



Tabling for Planning

Modeling and Solving Planning Problems With Picat

Neng-Fa Zhou

(CUNY Brooklyn College & GC)

With contributions by Roman Bartak, Agostino Dovier,
Hakan Kjellerstrand, and Jonathan Fruhman

ECAI'14 Tutorial, Prague

8/18/2014



Outline of Tutorial

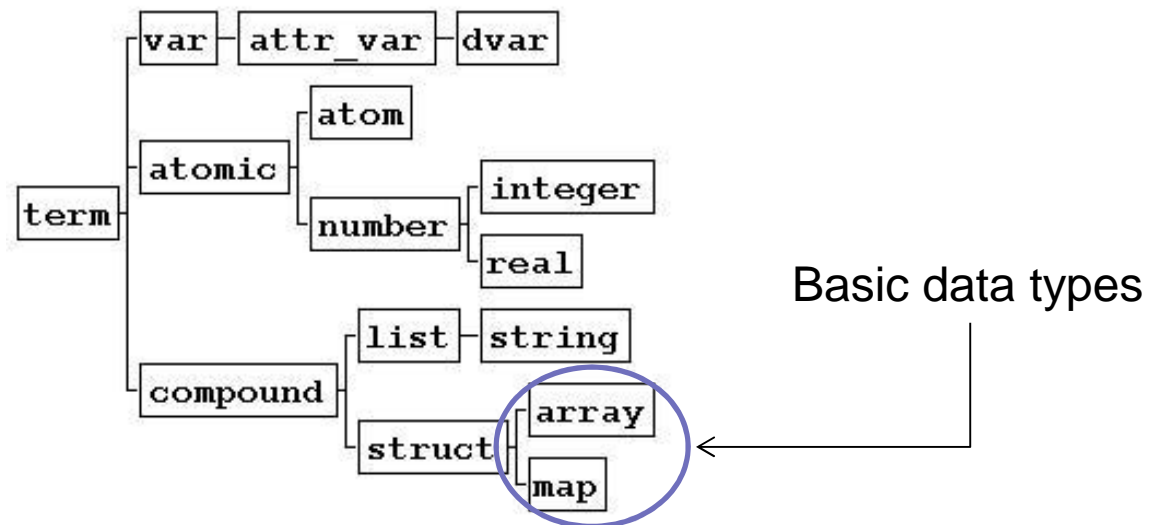
- An overview of Picat [20m]
- Tabling in Picat [10m]
- Planning with Picat [50m]
 - The planner module of Picat
 - Modeling techniques
 - Modeling examples
- Summary

An Overview of Picat



- Why the name “PICAT”?
 - Pattern-matching, Imperative, Constraints, Actors, Tabling
- Core logic programming concepts (Prolog-like)
 - Logic variables (arrays and maps are terms)
 - Implicit pattern-matching and explicit unification
 - Explicit non-determinism
- Language constructs for scripting
 - Functions, loops, and list comprehension
- Modeling and solving
 - CP, SAT, and MIP for constraint solving
 - Tabling for dynamic programming and planning

Logic Variables



```
Picat> A = new_array(2,3), A[1,1] = 1, A[2,3] = 5  
A = {{1, _3d4, _3d8}, {_3e0, _3e4, 5}}
```

```
Picat> M = new_map([alpha=1, beta=2]), M.get(alpha) = A  
M = (map)[alpha = 1,beta = 2]  
A = 1
```



Pre-created Maps

(Picat has no assert/retract)

- `Map = get_heap_map()`
 - Stored on the heap.
 - Like a normal map, updates are undone on backtracking.
- `Map = get_global_map()`
 - Stored in the global area.
 - Updates are not undone on backtracking.
- `Map = get_table_map()`
 - Stored in the table area; keys and values are hash-consed
 - Updates are not undone on backtracking.




Pattern Matching

Head, Cond => Body.

■ Pattern-matching rules

- Pattern-matching is adequate for many applications
 - NLP, state-space search, ...
- Pattern-matching rules are fully indexed
 - Picat can be more scalable than Prolog
- Explicit unification ($X=Y$) is supported
- As-pattern

```
is_sorted([X|L@[Y|_]]) => X =< Y, is_sorted(L).
```



Explicit Non-determinism

$\text{member}(X, [Y|_]) \text{ ?}=> X=Y.$
 $\text{member}(X, [_|L]) => \text{member}(X, L).$

■ Explicit non-determinism

- Although non-determinism is helpful for many computations, deterministic computations are the majority
- The cut operator is unneeded
- Use the `once` operator to remove choice points



Functions

$f(A_1, \dots, A_n) = \text{Exp}, \text{Cond} \Rightarrow \text{Body}.$

```
fib(N)=1, N=<1 => true.  
fib(N)=fib(N-1)+fib(N-2).
```

- Many logic languages support functions
 - Curry, Erlang, Mozart-OZ,...
- Functions are easier to use than relations
 - Function calls never fail (at least built-in functions)
 - Function calls can be nested
 - Directionality helps readability
- Special notation is needed for structures

```
Picat> S = $student(mary,cs,3.8)
```




Loops

```
foreach( $E_1$  in  $D_1$ ,  $Cond_1$  , . . . ,  $E_n$  in  $D_n$ ,  $Cond_n$ )  
    Goal  
end
```

- Loops are convenient for scripting and modeling purposes
 - Lisp, Python, C#, Java, C++11, ...
- Loops are compiled to tail-recursion



Loops

■ Scopes of variables in loops

- *Variables that occur within a loop but not before in its outer scope are local to each iteration*

```
p(A) =>  
  q(X),  
  foreach(I in 1 .. A.length)  
    A[I] = (X,Y,Y,_)  
end.
```

X is global, and Y is local.

The anonymous variable is always local.



