NOPT042 Constraint programming: Tutorial 12 - Planning

In [1]:
```
%load_ext ipicat
```

```
Picat version 3.9
```

The planning problem

Abstractly, planning refers to a class of problems where we are given:

- an initial state,
- final states (a set of them, or perhaps a property that makes a state final),
- a set of possible actions (how they transforming a state to another state),

and our task is to find a sequence of actions transforming the inital state to the final state, possibly optimizing some objective, and satisfying some resource limits. This problem appears in many practical applicatios (e.g. logistics, robotics), as well as in logical puzzles (e.g. the 15 puzzle) or games (e.g. Sokoban).

On an abstract level, we are trying to find a path (from the initial state to some final state) in the state graph (aka `state transition diagram`). The state graphis not explicitly constructed, as it is typically huge.

Several (all?) of the exercises from the last tutorial (on Tabling) can be seen as planning problems, generally following this pseudocode (the book):

```
table (+,-,min)

path(S,Plan,Cost),final(S) =>
    Plan=[],Cost=0.

path(S,Plan,Cost) =>
    action(S,NextS,Action,ACost),
    path(NextS,Plan1,Cost1),
    Plan = [Action|Plan1],
    Cost = ACost+Cost1.
```

The `planner` module

For planning problems, Picat provides the module `planner` which implements essentially the above pseudocode. It is enough to define the predicates

- `final(S)`,
- `action(S, NextS, Action, ACost)`,

and provide an initial state `SInit`. The `ACost` must be nonnegative. For example, if we only care about the number of steps, we set it to `1`. The actions are tried in the order in which they are defined (as rules defining the predicate `action`). Remember that if there are multiple actions, all but the last rule must be backtrackable (`?=>`).

The above pseudocode implements *depth-unbounded search*. The module `planner` also implements *depth-bounded search* (e.g. *iterative deepening* and *branch and bound*) and heuristic search.

See the guide, Chapter 8 for more details. (The module is not big, only ~300 LOC.)

In [2]:
```
#!wget http://picat-lang.org/download/planner.pi
!cat planner.pi
```

Depth-unbounded search

The module `planner` implements the following two predicates:

- `plan_unbounded(S, Limit, Plan, PlanCost)` : find any plan where `PlanCost <= Limit`
- `best_plan_unbounded(S, Limit, Plan, PlanCost)` : find the best plan

The arguments `PlanCost` and `Limit` can be omitted.

Resource-bounded search

The module `planner` also implements the following predicates:

- `plan(S, Limit, Plan, PlanCost)` : perform *resource-bounded search* (i.e., keep a resource amount, do not explore a state if the resource amount is negative or if the state has previously failed with the same or more resource), if the resource is plan length (number of actions), this is *depth-bounded search*.
- `best_plan(S, Limit, Plan, PlanCost)` : finds the lowest-cost plan, using *iterative deepening*; calls `plan/4` setting the initial cost to 0 and then iteratively increasing.
- `best_plan_bb(S, Limit, Plan, PlanCost)` : first find any plan using `plan/4`, then branch and bound lowering the limit.

And we can use the function `current resource() = Limit` which returns the resource of the last call of `plan` ; this can be used to implement a heuristic to prune the search (e.g. in 01-knapsack, if taking all of the remaining items, ignoring weight, won't give us sufficient total value, better than best so far).

Example: 15 puzzle

We will use the `planner` module to solve the 15 puzzle. Before checking the solution, think about what are the states and actions.

See this paper for a solution and more examples.

In [3]:
```
!picat puzzle15/puzzle15.pi
```

```
right
right
down
left
up
left
down
down
left
up
right
down
down
right
right
up
left
down
left
left
up
right
up
right
down
right
up
up
left
down
left
left
up
```

In [4]:
```
!cat puzzle15/puzzle15.pi
```

```
% Adapted from Constraint Solving and Planning with Picat, Springer
% by Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman
import planner.

main =>
    InitS = [(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(InitS,Plan),
    foreach (Action in Plan)
       println(Action)
    end.

final(State) => State=[(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)].

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),
    current_resource() > manhattan_dist(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).

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

Exercise: 01-Knapsack

Implement the 01-knapsack problem using the `planner` module. (Every CSP can be viewed as a planning problem where states are partial assignments and actions represent the choice of value for a variable. We are looking for a path from the root of the search tree to one of the leaves.)

In [5]:
```
!cat knapsack/instance.pi
```

```
instance(ItemNames, Capacity, Values, Weights) =>
    ItemNames = {"tv", "desktop", "laptop", "tablet", "vase", "bottle", "jacket"},
    Capacity = 23,
    Values = {500,350,230,115,180,75,125},
    Weights = {15,11,5,1,7,3,4}.
```

In [6]:
```
!cd knapsack && picat knapsack.pi instance.pi
```

```
(take,tv)
(leave,desktop)
(take,laptop)
(take,tablet)
(leave,vase)
(leave,bottle)
(leave,jacket)
```

In [7]:
```
!cat knapsack/knapsack.pi
```

```
import planner.

main([Filename]) =>
    cl(Filename),
    instance(ItemNames, TotalCapacity, Values, Weights),
    AllItems = [(ItemNames[I], Values[I], Weights[I]) : I in 1..ItemNames.length],

    % state: S@(RemainingItems, RemainingCapacity)
    InitialState = (AllItems, TotalCapacity),

    % PlanCost is the value of items we did not take
    best_plan(InitialState, Plan, PlanCost),
    foreach (Action in Plan)
       println(Action)
    end.

% take the current item
action(CurrentState@(Items, Capacity), NextState, Action, Cost) ?=>
    Items = [Item | RemainingItems],
    Item = (ItemName, ItemValue, ItemWeight),
    Action = (take, ItemName),

    % taking an item costs nothing
    Cost = 0,

    % is this action valid?
    Capacity >= ItemWeight,

    % take the item, lower capacity
    NextState = (RemainingItems, Capacity - ItemWeight).


% leave the current item
action(CurrentState@(Items, Capacity), NextState, Action, Cost) =>
    Items = [Item | RemainingItems],
    Item = (ItemName, ItemValue, ItemWeight),
    Action = (leave, ItemName),

    % leaving an item costs its value
    Cost = ItemValue,

    % leave the item, capacity does not change
    NextState = (RemainingItems, Capacity).


% finish if no remaining items
final(S@(Items, Capacity)) => Items = [].
```

Exercise: Jugs

Solve the Three Jugs Problem (exercise 3.12/8 in the book):

> There are 3 water jugs. The first jug can hold 3 liters of water, the second jug can hold 5 liters, and the third jug is an 8-liter container that is full of water. At the start, the first and second jugs are empty. The goal is to get exactly 4 liters of water in one of the containers. (We are not allowed to spill water).

Generalize to any number of jugs with arbitrary maximum and intial volumes, and any target volume, e.g.:

```
picat jugs.pi "[3,5,8]" "[0,0,8]" 4
```