Modeling and Solving Planning Problems With Picat

Neng-Fa Zhou

(CUNY Brooklyn College & GC)

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 ($S\times\Sigma \rightarrow \Re$)
  - ☐ 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)
- 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.
- Specify the actions
  - `action`(State,NextState,Action,ActionCost)
    - Encodes the state transition relation
    - States are tabled, and destructive updates of states (using :=) are banned.
- Define a heuristic function if necessary
  - `heuristic(State) = H =>` …
- Call a built-in on an initial state to find a plan

Ex: The Farmer’s Problem

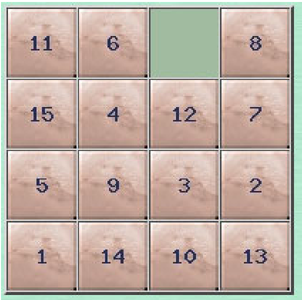
```
import planner.

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

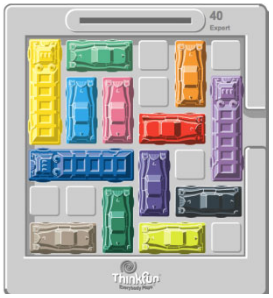
- Find a good representation for states
  - Keep the information minimal.
  - Use good data structures that facilitate
    - sharing
    - computation of heuristics
    - symmetry breaking
- Use heuristics and domain knowledge
  - A state should not be expanded if the travel from it to the final state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

Modeling Examples

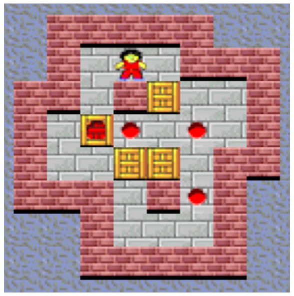
picat-lang.org/projects.html



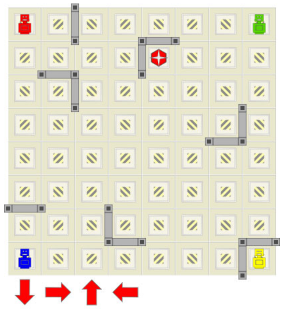
15-puzzle



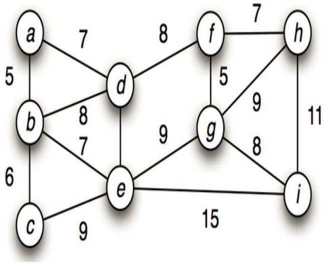
RushHour



Sokoban



Ricochet Robots

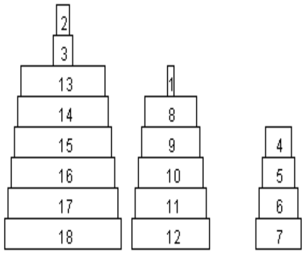


Logistics

⟨5♠⟩, ⟨3♡⟩, ⟨Q♣⟩, ⟨8♢⟩,
⟨K♠⟩, ⟨2♡⟩, ⟨7♣⟩, ⟨4♢⟩,
⟨8♠⟩, ⟨J♡⟩, ⟨9♣⟩, ⟨A♢⟩

Gilbreath's card trick

Rubik's Cube



Tower-of-Hanoi

15-Puzzle

4		3	6
12	1	11	7
9	5	10	15
13	8	14	2

Initial state

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Goal state

- State representation

```
main =>
    Init = [(1,2),(2,2),(4,4),(1,3),(1,1),(3,2),(1,4),(2,4),
            (4,2),(3,1),(3,3),(2,3),(2,1),(4,1),(4,3),(3,4)],
    best_plan(Init,Plan).

