Linear Programs

The Simplex Algorithm II

Bashar Dudin

October 16, 2018

EPITA

Where We Stand, What We Face

Working out a vague procedure we've managed to find supposedly optimal solutions for simple examples of linear programs (solutions were optimal, but we have no mean to know that for a fact). This heuristic is far from being satisfactory though : given a linear program L, here is what we're still missing :

- We have no way of testing if L is feasible, i.e. if it has at least a feasible solution

Where We Stand, What We Face

Working out a vague procedure we've managed to find supposedly optimal solutions for simple examples of linear programs (solutions were optimal, but we have no mean to know that for a fact). This heuristic is far from being satisfactory though : given a linear program L, here is what we're still missing :

- We have no way of testing if L is feasible, i.e. if it has at least a feasible solution
- How can we tackle the case when the basic solution is not feasible?

Where We Stand, What We Face

Working out a vague procedure we've managed to find supposedly optimal solutions for simple examples of linear programs (solutions were optimal, but we have no mean to know that for a fact). This heuristic is far from being satisfactory though : given a linear program L, here is what we're still missing :

- We have no way of testing if L is feasible, i.e. if it has at least a feasible solution
- How can we tackle the case when the basic solution is not feasible?
- How can we check if L is unbounded?

Where We Stand, What We Face

Working out a vague procedure we've managed to find supposedly optimal solutions for simple examples of linear programs (solutions were optimal, but we have no mean to know that for a fact). This heuristic is far from being satisfactory though : given a linear program L, here is what we're still missing :

- We have no way of testing if L is feasible, i.e. if it has at least a feasible solution
- How can we tackle the case when the basic solution is not feasible?
- How can we check if L is unbounded?
- Does the procedure we worked out even terminate in general?

Where We Stand, What We Face

Working out a vague procedure we've managed to find supposedly optimal solutions for simple examples of linear programs (solutions were optimal, but we have no mean to know that for a fact). This heuristic is far from being satisfactory though : given a linear program L, here is what we're still missing :

- We have no way of testing if L is feasible, i.e. if it has at least a feasible solution
- How can we tackle the case when the basic solution is not feasible?
- How can we check if L is unbounded?
- Does the procedure we worked out even terminate in general?
- Do we get an optimal objective value if procedure terminates?

Where We Stand, What We Face

Working out a vague procedure we've managed to find supposedly optimal solutions for simple examples of linear programs (solutions were optimal, but we have no mean to know that for a fact). This heuristic is far from being satisfactory though : given a linear program L, here is what we're still missing :

- We have no way of testing if L is feasible, i.e. if it has at least a feasible solution
- How can we tackle the case when the basic solution is not feasible?
- How can we check if L is unbounded?
- Does the procedure we worked out even terminate in general?
- Do we get an optimal objective value if procedure terminates?

Pivoting

Pivoting | Fixing Notation

In order to answer both previous questions we'll need to properly write down involved algorithms. Consider the linear program L

$$
\begin{array}{lll}
\text{maximize} & & z = v + \sum_{j=1}^{n} c_j x_j \\
\text{subject to} & & \\
 & \forall i \in \{1, \dots, m\}, & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\
\text{with} & \forall j \in N, & x_j \geq 0
\end{array}
$$

We write

- $\boldsymbol{A}$ for the (m, n) matrix of coefficients $\left(a_{ij}\right)_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$
- $\boldsymbol{b}$ for the m-tuple $(b_1, \dots, b_m)$
- $\boldsymbol{c}$ for the n-tuple $(c_1, \dots, c_n)$.

Pivoting | Fixing Notation

In order to answer both previous questions we'll need to properly write down involved algorithms. Consider the linear program L

$$\begin{aligned}
&\text{maximize} && z = v + \sum_{j=1}^{n} c_j x_j \\
&\text{subject to} && \\
& && \forall i \in \{1, \dots, m\}, \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\
&\text{with} && \forall j \in N, \quad x_j \geq 0
\end{aligned}$$

The linear program L in its standard form is determined by the data $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$. Initial program has $v = 0$ in general.

Pivoting | Fixing Notation

In order to answer both previous questions we'll need to properly write down involved algorithms. Consider the linear program L

$$\begin{aligned}
&\text{maximize} && z = v + \sum_{j=1}^{n} c_j x_j \\
&\text{subject to} && \\
& && \forall i \in \{1, \dots m\}, \quad x_{i+m} = b_i - \sum_{j=1}^{n} a_{ij} x_j \\
&\text{with} && \forall j \in N \cup B, \quad x_j \geq 0.
\end{aligned}$$

To encode the slack form we include the data N, B of non-basic and basic sets. A slack form is thus given by the data $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$. The set N is initialized at $\{1, \dots, n\}$ and B to $\{n+1, \dots, n+m\}$.

