Modeling and Solving AI Problems in Picat

Roman Barták, Neng-Fa Zhou
Pair up all the matching numbers on the grid with single continuous lines (or paths).

- The lines cannot branch off or cross over each other, and
- the numbers have to fall at the end of each line (i.e., not in the middle).

It is considered all the cells in the grid are filled.
Numberlink: a hard instance

Solved with the sat module of Picat and the Lingeling solver in 40s.

Picat, PRICAI'18
import sat.

numberlink(NP,NR,NC,InputM) =>
  M = new_array(NP,NR,NC),
  M :: 0..1,
  % no two numbers occupy the same square
  foreach(J in 1..NR, K in 1..NC)
    sum([M[I,J,K] : I in 1..NP]) #=1
  end,
  % connectivity constraints
  foreach(I in 1..NP, J in 1..NR, K in 1..NC)
    Neibs = [M[I,J1,K1] : (J1,K1) in [(J-1,K),(J+1,K),(J,K-1),(J,K+1)],
              J1>=1, K1>=1, J1=<NR, K1=<NC],
    (InputM[J,K]==I ->
      M[I,J,K] #=1, sum(Neibs) #= 1
    ;
      M[I,J,K] #=> sum(Neibs) #= 2
    )
  end,
  solve(M).
Introduction to Picat

- Picat’s programming constructs
- Logic programming
- Functional programming
- Dynamic programming

Combinatorial (optimization) problems in Picat

- A very short introduction to SAT, CP, MIP modules
- Sudoku
- Golomb ruler
- Multi-agent path finding

Wrap up
Part I:

INTRODUCTION TO PICAT
What is Picat?

Why the name “PICAT”?
  – P(attern-matching), I(nuitive), C(ontexts), A(c tors), T(abling)

Core logic programming concepts:
  – logic variables (arrays and maps are terms)
  – implicit pattern-matching and explicit unification
  – explicit non-determinism

Language constructs for scripting and modeling:
  – functions, loops, list and array comprehensions, and assignments

Facilities for combinatorial search:
  – tabling for dynamic programming
  – the cp, sat, and mip modules for CSPs
  – the planner module for planning
A variable name begins with a capital letter or the underscore.

Picat> var(X)
yes

Picat> X = a, var(X)
no

Picat> X.put_attr(a,1), attr_var(X)
yes

Picat> X.put_attr(a,1), Val = X.get_attr(a)
Val = 1
yes

Picat> import cp
Picat> X :: 1..10, dvar(X)
X = DV_010b48_1..10
yes
An unquoted atom name begins with a lower-case letter. A character is a single-letter atom.

```
Picat> atom(abc)
yes

Picat> atom('_abc')
yes

Picat> char(a)
yes

Picat> Code = ord(a)
Code = 97

Picat> A = chr(97)
A = a
```
Numbers

Picat> int(123)
yes

Picat> Big = 99999999999999999999999
Big = 99999999999999999999999

Picat> X = 0b111101
X = 61

Picat> X = 0xff0
X = 4080

Picat> real(1.23)
yes

Picat> X = 1.23e10
X = 12300000000.0
Lists are singly-linked lists.

```
Picat> L = [a,b,c], list(L)
L = [a,b,c]
yes

Picat> L = new_list(3)
L = [_101c8,_101d8,_101e8]

Picat> L = 1..2..10
L = [1,3,5,7,9]

Picat> L = [X : X in 1..10, even(X)]
L = [2,4,6,8,10]

Picat> L = [a,b,c], Len = len(L)
L = [a,b,c]
Len = 3

Picat> L = [a,b] ++ [c,d]
L = [a,b,c,d]
```
Strings are lists of characters.

Picat> S = "hello"
S = [h,e,l,l,o]

Picat> S = "hello" ++ "Picat"
S = [h,e,l,l,o,'P',i,c,a,t]

Picat> S = to_string(abc)
S = [a,b,c]

Picat> S = to_radix_string(123,16)
S = ['7','B']

Picat> X = to_int("123")
X = 123

Picat> X = parse_term("[1,2,3]")
X = [1,2,3]
Structures

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

Picat> S = new_struct(mary,3)
S = mary(_12ad0,_12ad8,_12ae0)

Picat> S = $f(a), A = arity(S), N = name(S)
A = 1
N = f

Picat> And = (a,b)
And = (a,b)

