Course "Automated Planning: Theory and Practice"
Chapter 16: Planning with Control Formulas

Teacher: Marco Roveri - marco.roveri@unitn.it

M.S. Course: Artificial Intelligence Systems (LM)
A.A.: 2023-2024
Where: DISI, University of Trento
URL: https://bit.ly/3zOkGk8

Last updated: Wednesday 15th November, 2023

Terms of Use and Copyright

The material (text, figures) in these slides is authored by Jonas Kvarnström and Marco Roveri.

Two Kinds of Search Guidance

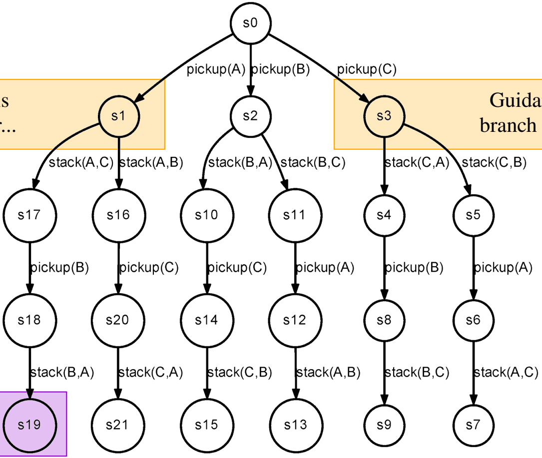
Prioritization: Which part of a search tree should be visited first?
Could use heuristic functions, could use other methods...

Guidance says: This branch *seems* better...

Guidance says: This branch *seems* worse...

Go this way first

Keep this, maybe return later
⟹ high memory usage



Goal: B on C on A

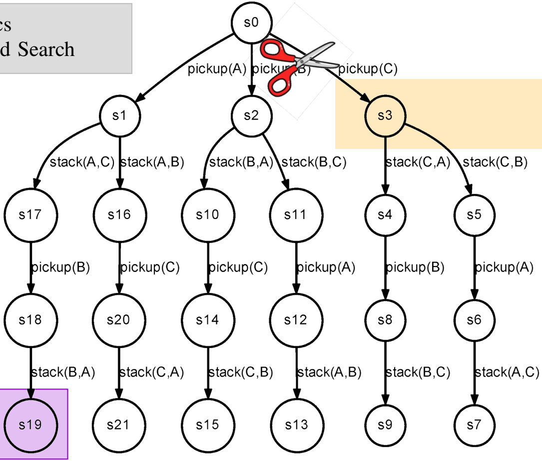
Prioritization

- 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

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...

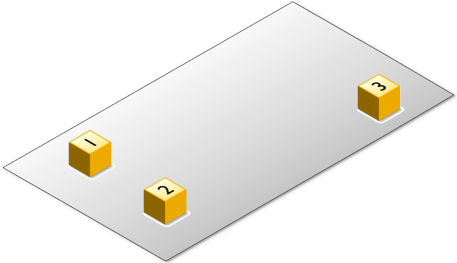
We found a
solution of cost 4

Pruning

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

Example: Emergency Service Logistics

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



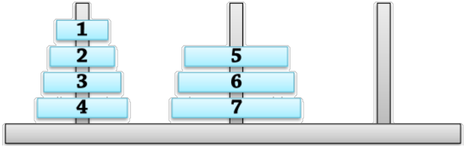
  - Picking up again is physically possible
  - It "destroys" `(at crate1 loc1)`, which is a goal – obviously stupid!

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

Example: Tower of Hanoi

Should we always prevent the destruction of achieved goals?

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

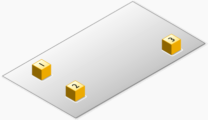


- Moving disk 1 to the third peg is possible but "destroys" a goal fact: `(on 1 2)`
  - Is this also obviously stupid?
  - No, it is necessary! Disk 1 is blocking us from moving disk 4, ...

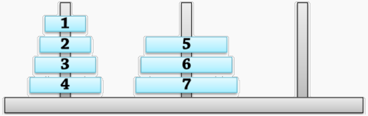
Heuristics

⟹ Heuristics may or may not detect:

Picking up crate 1 is bad



Moving disk 1 is good



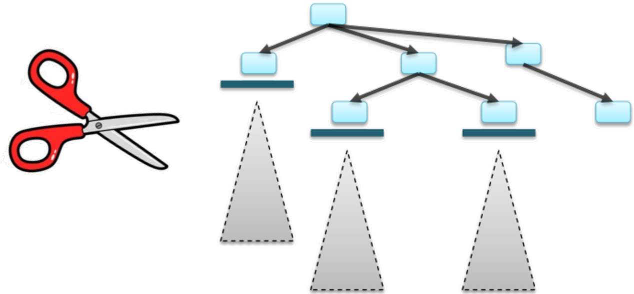
Will generally depend on the entire state
+ which alternative states exist,
not just the fact that you "destroy goal achievement"

