Linköping University

Automated Planning

Domain-Configurable Planning: Planning with Control Formulas

Jonas Kvarnström

Department of Computer and Information Science

Linköping University

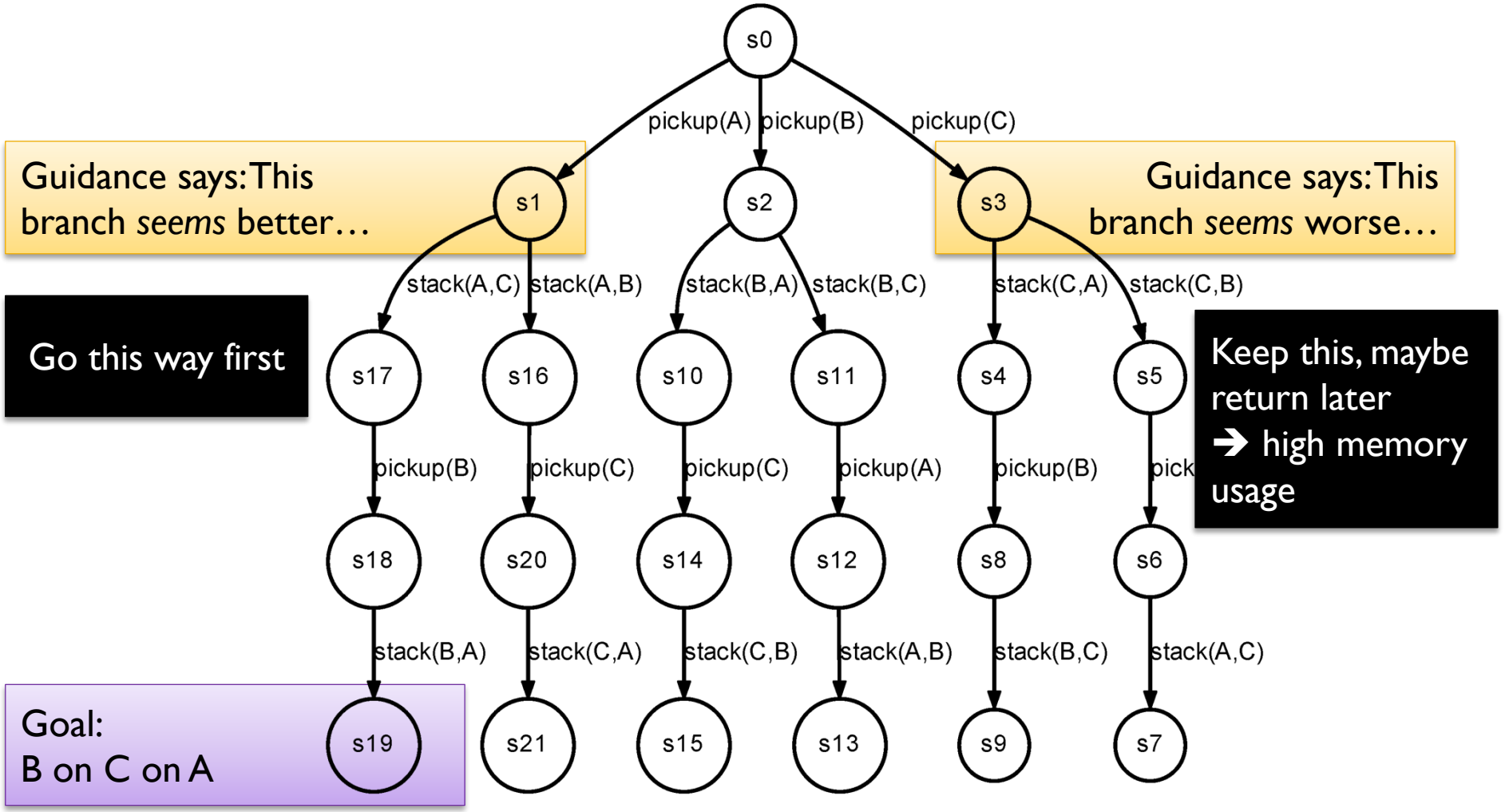
jonas.kvarnstrom@liu.se – 2019

Domain-Configurable: Prioritization and Pruning

Two Kinds of Search Guidance (1)

jonkv@ida

Prioritization: Which part of a search tree should be visited **first?**
Could use heuristic functions, could use other methods…



Prioritization

jonkv@ida

- Properties of prioritization:
  - We can always **return** to a node later
  - No need to be **absolutely** certain of your priorities

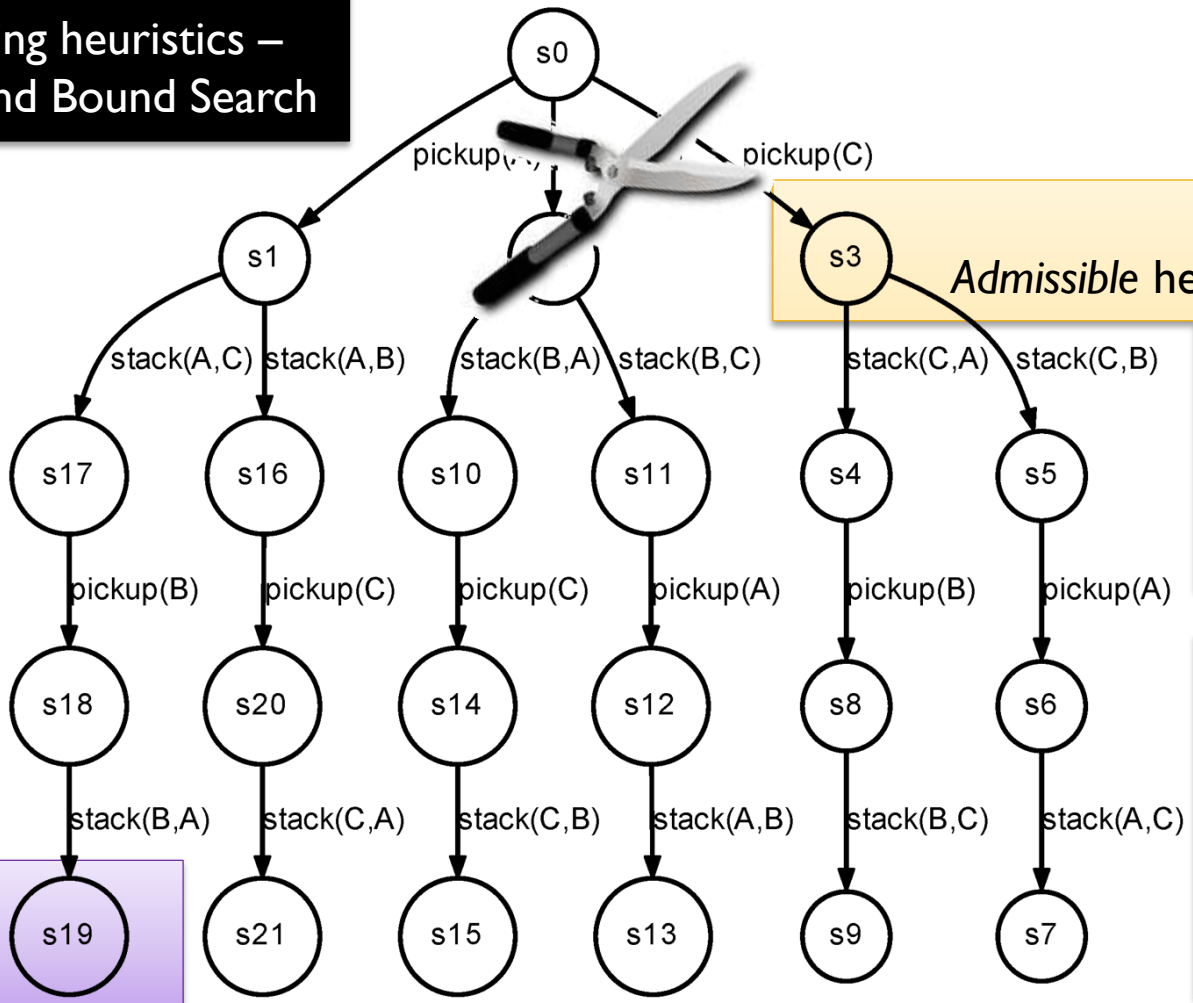
- **This** is why many domain-independent heuristics work well
  - Provide *reasonable* advice in *most* cases

Two Kinds of Search Guidance (2)

jonkv@ida

Pruning: Which part of a search tree are **definitely useless**?
Prune them!