Picat> Or = (a;b)
Or = (a;b)

Picat> Constr = (X #= Y)
Constr = (_10f18 #= _10f20)
Picat> A = {a,b,c}, array(A)
A = {a,b,c}
yes

Picat> A = new_array(3)
A = {_10528,_10530,_10538}

Picat> A = new_array(3,3)
A = {{_fdb0,_fdb8,_fdc0},...}

Picat> A = {X : X in 1..10, even(X)}
A = {2,4,6,8,10}

Picat> L = [a,b,c], A = to_array(L)
L = [a,b,c]
A = {a,b,c}

Picat> A = {a,b} ++ {c,d}
A = {a,b,c,d}
Maps and sets are hash tables.

Picat> M = new_map([ichi=1, ni=2]), map(M)
M = (map)[ni = 2,ichi = 1]
yes

Picat> M = new_map([ni=2]), Ni = M.get(ni)
Ni = 2

Picat> M = new_map(), M.put(ni,2)
M = (map)[ni = 2]

Picat> M = new_map(), Ni = M.get(ni,unknown)
M = (map)[]
Ni = unknown

Picat> S = new_set([a,b,c])
S = (map)[c,b,a]

Picat> S = new_set([a,b,c]), S.has_key(b)
yes
Index Notation

\[ X[I_1,\ldots,I_n] \] : \( X \) references a compound value

**Linear-time** access of **list** elements.

\begin{verbatim}
Picat> L = [a,b,c,d], X = L[4]
X = d
\end{verbatim}

**Constant-time** access of **structure** and **array** elements.

\begin{verbatim}
Picat> S = $student(mary,cs,3.8), GPA = S[3]
GPA = 3.8
\end{verbatim}

\begin{verbatim}
Picat> A = {{1, 2, 3}, {4, 5, 6}}, B = A[2, 3]
B = 6
\end{verbatim}
List Comprehension

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

Picat> \( L = [X : X \text{ in } 1..10, \text{even}(X)] \)
\( L = [2, 4, 6, 8, 10] \)

Picat> \( L = [(A,I) : A \text{ in } [a,b], I \text{ in } 1..2] \).
\( L = [(a,1), (a,2), (b,1), (b,2)] \)

Picat> \( L = [(A,I) : \{A,I\} \text{ in } \text{zip}([a,b], 1..2)] \)
\( L = [(a,1), (b,2)] \)

Picat> \( L = [X : I \text{ in } 1..5] \quad \% X \text{ is local} \)
\( L = [_\text{bee}8, _\text{bef}0, _\text{bef}8, _\text{bf}00, _\text{bf}08] \)

Picat> \( X = _, L = [X : I \text{ in } 1..5] \quad \% X \text{ is non-local} \)
\( L = [X, X, X, X, X] \)
Picat> Y = 13.to_binary_string()
Y = ['1', '1', '0', '1']

Picat> Y = 13.to_binary_string().reverse()
Y = ['1', '0', '1', '1']

% X becomes an attributed variable
Picat> X.put_attr(age, 35), X.put_attr(weight, 205), A = X.get_attr(age)
A = 35

% X is a map
Picat> X = new_map([age=35, weight=205]), X.put(gender, male)
X = (map)([age=35, weight=205, gender=male])

Picat> S = $point(1.0, 2.0), Name = S.name, Arity = S.len
Name = point
Arity = 2

Picat> Pi = math.pi % module qualifier
Pi = 3.14159
**The foreach Loop**

```
foreach(E₁ in D₁, Cond₁,..., Eₙ in Dₙ, Condₙ)
  Goal
end
```

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

```picat
Picat> A = new_array(5), foreach(I in 1..5) A[I] = X end
A = {_15bd0,_15bd8,_15be0,_15be8,_15bf0}

Picat> X = _, A = new_array(5), foreach(I in 1..5) A[I] = X end
A = {X,X,X,X,X,X}
```
Pattern-matching rules
  - No laziness or freeze
    The call `membchk(X, _)` fails
  - Facilitates indexing

Explicit unification
Explicit non-determinism
Functional Programming in Picat

Head = Exp, Cond => Body.

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

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

qsort([]) = [].
qsort([H|T]) = qsort([E : E in T, E=<H])++
    [H]++
    qsort([E : E in T, E>H]).