List Comprehension

$[T : E_1 \text{ in } D_1, \text{Cond}_1, \dots, E_n \text{ in } D_n, \text{Cond}_n]$

- Convenient for constructing lists
 - Supported by more than 30 languages, according to Wiki.
- Compiled to a foreach loop
 - The assignment operator ($:=$) is used to accumulate values



Picat = Python + Pattern Matching + Nondeterminism + ...

```
power_set([]) = [[]].  
power_set([H|T]) = P1++P2 =>  
    P1 = power_set(T),  
    P2 = [[H|S] : S in P1].
```

```
perm([]) = [[]].  
perm(Lst) = [[E|P] : E in Lst, P in perm(Lst.delete(E))].
```

```
matrix_multi(A,B) = C =>  
    C = new_array(A.length,B[1].length),  
    foreach(I in 1..A.length, J in 1..B[1].length)  
        C[I,J] = sum([A[I,K]*B[K,J] : K in 1..A[1].length])  
    end.
```



Outline of Tutorial

- An overview of Picat [20m]
- Tabling in Picat [10m]
- Planning with Picat [50m]
 - The planner module of Picat
 - Modeling techniques
 - Modeling examples
- Summary



Tabling

for Dynamic Programming

- The idea [Michie68, Tamaki&Sato86, Warren92]
 - Tabling memorizes calls and their answers in order to prevent infinite loops and to limit redundancy
- Tabling is exciting
 - Computers have more and more memory
 - Advanced implementation techniques have improved the scalability
- Tabling approaches
 - Suspension-based tabling and linear tabling



Tabling-All

```
table
fib(0)=1.
fib(1)=1.
fib(N)=fib(N-1)+fib(N-2).
```

```
table
reach(X,Y) ?=> edge(X,Y).
reach(X,Y) => reach(X,Z),edge(Z,Y).
```



Mode-directed Tabling in Picat

```
table(+,+,-,min)
shortest_path(X,Y,Path,W) ?=>
    Path = [(X,Y)],
    edge(X,Y,W),
shortest_path(X,Y,Path,W) =>
    Path = [(X,Z)|PathR],
    edge(X,Z,W1),
    shortest_path(Z,Y,PathR,W2),
    W = W1+W2.
```

■ Table modes

- + (input), - (output), min, max, nt (not-tabled)

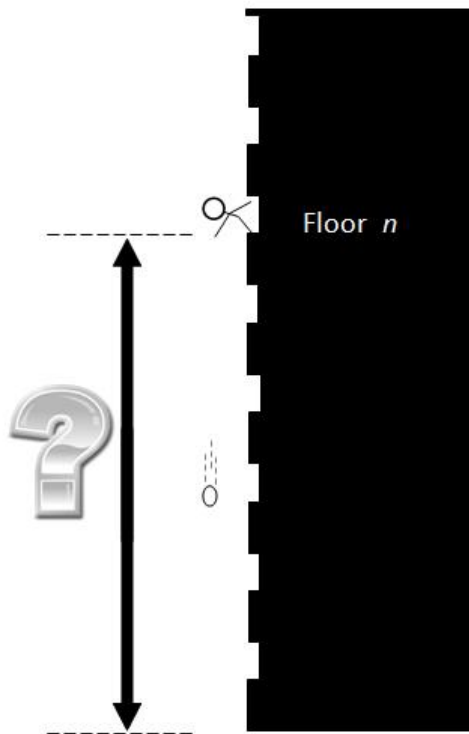
■ Semantics

- Table the best answer for each tuple of input arguments

■ Linear tabling

- Iteratively evaluate looping calls until the optimum is reached

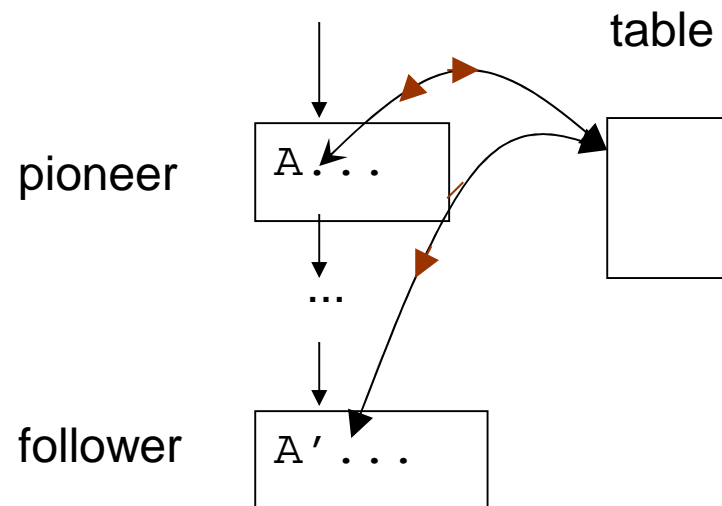
The Omelet Problem (Or The N-Eggs Problem)



```
table (+,+,min)
omelet(_,0,NTries) => NTries=0.
omelet(_,1,NTries) => NTries=1.
omelet(1,H,NTries) => NTries=H.
omelet(N,H,NTries) =>
    between(1,H,L),           % make a choice
    omelet(N-1,L-1,NTries1),  % the egg breaks
    omelet(N,H-L,NTries2),    % the egg survives
    NTries is max(NTries1,NTries2)+1.
```

<http://www.datagenetics.com/blog/july22012/>

Linear Tabling



A' , a variant of A , fails.

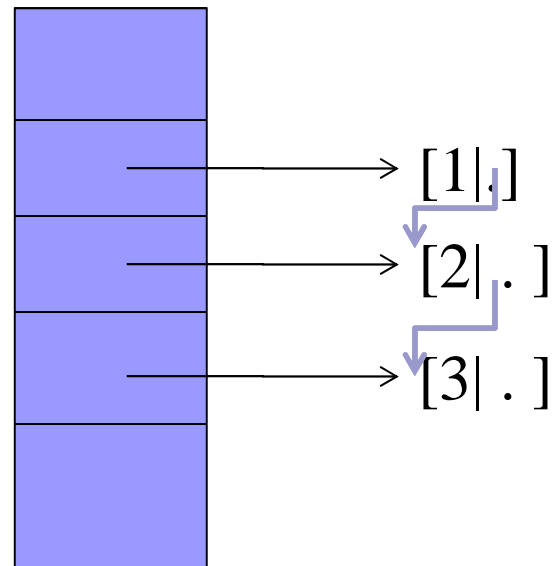
A needs to be re-evaluated in some cases.

Term Sharing (Hash-consing)

[1, 2, 3]
[2, 3]
[3]

Hash codes are tabled, so it's fast
to test if two states are the same.

