Planning and Scheduling: Hierarchical Task Network Planning

Prof. Dr.-Ing. Gerhard K. Kraetzschmar

Hochschule
Bonn-Rhein-Sieg

b-it Bonn-Aachen International Center for Information Technology

© 2009 Gerhard K. Kraetzschmar

Acknowledgements

These slides are based on those slides by Dana Nau, Gerhard Wickler, Hai Hoang, José Luis Ambite

Several improvements were applied by Iman Awaad

© 2009 Gerhard K. Kraetzschmar

Motivation

- We may already have an idea
  how to go about solving problems in a planning domain
- E.g.: 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. for flying:
    - buy ticket from local airport to remote airport
    - travel to local airport
    - fly to remote airport
    - travel to final destination
- How to enable planning systems to make use of such recipes?

© 2009 Gerhard K. Kraetzschmar

Control Rules v HTN Planning

1. Control rules (Chapter 10):
   - Write rules to prune every action that **does not** fit the recipe
2. Hierarchical Task Network (HTN) planning:
   - Describe the actions and subtasks that **do** fit the recipe

Objective of HTN planning: perform a given set of tasks

- Inputs include:
  - *Operators*: that can directly perform a *primitive* task
  - *Methods*: recipes for decomposing a complex/*non-primitive task* into simpler non-primitive or primitive subtasks
- Planning process:
  - *Decompose* non-primitive tasks recursively until primitive tasks are reached

© 2009 Gerhard K. Kraetzschmar

Hierarchical Decomposition & Problem Reduction

To *get to a conference* in ?x, *get to the airport, take a plane* to ?x, then *go to the conference hotel*

- To get to the airport, either drive or take a cab
- If you have money for the taxi fare:
- Enter the cab, say “I want to go to ?y”, wait until you are at ?y, pay the fare, then exit the taxi”

- Idea is to capture the hierarchical structure of the planning domain
  - contains complex tasks and schemas for reducing them.
- Reduction schemas:
  - given by the designer
  - express preferred ways to accomplish a task

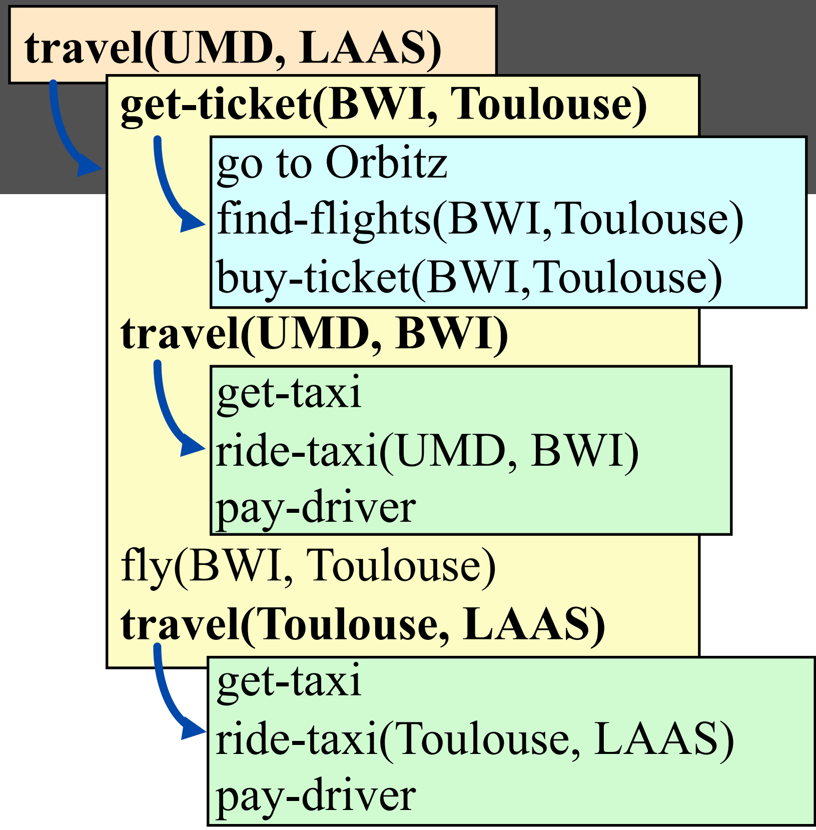
© 2009 Gerhard K. Kraetzschmar

Outline

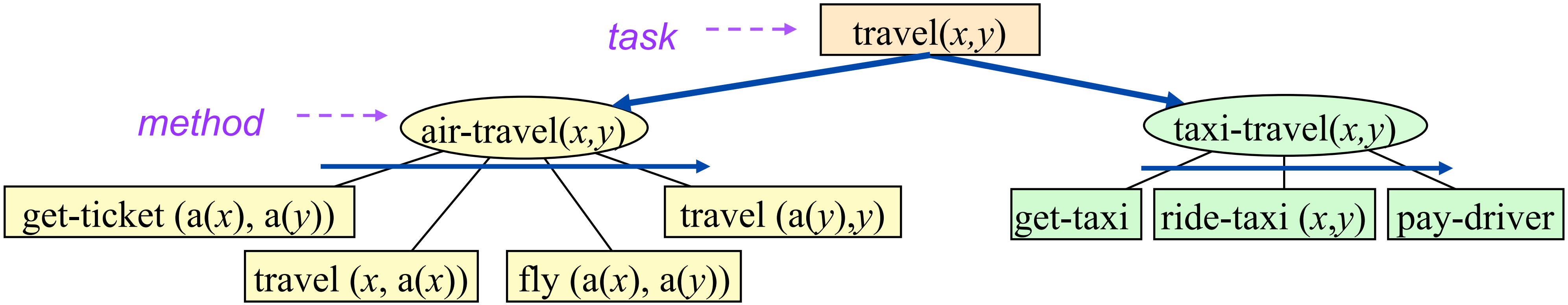
- Main idea behind HTN planning
- STNs: Representation and planning algorithms
  - Total order
  - Partial order
- Generalizing the formalism and algorithm to HTN
- Expressivity: comparison to classical planning and control rules
- Experimental Results

© 2009 Gerhard K. Kraetzschmar

HTN Planning



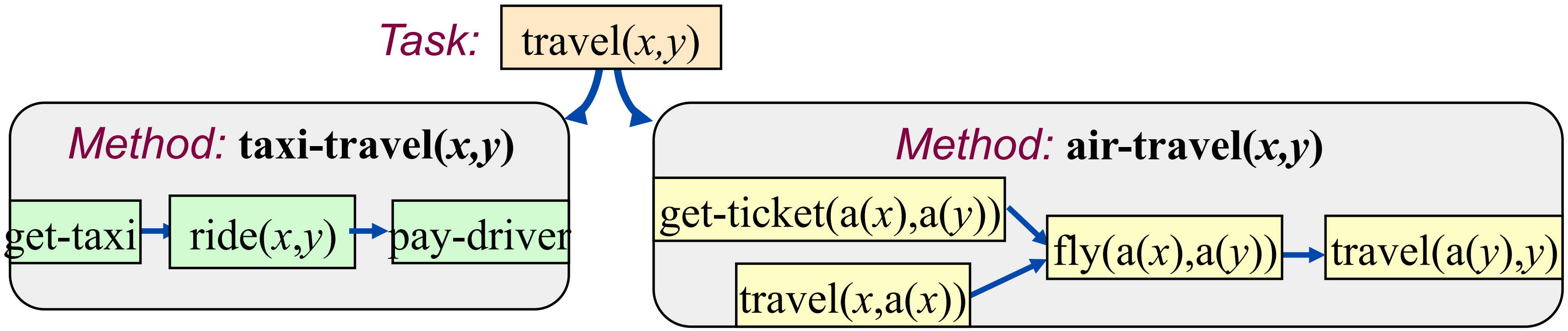
- A type of *problem reduction*
- Decompose *tasks* into *subtasks*
- Handle constraints (e.g., taxi not good for long distances)
- Resolve interactions (e.g., take taxi early enough to catch plane)
- If necessary, backtrack and try other decompositions



© 2009 Gerhard K. Kraetzschmar

HTN Planning

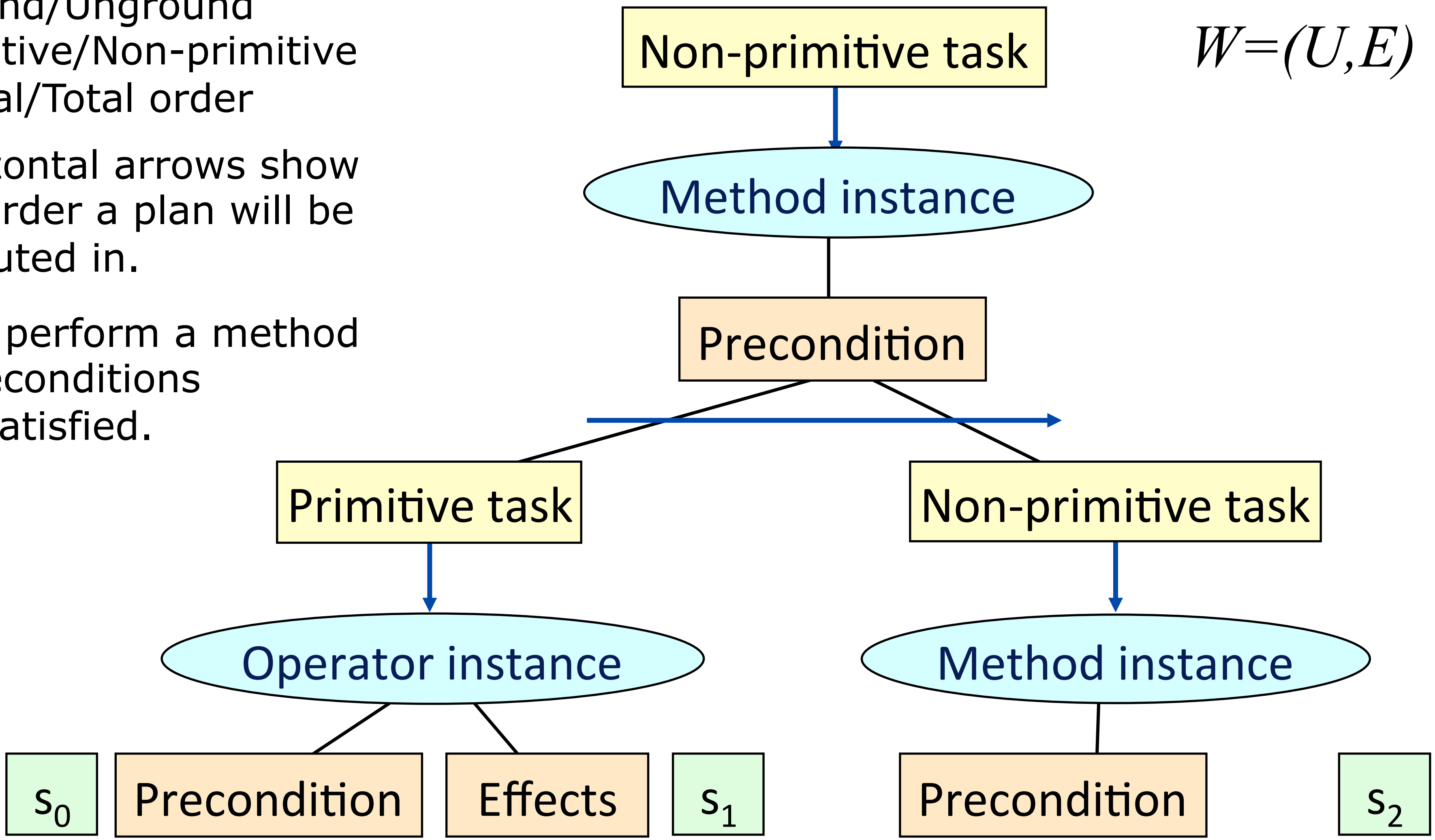
- HTN planners may be domain-specific
  - e.g., see Chapters 20 (robotics) and 23 (bridge)
- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description defining operators and also methods
- Problem description
  - domain description, initial state, initial task network



© 2009 Gerhard K. Kraetzschmar

HTN Planning: Task Networks

- Ground/Unground
- Primitive/Non-primitive
- Partial/Total order
- Horizontal arrows show the order a plan will be executed in.
- Only perform a method if preconditions are satisfied.