final(S) => S = [(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),
                 (3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,3),(4,4)].
```

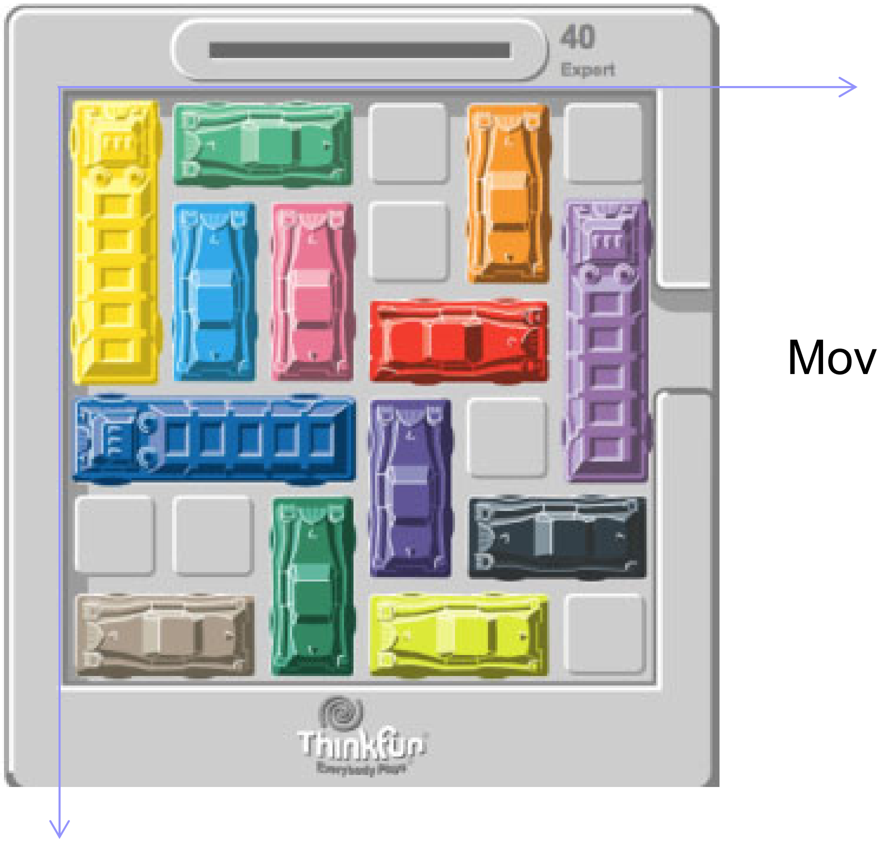
15-Puzzle: Actions

```
action([P0@(R0,C0)|Tiles],NextS,Action,Cost) =>
    Cost = 1,
    (R1 = R0-1, R1 >= 1, C1 = C0, Action = up;
     R1 = R0+1, R1 =< 4, C1 = C0, Action = down;
     R1 = R0, C1 = C0-1, C1 >= 1, Action = left;
     R1 = R0, C1 = C0+1, C1 =< 4, Action = right),
    P1 = (R1,C1),
    slide(P0,P1,Tiles,NTiles),
    NextS = [P1|NTiles].

% slide the tile at P1 to the empty square at P0
slide(P0,P1,[P1|Tiles],NTiles) =>
    NTiles = [P0|Tiles].
slide(P0,P1,[Tile|Tiles],NTiles) =>
    NTiles=[Tile|NTilesR],
    slide(P0,P1,Tiles,NTilesR).
```

15-Puzzle: Heuristics and Performance

```
heuristic([_|Tiles]) = Dist =>
    final([_|FTiles]),
    Dist = sum([abs(R-FR)+abs(C-FC) :
                {(R,C),(FR,FC)} in zip(Tiles,FTiles)]).
```

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×h` cars.
  - Symmetries are removed.

- Goal states

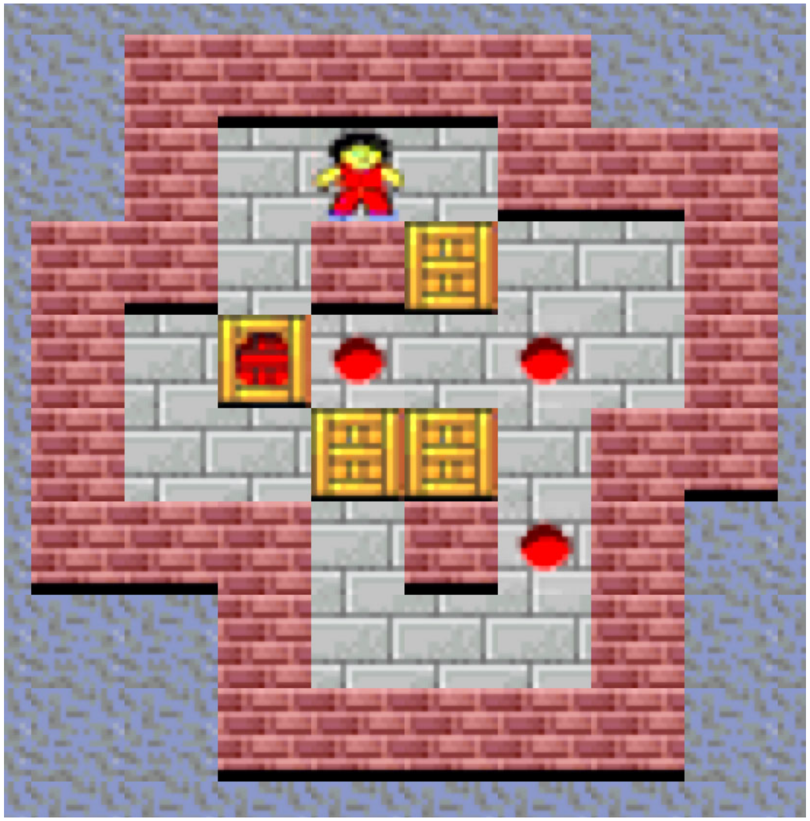
  ```
  final({[4|2],_,_,_,_,_}) => true.
  ```

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}.
…
```