Hashtable





Tabled Planning

```
table (+,-,min)
plan(S,Plan,Cost),final(S) => Plan=[],Cost=0.
plan(S,Plan,Cost) =>
    action(S,S1,Action,ActionCost),
    plan(S1,Plan1,Cost1),
    Plan = [Action|Plan1],
    Cost = Cost1+ActionCost.
```

- With tabling, state-space tree search becomes state-space graph search
- Depth-unbounded search



Tabled Planning

Depth-bounded Search

```
table (+,-,min,+)  
plan(S,Plan,Cost,Limit),final(S) => Plan=[],Cost=0.  
plan(S,Plan,Cost,Limit) =>  
    action(S,S1,Action,ActionCost),  
    Limit1 = Limit-ActionCost,  
    Limit1 >= 0,  
    plan(S1,Plan1,Cost1,Limit1),  
    Plan = [Action|Plan1],  
    Cost = Cost1+ActionCost.
```

- Pass the resource limit as an input argument
 - Calls with the same state and different resource limits are no longer variants



Tabled Planning

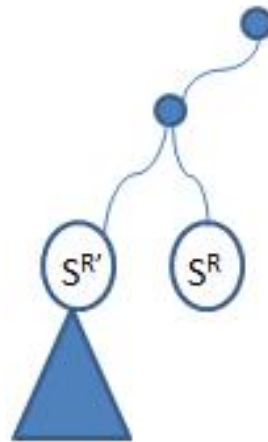
Depth-bounded Search

```
table (+,-,min,nt)
plan(S,Plan,Cost,Limit),final(S) => Plan=[],Cost=0.
plan(S,Plan,Cost,Limit) =>
    action(S,S1,Action,ActionCost),
    Limit1 = Limit-ActionCost,
    Limit1 >= 0,
    plan(S1,Plan1,Cost1,Limit1),
    Plan = [Action|Plan1],
    Cost = Cost1+ActionCost.
```

- Pass the resource limit as an *nt* argument
 - Once a call is completed with a failure, it will fail forever, no matter how big the resource limit is.
 - Soundness and completeness are not guaranteed.

Resource-Bounded Search

- Special treatment of the resource limit argument
 - It is tabled but not used in variant checking



S^R is the current node, where S is the state and R is the resource limit. $S^{R'}$ failed before. S^R can be failed immediately if $R \leq R'$.



Outline of Tutorial

- An overview of Picat [20m]
- Tabling in Picat [10m]
- Planning with Picat [50m]
 - The planner module of Picat
 - Modeling techniques
 - Modeling examples
- Summary



Classical Planning

- $P = (S, \Sigma, f, \delta, s_0, F)$
 - S : A set of states (finite or countably infinite)
 - Σ : A set of actions
 - f : A transition function or relation ($S \times \Sigma \rightarrow S$)
 - δ : A cost function ($S \times \Sigma \rightarrow \mathcal{R}$)
 - s_0 : An initial state
 - F : A set of goal states



Planning Formalisms

- Logic programming
 - PLANNER [Hewitt69], “*a language for proving theorems and manipulating models in a robot*”
 - Prolog for planning [Kowalski79,Warplan76]
 - ASP-based planners [Lifschitz02]
- STRIPS-based PDDL
 - The de facto language [McDermott98]
 - Many solvers (Arvand, LAMA, FD, SymBA*-2,...)
 - Extensions of PDDL (e.g., HTN)
- Planning as SAT and model checking



Planning With Picat

- A logic programming approach
 - Unlike PDDL and ASP, structured data can be used.
 - Domain-specific heuristics and control knowledge about determinism, dependency, and symmetry can be encoded.
- Tabled backtracking search
 - Every state generated during search is tabled.
 - Same idea as state-marking used in IDA* and other algorithms.
 - Term sharing: common ground terms are tabled only once.
 - Alleviate the *state explosion problem*.
 - Resource-bounded search
 - Unlike IDA*, results from previous rounds are reused.



Picat's planner Module

- Resource-bounded search
 - `plan(State,Limit,Plan,PlanCost)`
 - `best_plan(State,Limit,Plan,PlanCost)`
 - Iterative deepening (unlike IDA*, results from early rounds are reused)
 - `current_resource()`
 - `current_plan()`
- Depth-unbounded search
 - `plan_unbounded(State,Limit,Plan,PlanCost)`
 - `best_plan_unbounded(State,Limit,Plan,PlanCost)`
 - Like Dijkstra's algorithm



How to Use the Planner?

- Import the planner module
- Specify the goal states
 - `final(State)`
 - True if State is a goal state.
 - `final(State,Plan,Cost)`
 - True if a goal state can be reached from State by Plan with Cost.
- Specify the actions
 - `action(State,NextState,Action,ActionCost)`
 - Encodes the state transition relation
 - States are tabled, and destructive updates of states (using `:=`) are banned.
- Call a built-in on an initial state to find a plan



Ex: The Farmer's Problem

```
import planner.  
  
go =>  
    S0=[s,s,s,s],  
    best_plan(S0,Plan),  
    writeln(Plan).  
  
final([n,n,n,n]) => true.  
  
action([F,F,G,C],S1,Action,ActionCost) ?=>  
    Action=farmer_wolf,  
    ActionCost = 1,  
    opposite(F,F1),  
    S1=[F1,F1,G,C],  
    not unsafe(S1).  
...
```

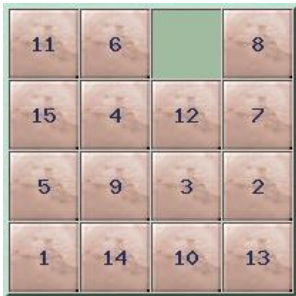


Modeling Techniques

- Find a good representation for states
 - Keep the information minimal.
 - Use good data structures that facilitate
 - sharing
 - computation of heuristics
 - symmetry breaking
- Use heuristics and domain knowledge
 - A state should not be expanded if the travel from it to the final state costs more than the limit .
 - Identify deterministic actions and macro actions.
 - Use landmarks.

Modeling Examples

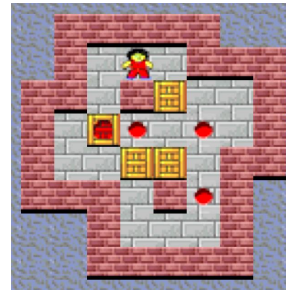
picat-lang.org/projects.html



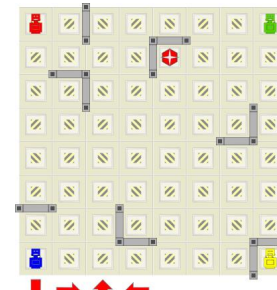
15-puzzle



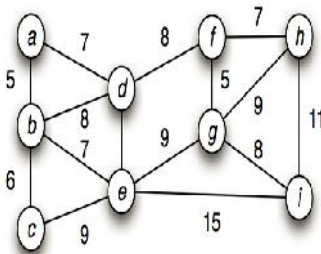
RushHour



Sokoban



Ricochet Robots



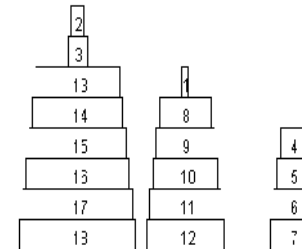
Logistics

$\langle 5\spadesuit, 3\heartsuit, Q\clubsuit, 8\diamondsuit \rangle,$
 $\langle K\spadesuit, 2\heartsuit, 7\clubsuit, 4\diamondsuit \rangle,$
 $\langle 8\spadesuit, J\heartsuit, 9\clubsuit, A\diamondsuit \rangle$