HTN v what we’ve seen so far

- What stays the same:
  - Each state of the world is represented by a set of atoms
  - Each action corresponds to a deterministic state transition
  - Terms, literals, operators, actions, plans have same meaning
  - E.g. (block b1) (block b2) (block b3) (block b4) (on-table b1) (on b2 b1) (clear b2) (on-table b3) (on b4 b3) (clear b4)

- What’s new:
  - Perform a set of tasks not achieve a set of goals
  - *Methods* describing ways in which tasks can be performed
  - Organized collections of tasks and subtasks called *task networks*

© 2009 Gerhard K. Kraetzschmar

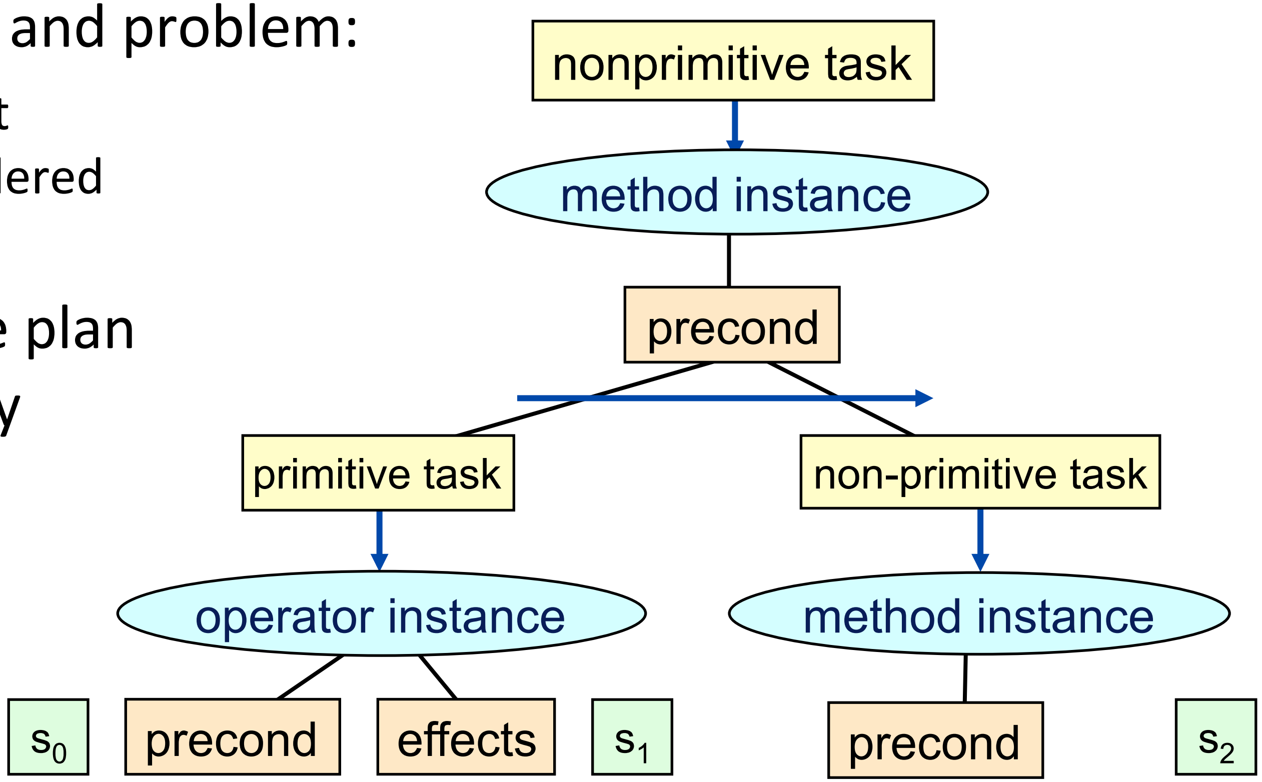
Simple Task Network (STN) Planning

- 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
  - *Non-primitive*: tasks that must be decomposed into subtasks
    - use methods (next slide)

© 2009 Gerhard K. Kraetzschmar

STN: Domains, Planning Problems, Solutions

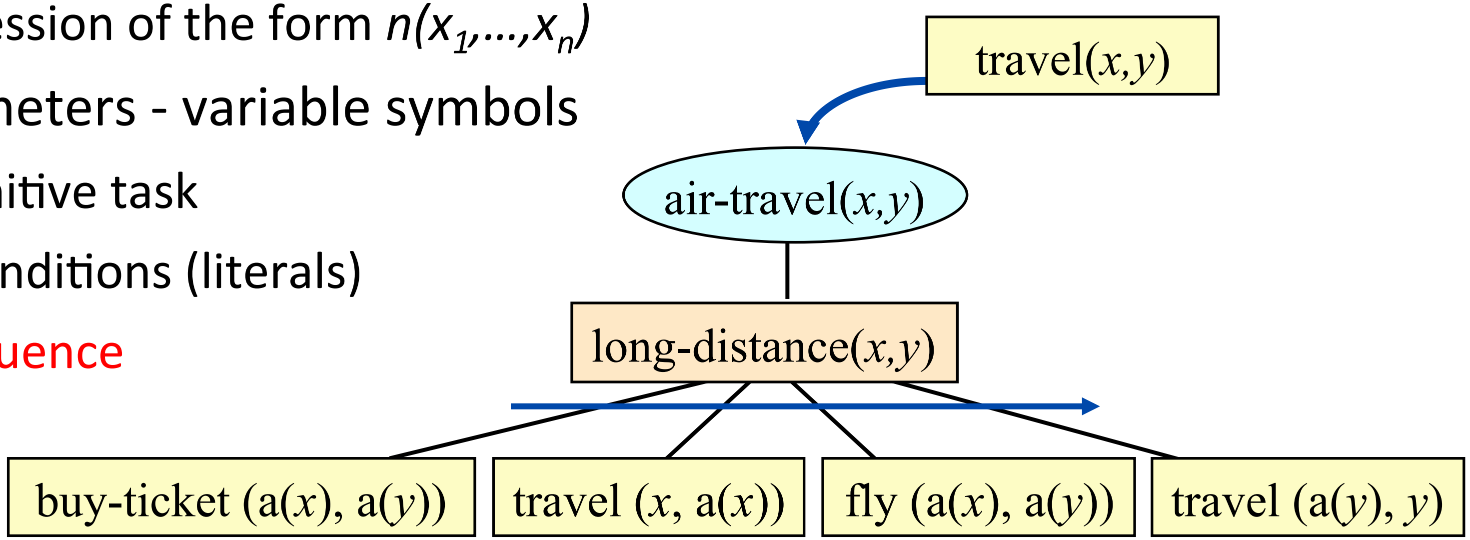
- Domain: methods, operators: $D=(O,M)$
- Problem: initial state, initial task network, operators, methods: $P=(S_0,w_j,O,M)$
- Total-order STN domain and problem:
  - Same as above except that all methods are totally ordered
- Solution: any executable plan that can be generated by recursively applying
  - methods to non-primitive tasks
  - operators to primitive tasks



© 2009 Gerhard K. Kraetzschmar

STN: Methods (Totally-ordered)

- Totally-ordered method: a 4-tuple
  m = (name(m), task(m), precond(m), subtasks(m))
  - name(m): an expression of the form $n(x_1,\dots,x_n)$
  - $x_1,\dots,x_n$ are parameters - variable symbols
  - task(m): a nonprimitive task
  - precond(m): preconditions (literals)
  - subtasks(m): a sequence of tasks $\langle t_1, \dots, t_k \rangle$



air-travel(x,y)

task: travel(x,y)

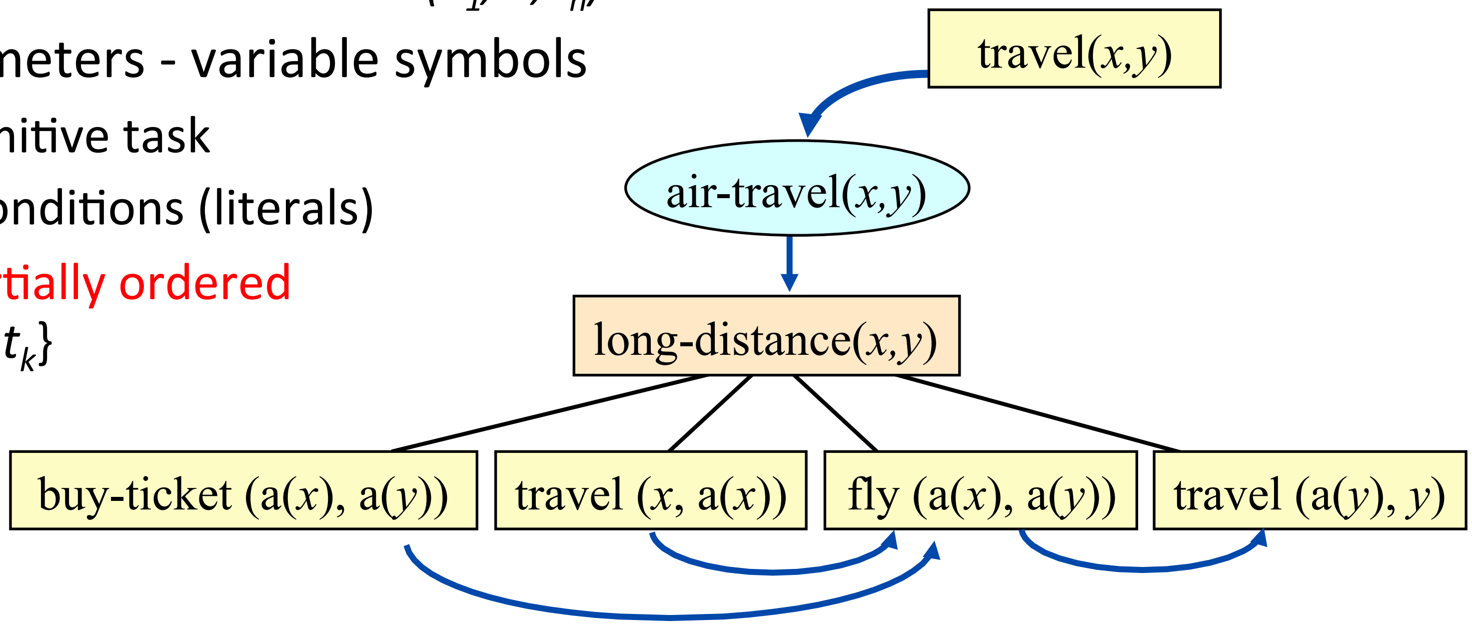
precond: long-distance(x,y)

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

© 2009 Gerhard K. Kraetzschmar

STN: Methods (Partially-ordered)

- 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), $\{(u_1,u_3), (u_2,u_3), (u_3,u_4)\}$

