Lecture slides for

Automated Planning: Theory and Practice

Chapter 11
Hierarchical Task Network Planning

Dana S. Nau

University of Maryland

2:26 PM April 18, 2012

Motivation

- We may already have an idea how to go about solving problems in a planning domain
- Example: travel to a destination that's far away:
  - Domain-independent planner:
    - » many combinations of vehicles and routes
  - Experienced human: small number of "recipes"

    e.g., flying:

    1. buy ticket from local airport to remote airport
    2. travel to local airport
    3. fly to remote airport
    4. travel to final destination

- How to enable planning systems to make use of such recipes?

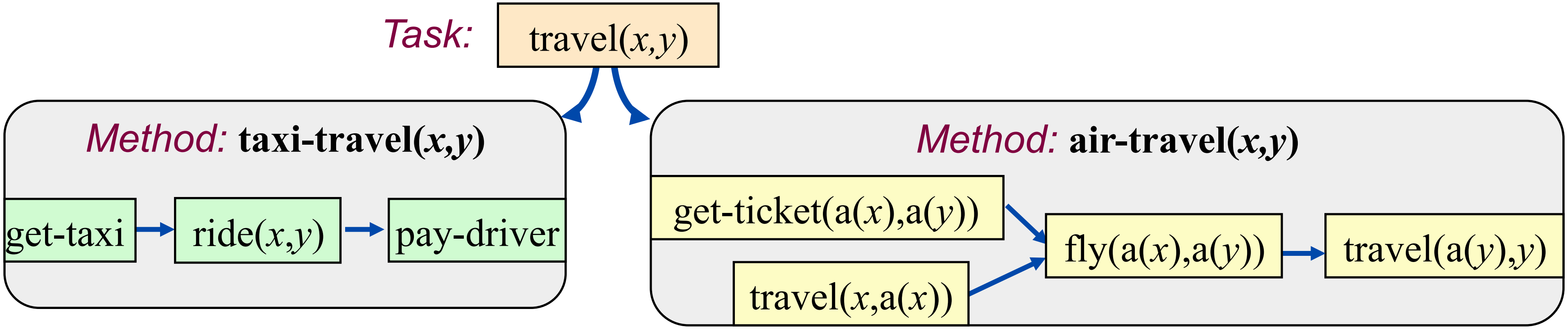
Two Approaches

- Control rules (previous chapter):
  - Write rules to prune every action that *doesn't* fit the recipe

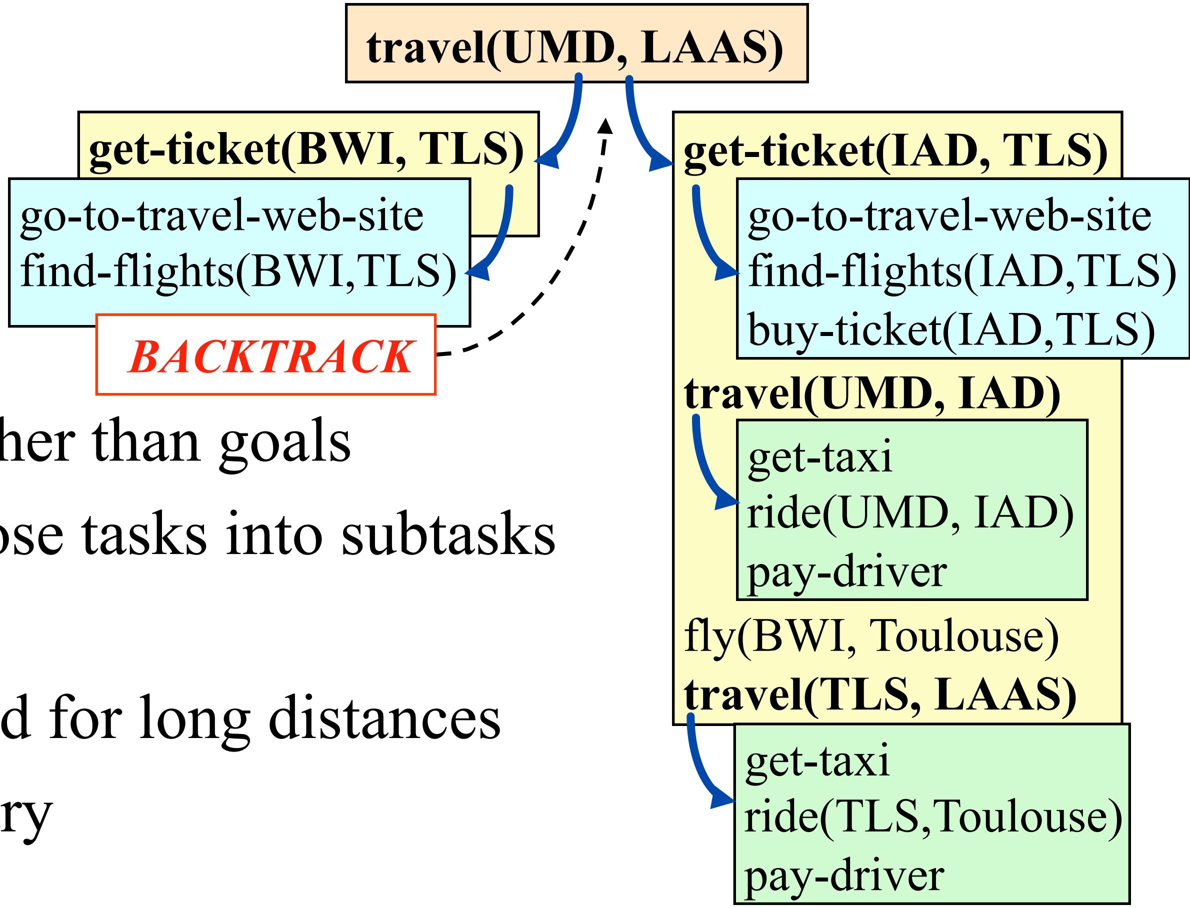
```
Abstract-search(u)
    if Terminal(u) then return(u)
    u ← Refine(u)            ;;  refinement step
    B ← Branch(u)            ;;  branching step
    B′ ← Prune(B)            ;;  pruning step
    if B′ = ∅ then return(failure)
    nondeterministically choose v ∈ B′
    return(Abstract-search(v))
end
```

- Hierarchical Task Network (HTN) planning:
  - Describe the actions and subtasks that *do* fit the recipe

```
Abstract-search(u)
    if Terminal(u) then return(u)
    u ← Refine(u)            ;;  refinement step
    B ← Branch(u)            ;;  branching step
    B′ ← Prune(B)            ;;  pruning step
    if B′ = ∅ then return(failure)
    nondeterministically choose v ∈ B′
    return(Abstract-search(v))
end
```



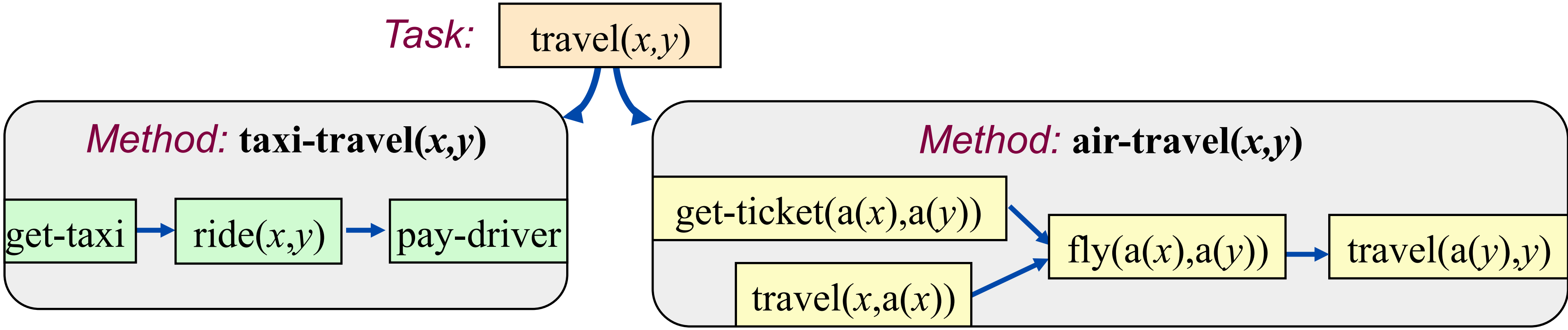
HTN Planning



- Problem reduction
  - *Tasks* (activities) rather than goals
  - *Methods* to decompose tasks into subtasks
  - Enforce constraints
    - » E.g., taxi not good for long distances
  - Backtrack if necessary

HTN Planning

- HTN planners may be domain-specific
  - e.g., see Chapters 20 (robotics) and 23 (bridge)
- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description that defines not only the operators, but also the methods
  - Problem description
    - » domain description, initial state, initial task network



Simple Task Network (STN) Planning

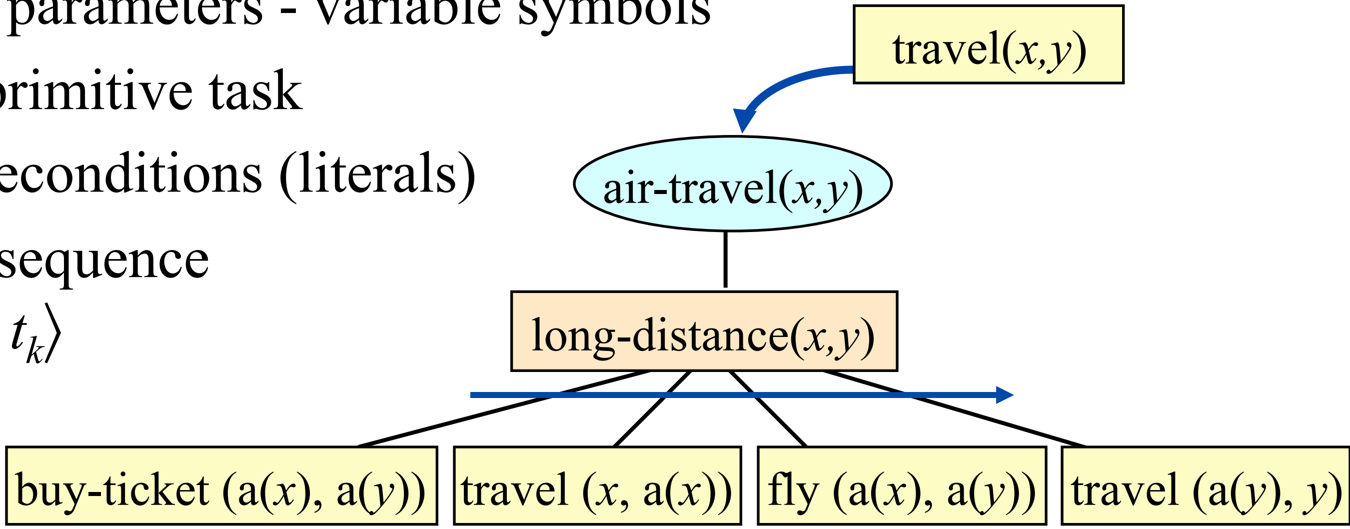
- A special case of HTN planning
- States and operators
  - The same as in classical planning
- *Task*: an expression of the form $t(u_1,\ldots,u_n)$
  - t is a *task symbol*, and each u_i is a term
  - Two kinds of task symbols (and tasks):
    - *primitive*: tasks that we know how to execute directly
      - task symbol is an operator name
    - *nonprimitive*: tasks that must be decomposed into subtasks
      - use *methods* (next slide)

