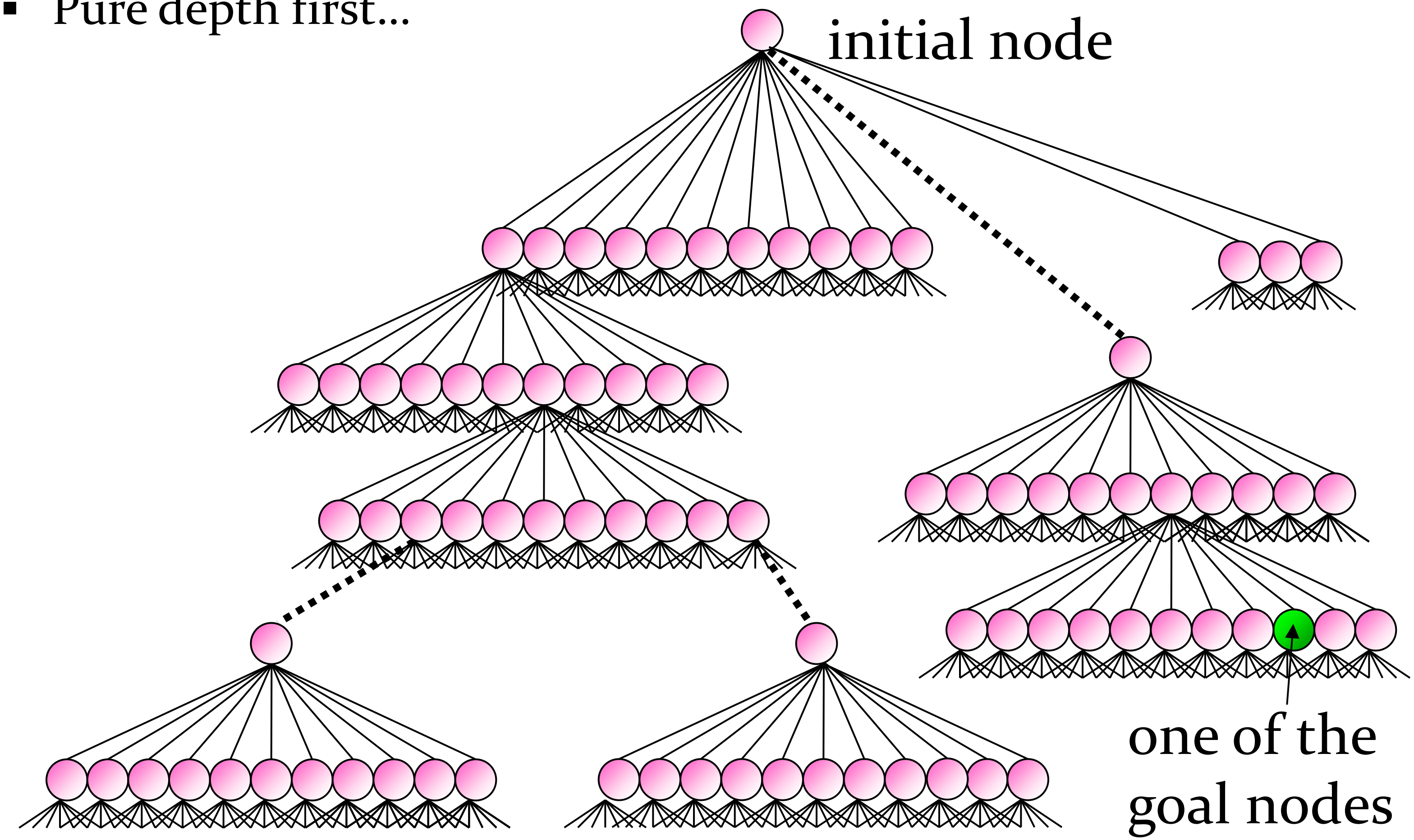
ZenoTravel Problem Instance

- A smaller problem instance



What Just Happened?