© 2009 Gerhard K. Kraetzschmar

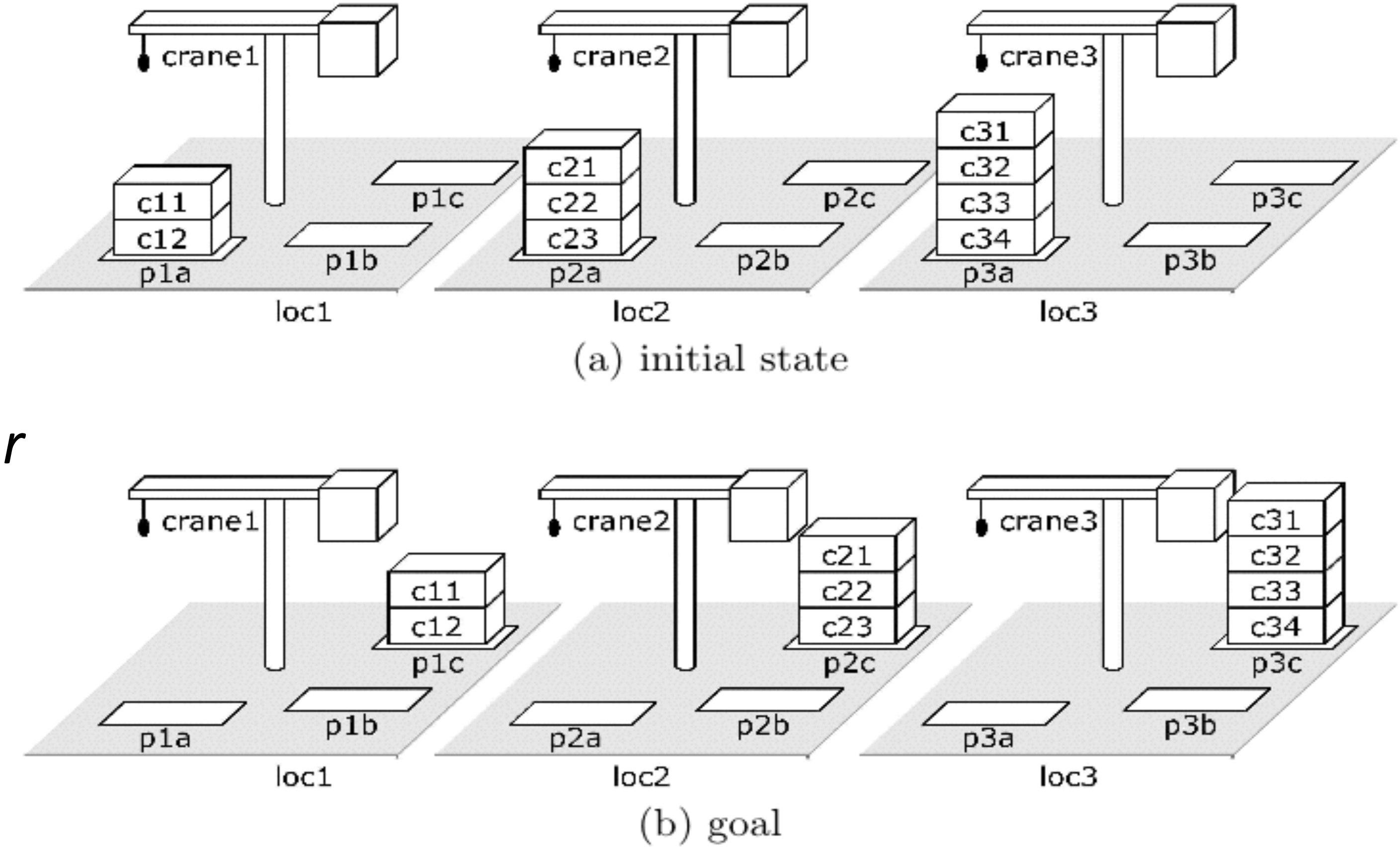
Example: DWR

- Task: Move three stacks of containers in a way that preserves the order of the containers

- One way to move each stack:

- First move the containers from *p* to an intermediate pile *r*

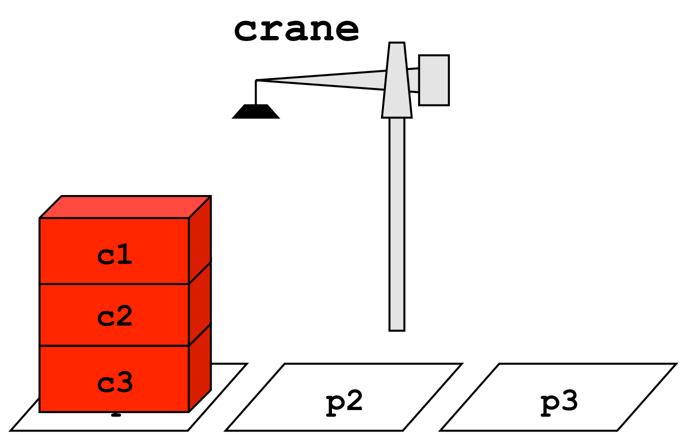
- Then move them from *r* to *q*



© 2009 Gerhard K. Kraetzschmar

Example: DWR

- (informal) methods:
  - move each stack twice: move stack to intermediate pile (reversing order) and then to final destination (reversing order again)
  - move stack: repeatedly/recursively move the topmost container until the stack is empty
  - move top-most: take followed by put action



© 2009 Gerhard K. Kraetzschmar

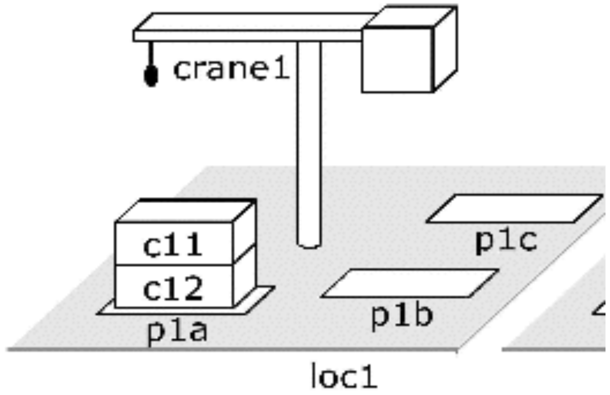
Example: DWR Total-Order Formulation

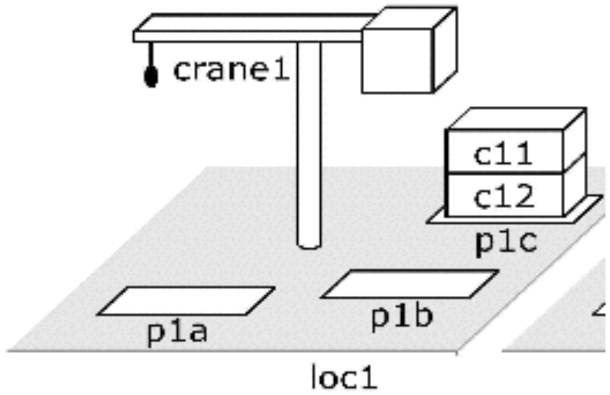
take-and-put$(c, k, l_1, l_2, p_1, p_2, x_1, x_2)$:
task: move-topmost-container(p_1, p_2)
precond: top(c, p_1), on(c, x_1), *; true if p_1 is not empty*
attached(p_1, l_1), belong(k, l_1), *; bind l_1 and k*
attached(p_2, l_2), top(x_2, p_2) *; bind l_2 and x_2*
subtasks: $\langle$take(k, l_1, c, x_1,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





© 2009 Gerhard K. Kraetzschmar

STN: Solving Total-Order 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)

state s; task list T=($\boxed{\mathbf{t_1}}$,t_2,…)
action $\boxed{a}$

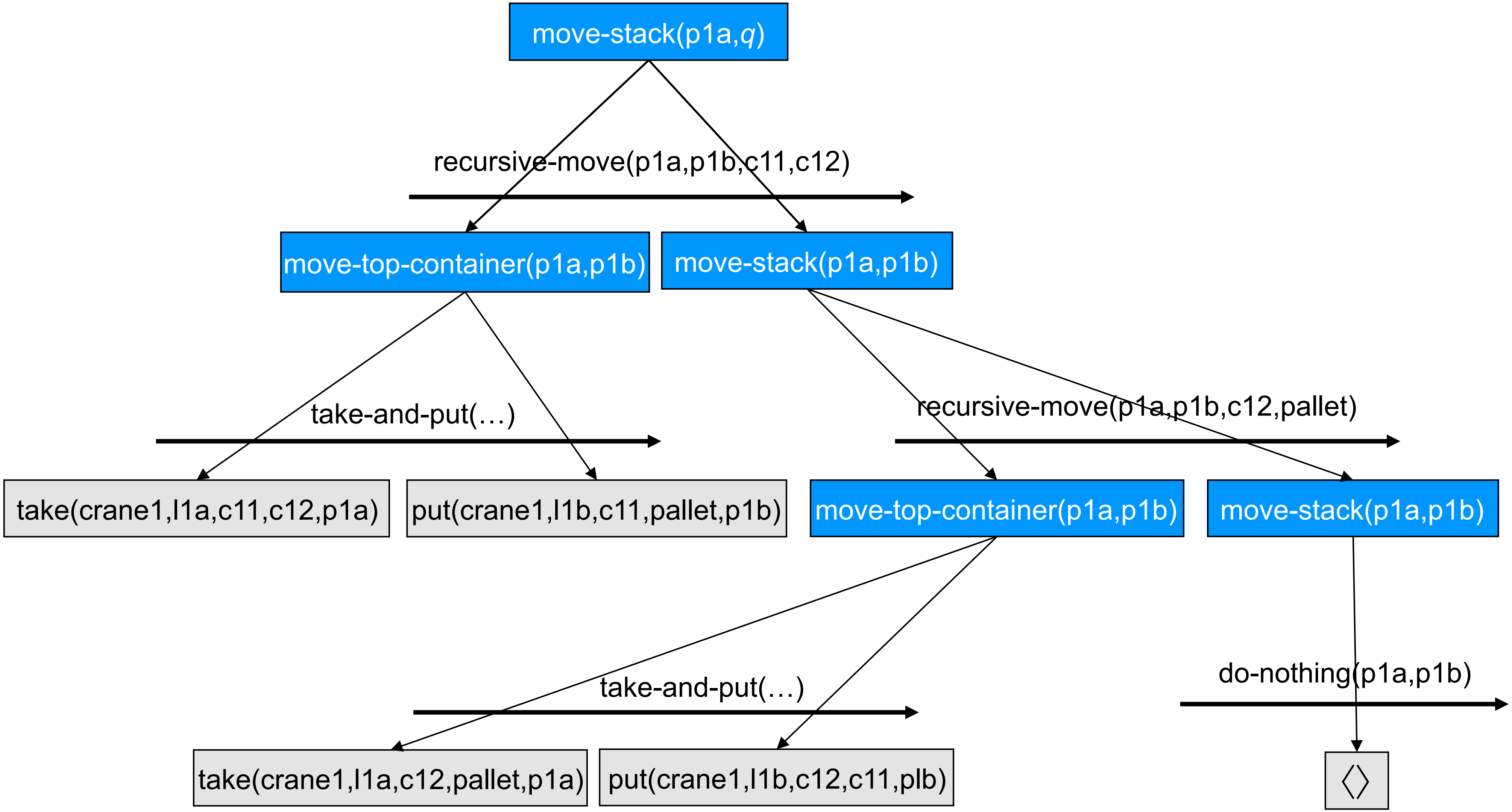
state $\boxed{\gamma(s,a)}$; task list T=(t_2, …)

task list T=($\boxed{\mathbf{t_1}}$,t_2,…)
method instance $\boxed{m}$

task list T=($\boxed{\mathbf{u_1,\ldots,u_k}}$,t_2,…)