Can be done using heuristics –
example: Branch and Bound Search



Cost g(n)=1
Admissible heuristic h(n)=4

Any solution below s3 will cost at least 5

Prune!
Never consider the node or its descendants again…

We found a solution of cost 4

Pruning

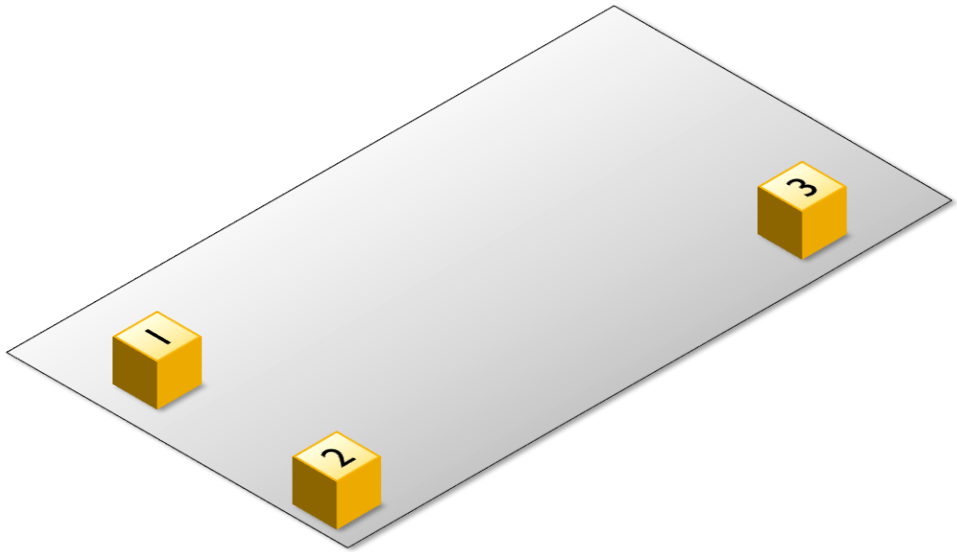
jonkv@ida

- Can we prune when we search for the **first** solution?
  - A single mistake may *remove all paths to solutions*
  - ➔ Difficult to find good *domain-independent* pruning criteria

Example: Emergency Services Logistics

jonkv@ida

- 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!**

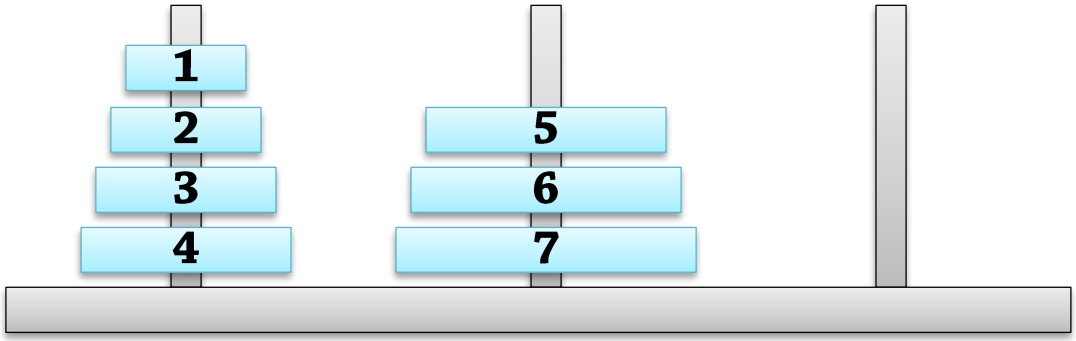
The **branch** beginning with pickup(crate1) could be **pruned** from the tree!
How do we **detect** this in a domain-independent way?

Example: Towers of Hanoi

jonkv@ida

Should we <u>always</u> 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)



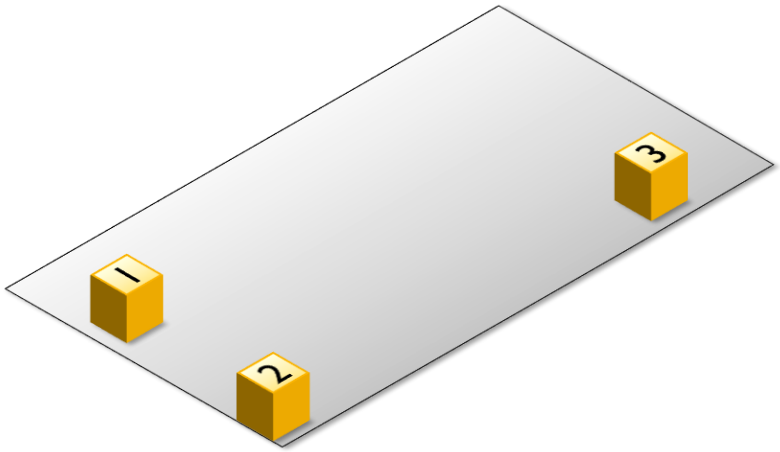
- Moving disk 1 to the third peg is **<u>possible</u>** but "destroys" a goal fact: on(1,2)
  - Is this also **<u>obviously stupid</u>**?
  - No, it is **<u>necessary</u>**! Disk 1 is blocking us from moving disk 4…

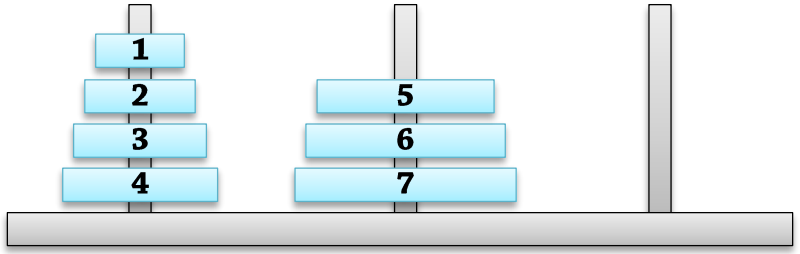
Heuristics

jonkv@ida

➔ Heuristics **may or may not** detect:

Picking up crate1 is bad	Moving disk1 is good





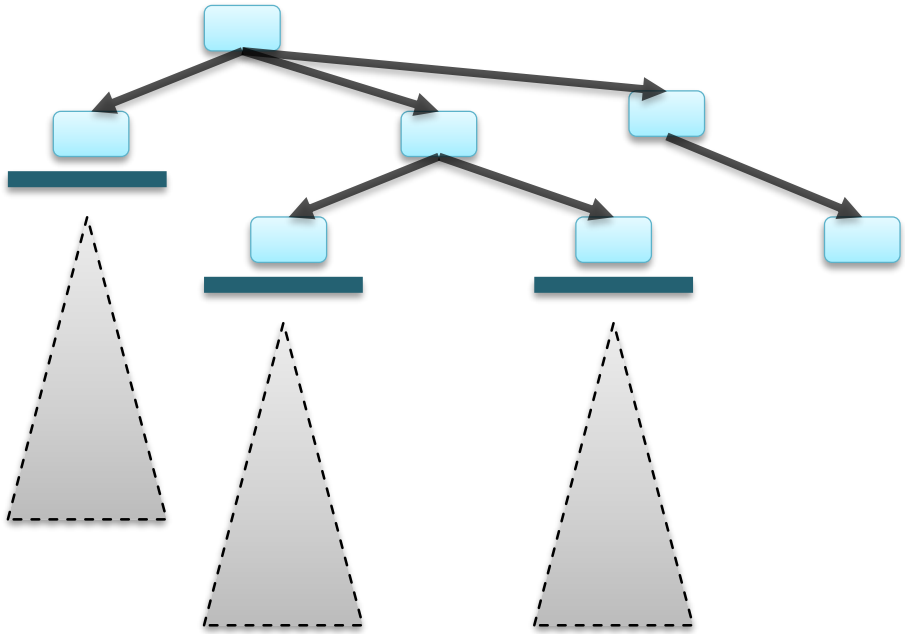
Will generally depend on **the entire state**
+ which **alternative states** exist,
not just the fact that you "destroy goal achievement"

Might **delay** investigating either alternative for a while,
return to try this later

Pruning

jonkv@ida

- With a **domain-configurable planner**:
  - We *could* provide *domain-specific heuristics*
    - Strongly discouraging the destruction of goals in Emergency Services Logistics
    - Would keep the *option* to investigate such actions later (not necessary!)
  - We can directly provide stronger domain-specific **pruning criteria**



Planning with Control Formulas

Control Formulas

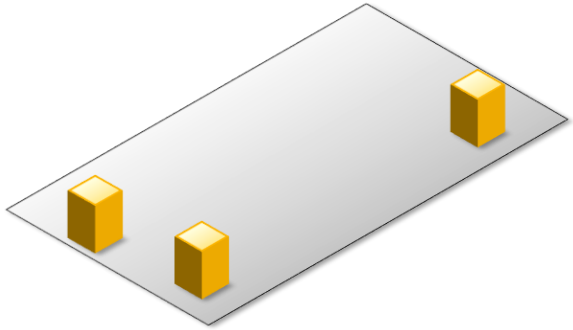
jonkv@ida

- **Control formulas**: One way of specifying **when** to prune
  - Motivation
  - Examples
  - Formalism
  - Evaluation of control formulas

Precondition Control

jonkv@ida

Simplest control information: **Precondition Control**



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

How to express this, given that the goal requires at(*crate*,*loc*)?