Methods

- Totally ordered method: a 4-tuple

  m = (name(m), task(m), precond(m), subtasks(m))

  - name(m): an expression of the form $n(x_1,\ldots,x_n)$
    - » $x_1,\ldots,x_n$ are parameters - variable symbols
  - task(m): a nonprimitive task
  - precond(m): preconditions (literals)
  - subtasks(m): a sequence of tasks $\langle t_1, \ldots, t_k \rangle$



air-travel(x,y)

task: travel(x,y)

precond: long-distance(x,y)

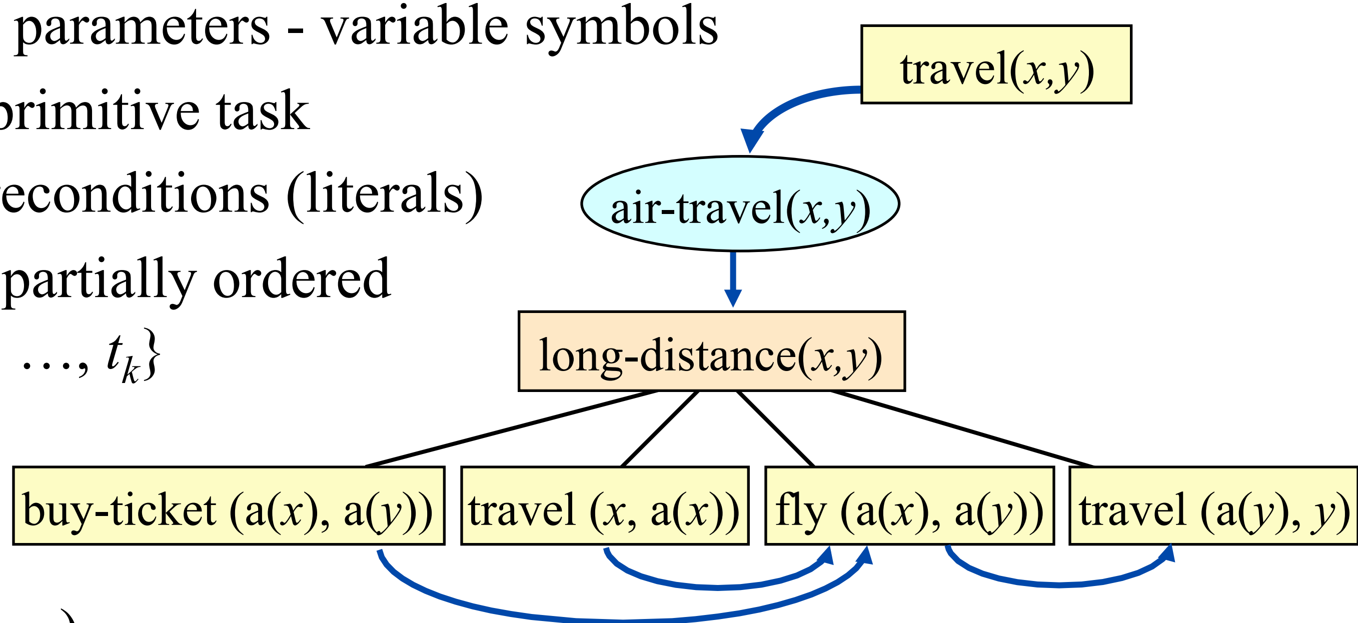
subtasks: $\langle$buy-ticket($a(x)$, $a(y)$), travel($x,a(x)$), fly($a(x)$, $a(y)$), travel($a(y),y$)$\rangle$

Methods (Continued)

- Partially ordered method: a 4-tuple

  m = (name(m), task(m), precond(m), subtasks(m))

  - name(m): an expression of the form $n(x_1,\ldots,x_n)$
    - » $x_1,\ldots,x_n$ are parameters - variable symbols
  - task(m): a nonprimitive task
  - precond(m): preconditions (literals)
  - subtasks(m): a partially ordered set of tasks $\{t_1, \ldots, t_k\}$



air-travel(x,y)

task: travel(x,y)

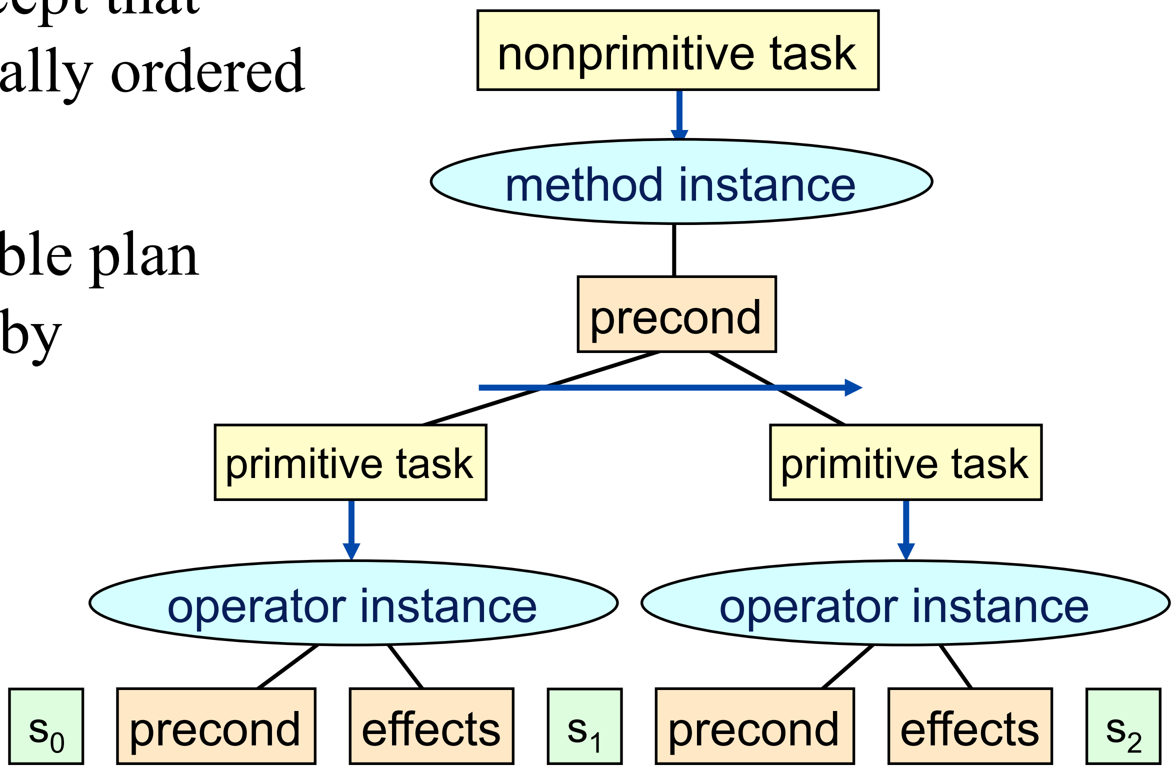
precond: long-distance(x,y)

network: u_1=buy-ticket($a(x),a(y)$), u_2= travel($x,a(x)$), u_3= fly($a(x)$, $a(y)$)

u_4= travel($a(y),y$), $\{(u_1,u_3), (u_2,u_3), (u_3,u_4)\}$

Domains, Problems, Solutions

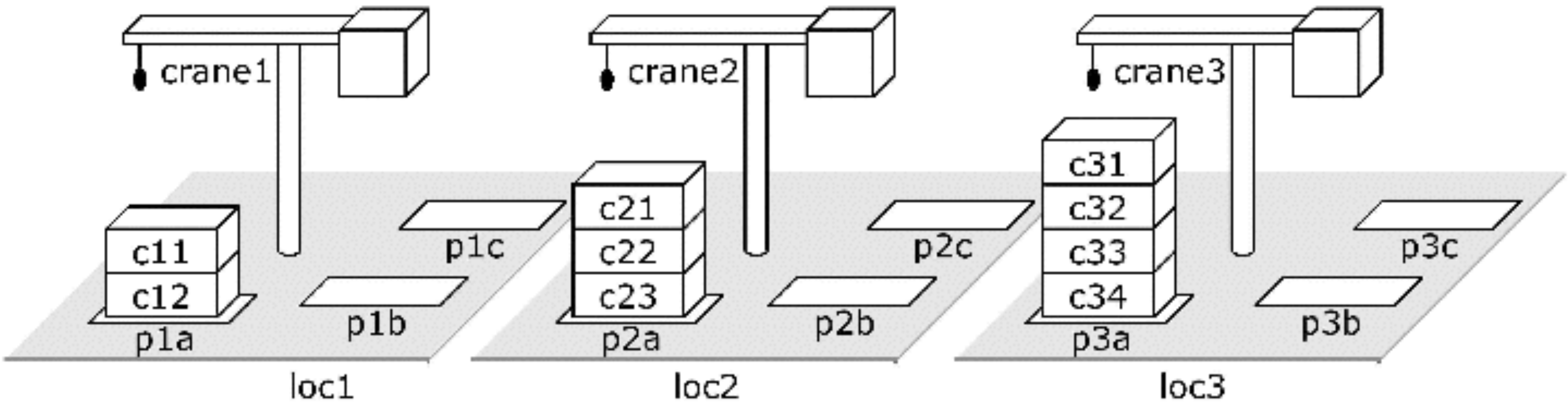
- STN planning domain: methods, operators
- STN planning problem: methods, operators, initial state, task list
- Total-order STN planning domain and planning problem:
  - Same as above except that all methods are totally ordered



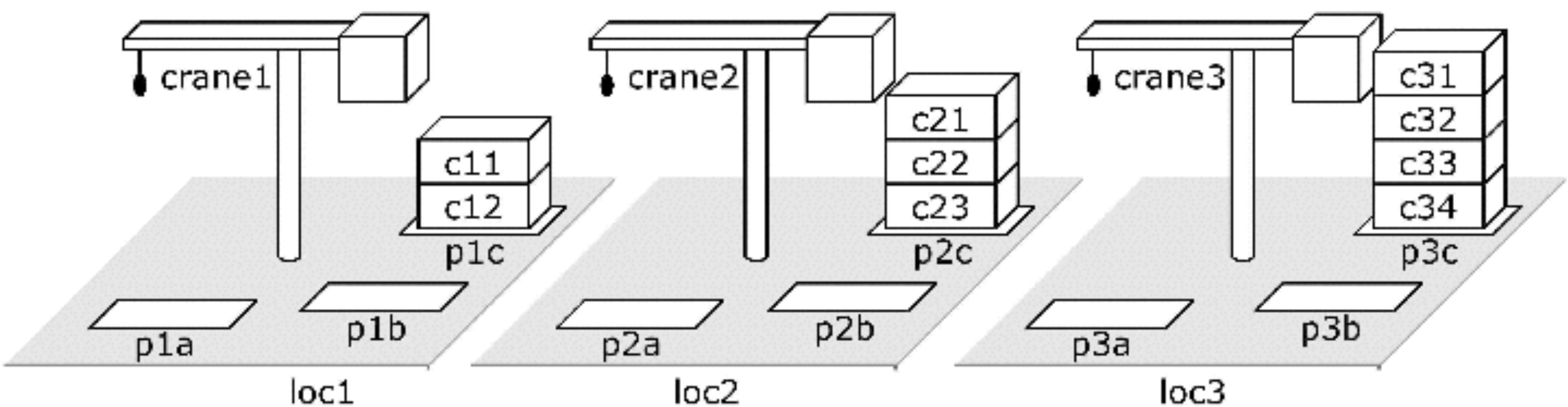
- Solution: any executable plan that can be generated by recursively applying
  - methods to nonprimitive tasks
  - operators to primitive tasks