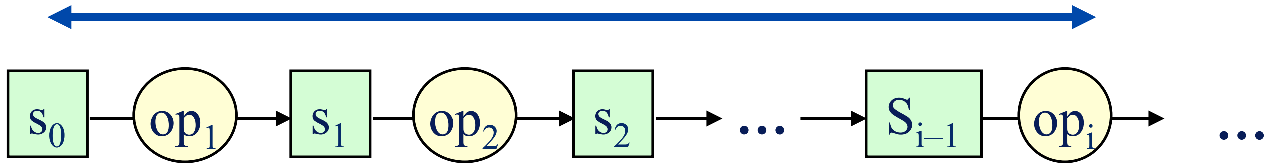
© 2009 Gerhard K. Kraetzschmar

Example: DWR Decomposition Tree - TFD

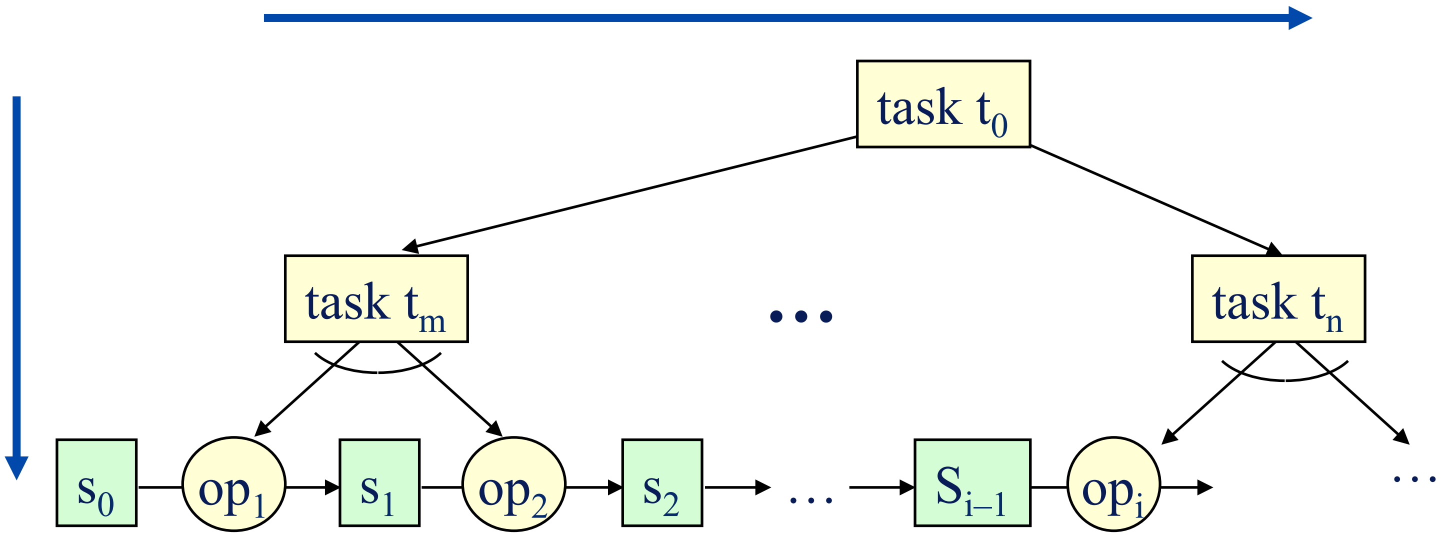


Comparison to Forward and Backward Search

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



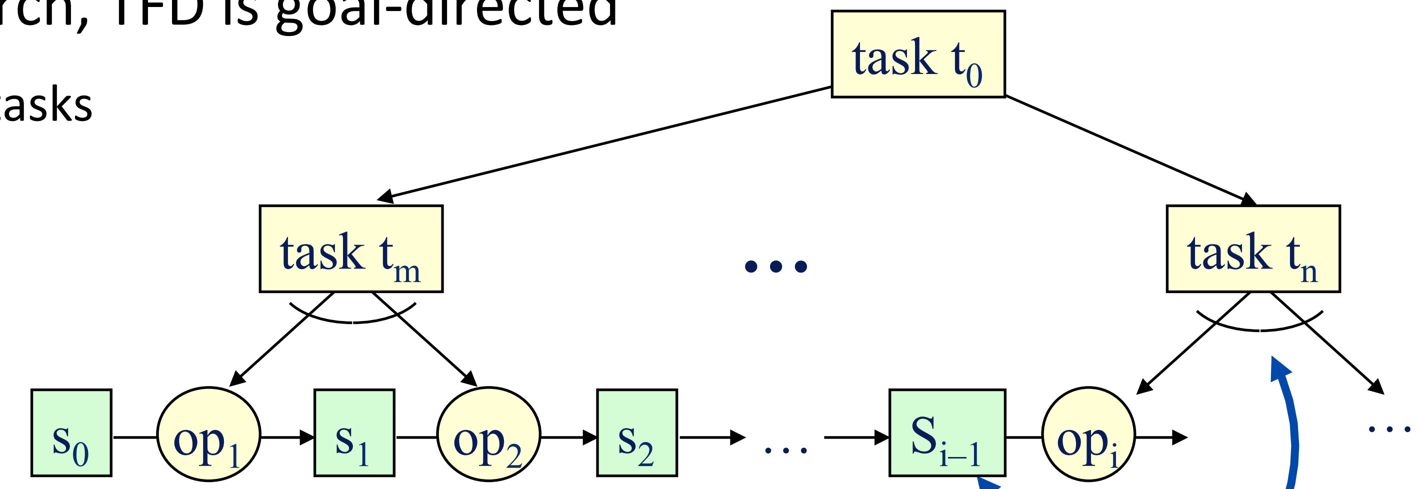
- In HTN planning, there are *two* choices to make about direction:
  - forward or backward
  - up or down
- TFD goes down and forward



© 2009 Gerhard K. Kraetzschmar

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

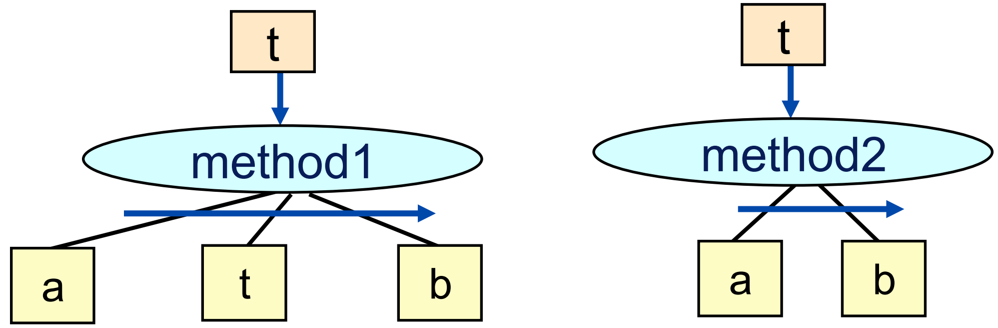
© 2009 Gerhard K. Kraetzschmar

Expressivity Relative to 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}, …, t_{cn}, o, where c1, c2, …, cn are the preconditions of o
    - Partial-ordering constraints: each t_{ci} precedes o
- There are HTN planning problems
  that cannot be translated into classical planning problems at all
- Example on the next page

© 2009 Gerhard K. Kraetzschmar

Example: Classical Planning can not represent this



- Two 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 $\{a^n b^n \mid n > 0\}$

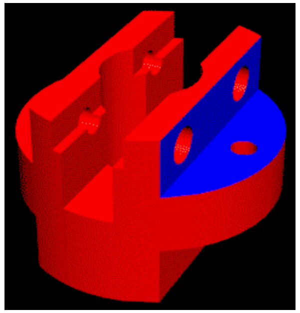
© 2009 Gerhard K. Kraetzschmar

Increasing Expressivity Further

- Knowing the current state makes it easy to do things that would be difficult otherwise
  - 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
- Example: SHOP   http://www.cs.umd.edu/projects/shop

© 2009 Gerhard K. Kraetzschmar

SHOP (Simple Hierarchical Ordered Planner)

- Domain-independent algorithm for ordered task decomposition
  - Sound and complete
- Input:
  - State: a set of ground atoms
  - Task List: a linear list of tasks
  - Domain: methods, operators, axioms
- Output: one or more plans, it can return:
  - the first plan it finds
  - all possible plans
  - a least-cost plan
  - all least-cost plans

© 2009 Gerhard K. Kraetzschmar

Example: SHOP

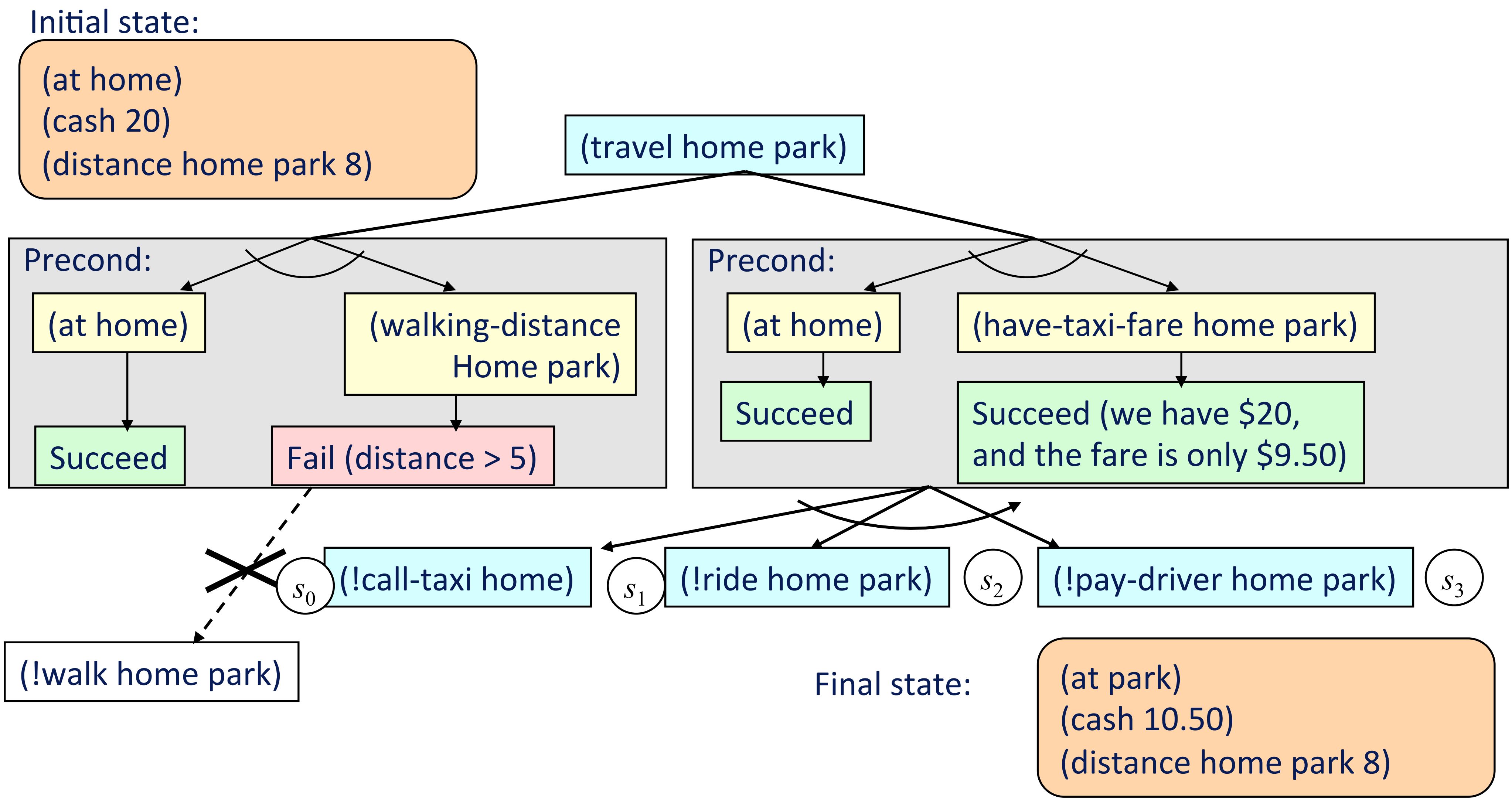
- **Initial task list:** ((travel home park))
- **Initial state:** ((at home) (cash 20) (distance home park 8))
- **Methods** (task, preconditions, subtasks):
  - (:method (travel ?x ?y)
    ((at x) (walking-distance ?x ?y)) ' ((!walk ?x ?y)) 1)
  - (:method (travel ?x ?y)
    ((at ?x) (have-taxi-fare ?x ?y))
    ' ((!call-taxi ?x) (!ride ?x ?y) (!pay-driver ?x ?y)) 1)