Pivoting | Fixing Notation

In order to answer both previous questions we'll need to properly write down involved algorithms. Consider the linear program L

$$\begin{array}{lr}
\text{maximize} & z - \sum_{j=1}^{n} c_j x_j = v \\
\text{subject to} & \\
& \forall i \in \{1, \ldots, m\}, \quad \sum_{j=1}^{n} a_{ij} x_j + x_{i+m} = b_i \\
\text{with} & \forall j \in N \cup B, \quad x_j \geq 0
\end{array}$$

To encode the slack form we include the data N, B of non-basic and basic sets. A slack form is thus given by the data $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$. The set N is initialized at $\{1, \ldots, n\}$ and B to $\{n+1, \ldots, n+m\}$. To get closer to a standard way of writing a linear system, we slightly modify presentation of slack form.

Pivoting | Tableau of a Linear Program

Linear constraints of L in slack form have an $(m, n+m)$ matrix $\underline{\boldsymbol{A}}$ given by concatenating $\boldsymbol{A}$ and $\boldsymbol{I}_m$ along rows of $\boldsymbol{A}$. The ***tableau*** T of linear program L is obtained by

Pivoting | Tableau of a Linear Program

Linear constraints of L in slack form have an $(m, n+m)$ matrix $\underline{\boldsymbol{A}}$ given by concatenating $\boldsymbol{A}$ and $\boldsymbol{I}_m$ along rows of $\boldsymbol{A}$. The ***tableau*** T of linear program L is obtained by

- concatenating $\underline{\boldsymbol{A}}$ and b^T column-wise along A

Pivoting | Tableau of a Linear Program

Linear constraints of L in slack form have an $(m, n+m)$ matrix $\underline{\boldsymbol{A}}$ given by concatenating $\boldsymbol{A}$ and $\boldsymbol{I}_m$ along rows of $\boldsymbol{A}$. The ***tableau*** T of linear program L is obtained by

- concatenating $\underline{\boldsymbol{A}}$ and b^T column-wise along A
- concatenate result with the vector $(-c_1, \ldots, -c_n, 0, \ldots 0, v)$ of size $n+m+1$ row-wise along $\boldsymbol{A}$.

Pivoting | Tableau of a Linear Program

Linear constraints of L in slack form have an $(m, n+m)$ matrix $\underline{\boldsymbol{A}}$ given by concatenating $\boldsymbol{A}$ and $\boldsymbol{I}_m$ along rows of $\boldsymbol{A}$. The ***tableau*** T of linear program L is obtained by

- concatenating $\underline{\boldsymbol{A}}$ and b^T column-wise along A
- concatenate result with the vector $(-c_1, \ldots, -c_n, 0, \ldots 0, v)$ of size $n+m+1$ row-wise along $\boldsymbol{A}$.

$$
\begin{array}{c|ccc|cccc|c}
 & \boldsymbol{x_1} & \cdots & \boldsymbol{x_n} & \boldsymbol{x_{n+1}} & \boldsymbol{x_{n+2}} & \cdots & \boldsymbol{x_{n+m}} & \\
\hline
 & -c_1 & \cdots & -c_n & 0 & 0 & \cdots & 0 & v \\
\hline
\boldsymbol{n+1} & a_{11} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 & b_1 \\
\boldsymbol{n+2} & a_{21} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 & b_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\boldsymbol{n+m} & a_{m1} & \cdots & a_{mn} & 0 & \cdots & \cdots & 1 & b_m
\end{array}
$$

The tableau T is of shape $(m+1, n+m+1)$.

Pivoting

We are given input $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and two indexes $e \in N$, $l \in B$ respectively corresponding to entering and leaving variables to and from the set of *basic* variables B.

Pivoting

We are given input $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and two indexes $e \in N$, $l \in B$ respectively corresponding to entering and leaving variables to and from the set of *basic* variables B.

- Express x_e in terms of other variables in equation l

```
T[l, :] = (1/T[l, e])*T[l, :]
```

Pivoting

We are given input $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and two indexes $e \in N$, $l \in B$ respectively corresponding to entering and leaving variables to and from the set of *basic* variables B.