Gilbreath's card trick



Rubik's Cube



Tower-of-Hanoi

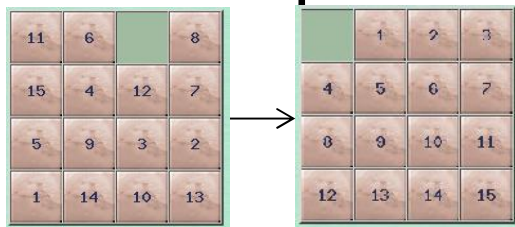


Setting for the Experiments

- Linux
- CPU 4-core AMD II X4 945
- 8GM RAM
- Timeout was set to 15 minutes per instance

15-Puzzle

■ State representation



```
State = {SpaceLoc, CurTiles, GoalTiles},  
SpaceLoc = [1|3],  
CurTiles = [[4|1],[3|4],...],  
GoalTiles= [[1|2],[1|3],...]
```

Use a cons to represent a location.

A cons [Row|Col] consumes 2 words,
and a pair (Row,Col) consumes 3 words.

■ The goal state

```
final([1|1],Tiles,Tiles) => true.
```

15-Puzzle

■ Actions

```
action({P0@[R0|C0],Tiles,GTiles},NextS,Move,Cost) ?=>
  R1 = R0-1,
  R1 >= 1,
  Move = up,
  Cost = 1,
  P1 = [R1|C0],
  update(Tiles,P0,P1,NTiles),
  manhattan_distance(NTiles,GTiles) < current_resource(),
  NextS={P1,NTiles,GTiles}.
```

...

NOTE: Tabled data cannot be destructively updated using the := operator!



15-Puzzle

■ Experimental Results

- 15 instances from ASP'09 that require 30-45 steps were used.
- Picat solved all 15 instances in less than 1s per instance.
 - The Manhattan distance heuristic is important.
- Prolog using the same heuristic is over 100 times slower than Picat
 - Tabling is important.
- Potassco (Clasp) failed to solve 5 of the 15 instances.

Rush Hour Puzzle



Move the red car to the exit (4,2).



Rush Hour Puzzle

■ State representation

`{RedLoc, L11, L12, L21, L13, L31}`

- L11 -- an ordered list of locations of the spaces.
- Lwh -- an ordered list of locations of the $w \times h$ cars.
- Symmetries are removed.

■ Goal states

`final({[4|2],_,_,_,_,_}) => true.`



Rush Hour Puzzle

■ Actions

```
% move the red car
action({LocRed,L11,L12,L21,L13,L31},NewS,Action,Cost) ?=>
    Cost=1,
    move_car(2,1,LocRed,NLocRed,L11,NL11,Action),
    NewS = {NLocRed,NL11,L12,L21,L13,L31}.

% move a 1*2 car
action({LocRed,L11,L12,L21,L13,L31},NewS,Action,Cost) ?=>
    Cost=1,
    select(Loc,L12,L12R),
    move_car(1,2,Loc,NLoc,L11,NL11,Action),
    NL12 = L12R.insert_ordered(NLoc),
    NewS = {LocRed,NL11,NL12,L21,L13,L31}.

...
```

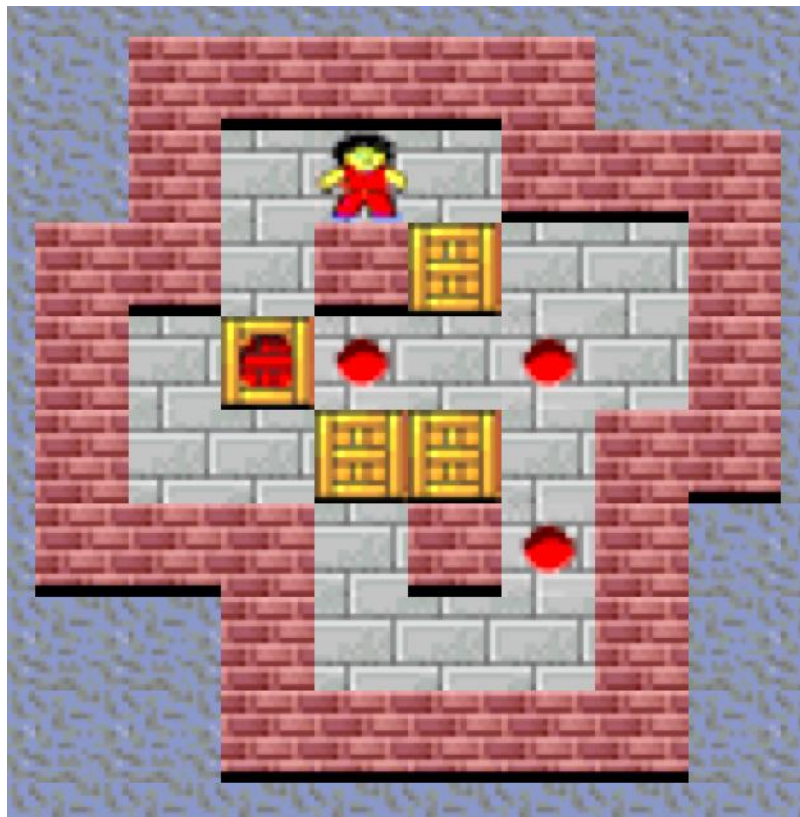


Rush Hour Puzzle

■ Experimental Results

- Picat found a best plan of 81 steps in 2s (explored 30493 states)
- ProB, with breadth-first search, took about the same amount of time to find a best plan.

Sokoban



In the ASP'13 version, there may be more stones than goal locations. This makes *reversed solving* difficult.

source: takaken



Sokoban

■ State representation

- { SoLoc , GStLocs , NonGStLocs }
- SoLoc – the location of the man.
- GStLocs – an ordered list of locations of the goal stones.
- NonGStLocs – an ordered list of locations of the non-goal stones.