Sokoban



source: takaken

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

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

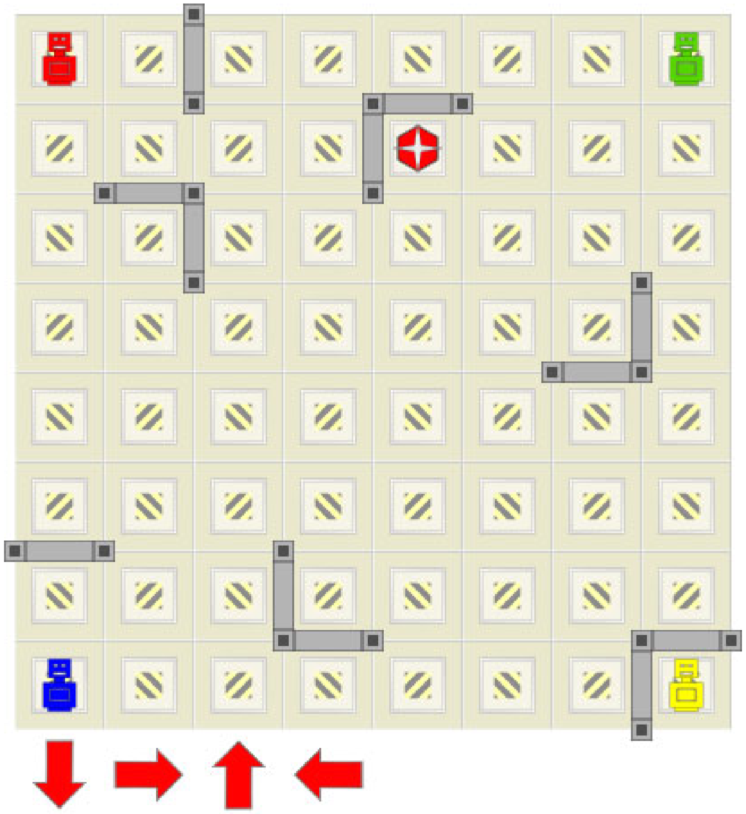
- Actions

```
% 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

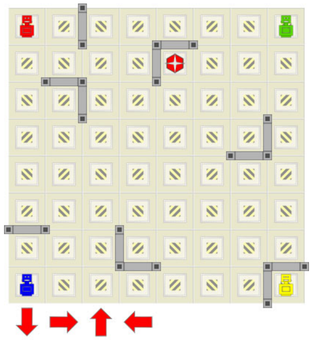


source:Martin Gebser et al.

Ricochet Robots

- State representation

`{[CurLoc|TargetLoc],ORobotLocs}`



`{[(1,1)|(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

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

Ricochet Robots

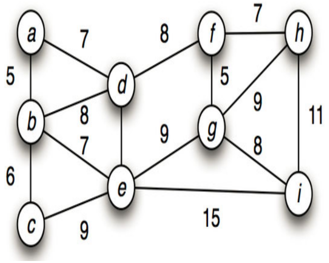
- Actions

