Search Complexities for HTN Planning

Ron Alford

UMCP

December 13th, 2013

Hierarchical Task Network (HTN) planning

HTN planning is the problem of decomposing an initial task to accomplish into a sequence of executable steps:

- Often used to solve classical planning problems faster
- More expressive than classical planning
  - Can express undecidable problems

But...

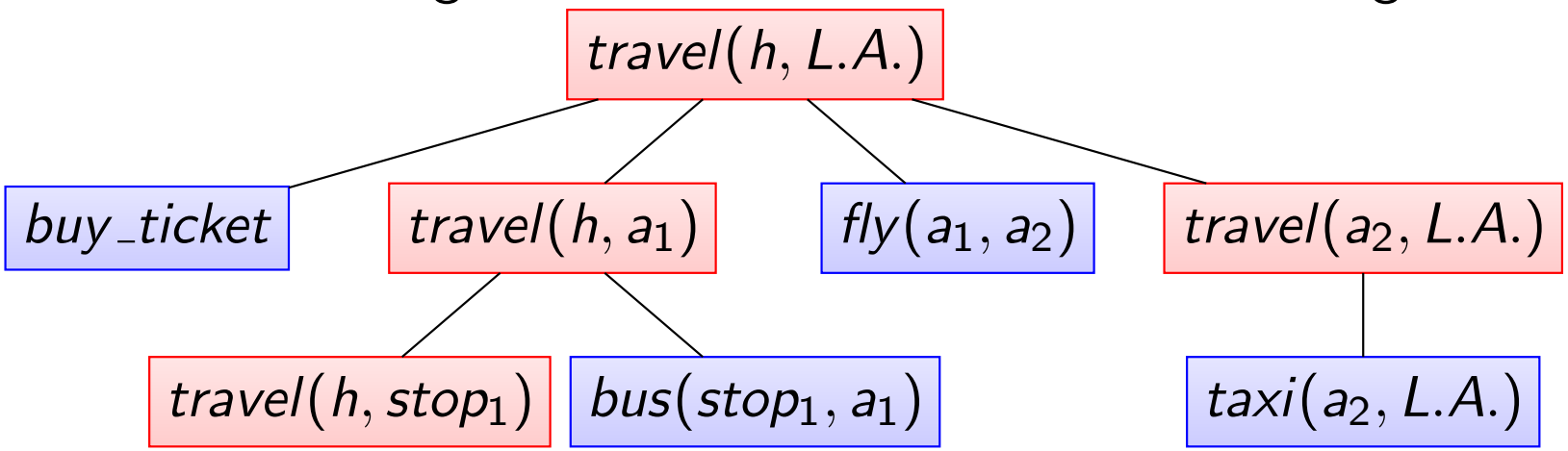
- There are many ways to solve HTN problems
  - Two main algorithms in practice, two new algorithms in thesis
  - What problems are these algorithms guaranteed to solve?
  - How efficient are they?
- Search space is huge
  - Most problems in PSPACE through EXPSPACE
  - What domain independent heuristics can we design for HTN planning?
  - Can these heuristics be effective?

Outline

HTN Planning (overview)

The purpose of HTN planning is to complete a *task*. Tasks are either

- *Primitive*, which corresponds to some concrete action we know how to perform
  - E.g: $walk(room, hall)$, or $drink(coffee)$
- *Non-primitive*, which is an abstract task. E.g. $travel(home, L.A.)$
- Must recursively decompose non-primitive tasks until we get primitive tasks we know how to execute directly
- Given a set of *methods*, which are recipes on how to accomplish abstract tasks. E.g., to travel from *home* to *L.A.*, we might



Planning: Propositional and FOL

- There are two common ways to define HTN classical planning:
  - Over a propositions, and propositional sentences
  - Over a function-free fragment of first-order logic
- My work and this talk are concerned with propositional planning
  - Applies directly to non-propositional planning
  - Complexity results usually go up by an exponential amount

HTN Planning: $D = (\mathbf{O}, M)$, $P = (D, \mathbf{s}_0, tn_0)$

Every HTN problem P contains an *initial state* s_0, which is just a collection of literals, e.g., $\{thirsty, tired\}$.

Actions in our domain work over these states.

Actions have:

- A name (the task)
- A precondition (logical formula over the state)
  - Condition must hold for the action to be executed
- An effect on the state
  - A set of literals to add
  - A set of literals to delete
- Denote execution by a partial function γ:
  - $\gamma(\{thirsty, tired\}, drink) = \{tired\}$

Operator: *sleep*

pre $= tired \wedge \neg thirsty$

eff $= \{\neg tired\}$

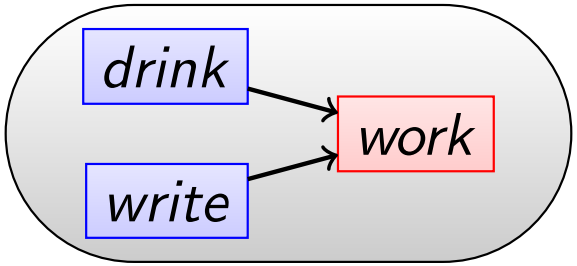
Operator: *drink*

pre $= true$

eff $= \{\neg thirsty\}$

HTN Planning: $D = (O, \mathbf{M})$, $P = (D, s_0, \mathbf{tn}_0)$

- An initial *task network*, tn_0, which is a labeled DAG of *tasks* to accomplish:
  - Edges represent ordering constraints
  - Nodes are labeled with *tasks*, which are either:
    - Primitive tasks (i.e., action names from O: $\{drink, write\}$)
    - Non-primitive tasks (say, $\{work\}$)

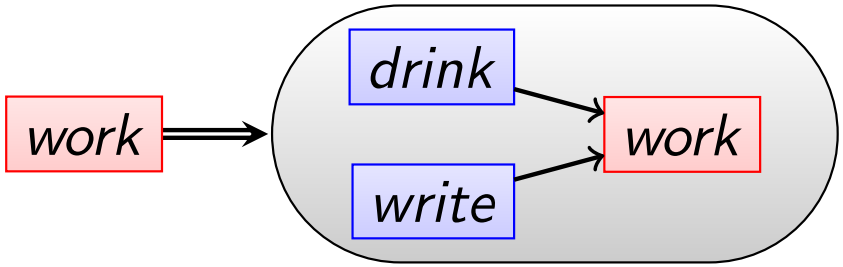


- M, a set of methods to *decompose* non-primitive tasks

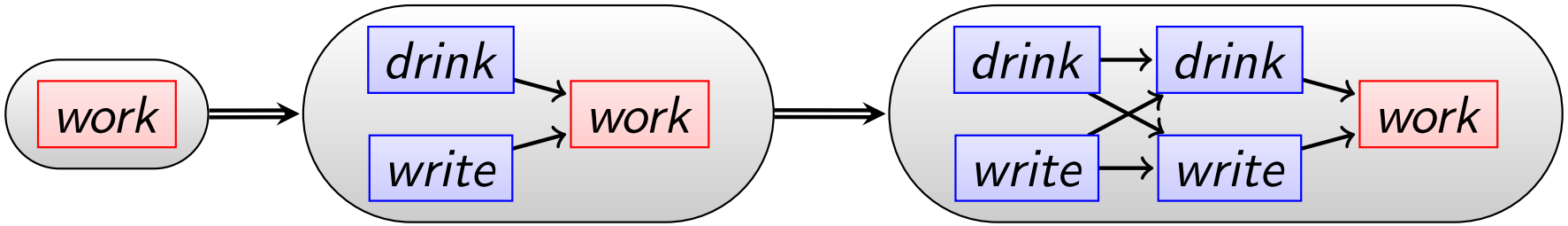
Methods and Decomposition

- A method (t, tn) is a non-primitive task t paired with a network tn

Method:

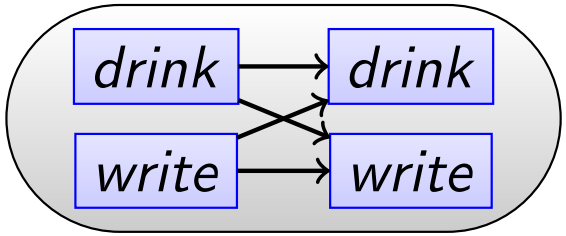


- We *decompose* a task network by replacing a node in the network with a corresponding method's network.



Primitive task networks:

- Two states:
  - $thirsty, \neg thirsty$
- Two actions:
  - $\forall_s \gamma(s, write) = thirsty$
  - $\forall_s \gamma(s, drink) = \neg thirsty$



- All names are actions $\Rightarrow$ network is *primitive*
- It is *executable* in a state if there exists a consistent total order (sequence) which is executable.
  - Four possible orderings
  - All executable in both *thirsty* and $\neg thirsty$
  - Could end in either state.
- Objective: Decompose initial network to a primitive task network that is executable in s_0.

Classical (STRIPS) planning

Differences between HTN and Classical Planning:

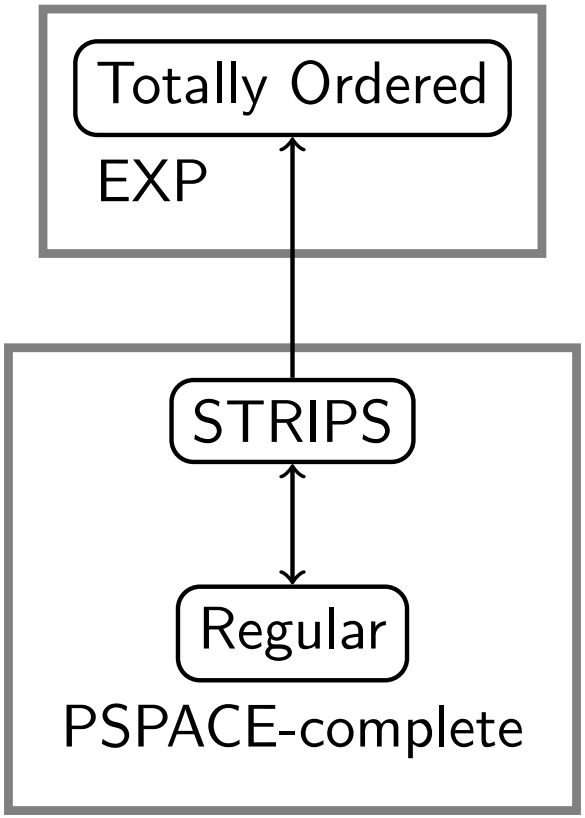
- $P = (O, s_0, G)$
- No methods and no initial task network.
- Objective: Find a plan (any executable sequence) that take us to a goal state.
  - No restriction (other than executability) on when planner can insert an action
- PSPACE-complete for propositional problems

Outline

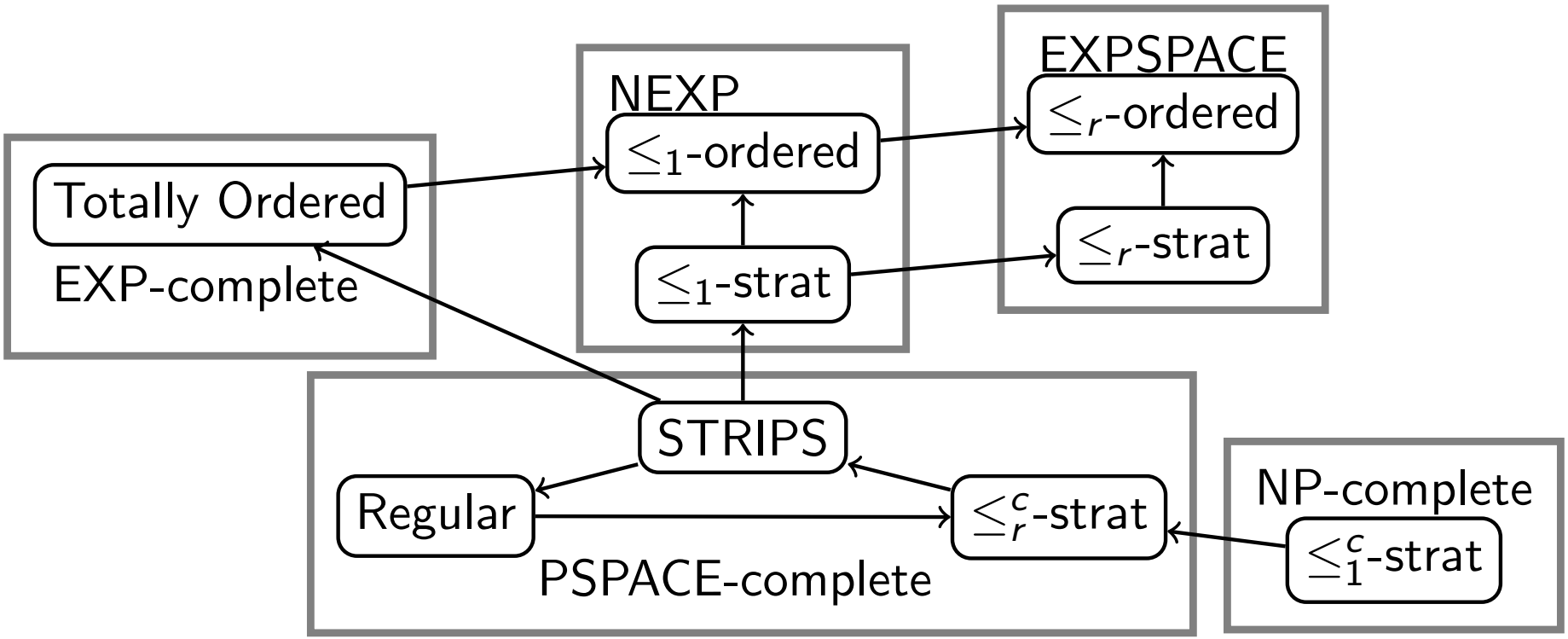
HTN Algorithms and Complexity

- Most HTN planners are sound and complete
  - HTN planning is undecidable
  - There are problems on which any set of planners will never terminate
  - In practice, we observe differences in how well they work. Why?
- General strategy:
  - Identify what problems each HTN algorithm halts on
  - Bound the run-time of the planner on those problems
- Given these problem classes, we can propose new HTN algorithms
  - Compare them to current ones on more than just an experimental basis
  - Two new algorithms in thesis

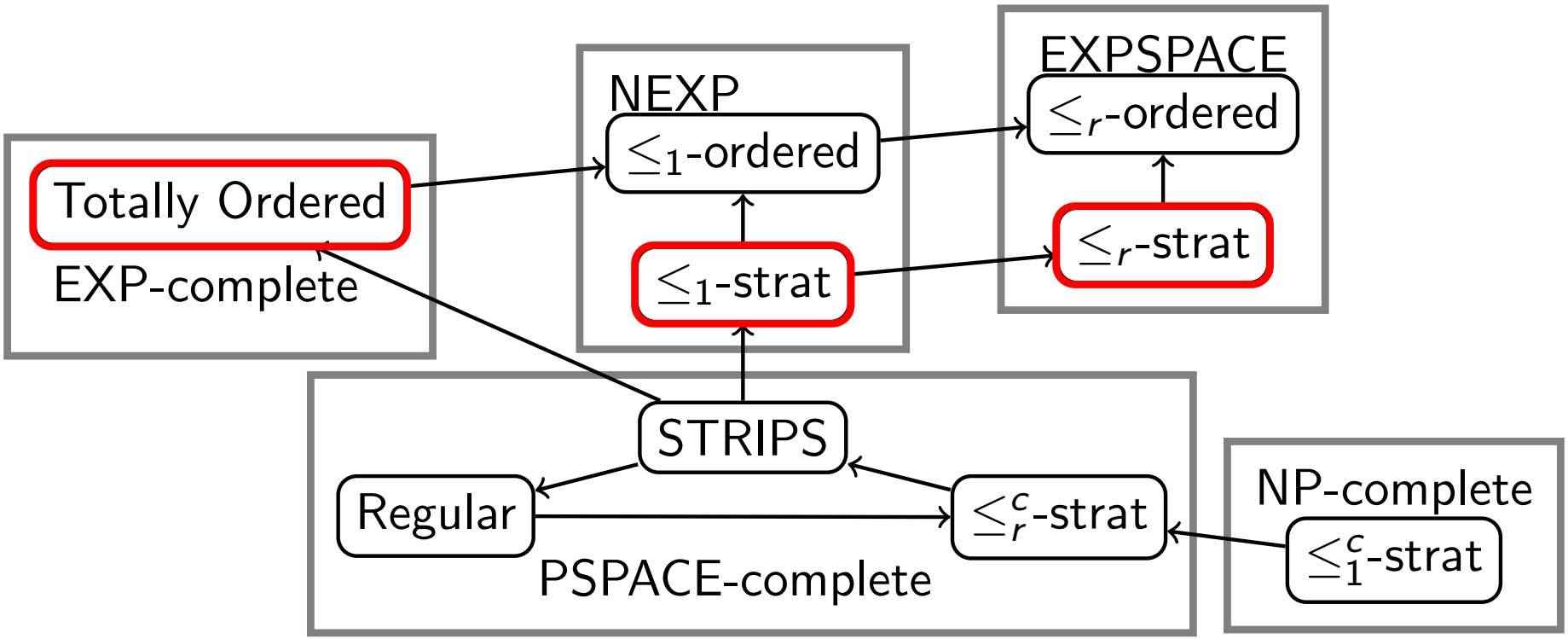
Before thesis



In the thesis

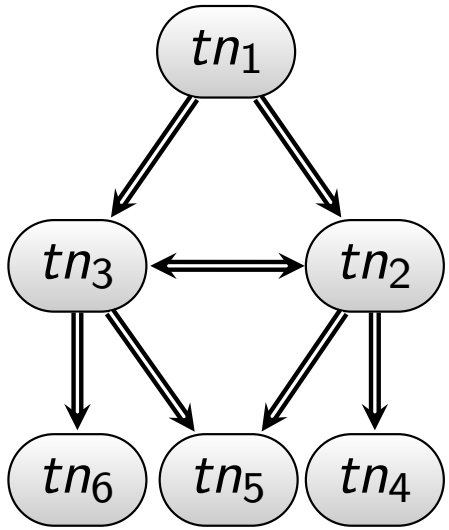


In the thesis



Decomposition as a graph

- UMCP (Erol 94), Landmark-aware HTN planner (Elkawkagy 2012) directly use the definition of decomposition to find solutions
- Starting at the initial task network, the choices form a graph:
  - Each vertex is a task network reachable via decomposition
  - Each edge is a decomposition
- Call this the *decomposition space* of a problem
- Plan by searching graph (not always finite)



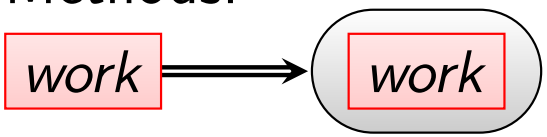
Decomposition-decided problems: $\leq_1$-stratification

- A domain is $\leq_1$-*stratifiable* if we can stratify its task names such that for every method $(t, tn) \in M$:
  - If tn has only one task, then tn's task is on an equal or lower stratum than t (trivially recursive decomposition)
  - Otherwise, every task in tn is on a lower stratum than t (acyclic decomposition)

Constraints:

- $strat(work) \leq strat(work)$

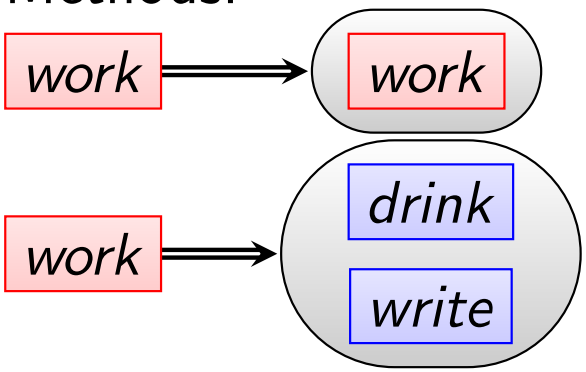
Methods:



Decomposition-decided problems: $\leq_1$-stratification

- A domain is $\leq_1$-*stratifiable* if we can stratify its task names such that for every method $(t, tn) \in M$:
  - If tn has only one task, then tn's task is on an equal or lower stratum than t (trivially recursive decomposition)
  - Otherwise, every task in tn is on a lower stratum than t (acyclic decomposition)

Methods:



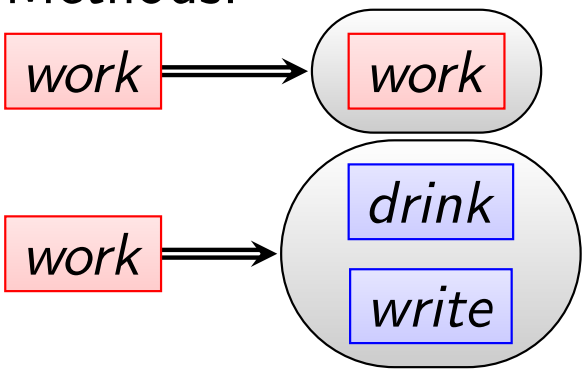
Constraints:

- $strat(work) \leq strat(work)$
- $strat(drink) < strat(work)$
- $strat(write) < strat(work)$

Decomposition-decided problems: $\leq_1$-stratification

- A domain is $\leq_1$-*stratifiable* if we can stratify its task names such that for every method $(t, tn) \in M$:
  - If tn has only one task, then tn's task is on an equal or lower stratum than t (trivially recursive decomposition)
  - Otherwise, every task in tn is on a lower stratum than t (acyclic decomposition)

Methods:



Constraints:

- $strat(work) \leq strat(work)$
- $strat(drink) < strat(work)$
- $strat(write) < strat(work)$

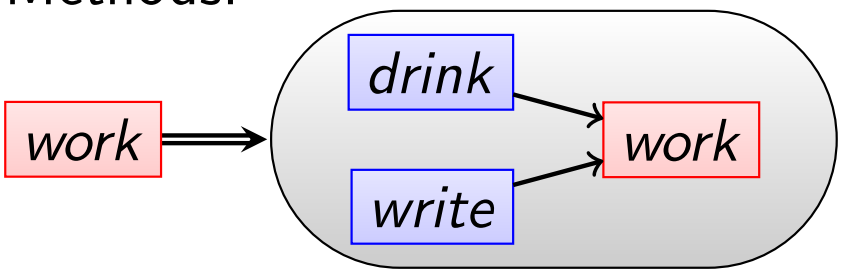
Here's one stratification:

$strat(work) =$	2
$strat(drink) =$	1
$strat(write) =$	1

Decomposition-decided problems: $\leq_1$-stratification

- A domain is $\leq_1$-*stratifiable* if we can stratify its task names such that for every method $(t, tn) \in M$:
  - If tn has only one task, then tn's task is on an equal or lower stratum than t (trivially recursive decomposition)
  - Otherwise, every task in tn is on a lower stratum than t (acyclic decomposition)

Methods:



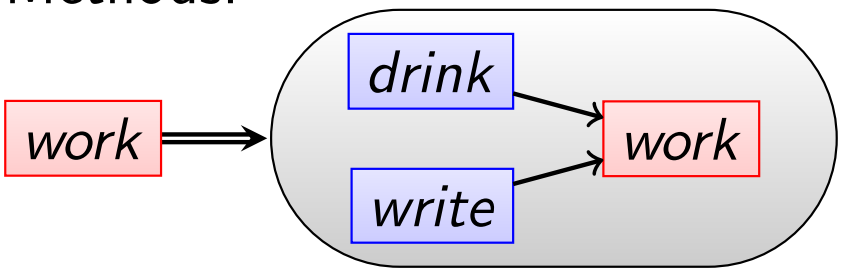
Constraints:

- $strat(work) < strat(work)$
- $strat(drink) < strat(work)$
- $strat(write) < strat(work)$

Decomposition-decided problems: $\leq_1$-stratification

- A domain is $\leq_1$-*stratifiable* if we can stratify its task names such that for every method $(t, tn) \in M$:
  - If tn has only one task, then tn's task is on an equal or lower stratum than t (trivially recursive decomposition)
  - Otherwise, every task in tn is on a lower stratum than t (acyclic decomposition)

Methods:

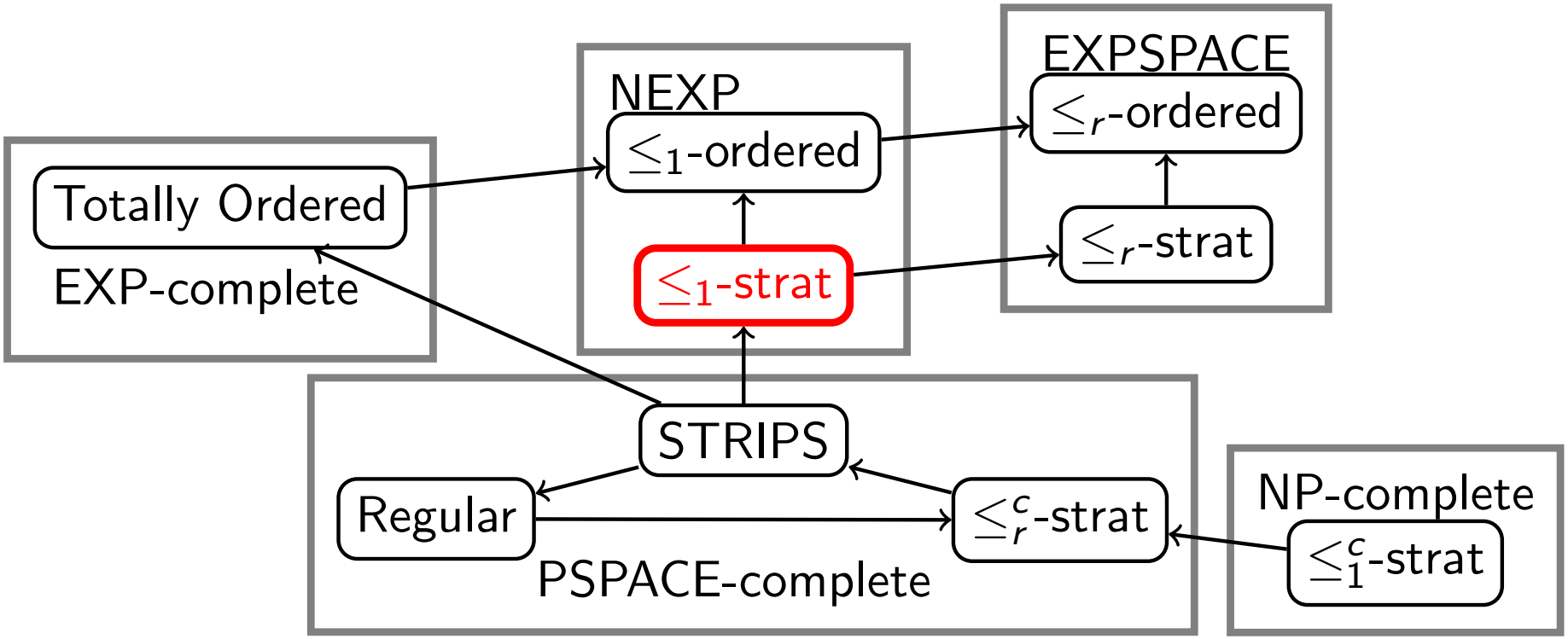


Constraints:

- ~~$strat(work) < strat(work)$~~
- $strat(drink) < strat(work)$
- $strat(write) < strat(work)$

No stratification!

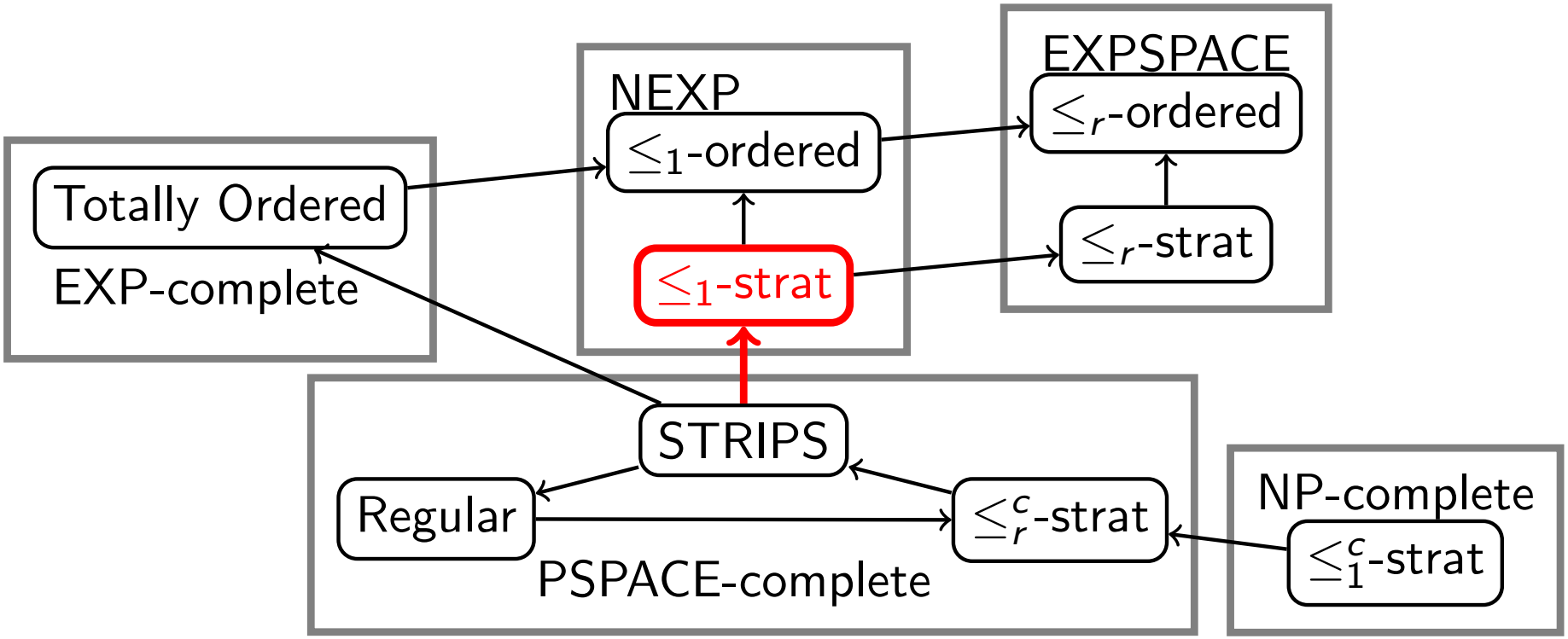
Decomposition spaces and $\leq_1$-stratifiable problems



From the thesis (but not in this talk):

- Theorem: Decomposition decides $\leq_1$-stratifiable problems and nothing else
- Plan existence is in NEXP for $\leq_1$-stratifiable problems
  - Can preprocess $\leq_1$-stratifiable problems into acyclic problems.

Decomposition spaces and $\leq_1$-stratifiable problems



From the thesis (but not in this talk):

- Theorem: Decomposition decides $\leq_1$-stratifiable problems and nothing else
- Plan existence is in NEXP for $\leq_1$-stratifiable problems
  - Can preprocess $\leq_1$-stratifiable problems into acyclic problems.
- But how hard is $\leq_1$-stratifiable planning?

$\leq_1$-stratifiable HTN planning is PSPACE-hard

Propositional classical planning (Propositional STRIPS)

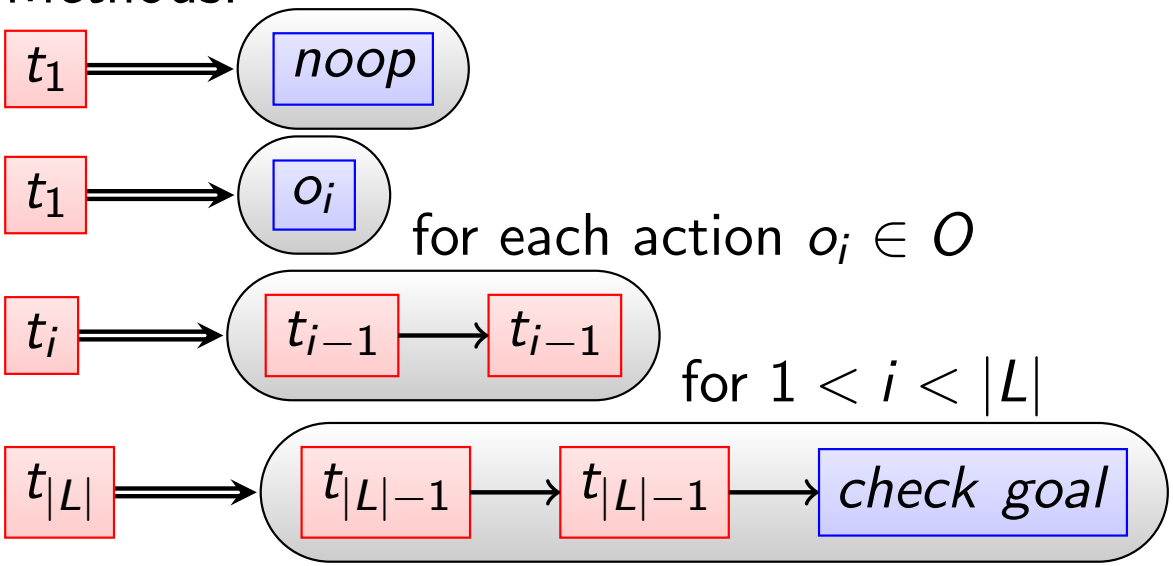
- Similar to HTN planning, but no methods
- Use any sequence of operators to obtain a given goal
- Plan-existence is PSPACE-complete (Erol et. al.)
- Shortest solution is always $< 2^{|L|}$, where L is the set of propositions in the language.

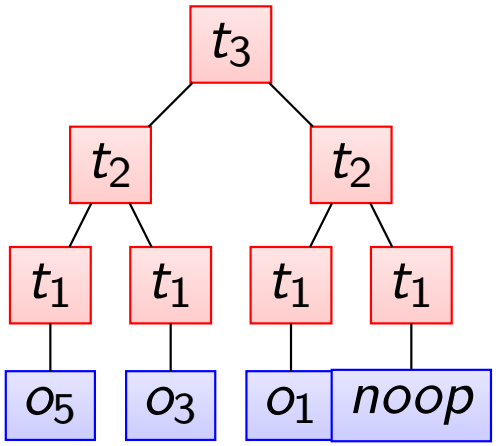
$\leq_1$-stratifiable HTN planning is PSPACE-hard

Encoding STRIPS in $\leq_1$-strat HTNs

Given a propositional STRIPS problem $(L, O, s, goal)$, create an HTN domain D with O, L, the tasks $t_1, \ldots, t_{|L|}$, and the following methods:

Methods:





Translated HTN Problem: $\left(D, s_0, t_{|L|}\right)$

Progression spaces for HTN planning

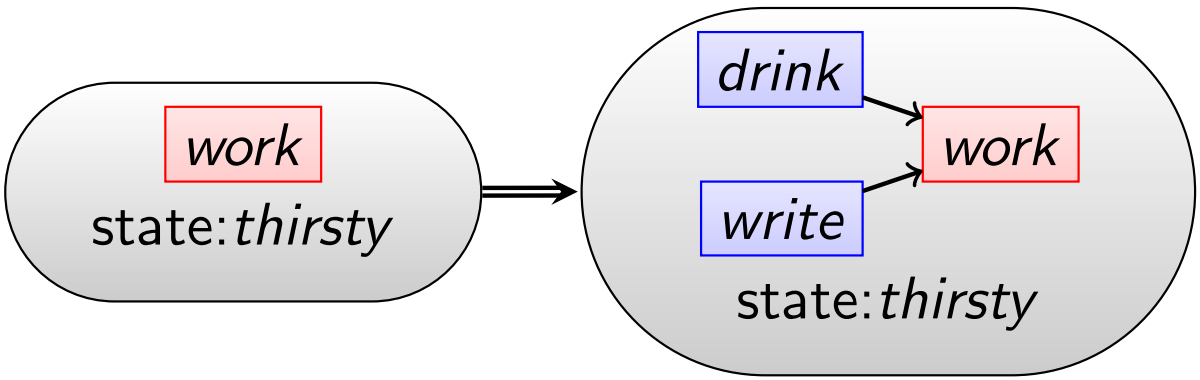
A number of HTN planners (SHOP, SHOP2, etc) plan for tasks in the order that they are to be executed. This is called *progression*.
For an HTN problem $P = (D, s_0, tn_0)$, we *progress* P by picking a task t in tn_0 with no predecessors. Then:

- If t is non-primitive:
  - Decompose t in tn_0, let tn' be the result
  - Then (D, s_0, tn') is a progression of P.
- If t is primitive ($t \in O$) and executable, then:
  - Apply t: $\gamma(s_0, t) = s_1$
  - Remove it from tn_0: $tn' = tn_0 \setminus \{t\}$
  - Then (D, s_1, tn') is a progression of P.

Progression spaces for HTN planning

A number of HTN planners (SHOP, SHOP2, etc) plan for tasks in the order that they are to be executed. This is called *progression*.
For an HTN problem $P = (D, s_0, tn_0)$, we *progress* P by picking a task t in tn_0 with no predecessors. Then:

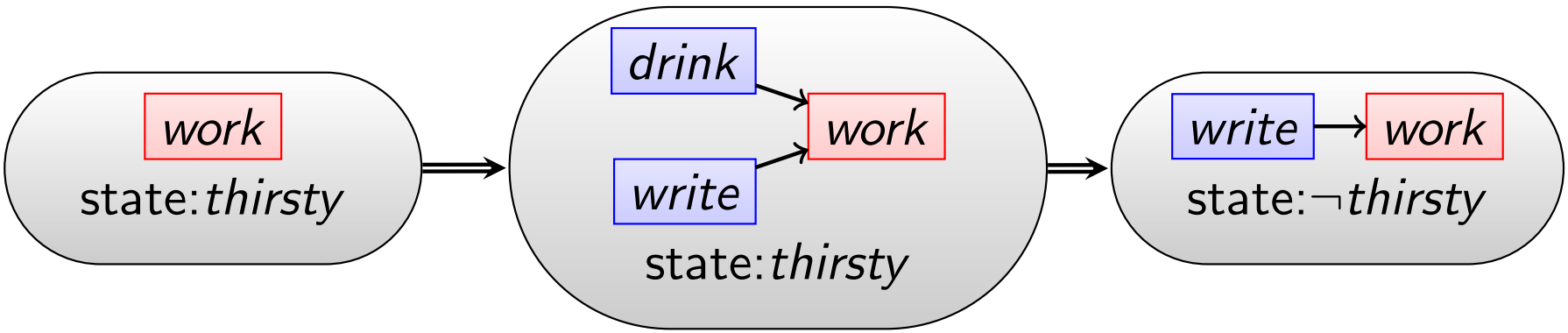
- If t is non-primitive:
  - Decompose t in tn_0, let tn' be the result
  - Then (D, s_0, tn') is a progression of P.
- If t is primitive ($t \in O$) and executable, then:
  - Apply t: $\gamma(s_0, t) = s_1$
  - Remove it from tn_0: $tn' = tn_0 \setminus \{t\}$
  - Then (D, s_1, tn') is a progression of P.



Progression spaces for HTN planning

A number of HTN planners (SHOP, SHOP2, etc) plan for tasks in the order that they are to be executed. This is called *progression*.
For an HTN problem $P = (D, s_0, tn_0)$, we *progress* P by picking a task t in tn_0 with no predecessors. Then:

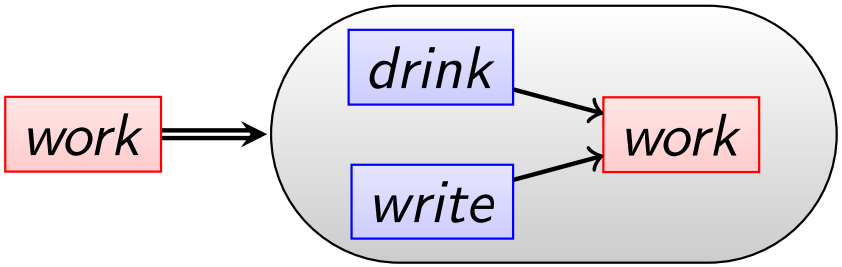
- If t is non-primitive:
  - Decompose t in tn_0, let tn' be the result
  - Then (D, s_0, tn') is a progression of P.
- If t is primitive ($t \in O$) and executable, then:
  - Apply t: $\gamma(s_0, t) = s_1$
  - Remove it from tn_0: $tn' = tn_0 \setminus \{t\}$
  - Then (D, s_1, tn') is a progression of P.



Progression-decided problems: $\leq_r$-stratification

- A $\leq_r$-stratification of task names: For every method (t, tn),
  - **Last task** on equal or lower stratum (tail recursion).
  - Every other task of tn must be on lower strata.

Methods:



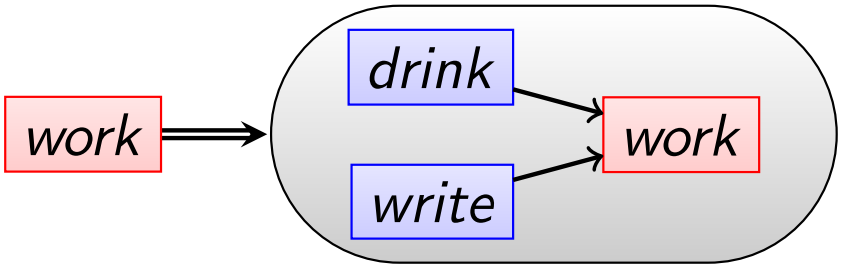
Constraints:

- $strat(work) \leq strat(work)$
- $strat(drink) < strat(work)$
- $strat(write) < strat(work)$

Progression-decided problems: $\leq_r$-stratification

- A $\leq_r$-stratification of task names: For every method (t, tn),
  - **Last task** on equal or lower stratum (tail recursion).
  - Every other task of tn must be on lower strata.