Dynamically typed
List and array comprehensions
Strict (not lazy)
Higher-order functions
Function calls cannot occur in head patterns. Index notations, ranges, dot notations, and comprehensions cannot occur in head patterns.

**As-patterns:**

\[
\begin{align*}
\text{merge}([], Ys) &= Ys. \\
\text{merge}(Xs, []) &= Xs. \\
\text{merge}([X|Xs], Ys@[Y|_]) &= [X|Zs], X < Y \Rightarrow Zs = \text{merge}(Xs, Ys). \\
\text{merge}(Xs, [Y|Ys]) &= [Y|Zs] \Rightarrow Zs = \text{merge}(Xs, Ys).
\end{align*}
\]
main =>
    print("enter an integer:");
    N = read_int(),
    foreach(I in 0..N)
        Num := 1,
        printf("%*s", N-I, ""), % print N-I spaces
        foreach(K in 0..I)
            printf("%d ", Num),
            Num := Num*(I-K) div (K+1)
        end,
    end,
    nl
end.

$ picat pascal
enter an integer:5
1
  1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

SSA (Static Single Assignment)
Loops
Dynamic Programming in Picat

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

- Linear tabling
- Mode-directed tabling
- Term sharing
\[
\binom{n}{0} = \binom{n}{n} = 1 \quad \text{for all integers } n \geq 0,
\]
\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \leq k \leq n - 1,
\]

**table**

\[
c(_, 0) = 1.
c(N, N) = 1.
c(N, k) = c(N-1, k-1) + c(N-1, k).
\]
• Tabled parser

% E -> E + T | E - T | T

table
ex(Si,So) ?=>
    ex(Si,S1),
    S1 = ['+'|S2],
    term(S2,So).
ex(Si,So) ?=>
    ex(Si,S1),
    S1 = ['-'|S2],
    term(S2,So).
ex(Si,So) =>
    term(Si,So).

• Non-tabled parser

% E -> TE'

ex(Si,So) =>
    term(Si,S1),
    ex_prime(S1,So).

% E' -> + TE' | - TE' | ε

ex_prime(['+'|Si],So) =>
    term(Si,S1),
    ex_prime(S1,So).
ex_prime(['-'|Si],So) =>
    term(Si,S1),
    ex_prime(S1,So).
ex_prime(Si,So) => So = Si.

Dynamic Programming: Path-finding

```
table (+, -, min)
path(S, Path, Cost), final(S) =>
    Path = [], Cost = 0.
path(S, Path, Cost) =>
    action(S, S1, Action, ActionCost),
    path(S1, Path1, Cost1),
    Path = [Action | Path1],
    Cost = Cost1 + ActionCost.
```
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,Cost) ?=>
    Action=farmer_wolf,
    Cost = 1,
    opposite(F,F1),
    S1=[F1,F1,G,C],
    not unsafe(S1).

• Based on tabling
• Allows use of structures to represent states
• Supports domain knowledge and heuristics
• Provides search predicates
  – Depth-unbounded & depth-bounded
  – IDA & branch-and-bound
15-Puzzle

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),
    println(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)].
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).

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

COMBINATORIAL (OPTIMIZATION) PROBLEMS IN PICAT
import cp.          import sat.          import mip.

Constraints:

Domain
X :: Domain, X notin Domain

Arithmetic
(X #= Y), (X #!= Y), (X #> Y), (X #>= Y), ...

Boolean
(X #\ Y), (X #\ Y), (X #<=> Y), (X #=> Y), (X #^ Y), (#~ X)

Table
table_in(VarTuple,Tuples),  table_notin(VarTuple,Tuples)

Global
all_different(L), element(I,L,V), circuit(L), cumulative(...), ...

Solver invocation:
solve(Options,Vars)
Example: Send More Money

\[
\begin{array}{c}
S E N D \\
\downarrow \\
+ M O R E \\
\downarrow \\
M O N E Y \\
\end{array}
\quad
\begin{array}{c}
9 5 6 7 \\
\downarrow \\
+ 1 0 8 5 \\
\downarrow \\
1 0 6 5 2 \\
\end{array}
\]

import cp.

\[
\text{send\_more\_money} \Rightarrow \\
\begin{align*}
\text{Vars} &= [S,E,N,D,M,O,R,Y], \\
\text{Vars} &: : 0..9, \\
\text{all\_different}(\text{Vars}), \\
S &\neq 0, \\
M &\neq 0, \\
1000*S+100*E+10*N+D \\
+1000*M+100*O+10*R+E \\
&\neq 10000*M+1000*O+100*N+10*E+Y,
\end{align*}
\]
solve(\text{Vars}), \quad \% \text{label variables}
\text{writeln}(\text{Vars}).

import sat.

