JuMP (part 2): Exercises

Skills targeted:

- Structure, model and solve a concrete optimisation situation.
- Solve an optimisation problem using an algebraic language and a MIP solver.
- Formulate an implicit linear programming model with JuMP.
- Handle vectors and matrices with Julia.
- Present the optimisation results according a specified format.

Activities:

- Write the linear programming model corresponding to a problem.
- Write the obtained model with JuMP.
- Write the all-in-one program which

```
1 brings together the data into an adequate datastructure,
2 states the optimisation model,
3 computes the optimal solution.
```

Situation 1 (🌶️):

Assigning agents to tasks

Situation

The linear assignment problem is a fundamental combinatorial optimization problem. It can be stated as follows:

The problem instance has a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform as many tasks as possible by assigning at most one agent to each task and at most one task to each agent, in such a way that the total cost of the assignment is minimized.

Example

A company has 4 machines available for assignment to 4 tasks. Any machine can be assigned to any task, and each task requires processing by one machine. The time required to set up each machine for the processing of each task is given in the table below.

Machines	Task1	Task2	Task3	Task4
Machine 1	13	4	7	6
Machine 2	1	11	5	4
Machine 3	6	7	2	8
Machine 4	1	3	5	9

In this example, each value represents a time (hours).

Question

Write an implicit model which minimize the total setup time needed for the processing of all four tasks. Find the corresponding minimal value and a corresponding assignement.

Solution

Entrée [ ]:

```

```

Situation 2 (🌶️🌶️):

Guiding perseverance to discover Mars

Situation

While the real Perseverance rover is having fun on Mars, we imagine an alternative version that scouts out an *N*×N* grid of Mars according to the following rules:

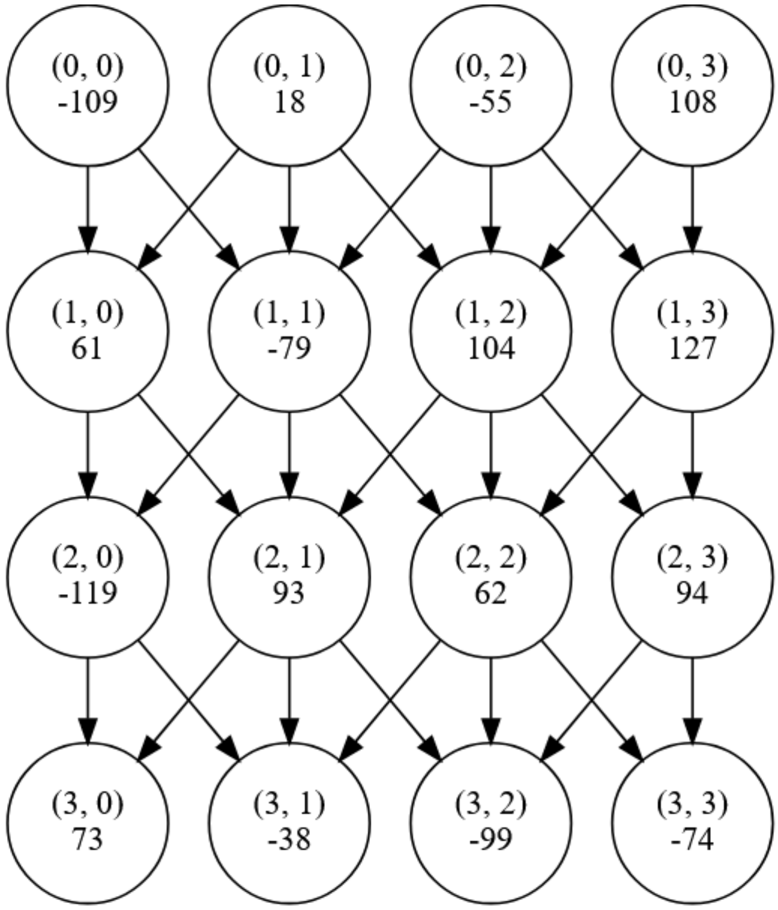
- Surveying a cell is possible only if all its upper neighbors were already explored. The upper neighbors of *(a,b)* are defined as *(a-1,b-1), (a-1,b), (a-1,b+1)*. Cells that are not on the *N*×N* grid do not need to be surveyed first.
- Each cell has a "score" between *0-255* points, indicating how valuable it is to explore it.
- Exploring a cell also requires rover maintenance, equivalent to a "cost" of *128* points.