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

Supported by all planners

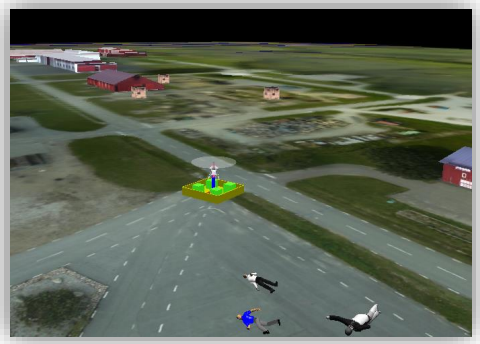
- Alternative 2: **New language extension** "goal(ϕ)"
  - Evaluated in the set of goal states, not in the current state
  - **precond**: ¬**goal**(at(*crate*, *location*))

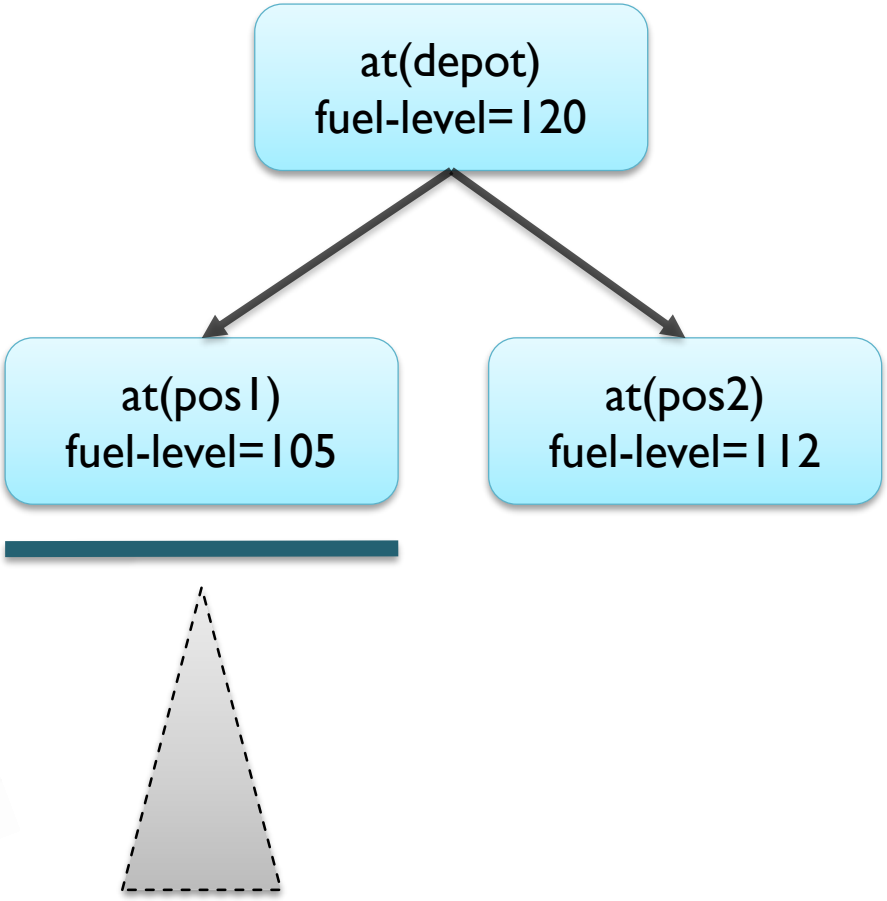
Requires extensions, but more convenient

State Constraints

jonkv@ida

- A UAV should never be where it **can't reach** a refueling point
  - Can't possibly extend such plans into solutions



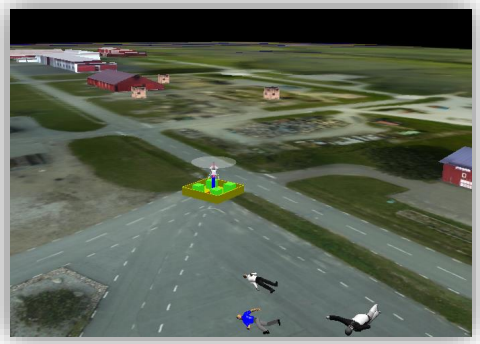


Reaching nearest refueling point requires 108 fuel

State Constraints

jonkv@ida

- A UAV should never be where it **can't reach** 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 **preconditions** again?

Must be verified for **every** action:
fly, scan-area, take-off, …

Must be checked even when the UAV is idle, hovering

Inconvenient!

Using **state constraints**?

Defined **once**,
applied to **every generated state**

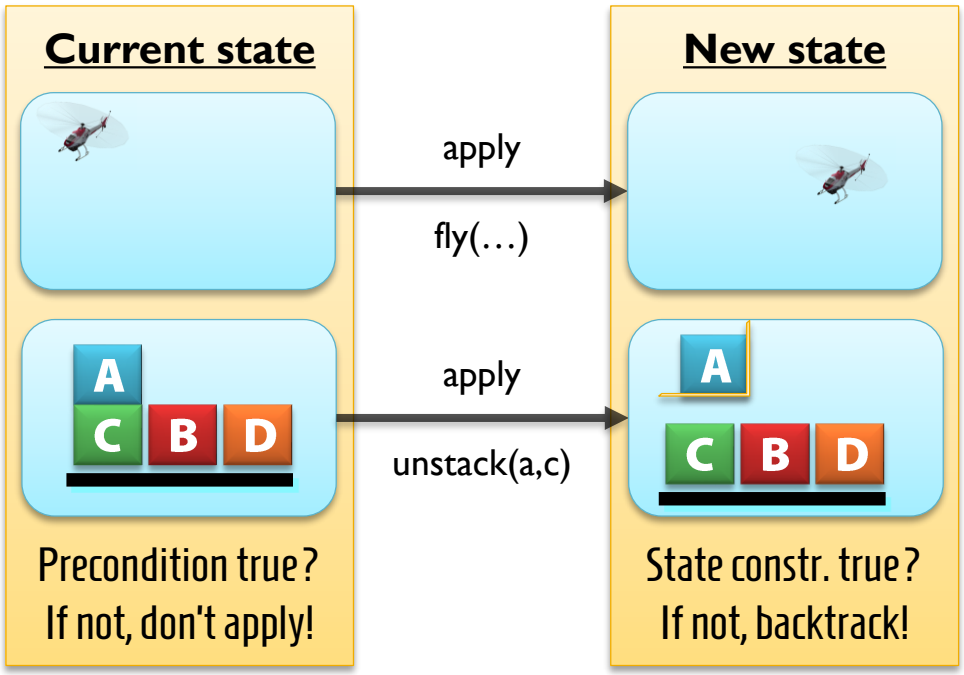
$\forall(u:\text{uav})\ [$
  $\exists(rp:\text{refueling-point})\ [$
    $\text{dist}(u,rp) * \text{fuel-usage}(u) < \text{fuel-avail}(u)$
  $]$
$]$

Comparatively simple extension!

Testing State Constraints

jonkv@ida

- Testing such **state constraints** is simple
  - Apply an action ➔ new state is generated
    - Formula false in that state ➔ Prune!
  - Similar to preconditions
    - But tested in the state **after** an action is applied, not **before**!



Temporal Conditions (1)

jonkv@ida

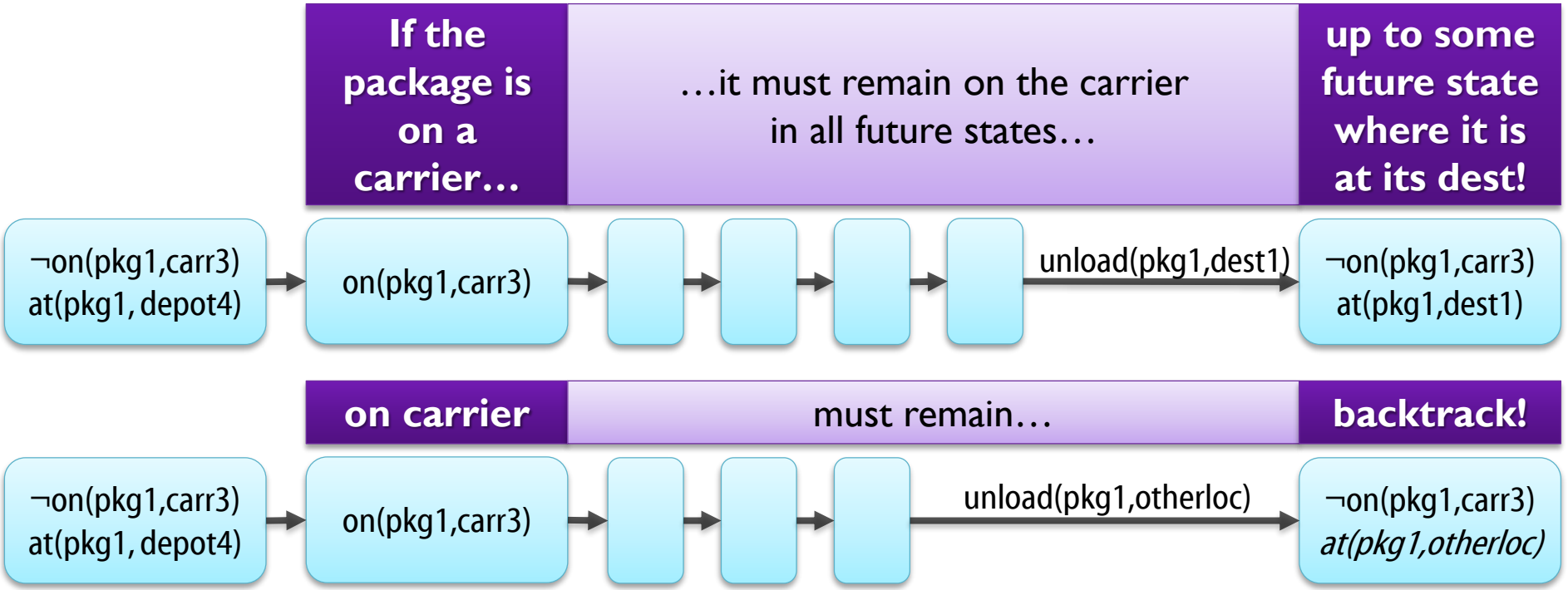
- A package on a carrier should **remain there** until it reaches its destination
  - For any plan π where we move it prematurely, there is a more efficient plan π' where we don't

How to express this as a single formula?