Common interface to CP, SAT, and MIP
import sat.

crossword(Vars) =>
  Vars = [X1, X2, X3, X4, X5, X6, X7],
  Words2 = [{ord('A'), ord('S')},
             {ord('G'), ord('O')},
             {ord('I'), ord('F')},
             {ord('I'), ord('N')},
             {ord('T'), ord('O')}],
  Words3 = [{ord('F'), ord('U'), ord('N')},
             {ord('N'), ord('A'), ord('G')},
             {ord('S'), ord('A'), ord('G')},
             {ord('T'), ord('A'), ord('D')}],
  table_in([{X1, X2}, {X1, X3}, {X5, X7}, {X6, X7}], Words2),
  table_in([{X3, X4, X5}, {X2, X4, X6}], Words3),
  solve(Vars),
  writeln([chr(Code) : Code in Vars]).
Combinatorial puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

Solving Sudoku

Use information that each digit appears exactly once in each row, column and sub-grid.
We can see every cell as a variable with possible values from domain \{1,\ldots,9\}.

There is a binary inequality constraint between all pairs of variables in every row, column, and sub-grid.

Such formulation of the problem is called a constraint satisfaction problem.
import cp.

sudoku(Board) =>
    N = Board.length,
    N1 = ceiling(sqrt(N)),
    Board :: 1..N,
    foreach(R in 1..N)
        all_different([Board[R,C] : C in 1..N])
    end,
    foreach(C in 1..N)
        all_different([Board[R,C] : R in 1..N])
    end,
    foreach(R in 1..N1..N, C in 1..N1..N)
        all_different([Board[R+I,C+J] : I in 0..N1-1, J in 0..N1-1])
    end,
    solve(Board).

Sudoku in Picat

{{_, 6, _, 1, _, 4, _, 5, _},
 {_, _, 8, 3, _, 5, 6, _, _},
 {2, _, 7, 2, _, 1, _, 4, _},
 {_, _, 4, 7, _, 3, _, _, _},
 {8, _, 6, _, 3, 1, _, _, _},
 {_, _, 6, 1, _, _, 3, _, _},
 {7, _, 9, 1, _, _, _, 2, _},
 {_, _, 7, 2, _, 6, 9, _, _},
 {_, 4, _, 5, _, 8, _, 7, _}}
A **ruler with M marks** such that distances between any two marks are **different**.

The **shortest ruler** is the optimal ruler.

![Golomb ruler table](image)

Hard for $M \geq 16$, no exact algorithm for $M \geq 24$!

Applied in **radioastronomy**.

---

**Solomon W. Golomb**  
**Professor**  
**University of Southern California**  
http://csi.usc.edu/faculty/golomb.html
distributed.net vyřešili optimálně problémy 24-27 pomocí masivního paralelismu

Roman Barták, 10/15/2016
A base model:

Variables $X_1, \ldots, X_M$ with the domain $0..M*M$

$X_1 = 0$  

$ruler$ $start$

$X_1 < X_2 < \ldots < X_M$  

$no$ $permutations$ $of$ $variables$

$\forall i < j \ D_{i,j} = X_j - X_i$  

$difference$ $variables$

$all$ $different(\{ D_{1,2}, D_{1,3}, \ldots, D_{1,M}, D_{2,3}, \ldots, D_{M-1,M} \})$

Model extensions:

$D_{1,2} < D_{M-1,M}$  

$symmetry$ $breaking$

better bounds (implied constraints) for $D_{i,j}$

$D_{i,j} = D_{i,i+1} + D_{i+1,i+2} + \ldots + D_{j-1,j}$

so $D_{i,j} \geq \Sigma_{j-i} = (j-i)*(j-i+1)/2$  

$lower$ $bound$

$X_M = X_M - X_1 = D_{1,M} = D_{1,2} + D_{2,3} + \ldots + D_{i-1,i} + D_{i,j} + D_{j,j+1} + \ldots + D_{M-1,M}$