■ Goal states

```
final({_,GStLocs,_}) =>
  foreach(Loc in GStLocs)
    goal(Loc)
end.
```

Sokoban

■ Actions

```
% push a goal stone
action( {SoLoc, GStLocs, NonGStLocs} , NextState, Action, Cost) ?=>
    NextState = {NewSoLoc, NewGStLocs, NonGStLocs} ,
    Action = $move_push( SoLoc, StLoc, StDest, Dir) ,
    Cost = 1,
    choose_goal_stone( Dir, SoLoc, NewSoLoc, GStLocs, StLoc,
                      StDest, GStLocs1, NonGStLocs) ,
    NewGStLocs = insert_ordered( GStLocs1, StDest) .
% push a non-goal stone
action( {SoLoc, GStLocs, NonGStLocs} , NextState, Action, Cost) ?=>
    ...
% Sokoban moves alone
action( {SoLoc, GStLocs, NonGStLocs} , NextState, Action, Cost) =>
    ...
```

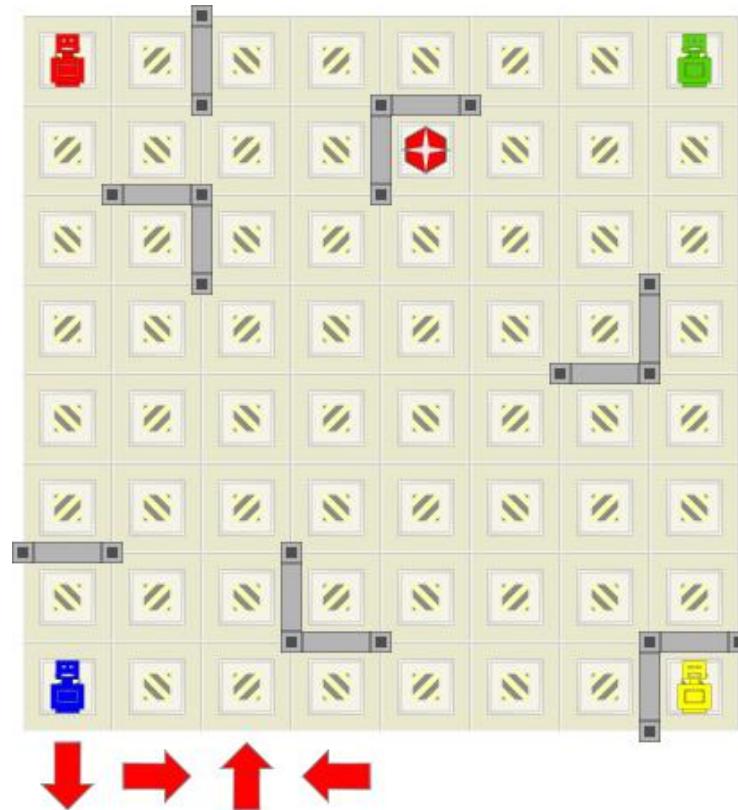


Sokoban

■ Experimental Results

- 30 instances from ASP'13 were used.
- Picat (using `plan_unbounded`) solved all the 30 instances (on average less than 1s per instance).
- Depth-unbounded search is faster than depth-bounded search.
- Potassco solved only 14 of the 30 instances.
- Not as competitive as Rolling Stone, a specialized Sokoban planner.

Ricochet Robots

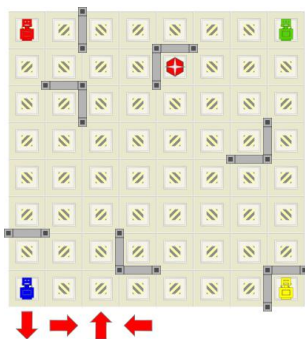


source:Martin Gebser et al.

Ricochet Robots

■ State representation

$\{ [\text{CurLoc} \mid \text{TargetLoc}], \text{ORobotLocs} \}$



$\{ [(1,1) \mid (2,5)], [(1,8), (8,1), (8,8)] \}$

Non-target robots are represented as an ordered list of locations. This representation breaks symmetries.

■ Goal states

$\text{final}(\{ [\text{Loc} \mid \text{Loc}], _ \}) \Rightarrow \text{true}.$



Ricochet Robots

■ Actions

```
action({[From|To],ORobotLocs},NextState,Action,Cost) ?=>
  NextState = {[Stop|To],ORobotLocs},
  Action = [From|Stop], Cost = 1,
  choose_move_dest(From,ORobotLocs,Stop).
action({FromTo@[From|_],ORobotLocs},NextState,Action,Cost) =>
  NextState = {FromTo,ORobotLocs2},
  Action = [RFrom|RTo], Cost = 1,
  select(RFrom, ORobotLocs,ORobotLocs1),
  choose_move_dest(RFrom,[From|ORobotLocs1],RTo),
  ORobotLocs2 = insert_ordered(ORobotLocs1,RTo).
```



Ricochet Robots

■ Use heuristics

```
action({[From|To],ORobotLocs},NextState,Action,Cost) ?=>
  NextState = {[Stop|To],ORobotLocs},
  Action = [From|Stop], Cost = 1,
  choose_move_dest(From,ORobotLocs,Stop),
  current_resource() > heuristic_val(NextState).
action({FromTo@[From|_],ORobotLocs},NextState,Action,Cost) =>
  NextState = {FromTo,ORobotLocs2},
  Action = [RFrom|RTo], Cost = 1,
  select(RFrom, ORobotLocs,ORobotLocs1),
  choose_move_dest(RFrom,[From|ORobotLocs1],RTo),
  ORobotLocs2 = insert_ordered(ORobotLocs1,RTo),
  current_resource() > heuristic_val(NextState).
```

- The current state is at least three steps away from the final state if the target robot is not in the same row or the same column, and the target position has no obstacle around it.



Ricochet Robots

■ Experimental results

- 30 instances of 16×16 used in ASP'13 were used.
- Picat solved all 30 instances
 - On average 9s per instance when no heuristic was used.
 - On average 2s per instance when the heuristic was used.
- Prolog struggles on 5×5 instances.
 - Tabling is important.
- Potassco also solved all 30 instances
 - On average 49s per instance.

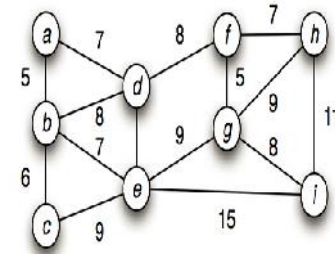


Logistics

- IPC domains
 - Nomystery
 - Airport pickup
 - Drivelog
 - Elevator planning
 - Petrobrass planning
 - ...

Nomystery

- There is only one truck involved.
- The truck has a fuel level.
- A number of packages need to be transported between nodes in a graph.
- The graph is weighted and the weight of an edge is the fuel cost.