Temporal Conditions (2)

jonkv@ida

- "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

jonkv@ida

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
    - $\Diamond f$ – (**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
    - $f_1 \, U \, f_2$ – (**until** f_1 f_2) — (f_2 is true now), **or** (f_2 is true in some future state s' and f_1 is true in all states until/before 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

jonkv@ida

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

(forall (?var) (type-predicate ?var) ϕ):
$\forall v.\, \text{type-predicate}(v) \rightarrow \phi$
For all values of ?var that satisfy type-predicate, ϕ must be true

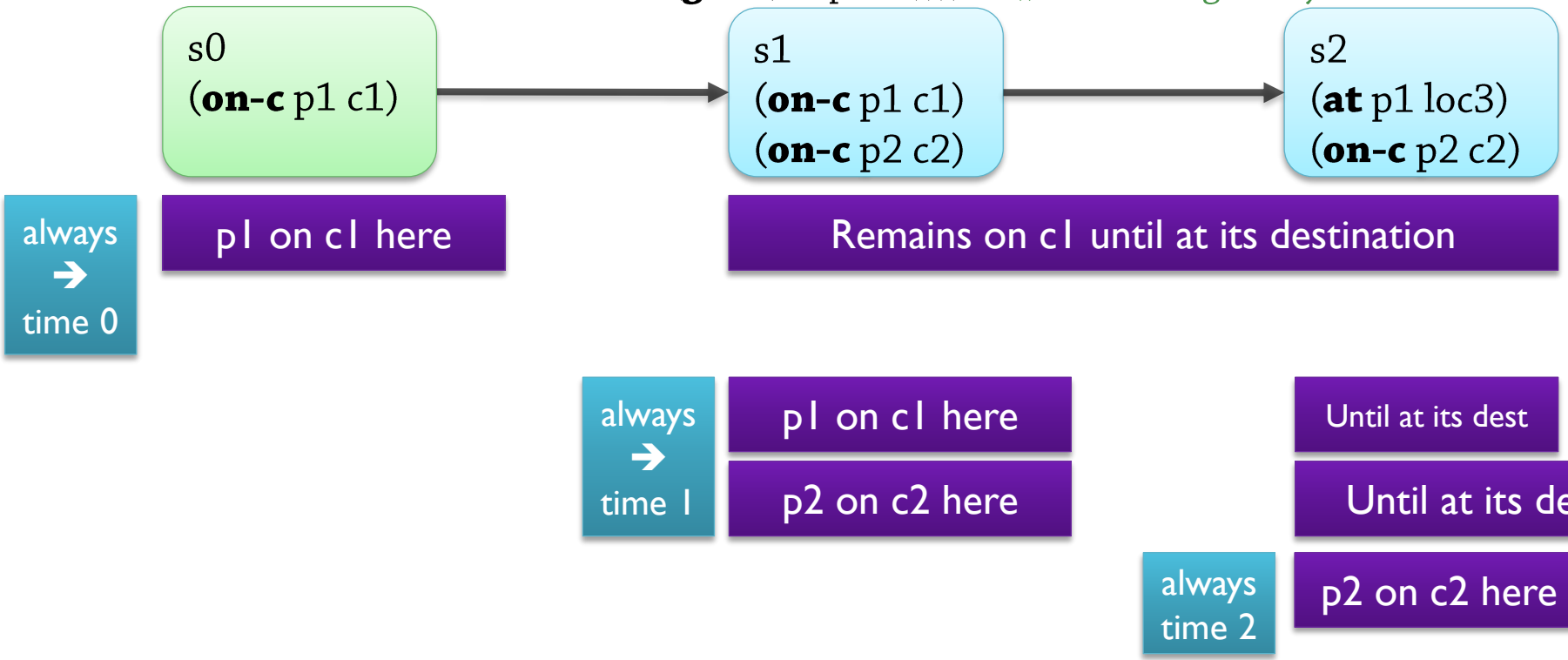
  - (**always**
    (**forall** (?c) (carrier ?c) ;; For all carriers
    (**forall** (?p) (package ?p) ;; For all packages
    (**implies**
    (on-carrier ?p ?c) ;; If the package is on the carrier
    (**until** (on-carrier ?p ?c) ;; ...then it remains on the carrier
    (**exists** (?loc) ;; until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

  - Should be true **starting in the initial state**

Control Formula

jonkv@ida

- (**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

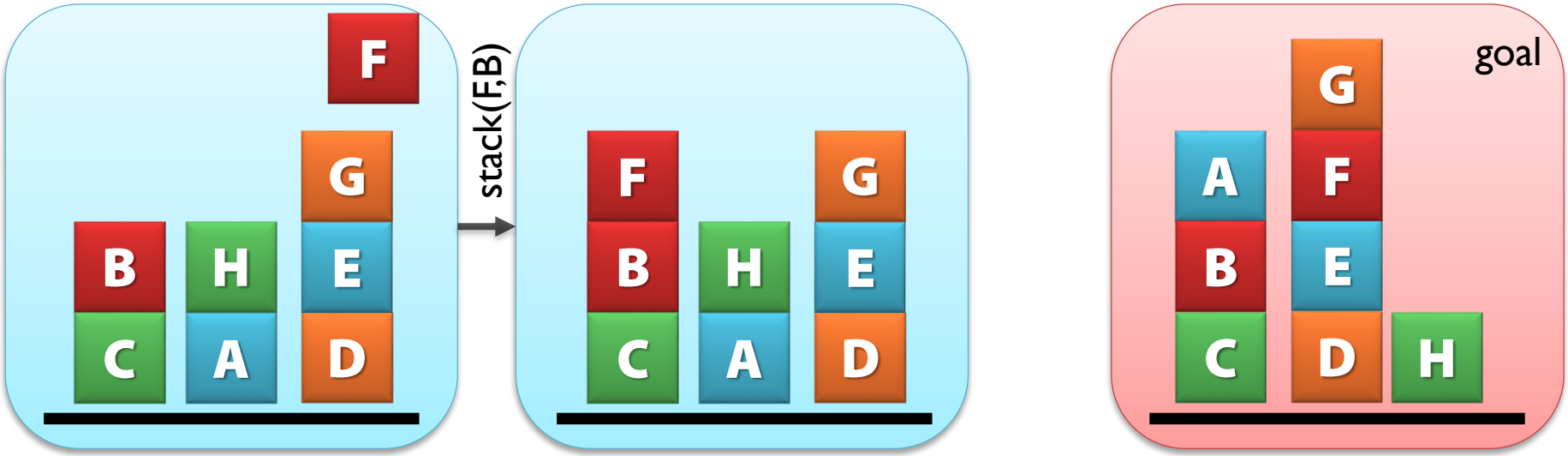


Finding Control Formulas

Blocks World

jonkv@ida

- 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:

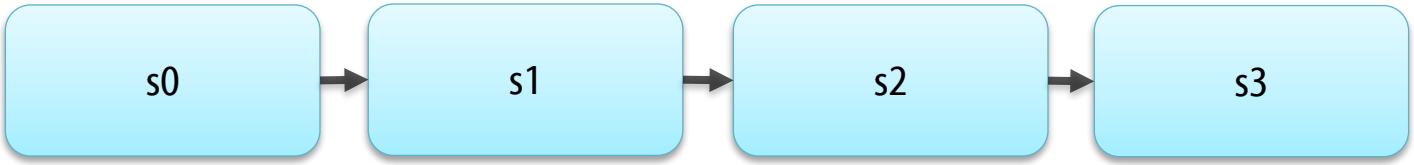


  - Placing **F on top of B** is stupid, because we'll have to remove it later
    - Would have been better to put F on the table!
  - Conclusion: Should not **extend** a **good tower** the wrong way
    - *Good tower*: a tower of blocks that will never need to be moved

Blocks World Example (continued)

jonkv@ida

- 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
    )))



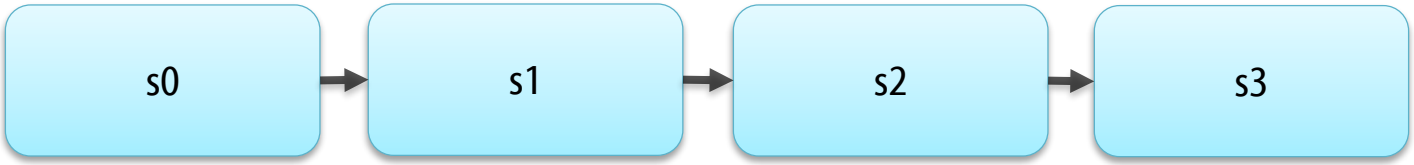
goodtower(x)? ➔ clear(x) or goodtower(y)

What about the rest?