$D_{i,j} = X_M - (D_{1,2} + \ldots + D_{i-1,i} + D_{j,j+1} + \ldots + D_{M-1,M})$

so $D_{i,j} \leq X_M - (M-1-j+i)*(M-j+i)/2$  

$upper$ $bound$
import cp.

golomb(M) =>
    X = new_list(M),
    X :: 0..(M*M),               % domains for marks
    X[1] = 0,

    foreach(I in 1..(M-1))
        X[I] #< X[I+1]               % no permutaions
    end,

    D = new_array(M,M),           % distances
    foreach(I in 1..(M-1), J in (I+1)..M)
        D[I,J] #= X[J] - X[I],
        D[I,J] #>= (J-I)*(J-I+1)/2,  % bounds
        D[I,J] #=< X[M] - (M-1-J+I)*(M-J+I)/2
    end,

    D[1,2] #< D[M-1,M],           % symmetry breaking
    all_different([D[I,J] : I in 1..(M-1),
                   J in (I+1)..M]),

    solve(${[min(X[M])]}],X),
    println(X).
**What is the effect of different constraint models?**

<table>
<thead>
<tr>
<th>size</th>
<th>base model</th>
<th>base model + symmetry</th>
<th>base model + symmetry + implied constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>12</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>94</td>
<td>44</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>860</td>
<td>353</td>
<td>143</td>
</tr>
<tr>
<td>10</td>
<td>7 494</td>
<td>3 212</td>
<td>1 091</td>
</tr>
<tr>
<td>11</td>
<td>147 748</td>
<td>57 573</td>
<td>23 851</td>
</tr>
</tbody>
</table>

Time in milliseconds on 1.7 GHz Intel Core i7, Picat 1.9#6

**What is the effect of different search strategies?**

<table>
<thead>
<tr>
<th>size</th>
<th>fail first</th>
<th>leftmost first</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>enum</td>
<td>split</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>67</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>537</td>
<td>537</td>
</tr>
<tr>
<td>10</td>
<td>4 834</td>
<td>4 721</td>
</tr>
<tr>
<td>11</td>
<td>134 071</td>
<td>132 046</td>
</tr>
</tbody>
</table>

Time in milliseconds on 1.7 GHz Intel Core i7, Picat 1.9#6
Setting:

- a **graph** (directed or undirected)
- a set of **agents**, each agent is assigned to two locations (nodes) in the graph (start, destination)

MAPF problem:

Find a **collision-free** plan (path) for each agent.
In MAPF, we do not know the lengths of plans (due to possible re-visits of nodes)!
We can encode plans of a known length using a **layered graph** (temporally extended graph).

Each layer corresponds to one time slice and indicates positions of agents at that time.
Using **layered graph** describing agent positions at each time step

\[ B_{tav} : \text{agent } a \text{ occupies vertex } v \text{ at time } t \]

**Constraints:**

- each agent occupies exactly one vertex at each time.
  \[ \sum_{v=1}^{n} B_{tav} = 1 \text{ for } t = 0, \ldots, m, \text{ and } a = 1, \ldots, k. \]

- no two agents occupy the same vertex at any time.
  \[ \sum_{a=1}^{k} B_{tav} \leq 1 \text{ for } t = 0, \ldots, m, \text{ and } v = 1, \ldots, n. \]

- if agent \( a \) occupies vertex \( v \) at time \( t \), then \( a \) occupies a neighboring vertex or stay at \( v \) at time \( t + 1 \).
  \[ B_{tav} = 1 \Rightarrow \sum_{u \in \text{neibs}(v)} (B_{(t+1)au}) \geq 1 \]

**Preprocessing:**

\[ B_{tav} = 0 \text{ if agent } a \text{ cannot reach vertex } v \text{ at time } t \text{ or } a \text{ cannot reach the destination being at } v \text{ at time } t \]
import sat.

path(N,As) =>
    K = len(As),
    lower_upper_bounds(As, LB, UB),
    between(LB, UB, M),
    B = new_array(M+1, K, N),
    B :: 0..1,

    % Initialize the first and last states
    foreach (A in 1..K)
        (V, FV) = As[A],
        B[1, A, V] = 1,
        B[M+1, A, FV] = 1
    end,

    % Each agent occupies exactly one vertex
    foreach (I in 1..M+1, A in 1..K)
        sum([B[I, A, V] : V in 1..N]) #= 1
    end,

    % No two agents occupy the same vertex
    foreach (I in 1..M+1, V in 1..N)
        sum([B[I, A, V] : A in 1..K]) #=< 1
    end,

    % Every transition is valid
    foreach (I in 1..M, A in 1..K, V in 1..N)
        neibs(V, Neibs),
        B[I, A, V] #->
        sum([B[I+1, A, U] : U in Neibs]) #=> 1
    end,

    solve(B),
    output_plan(B).