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

Pruning

- 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

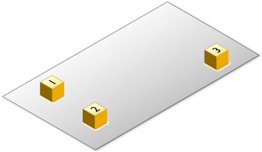


Control Formulas

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

Precondition Control

Simplest control information: Precondition Control



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

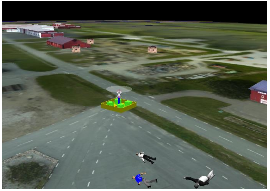
  How to express this, given that the goal requires `(at ?crate ?location)`?
- Alternative 1: New predicate `"(destination ?crate ?location)"`
  - Duplicates the information already specified in the goal
  - **precond**: ¬ `(destination ?crate ?location))`
- Alternative 2: New language extension `"goal(`φ`)"`
  - Evaluated in the set of goal states, not in the current state
  - **precond**: ¬ `(goal (at ?crate ?location))`

Supported by all planners

Requires extensions, but more convenient

STATE CONSTRAINTS

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



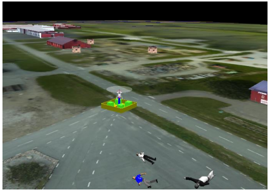
(at depot)
(= (fuel-level) 120)

(at pos1)
(= (fuel-level) 105)

(at pos2)
(= (fuel-level) 112)

Reaching nearest refueling point requires 108 fuel

State Constraints (cont.)



- 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?

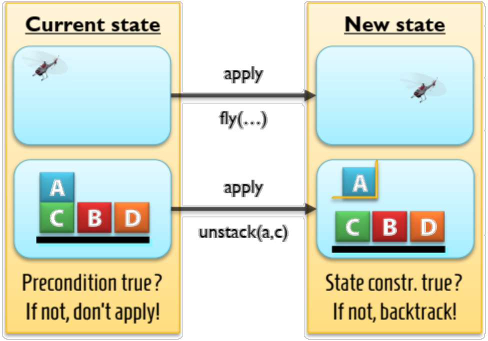
- Defined once, applied to every generated state

```
∀?u : uav [
  ∃?rp : refueling-point[
    (< (* (dist ?u ?rp)(fuel-usage ?u))
          (fuel-avail ?u)) ]
]
```

- Comparatively simple extension!

Testing State Constraints

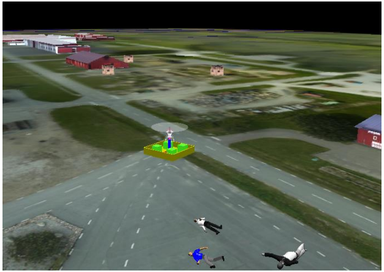
- Testing such state constraints is simple
  - Apply an action $\Longrightarrow$ new state is generated
    - Formula false in that state $\Longrightarrow$ Prune!
  - Similar to preconditions
    - But tested in the state after an action is applied, not before!



TEMPORAL CONDITIONS

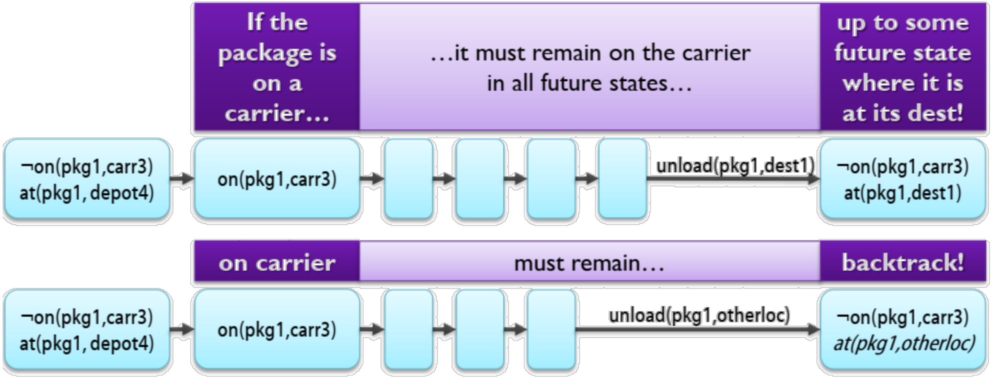
- A package on a carrier should remain there until it reaches its destination
  - For any plan π where we move it prematurely,
    there is a more efficient plan π' where we don't

How to express this as a single formula?



Temporal Conditions (cont.)

- "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 (LTL) Pnueli [6] (as in `TLPlan` Bacchus and Kabanza [1])
  - All formulas evaluated relative to a state sequence and a current state
    - Assuming that f is an LTL formula:
      - $\mathbf{X}\,f$ — `(next f)` — f is true in the next state
      - $\mathbf{F}\,f$ — `(eventually f)` — f is true either now or in some future state
      - $\mathbf{G}\,f$ — `(globally f)` — f is true now and in all future states
      - $f_1\,\mathbf{U}\,f_2$ — `(until f1 f2)` — (f_1 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 (cont.)