Optional cost; default is 1

- **Axioms:**
  - (:- (walking-dist ?x ?y) ((distance ?x ?y ?d) (eval (<= ?d 5))))
  - (:- (have-taxi-fare ?x ?y)
    ((have-cash ?c) (distance ?x ?y ?d) (eval (>= ?c (+ 1.50 ?d))))
- **Primitive operators** (task, delete list, add list)
  - (:operator (!walk ?x ?y) ((at ?x)) ((at ?y)))
  - …

© 2009 Gerhard K. Kraetzschmar

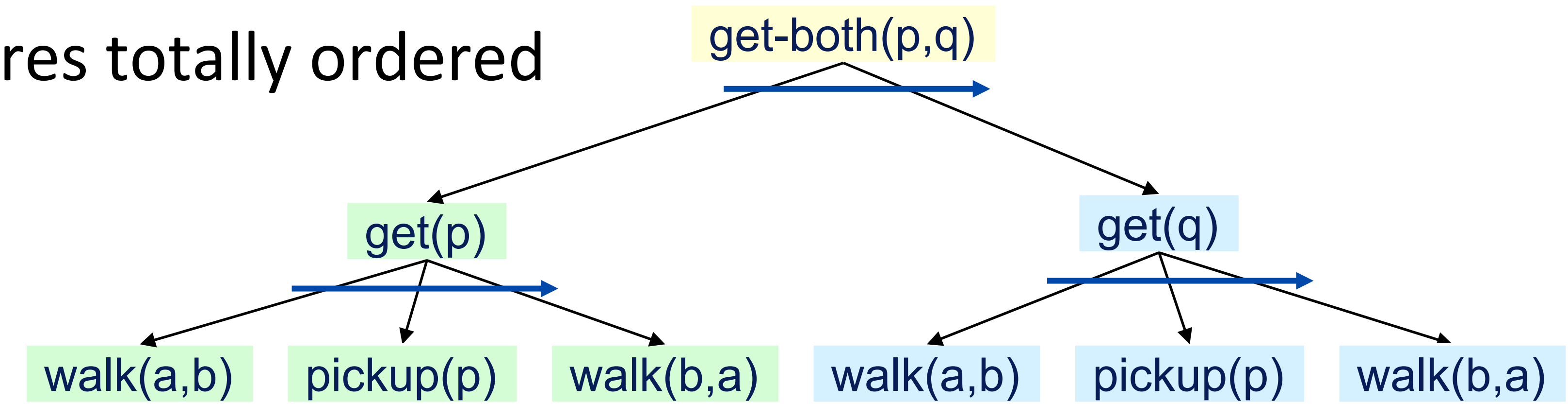
Example: SHOP (Continued)



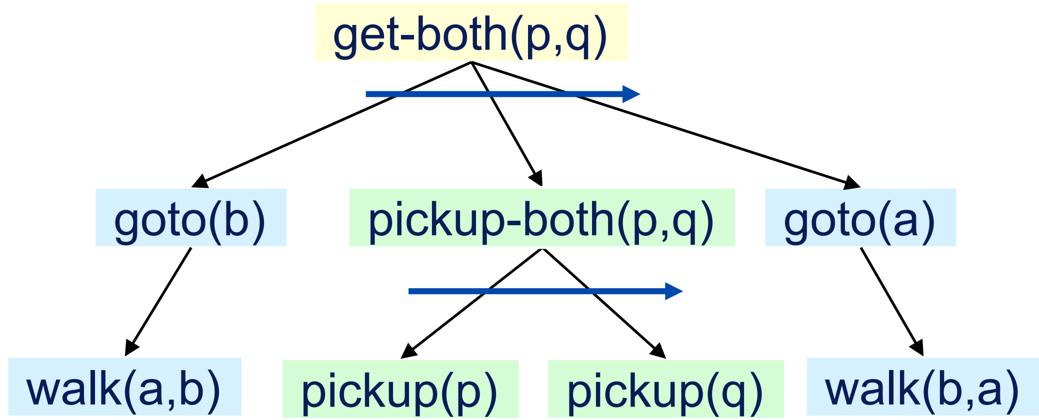
© 2009 Gerhard K. Kraetzschmar

Limitation of Ordered-Task Planning

- TFD requires totally ordered methods

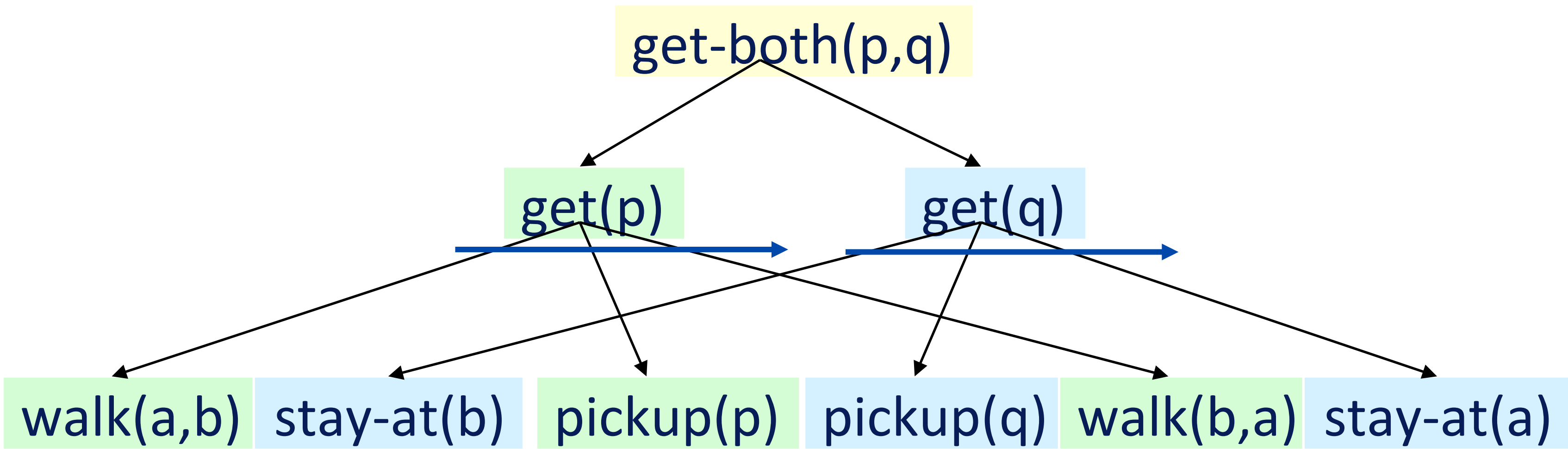


- Can't interleave subtasks of different tasks
- Sometimes this makes things awkward
  - Need to write methods that reason globally instead of locally



Generalize TFD to interleave subtasks

- Generalize methods to allow the subtasks to be partially ordered
- Consequence: plans may interleave subtasks of different tasks

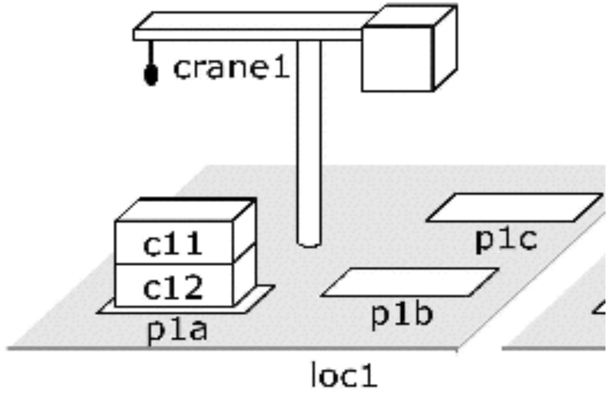


- This makes the planning algorithm more complicated

© 2009 Gerhard K. Kraetzschmar

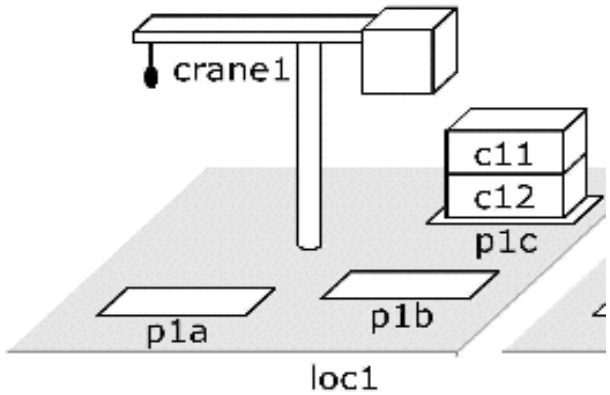
Example: DWR Partial-Order Formulation

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 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 $\mathrm{name}(a) = \sigma(t_u)$, 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 $\mathrm{name}(m) = \sigma(t_u)$, 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))

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

operator instance $\boxed{\boldsymbol{a}}$

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

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

method instance $\boxed{\boldsymbol{m}}$

$w'=\{\boxed{\mathbf{u_1,\ldots,u_k}},t_2,\ldots\}$

© 2009 Gerhard K. Kraetzschmar

Solving 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 talk about *t* and ignore *u*

$\{a_1,\ldots,a_k\}$; $w=\{\boxed{\mathbf{t_1}}, t_2, t_3\ldots\}$
operator instance $\boxed{a}$

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

else
active $\leftarrow \{(m, \sigma) \mid m$ is a ground instance of a method in M,
σ is a substitution such that $\mathrm{name}(m) = \sigma(t_u)$,
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))