- Express x_e in terms of other variables in equation l
- Replace x_e by previously obtained expression in linear constraints

```
T[l, :] = (1/T[l, e])*T[l, :]
```

```
if i != l:
  T[i, :] -= T[i, e]*T[l, :]
```

Pivoting

We are given input $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and two indexes $e \in N$, $l \in B$ respectively corresponding to entering and leaving variables to and from the set of *basic* variables B.

- Express x_e in terms of other variables in equation l
- Replace x_e by previously obtained expression in linear constraints
- Replace x_e by corresponding expression in the value function

```
T[l, :] = (1/T[l, e])*T[l, :]
```

```
if i != l:
  T[i, :] -= T[i, e]*T[l, :]
```

Pivoting

We are given input $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and two indexes $e \in N$, $l \in B$ respectively corresponding to entering and leaving variables to and from the set of *basic* variables B.

- Express x_e in terms of other variables in equation l
- Replace x_e by previously obtained expression in linear constraints
- Replace x_e by corresponding expression in the value function
- Update basic and none basic sets of variables.

```
T[l, :] = (1/T[l, e])*T[l, :]
```

```
if i != l:
  T[i, :] -= T[i, e]*T[l, :]
```

```
N.insert(N.index(e), l).remove(e)
B.insert(B.index(l), e).remove(l)
```

Pivoting | Full Function

```python
def pivot(N, B, T, e, l):
    """Pivoting in linear programs.

    Pivots entering and leaving variables in linear
    program given as tableau. Done in place.
    """
    T[l, :] = (1/T[l, e])*T[l, :]
    for i in range(m+1):
        if i != l:  # ugly
            T[i, :] -= T[i, e]*T[i, :]
    N.insert(N.index(e), l).remove(e)
    B.remove(B.index(l), e).remove(l)
```

Pivoting | Full Function

- Index e is an element of N and l one of B.

Pivoting | Full Function

- Index e is an element of N and l one of B.
- There better be no entries e, l such that `T[l, e] == 0`. We shall ensure this is never the case when `pivot` is used.

Pivoting | Full Function

- Index e is an element of N and l one of B.
- There better be no entries e, l such that `T[l, e] == 0`. We shall ensure this is never the case when `pivot` is used.
- The objective value and basic solution of obtained linear program can be read on last column to the right.

Facing Unboundedness

Testing Boundedness

Let L be the following linear program in standard form

$$
\begin{aligned}
&\text{maximize} && z = v + \sum_{j=1}^{n} c_j x_j \\
&\text{subject to} && \\
& && \forall i \in \{1,\ldots,m\},\quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\
&\text{with} && \forall j \in N,\quad x_j \geq 0
\end{aligned}
$$

Fact

If there was an index j such that $c_j > 0$ and all coefficients of x_j in the linear constraints were non-positive then L is unbounded.

Testing Boundedness

Let L be the following linear program in standard form

$$\begin{array}{lll} \text{maximize} & & z = v + \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & & \\ & \forall i \in \{1, \ldots, m\}, & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\ \text{with} & \forall j \in N, & x_j \geq 0 \end{array}$$

Fact

Equivalently, if there was a column in the program's tableau having negative first entry and only non-positive ones afterwards then L is unbounded.

Testing Boundedness

Using the previous fact we can test, each time we call `pivot`, that the input linear program ***doesn't*** satisfy the property :

> *There is an index j such that* $c_j > 0$ *and all coefficients of* x_j *in the linear constraints are non-positive.*

Testing Boundedness

Using the previous fact we can test, each time we call `pivot`, that the input linear program ***doesn't*** satisfy the property :

> *There is an index j such that $c_j > 0$ and all coefficients of x_j in the linear constraints are non-positive.*

This is a necessary healthy check, but there is yet no garantee that this condition is fullfilled when the linear program we work with is unbounded. We will not have the needed machinery to properly answer this question untill we introduce some duality.

Testing Boundedness

Using the previous fact we can test, each time we call `pivot`, that the input linear program ***doesn't*** satisfy the property :

> *There is an index j such that $c_j > 0$ and all coefficients of x_j in the linear constraints are non-positive.*

This is a necessary healthy check, but there is yet no garantee that this condition is fullfilled when the linear program we work with is unbounded. We will not have the needed machinery to properly answer this question untill we introduce some duality.

For the time being we'll have to accept that the naive check at hand will be enough.