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

`(forall (?var) (type-predicate ?var) Φ):`

$$\forall v.type - predicate(v) \rightarrow \Phi$$

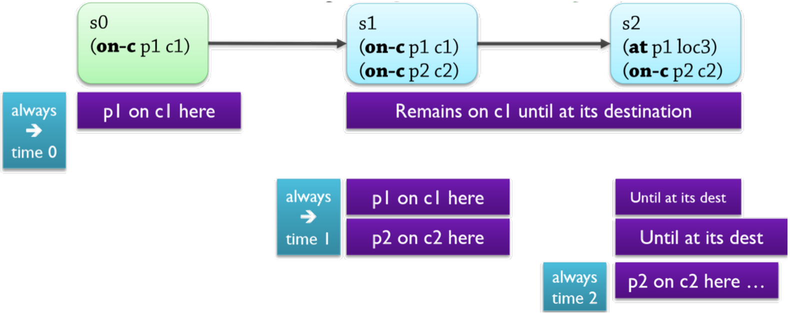
For all values of `?var` that satisfy `type-predicate`, ϕ must be true

```
(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                              ;;; ... then it remains on the carrier
                (on-carrier ?p ?c)                ;;; until there exists a location
                (exists (?loc) (at ?p ?loc)   ;;; where it is, and the goal
                  (goal (at ?p ?loc)))))))))  ;;; says it should be
```

- Should be true starting in the initial state

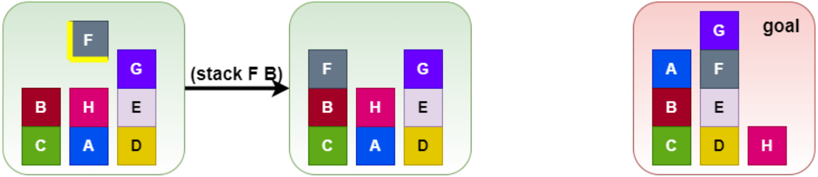
Control Formula (cont.)

```
(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                              ;;; ... then it remains on the carrier
                (on-carrier ?p ?c)                ;;; until there exists a location
                (exists (?loc) (at ?p ?loc)       ;;; where it is, and the goal
                  (goal (at ?p ?loc))))))))       ;;; says it should be
```



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 (cont.)

- 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 block)
        (next (or               ;; ... then in the next state, either
                 (clear ?x)           ;; ?x remain clear (didn’t extend the tower)
                 (exists (?y)         ;; or there is a block ?y
                    (on ?y ?x)        ;;   which is on ?x
                    (goodtower ?y);;   which is a goodtower
                 )))))
    ```

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4$

`(goodtower x)?` $\Longrightarrow$ `(clear x) or (goodtower y)`

What about the rest?
    ```

Blocks World (cont.)

- Rule 1: second attempt
  - ```
    (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 block)
        (next (or              ;; ... then in the next state, either
                  (clear ?x)         ;; ?x remain clear (didn't extend the tower)
                  (exists (?y)       ;; or there is a block ?y
                     (on ?y ?x)      ;;   which is on ?x
                     (goodtower ?y);;   which is a goodtower
                  )))))
    ```

$s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow s_4$

(goodtower x)? $\Longrightarrow$ (clear x) or (goodtower y)

(goodtower x)? $\Longrightarrow$ (clear x) or (goodtower y)

(goodtower x)? $\Longrightarrow$ (clear x) or (goodtower y)
    ```

Supporting Predicates

- Some planners allow us to define a predicate recursively
  - `(goodtowerbelow x)` means we will not have to move x
    - $$\begin{aligned} & goodtowerbelow(x) \leftrightarrow \\ & \quad [ontable(x) \wedge \neg\exists[y : GOAL(on(x,y))]] \\ & \vee \\ & \exists[y : on(x,y)]\{ \\ & \qquad \neg GOAL(ontable(x)) \wedge \\ & \qquad \neg GOAL(holding(y)) \wedge \\ & \qquad \neg GOAL(clear(y)) \wedge \\ & \qquad \forall[z : GOAL(on(x,z))](z = y) \wedge \\ & \qquad \forall[z : GOAL(on(z,y))](z = x) \wedge \\ & \qquad goodtowerbelow(y)\} \end{aligned}$$

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

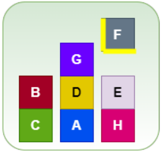
x is on something else

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

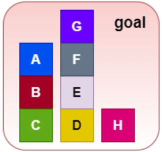
If x should be on z, then it is (z is y)

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

The remainder of the tower is also good

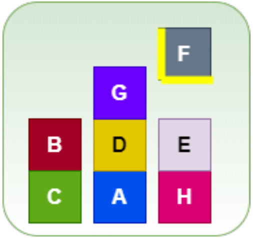


goodtowerbelow: B, C, H

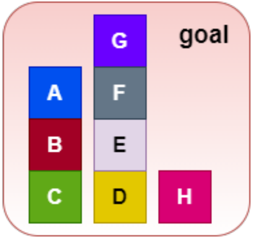


Supporting Predicates (cont.)

- `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
  - $badtower(x) \leftrightarrow clear(x) \wedge \neg goodtower(x)$

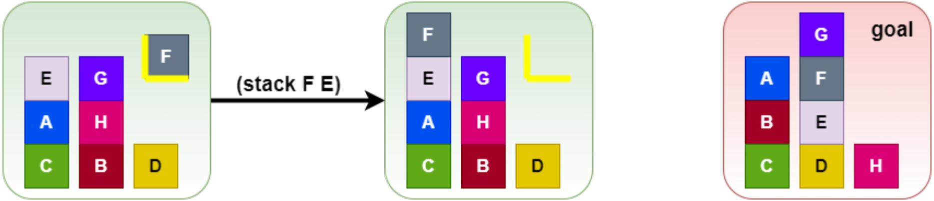


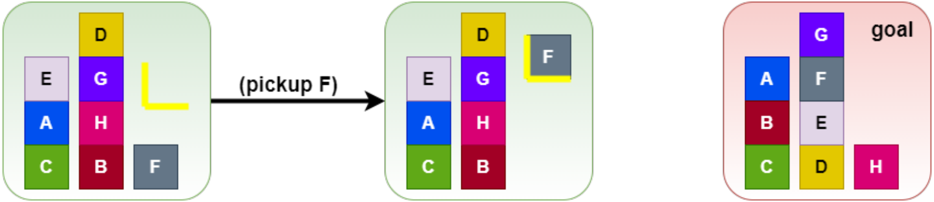
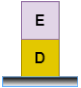
goodtower: B
goodtowerbelow: B, C, H
badtower: G, F
(neither: D, A)



Blocks World

- Step 2: Is this stupid?



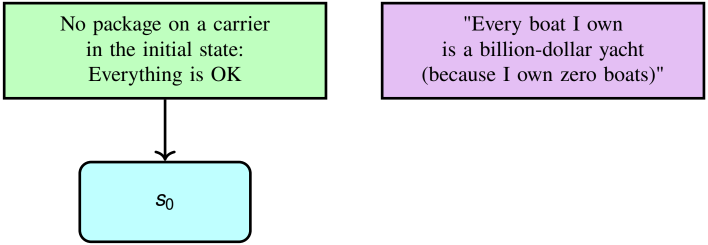
- Placing F on top of E is stupid, because we have to move E later...
  - Would have been better to put F on the table!
  - But E was not a goodtower, so the previous rule didn't detect the problem
- Never put anything on a badtower!
  - ```
    (always (forall (?x)
               (clear ?x)        ;;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

- 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 on 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

- 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

```
(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                                  ;;; ... then it remains on the carrier
                (on-carrier ?p ?c)                    ;;; until there exists a location
                (exists (?loc) (at ?p ?loc))          ;;; where it is, and the goal
                  (goal (at ?p ?loc))))))))           ;;; says it should be
```

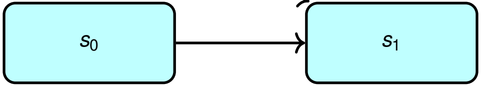


Evaluation (cont.)

```
(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                                ;;; ... then it remains on the carrier
                (on-carrier ?p ?c)                  ;;; until there exists a location
                (exists (?loc) (at ?p ?loc)         ;;; where it is, and the goal
                  (goal (at ?p ?loc))))))))         ;;; 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!



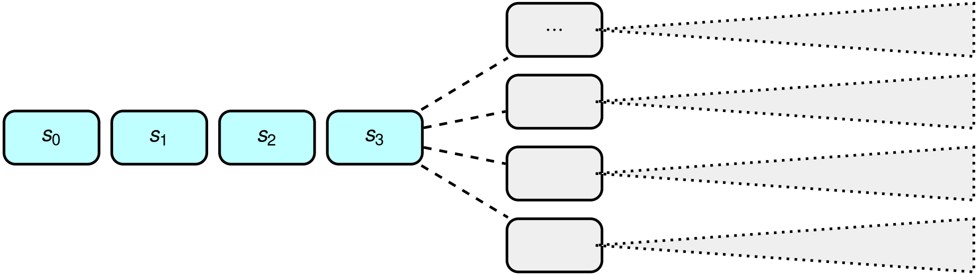
The formula is violated, but only because the solution is not complete yet! We must be allowed to continue, generating new states...

Evaluation: 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



We have applied some actions, yielding a sequence of states

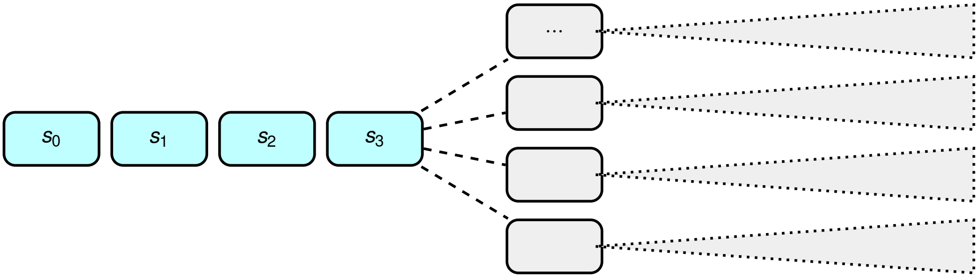
We intend to generate additional actions and states, but right now we don't know which ones

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 (cont.)



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

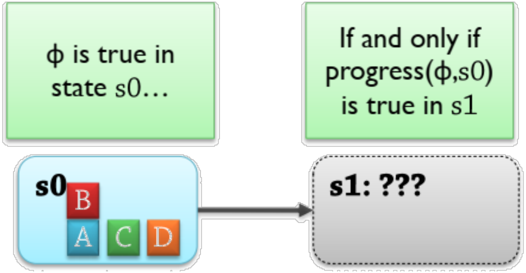
Evaluate those parts of the
formula that refer to known states

Leave other parts of the formula
to be evaluated later

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

Progressing Temporal Formulas

- 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



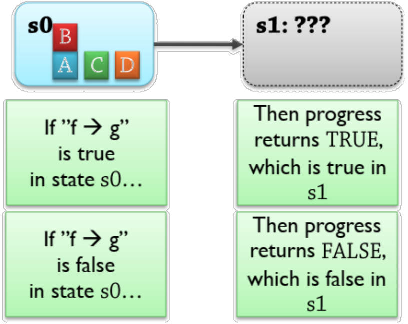
Progressing Temporal Formulas (cont.)

- 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.

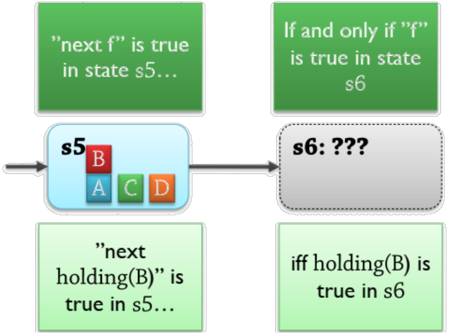
Progressing Temporal Formulas (cont.)

- Base case: Formulas without temporal operators ("`(on A B)` $\rightarrow$ `(on C D)`")
  - Must be true here, in this state
  - $\mathit{progress}(\Phi, s) = \mathit{TRUE}$ if Φ holds in s (we already know how to test this)
  - $\mathit{progress}(\Phi, s) = \mathit{FALSE}$ otherwise



Progressing Temporal Formulas (cont.)

- 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!



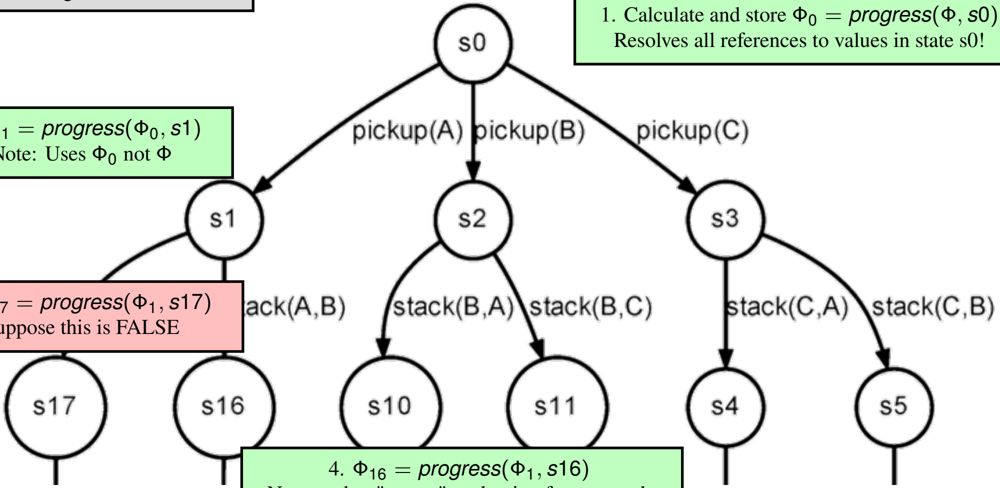
Additional cases are discussed in the book (**always**, **eventually**, **until**, ...)

PROGRESSION IN DEPTH FIRST SEARCH

Φ = original control formula

1. Calculate and store $\Phi_0 = \mathit{progress}(\Phi, s0)$
Resolves all references to values in state s0!

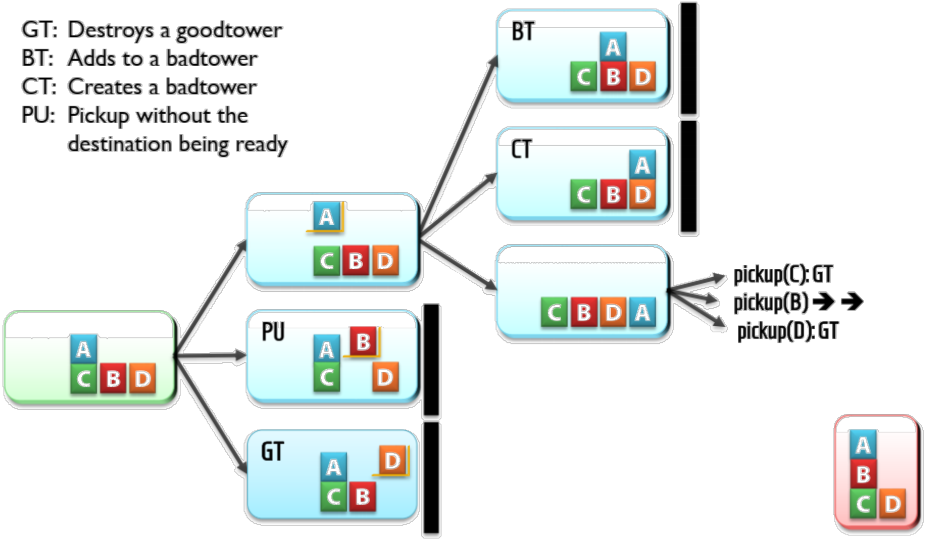
2. $\Phi_1 = \mathit{progress}(\Phi_0, s1)$
Note: Uses Φ_0 not Φ



3. $\Phi_{17} = \mathit{progress}(\Phi_1, s17)$
Suppose this is FALSE

4. $\Phi_{16} = \mathit{progress}(\Phi_1, s16)$
Non need to "restart" evalaution from scratch

Depth First Search with Pruning



Performance

- International Planning Competition
  - 2000 `TALplanner` by Doherty and Kvarnström [2] received the top award for a "hand-tailored" (i.e., domain-configurable) planner
- 2002 International Planning Competition
  - `TLplan` by Bacchus and Kabanza [1] 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

Conclusions

- 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

Helpful Actions: Running Example

- Seen when discussing Relaxed Plan Graph Heuristic
- Prepare and serve a surprise dinner,
  take out the garbage,
  and make sure the present is wrapped before waking your sweetheart!
  - $s_0 = \{\texttt{clean}, \texttt{garbage}, \texttt{asleep}\}$
  - $g = \{\texttt{clean}, \neg\texttt{garbage}, \texttt{served}, \texttt{wrapped}\}$
  -

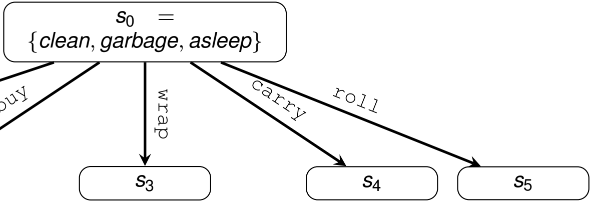
Action	**Preconditions**	**Effects**
(cook)	clean	dinner
(serve)	dinner	served
(wrap)	asleep	wrapped
(carry)	garbage	¬ garbage, ¬ clean
(roll)	garbage	¬ garbage, ¬ asleep
(clean)	¬ clean	clean
(buy)	–	dinner

Helpful Actions: Running Example (cont.)

- Let's do heuristic forward state space search with h_{FF} ...