- No additional domain knowledge specified yet!
  - Pure depth first…



Control Rules

- **First problem** in the example:
  - Passengers debark whenever possible.
  - Rule: "At any timepoint, if a passenger debarks, he is at his goal."

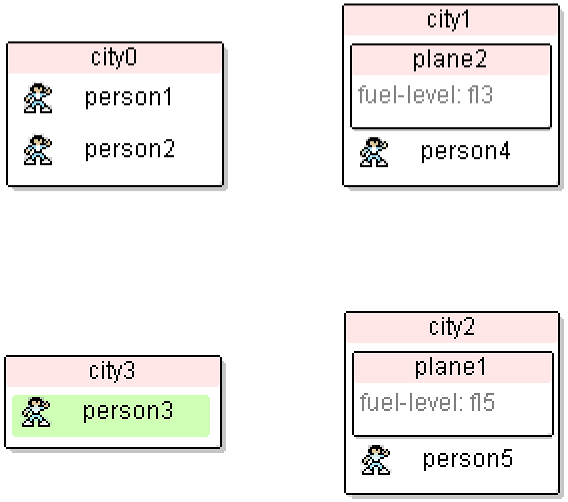
  - #**control** :name "only-debark-when-in-goal-city"
    forall t, *person*, *aircraft* [
    [t] **in**(*person*, *aircraft*) →
    [t+1] **in**(*person*, *aircraft*) ∨
    exists city [
    [t] **at**(*aircraft*, *city*) ∧
    goal(**at**(*person*, *city*))]]

[t]: "now"
[t+1]: "next"

Control Rules

- **Second problem** in the example:
  - Passengers board planes, even at their destinations
  - Rule: "At any timepoint, if a passenger boards a plane, he was not at his destination."
  - #**control** :name "only-board-when-necessary"
    forall t, *person*, *aircraft* [
    ([t] !**in**(*person*, *aircraft*) ∧
    [t+1] **in**(*person*, *aircraft*)) →
    exists *city1*, *city2* [
    [t] **at**(*person*, *city1*) ∧
    goal(**at**(*person*, *city2*)) ∧
    city1 != *city2*]]

Zeno Travel, second attempt



What's Wrong This Time?

- Only constrained passengers
- Forgot to constrain airplanes

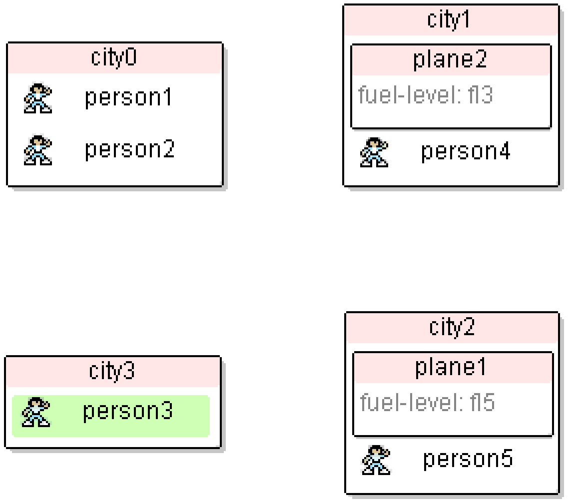
  - Which cities are reasonable destinations?

  - 1. A passenger's destination
  - 2. A place where a person wants to leave
  - 3. The airplane's destination

Control Rules

- #control :name "planes-always-fly-to-goal"
  forall t, aircraft, city [
  [t] at(aircraft, city) →
  ([t+1] at(aircraft, city)) |
  exists city2 [
  city2 != city &
  ([t+1] at(aircraft, city2)) &
  [t] reasonable-destination(aircraft, city2)]]
- #**define** [t] reasonable-destination(aircraft, city):
  [t] has-passenger-for(aircraft, city) |
  exists person [
  [t] at(person, city) &
  [t] in-wrong-city(person)] |
  goal(at(aircraft, city)) &
  [t] empty(aircraft) &
  [t] all-persons-at-their-destinations-or-in-planes]

Zeno Travel, third attempt



Progression and Execution Monitoring

Problem: Plans Will Fail!