The goal of the rover is to earn the maximum score possible from the grid. This means choosing which cells to explore that satisfy condition 1, such that the total score gained, considering 2 and 3, is the maximum score possible.

We represent the grid as an *N*×N* array of numbers given in hexadecimal format. As an example, consider the following *4*×4* grid representation:

```
13 92 49 EC
BD 31 E8 FF
09 DD BE DE
C9 5A 1D 36
```

Which represents the following grid (the arrow *A**→B* means "Exploring *A* is a prerequisite to exploring *B*"):



For example, the value *-109* in cell *(0,0)* is obtained by converting *13* in hexadecimal notion to *16+3=19* and subtracting *128*, obtaining *-109*. Similarly, the value *18* in *(0,1)* is obtained by converting *92* to *9*\16+2=146* and subtracting *128*.

For the grid above, the optimal score is 424, and can be achieved via the following set:

$$[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]$$

Question

Find the maximum score and a set of cells achieving it for the following *20*×*20* grid:

```
                  BC E6 56 29 99 95 AE 27 9F 89 88 8F BC B4
2A 71 44 7F AF 96
                  72 57 13 DD 08 44 9E A0 13 09 3F D5 AA 06
5E DB E1 EF 14 0B
                  42 B8 F3 8E 58 F0 FA 7F 7C BD FF AF DB D9
13 3E 5D D4 30 FB
                  60 CA B4 A1 73 E4 31 B5 B3 0C 85 DD 27 42
4F D0 11 09 28 39
                  1B 40 7C B1 01 79 52 53 65 65 BE 0F 4A 43
CD D7 A6 FE 7F 51
                  25 AB CC 20 F9 CC 7F 3B 4F 22 9C 72 F5 FE
F9 BF A5 58 1F C7
                  EA B2 E4 F8 72 7B 80 A2 D7 C1 4F 46 D1 5E
FA AB 12 40 82 7E
                  52 BF 4D 37 C6 5F 3D EF 56 11 D2 69 A4 02
0D 58 11 A7 9E 06
                  F6 B2 60 AF 83 08 4E 11 71 27 60 6F 9E 0A
D3 19 20 F6 A3 40
                  B7 26 1B 3A 18 FE E3 3C FB DA 7E 78 CA 49
F3 FE 14 86 53 E9
                  1A 19 54 BD 1A 55 20 3B 59 42 8C 07 BA C5
27 A6 31 87 2A E2
                  36 82 E0 14 B6 09 C9 F5 57 5B 16 1A FA 1C
8A B2 DB F2 41 52
                  87 AC 9F CC 65 0A 4C 6F 87 FD 30 7D B4 FA
CB 6D 03 64 CD 19
                  DC 22 FB B1 32 98 75 62 EF 1A 14 DC 5E 0A
A2 ED 12 B5 CA C0
                  05 BE F3 1F CB B7 8A 8F 62 BA 11 12 A0 F6
79 FC 4D 97 74 4A
                  3C B9 0A 92 5E 8A DD A6 09 FF 68 82 F2 EE
9F 17 D2 D5 5C 72
```

