Report - Algorithmic Project IFEBY270

Table of contents

1 Introduction

An algorithmic project for a University course.

- Implemented
  - Simplexe
  - Nash Equilibrium
  - Knapsack + Reduction basis + Subset sum

2 Execution

Execute tests

```
./run_tests # first method
make test_verbose # second method
python3 -m unittest tests.<test_name> # for specific test
```

Update Gamut tests

```
make update_gamut
```

3 More Tests

To incorporate additional tests using unittest into `src/algorithm/<problem>/<problem>.py`, modify `tests/<problem>.py`.

All methods with a name beginning with test_ will be executed as tests.

Execute tests with:

```
python3 -m unittest tests.<test_name>
```

3.1 Example

To include another Nash Equilibrium example, edit `tests/nash_equilibrium.py`, and add the following method to the TestNashEquilibrium class:

```
def test_example(self):
    self.check_equilibrium(
        A = np.array([[3, 2], [1, 4]]),
        B = np.array([[2, 1], [3, 2]])
    )
```

Execute the test with:

```
python3 -m unittest tests.nash_equilibrium
```

4 Simplexe

4.1 Introduction

In linear optimization, consider a problem in canonical form:

$$\max\{c^T x \mid Ax \leq b, x \geq 0\}$$

where:
- x is the vector of variables to be determined
- c is the vector of coefficients of the objective function
- A is the matrix of coefficients of the constraints

Any problem in canonical form can be reformulated into a problem of the form:

$$\max\{c_0 + c^T x \mid Ax + x' = b, x \geq 0\, x' \geq 0\}$$

This form is known as the standard form.

The Simplex algorithm is a method to move from one basic solution to another while seeking to improve the objective function.
When an optimal basic solution is reached, i.e. one that maximizes the objective function, the algorithm stops.

4.1.1 Definition Basic solution and optimal basic solution

In the standard form below, we assume $b \geq 0$. The constant c_0 is introduced for technical reasons.
In this case, $(x, x' \ ) = (0, b)$ constitutes a basic solution of the problem, and we also refer to the variables x' as basic and the variables x as non-basic. If the problem is unbounded, it has no optimal solution, and the solution below is admissible.

4.1.2 Foreplay Data organization

Let n be the number of non-basic variables, and m be the number of basic variables.
To perform the calculations, it is useful to organize the data in a table A which initially has the following form:

_	c_1	...	c_n	0	0...	0	$-c_0$
x_{n+1}	$a_{1,1}$	...	$a_{1,n}$	1	0...	0	b_1
...	...	...	...	...	......	...	...
x_{n+m}	$a_{m,1}$	...	$a_{m,n}$	0	0...	1	b_m

On this table, we will perform the algorithm explained in the following section. At each iteration of the algorithm, we move from one basic solution to another, each time increasing the non-basic variables (while respecting the constraints).

4.2 Algorithm

Each iteration of the algorithm consists of three consecutive operations:

4.2.1 Step 1 Pivot selection

In this step, an incoming variable (non-basic) and an outgoing variable (basic) are selected as follows:

– We find $1 \leq e \leq n$ such that $c_e \geq 0$.
– There are two possible scenarios:

- $\forall i \in [n+1, m], a_{i,e} \leq 0$. In this case, the problem is unbounded and has no optimal solution.

- We look for a basic variable minimizing the quantity $b_i/a_{i,e}$. We denote it $x_s, s \in [n+1, m]$.
  We know that $a_{s,e} > 0$ and will use this value as a pivot.

4.2.2 Step 2 Table update

This is the heart of the algorithm.
Given x_e as the incoming variable, x_s as the outgoing variable, and $a_{s-n,e}$ as the pivot, we perform the following operations and obtain a new table A' (in the following order):

```
# 1) Process lines except pivot
```

- $\forall i \in [1, m+n]$, $\forall j \in [1, m], j \neq s-n$, $a'_{j,i} \leftarrow a_{j,i} - a_{s-n,i} * a_{j,e}/a_{s-n,e}$
- $\forall j \in [1, m], j \neq s-n$, $b'_j \leftarrow b_j - a_{j,e} * b_{s-n}/a_{s-n,e}$

```
# 2) Process pivot line
```

- $\forall i \in [1, m+n], a'_{s-n,i} \leftarrow a_{s-n,i}/a_{s-n,e}$
- $b_{s-n} \leftarrow b'_{s-n}/a_{s-n,e}$

```
# 3) Process coefficients line
```

- $\forall i \in [1, m+n]$, $c'_i \leftarrow c_i - c_e * a'_{s-n,i}$
- $c'_0 \leftarrow c_0 + c_e * b_{s-n}$

4.2.3 Step 3 Solution optimality

After each iteration of Step 1 and Step 2, we check whether the stop conditions have been met.
The program stops if:

- all coefficients c_i are negative. in this case, the basic solution associated with our matrix A is an optimal basic solution to our problem, and the value of our objective function as a function of this solution is equal to c_0.
- all non-basic variables have already been output. In this case, the problem has no optimal solution, and $(0, b)$ is an admissible solution.

4.3 Code

To transform a problem into equation form, and find its optimal basic solution using the Simplex algorithm:

```
simplexe = Simplexe(canonical_form)
basic_sol, obj_value = simplexe.execute_simplexe()
```

To print the table obtained at the end of the calculation:

```
simplexe.print_table()
```

4.3.1 Examples

2.3.1.1 Bounded Problem

$\max 3x_1 + x_2 + 2x_3$
$x_1 + x_2 + 3x_3 \leq 30$
$2x_1 + 2x_2 + 5x_3 \leq 24$
$4x_1 + 1x_2 + 2x_3 \leq 36$
$x_1, x_2, x_3 \geq 0$

```
canonical_form = [
    [3, 1, 2],
    [1, 1, 3, 30],
    [2, 2, 5, 24],
    [4, 1, 2, 36]
]
simplexe = Simplexe(canonical_form)
basic_sol, obj_value = simplexe.execute_simplexe()
```

```
[ ]Optimal solution found.

Basic Solution:  [8.0, 4.0, 0, 18.0, 0, 0]
Objective function value:  28.0
```

```
simplexe.print_table()
```

```
['0', '0', '-1/6', '0', '-1/6', '-2/3', '-28']
['0', '0', '1/2', '1', '-1/2', '0', '18']
['0', '1', '8/3', '0', '2/3', '-1/3', '4']
['1', '0', '-1/6', '0', '-1/6', '1/3', '8']
```

2.3.1.2 Unbounded Problem

$\max x_1 + 2x_2$
$-x_1 - x_2 \leq 3$
$-2x_1 - 3x_2 \leq 5$
$x_1, x_2 \geq 0$

```
canonical_form = [
    [1, 2],
    [-1, -1, 3],
    [-2, -3, 5],
]

simplexe = Simplexe(canonical_form)
basic_sol, obj_value = simplexe.execute_simplexe()
```

```
[]No optimal solution found.

Basic Solution:  [0, 0, 3, 5]
Objective function value:  0
```

5 Nash Equilibrium

5.1 Introduction

In game theory we consider two rational agents x and y that both want to maximize their score.
In discrete cases, x has m possible actions, y has n possible actions.

If x and y play simultaneously, we can considers the matrices A and B of dimension (m, n). $A_{i,j}$ and $B_{i,j}$ represents respectivily the score of x and y when x plays his i-th action and y his j-th action.

For convenience, we will also call player x as a, and y as b.

5.1.1 Definition Stochastic vector and Strategie

Let v a vector. If $\sum v_i = 1$ and $\forall i, v_i \geq 0$, v is a stochastic vector.

We call the strategy of the player x and y, the stochastic vectors x, y that represents the probabilty of choosing each actions. If only one coefficient of a strategy is none negative (ie. equal to one), we call it a pure strategy.

A best strategy $\overline{v}$ is a strategy that maximise the gain of its player :

- $\overline{x} \in \arg\max_x xAy^t$
- $\overline{y} \in \arg\max_x xBy^t$

Propertie :

A best strategy is always of convex combination of pure strategies.

5.1.2 Definition Zero-sum game

Zero-sum game is defined by $B = -A$, which means every gain for x is a loss of the same amplitude to y, and the other way around.

5.1.3 Definition Nash equilibrium

A nash equilibrium is a couple (x, y) of strategies, where x and y are both best strategies.

5.1.3.1 Theorem: A nash equilibrium always exists.

Remark: If we impose strategies to be pure, the theorem doesn't hold. (eg: paper, rock, scissor)

5.2 Code

To find the nash equilibrium, create a NashEquilibrium object, and solve it.

```
solution_x, solution_y = NashEquilibrium(A, B).solve()
```

5.2.1 Examples

```
# Paper Rock Scissor
A = np.array([
    [1,0,-1],
    [0,-1,1],
    [-1,1,0]
])
solution_x, solution_y = NashEquilibrium(A, -A).solve()
```

```
x: [0.33333333 0.33333333 0.33333333]
y: [0.33333333 0.33333333 0.33333333]
```

```
# Non-zero sum game
A = np.array([[1, 2], [3, 4]])
B = np.array([[4, 3], [2, 1]])
solution_x, solution_y = NashEquilibrium(A, B).solve()
```

```
x: [0. 1.]
y: [1. 0.]
```

```
A = np.array([[3, 2], [1, 4]])
B = np.array([[2, 1], [3, 2]])
solution_x, solution_y = NashEquilibrium(A, B).solve()
```

```
x: [1. 0.]
y: [1. 0.]
```

5.3 Modelization

We solve this problem using pulp with mixted linear programming (linear + discrete).

5.3.1 Variables :

Variable	Player a	Player b	Domain
Strategies	$x_{a1}, \ldots, x_{am}$	$x_{b1}, \ldots, x_{bn}$	$[0, 1]$
Strategies supports	$s_{a1}, \ldots s_{am}$	$s_{b1}, \ldots, s_{bn}$	$\{0, 1\}$
Regrets	$r_{a1}, \ldots, r_{am}$	$r_{b1}, \ldots r_{bn}$	$[0, M]$
Potential Gain	potential_gain_a	potential_gain_b	$[LG, HG]$
Max Gain (scalar)	max_gain_a	max_gain_b	$[LG, HG]$

M represents respectively the max regret for a and b.
$M_a := \max A_{ij} - \min A_{ij} \quad M_b := \max B_{ij} - \min B_{ij}$
HG represents respectively the highest gain for a and b.
$HG_a := \max A_{ij} \quad HG_b := \max B_{ij}$
LG represents respectively the lowest gain for a and b.
$LG_a := \min A_{ij} \quad LG_b := \min B_{ij}$

5.3.2 Constraints

5.3.2.1 [Eq Constraint] Stochastic vectors

$\sum x_{ai} = 1$
$\sum x_{bi} = 1$

5.3.2.2 [Eq Constraint] Potential gain

potential_gain_a $= Ay^t$
potential_gain_b $= xB$

5.3.2.3 [Constraint] Max gain

$\forall i$ max_gain_a $\geq$ potential_gain_a$_i$
$\forall j$ max_gain_b $\geq$ potential_gain_b$_j$

5.3.2.4 [Eq Constraint] Risk

$\vec{r_a} =$ potential_gain_a $-$ max_gain_a
$\vec{r_b} =$ potential_gain_b $-$ max_gain_b

5.3.2.5 [Constraint] Best strategy constraint

$$\forall i: \quad x_{ai} \leq s_{ai} \qquad r_{ai} \leq (1 - s_{ai}) \times M_a$$
$$\forall j: \quad x_{bj} \leq s_{bi} \qquad r_{bj} \leq (1 - s_{bj}) \times M_b$$

5.3.3 Objective function :

Minimize : $\sum r_{ai} + \sum r_{bi}$

6 KnapSac

6.1 Introduction

In integer linear optimization, we consider a weight capacity for a bag, a list of weights of the items, and a list of values of the items. The goal of the Knapsack problem is to maximize the value of items placed in the bag without exceeding its weight capacity.

More formally : let $b \in \mathbb{N}, w_1, ..., w_n \in \mathbb{N}^n, v_1, ..., v_n \in \mathbb{N}^n$
We want to determine $\max_i\{\sum_{i=1}^n v_i x_i | \sum_{i=1}^n w_i x_i < b, x_i \in 0, 1, i = 1, .., n\}$

6.2 Algorithm

6.3 Code

To find the solution to the KnapSack problem, create a KnapSack :

```
KnapSack(weight_capacity,array of items weight, array of items value)
```

Then chose a solving algorithm between :

- lower_bound (gives you a lower bound to the knapsack solution)
- upper_bound (gives you an upper bound to the knapsack solution)
- solve_branch_and_bound (gives you the exact solution to the knapsack)
- solve_dynamic_prog (gives you the exact solution to the knapsack)
- solve_dynamic_prog_scale_change (give you a lower_bound really close to the solution on large KnapSack)

6.3.1 Examples

```python
A = KnapSack(3,[1,1,1,2],[3,2,1,1])
lower_bound  = A.lower_bound()
upper_bound  = A.upper_bound()
solution1    = A.solve_branch_and_bound()
solution2    = A.solve_dynamic_prog()
lower_bound2 = A.solve_dynamic_prog_scale_change()
```

```
lower bound: 6
upper bound: 6.0
solution 1: 6.0
solution 2: 6
lower bound 2: 4
```

You can make solve_dynamic_prog_scale_change run faster and use less memory by passing a larger mu as parameter

```python
B = A = KnapSack(3,[1,1,1,2],[8,7,3,1])
solution = A.solve_dynamic_prog()
lower_bound3 = A.solve_dynamic_prog_scale_change(2)
lower_bound4 = A.solve_dynamic_prog_scale_change(4)
lower_bound5 = A.solve_dynamic_prog_scale_change(8)
```

```
solution: 18
lower bound 3: 16
lower bound 4: 12
lower bound 5: 8
```