Example

- Suppose we want to move three stacks of containers in a way that preserves the order of the containers



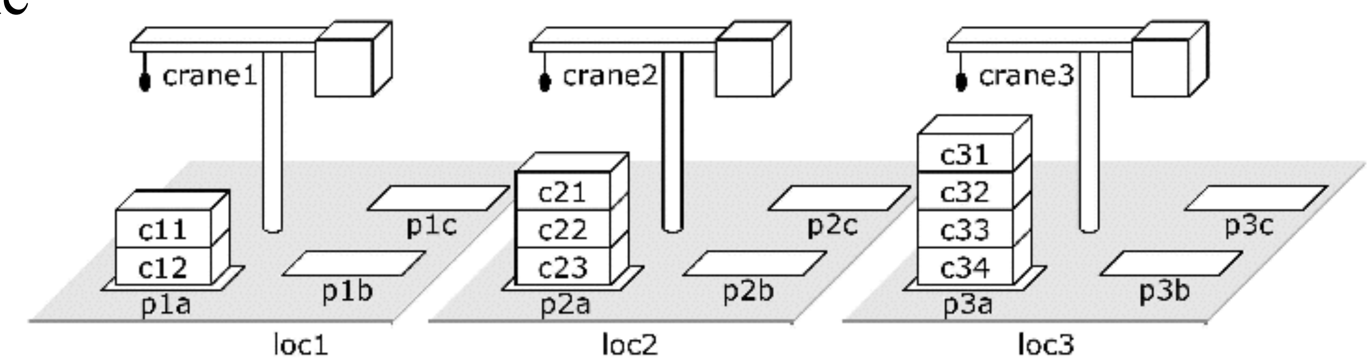
(a) initial state



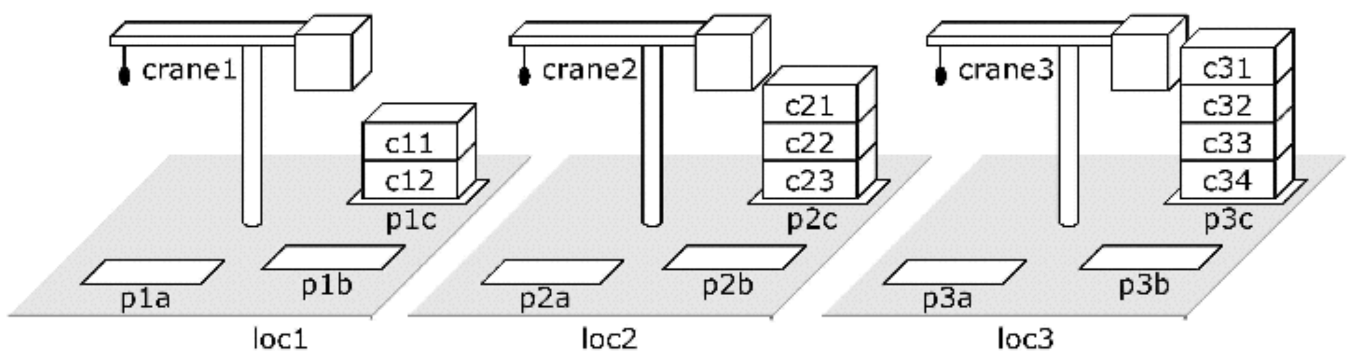
(b) goal

Example (continued)

- A way to move each stack:
  - first move the containers from p to an intermediate pile r
  - then move them from r to q



(a) initial state

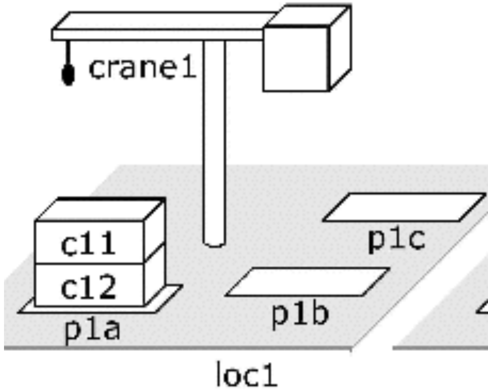


(b) goal

Total-Order Formulation

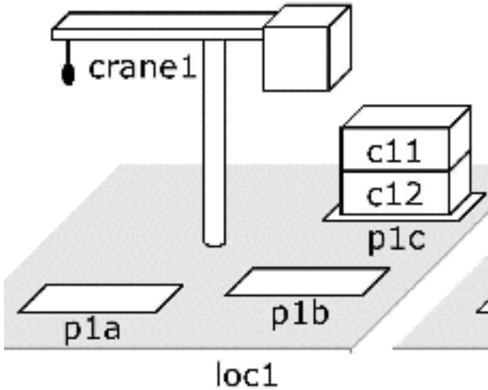
take-and-put($c, k, l_1, l_2, p_1, p_2, x_1, x_2$):
  task: move-topmost-container(p_1, p_2)
  precond: top(c, p_1), on(c, x_1), ; *true if p_1 is not empty*
  attached(p_1, l_1), belong(k, l_1), ; *bind l_1 and k*
  attached(p_2, l_2), top(x_2, p_2) ; *bind l_2 and x_2*
  subtasks: ⟨take(k, l_1, c, x_1, p_1), put(k, l_2, c, x_2, p_2)⟩

recursive-move(p, q, c, x):
  task: move-stack(p, q)
  precond: top(c, p), on(c, x) ; *true if p is not empty*
  subtasks: ⟨move-topmost-container(p, q), move-stack(p, q)⟩
  ;; *the second subtask recursively moves the rest of the stack*



do-nothing(p, q)
  task: move-stack(p, q)
  precond: top($pallet, p$) ; *true if p is empty*
  subtasks: ⟨⟩ ; *no subtasks, because we are done*

move-each-twice()
  task: move-all-stacks()
  precond: ; *no preconditions*
  subtasks: ; *move each stack twice:*
  ⟨move-stack(p1a,p1b), move-stack(p1b,p1c),
  move-stack(p2a,p2b), move-stack(p2b,p2c),
  move-stack(p3a,p3b), move-stack(p3b,p3c)⟩



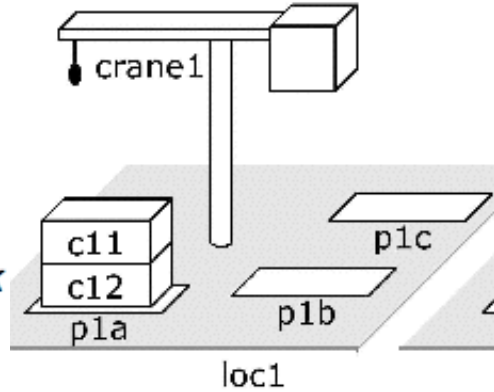
Partial-Order Formulation

take-and-put$(c, k, l_1, l_2, p_1, p_2, x_1, x_2)$:

- task: move-topmost-container(p_1, p_2)
- precond: top(c, p_1), on(c, x_1), *; true if p_1 is not empty*
  attached(p_1, l_1), belong(k, l_1), *; bind l_1 and k*
  attached(p_2, l_2), top(x_2, p_2) *; bind l_2 and x_2*
- subtasks: $\langle$take(k, l_1, c, x_1, p_1), put$(k, l_2, c, x_2, p_2)\rangle$

recursive-move(p, q, c, x):

- task: move-stack(p, q)
- precond: top(c, p), on(c, x) *; true if p is not empty*
- subtasks: $\langle$move-topmost-container(p, q), move-stack$(p, q)\rangle$
  ;; the second subtask recursively moves the rest of the stack

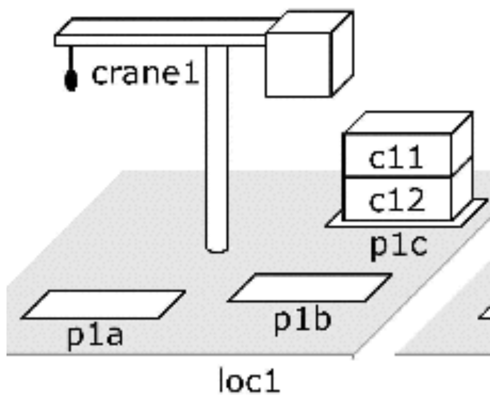


do-nothing(p, q)

- task: move-stack(p, q)
- precond: top$(pallet, p)$ *; true if p is empty*
- subtasks: $\langle\rangle$ *; no subtasks, because we are done*

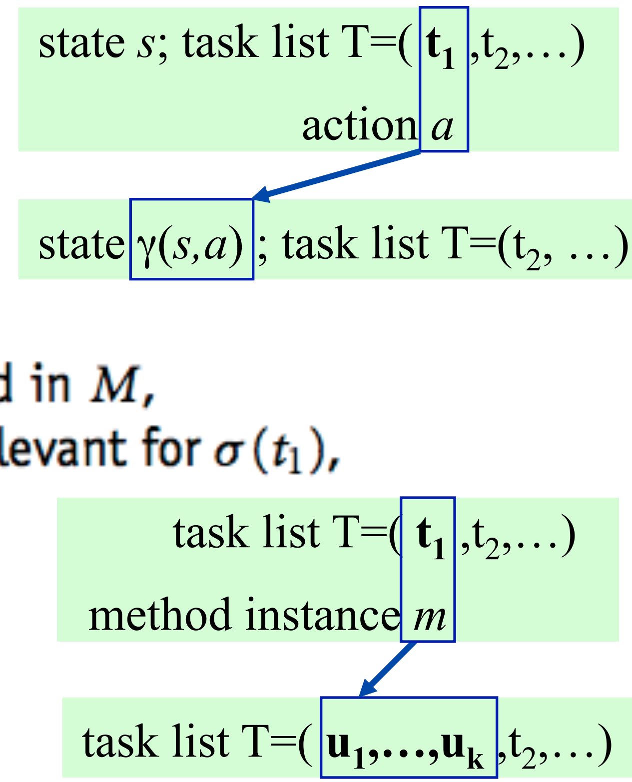
move-each-twice()