Solution

Entrée [1]:

```
# The matrix for the full size example:

v=[0xBC 0xE6 0x56 0x29 0x99 0x95 0xAE 0x27 0x9F 0x89 0x88 0x8F 0xBC
   0x72 0x57 0x13 0xDD 0x08 0x44 0x9E 0xA0 0x13 0x09 0x3F 0xD5 0xAA
   0x42 0xB8 0xF3 0x8E 0x58 0xF0 0xFA 0x7F 0x7C 0xBD 0xFF 0xAF 0xDB
   0x60 0xCA 0xB4 0xA1 0x73 0xE4 0x31 0xB5 0xB3 0x0C 0x85 0xDD 0x27
   0x1B 0x40 0x7C 0xB1 0x01 0x79 0x52 0x53 0x65 0x65 0xBE 0x0F 0x4A
   0x25 0xAB 0xCC 0x20 0xF9 0xCC 0x7F 0x3B 0x4F 0x22 0x9C 0x72 0xF5
   0xEA 0xB2 0xE4 0xF8 0x72 0x7B 0x80 0xA2 0xD7 0xC1 0x4F 0x46 0xD1
   0x52 0xBF 0x4D 0x37 0xC6 0x5F 0x3D 0xEF 0x56 0x11 0xD2 0x69 0xA4
   0xF6 0xB2 0x60 0xAF 0x83 0x08 0x4E 0x11 0x71 0x27 0x60 0x6F 0x9E
   0xB7 0x26 0x1B 0x3A 0x18 0xFE 0xE3 0x3C 0xFB 0xDA 0x7E 0x78 0xCA
   0x1A 0x19 0x54 0xBD 0x1A 0x55 0x20 0x3B 0x59 0x42 0x8C 0x07 0xBA
   0x36 0x82 0xE0 0x14 0xB6 0x09 0xC9 0xF5 0x57 0x5B 0x16 0x1A 0xFA
   0x87 0xAC 0x9F 0xCC 0x65 0x0A 0x4C 0x6F 0x87 0xFD 0x30 0x7D 0xB4
   0xDC 0x22 0xFB 0xB1 0x32 0x98 0x75 0x62 0xEF 0x1A 0x14 0xDC 0x5E
   0x05 0xBE 0xF3 0x1F 0xCB 0xB7 0x8A 0x8F 0x62 0xBA 0x11 0x12 0xA0
   0x3C 0xB9 0x0A 0x92 0x5E 0x8A 0xDD 0xA6 0x09 0xFF 0x68 0x82 0xF2
   0x76 0xCD 0x8D 0x05 0x61 0xBB 0x41 0x94 0xF9 0xFD 0x5C 0x72 0x71
   0x45 0x3F 0x00 0x43 0xBB 0x07 0x1D 0x85 0xFC 0xE2 0x24 0xCE 0x76
   0xFB 0x89 0xD1 0xE3 0x81 0x0C 0xE1 0x4C 0x37 0xB2 0x1D 0x60 0x40
```

Out[1]:

```
20×20 Matrix{UInt8}:
 0xbc  0xe6  0x56  0x29  0x99  0x95  …  0x2a  0x71  0x44  0x7f  0x
af  0x96
 0x72  0x57  0x13  0xdd  0x08  0x44     0x5e  0xdb  0xe1  0xef  0x
14  0x0b
 0x42  0xb8  0xf3  0x8e  0x58  0xf0     0x13  0x3e  0x5d  0xd4  0x
30  0xfb
 0x60  0xca  0xb4  0xa1  0x73  0xe4     0x4f  0xd0  0x11  0x09  0x
28  0x39
 0x1b  0x40  0x7c  0xb1  0x01  0x79     0xcd  0xd7  0xa6  0xfe  0x
7f  0x51
 0x25  0xab  0xcc  0x20  0xf9  0xcc  …  0xf9  0xbf  0xa5  0x58  0x
1f  0xc7
 0xea  0xb2  0xe4  0xf8  0x72  0x7b     0xfa  0xab  0x12  0x40  0x
82  0x7e
 0x52  0xbf  0x4d  0x37  0xc6  0x5f     0x0d  0x58  0x11  0xa7  0x
9e  0x06
 0xf6  0xb2  0x60  0xaf  0x83  0x08     0xd3  0x19  0x20  0xf6  0x
a3  0x40
 0xb7  0x26  0x1b  0x3a  0x18  0xfe     0xf3  0xfe  0x14  0x86  0x
53  0xe9
 0x1a  0x19  0x54  0xbd  0x1a  0x55  …  0x27  0xa6  0x31  0x87  0x
2a  0xe2
 0x36  0x82  0xe0  0x14  0xb6  0x09     0x8a  0xb2  0xdb  0xf2  0x
41  0x52
 0x87  0xac  0x9f  0xcc  0x65  0x0a     0xcb  0x6d  0x03  0x64  0x
cd  0x19
 0xdc  0x22  0xfb  0xb1  0x32  0x98     0xa2  0xed  0x12  0xb5  0x
ca  0xc0
 0x05  0xbe  0xf3  0x1f  0xcb  0xb7     0x79  0xfc  0x4d  0x97  0x
74  0x4a
 0x3c  0xb9  0x0a  0x92  0x5e  0x8a  …  0x9f  0x17  0xd2  0xd5  0x
5c  0x72
 0x76  0xcd  0x8d  0x05  0x61  0xbb     0x54  0x3f  0x3b  0x32  0x
e6  0x8f
 0x45  0x3f  0x00  0x43  0xbb  0x07     0x96  0x40  0x10  0xfb  0x
64  0x88
```

```
 0xfb  0x89  0xd1  0xe3  0x81  0x0c     0xa5  0x2d  0x3b  0xe4  0x
85  0x87
 0xe5  0xd7  0x05  0xd7  0x7d  0x9c     0x83  0x46  0x79  0x0d  0x
49  0x59
```