  - First step: compute $h_{FF}(s_0)$ where

    $s_0 = \{clean, garbage, asleep\}$
  - Does not satisfy the goal
  - Let's create successors
    - 5 applicable actions
    - 5 successor states

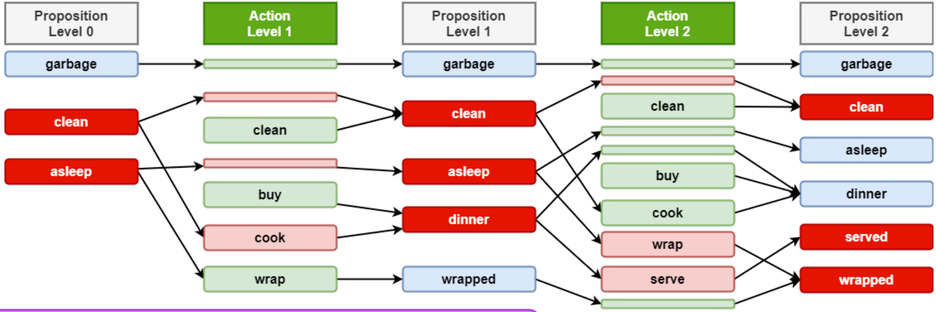
Action	**Prec.**	**Effects**
(cook)	clean	dinner
(serve)	dinner	served
(wrap)	asleep	wrapped
(carry)	garbage	¬ garbage, ¬ clean
(roll)	garbage	¬ garbage, ¬ asleep
(clean)	¬ clean	clean
(buy)	–	dinner



Are all successors equally likely to be "useful"?

Helpful Actions: Running Example (cont.)

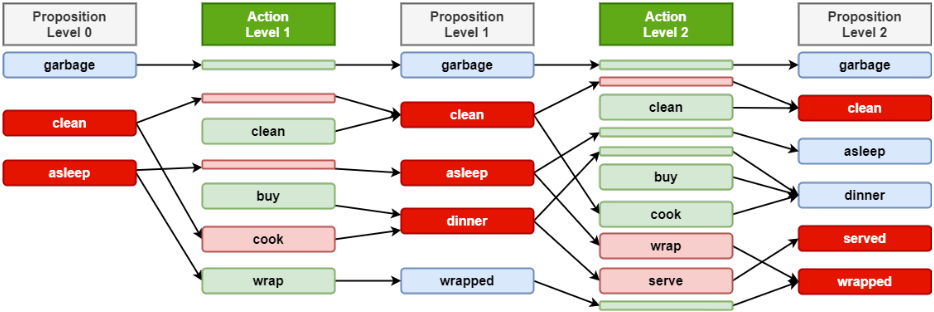
- What did we do when we computed $h_{FF}(s_0)$?
  - Construct a relaxed planning graph starting in s_0
  - Extract a relaxed plan (sufficient for achieving the goal in the relaxed problem)
  - $h_{FF}(s_0)$ = the cost of the relaxed plan



Example: There are *other* plans that could be generated

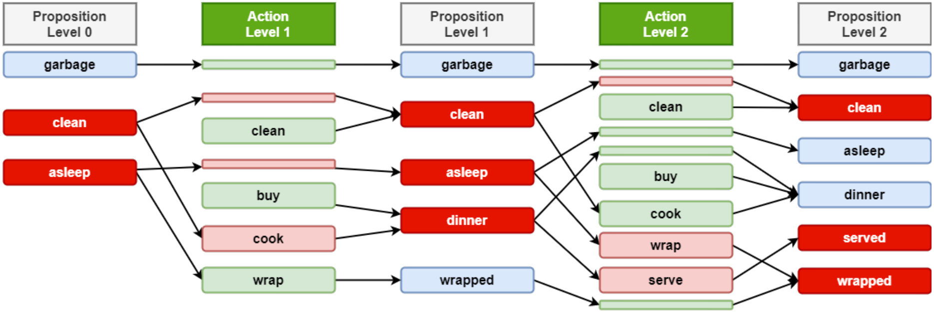
Helpful Actions: Running Example (cont.)

- Consider the actions selected at action level 1
  - Might be more likely to be useful as first actions ...
    - Were useful in the relaxed problem, where you found a complete relaxed plan!
  - $\Longrightarrow$ First action `cook`?
    - No, too restrictive!



Helpful Actions: Running Example (cont.)

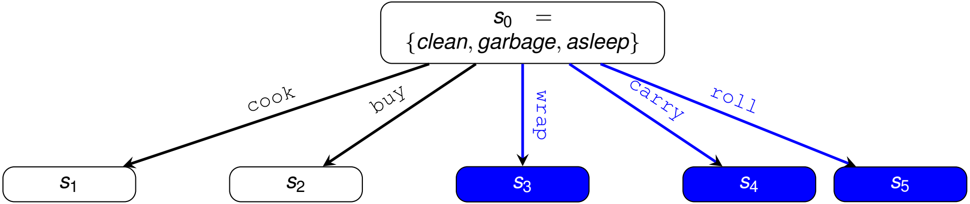
- Consider the actions selected at action level 1
  - Then consider all alternative actions that could achieve the same facts
  - $H(s) = \{a | pre(a) \subseteq s \wedge eff^{+}(a) \cap PropLevel_1 \neq \emptyset\}$ = {`cook`, `buy`}
  - Called helpful actions or preferred operators



Helpful Actions: Running Example (cont.)

- New beginning of the search tree:
  - 2 applicable helpful actions
  - 2 successor states

Action	Prec.	Effects
(cook)	clean	dinner
(serve)	dinner	served
(wrap)	asleep	wrapped
(carry)	garbage	¬ garbage, ¬ clean
(roll)	garbage	¬ garbage, ¬ asleep
(clean)	¬ clean	clean
(buy)	–	dinner



Generated...

But not guaranteed to be useful in practice

Not generated...

Though not guaranteed to be useless...

Heuristic Search with Helpful Action Pruning