The Simplex Algorithm : Second Try

The Simplex Algorithm *Restricted*

```python
# Under construction function!
def _simplex(N, B, T):
    """Restricted simplex algorithm

    Runs simplex algorithm on basic feasible
    linear program in slack form.

    Args:
      N, B (list[int]): lists of non-basic
      and basic variables.
      T (ndarray[float]): numpy array for
      tableau of linear program.
    Output:
      (ndarray[float], float) tuple of optimal
      point and objective value.
    """
    m, margins = len(B), dict()
    aug_var = [i for i in N if T[0, i] < 0]
    while aug_var:
      e = random.choice(augmenting_var)
      for i in range(m):
        if T[i, e] > 0:
          margins[B[i]] = T[i, -1]/T[i, e]
      if not margins:
        raise Exception("Unbounded LP")
      min_margin = min(margins.values())
      minima = [i for i in margins\
                  if margins[i] == min_margin]
      l = random.choice(minima)
      pivot(N, B, T, e, l)
      aug_var = [i for i in N if T[0, i] < 0]
    return np.array([T[i:-1] for i in range(len(N))]), T[0:-1]
```

The Simplex Algorithm : First Validity Checks

There are two things we need to check in order to temporarily accept the previous version of the simplex algorithm

The Simplex Algorithm : First Validity Checks

There are two things we need to check in order to temporarily accept the previous version of the simplex algorithm

1. after each call to `pivot` the linear program we get has feasible basic solution

The Simplex Algorithm : First Validity Checks

There are two things we need to check in order to temporarily accept the previous version of the simplex algorithm

1. after each call to `pivot` the linear program we get has feasible basic solution
2. the objective value does not decrease while looping

The Simplex Algorithm : First Validity Checks

There are two things we need to check in order to temporarily accept the previous version of the simplex algorithm

1. after each call to `pivot` the linear program we get has feasible basic solution
2. the objective value does not decrease while looping **(We choose it that way!)**

The Simplex Algorithm : First Validity Checks

There are two things we need to check in order to temporarily accept the previous version of the simplex algorithm

1. after each call to `pivot` the linear program we get has feasible basic solution
2. the objective value does not decrease while looping **(We choose it that way!)**

and a last point to *understand* :

3. what does it mean for `_simplex` not to terminate? how can we deal with it?

Feasability of the basic solution after each call for `pivot`

Input is an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ having feasible basic solution, i.e. `T[1:, -1]` is non-negative. We show effect of `pivot` leaves an LP with feasible basic solution as well. Since `pivot` is called, `T[l, e] > 0`.

Feasability of the basic solution after each call for `pivot`

Input is an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ having feasible basic solution, i.e. `T[1:, -1]` is non-negative. We show effect of `pivot` leaves an LP with feasible basic solution as well. Since `pivot` is called, `T[l, e] > 0`.

Updates of $\boldsymbol{b}$ are give by the relations

```
T[l, -1] = (1/T[l, e])*T[l, -1]
```

and for each $i \in \{1, \ldots, m+1\}$

```
T[i, -1] -= (T[i, e]/T[l, e])*T[i, -1]
```

Feasability of the basic solution after each call for `pivot`

Input is an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ having feasible basic solution, i.e. `T[1:, -1]` is non-negative. We show effect of `pivot` leaves an LP with feasible basic solution as well. Since `pivot` is called, `T[l, e] > 0`.

Updates of $\boldsymbol{b}$ are give by the relations

```
T[l, -1] = (1/T[l, e])*T[l, -1]
```

and for each $i \in \{1, \ldots, m+1\}$

```
T[i, -1] -= (T[i, e]/T[l, e])*T[i, -1]
```

The only possibility for such updates to give a negative result is when `T[i, e] > 0`. In that case `_simplex` chooses `l` in such a way that

```
T[l, -1]/T[l, e] <= T[i, -1]/T[i, e]
```

Feasability of the basic solution after each call for `pivot`

Input is an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ having feasible basic solution, i.e. `T[1:, -1]` is non-negative. We show effect of `pivot` leaves an LP with feasible basic solution as well. Since `pivot` is called, `T[l, e] > 0`.

Updates of $\boldsymbol{b}$ are give by the relations

```
T[l, -1] = (1/T[l, e])*T[l, -1]
```

and for each $i \in \{1, \ldots, m+1\}$

```
T[i, -1] -= (T[i, e]/T[l, e])*T[i, -1]
```

The only possibility for such updates to give a negative result is when `T[i, e] > 0`. In that case `_simplex` chooses `l` in such a way that

```
T[l, -1]/T[l, e] <= T[i, -1]/T[i, e]
```

Therefore, if `-T[i, e] < 0`, mutliplying previous inequality by it and adding `T[i, -1]` we get that update after `pivot` is non-negative.