Nomystery

■ State representation

□ { TruckLoc , LCGs , WCGs }

- LCGs – an ordered list of destinations of loaded cargoes
- WCGs – an ordered list of source-destination pairs of waiting cargoes

■ Goal states

```
final({_,[],[]}) => true.
```

Nomystery

■ Actions

```
action({Loc,LCGs,WCGs},NextState,Action,Cost),
  select(Loc,LCGs,LCGs1)
=>
  Action = $unload(Loc),
  Cost = 0,
  NextState= {Loc,LCGs1,WCGs}.
action({Loc,LCGs,WCGs},NextState,Action,Cost),
  select([Loc|CargoDest],WCGs,WCGs1)
=>
  Action = $load(Loc,CargoDest),
  Cost = 0,
  NextState = {Loc,LCGs1,WCGs1},
  LCGs1 = insert_ordered(LCGs,CargoDest).
action({Loc,LCGs,WCGs},NextState,Action,Cost) =>
  Action = $drive(Loc,Loc1),
  NextState = {Loc1,LCGs,WCGs},
  fuelcost(Cost,Loc,Loc1).
```

■ Domain knowledge

- If the truck is at the destination of a loaded cargo, then unload it *deterministically*.
- If the truck is at a location where there is a cargo that needs to be delivered, then load it *deterministically*.

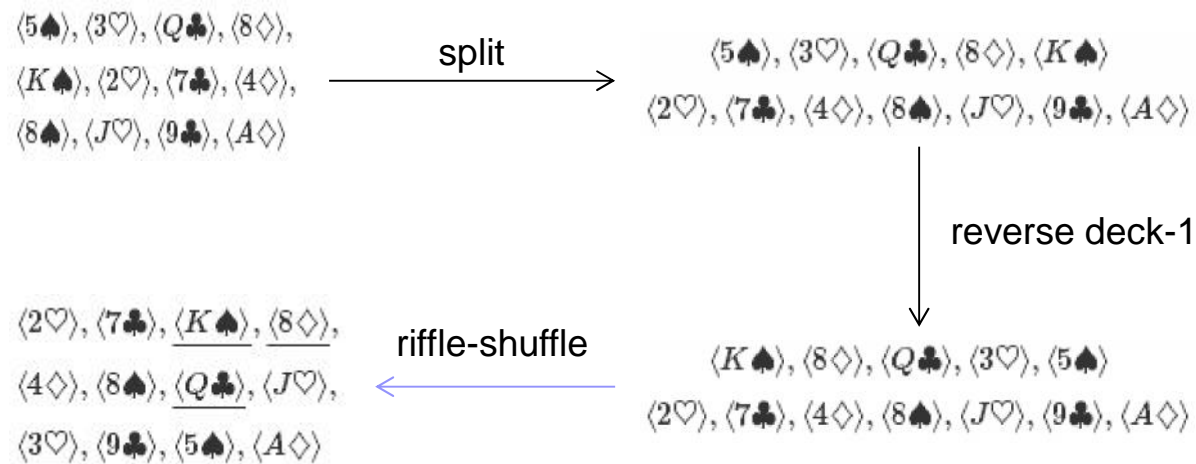


Nomystery

■ Experimental results

- 30 instances from ASP'13 were used.
- Picat solved all the 30 instances.
 - On average less than 0.1s per instance.
- Potassco solved only 17 of the 30 instances.
- Picat solved all the instances used in IPC'11, including the hardest instance that was not solved by any of the participating solvers.

Gilbreath's Card Trick



Each quartet contains a card from each suit

Take from "Unraveling a Card Trick", by Tony Hoare & Natarajan Shankar



Gilbreath's Card Trick

■ State representation

```
init([s,h,c,d,s,h,c,d,s,h,c,d])
```

```
splitted(Deck1,Deck2)
```

```
shuffled(Cards)
```

■ Goal states

```
final(shuffled([C1,C2,C3,C4,C5,C6,C7,C8|_])) =>  
  Suites = [c,d,h,s],  
  ( sort([C1,C2,C3,C4]) != Suites  
    ;  
    sort([C5,C6,C7,C8]) != Suites  
  ).
```




Gilbreath's Card Trick

■ Actions

```
action(init(Cards),NewS,Action,ActionCost) =>  
  NewS = $splitted(Deck1,RDeck2),  
  Action = split,  
  ActionCost = 1,  
  append(Deck1,Deck2,Cards),  
  Deck1 != [],  
  Deck2 != [],  
  RDeck2 = Deck2.reverse().
```

```
action(splitted(Deck1,Deck2),NewS,Action,ActionCost) =>  
  NewS = $shuffled(Cards),  
  Action = shuffle,  
  ActionCost = 1,  
  shuffle(Deck1,Deck2,Cards).
```



Gilbreath's Card Trick

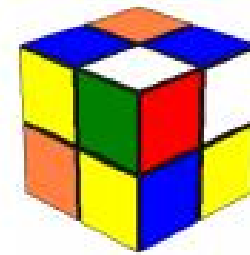
- Experimental results

#Cards	#States	Time
16	13,840	0.08s
20	165,908	1.43s
24	1,990,680	34.22s

Rubik's Cube



$12! \times 2^{12} \times 8! \times 3^8 = 43,252,003,274,489,856,000$
43 quintillion possible states!



$8! \times 3^7 = 88,179,840$



Rubik's Cube

■ State representation

`pieces(Es, Cs)`

`Es` : A list of positions of edge pieces.

Edge positions: `[bd, db, ..., ru, ur]`.

`Cs` : A list of positions of corner pieces.

Corner positions: `[bdl, bld, ..., ufr, urf]`

■ The goal state

`final(pieces(Es, Cs)) =>`

`Es = [bd, bl, br, bu, df, dl, dr, fl, fr, fu, lu, ru],`

`Cs = [bdl, bdr, blu, bru, dfl, dfr, flu, fru].`

Rubik's Cube

- Expand the goal state into a goal region

Depth	Nodes
1	18
2	243
3	3,240
4	43,254
5	577,368
6	7,706,988
7	102,876,480
8	1,373,243,544
9	18,330,699,168
10	244,686,773,808
11	3,266,193,870,720
12	43,598,688,377,184
13	581,975,750,199,168
14	7,768,485,393,179,328
15	103,697,388,221,736,960
16	1,384,201,395,738,071,424
17	18,476,969,736,848,122,368
18	246,639,261,965,462,754,048

```
final(S,Plan,Cost) =>  
  M = get_table_map(),  
  M.get(S,[]) = (Plan,Cost).
```

From Richard E. Korf'97



Rubik's Cube

■ Actions

```
action(S,NewS,Action,Cost) =>  
    current_resource_plan_cost(Limit,CurPlan,_CurPlanLen),  
    actions(Actions),  
    Cost = 1,  
    member(Action,Actions),  
    not nogood_action(CurPlan,Action),  
    transform(Action,S,NewS).
```

■ Some domain knowledge

- Do not turn one face consecutively.
- Do not turn opposite faces consecutively.