```
function SEARCH(problem)
    initial-node ← MAKE-INITIAL-NODE(problem)
    open ← {initial-node}
    while (open ≠ ∅) do
        node ← SEARCH-STRATEGY-REMOVE-FROM(open)
        if IS-SOLUTION(node) then
            return EXTRACT-PLAN-FROM(node)
        end if
        for each newnode ∈ SUCCESSORS(node) do
            open ← open ∪ {newnode}
        end for
    end while
    return Failure
end function
```

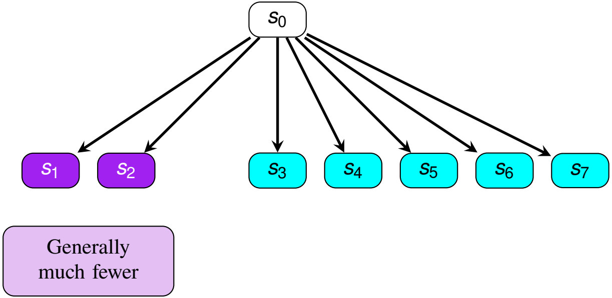
- SUCCESSORS(node) is tweaked to only generate successors using actions in $H(node)$!
- Incomplete if there are dead ends!
  - Actions not in $H(node)$ may be required;
    - not detected due to relaxation...
- If the search fails (e.g. via EHC) fall back on best first search using e.g. $h(node) = h_{FF}(node)$ or any other heuristic!

Helpful Actions and Completeness

- Using helpful actions for pruning leads to incompleteness
  - May search for a long time,
    exhaust the search space,
    then start over using complete search
- "Helpful actions" are more likely to be helpful
  - But skipping the other actions completely is too strict!

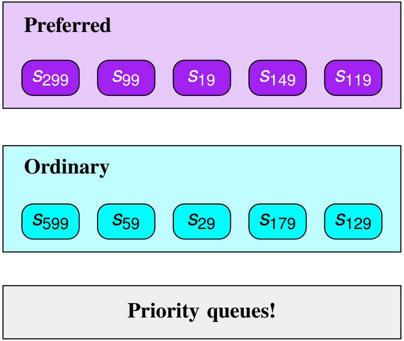
Pruning vs Prioritization

- Many state of the art planners (e.g. Fast Downward) prioritize helpful actions
  - Successors created by helpful actions in $H(s)$ are preferred successors
  - Successors created by other actions are ordinary successors



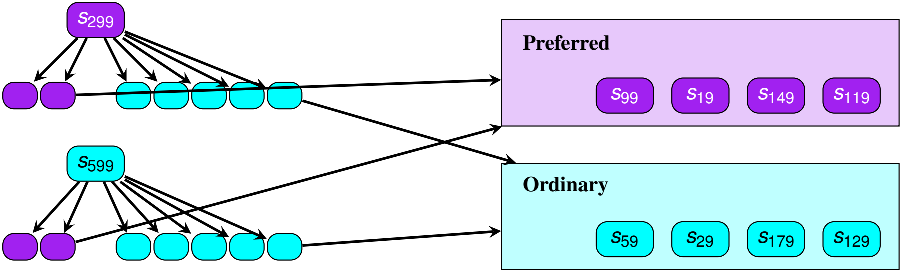
Dual Queues

- Use dual queues (two "open lists")
  - One for states generated as preferred successors
  - One for the ordinary states



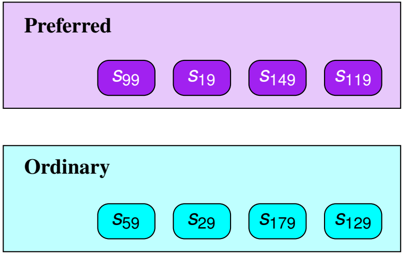
Dual Queues (cont.)

- To expand a state
  - Pick the best state from the preferred queue, and expand it
  - Pick the best state from the ordinary queue, and expand it
  - Place all new states where they belong



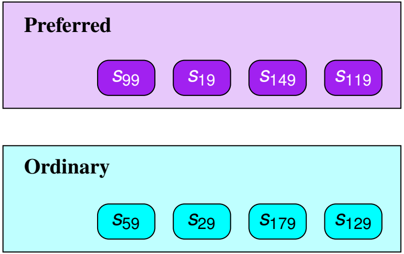
Dual Queues (cont.)

- Fewer states are preferred
  - Reached more quickly in the queue
- If we "misclassified" an action as non-helpful:
  - Don't have to exhaust the "preferred part" of the search space before we can "recover"
  - Search is complete



Boosted Dual Queues

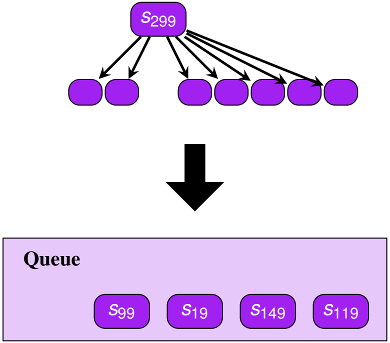
- Whenever progress is made (better h-value reached):
  - Choose e.g. 1000 times from the preferred queue
  - (Each chosen state is expanded as usual, *modifying* both queues... Then you pick again)
- If progress is made again within these e.g. 1000 successors:
  - Add another e.g. 1000, accumulating
  - (Progress made after e.g. 300 $\Longrightarrow$ keep expanding e.g. 1700 more)
- After reaching the preferred successor limit:
  - Expand a single node from the non-preferred queue
- Still complete
  - More aggressive than ordinary dual queues
  - Less aggressive than pure pruning



Deffered Evaluation

Typically takes most of the time

- Standard best-first search:
  - Remove the "best" (most promising) state from the open list / priority queue
  - Check whether it satisfies the goal
  - Generate all successors
  - Calculate their heuristic values
  - Place in priority queue(s)
- Deferred Evaluation (Fast Downward, ...)
  - Remove the "best" state from the priority queue
  - Check whether it satisfies the goal
  - Calculate its heuristic value (only one!)
  - Generate all successors
  - Place in priority queue using the parent's heuristic value
- Takes less time, but less accurate heuristic - "one step behind"
  - Often faster but lower-quality plans



References I

[1] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116 (1-2):123–191, 2000. doi: 10.1016/S0004-3702(99)00071-5. URL `https://doi.org/10.1016/S0004-3702(99)00071-5`. 18, 40

[2] Patrick Doherty and Jonas Kvarnström. Talplanner: A temporal logic-based planner. *AI Mag.*, 22(3):95–102, 2001. doi: 10.1609/aimag.v22i3.1581. URL `https://doi.org/10.1609/aimag.v22i3.1581`. 40

[3] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN 9781608459698. doi: 10.2200/S00513ED1V01Y201306AIM022. URL `https://doi.org/10.2200/S00513ED1V01Y201306AIM022`.

[4] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

[5] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 978-1-107-03727-4. URL `http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB`.

[6] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi: 10.1109/SFCS.1977.32. URL `https://doi.org/10.1109/SFCS.1977.32`. 18