Agent moves to a neighboring vertex

Incremental generation of layers

Setting the initial and destination locations

Agent occupies one vertex at any time

No conflict between agents

L-robustness
If any agent is delayed then trains may cause collisions during execution.

To prevent such collisions we may introduce more space between agents.
<table>
<thead>
<tr>
<th>Instance</th>
<th>Makespan</th>
<th>Sum of costs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Picat</td>
<td>MDD</td>
</tr>
<tr>
<td>g16_p10_a05</td>
<td>0.27</td>
<td>0.02</td>
</tr>
<tr>
<td>g16_p10_a10</td>
<td>1.37</td>
<td>0.14</td>
</tr>
<tr>
<td>g16_p10_a20</td>
<td>2.76</td>
<td>0.76</td>
</tr>
<tr>
<td>g16_p10_a30</td>
<td>3.11</td>
<td>0.79</td>
</tr>
<tr>
<td>g16_p10_a40</td>
<td>8.25</td>
<td>4.71</td>
</tr>
<tr>
<td>g16_p20_a05</td>
<td>1.01</td>
<td>0.16</td>
</tr>
<tr>
<td>g16_p20_a10</td>
<td>1.5</td>
<td>0.31</td>
</tr>
<tr>
<td>g16_p20_a20</td>
<td>2.12</td>
<td>0.46</td>
</tr>
<tr>
<td>g16_p20_a30</td>
<td>4.37</td>
<td>1.45</td>
</tr>
<tr>
<td>g16_p20_a40</td>
<td>3.48</td>
<td>1.15</td>
</tr>
<tr>
<td>g32_p10_a05</td>
<td>1.98</td>
<td>0.53</td>
</tr>
<tr>
<td>g32_p10_a10</td>
<td>3.08</td>
<td>1.21</td>
</tr>
<tr>
<td>g32_p10_a20</td>
<td>8.71</td>
<td>6.8</td>
</tr>
<tr>
<td>g32_p10_a30</td>
<td><strong>34.48</strong></td>
<td>40.13</td>
</tr>
<tr>
<td>g32_p10_a40</td>
<td>34.95</td>
<td><strong>24.87</strong></td>
</tr>
<tr>
<td>g32_p20_a05</td>
<td>5.75</td>
<td>2.77</td>
</tr>
<tr>
<td>g32_p20_a10</td>
<td>2.97</td>
<td>1.11</td>
</tr>
<tr>
<td>g32_p20_a20</td>
<td>16.93</td>
<td>13.73</td>
</tr>
<tr>
<td>g32_p20_a30</td>
<td>12.98</td>
<td>4.54</td>
</tr>
<tr>
<td>g32_p20_a40</td>
<td>16.51</td>
<td>8.17</td>
</tr>
<tr>
<td>Total solved</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>
WRAP UP
**Picat** is a logic-based multi-paradigm language that integrates logic programming, functional programming, constraint programming, and scripting.

- logic variables, unification, backtracking, pattern-matching rules, functions, list/array comprehensions, loops, assignments
- tabling for dynamic programming and planning
- **constraint solving** with CP (constraint programming), SAT (satisfiability), and MIP (mixed integer programming).
I enjoy programming in Picat because it suits my mindset very well. -- Hakan Kjellerstrand

Thank you for your beautiful project! Using Picat, I felt "at home" almost right away. -- Stefan Kral

The Picat language is really cool; it's a very usable mix of logic, functional, constraint, and imperative programming. Scripts can be made quite short but also easily readable. And the built-in tabling is really cool for speeding up recursive programs. I think Picat is like a perfect Swiss army knife that you can do anything with. -- Lorenz Schuffmann

In some cases the use of Picat simplifies the implementation compared to conventional imperative programming languages, while in others it allows to directly convert the problem statement into an efficiently solvable declarative problem specification without inventing an imperative algorithm. -- Sergii Punchenko
Constraint Solving and Planning with Picat


Modeling and Solving AI Problems in Picat

Roman Barták, Neng-Fa Zhou