- task: move-all-stacks()
- precond: *; no preconditions*
- network: *; move each stack twice:*
  u_1 =move-stack(p1a,p1b), u_2 =move-stack(p1b,p1c),
  u_3 =move-stack(p2a,p2b), u_4 =move-stack(p2b,p2c),
  u_5 =move-stack(p3a,p3b), u_6 =move-stack(p3b,p3c),
  $\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$



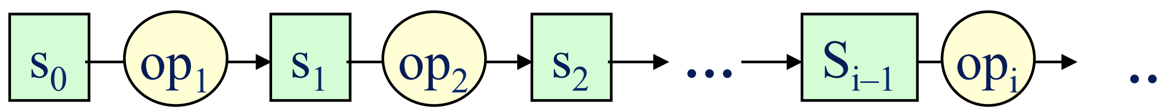
Solving Total-Order STN Planning Problems

TFD$(s, \langle t_1, \ldots, t_k \rangle, O, M)$
  if $k = 0$ then return $\langle\rangle$ (i.e., the empty plan)
  if t_1 is primitive then
    $active \leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in O,
      σ is a substitution such that a is relevant for $\sigma(t_1)$,
      and a is applicable to $s\}$
    if $active = \emptyset$ then return failure
    nondeterministically choose any $(a, \sigma) \in active$
    $\pi \leftarrow$ TFD$(\gamma(s, a), \sigma(\langle t_2, \ldots, t_k \rangle), O, M)$
    if π = failure then return failure
    else return $a.\pi$
  else if t_1 is nonprimitive then
    $active \leftarrow \{m \mid m$ is a ground instance of a method in M,
      σ is a substitution such that m is relevant for $\sigma(t_1)$,
      and m is applicable to $s\}$
    if $active = \emptyset$ then return failure
    nondeterministically choose any $(m, \sigma) \in active$
    $w \leftarrow$ subtasks$(m).\sigma(\langle t_2, \ldots, t_k \rangle)$
    return TFD(s, w, O, M)

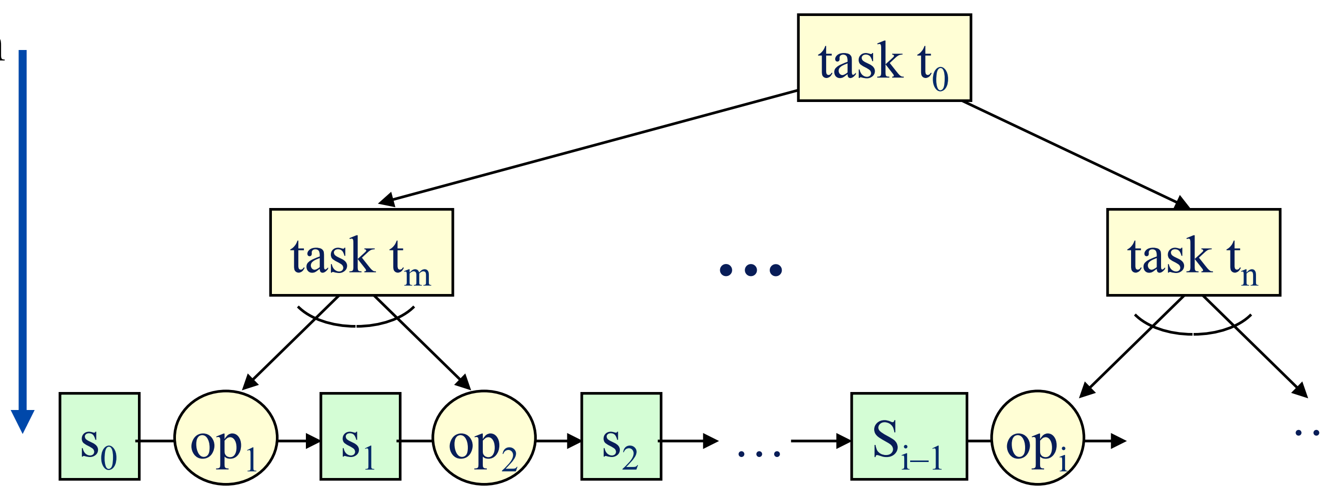


Comparison to Forward and Backward Search

- In state-space planning, must choose whether to search forward or backward



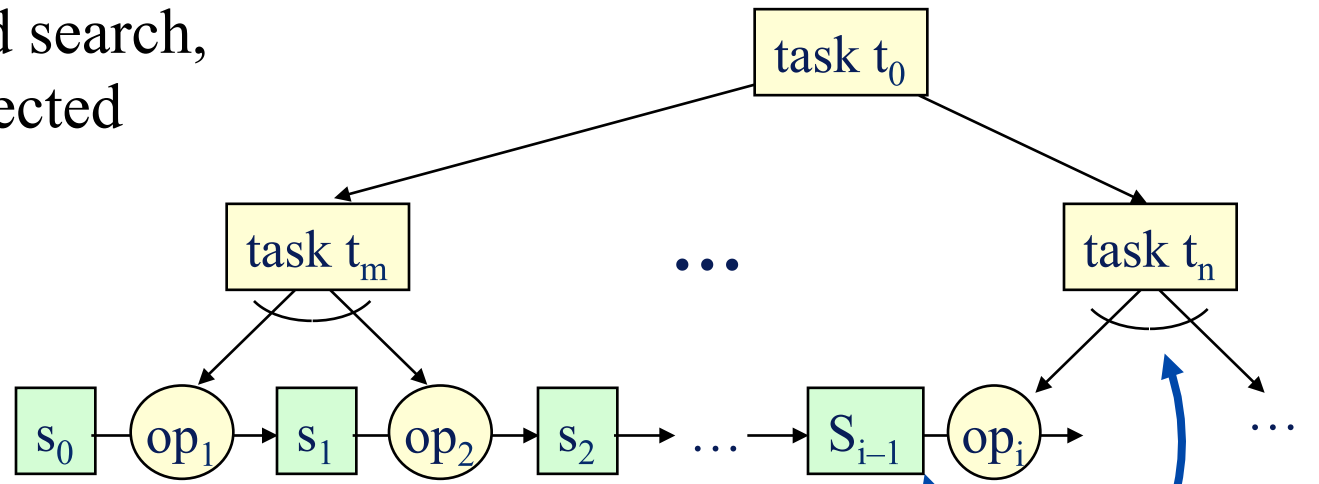
- In HTN planning, there are *two* choices to make about direction:
  - forward or backward
  - up or down



- TFD goes *down* and *forward*

Comparison to Forward and Backward Search

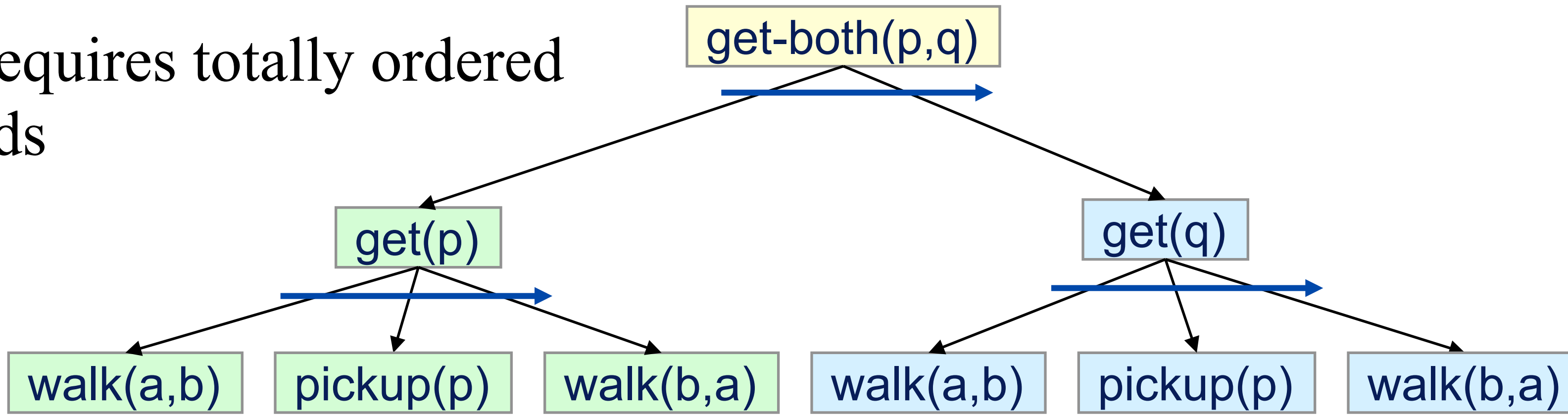
- Like a backward search, TFD is goal-directed
  - Goals correspond to tasks



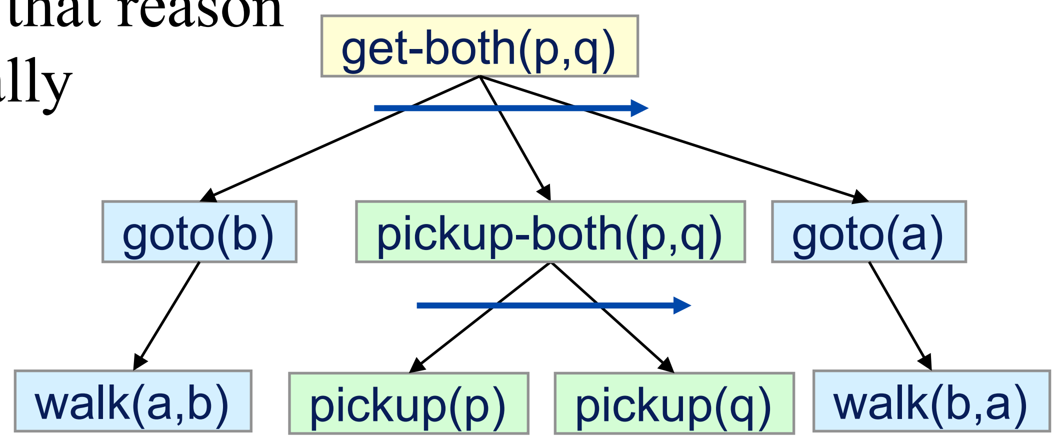
- Like a forward search, it generates actions in the same order in which they'll be executed
- Whenever we want to plan the next task
  - we've already planned everything that comes before it
  - Thus, we know the current state of the world

Limitation of Ordered-Task Planning

- TFD requires totally ordered methods

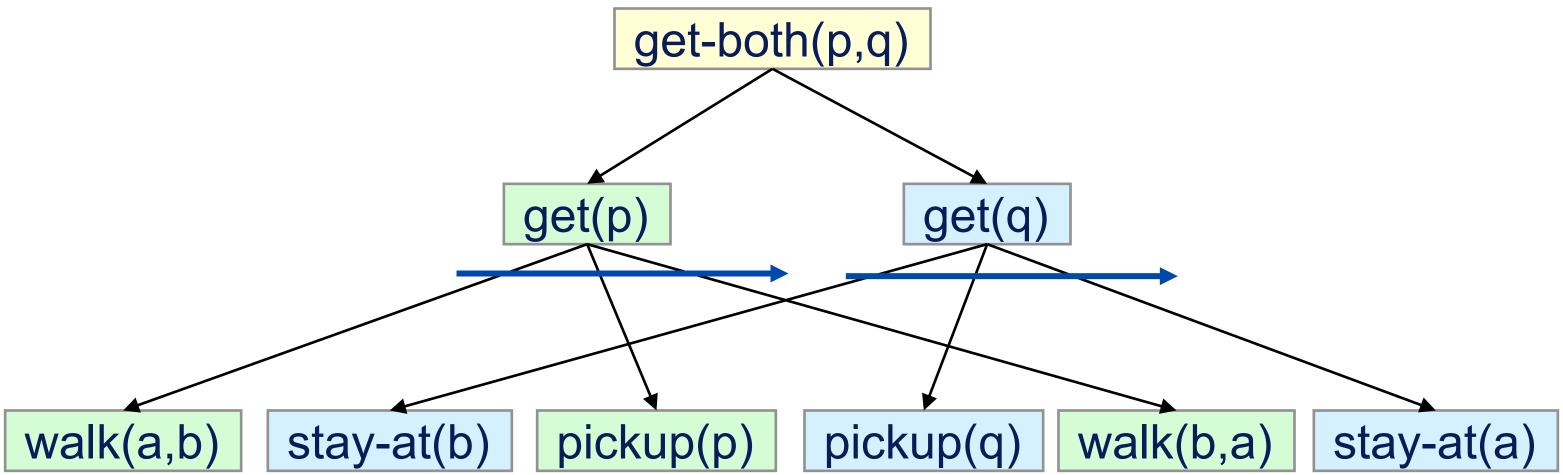


- Can't interleave subtasks of different tasks
- Sometimes this makes things awkward
  - Need to write methods that reason globally instead of locally



Partially Ordered Methods

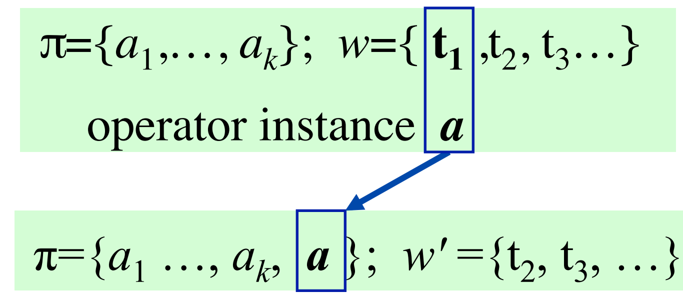
- With partially ordered methods, the subtasks can be interleaved

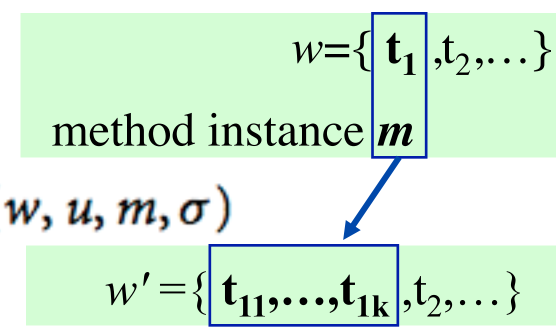


- Fits many planning domains better
- Requires a more complicated planning algorithm

Algorithm for Partial-Order STNs

PFD(s, w, O, M)
  if $w = \emptyset$ then return the empty plan
  nondeterministically choose any $u \in w$ that has no predecessors in w
  if t_u is a primitive task then
    active $\leftarrow \{(a, \sigma) \mid a$ is a ground instance of an operator in O,
      σ is a substitution such that name$(a) = \sigma(t_u)$,
      and a is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(a, \sigma) \in$ *active*
    $\pi \leftarrow$ PFD$(\gamma(s, a), \sigma(w - \{u\}), O, M)$
    if $\pi =$ failure then return failure
    else return $a.\pi$
  else
    active $\leftarrow \{(m, \sigma) \mid m$ is a ground instance of a method in M,
      σ is a substitution such that name$(m) = \sigma(t_u)$,
      and m is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(m, \sigma) \in$ *active*
    nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
    return(PFD(s, w', O, M))





Algorithm for Partial-Order STNs

PFD(s, w, O, M)
    if $w = \emptyset$ then return the empty plan
    nondeterministically choose any $u \in w$ that has no predecessors in w

- Intuitively, w is a partially ordered set of tasks $\{t_1, t_2, \ldots\}$
  - But w may contain a task more than once
    - » e.g., travel from UMD to LAAS twice
  - The mathematical definition of a set doesn't allow this
- Define w as a partially ordered set of *task nodes* $\{u_1, u_2, \ldots\}$
  - Each task node u corresponds to a task t_u
- In my explanations, I'll talk about t and ignore u

    else return $a.\pi$
  else
    active $\leftarrow \{(m,\sigma) \mid m$ is a ground instance of a method in M,
      σ is a substitution such that name$(m) = \sigma(t_u)$,
      and m is applicable to $s\}$
    if *active* $= \emptyset$ then return failure
    nondeterministically choose any $(m,\sigma) \in$ *active*
    nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
    return(PFD(s, w', O, M))

$\}$; $w=\{$ $\mathbf{t_1}$ $,t_2, t_3\ldots\}$
instance $\boldsymbol{a}$

$\boldsymbol{a}$ $\}$; $w' = \{t_2, t_3, \ldots\}$

$w=\{$ $\mathbf{t_1}$ $,t_2,\ldots\}$
method instance $\boldsymbol{m}$

$w' = \{$ $\mathbf{t_{11},\ldots,t_{1k}}$ $,t_2,\ldots\}$

Algorithm for Partial-Order STNs

```
PFD(s, w, O, M)
    if w = ∅ then return the empty plan
    nondeterministically choose any u ∈ w that has no predecessors in w
    if t_u is a primitive task then
        active ← {(a, σ) | a is a ground instance of an operator in O,
                           σ is a substitution such that name(a) = σ(t_u),
                           and a is applicable to s}
        if active = ∅ then return failure
        nondeterministically choose any (a, σ) ∈ active
        π ← PFD(γ(s, a), σ(w − {u}), O, M)
        if π = failure then return failure
        else return a.π
    else
        active ← {(m, σ) | m is a ground instance of a method in M,
                           σ is a substitution such that name(m) = σ(t_u),
                           and m is applicable to s}
        if active = ∅ then return failure
        nondeterministically choose any (m, σ) ∈ active
        nondeterministically choose any task network w′ ∈ δ(w, u, m, σ)
        return(PFD(s, w′, O, M))
```

$\pi=\{a_1,\ldots,a_k\}$; $w=\{\mathbf{t_1},t_2,t_3\ldots\}$

operator instance $\boldsymbol{a}$

$\pi=\{a_1 \ldots, a_k, \boldsymbol{a}\}$; $w'=\{t_2, t_3, \ldots\}$

$w=\{\mathbf{t_1},t_2,\ldots\}$

method instance $\boldsymbol{m}$

$w'=\{\mathbf{t_{11},\ldots,t_{1k}},t_2,\ldots\}$

Algorithm for Partial-Order STNs

```
PFD(s, w, O, M)
    if w = Ø then return the empty plan
    nondeterministically choose any u ∈ w that has no predecessors in w
    if t_u is a primitive task then
        active ←
        if active
        nondeter
        π ← PFD
        if π = failure then return failure
        else return a.π
    else
        active ← {(m,σ) | m is a ground instance of a method in M,
                        σ is a substitution such that name(m) = σ(t_u),
                        and m is applicable to s}
        if active = Ø then return failure
        nondeterministically choose any (m,σ) ∈ active
        nondeterministically choose any task network w′ ∈ δ(w, u, m, σ)
        return(PFD(s, w′, O, M))
```

δ(*w, u, m,* σ) has a complicated definition in the book. Here's what it means:

- We nondeterministically selected t_1 as the task to begin first
  - i.e., do t_1's first subtask before the first subtask of every $t_i \neq t_1$
- Insert ordering constraints to ensure that this happens

$\pi = \{a_1 \ldots, a_k, \boldsymbol{a}\}$; $w' = \{t_2, t_3 \ldots\}$

$w = \{\mathbf{t_1}, t_2, \ldots\}$

method instance ***m***

$w' = \{\mathbf{t_{11}, \ldots, t_{1k}}, t_2, \ldots\}$

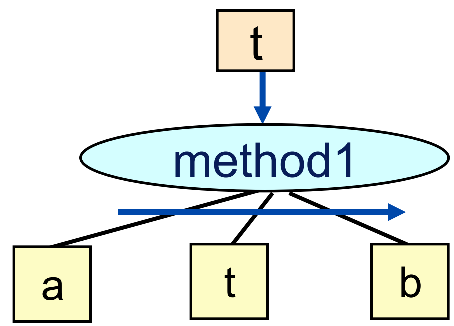
Comparison to Classical Planning

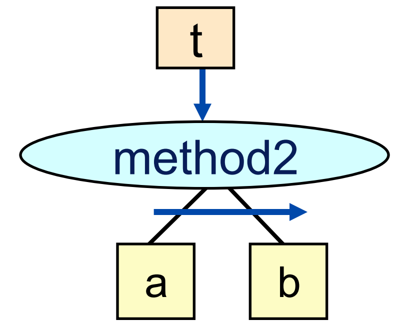
STN planning is strictly more expressive than classical planning

- Any classical planning problem can be translated into an ordered-task-planning problem in polynomial time
- Several ways to do this. One is roughly as follows:
  - For each goal or precondition *e*, create a task t_e
  - For each operator o and effect *e*, create a method $m_{o,e}$
    - » Task: t_e
    - » Subtasks: $t_{c1}, t_{c2}, \ldots, t_{cn}, o$, where $c_1, c_2, \ldots, c_n$ are the preconditions of *o*
    - » Partial-ordering constraints: each t_{ci} precedes *o*
- (I left out some details, such as how to handle deleted-condition interactions)

Comparison to Classical Planning (cont.)

- Some STN planning problems aren't expressible in classical planning
- Example:
  - Two STN methods:
    - No arguments
    - No preconditions





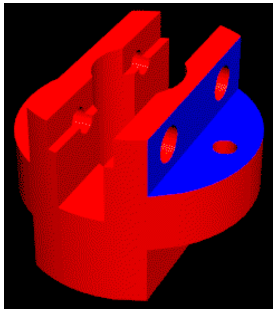
  - Two operators, a and b
    - Again, no arguments and no preconditions
  - Initial state is empty, initial task is t
  - Set of solutions is $\{a^n b^n \mid n > 0\}$
  - No classical planning problem has this set of solutions
    - The state-transition system is a finite-state automaton
    - No finite-state automaton can recognize $\{a^n b^n \mid n > 0\}$
- Can even express undecidable problems using STNs

Increasing Expressivity Further

- If we always know the current state, we can make several enhancements:
  - States can be arbitrary data structures

Us:	East declarer, West dummy
Opponents:	defenders, South & North
Contract:	East – 3NT
On lead:	West at trick 3

East:	♠KJ74
West:	♠A2
Out:	♠QT98653



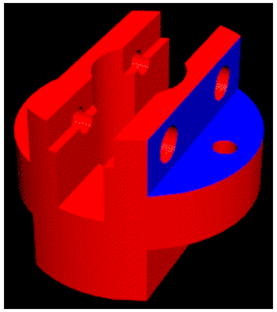
  - Preconditions and effects can include
    - » logical inferences (e.g., Horn clauses)
    - » complex numeric computations
    - » interactions with other software packages
- e.g., SHOP and SHOP2
  - http://www.cs.umd.edu/projects/shop
  - algorithms similar to PFD and PFD, with the above enhancements
  - SHOP2 won an award at the 2002 Planning Competition

Increasing Expressivity Further

- If we always know the current state, we can make several enhancements:
  - States can be arbitrary data structures

Us:	East declarer, West dummy
Opponents:	defenders, South & North
Contract:	East – 3NT
On lead:	West at trick 3

East:	♠KJ74
West:	♠A2
Out:	♠QT98653



  - Preconditions and effects can include
    - » logical inferences (e.g., Horn clauses)
    - » complex numeric computations
    - » interactions with other software packages

- TLPlan and TALplanner also have some (but not all) of these enhancements

- What about adding them to a planner like FastForward?