Blocks World Example (continued)

jonkv@ida

- 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
    ))))



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

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

goodtower(x)? ➔ clear(x) or

Supporting Predicates

jonkv@ida

- Some planners allow us to **define** a predicate recursively
  - **goodtowerbelow**(x) means we **will not have to move** 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

goal

Supporting Predicates

jonkv@ida

- **goodtower**(*x*) means *x* is the block at the top of a good tower
  - *goodtower*(*x*) ⇔ *clear*(x) ∧ ¬ GOAL(*holding*(*x*)) ∧ *goodtowerbelow*(*x*)

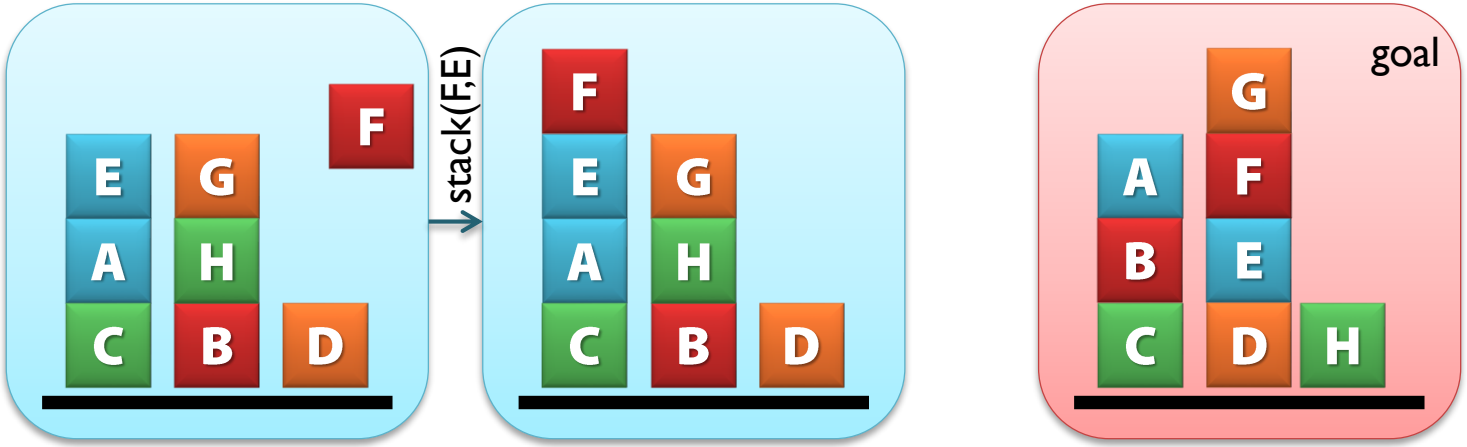
- **badtower**(*x*) means *x* is the top of a tower that isn't good
  - *badtower*(*x*) ⇔ *clear*(*x*) ∧ ¬*goodtower*(*x*)

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

Blocks World

jonkv@ida

- Step 2: Is this stupid?

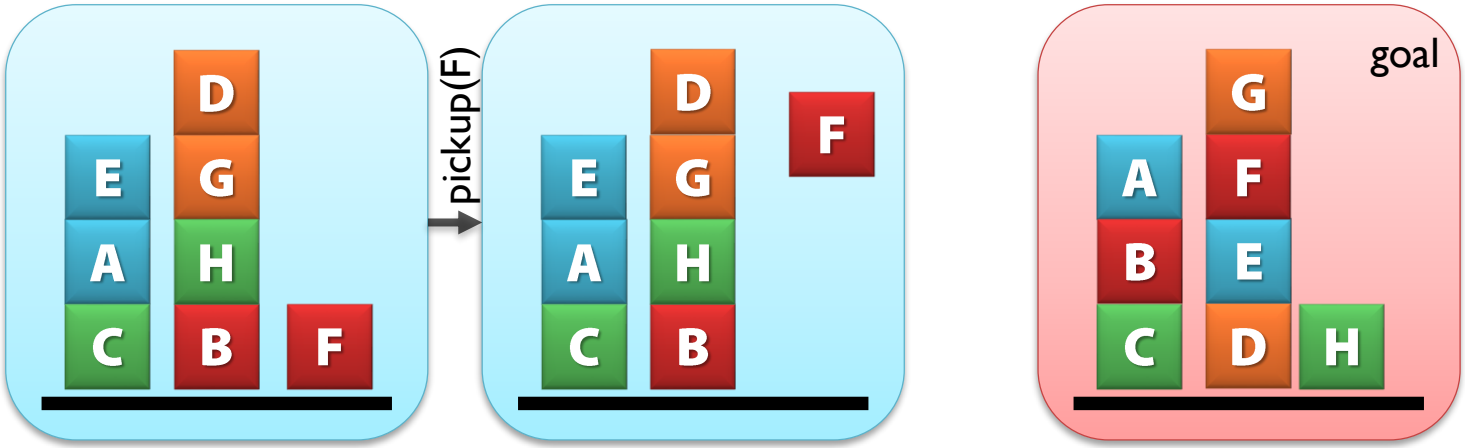


- Placing **F on top of E** is stupid, because we have to move E later…
  - Would have been better to put F on the table!
  - But E was not a goodtower, so the previous rule didn't detect the problem
- Never put anything on a badtower!
  - (**always**
    (**forall** (?x) (clear ?x) ;; For all blocks 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

jonkv@ida

- Step 3: Is this stupid?



  - **Picking up F** 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

jonkv@ida

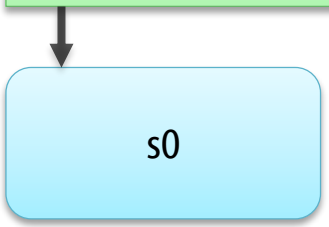
- How do we decide **when to prune** the search tree?
  - Obvious idea:
    - Take the **state sequence** corresponding to the **current action sequence**
    - **Evaluate** the formula over that sequence
    - If it is false: Prune / backtrack!

Evaluation 1

jonkv@ida

- 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 zero boats)"**

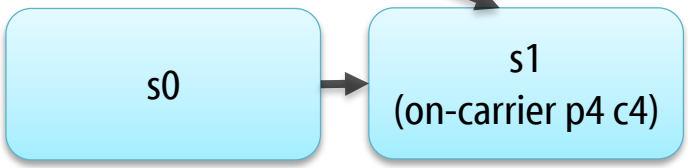
Evaluation 2

jonkv@ida

- 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?

jonkv@ida

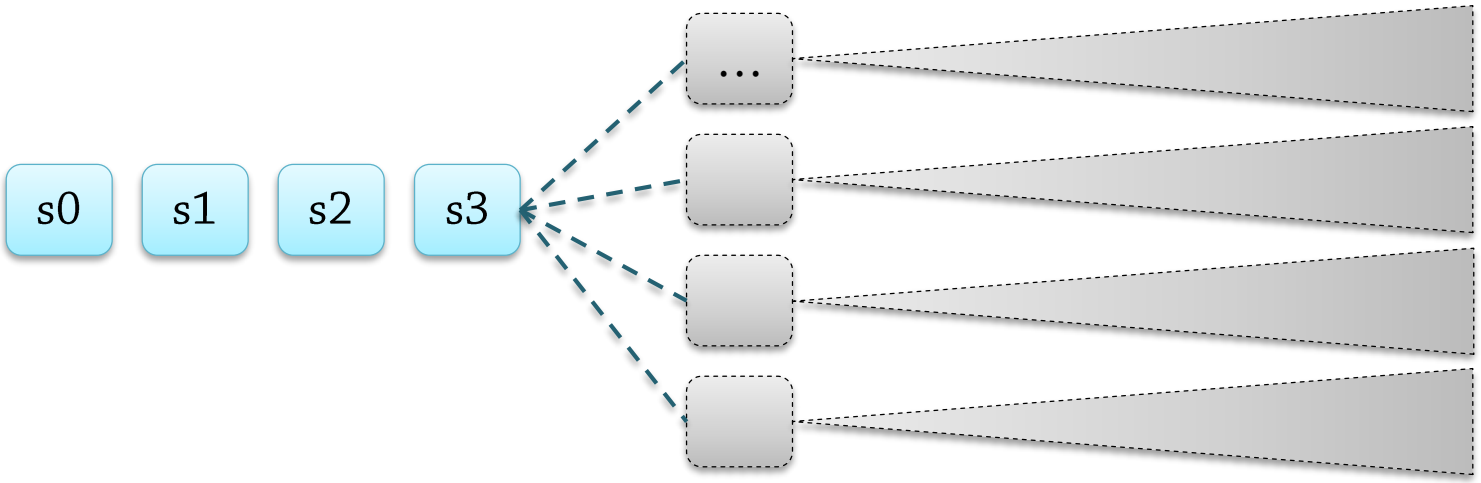
- We had an **obvious** idea:
  - Take the state sequence corresponding to the current plan
  - Evaluate the formula over that sequence
  - If it is false: Prune / backtrack!