- "No plans survive first contact with the enemy!"
  - **The environment** – does not behave as we expect it to
    - *Unusually strong wind today*

  - **Other agents** – do not behave as we want them to
    - *Someone took the last medicine crate from this depot*

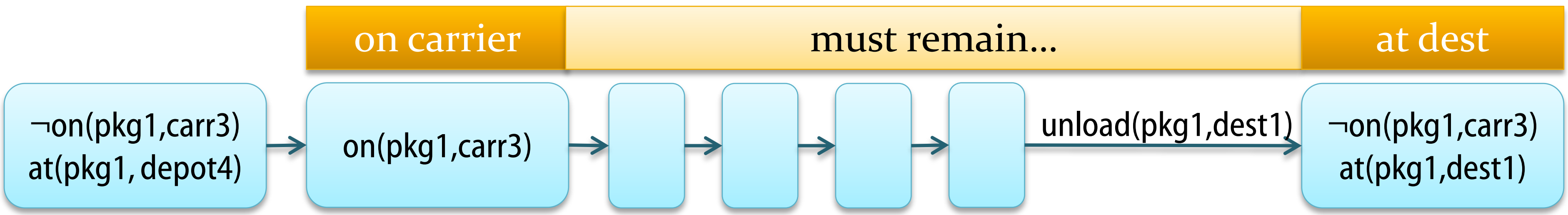
  - **Ignorance and mistaken beliefs** – our models are not perfect
    - *We thought we could lift 4 crates – we could only lift 3*
  - **Sensors and actuators** – our hardware is not perfect
    - *A crate was dropped during flight*

Partial Solutions

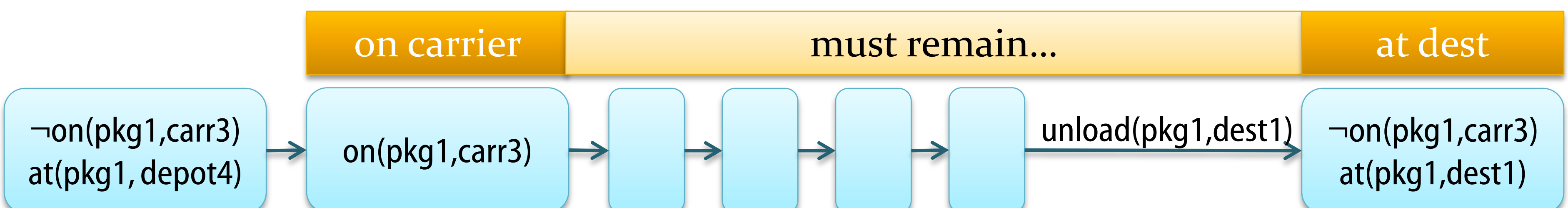
- **Execution monitoring** is important!
  - **Acknowledge** that plans will fail
  - **Detect** problems at runtime
  - **Distinguish** failure types and **recover**
  - We will show **one specific example** of how you can do this

Idea

- Idea: Similar to **control formulas**
  - At plan time we **predict** what will happen
    - **Control formulas** violated ➔ backtrack – *make the right decisions*

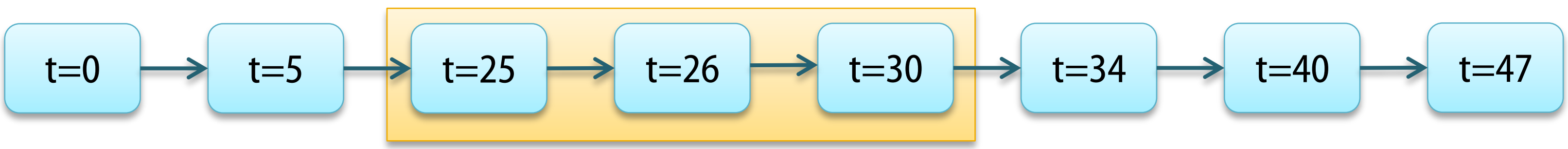


  - And at runtime, we **sense** what actually happens
    - We can use very similar **monitor formulas** to describe what **should happen** – *detect failures*