```
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).
```

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

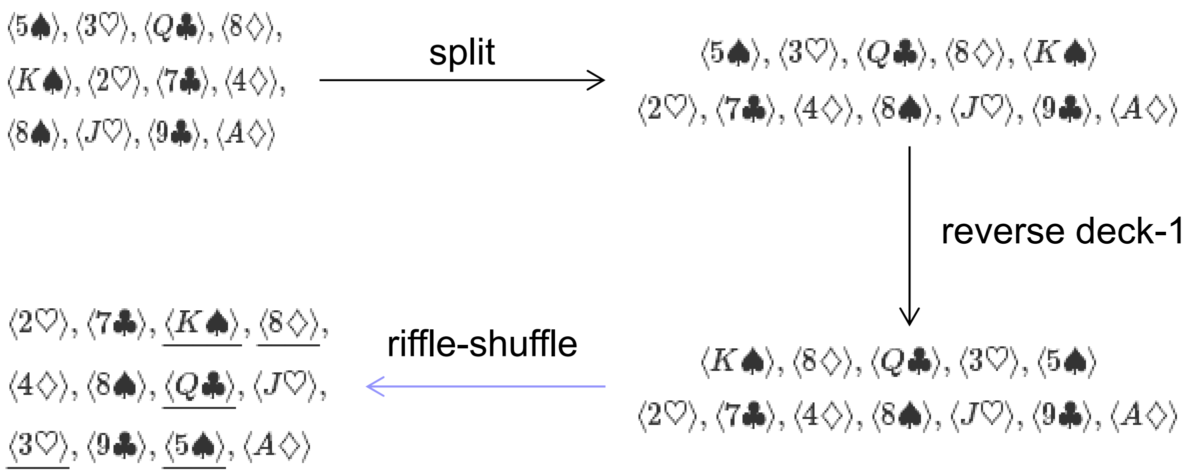
```
action({Loc,LCGs,WCGs},NextState,Action,Cost),
    select(Loc,LCGs,LCGs1)
=>
    Action = $unload(Loc),
    Cost = 0,
    NextState= {Loc,LCGs1,WCGs}.
action({Loc,LCGs,WCGs},NextState,Action,Cost),
    select([Loc|CargoDest],WCGs,WCGs1)
=>
    Action = $load(Loc,CargoDest),
    Cost = 0,
    NextState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



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

split

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

reverse deck-1

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

riffle-shuffle

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

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

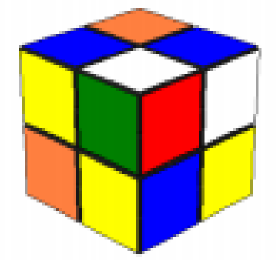
```
final(shuffled(Cards)) =>
    test_quartet(Cards,[c,d,h,s]).

test_quartet([C1,C2,C3,C4|_Cards],Suits),
    sort([C1,C2,C3,C4]) !== Suits
=> true.
test_quartet([_,_,_,_|Cards],Suits) =>
    test_quartet(Cards,Suits).
```