- 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

jonkv@ida

- 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

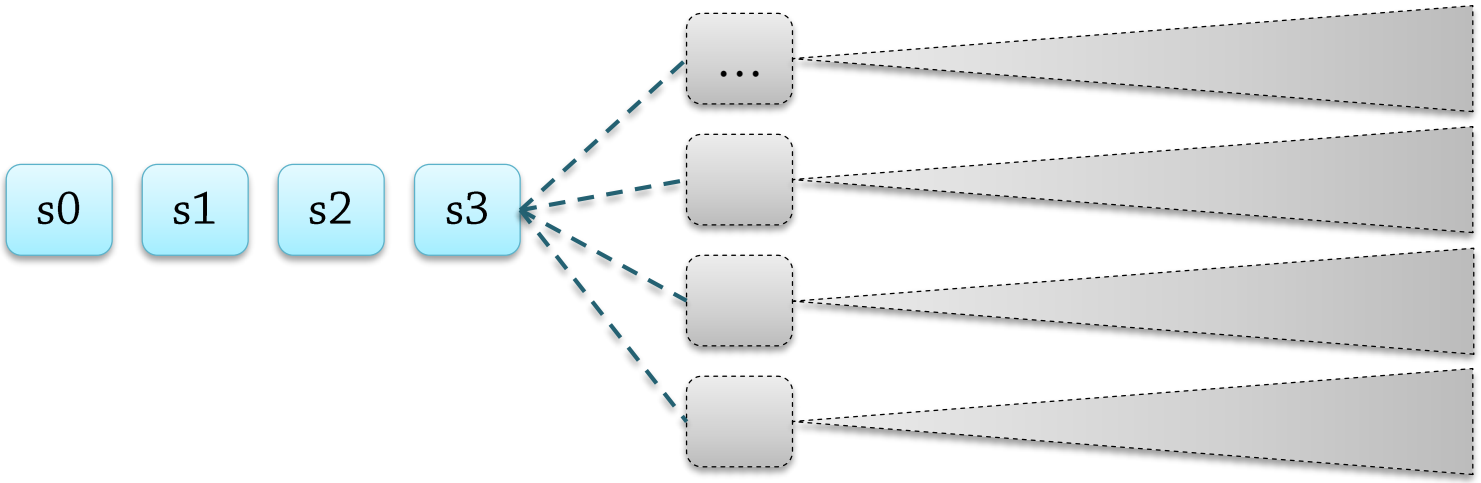
We only know **some** of those states

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

Analysis 2

jonkv@ida

- 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

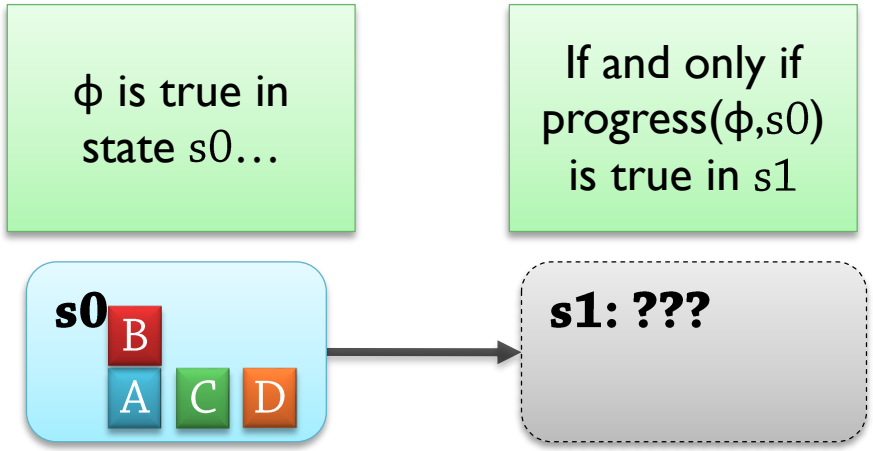
Leave other parts of the formula to be evaluated later

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

Progressing Temporal Formulas (1)

jonkv@ida

- 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)

jonkv@ida

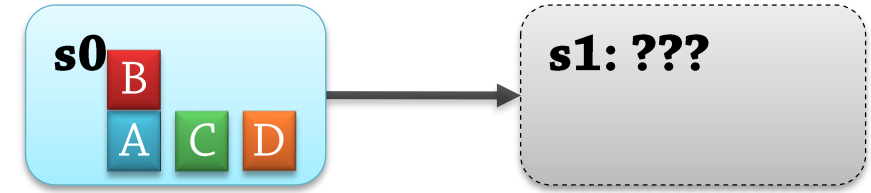
- More intuitions?
  - Suppose you are reading a book. A *page* is analogous to a *state*.

"The figure appears 5 pages from here" is true on page 7	⟺	"The figure appears 4 pages from here" is true on page 8
"Fig 1 appears 3 pages from here and fig 2 appears 5 pages from here" is true on page 7	⟺	"Fig 1 appears 2 pages from here and fig 2 appears 4 pages from here" is true on page 8
"Starting where I am now, there's a figure on every page" is true on page 7	⟺	There actually is a figure on page 7, and "Starting where I am now, there's a figure on every page" is true on page 8
"If there's a figure on this page, there is no figure on the next page [otherwise I don't care]" is true on page 7	⟺	Check if there *is* a figure on this page. If so: f="There's no figure on this page". Otherwise: f="true". Check whether f is true on page 8.

Progressing Temporal Formulas (3)

jonkv@ida

- Base case: Formulas **without** temporal operators ("on(A,B) → on(C,D)")
  - Must be true **here**, in **this** state
  - 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

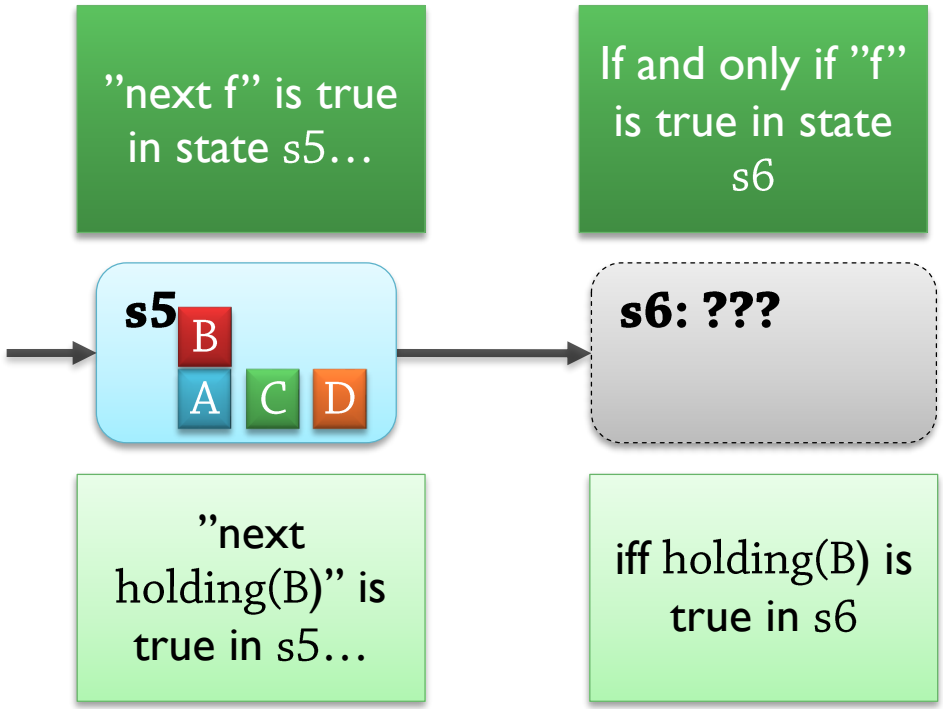
If "f → g" is false in state s0…

Then progress returns FALSE, which is false in s1

Progressing Temporal Formulas (4)

jonkv@ida

- Simple case: **next**
  - 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!



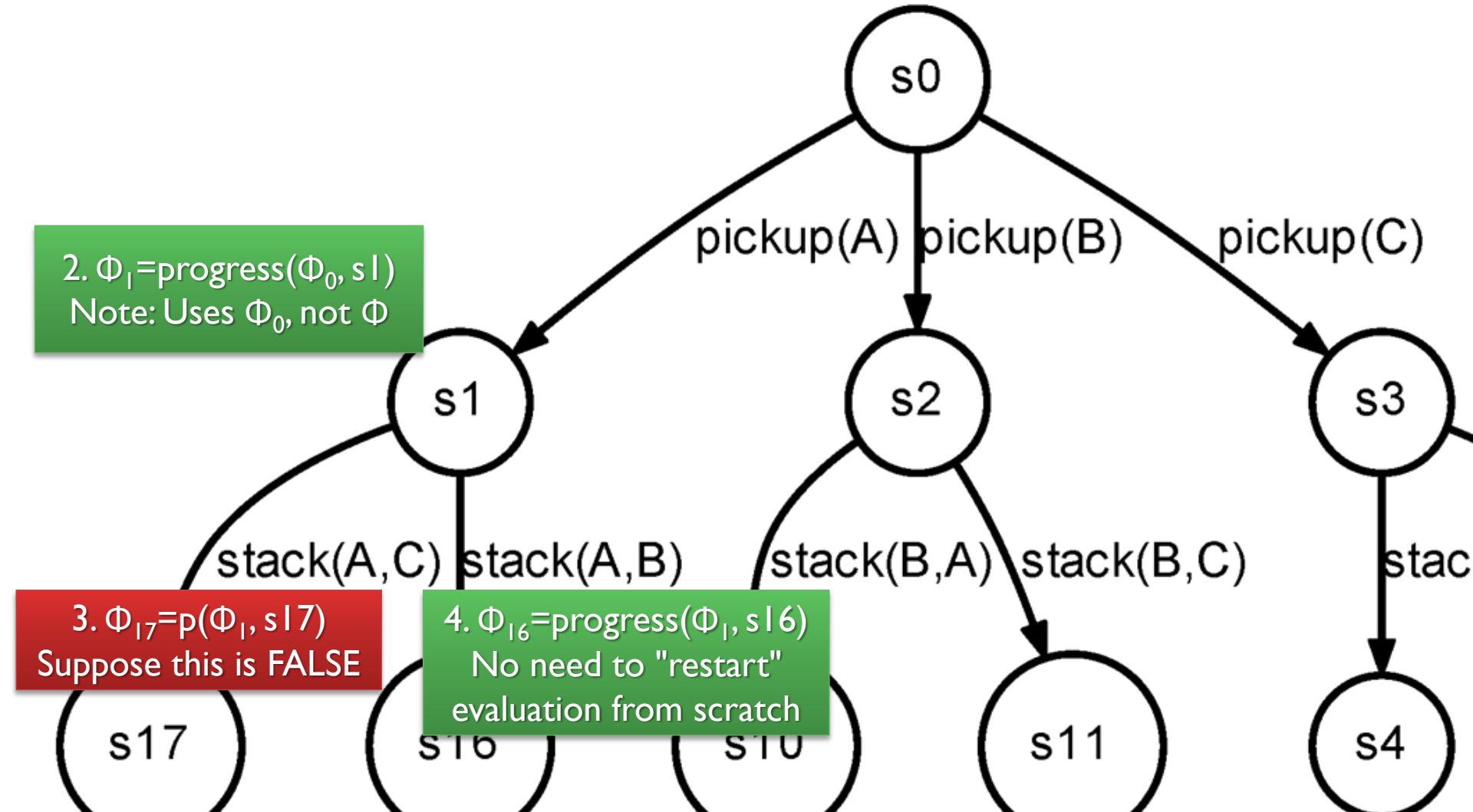
Additional cases are discussed in the book (always, eventually, until, …)