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

$w'=\{\boxed{\mathbf{u_1,\ldots,u_k}}, t_2,\ldots\}$

© 2009 Gerhard K. Kraetzschmar

Solving 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))
```

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

operator instance $\boxed{a}$

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

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

method instance $\boxed{m}$

$w'=\{\boxed{\mathbf{u_1,\ldots,u_k}},t_2,\ldots\}$

© 2009 Gerhard K. Kraetzschmar

Solving Partial-order STNs

PFD(s, w, O, M)
    if $w = \emptyset$ the
    nondetermin
    if t_u is a pri
        active $\leftarrow$

        if *active*
        nondeter
        $\pi \leftarrow$ PFD
        if $\pi =$ fa
        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))

$\delta(w, u, m, \sigma)$ has a complicated definition in the book. Here's what it means:

- We non-deterministically selected t_1 as the task to do first
- Must 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

$t_2, t_3 \ldots\}$

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

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

method instance $\boxed{\boldsymbol{m}}$

$w' = \{\boxed{\mathbf{u_1, \ldots, u_k}}, t_2, \ldots\}$

© 2009 Gerhard K. Kraetzschmar

STN Summary

- PFD is sound and complete
- STN – simplified version of HTN
  - TFD – Total-order Forward Decomposition (used in SHOP)
    - Input: tasks are totally ordered
    - Output: totally ordered plan
  - PFD – Partial-order Forward Decomposition (SHOP2)
    - Input: tasks are partially ordered
    - Output: totally ordered plan
- SHOP2:
  - Won one of the top four awards in the AIPS-2002 Planning Competition
  - Freeware, open source
  - Implementation available at http://www.cs.umd.edu/projects/shop

© 2009 Gerhard K. Kraetzschmar

STN v HTN

- HTN – generalization of STN
  - More freedom about how to construct the task networks.
  - Can use other decomposition procedures not just forward-decomposition.