Methods:



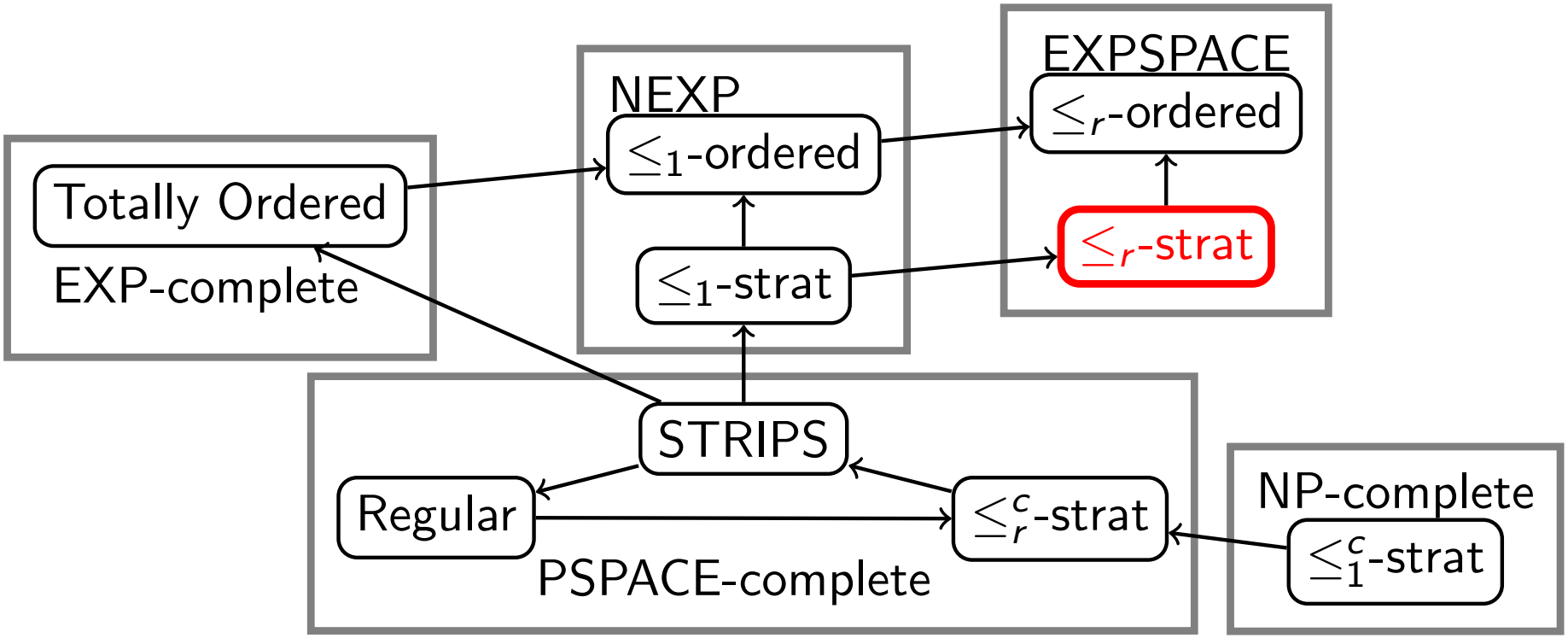
Constraints:

- $strat(work) \leq strat(work)$
- $strat(drink) < strat(work)$
- $strat(write) < strat(work)$

Here's one stratification:

$strat(work) =$	2
$strat(drink) =$	1
$strat(write) =$	1

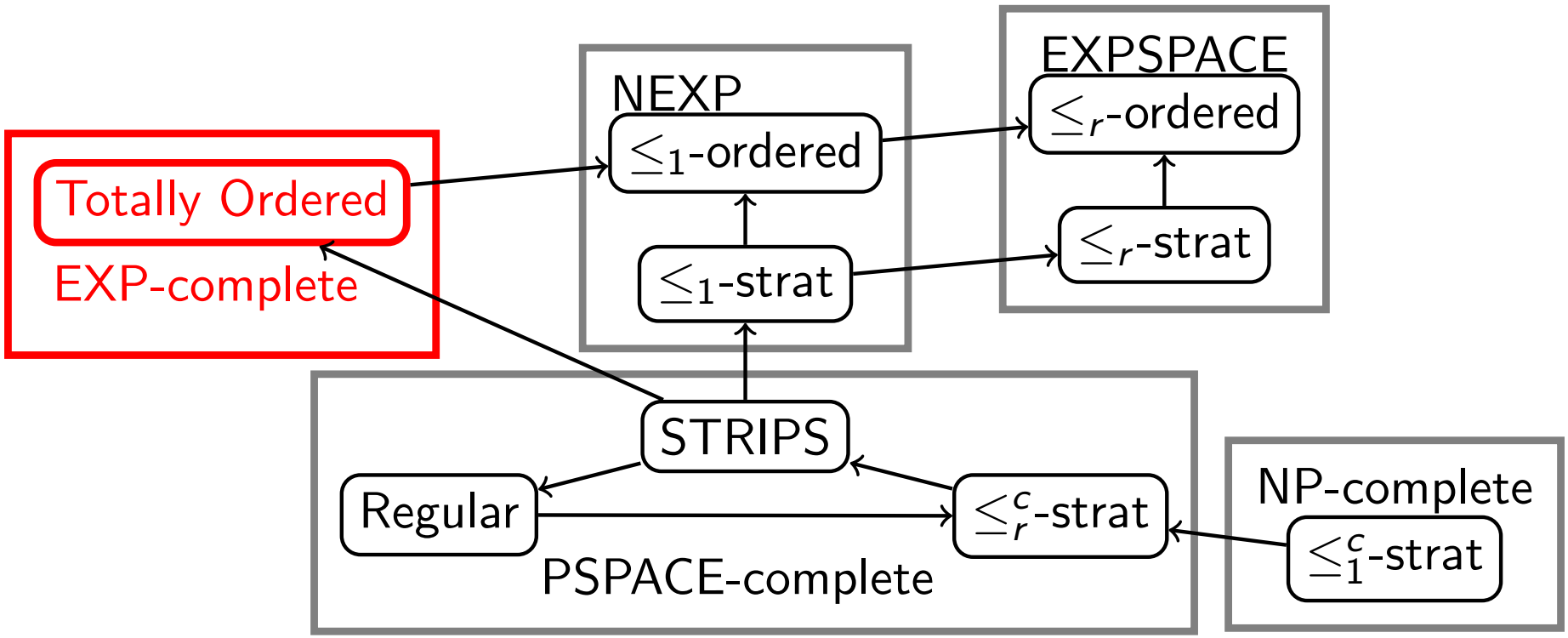
Progression spaces and $\leq_r$-stratifiable problems



From the thesis (but not in this talk):

- Property: Every $\leq_1$-stratifiable problem is $\leq_r$-stratifiable
- Theorems: Progression decides $\leq_r$-stratifiable domains
  - No larger set of progression-decided problems identifiable from methods only
  - Plan-existence in EXPSPACE for $\leq_r$-stratifiable since it's unclear what the bound is for length of a solution

Totally-Ordered Planning is EXP-time complete



- Totally-Ordered HTN Problems: Initial task network and all method networks are totally ordered.
- Known to be in EXP-time (Erol et. al., TOPHTN from thesis)

Totally-Ordered Planning is EXP-time complete

Totally Ordered Planning:

- Good at Palindromes! The following methods always return to the same state the task '*check*' starts in:

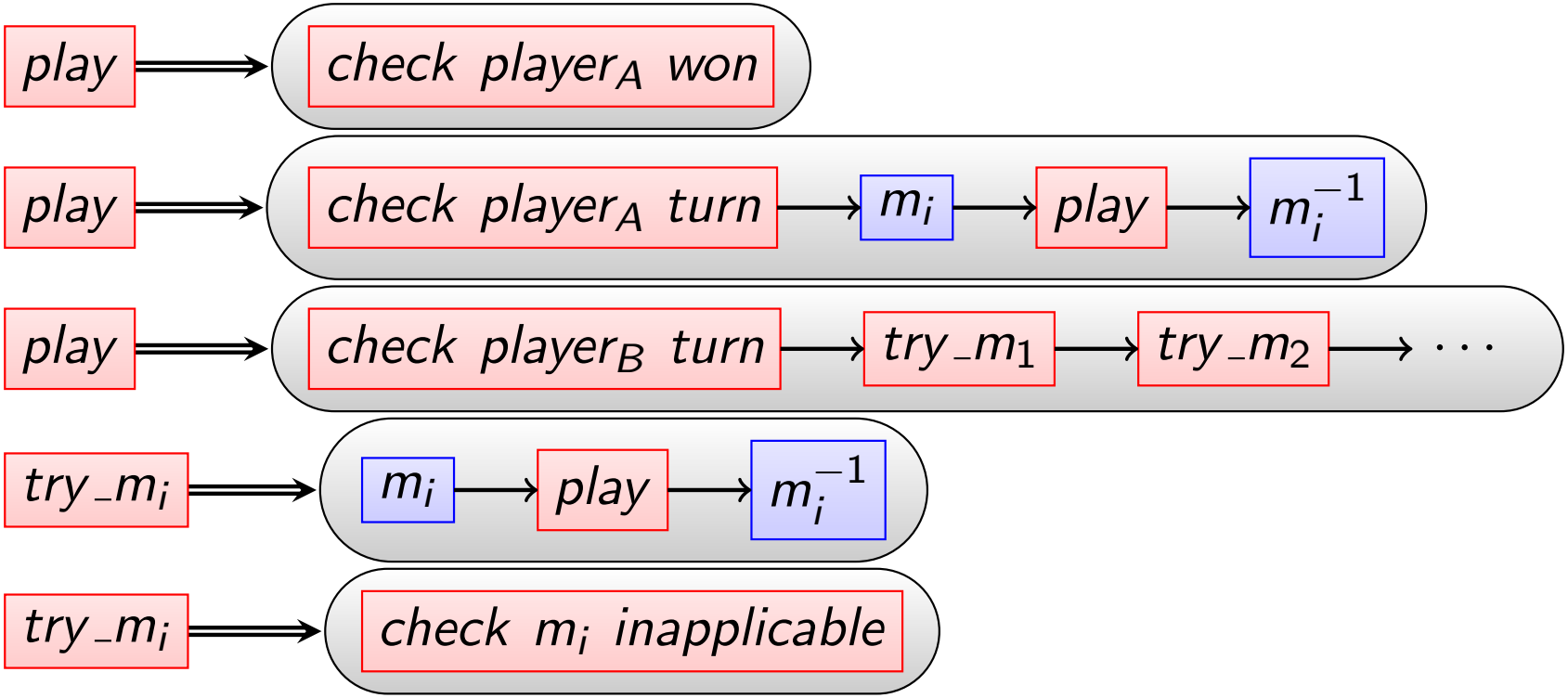
Methods:

- Generalized checkers, chess, versions of Go are all EXP-complete
- We can use palindrome like methods to encode position evaluation as an HTN problem

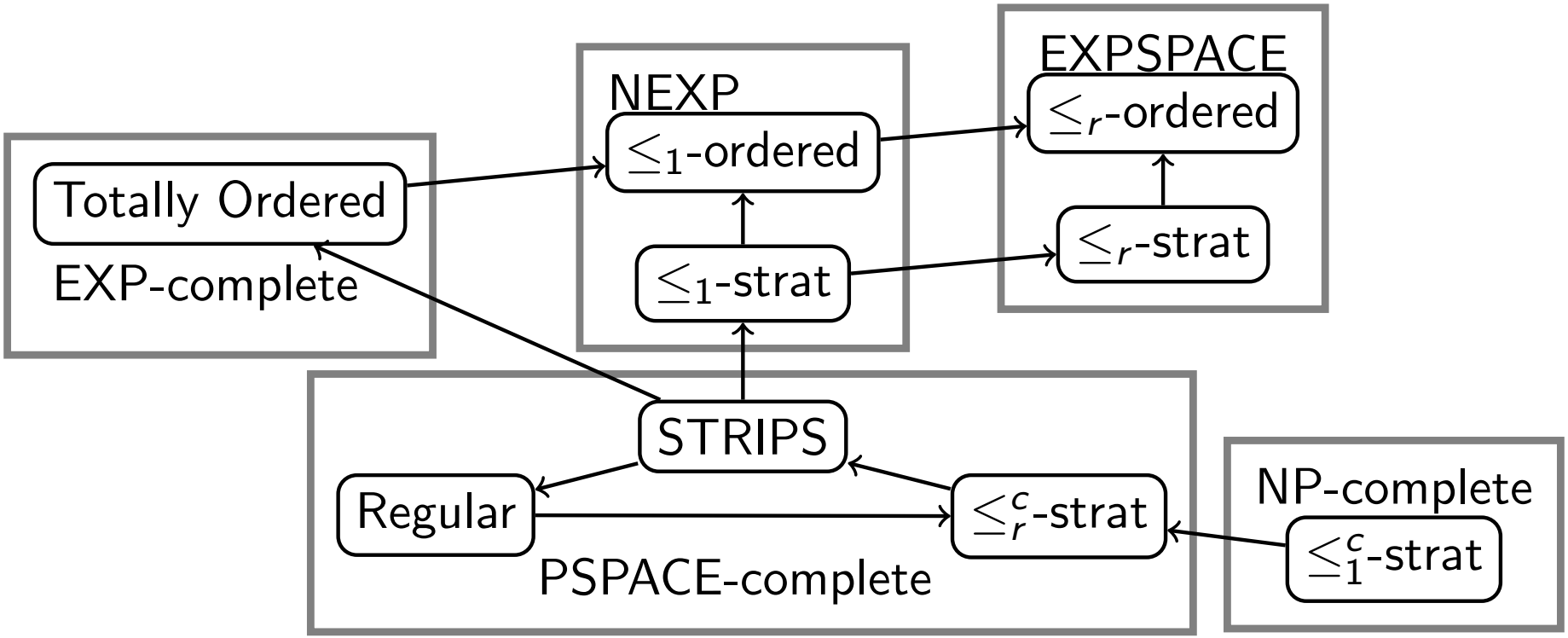
Encoding game tree search

Encoding game tree search: Can $player_A$ force a win?

Methods:



Summary



Characterized four different problem spaces for HTN problems:

- Provided terminating HTN planning algorithms for each space.
- Provided syntactic checks for finiteness for each space.
- Provided expressivity and running-times for syntactically identifiable problems

Outline

Heuristics

Domain independent heuristics in search:

- Problem spaces are still intractably large to search
- A heuristic underestimates the distance to nearest solution, or we pretend it does
- Usually based on solving a *relaxed problem*
  - Should take low-order polynomial time to solve
  - Solution to full problem implies solution to relaxed problem (but not vice versa)

Delete-Free Planning

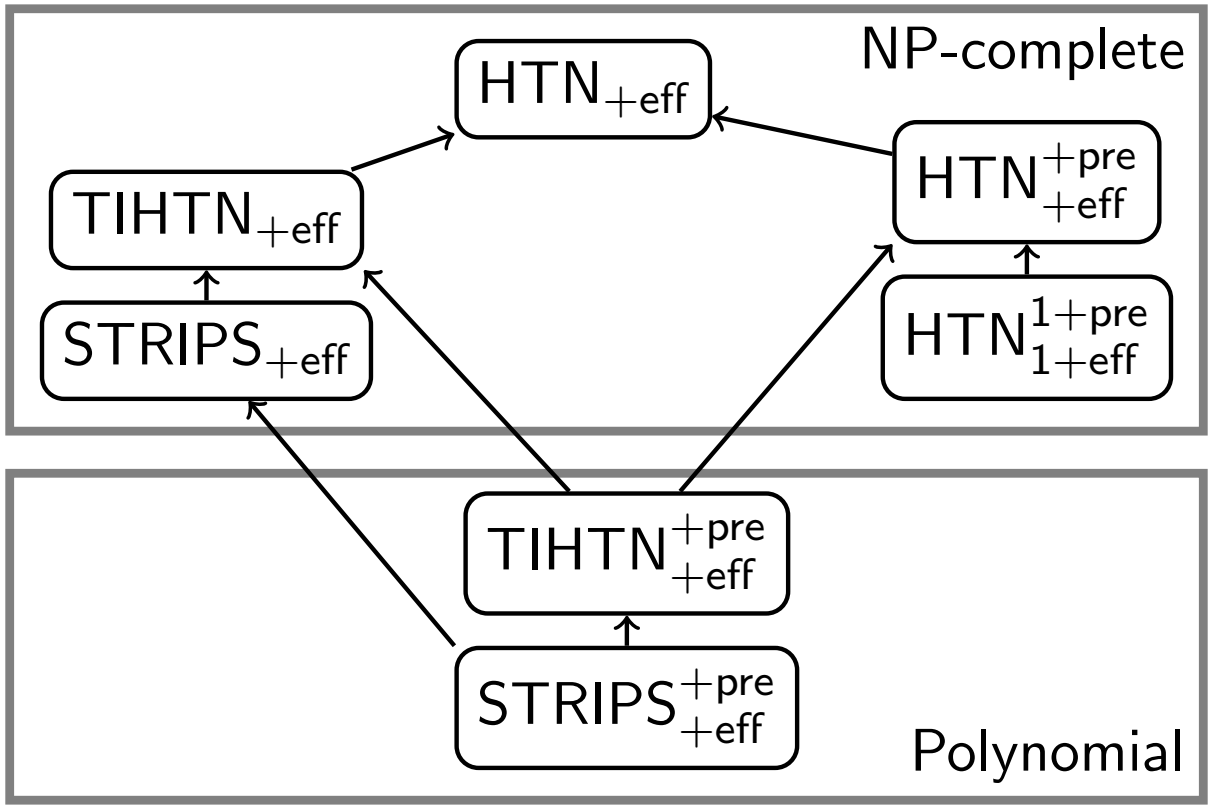
Delete-Free Planning In STRIPS:

- All operators have positive preconditions and effects; positive goal
- Poly-time to find a plan: just apply operators until a fixed point is reached
- NP-hard to find optimal plan
- Basis for the most influential classical planning heuristics

Delete-Free HTN Planning:

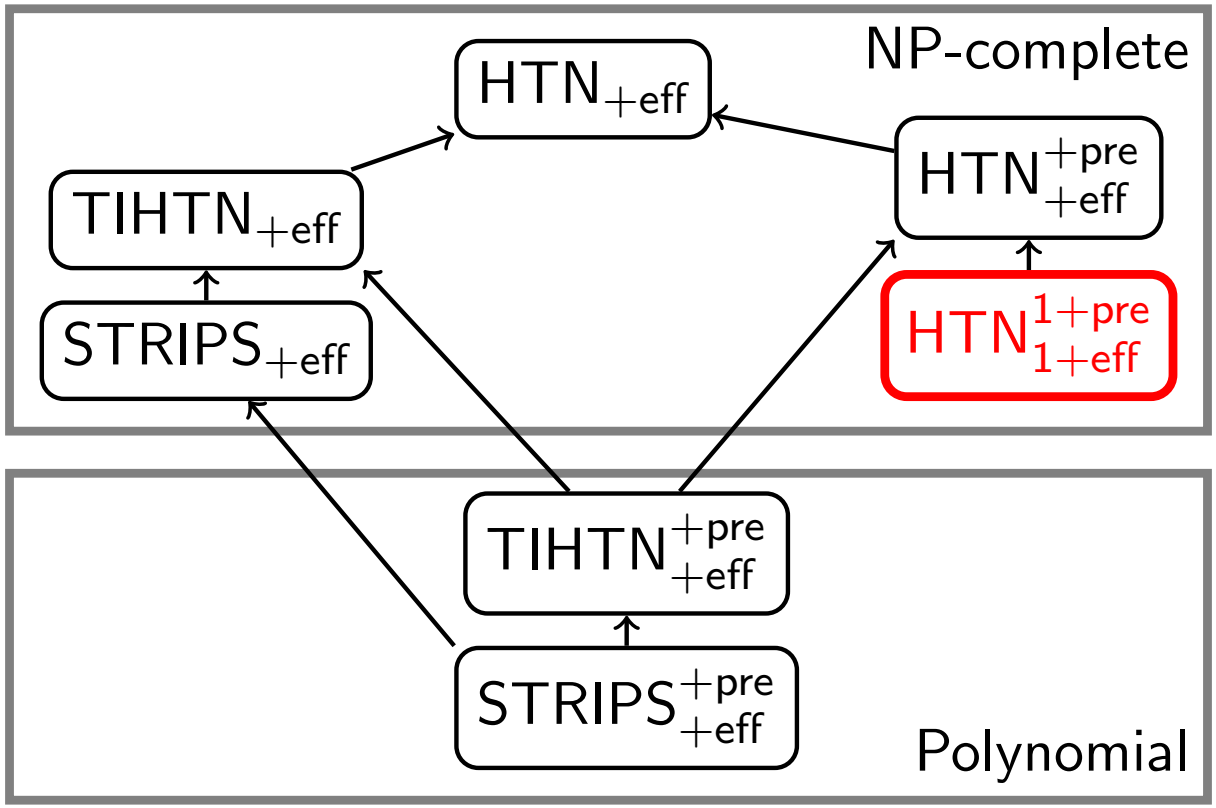
- Positive preconditions and effects
- Decidability and complexity previously unknown

Delete-free HTN planning is NP-complete



- Complexity of plan-existence for propositional delete-free STRIPS and HTN planning with various restrictions. Arrows represent subclass relationships
- The STRIPS results are from Bylander; the other results are new

Delete-free HTN planning is NP-complete



- Complexity of plan-existence for propositional delete-free STRIPS and HTN planning with various restrictions. Arrows represent subclass relationships
- The STRIPS results are from Bylander; the other results are new

Delete-Free HTN planning is NP-hard

CNF-SAT: $(V_1 \vee \neg V_3 \vee V_4) \wedge (V_2 \vee V_3 \vee \neg V_4) \wedge \cdots$

Translation of setting variables to delete-free HTN planning:

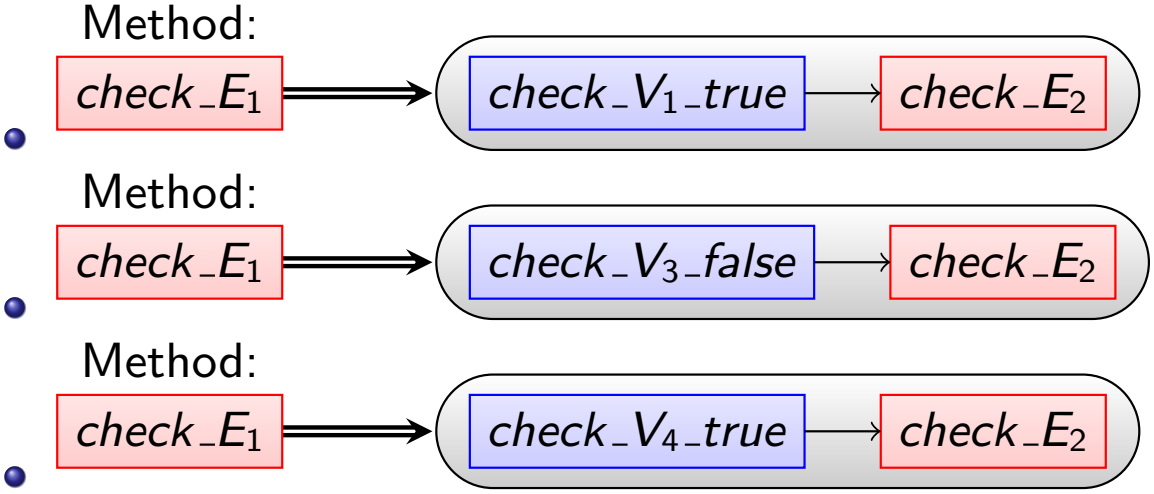
- For each variable V_i:
  - Two propositions: V_i_true, V_i_false
  - Two test operators: $check\_V_i\_true$, $check\_V_i\_false$
  - Two set operators: $set\_V_i\_true$, $set\_V_i\_false$
  - Two methods:
    - Method: set_V_i ⟹ $set\_V_i\_true$ → set_V_{i+1}
    - Method: set_V_i ⟹ $set\_V_i\_false$ → set_V_{i+1}
  - For last variable, replace last task with $check\_E_1$, instead

Delete-Free HTN planning is NP-hard

CNF-SAT: $(V_1 \vee \neg V_3 \vee V_4) \wedge (V_2 \vee V_3 \vee \neg V_4) \wedge \cdots$

Translation of testing the clauses to delete-free HTN planning:

- For disjunctive clause E_i:
  - A method for each literal



  - For last clause, drop the final *check* task

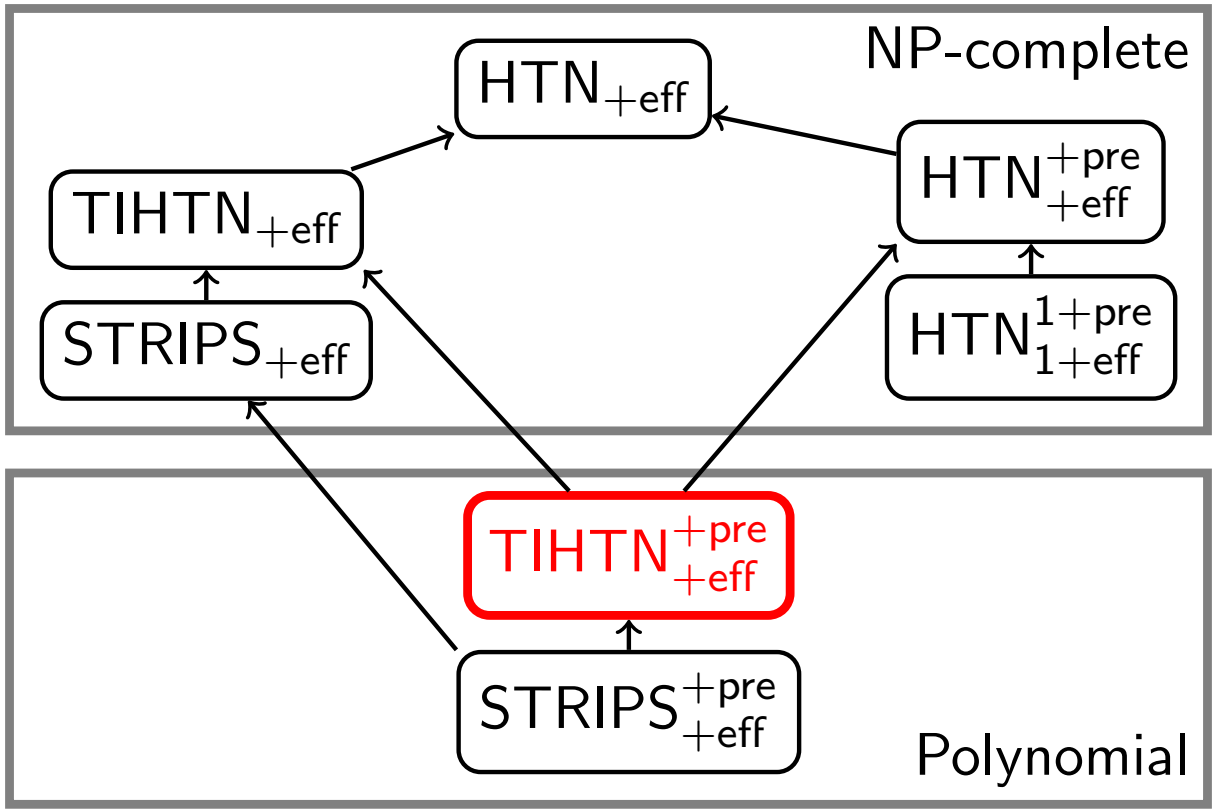
Delete-Free HTN planning is NP-complete

- Polynomial encoding of CNF-SAT means delete-free HTN planning is NP-hard.
- We can also show delete-free HTN planning is in NP(in thesis), so it is NP-complete.
- So we need to relax the semantics of HTN planning to find a poly-time heuristic.

Delete-Free Task-Insertion HTN Planning

- Need to relax HTN semantics to find poly-time heuristic
- Task-Insertion HTN (TIHTN) Planning:
  - Allows insertion of arbitrary operators into task network
- Delete-free TIHTN planning is poly-time:
  - Insert operators until you reach a fixpoint
  - Test to see if tasks are executable in the fixpoint state (similar to testing if a CFG is empty).

Relaxed-semantics HTN heuristics exist



- So there exist poly-time heuristics for HTN planning. But if they deviate significantly from HTN semantics, will they still be useful?

Outline

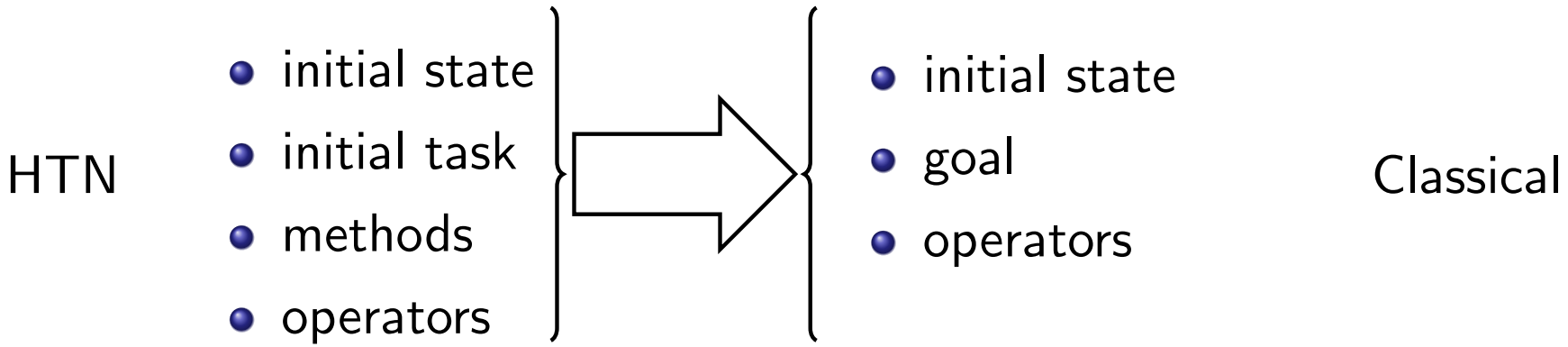
Translating to Classical Planning:

Motivation

- Variety of domain independent heuristics
- Highly tuned search implementations
- Want to make use of these for HTN planning!

HTN Planning via translation to Classical Planning

- Both STRIPS and totally-ordered $\leq_r$-stratifiable planning are PSPACE-complete
- We can translate any totally-ordered $\leq_r$-stratifiable HTN problem into a classical planning problem.
- To do this, we translate the progression space of an HTN planner into the state space of a classical planner:

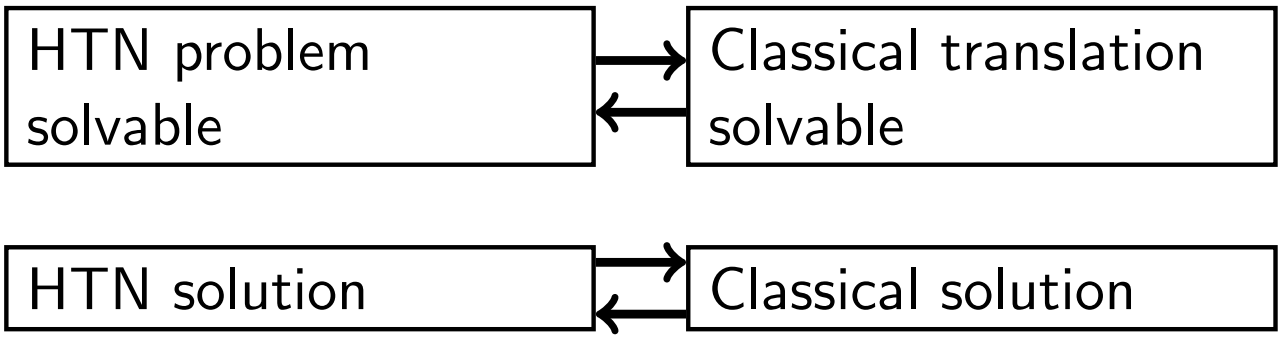


- (detail in the thesis)

Formal Properties

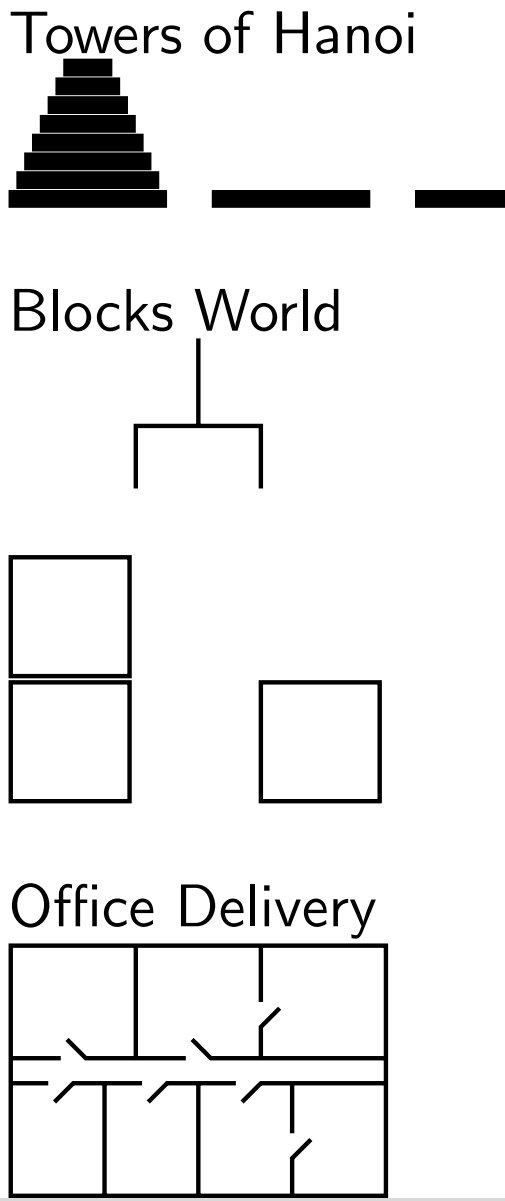
Given an $\leq_r$-stratifiable problem:

- A solution to the translated problem $\iff$ a solution to the HTN problem
- Any classical planner is now an HTN planner (free heuristic function to guide the search and often guaranteed termination).



Experiments

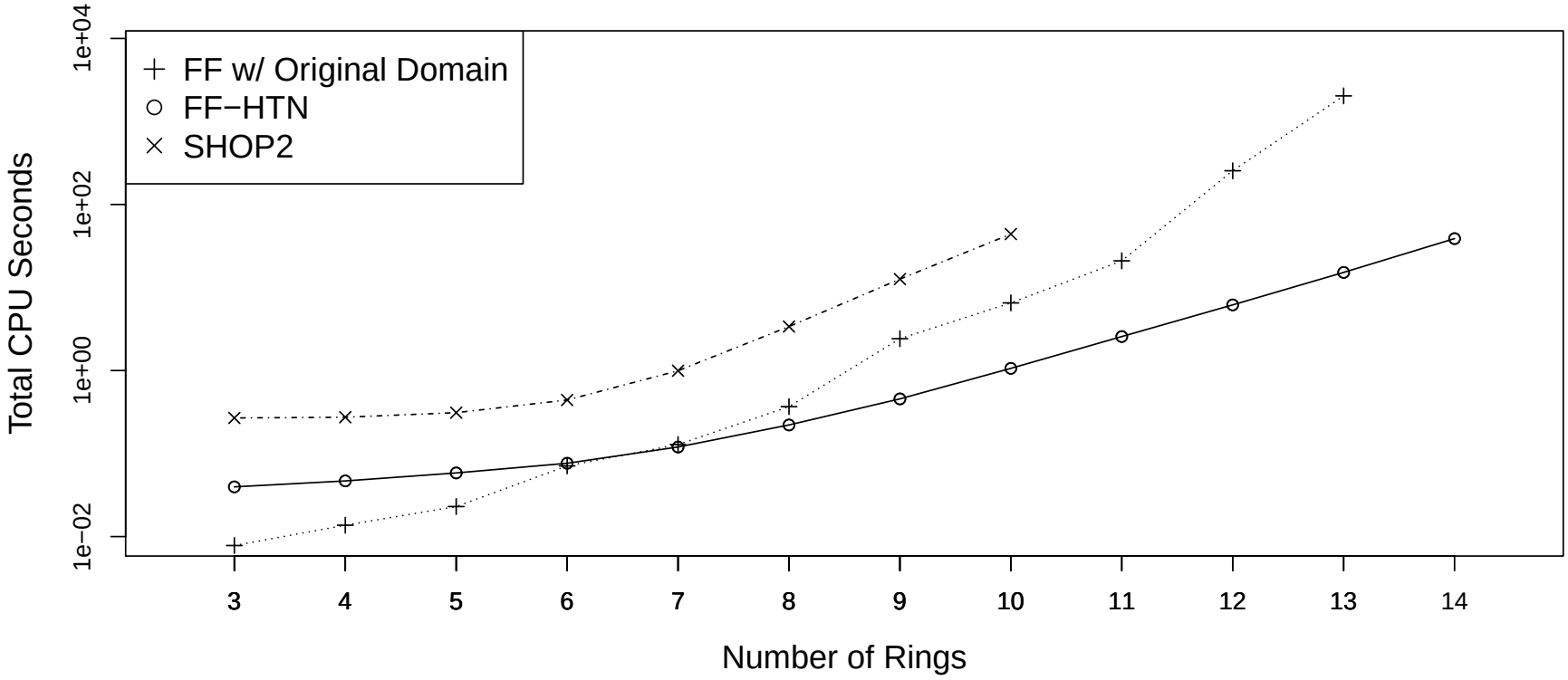
- Took FastForward (FF) [Hoffmann and Nebel, 2001]
  - FF is a well known classical planner
  - Investigated how well FF did on its own
  - Investigated how FF did with a translated HTN
- Compared against SHOP2 [Nau et al., 2003]
  - SHOP2 is an influential HTN planner
  - Used here with depth-first search
- 3 domains, about 5000 experiments:
  - Towers of Hanoi
  - Blocks World
  - An office delivery domain



Results: Towers of Hanoi

- Exponentially long plans
- No branching in the HTN domain
- Average of 100 runs / point

- FF with the translated HTNs runs exponentially faster
- SHOP2 timed out after 10 rings
  - Most likely a performance bug

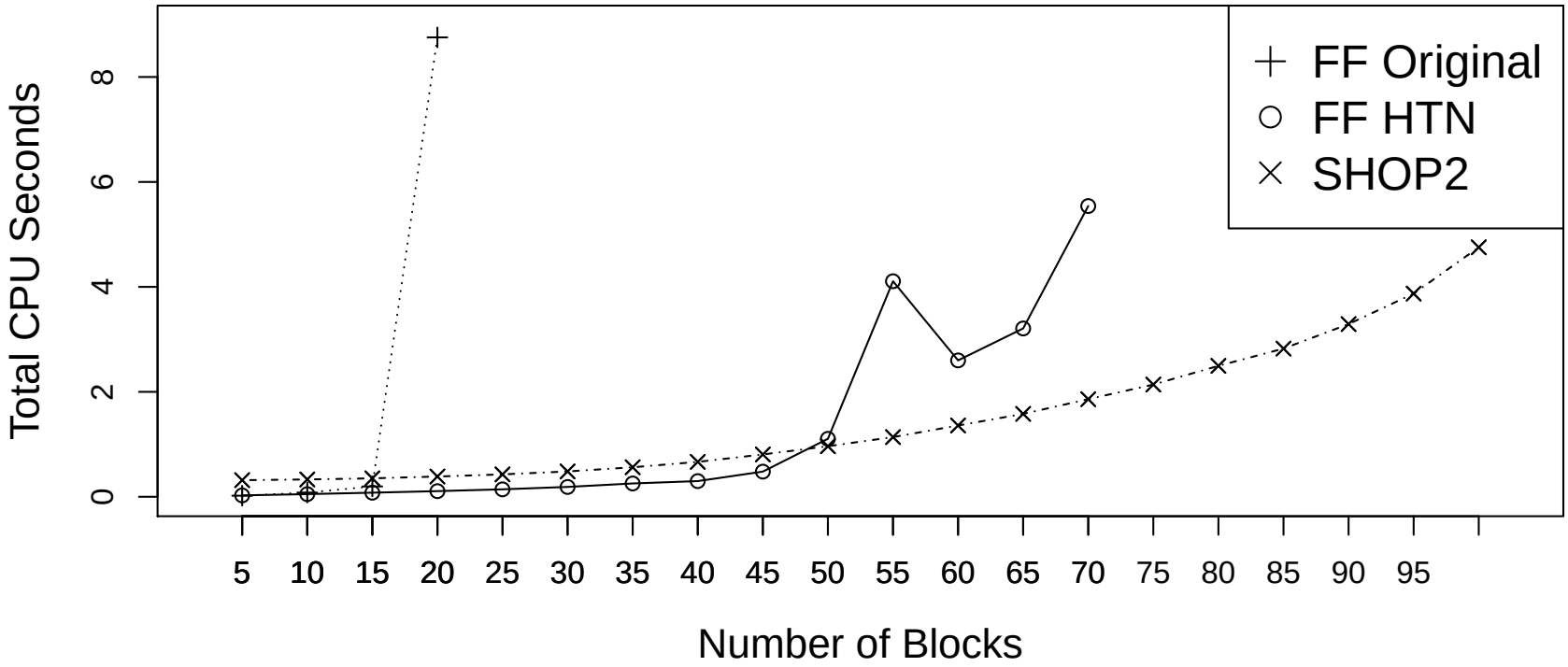


Results: Blocks World Time

HTN description

- Forces a polynomial length plan
- Does not enforce optimal solution

- FF's heuristic is known poor at blocks world
- FF with translated HTNs can solve problems up to 70 blocks in size
- SHOP2 scaled better after 50 blocks.

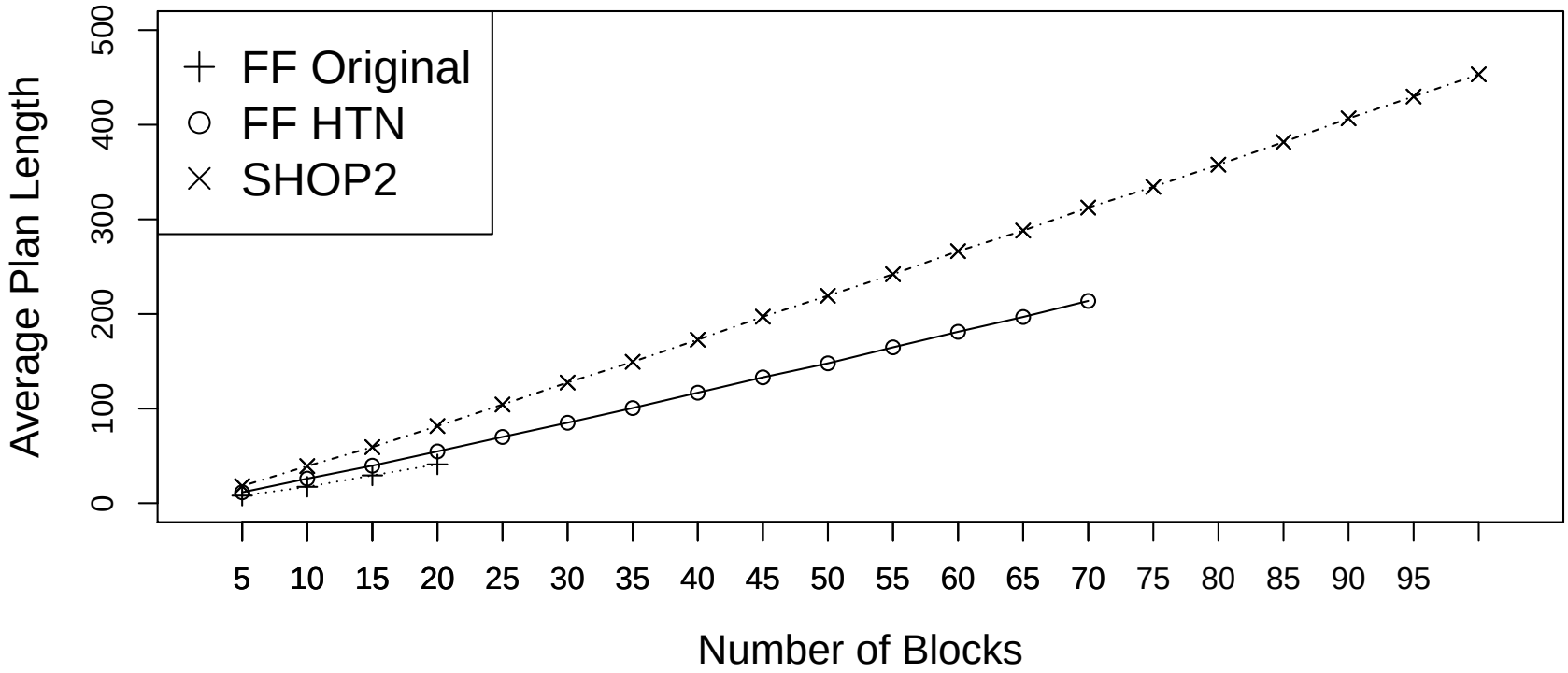


Results: Blocks World Plan Length

HTN description

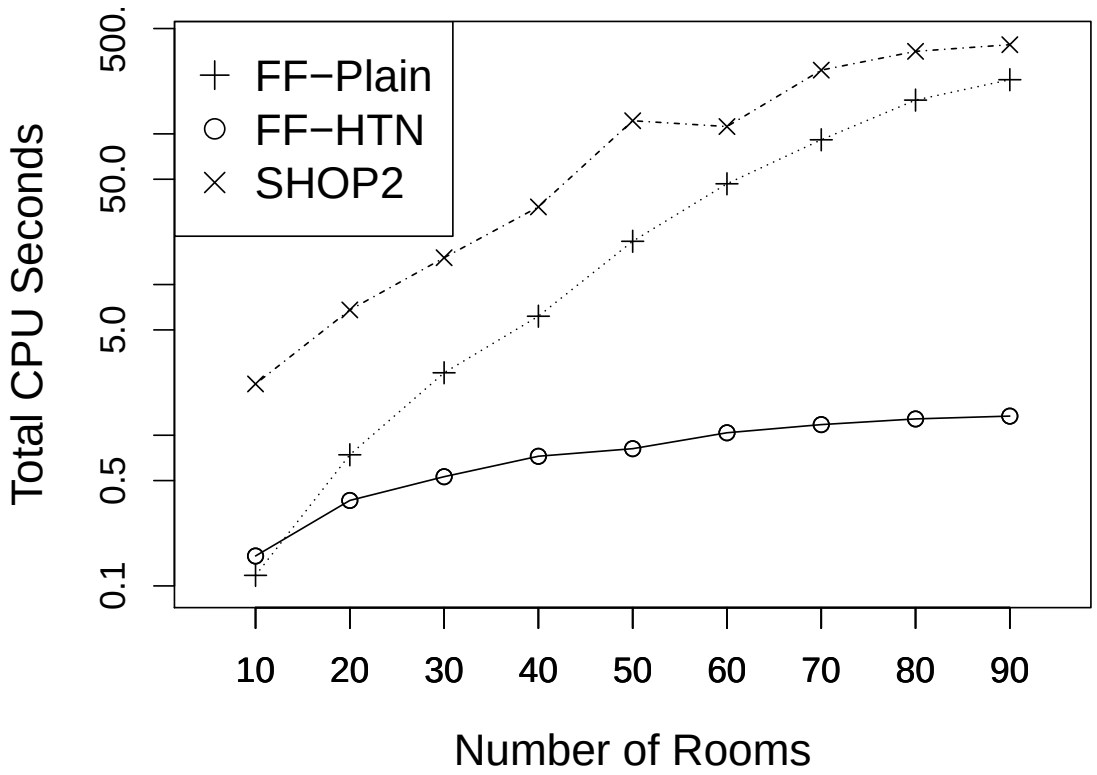
- Forces a polynomial length plan
- Does not enforce optimal solution

- FF with HTNs produced better plans than SHOP2
- HTNs forbid some optimal solutions, hence the plans were not the shortest possible



Results: An Office Delivery Domain

- Office delivery: Pick up packages, deliver to another room.
- HTNs orders 'pickup'/'putdown' actions
- No guidance on what room to move to.