Progression in Depth First Search

jonkv@ida

Φ = original control formula

1. Calculate and store Φ_0=progress(Φ, s0)
Resolves all references to values in state s0!



2. Φ_1=progress(Φ_0, s1)
Note: Uses Φ_0, not Φ

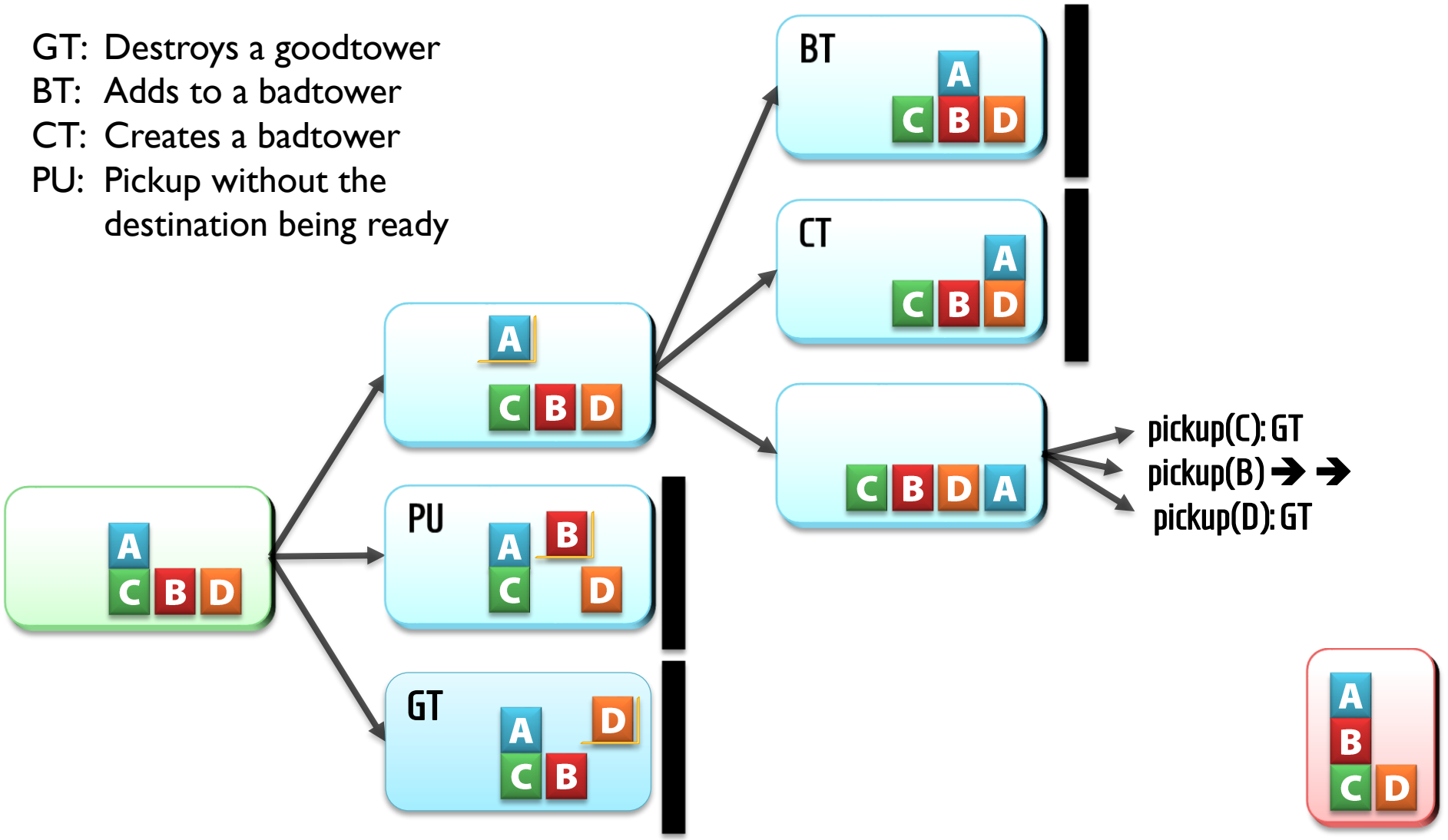
3. Φ_{17}=p(Φ_1, s17)
Suppose this is FALSE

4. Φ_{16}=progress(Φ_1, s16)
No need to "restart" evaluation from scratch

DFS with Pruning

jonkv@ida

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



Performance

jonkv@ida

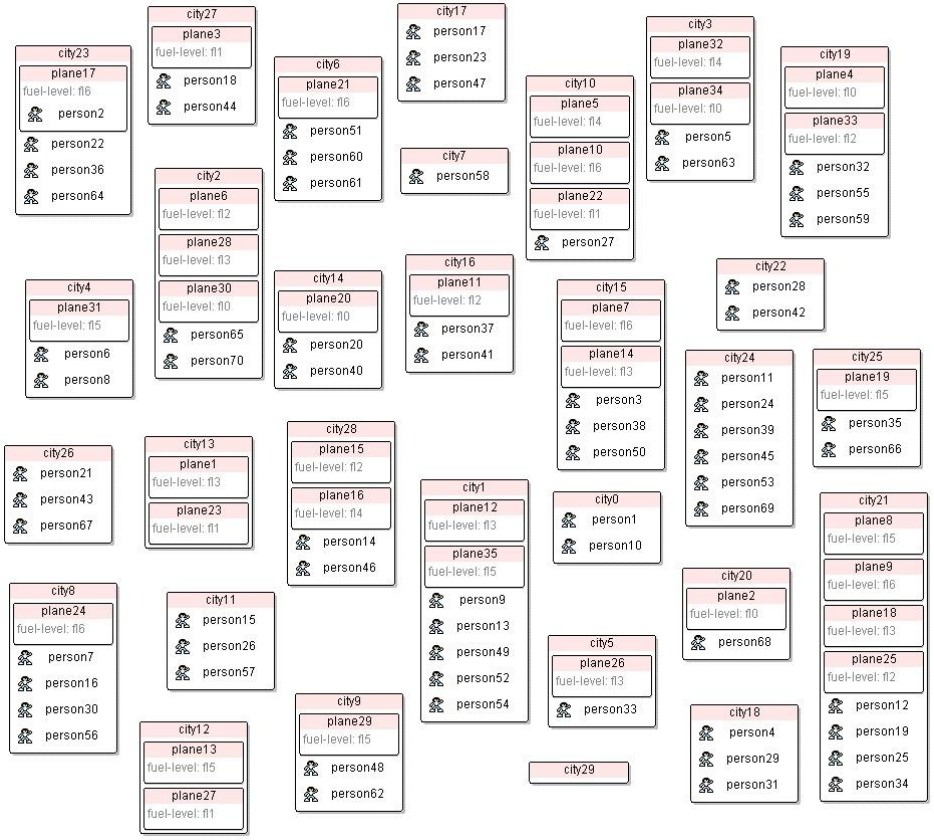
- 2000 International Planning Competition
  - **TALplanner** received the top award for a "hand-tailored" (i.e., domain-configurable) planner
- 2002 International Planning Competition
  - **TLplan** won the same award
- Both of them (as well as SHOP, an HTN planner):
  - Ran **several orders of magnitude** 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
  - Required a **considerably greater modeling effort** for each planning domain