The program in Julia and JuMP:

Entrée [ ]:

```
```

Situation 3 (🌶🌶🌶):

Packing different rectangles in a minimum-area rectangle

Situation

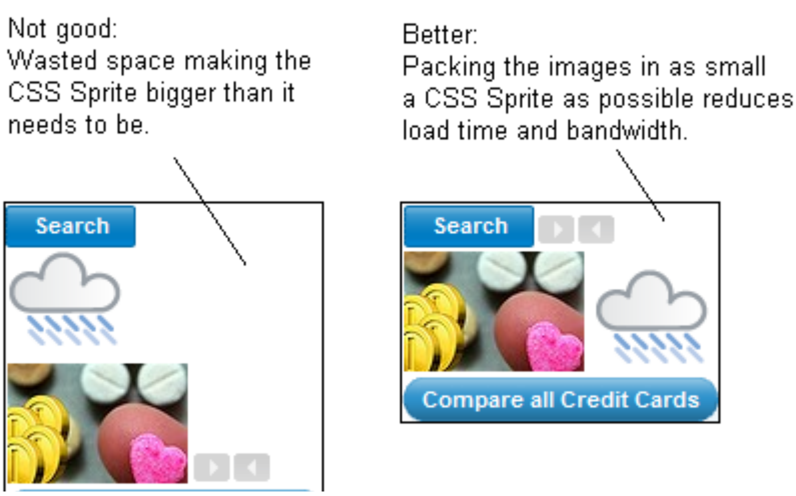
Rectangle packing is a packing problem where the objective is to determine whether a given set of small rectangles can be placed inside a given large polygon, such that no two small rectangles overlap.

Several variants exist and we consider here the variant where the objective is to pack different rectangles in a minimum-area rectangle. In this variant, the small rectangles can have varying lengths and widths, and their orientation is fixed (they cannot be rotated). The goal is to pack them in an enclosing rectangle of minimum area, with no boundaries on the enclosing rectangle's width or height.

This problem has an important application in combining images into a single larger image. A web page that loads a single larger image often renders faster in the browser than the same page loading multiple small images, due to the overhead involved in requesting each image from the web server.

(Definition from [https://en.wikipedia.org/wiki/Rectangle_packing (https://en.wikipedia.org/wiki/Rectangle_packing)](https://en.wikipedia.org/wiki/Rectangle_packing))

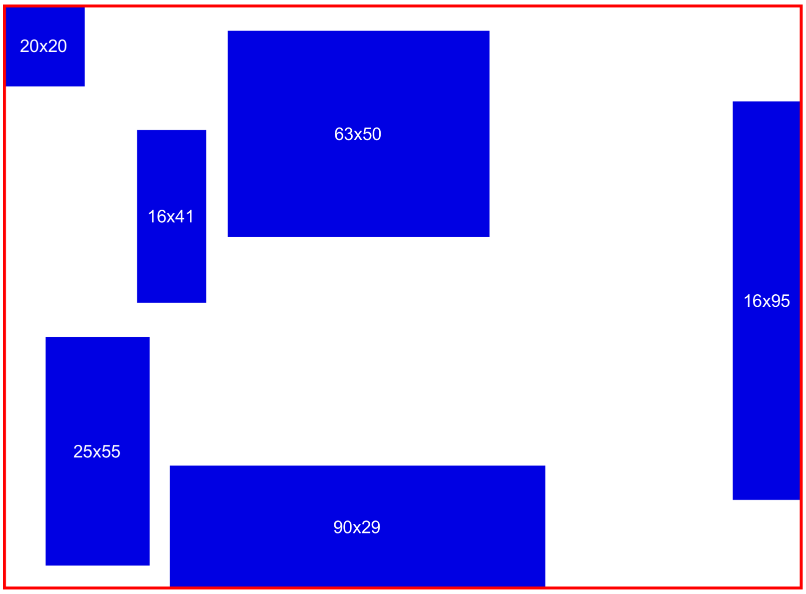
Example of the application of this optimization problem for building CSS sprites:



(Image from [https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu (https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu)](https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu))

Question

Find the minimum-area rectangle for the following rectangles:



with

```
w=[20,63,16,16,25,90]
h=[20,50,41,95,55,29]
```

Display automatically your optimal solution found using the plotting tools available in Julia.

Solution

The program in Julia and JuMP:

Entrée [ ]: