NOPT042 Constraint programming: Tutorial 12 - Planning

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

```
Picat version 3.5#5
```

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 = Cost1+ACost.
```

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.

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
```

```
/*********************************************************
  15_puzzle.pi
  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)]).
```

See this paper for a solution and more examples.

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
```

Homework: Farmer's problem

The Farmer's problem, also known as the Wolf, goat, and cabbage problem, is the following problem (description from Wikipedia:

> A farmer went to a market and purchased a wolf, a goat, and a cabbage. On his way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only himself and a single one of his purchases: the wolf, the goat, or the cabbage. If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage. The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact.

Solve the problem (find an optimal plan) using the `planner` module. Output the number boat rides needed, and some reasonable description of the direction and boat occupants for each ride. (Only the farmer can row the boat.) Running

```
picat farmer.pi
```

should output the value `7` and some description of the seven boat rides required. Optionally, you can create a constraint model and use the cp solver.

Additional homework assignments

If you need extra points, you can solve the following additional assignments. Each is worth one point. The deadline is end of March; if you want me to grade your solution earlier, email me. There is only one GitHub Classroom repository for all of the assignments.

There are no tests for the autograder. It is up to you to choose a representation of the instance. Prepare at least one simple satisfiable instance, one unsatisfiable instance, and one reasonably large instance (so that your model runs for at least a few seconds). Describe briefly how to run your model.

If you have any questions, or need help, let me know!

1. 3D packing

We have several small boxes (cuboids, i.e. 3D rectangles) given as a list of triples of positive integers (length, width, height). We want to pack them all in a box of the shape of a cube. What is the smallest possible length of the side of such cube?

Each of the small boxes can be placed on any of their sides. Boxes can be placed on top of each other, but:

- the base of the top box must lie completely on another box,
- a box can only be placed on top of a box that is at least as heavy, assuming uniform density.

2. Minimum common substring partition

We are given two strings. We want to partition the strings into substrings so that each string can be obtained from the other by rearranging its parts (if this is possible). The goal is to find a partition into the smallest number of parts.

For example, if `s1 = GAGACTA` and `s2 = AACTGAG`, then the optimal partition has size 3: `s1 = GAG|ACT|A`, `s2 = A|ACT|GAG` (the instance is taken from this paper).

3. DNA double digest

Using enzyme A, a DNA sequence is cut at certain points into a number of fragments. Using enzyme B, it is cut at possibly (but not neccessarily!) different points and into a possibly different number of pieces. We know the lengths of the fragments when using enzyme A, enzyme B, and both the enzymes at once. Find a permutation of the fragments using enzyme A, and a permutation of the fragments using enzyme B, that fits with the fragments using both enzymes.

See this presentation for more details and the following sample instance:

```
A = {375, 282, 2205, 746, 2352, 9040}
B = {3518, 1887, 389, 5916, 2017, 1273}
AB = {375, 282, 2205, 656, 90, 1797, 389, 166, 5750, 2017, 1273}
```

for which a possible solution is `([3, 0, 1, 5, 2, 4], [1, 4, 3, 0, 2, 5])`.

4. Tower of Hanoi

Solve the Tower of Hanoi puzzle for an arbitrary number of pegs and discs. Use the `planner` module. This is Exercise 6.9/7 in the book.