  - Like Partial-Order Planning combined with STN
    - Input: Partial-order tasks
    - Output: The resulting plan is partially ordered
  - Plans can be totally ordered or partially ordered
  - Can have constraints associated with tasks and methods
  - Things that must be true before a state, in between two given states, or after a state (replaces STN preconditions)
  - Some algorithms use causal links and threats like those in PSP

© 2009 Gerhard K. Kraetzschmar

TLPlan's Expressivity Compared with SHOP and SHOP2

- Equivalent expressive power
- Both know the current state at each step of the planning process, and use this to prune operators
- Both can call external subroutines
  - SHOP uses "eval" to call LISP functions
  - In TLPlan, a function symbol can correspond to a computed function
- Main difference
  - in SHOP and SHOP2, the methods talk about what *can* be done
    - SHOP and SHOP2 don't do anything unless a method says to do it
  - TLPlan's control rules talk about what *cannot* be done
    - TLPlan will try everything that the control rules don't prohibit
- Which approach is more convenient depends on the problem domain

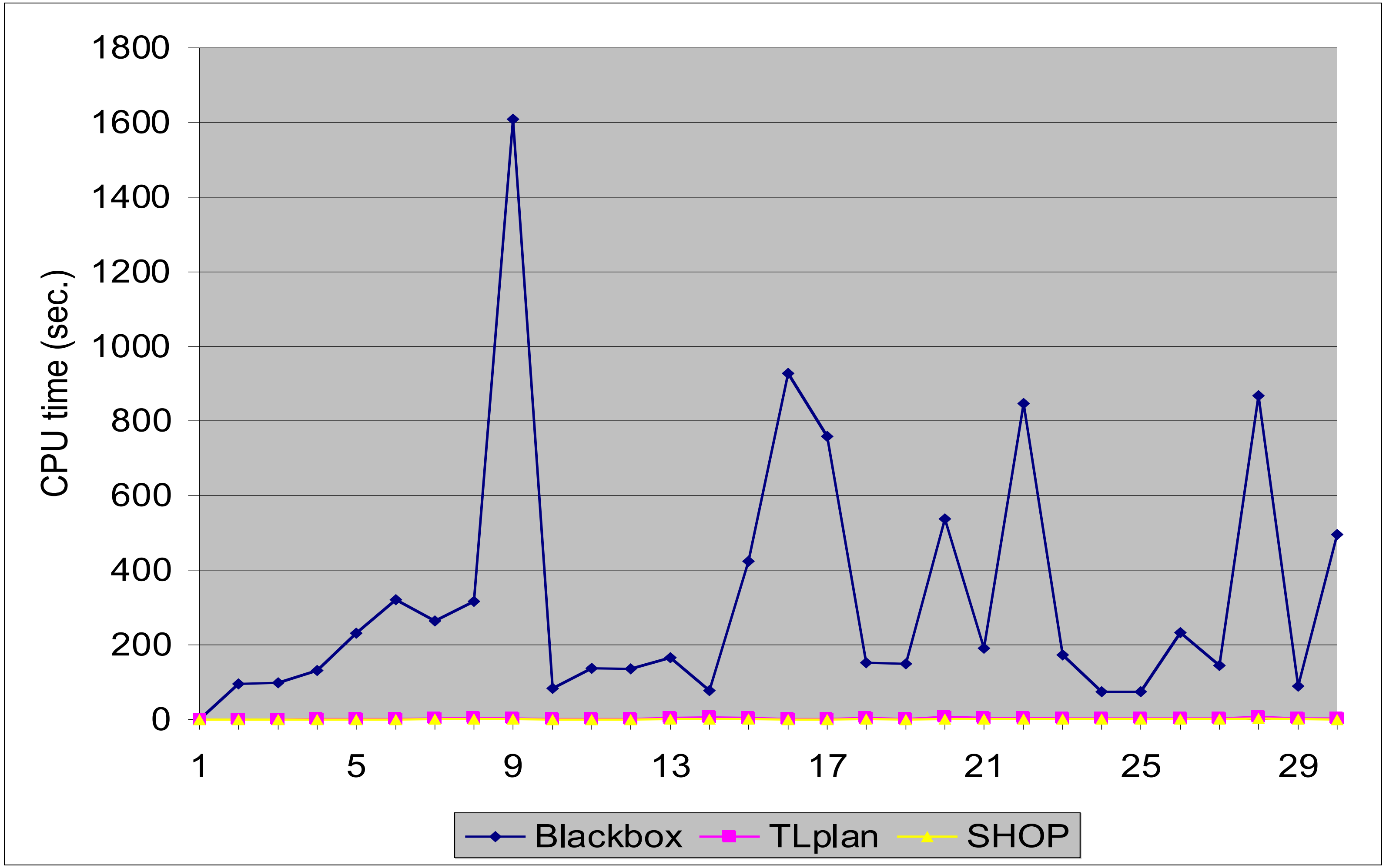
© 2009 Gerhard K. Kraetzschmar

Experimental Comparison

- Several years ago, we did a comparison of SHOP, TLPlan, and Blackbox
  - Blackbox is a domain-independent planner that uses a combination of Graphplan and satisfiability
  - One of the two fastest planners in the 1998 planning competition
- Test domain: the logistics domain
  - A classical planning problem
    - Much simpler than real logistics planning
  - Scenario: use trucks and airplanes to deliver packages
  - Like a simplified version of the DWR domain in which containers don't get stacked on each other
- Test conditions
  - SHOP and TLPlan on a 167-MHz Sun Ultra with 64 MB of RAM
  - We couldn't run Blackbox on our machine
  - Published results: Blackbox on a faster machine with 8 GB of RAM

© 2009 Gerhard K. Kraetzschmar

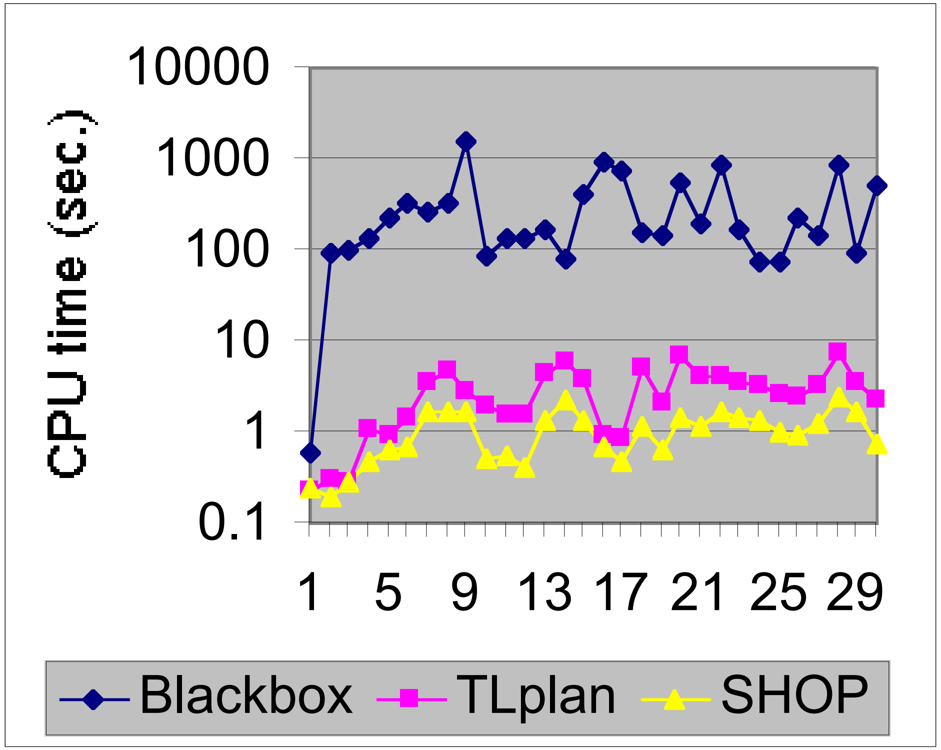