Example

method travel-by-foot
- precond: $distance(x, y) \leq 2$
- task: travel(a, x, y)
- subtasks: walk(a, x, y)

method travel-by-taxi
- task: travel(a, x, y)
- precond: $cash(a) \geq 1.5 + 0.5 \times distance(x, y)$
- subtasks: $\langle$call-taxi(a, x), ride(a, x, y), pay-driver$(a, x, y)\rangle$

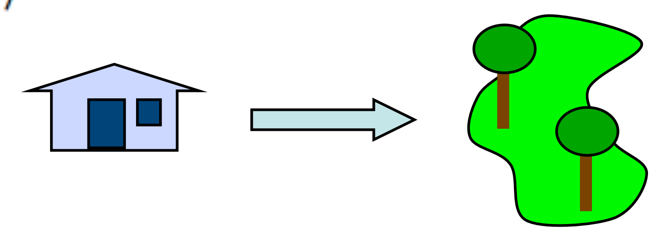
operator walk
- precond: $location(a) = x$
- effects: $location(a) \leftarrow y$

operator call-taxi(a, x)
- effects: $location(taxi) \leftarrow x$

operator ride-taxi (a, x)
- precond: $location(taxi) = x$, $location(a) = x$
- effects: $location(taxi) \leftarrow y$, $location(a) \leftarrow y$

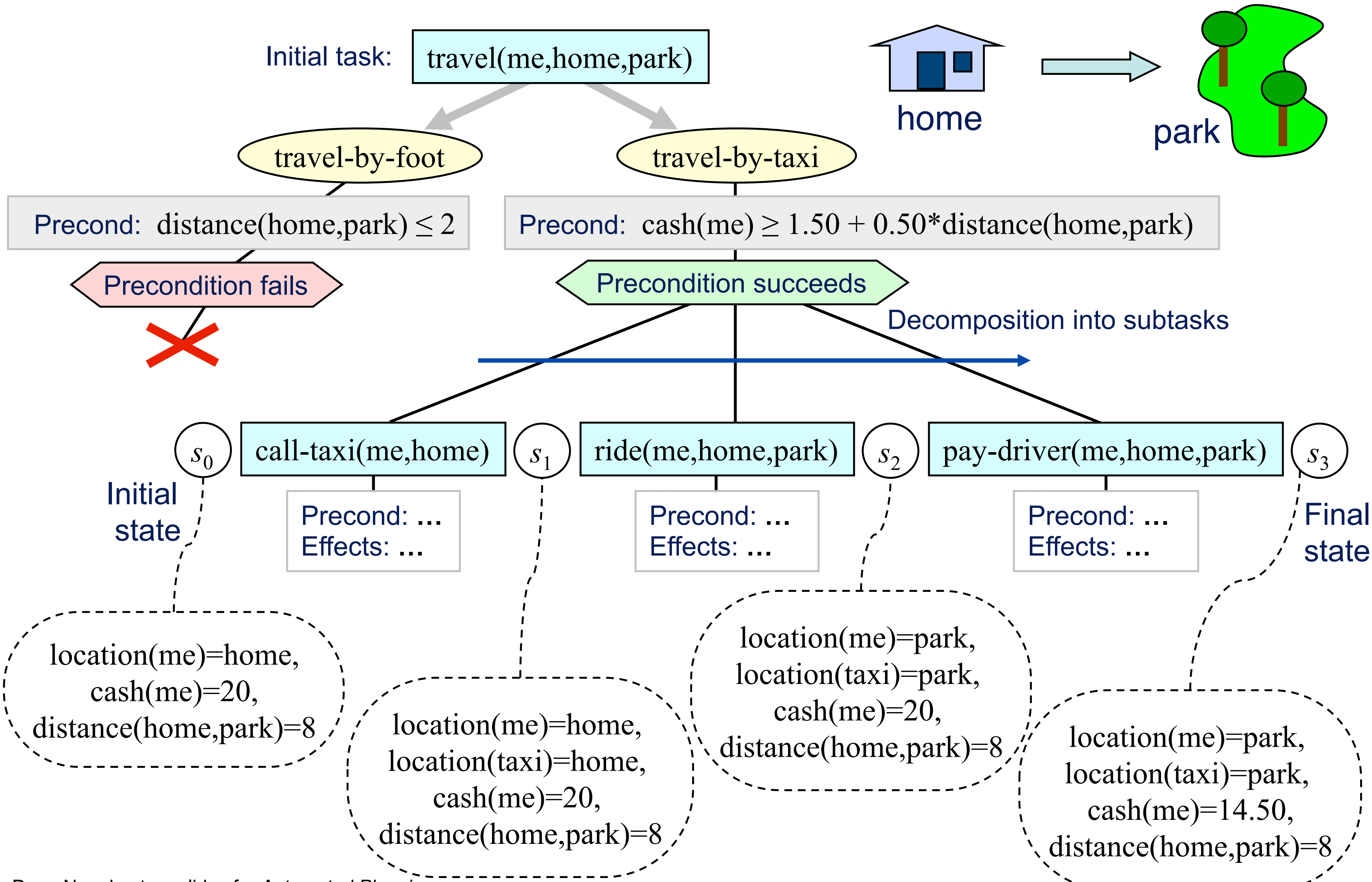
operator pay-driver(a, x, y)
- precond: $cash(a) \geq 1.5 + 0.5 \times distance(x, y)$
- effects: $cash(a) \leftarrow cash(a) - 1.5 - 0.5 \times distance(x, y)$

- Simple travel-planning domain
  - State-variable formulation
- Planning problem:
  - I'm at home, I have \$20
  - Want to go to a park 8 miles away



  - s_0 = {location(me) = home, cash(me) = 20, distance(home,park) = 8}
  - t_0 = travel(me,home,park)

Example, Continued



HTN Planning

- HTN planning can be even more general
  - Can have constraints associated with tasks and methods
    - » Things that must be true before, during, or afterwards
  - Some algorithms use causal links and threats like those in PSP
- There's a little about this in the book
  - I won't discuss it

SHOP & SHOP2 vs. TLPlan & TALplanner

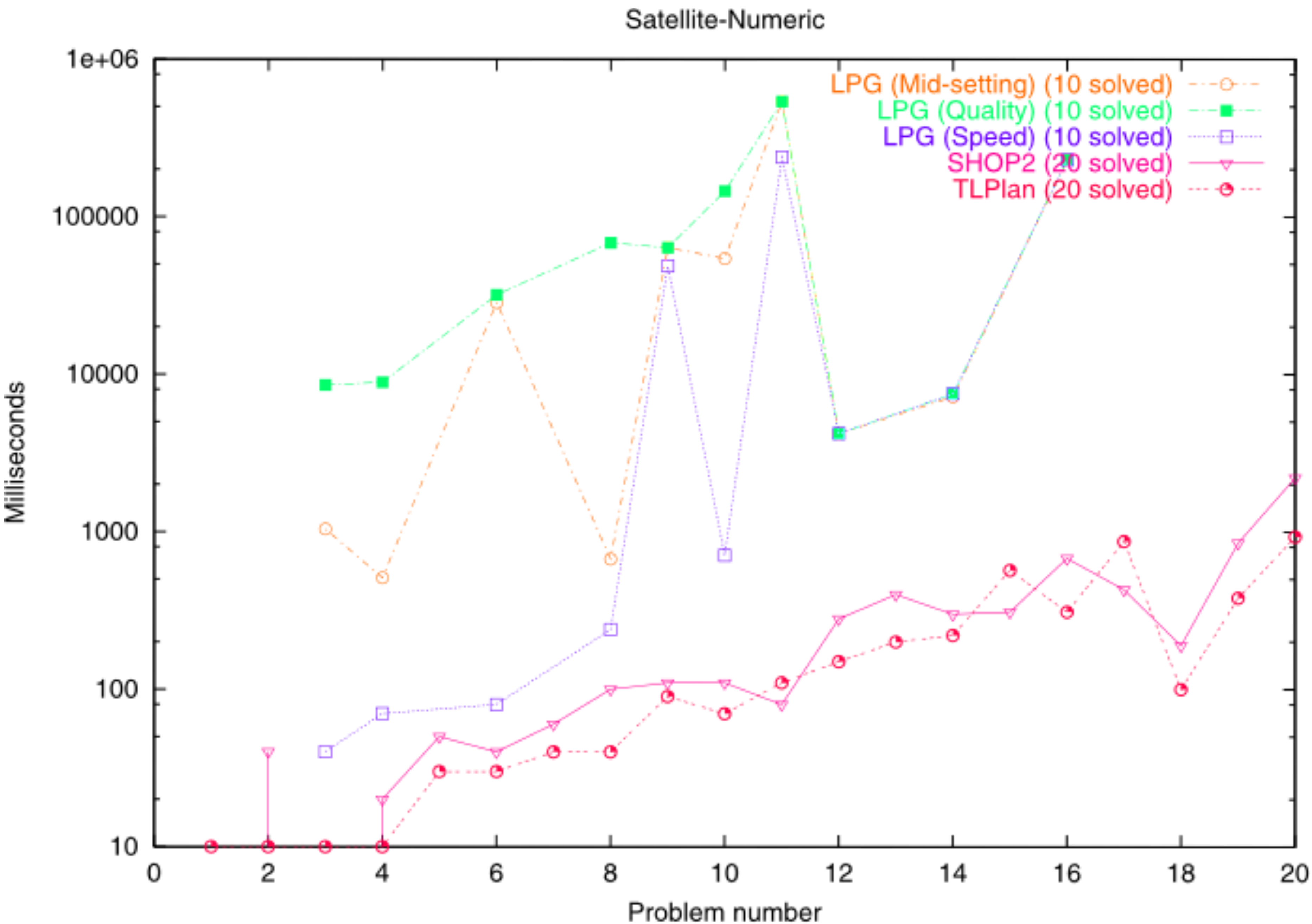
- These planners have equivalent expressive power
  - Turing-complete, because both allow function symbols
- They know the current state at each point during the planning process, and use this to prune actions
  - Makes it easy to call external subroutines, do numeric computations, etc.
- Main difference: how the pruning is done
  - SHOP and SHOP2: the methods say what *can* be done
    - » Don't do anything unless a method says to do it
  - TLPlan and TALplanner: the say what *cannot* be done
    - » Try everything that the control rules don't prohibit
- Which approach is more convenient depends on the problem domain