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).
```

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

From Richard E. Korf’97

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

Rubik’s Cube

- Actions

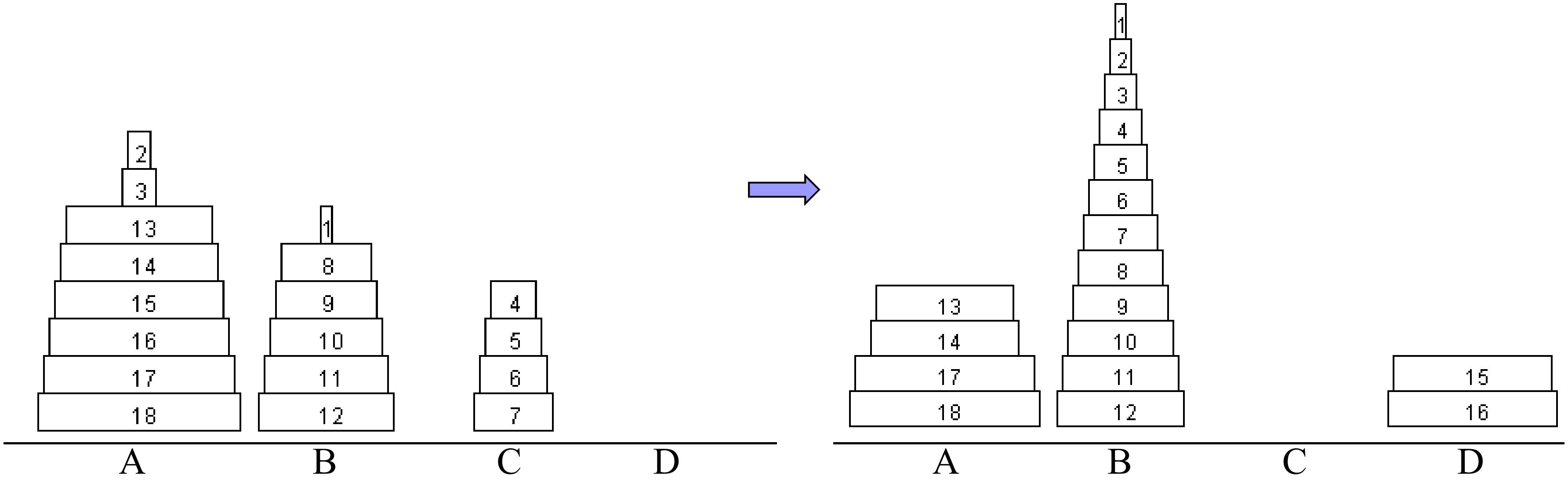
```
action(S,NewS,Action,Cost) =>
    current_resource_plan_cost(Limit,CurPlan,_CurPlanLen),
    actions(Actions),
    Cost = 1,
    member(Action,Actions),
    not nogood_action(CurPlan,Action),
    transform(Action,S,NewS).
```

- Some domain knowledge
  - ☐ Do not turn one face consecutively.
  - ☐ Do not turn opposite faces consecutively.