- SHOP2 has no heuristic to guide navigation
- FF without HTNs is poor at package management
- FF with HTNs returns plans exponentially faster than either of the above.

Summary

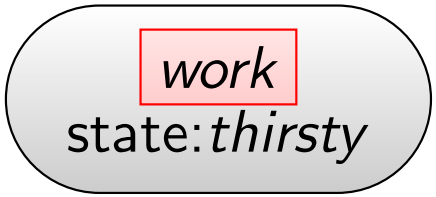
- Four problems spaces for HTN planning
  - Syntactic checks for when those spaces must be finite
  - Algorithms to search these spaces (unimplemented)
  - Expressivity and running times for identified spaces
- Theoretical results impacting HTN heuristics:
  - Delete-free HTN planning is NP-complete
  - Poly-time relaxed-semantics variants exist
- An HTN-to-Classical translation for any totally-ordered $\leq_r$-stratifiable problem

Contributions

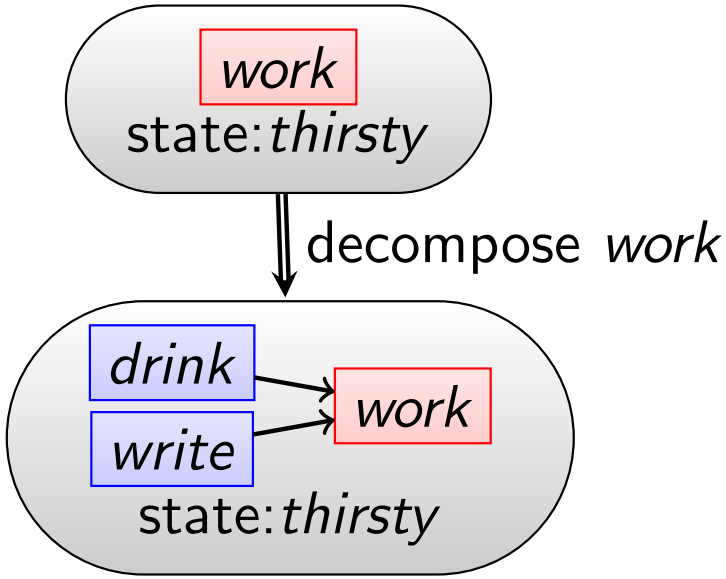
Provided:

- Theoretical background to compare HTN planning search algorithms
- Theoretical limits on fidelity of HTN heuristics
- Evidence that heuristic search is effective for HTN planning

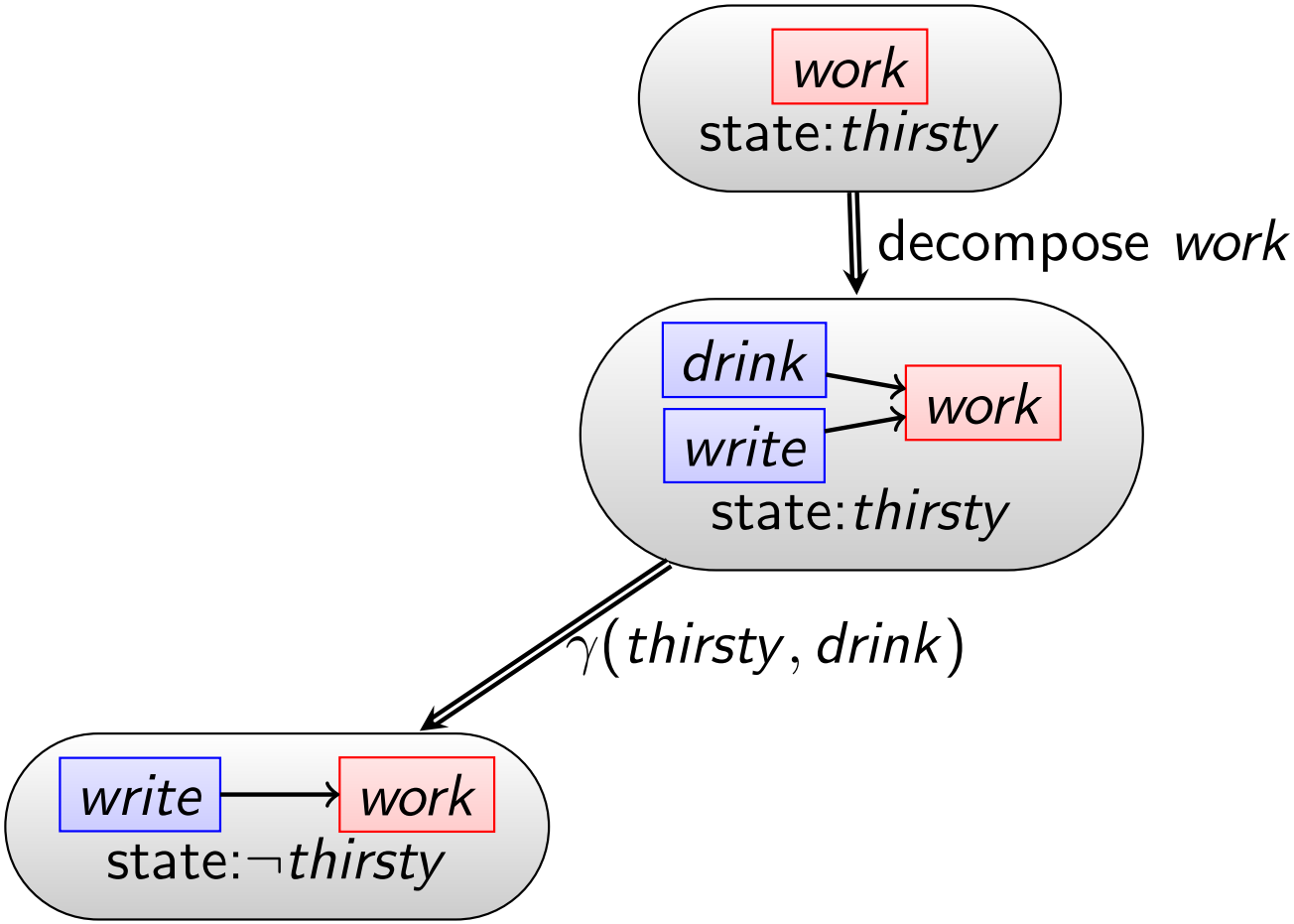
Progression Example (over problems, not task networks)



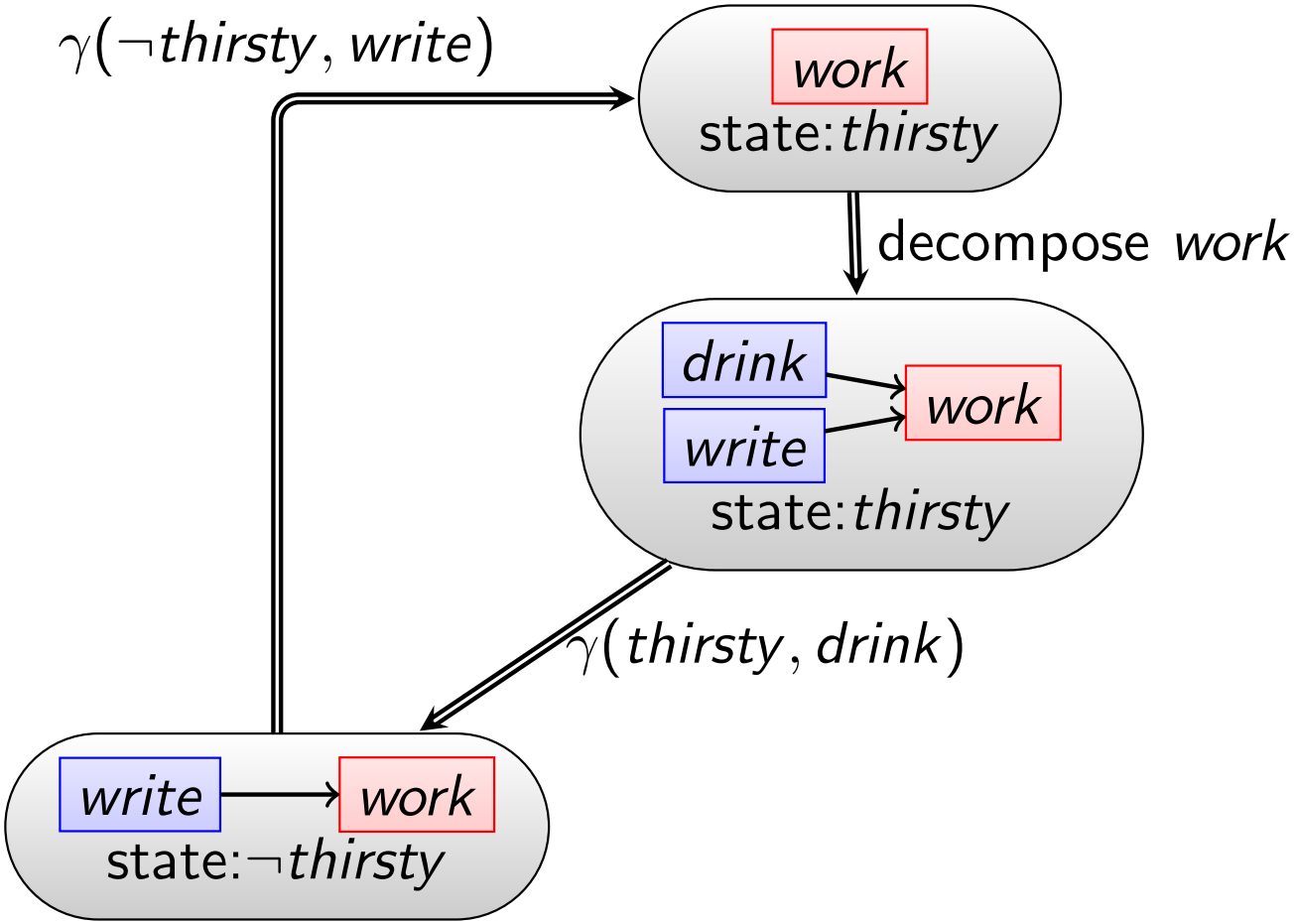
Progression Example (over problems, not task networks)



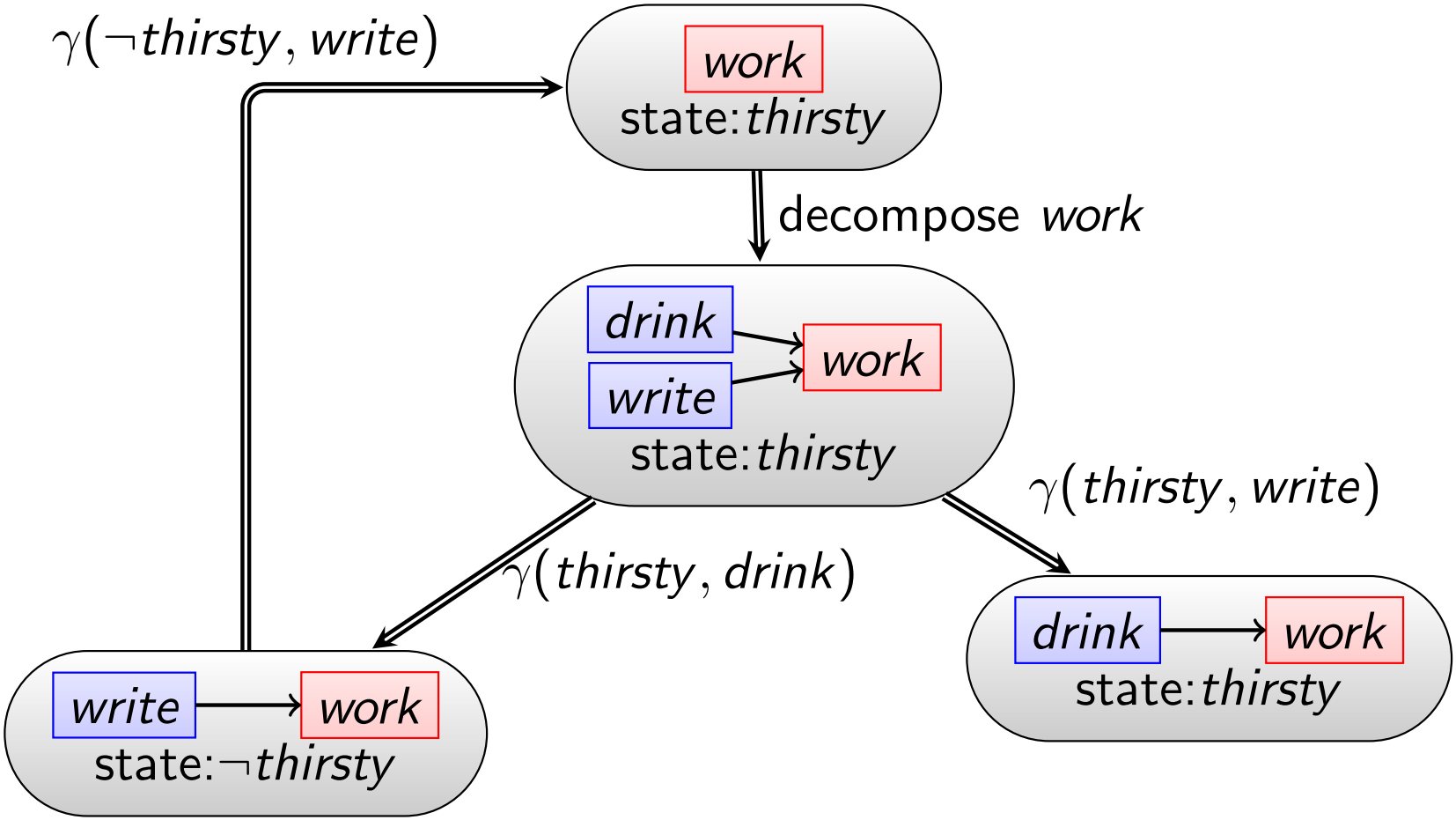
Progression Example (over problems, not task networks)



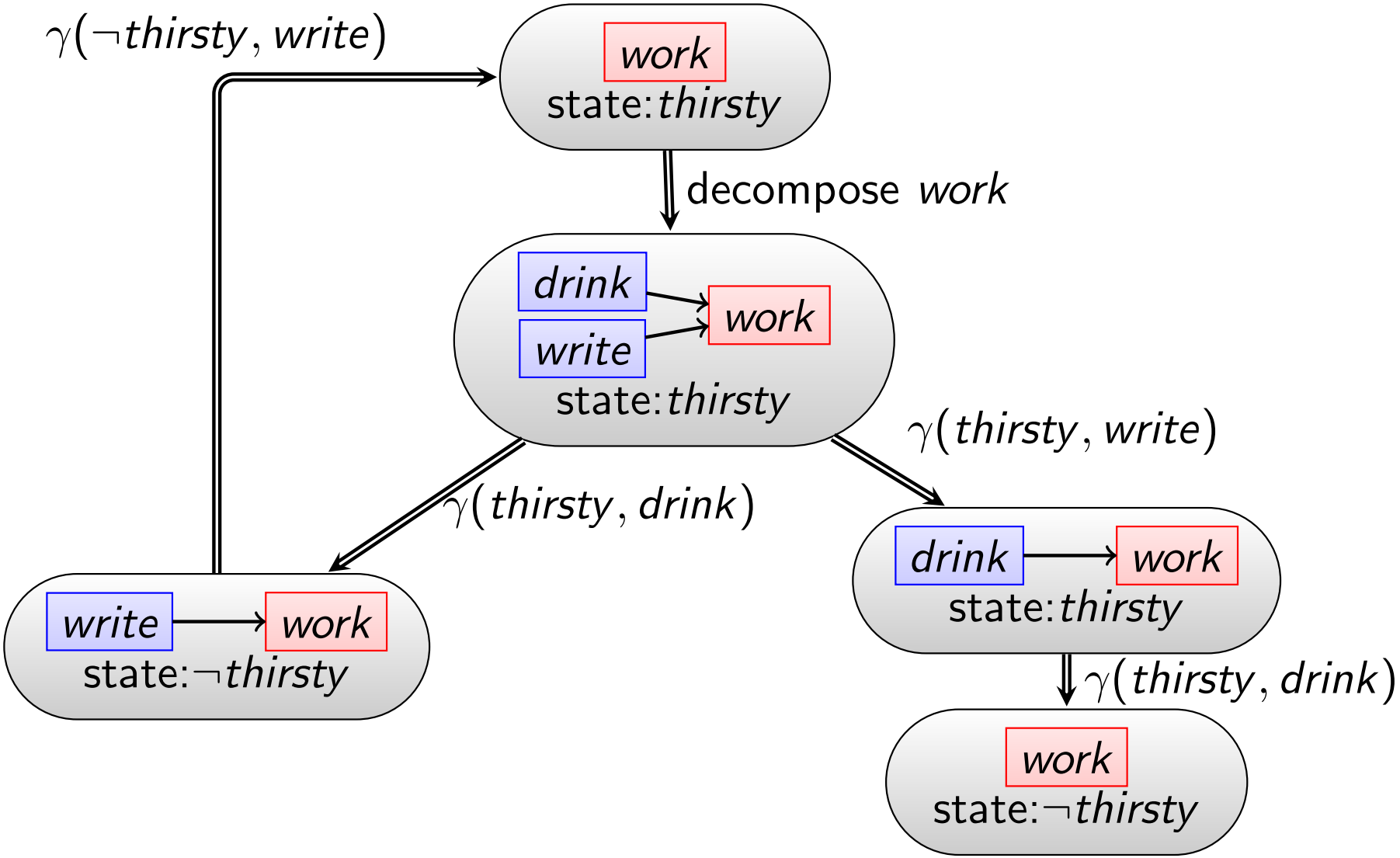
Progression Example (over problems, not task networks)



Progression Example (over problems, not task networks)



Progression Example (over problems, not task networks)



Totally-Ordered $\leq_r$-stratifiable problems

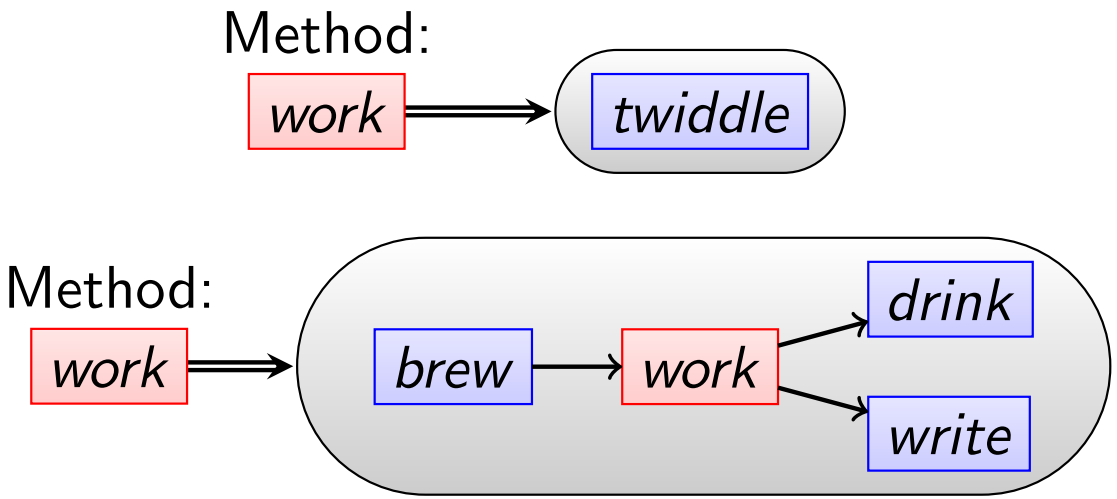
- Totally-Ordered HTN Problems: Initial task network and all method networks are totally ordered.
- Theorem: Totally-ordered $\leq_r$-stratifiable planning is PSPACE-complete. Proof:
  - Hardness: Contains *regular* HTN planning (Erol et. al.)
  - Progression space contains only totally-ordered problems
  - Progression can only add lower stratum tasks to the beginning of the task network
  - So $\langle 4, 4, 5, 5 \rangle \rightarrow \langle 3, 3, 3, 4, 4, 5, 5 \rangle$
  - Max network size: $m \cdot h$, where m size of largest method, h height of stratification.
- Contains totally-ordered $\leq_1$-stratifiable planning. Searching progression space is optimal, searching decomposition space is not.