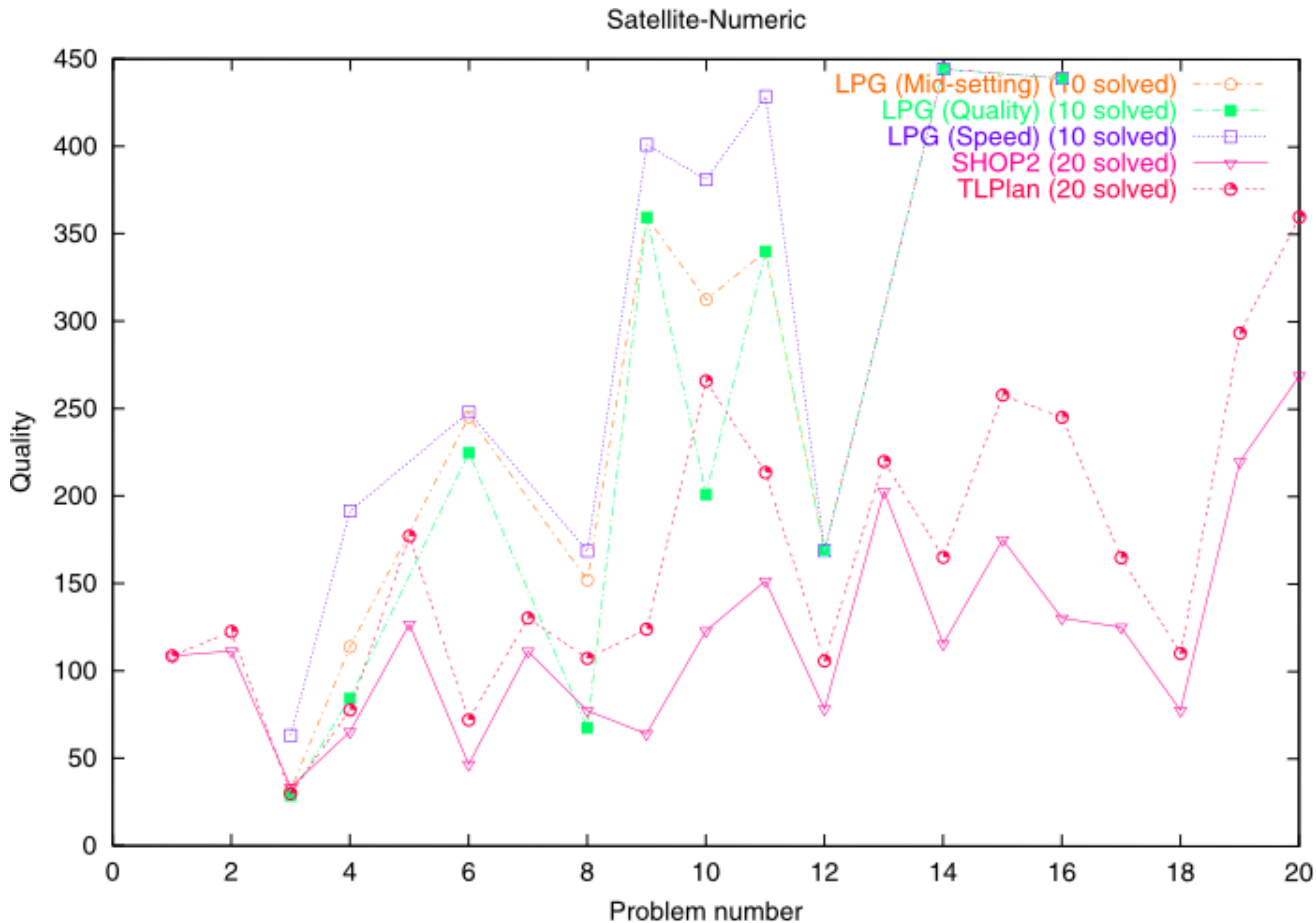
Domain-Configurable Planners Compared to Classical Planners

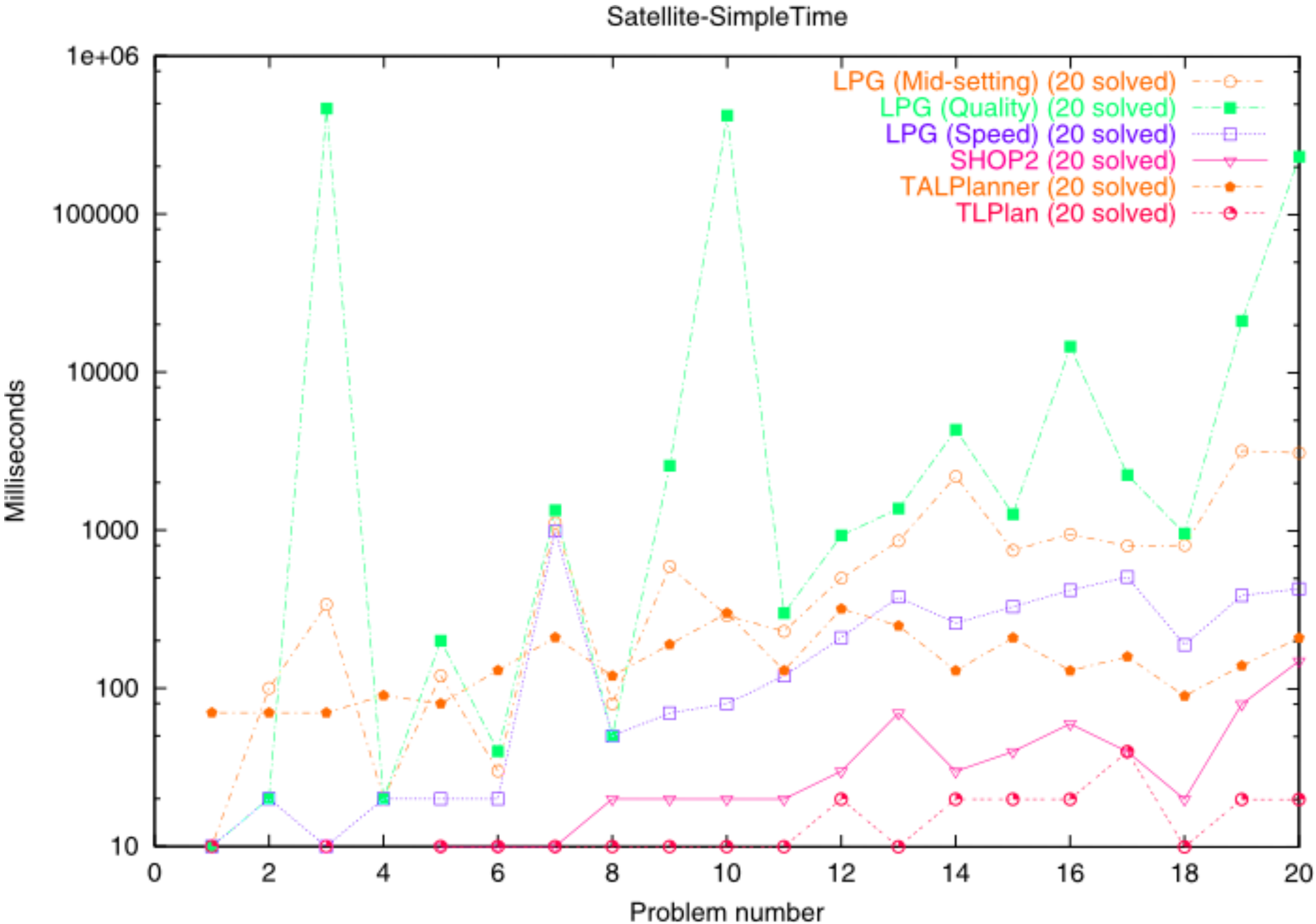
- Disadvantage: writing a knowledge base can be more complicated than just writing classical operators
- Advantage: can encode "recipes" as collections of methods and operators
  - Express things that can't be expressed in classical planning
  - Specify standard ways of solving problems
    - Otherwise, the planning system would have to derive these again and again from "first principles," every time it solves a problem
    - Can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)

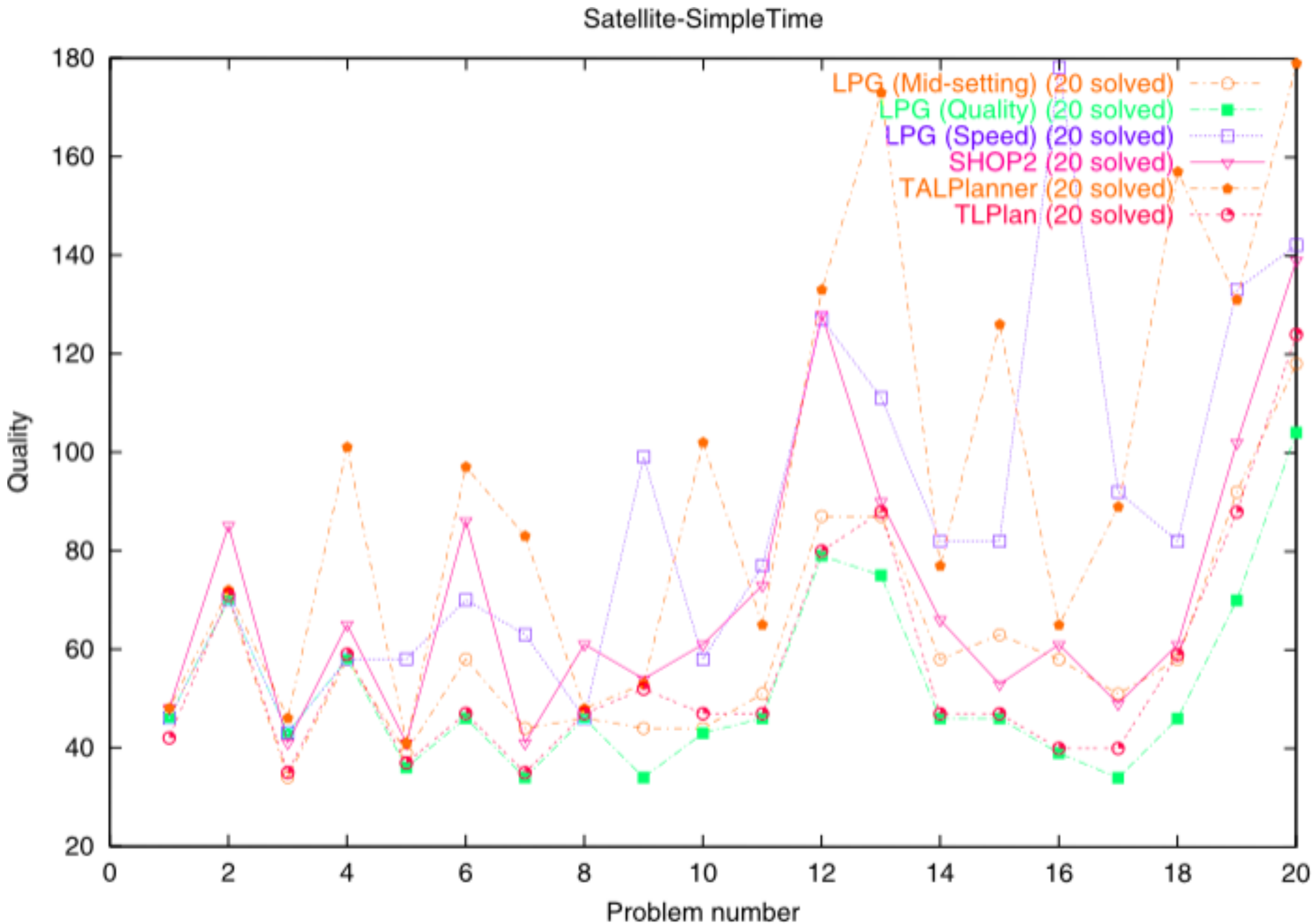
Example from the AIPS-2002 Competition

- **The satellite domain**
  - Planning and scheduling observation tasks among multiple satellites
  - Each satellite equipped in slightly different ways
- Several different versions. I'll show results for the following:
  - **Simple-time:**
    - concurrent use of different satellites
    - data can be acquired more quickly if they are used efficiently
  - **Numeric:**
    - fuel costs for satellites to slew between targets; finite amount of fuel available.
    - data takes up space in a finite capacity data store
    - Plans are expected to acquire all the necessary data at minimum fuel cost.
  - **Hard Numeric:**
    - *no logical goals at all* – thus even the null plan is a solution
    - Plans that acquire more data are better – thus the null plan has no value
    - None of the classical planners could handle this