Rubik's Cube

■ Experimental results

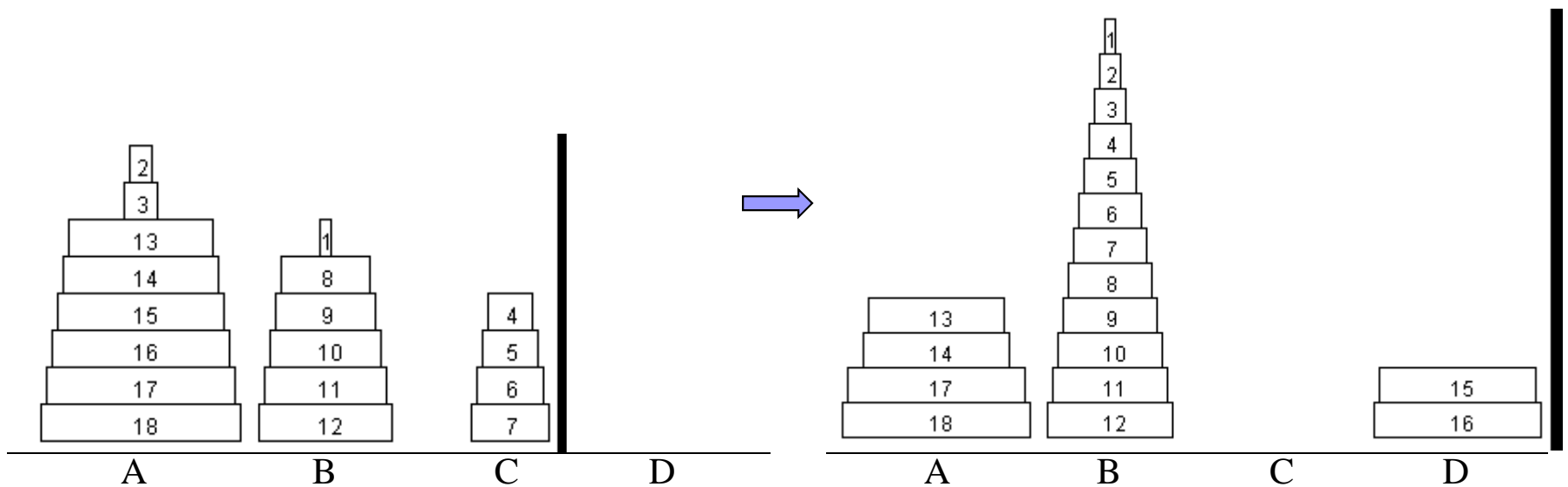
□ $2 \times 2 \times 2$

- Out-of-memory for table area if no goal region is used.
- When the goal is expanded backward by 5 steps, Picat solves most instances in seconds.

□ $3 \times 3 \times 3$

- Picat can solve only easy instances that require up to 14 steps.
- Hard instances normally require 18 steps (in theory, no more than 20 steps).
- Korf's pattern database is too big to store in the table area.

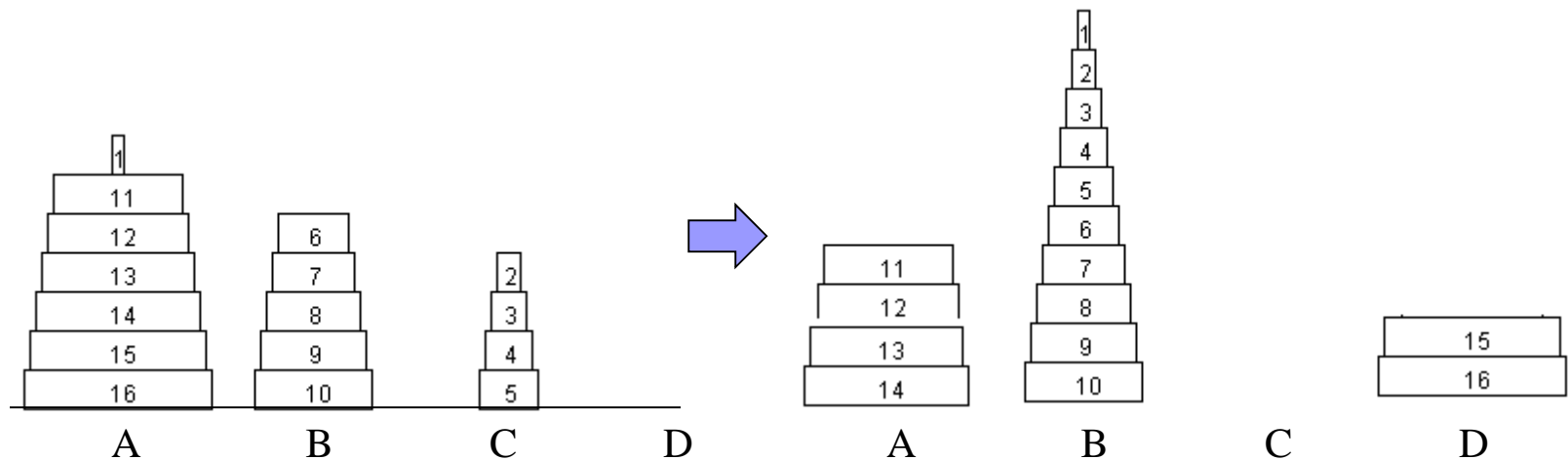
Hanoi Tower (4 Pegs)