Motivating Example: Morning coffee

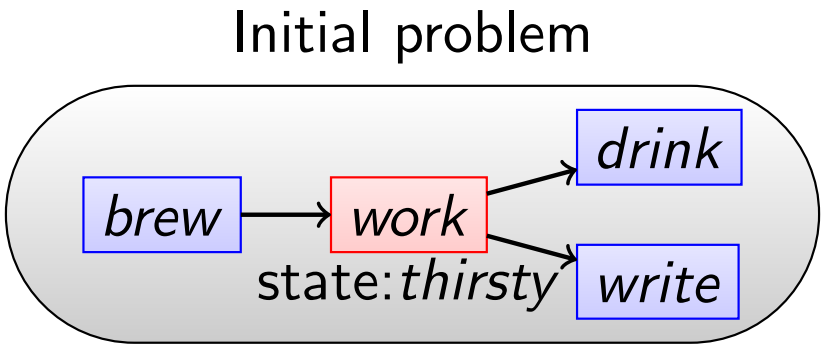
- Brew as much coffee in the morning as you need throughout the day.
- Not $\leq_r$-stratifiable ('*work*' now in the middle)
  - '*work*' is a subtask of itself, but not the last task.

Domain actions:

- $\forall_s \gamma(s, brew) = s$
- $\forall_s \gamma(s, twiddle) = s$
- $\forall_s \gamma(s, write) = thirsty$
- $\forall_s \gamma(s, drink) = \neg thirsty$



Total Order Partition spaces



- Plan by splitting problems into smaller problems
- If there is a total order over chunks of the task network:
  - Serialize network into smaller chunks
  - End state of one is the start state of the next.
  - If last chunk solvable, then the whole problem is solvable
- When no total order, use progression
- Call this the Total Order Progression (TOP) space
  - No existing planner terminates on all problems with a finite TOP space
  - Possible to write an algorithm that does (see thesis for details)

Total Order Partition spaces

$\leq_r$-ordered

- For finiteness, need to consider the structure of networks in the problem.
- A task network tn is $\leq_r$-*ordered* if $\langle tn_1, \ldots, tn_n \rangle$ are the totally ordered chunks of tn and every tn_i is either:
  - singular (only one task)
  - $\leq_r$-stratifiable
- A problem is $\leq_r$-*ordered* if the initial task network and all methods' task networks are $\leq_r$-ordered
- $\leq_1$-ordered is defined similarly over $\leq_1$-stratification

TOP spaces and $\leq_r$-ordered problems

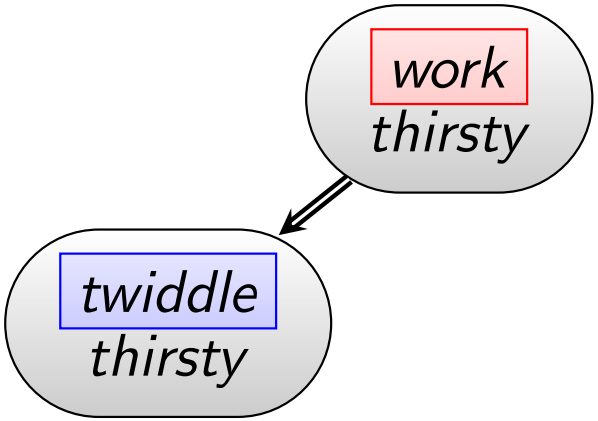
From the thesis (but not in this talk):

- Property: If a problem is $\leq_r$-stratifiable, it is $\leq_r$-ordered
- Property: If a problem is totally ordered, it is $\leq_r$-ordered
- Theorem: If a problem is $\leq_r$-ordered, its TOP space is finite.
  - SHOP and SHOP2 may not terminate on these problems.
  - Possible to implement a planner that does (TOPHTN).
- Plan-existence in EXPSPACE for $\leq_r$ ordered, in NEXP for $\leq_1$-ordered

Example

End states of $(D, thirsty, work)$:

Example



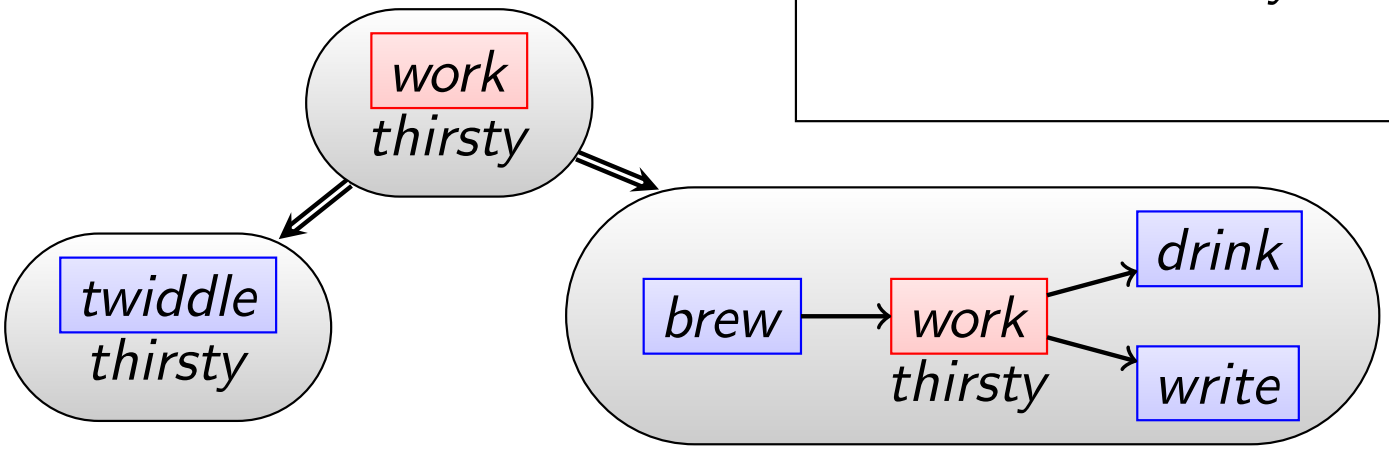
End states of $(D, thirsty, work)$:

thirsty

Example

End states of $(D, thirsty, work)$:

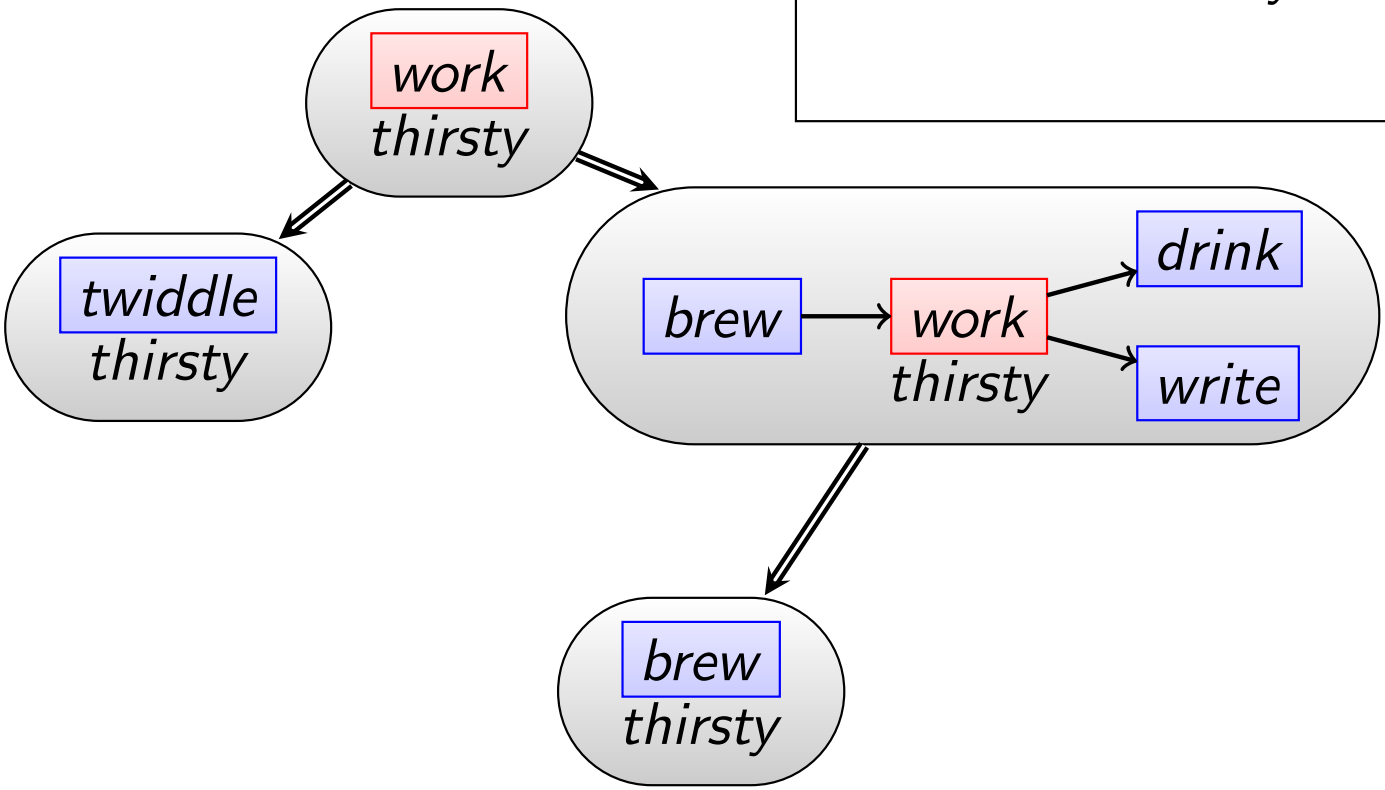
thirsty



Example

End states of $(D, thirsty, work)$:

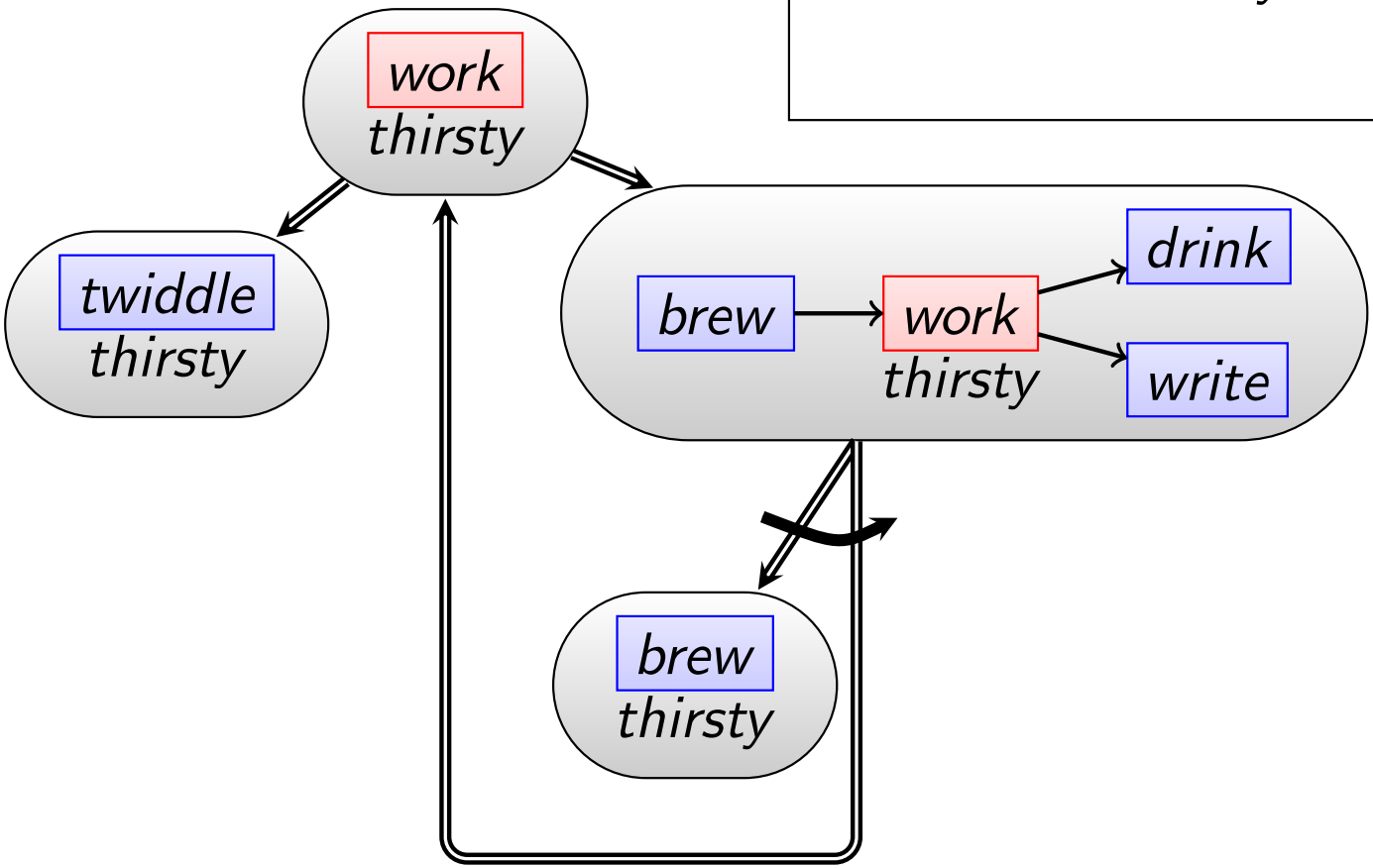
thirsty



Example

End states of $(D, thirsty, work)$:

thirsty

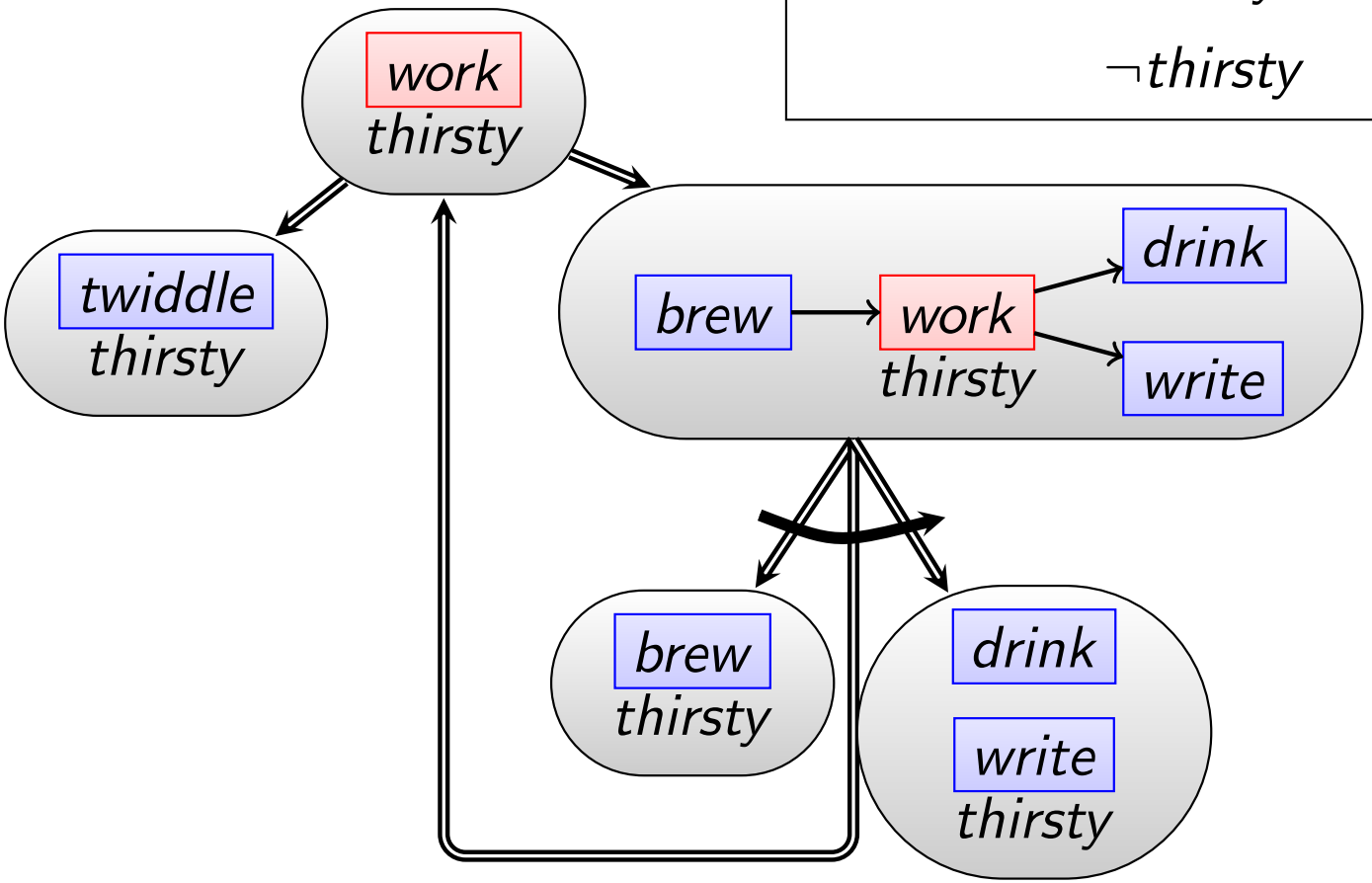


Example

End states of $(D, thirsty, work)$:

$thirsty$

$\neg thirsty$

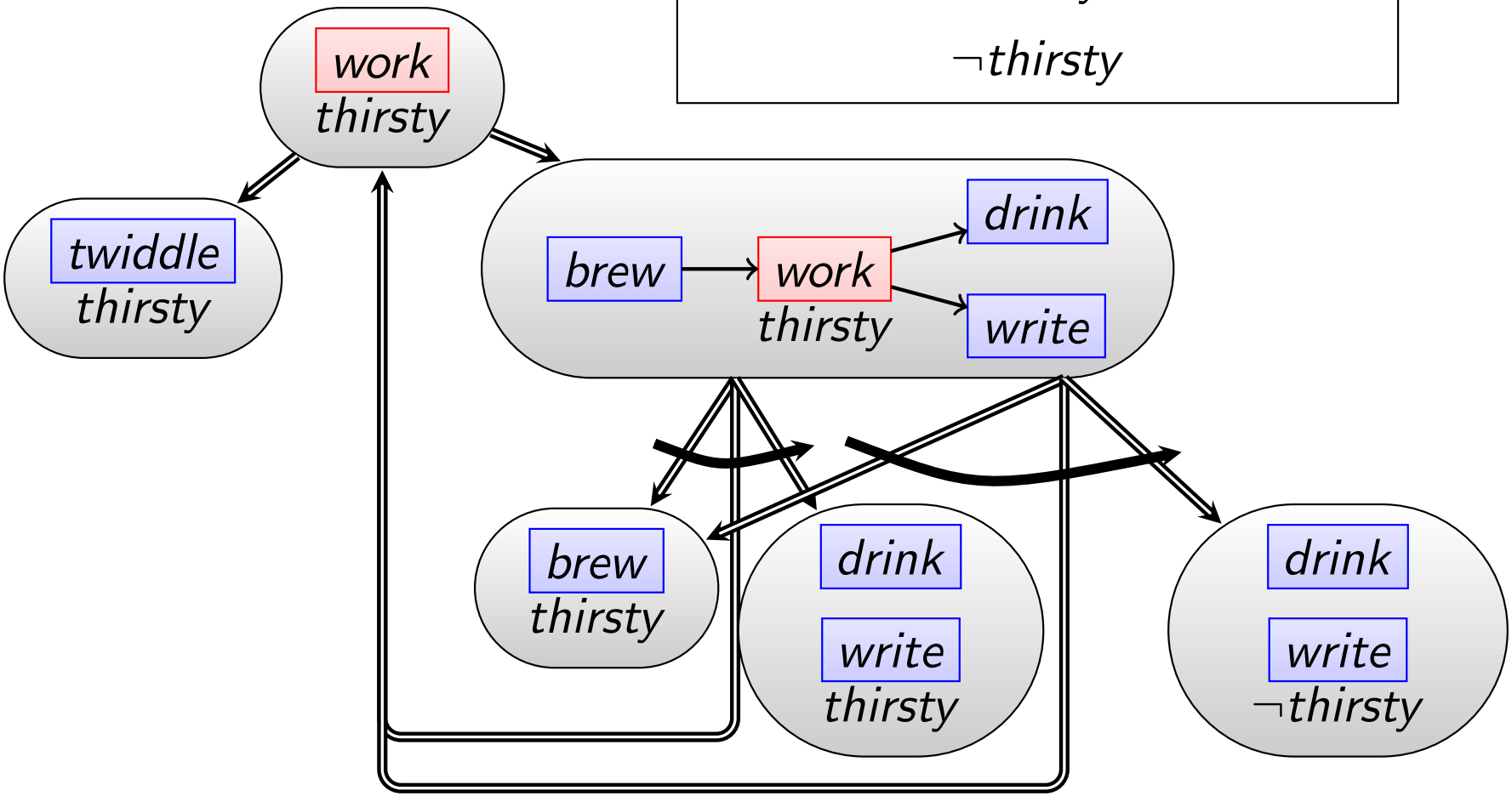


Example

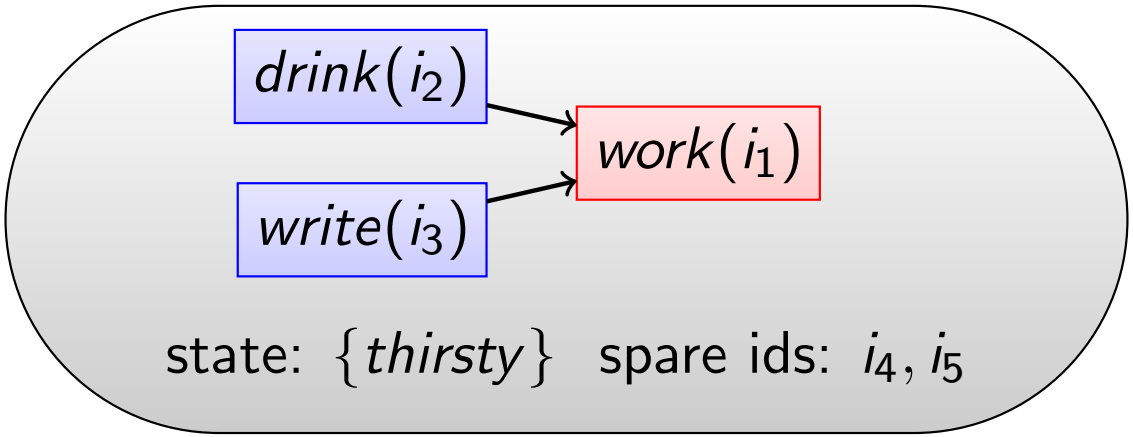
End states of $(D, thirsty, work)$:

$thirsty$

$\neg thirsty$

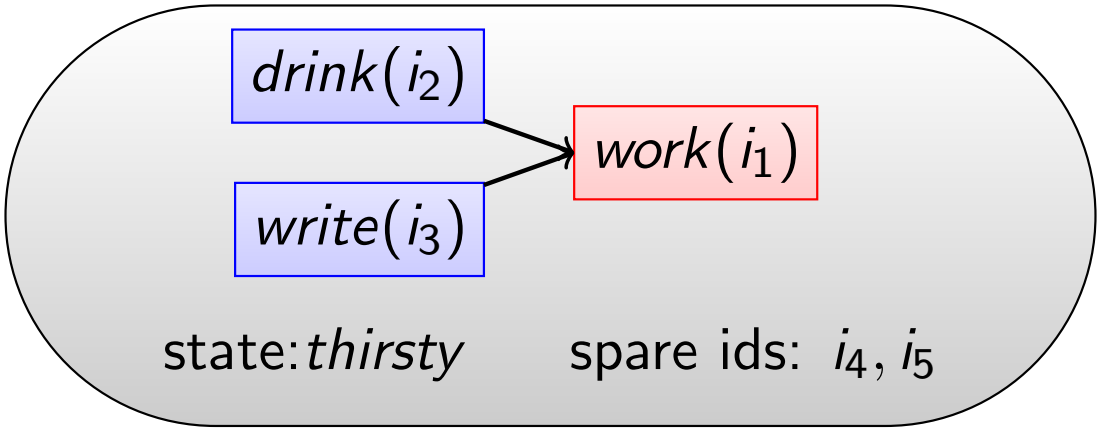


Translating a Task Network



- Associate each task with a unique identifier: *brew*(i_1), *drink*(i_2), *write*(i_3), *work*(i_4)
- Mark which task nodes have no preceding tasks: *running*(i_1)
- Describe the ordering relationships between the nodes
- Add a pool of spare identifiers

Translating Operators



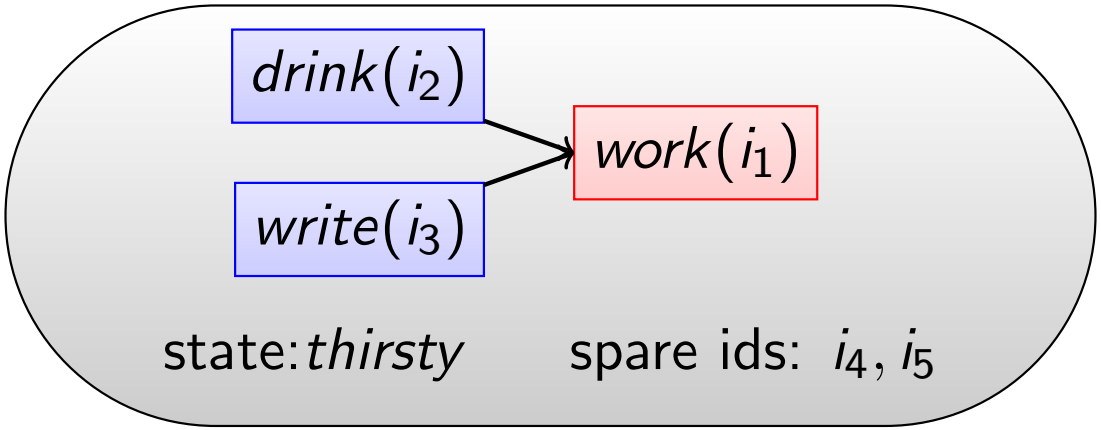
- For operators
  - Syntactically the same in HTN and classical planning
  - Semantically, can only apply an operator when it's in the task network and has no predecessors
  - Add task literal to precondition of operator

Operator: *brew*(i)

pre = {*brew*(i), *running*(i) ...}

eff = {¬*brew*(i), ¬*running*(i), *finished*(i) ...}

Translating Operators



Execute *drink*: $\gamma(thirsty, drink(i_2)) = \neg thirsty$

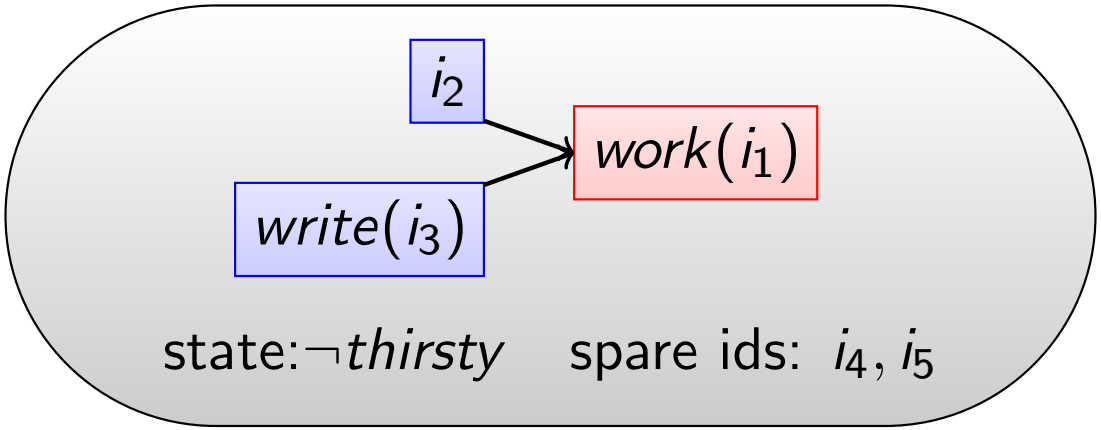
- For operators
  - Syntactically the same in HTN and classical planning
  - Semantically, can only apply an operator when it's in the task network and has no predecessors
  - Add task literal to precondition of operator

Operator: $brew(i)$

$pre = \{brew(i), running(i) \dots\}$

$eff = \{\neg brew(i), \neg running(i), finished(i) \dots\}$

Translating Operators

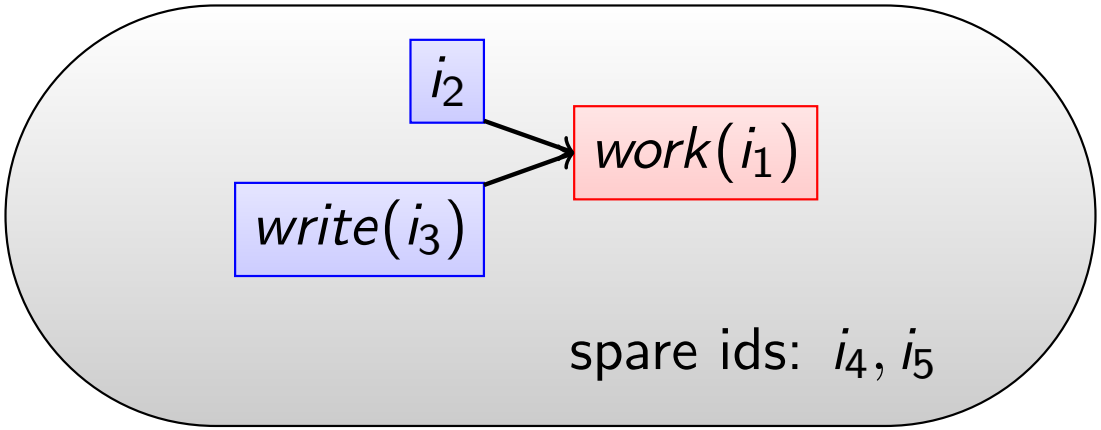


- For operators
  - Syntactically the same in HTN and classical planning
  - Semantically, can only apply an operator when it's in the task network and has no predecessors
  - Add task literal to precondition of operator

Operator: $brew(i)$

$pre = \{brew(i),$
$running(i) \dots\}$
$eff = \{\neg brew(i),$
$\neg running(i),$
$finished(i) \dots\}$

Translating Operators



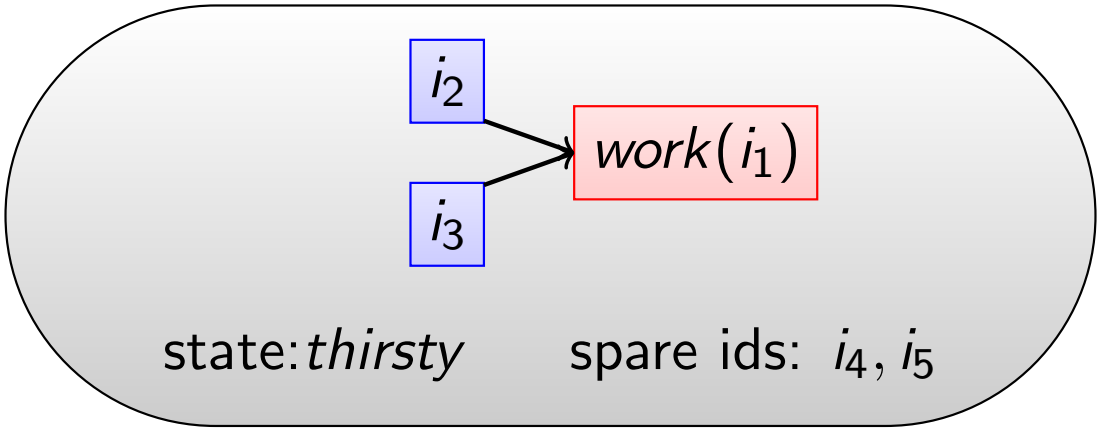
Execute *write*: $\gamma(\neg thirsty, write(i_3)) = thirsty$

- For operators
  - Syntactically the same in HTN and classical planning
  - Semantically, can only apply an operator when it's in the task network and has no predecessors
  - Add task literal to precondition of operator

Operator: $brew(i)$

$pre = \{brew(i),$
$running(i) \dots\}$
$eff = \{\neg brew(i),$
$\neg running(i),$
$finished(i) \dots\}$

Translating Operators

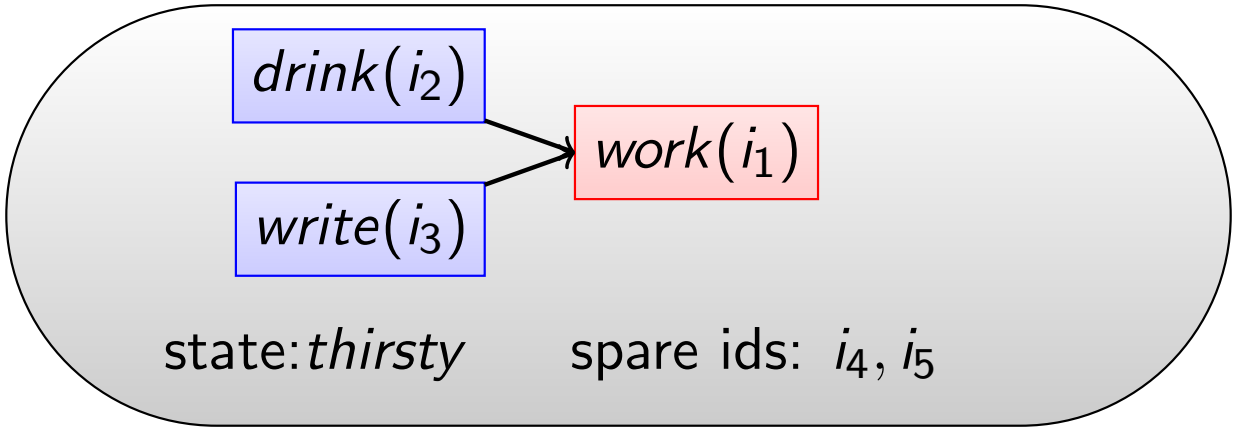


- For operators
  - Syntactically the same in HTN and classical planning
  - Semantically, can only apply an operator when it's in the task network and has no predecessors
  - Add task literal to precondition of operator

Operator: $brew(i)$

$pre = \{brew(i),$
$running(i) \dots\}$
$eff = \{\neg brew(i),$
$\neg running(i),$
$finished(i) \dots\}$

Managing precedence constraints



Manage precedence constraints with operators:

- Wait for tasks to be finished
- Free the identifiers
- Mark task as ready to run

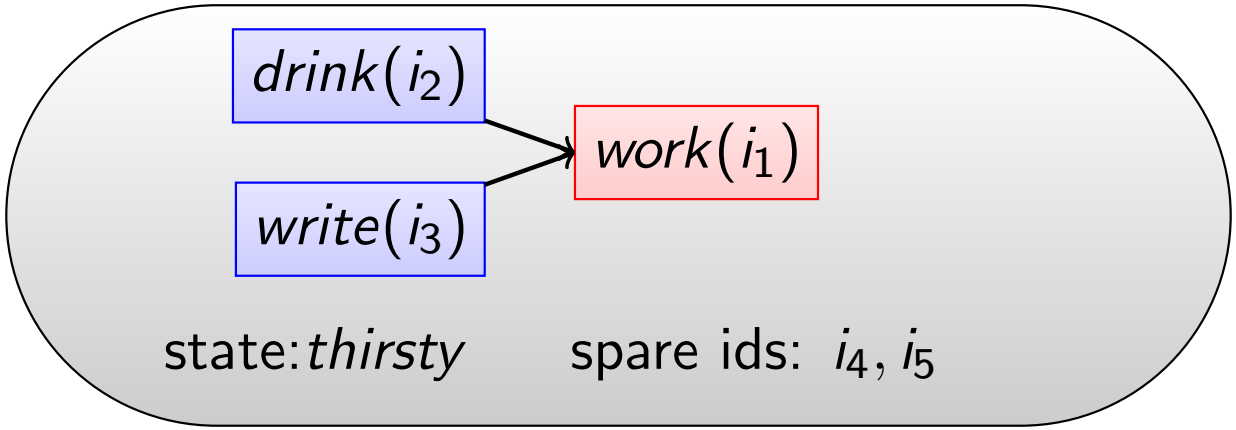
Wait op: $wait_{22}(i_1, i_2, i_3)$

$\text{pre} = \{finished(i_2), finished(i_3), \ldots\}$

$\text{eff} = \{\texttt{free } i_2 \texttt{ and } i_3$

$running(i_1)\}$

Managing precedence constraints



Execute *drink*: $\gamma(thirsty, drink(i_2)) = \neg thirsty$

Manage precedence constraints with operators:

- Wait for tasks to be finished
- Free the identifiers
- Mark task as ready to run

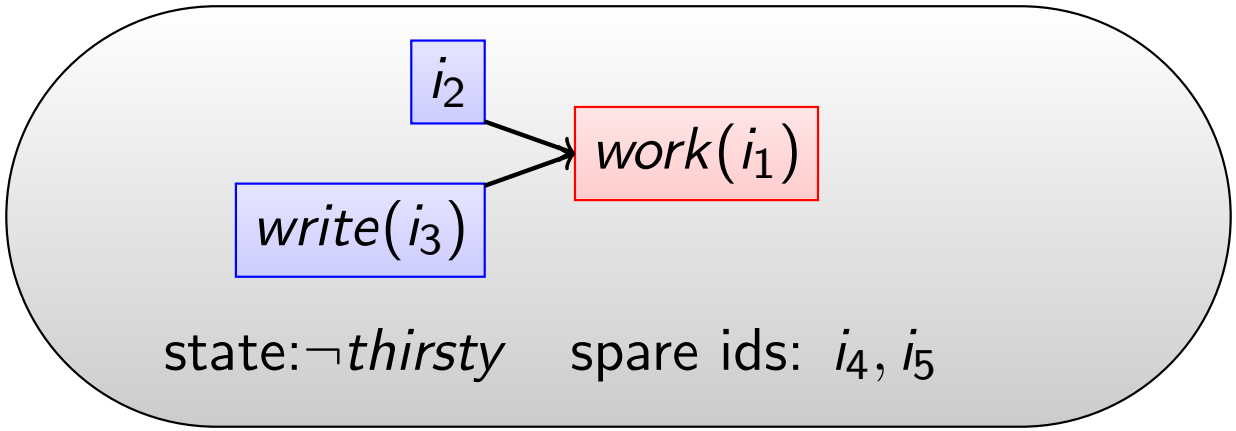
Wait op: $wait_{22}(i_1, i_2, i_3)$

$\text{pre} = \{finished(i_2), finished(i_3), \ldots\}$

$\text{eff} = \{\texttt{free } i_2 \texttt{ and } i_3$

$running(i_1)\}$

Managing precedence constraints



Manage precedence constraints with operators:

- Wait for tasks to be finished
- Free the identifiers
- Mark task as ready to run

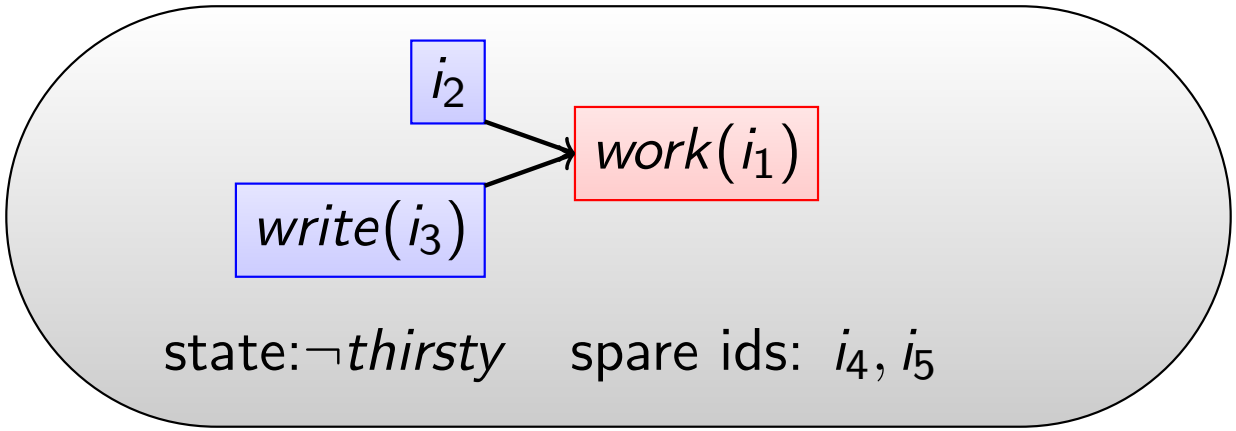
Wait op: $wait_{22}(i_1, i_2, i_3)$

$\text{pre} = \{finished(i_2), finished(i_3), \ldots\}$

$\text{eff} = \{\texttt{free } i_2 \texttt{ and } i_3$

$running(i_1)\}$

Managing precedence constraints



Execute *write*: $\gamma(\neg thirsty, write(i_3)) = thirsty$

Manage precedence constraints with operators:

- Wait for tasks to be finished
- Free the identifiers
- Mark task as ready to run

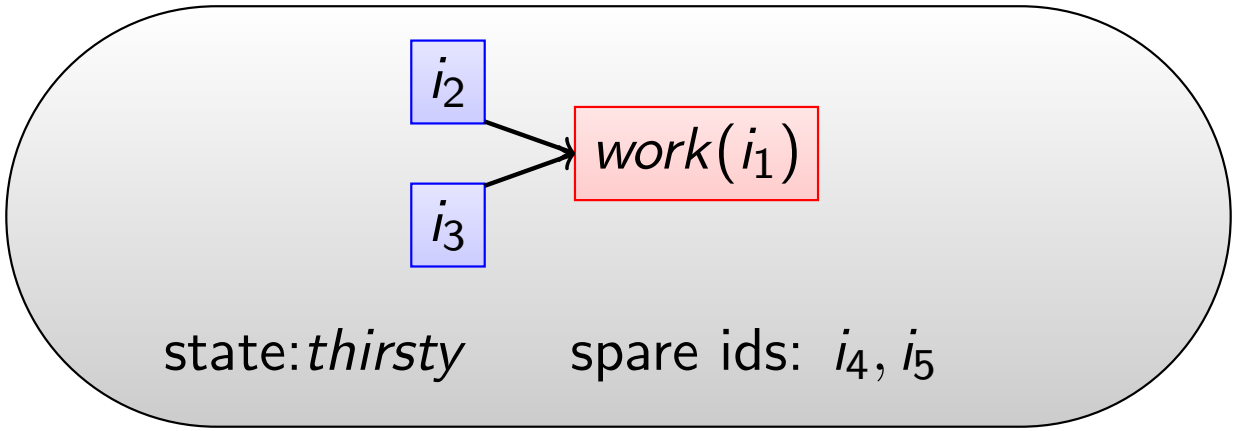
Wait op: $wait_{22}(i_1, i_2, i_3)$

$\text{pre} = \{finished(i_2), finished(i_3), \ldots\}$

$\text{eff} = \{\texttt{free } i_2 \texttt{ and } i_3$

$running(i_1)\}$

Managing precedence constraints



Manage precedence constraints with operators:

- Wait for tasks to be finished
- Free the identifiers
- Mark task as ready to run