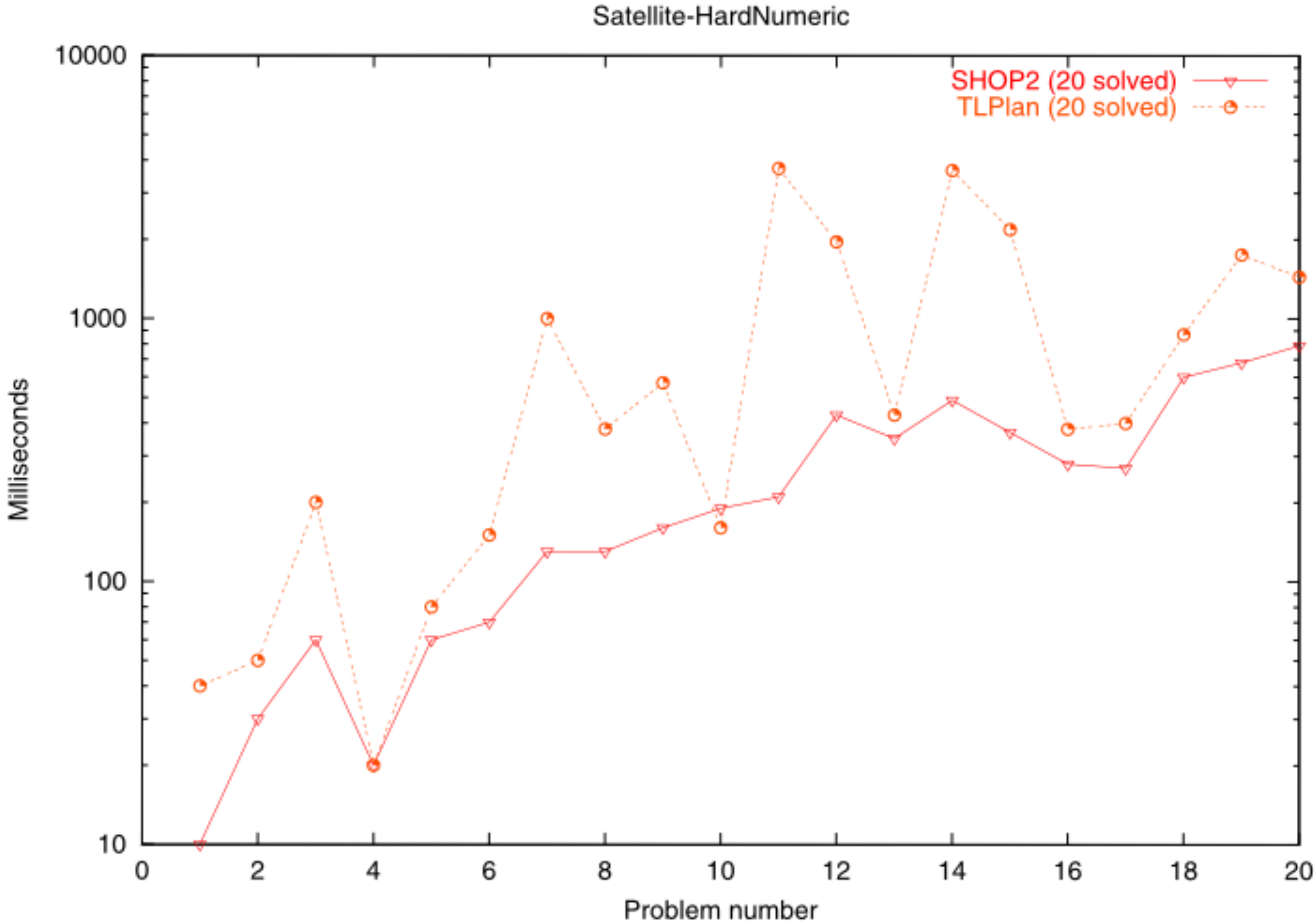

Satellite-Numeric
Milliseconds
Problem number
LPG (Mid-setting) (10 solved)
LPG (Quality) (10 solved)
LPG (Speed) (10 solved)
SHOP2 (20 solved)
TLPlan (20 solved)
1e+06
100000
10000
1000
100
10
0
2
4
6
8
10
12
14
16
18
20

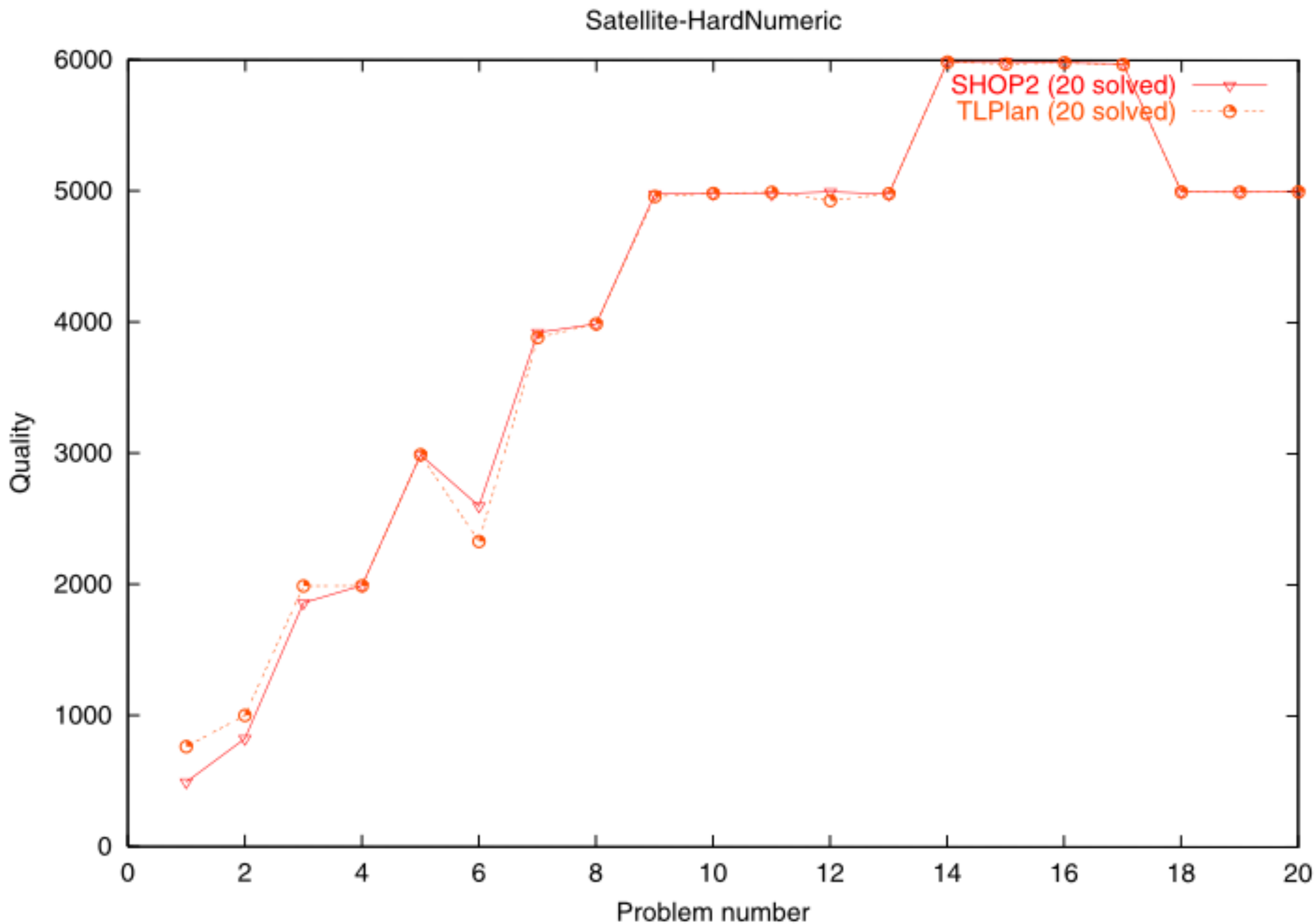

Satellite-Numeric
Quality
Problem number
0
50
100
150
200
250
300
350
400
450
0
2
4
6
8
10
12
14
16
18
20
LPG (Mid-setting) (10 solved)
LPG (Quality) (10 solved)
LPG (Speed) (10 solved)
SHOP2 (20 solved)
TLPlan (20 solved)


Satellite-SimpleTime
1e+06
100000
10000
1000
100
10
Milliseconds
0
2
4
6
8
10
12
14
16
18
20
Problem number
LPG (Mid-setting) (20 solved)
LPG (Quality) (20 solved)
LPG (Speed) (20 solved)
SHOP2 (20 solved)
TALPlanner (20 solved)
TLPlan (20 solved)



Satellite-SimpleTime
Quality
Problem number
LPG (Mid-setting) (20 solved)
LPG (Quality) (20 solved)
LPG (Speed) (20 solved)
SHOP2 (20 solved)
TALPlanner (20 solved)
TLPlan (20 solved)
180
160
140
120
100
80
60
40
20
0
2
4
6
8
10
12
14
16
18
20


Satellite-HardNumeric
SHOP2 (20 solved)
TLPlan (20 solved)
Milliseconds
10000
1000
100
10
0
2
4
6
8
10
12
14
16
18
20
Problem number


Satellite-HardNumeric
SHOP2 (20 solved)
TLPlan (20 solved)
Quality
Problem number
0
1000
2000
3000
4000
5000
6000
0
2
4
6
8
10
12
14
16
18
20