Rubik's Cube

- Experimental results
  - 2×2×2
    - 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.
  - 3×3×3
    - 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.

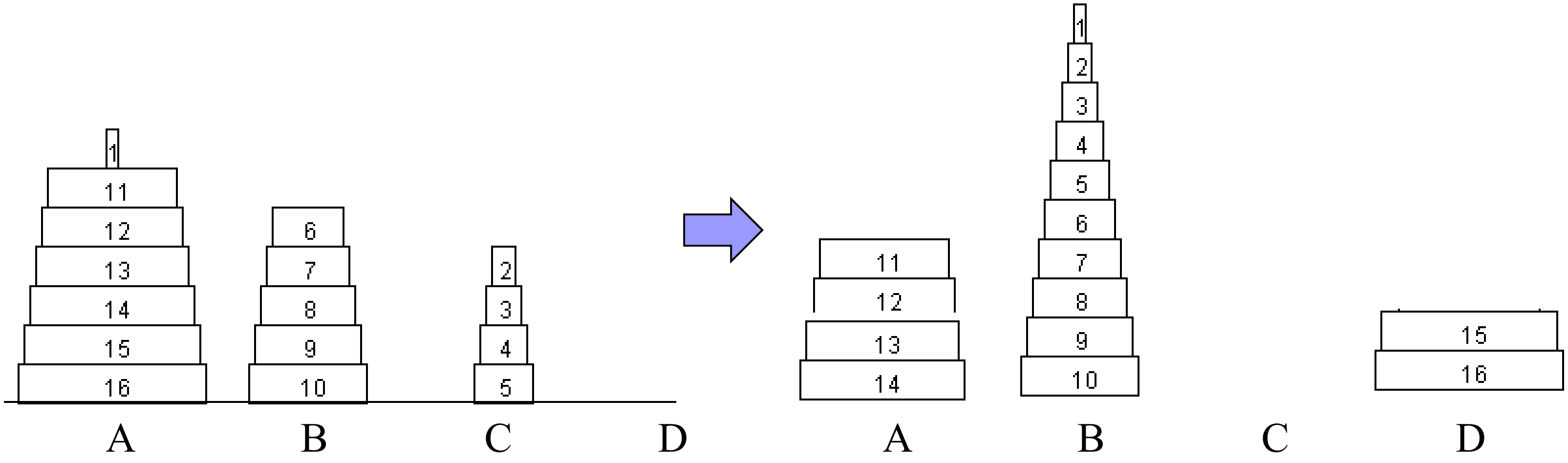
Hanoi Tower (4 Pegs)



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

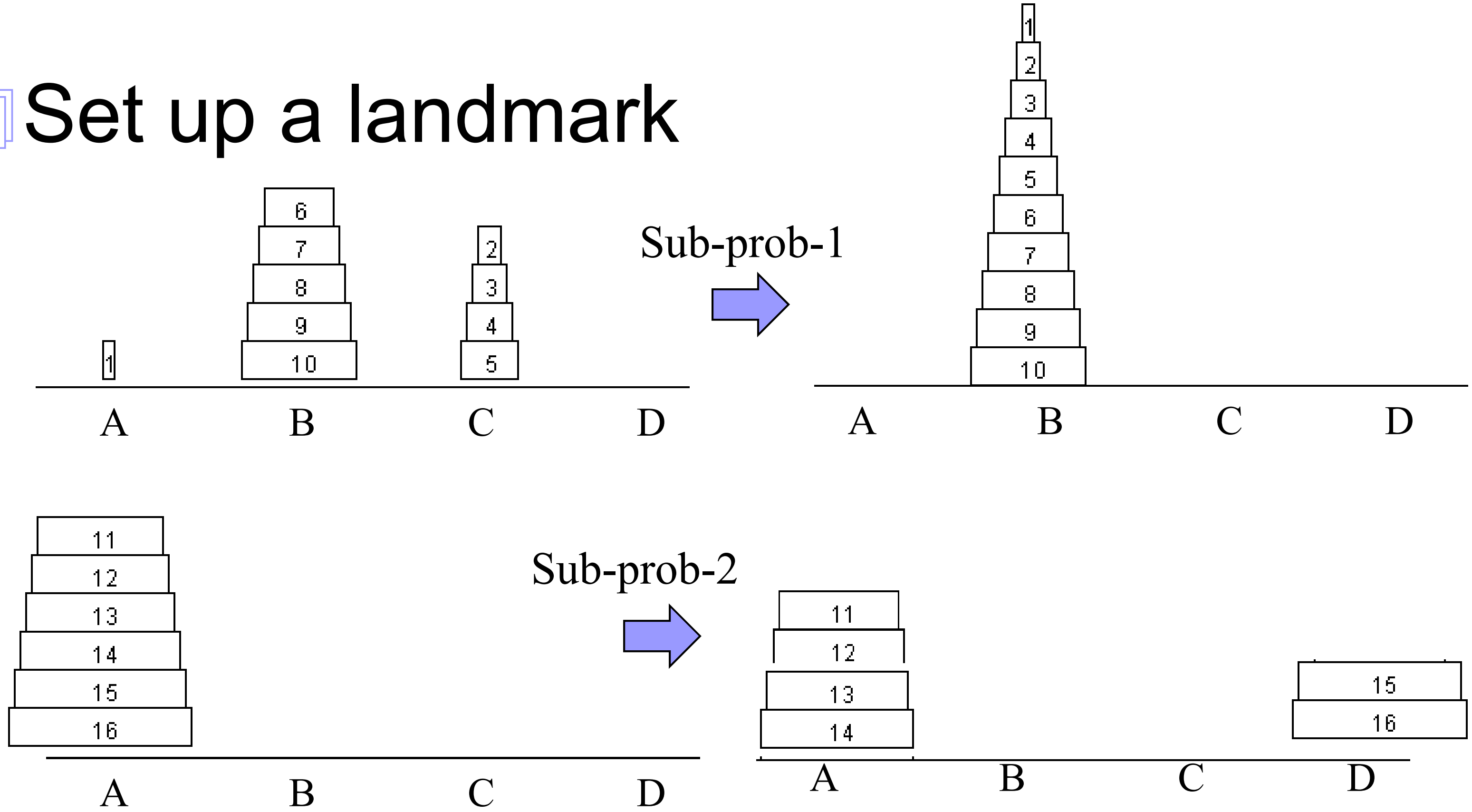
Hanoi Tower (4 Pegs)

- 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,CT,GT},Plan,Cost) =>
    nth(From,CT,[_]),
    nth(To,GT,[_]),
    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 was 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)