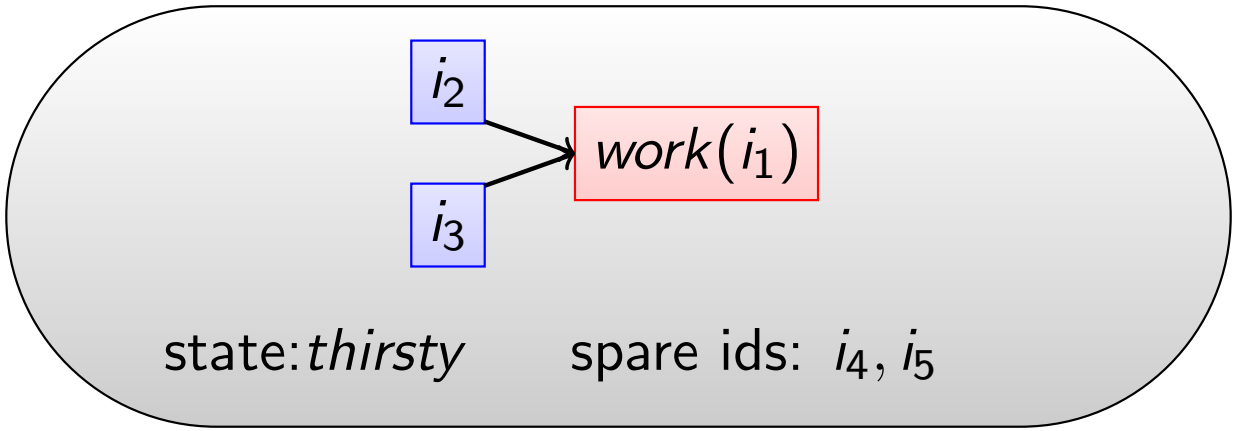
Wait op: $wait_{22}(i_1, i_2, i_3)$

$\text{pre} = \{finished(i_2), finished(i_3), \ldots\}$

$\text{eff} = \{\texttt{free } i_2 \texttt{ and } i_3$

$running(i_1)\}$

Managing precedence constraints



Remove constraints with $wait_{22}(i_1, i_2, i_3)$

Manage precedence constraints with operators:

- Wait for tasks to be finished
- Free the identifiers
- Mark task as ready to run

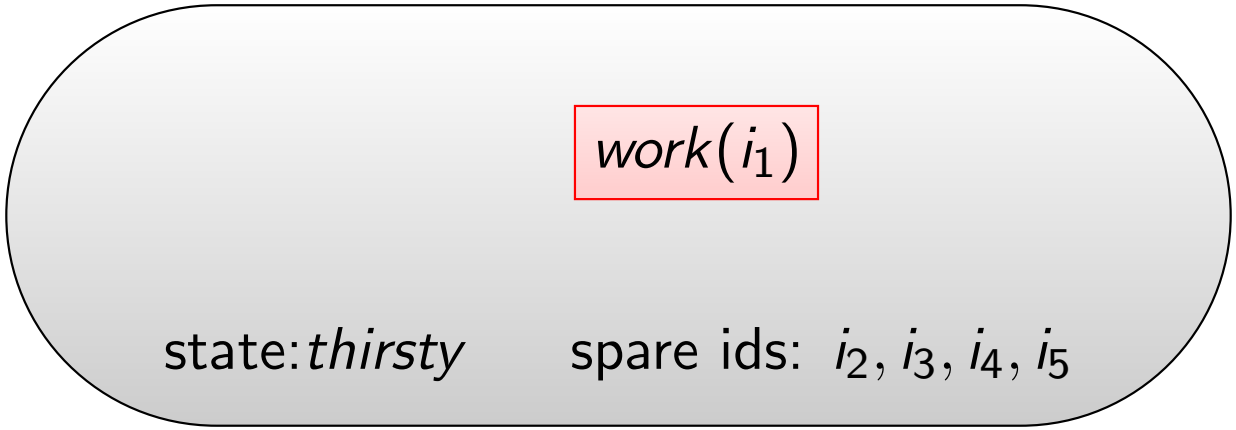
Wait op: $wait_{22}(i_1, i_2, i_3)$

$\text{pre} = \{finished(i_2), finished(i_3), \ldots\}$

$\text{eff} = \{\texttt{free } i_2 \texttt{ and } i_3$

$running(i_1)\}$

Managing precedence constraints



Manage precedence constraints with operators:

- Wait for tasks to be finished
- Free the identifiers
- Mark task as ready to run

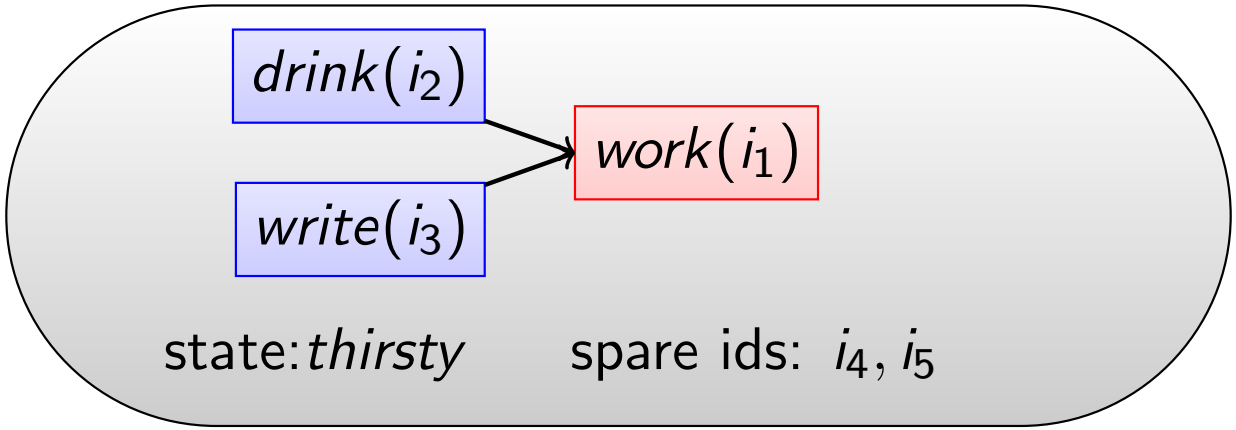
Wait op: $wait_{22}(i_1, i_2, i_3)$

$\text{pre} = \{finished(i_2), finished(i_3), \ldots\}$

$\text{eff} = \{\texttt{free } i_2 \texttt{ and } i_3$

$running(i_1)\}$

Translating Methods: Overview



When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

$\text{pre} = \{work(i_1), running(i_1)\}$

$\text{eff} = \{$`allocate` i_1, i_2

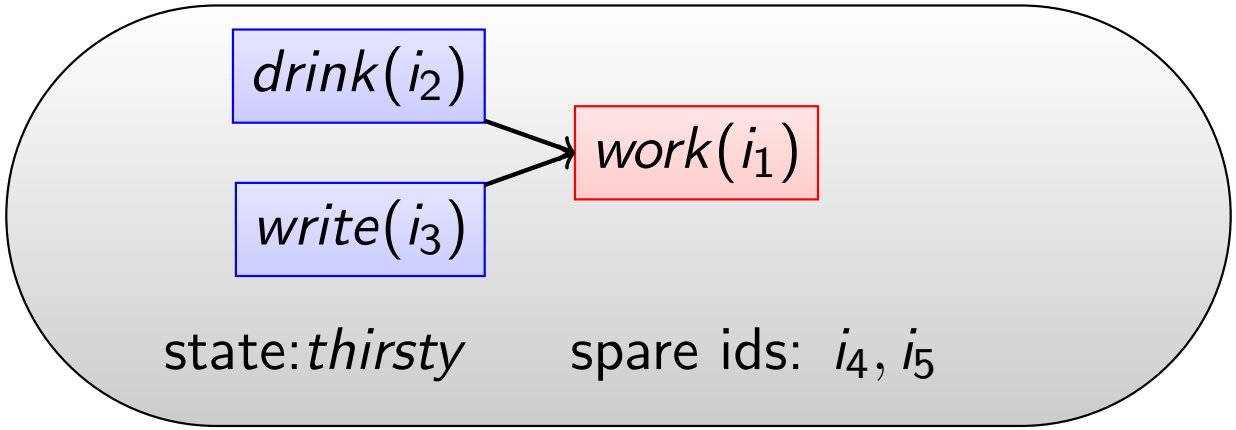
$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3)$,

$running(i_2), running(i_3)$

$running(i_\cdot)\}$

Translating Methods: Overview



Execute *drink*(i_2)

When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: *work*(i_1, i_2, i_3)

pre $= \{$*work*(i_1), *running*(i_1)$\}$

eff $= \{$`allocate` i_1, i_2

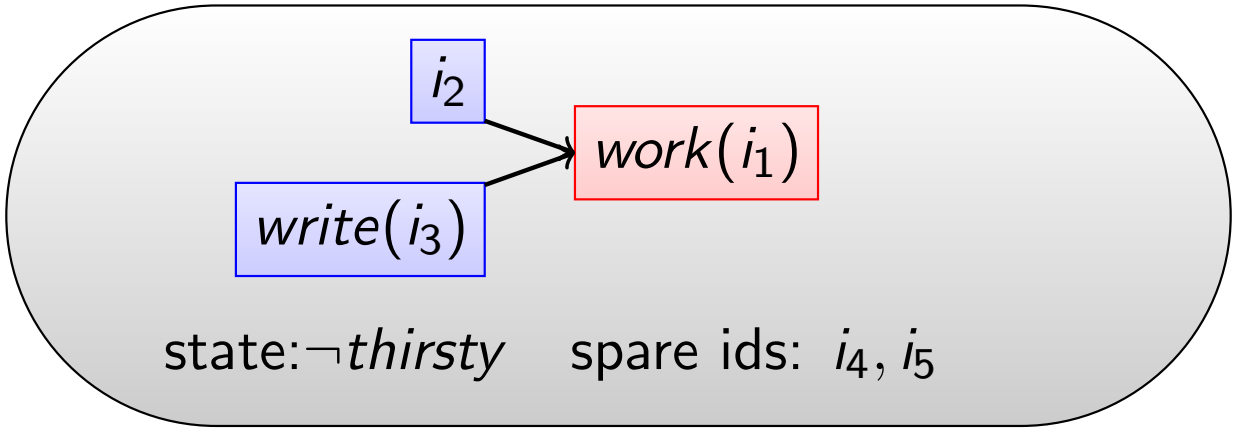
drink(i_2), *brew*(i_3), *work*(i)

wait$_{22}$(i, i_2, i_3),

running(i_2), *running*(i_3)

running(i)$\}$

Translating Methods: Overview



When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

pre $= \{work(i_1), running(i_1)\}$

eff $= \{$`allocate` i_1, i_2

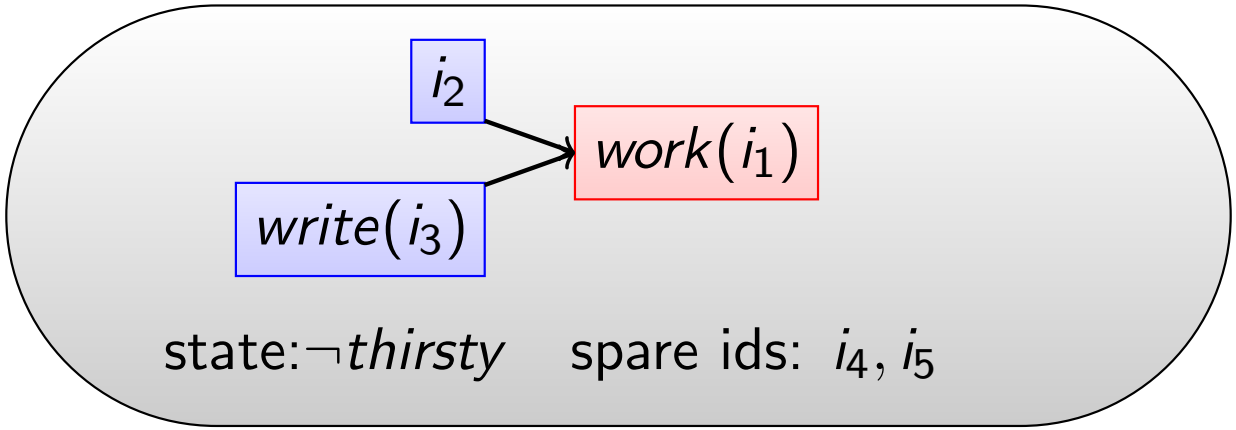
$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3)$,

$running(i_2), running(i_3)$

$running(i_4)\}$

Translating Methods: Overview



Execute $write(i_3)$

When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

pre $= \{work(i_1), running(i_1)\}$

eff $= \{$`allocate` i_1, i_2

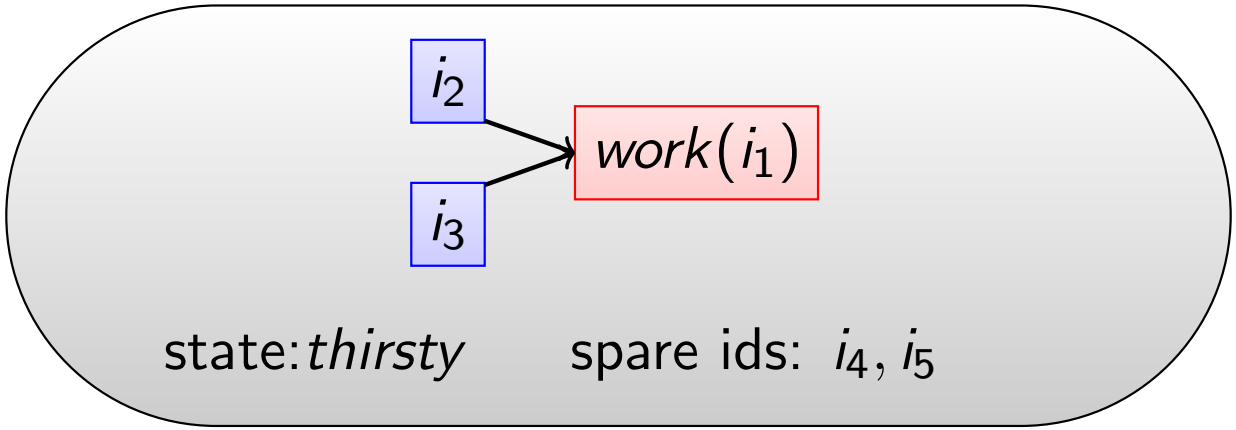
$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3)$,

$running(i_2), running(i_3)$

$running(i_1)\}$

Translating Methods: Overview



When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

$pre = \{work(i_1), running(i_1)\}$

$eff = \{$`allocate` i_1, i_2

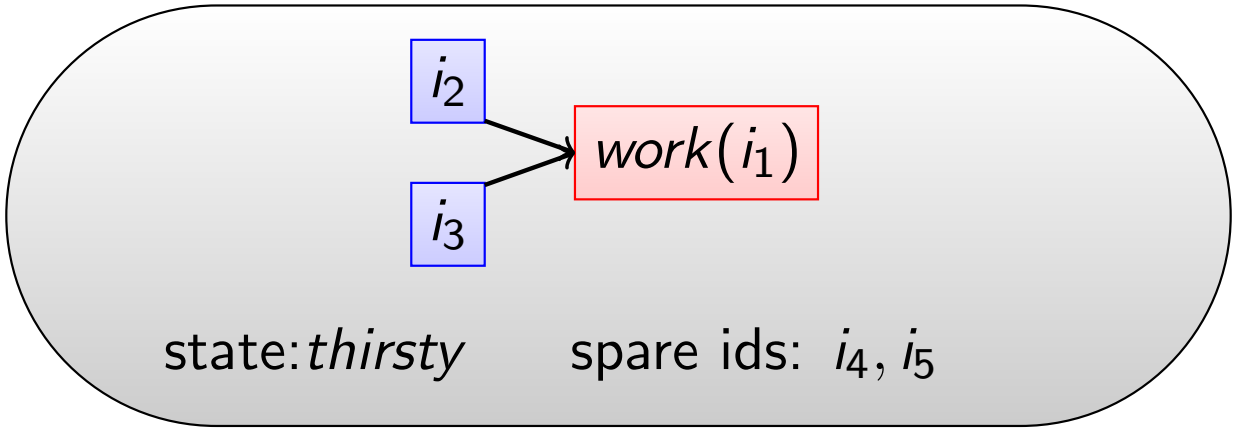
$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3),$

$running(i_2), running(i_3)$

$running(i_3)\}$

Translating Methods: Overview



Execute $wait_{22}$

When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

$\text{pre} = \{work(i_1), running(i_1)\}$

$\text{eff} = \{\texttt{allocate } i_1, i_2$

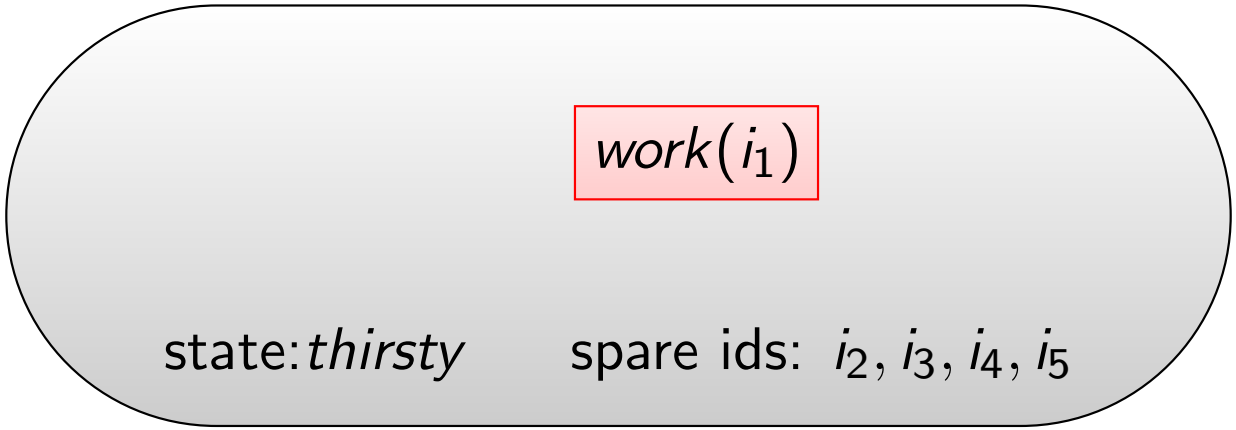
$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3),$

$running(i_2), running(i_3)$

$running(i_1)\}$

Translating Methods: Overview



When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

$\text{pre} = \{work(i_1), running(i_1)\}$

$\text{eff} = \{\texttt{allocate}\ i_1, i_2$

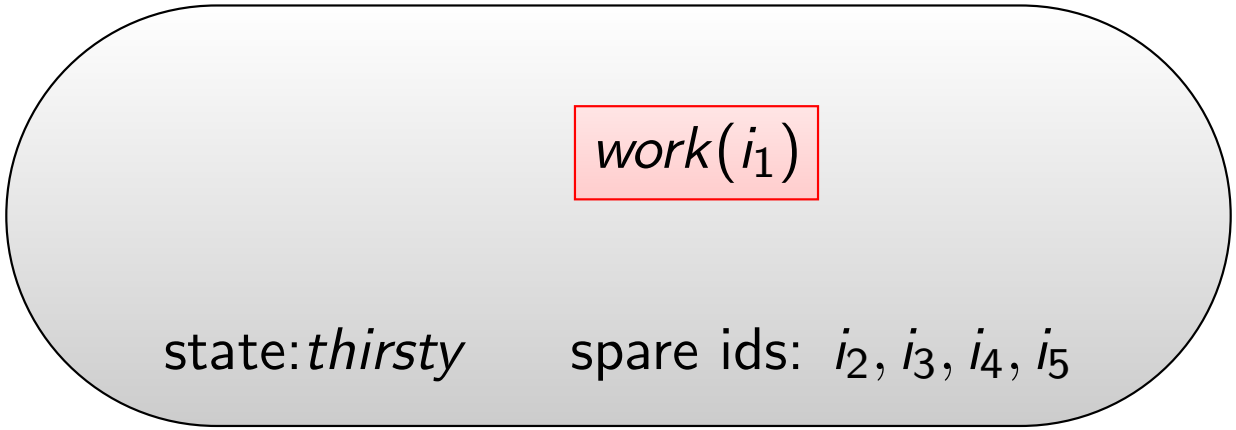
$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3),$

$running(i_2), running(i_3)$

$running(i_\cdot)\}$

Translating Methods: Overview



Execute $work(i_1, i_2, i_3)$

When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

$\text{pre} = \{work(i_1), running(i_1)\}$

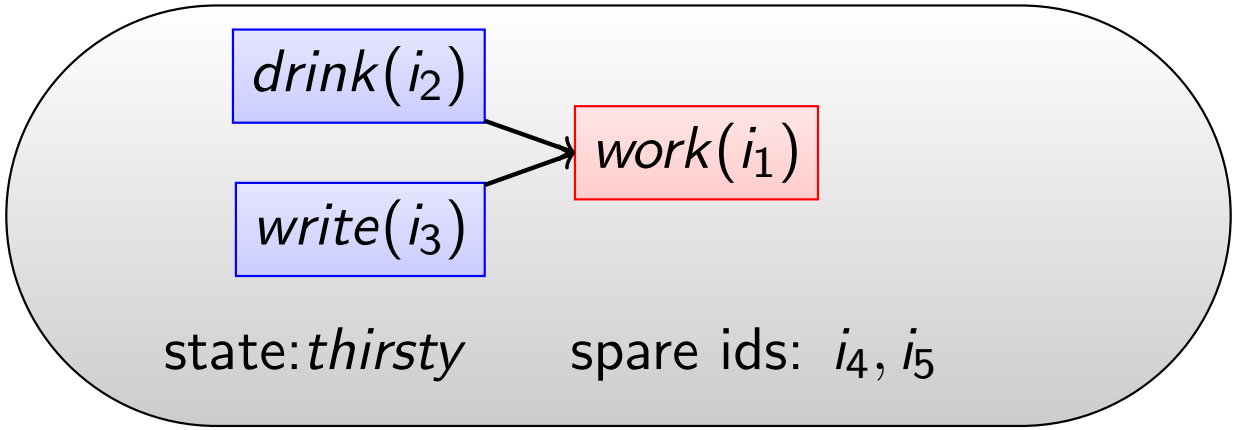
$\text{eff} = \{\texttt{allocate}\ i_1, i_2$

$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3),$

$running(i_2), running(i_3)$

Translating Methods: Overview



When translating a method (> 2 tasks):

- Allocate identifiers
- Assert task network into state
  - Reuse task identifier for last task
- Mark first tasks as ready to run

Method op: $work(i_1, i_2, i_3)$

$\text{pre} = \{work(i_1), running(i_1)\}$

$\text{eff} = \{$`allocate` i_1, i_2

$drink(i_2), brew(i_3), work(i)$

$wait_{22}(i, i_2, i_3)$,

$running(i_2), running(i_3)$

$running(i_\cdot)\}$