Two snapshots from the sequence of the *Frame-Stewart* algorithm

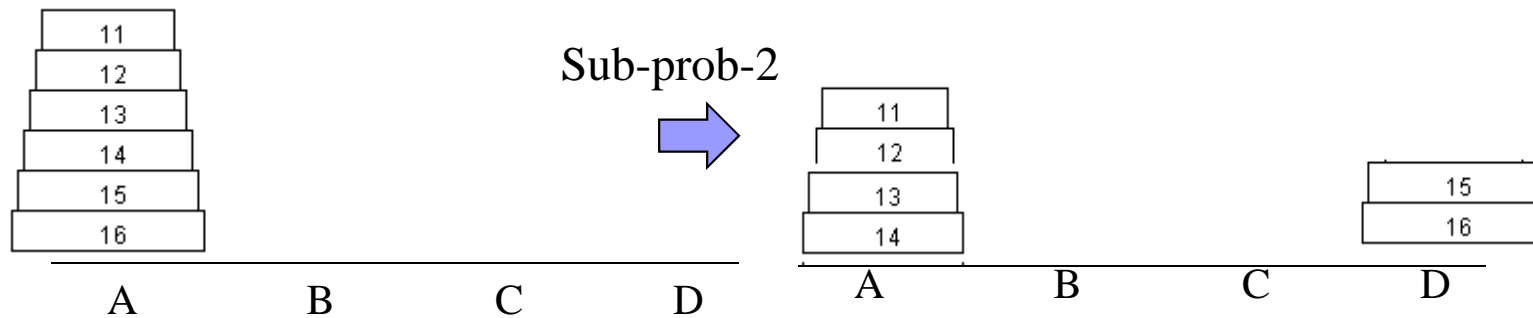
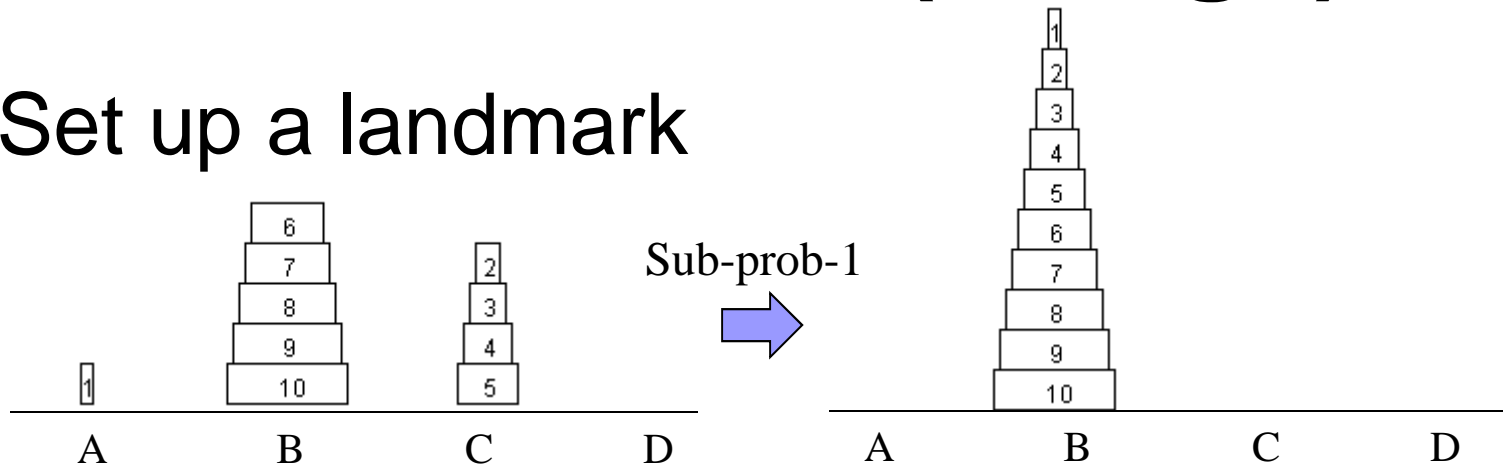
Hanoi Tower (4 Pegs)

- Remove correctly-positioned largest disks



Hanoi Tower (4 Pegs)

Set up a landmark





Hanoi Tower (4 Pegs)

- State representation

```
{N, CurTower, GoalTower}
  CurTower  = [CPeg1, CPeg2, CPeg3, CPeg4]
  GoalTower = [GPeg1, GPeg2, GPeg3, GPeg4]
  Pegi      = [D1, D2, ..., Dk], D1 > D2 > ... > Dk
```



Hanoi Tower (4 Pegs)

```
table (+,-,min)
hanoi4({0,_,_},Plan,Cost) => Plan=[],Cost=0.
% reduce the problem if the largest disk already is on the right peg
hanoi4({N,[[N|CPeg1]|CPegs],[[N|GPeg1]|GPegs]},Plan,Cost) =>
    NewS = {N-1,[CPeg1|CPegs],[GPeg1|GPegs]},
    hanoi4(NewS,Plan,Cost).

...
hanoi4({1,CState,GState},Plan,Cost) =>
    nth(From,CState,[_]),
    nth(To,GState,[_]),
    Plan = [$move(From,To)],
    Cost = 1.
% divide the problem into sub-problems
hanoi4({N,CState,GState},Plan,Cost) =>
    partition_disks(N,CState,GState,ItState,M,Peg),           % set up a landmark
    remove_larger_disks(CState,M) = CState1,
    hanoi4({M,CState1,ItState},Plan1,Cost1),                 % sub-problem1
    remove_smaller_or_equal_disks(CState,M) = CState2,
    remove_smaller_or_equal_disks(GState,M) = GState2,
    N1 is N-M,
    hanoi3({N1,CState2,GState2,Peg},Plan2,Cost2),           % sub-problem2, 3-peg version
    remove_larger_disks(GState,M) = GState1,
    hanoi4({M,ItState,GState1},Plan3,Cost3),                 % sub-problem3
    Plan = Plan1 ++ Plan2 ++ Plan3,
    Cost = Cost1 + Cost2 + Cost3.
```



Hanoi Tower (4 Pegs)

- Experimental results
 - 15 instances from ASP'11 were used
 - Picat solved all
 - In less than 0.1s when no partition heuristic was used.
 - Is even faster if a partition heuristic is used.
 - Clasp also solved all 15 instances
 - On average 20s per instance



Summary

Modeling Techniques

- Use an ordered list to represent positions
 - Rush Hour, Sokoban, Ricochet Robots, and Nomystery.
 - Breaks symmetry and facilitates sharing
- Use heuristics (15-puzzle and Ricochet)
- Identify deterministic actions (Nomystery)
- Goal expansion (Rubik's cube)
- Use landmarks (4-peg Hanoi Tower)



Summary

Picat Vs. PDDL&ASP

■ Picat

- Structures can be used to represent states
- Tabled state-space search (based on IDA* but different)
- Rely on programmers to encode domain knowledge
 - Heuristics, control, determinism, dependency, and symmetry

■ PDDL&ASP

- States are represented as propositional facts
- Various kinds of algorithms for PDDL and SAT for ASP
- Rely on the solver to learn domain knowledge



References

- R. Barták and N.F. Zhou, Using Tabled Logic Programming to Solve the Petrobras Planning Problem, TPLP 2014.
- Håkan Kjellerstrand, Picat: A Logic-based Multi-paradigm Language, ALP Newsletter, 2014.
- N.F. Zhou, Combinatorial Search With Picat, ICLP 2014.
- N.F. Zhou and A. Dovier, A Tabled Prolog Program for Solving Sokoban, Fundamenta Informaticae, 2013.
- N.F. Zhou and J. Fruhman: Toward a Dynamic Programming Solution for the 4-peg Tower of Hanoi Problem with Configurations. CoRR abs/1301.7673 (2013)