Metric Temporal Logic

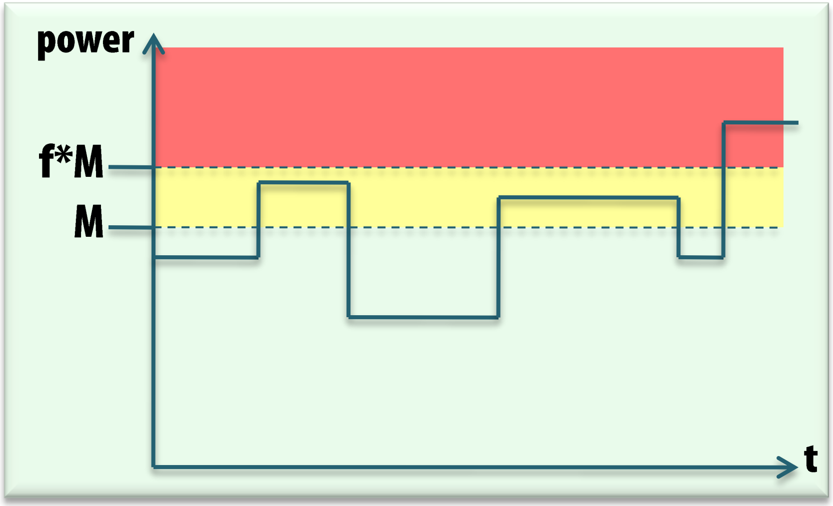
- Since timing is important, a **metric** temporal logic is used
  - □ [t1,t2] f - **always** f
    - f holds in all states at a time of [t1,t2] from "now"
    - Example: At t=5, we specify the formula □ [21,28] f
    - Then f should hold in all states with timestamps in [26,33]



  - ◇ [t1,t2] f - **eventually** f
    - f holds in **some** state whose distance from "now" is in [t1,t2]
  - f_1 U [t1,t2] f_2 - f_1 **until** f_2
    - f_2 holds in some state at a distance of [t1,t2] from "now", and f_1 holds until then

Global Formulas

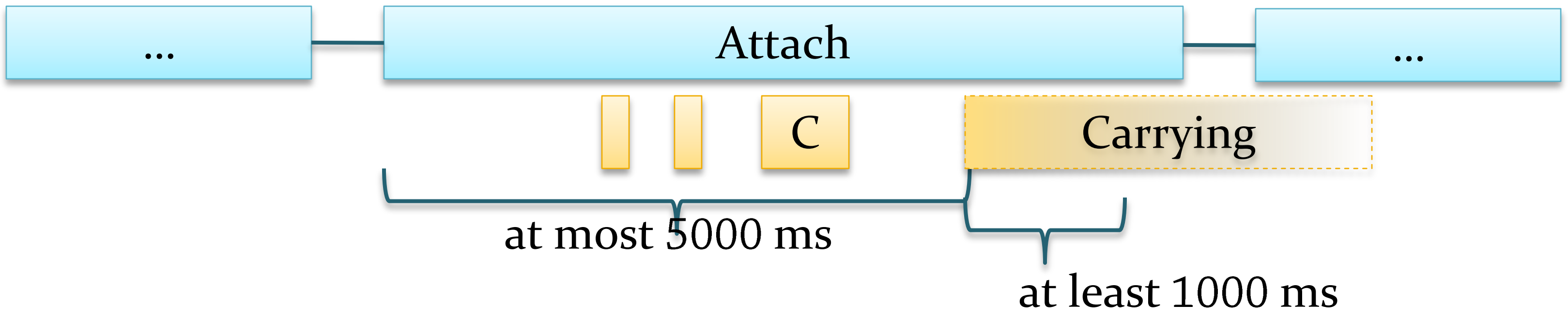
- <u>**Global**</u> monitor formulas are always active
  - Planner ensures <u>**predicted**</u> power usage within limits
  - Monitor ensures <u>**actual**</u> power usage within limits
    - **always forall uav. power(uav) ≤ M**
  - Very expressive formalism!
    - May exceed the nominal maximum by a factor of **f**, for a limited time, in certain conditions
    - **always forall uav.**
      power(uav) > M → (
      power(uav) ≤ f*M
      until [0,τ]
      always [0,τ′] power(uav) ≤ M
)



Operator-Specific Formulas

- Plan provides context: **Operator-specific** formulas
  - Example: A desired effect must occur, *and not just temporarily*
    - Temporary electromagnet lock ➔ "carrying" temporarily true
    - operator **attach**(*uav*, *crate*, *x*, *y*, …)
      :monitor **eventually [0,5000] always [0,1000] carrying(*uav*, *crate*)**

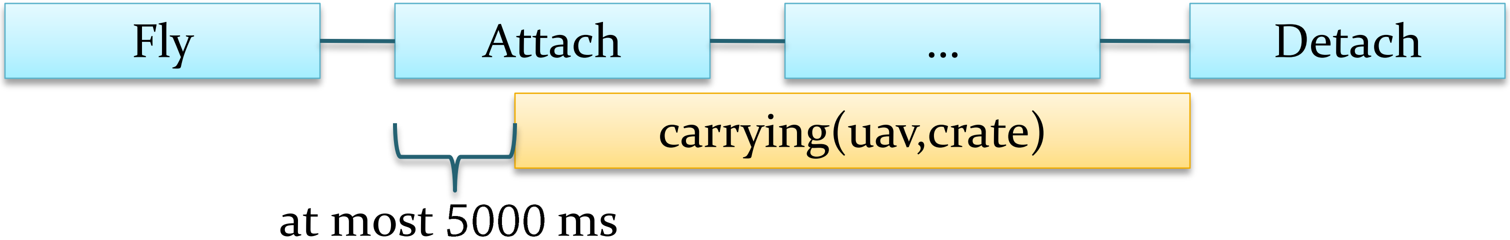
(Time in ms)



Introspection

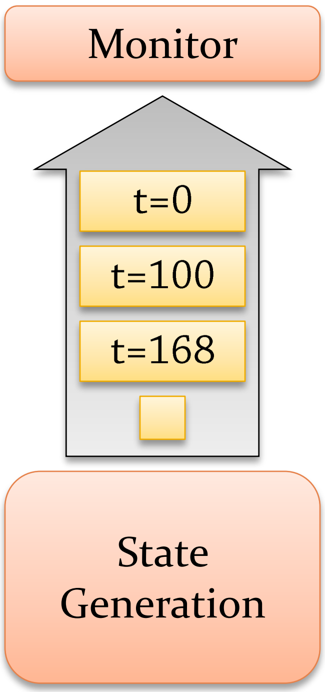
- Introspection: What operators are being executed?
  - Operator **detach**(*uav*, *crate*) ➔ flag **executing-detach(*uav, crate*)**

- Conditions can span multiple actions
  - Attach a crate ➔ remain attached until explicitly detached
    - operator **attach**(*uav*, *crate*)
      :monitor **executing-attach(*uav, crate*) until [0,5000]**
      (carrying(*uav, crate*) until executing-detach(*uav, crate*))
    - Operator-specific, but remains after execution of ***this*** operator



Progression

- Monitoring is an incremental process
  - States are generated at regular or irregular intervals
    - Using multiple sensors,
      sensor fusion techniques,
      state synchronization, ...
  - Formulas are tested against states using **progression**
    - ϕ holds in [s0,s1,...] iff Progress(ϕ, s0, Δt) holds in [s1,...], where Δt is the *duration* of state s1
    - Progress() returns ⊥ ➔ proven violation



φ is true
starting in state
so...

If and only if
progress(φ,s0,Δt)
is true
starting in s1