Logistics Domain Results



© 2009 Gerhard K. Kraetzschmar

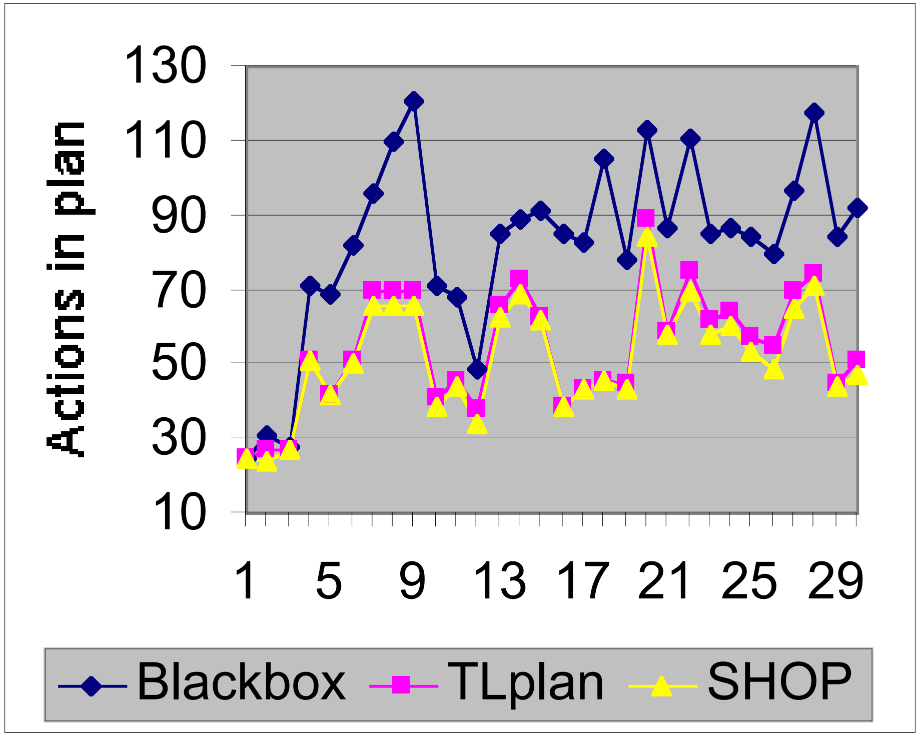
Logistics Domain Results (continued)

- Same graph as before, but on a logarithmic scale



Average CPU time	Blackbox	TLPlan	SHOP
	327.1	2.9	1.1

- Number of actions in the plans



Average no. of actions	Blackbox	TLPlan	SHOP
	82.5	54.5	51.9

© 2009 Gerhard K. Kraetzschmar

Summary: Results

- TLPlan and SHOP took similar amounts of time
  - In this experiment, SHOP was slightly faster, but in others TLPlan may be faster
- Blackbox took about 1000 times as much time
  and needed about 100 times as much memory
- Reasons why:
  - SHOP's input included domain-specific methods & axioms
  - TLPlan's input included domain-specific control rules
    - This enabled them to find near-optimal solutions in low-order polynomial time and space
  - Blackbox is a fully automated planner
    - No domain-specific knowledge
    - trial-and-error search, exponential time and space

© 2009 Gerhard K. Kraetzschmar

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)

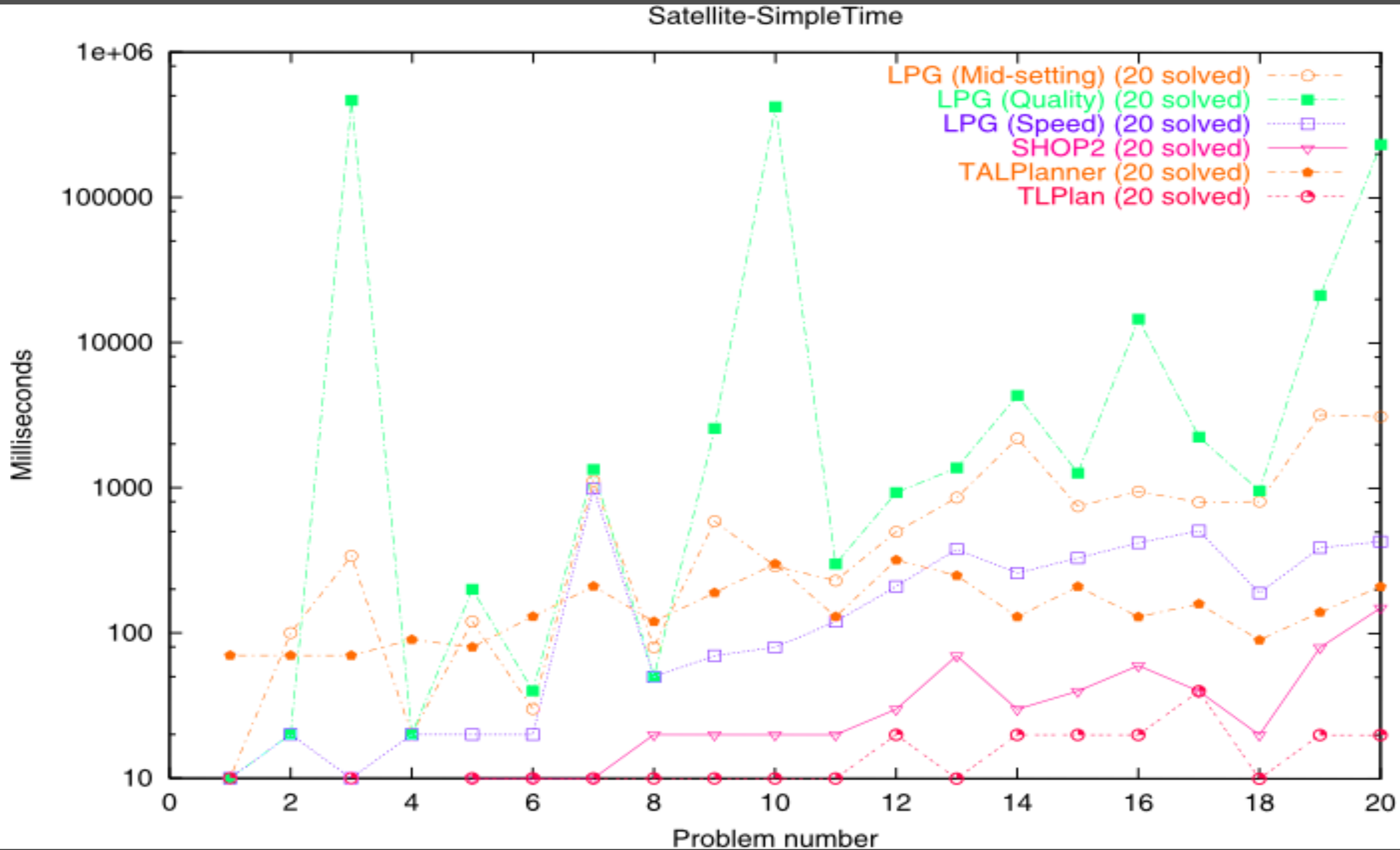
© 2009 Gerhard K. Kraetzschmar

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. Results are shown 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

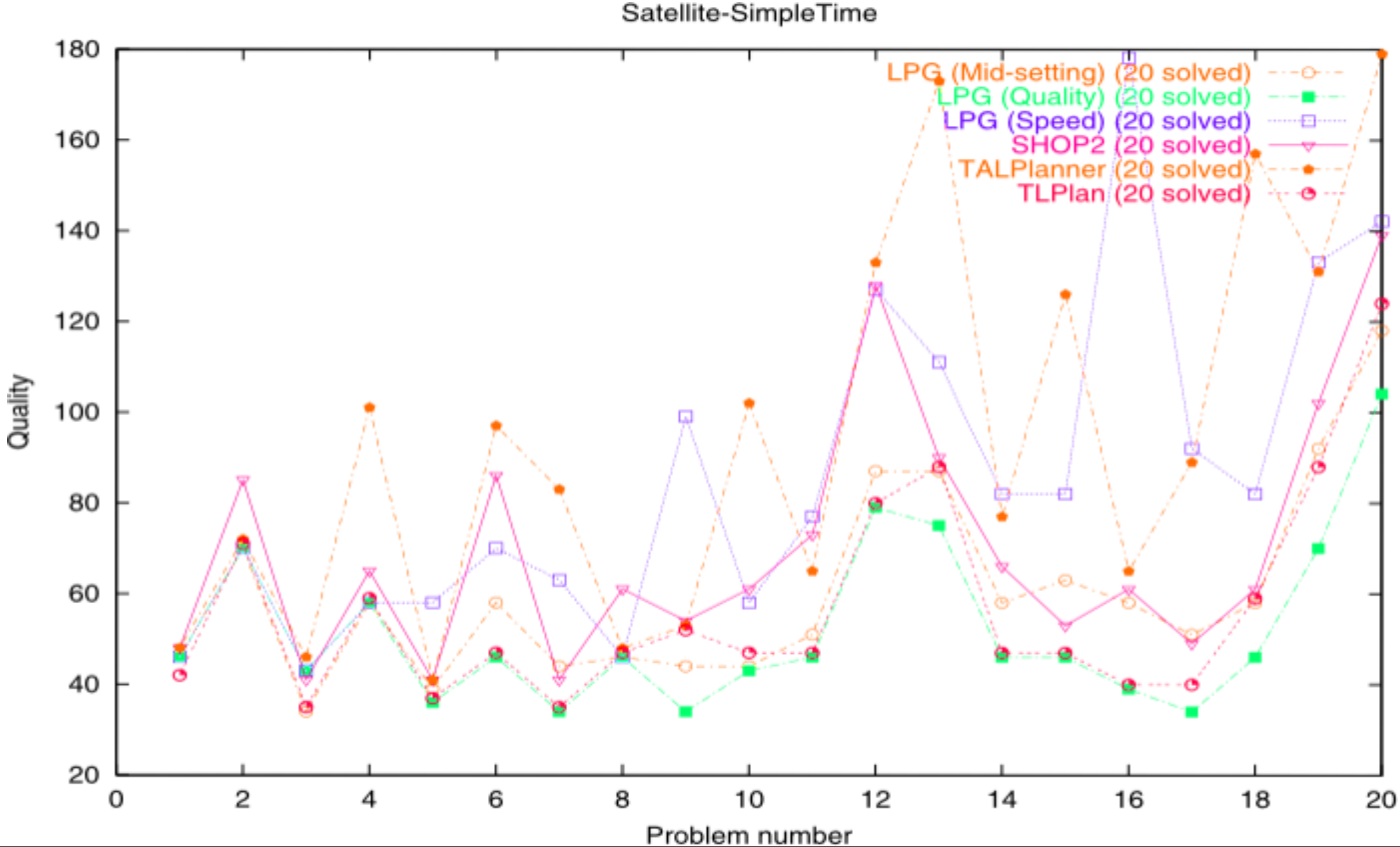
© 2009 Gerhard K. Kraetzschmar

Satellite Problem Domain: Simple Time: Runtime



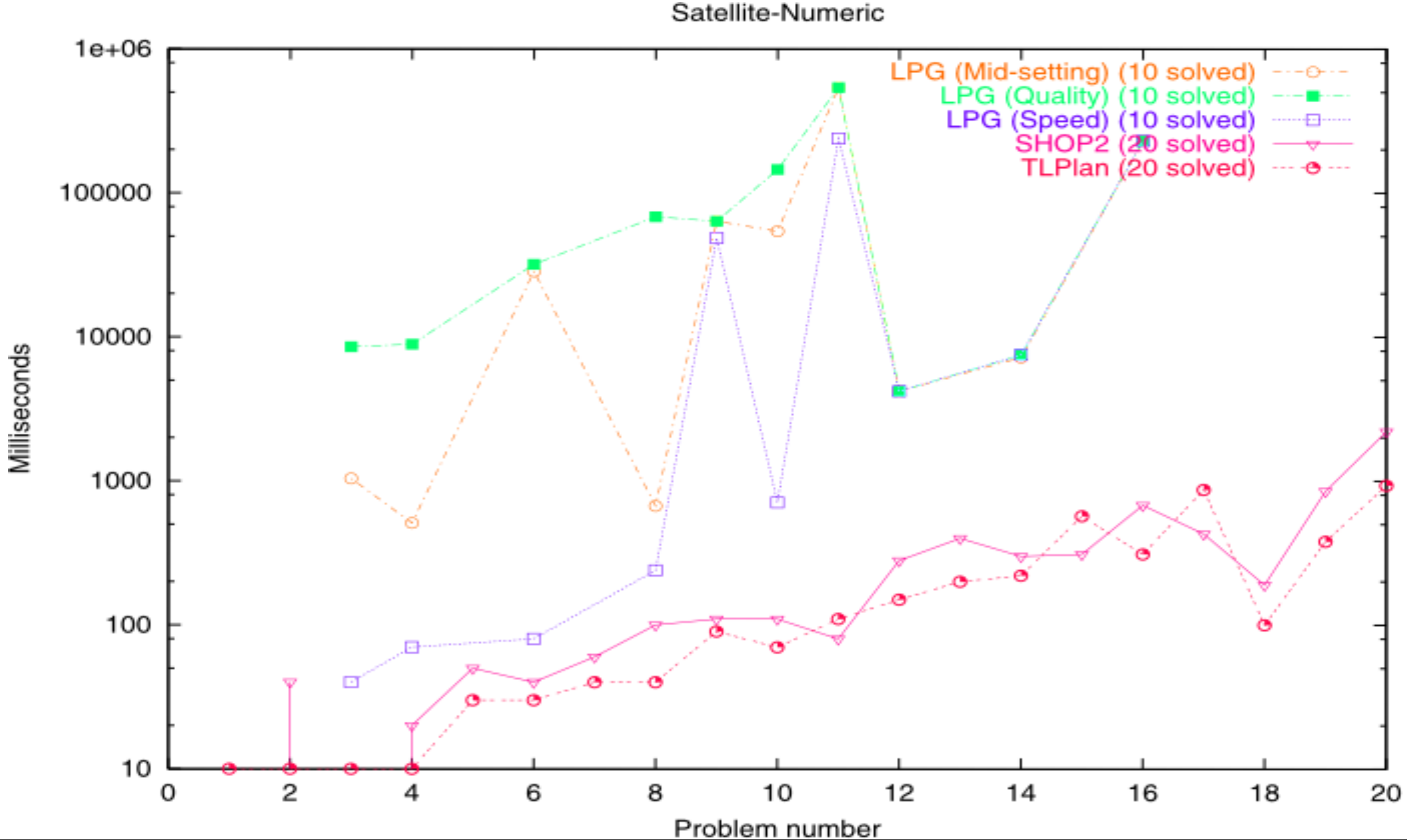
© 2009 Gerhard K. Kraetzschmar

Satellite Problem Domain: Simple Time: Plan Quality



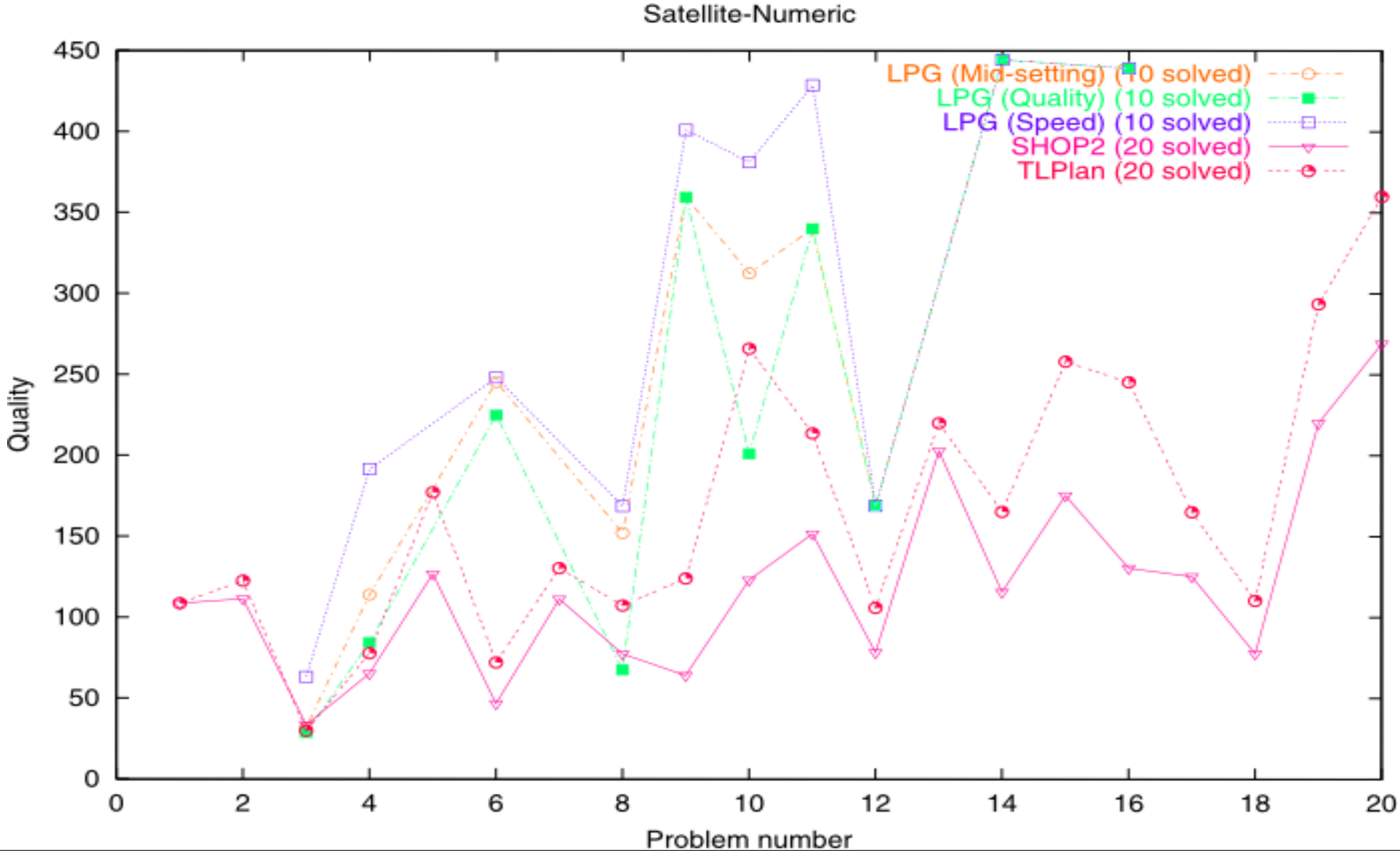
© 2009 Gerhard K. Kraetzschmar

Satellite Problem Domain: Numeric: Runtime



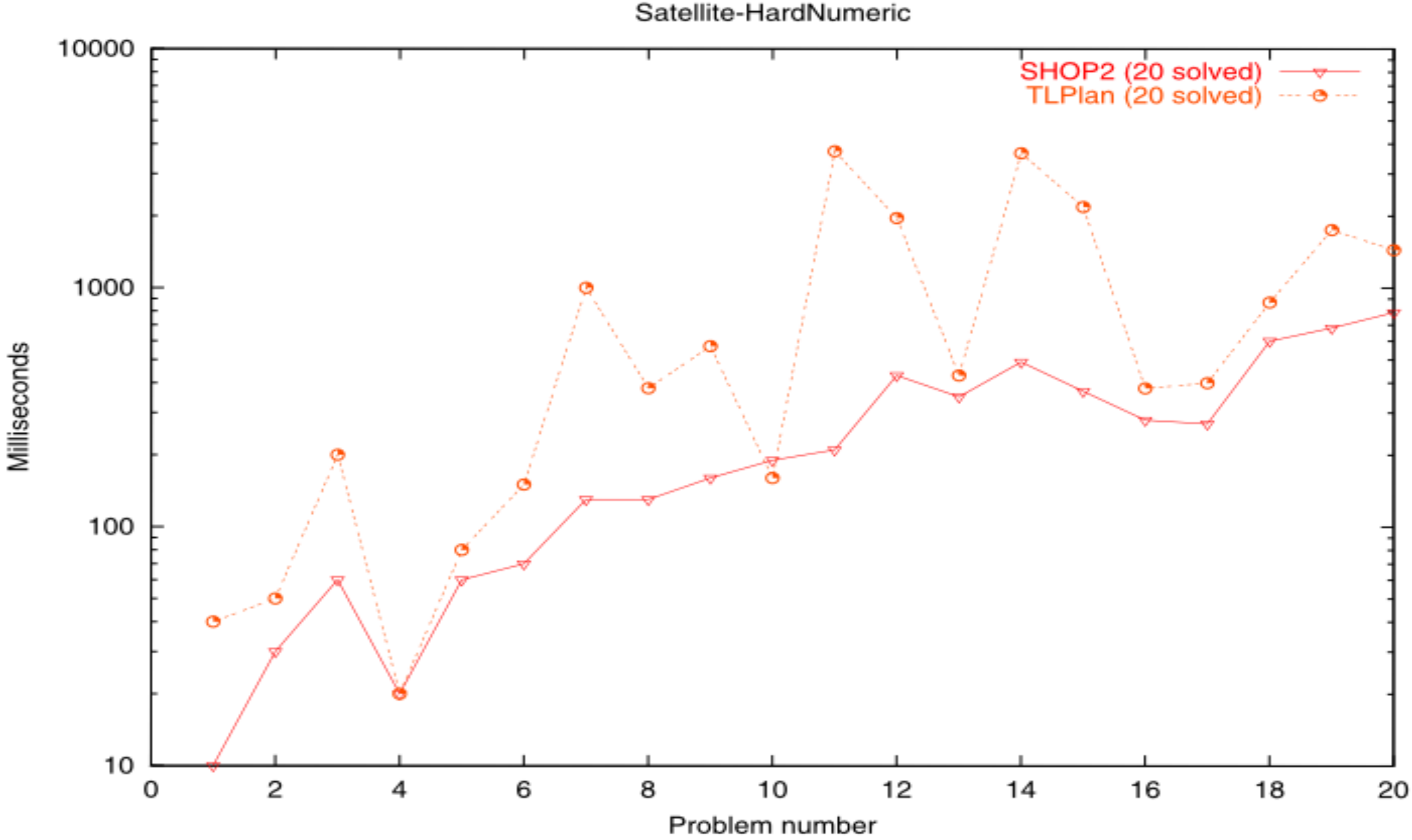
© 2009 Gerhard K. Kraetzschmar

Satellite Problem Domain: Numeric: Plan Quality



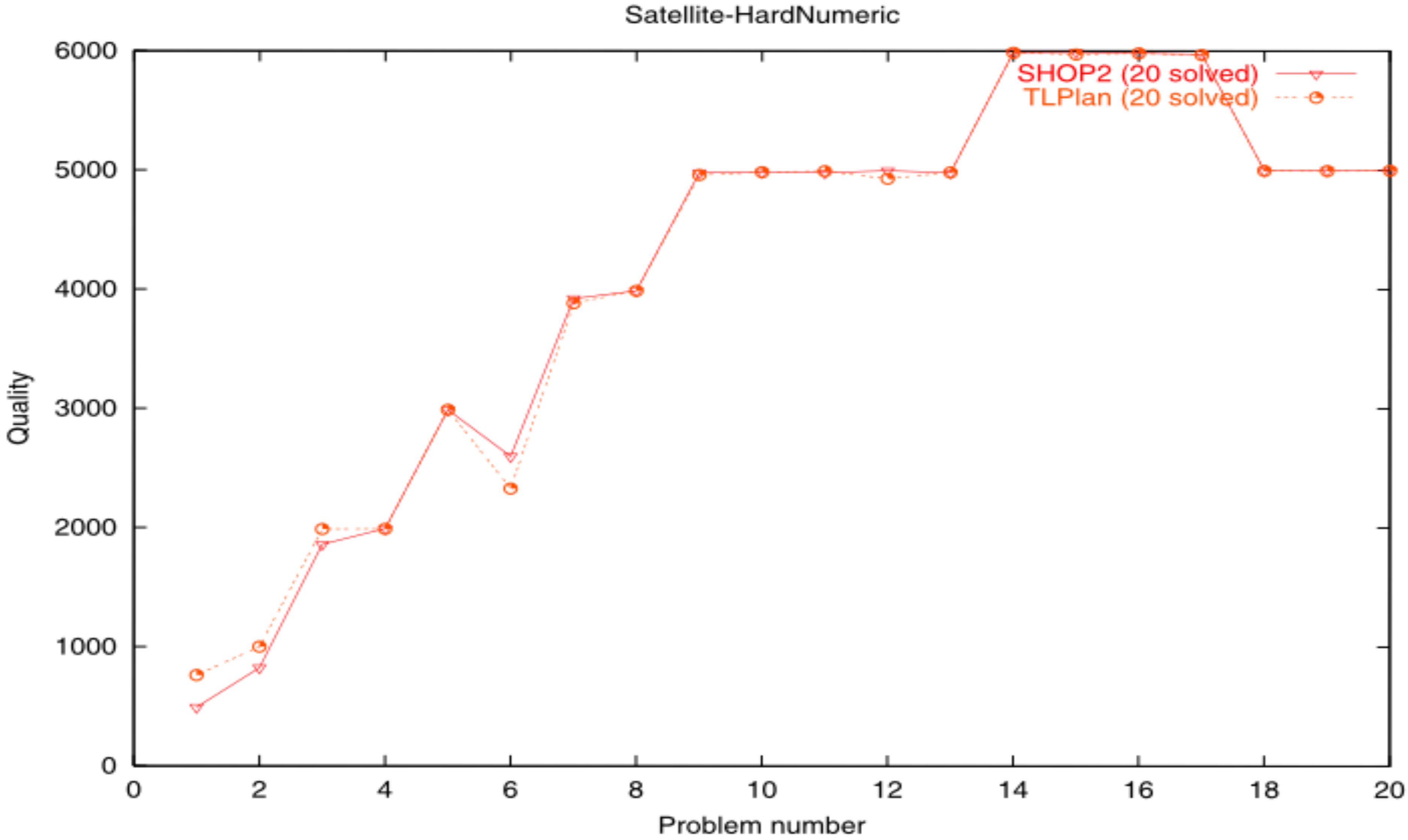
© 2009 Gerhard K. Kraetzschmar

Satellite Problem Domain: Hard Numeric: Runtime



© 2009 Gerhard K. Kraetzschmar

Satellite Problem Domain: Hard Numeric: Plan Quality



© 2009 Gerhard K. Kraetzschmar