TALplanner: A demonstration

TALplanner Example Domain

jonkv@ida

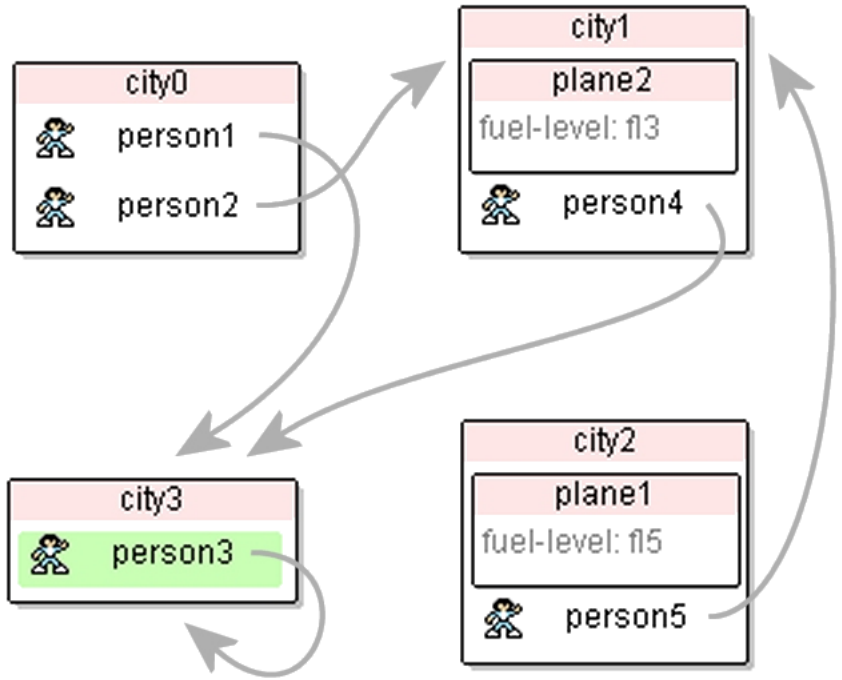
- Example Domain: ZenoTravel
  - Planes move people between cities (**board**, **debark**, **fly**)
  - Planes have limited fuel level; must **refuel**
  - Example instance:
    - 70 people
    - 35 planes
    - 30 cities



ZenoTravel Problem Instance

jonkv@ida

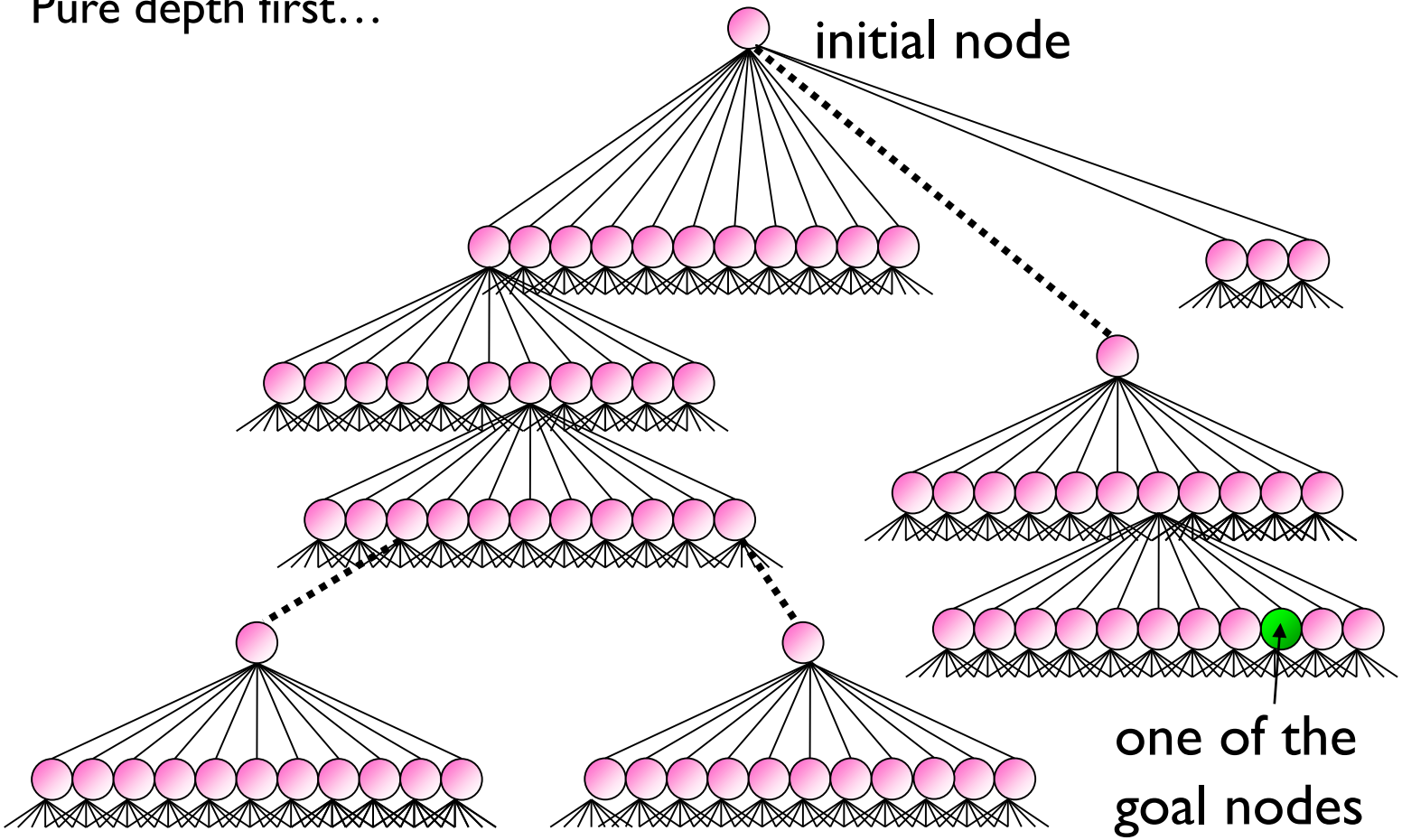
- A smaller problem instance



What Just Happened?

jonkv@ida

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



Control Rules

jonkv@ida

- **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

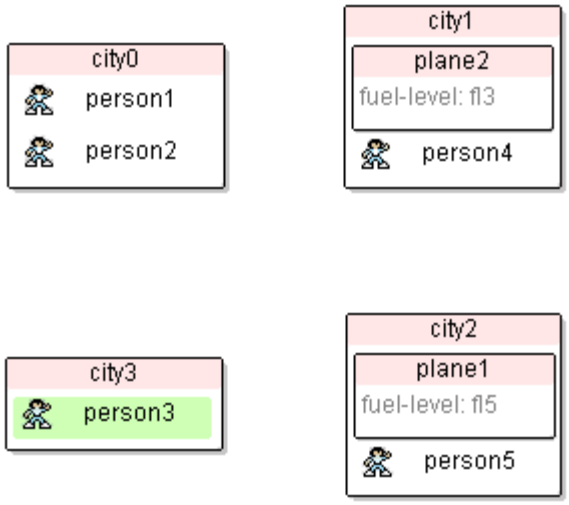
jonkv@ida

- **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"

$$\begin{aligned}
&\textbf{forall } t, \textit{person}, \textit{aircraft} \; [\\
&\quad ([t] \; !\textbf{in}(\textit{person}, \textit{aircraft}) \wedge \\
&\quad\; [t+1] \; \textbf{in}(\textit{person}, \textit{aircraft})) \rightarrow \\
&\quad \textbf{exists } \textit{city1}, \textit{city2} \; [\\
&\qquad [t] \; \textbf{at}(\textit{person}, \textit{city1}) \wedge \\
&\qquad \textbf{goal}(\textbf{at}(\textit{person}, \textit{city2})) \wedge \\
&\qquad \textit{city1} \; != \textit{city2} \;] \;]
\end{aligned}$$

Zeno Travel, second attempt

jonkv@ida



What's Wrong This Time?

jonkv@ida

- 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

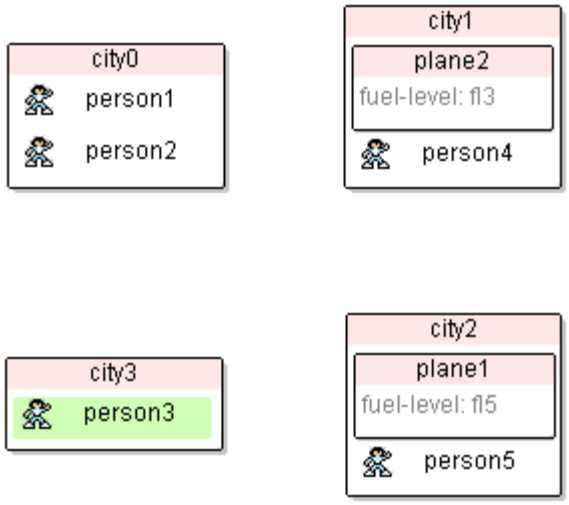
Control Rules

jonkv@ida

- #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

jonkv@ida



Conclusion

Conclusion

jonkv@ida

- Control Rules or Hierarchical Task Networks?
  - Both can be very efficient and expressive
  - If you have ”**recipes**” for everything, HTN can be more convenient
    - **Can** be modeled with control rules, but not intended for this purpose
    - You have to forbid everything that is ”outside” the recipe
  - If you have knowledge about ”**some things that shouldn’t be done**”:
    - With control rules, the default is to ”try everything”
    - Can more easily express localized knowledge about what should and shouldn’t be done
    - Doesn’t require knowledge of all the ways in which the goal can be reached