Cycling

Each time we step into the `while` loop of the `_simplex` algorithm we either *increase or keep constant* the objective value or discover the LP is *unbounded*. A priori, `_simplex` might run on indefinitely among equivalent slack forms without ever increasing the objective value. We are going to check such behaviour can be detected.

Cycling

Each time we step into the `while` loop of the `_simplex` algorithm we either *increase or keep constant* the objective value or discover the LP is *unbounded*. A priori, `_simplex` might run on indefinitely among equivalent slack forms without ever increasing the objective value. We are going to check such behaviour can be detected.

Proposition C

Let L be an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ where $\boldsymbol{A}$ is an (m, n) matrix. If `_simplex` runs more than $\binom{n+m}{m}$ iterations it does cycle, i.e. it wanders indefinitely along the same finite set of slack forms with same given objective value.

Remark : This means that whenever `_simplex` runs more than $\binom{n+m}{m}$ times then one can return any current objective value and basic solution.

Cycling

The proof of the above proposition is based on two facts :

- if we ever get back to a previously obtained slack form, then we are going to have the same set of options we had for equivalent slack forms once again.
- There is only a finite set of possible slack forms for the same given linear program.

Cycling

The proof of the above proposition is based on two facts :

- if we ever get back to a previously obtained slack form, then we are going to have the same set of options we had for equivalent slack forms once again.
- There is only a finite set of possible slack forms for the same given linear program.

The second point is the only point we need to make clear.

Cycling

Lemma B

Let L be an LP given by $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$. A slack form of L appearing in `_simplex` is determined by the choice of a set B of basic variables.

Cycling

Lemma B

Let L be an LP given by $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$. A slack form of L appearing in `_simplex` is determined by the choice of a set B of basic variables.

Proof : Such slack forms $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and $(N, B, \boldsymbol{A}', \boldsymbol{b}', \boldsymbol{c}', v')$ are equivalent through the identity map.

Cycling

Lemma *B*

Let L be an LP given by $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$. A slack form of L appearing in `_simplex` is determined by the choice of a set B of basic variables.

Proof : Such slack forms $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and $(N, B, \boldsymbol{A}', \boldsymbol{b}', \boldsymbol{c}', v')$ are equivalent through the identity map. Substracting the linear constraints, we get the relations :

$$0 = (v - v') + \sum_{j \in N} (c_j - c'_j) x_j$$

and for each $i \in B$

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j.$$

We're abusing notation here ; index of coefficients is the one corresponding to line of $\boldsymbol{A}$ supporting basic variable i.

Cycling

Lemma B

Let L be an LP given by $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$. A slack form of L appearing in `_simplex` is determined by the choice of a set B of basic variables.

Proof : Such slack forms $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ and $(N, B, \boldsymbol{A}', \boldsymbol{b}', \boldsymbol{c}', v')$ are equivalent through the identity map. Substracting the linear constraints, we get the relations :

$$0 = (v - v') + \sum_{j \in N} (c_j - c'_j) x_j$$

and for each $i \in B$

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j.$$

We're abusing notation here ; index of coefficients is the one corresponding to line of $\boldsymbol{A}$ supporting basic variable i.

These relations being true for any vector $(x_1, \ldots, x_n)$ it is clear that for each $i \in B$ and $j \in N$ we have $v = v'$, $b_i = b'_i$, $c_j = c'_j$, $a_{ij} = a'_{ij}$. ■

Cycling

Proposition C

Let L be an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ where $\boldsymbol{A}$ is an (m, n) matrix. If `_simplex` runs more than $\binom{n+m}{n}$ iterations it does cycle, i.e. it wanders indefinitely along the same finite set of slack forms with same given objective value.

Proof : By lemma B, there are at most as many different slack forms as the number of possible choices of basic sets of variables. There are $\binom{m+n}{m}$ such choices. Thus if SIMPLEX runs more iterations than $\binom{m+n}{m}$ then we already obtained twice the same slack form. ∎

Proposition C

Let L be an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ where $\boldsymbol{A}$ is an (m, n) matrix. If `_simplex` runs more than $\binom{n+m}{n}$ iterations it does cycle, i.e. it wanders indefinitely along the same finite set of slack forms with same given objective value.

Proof : By lemma B, there are at most as many different slack forms as the number of possible choices of basic sets of variables. There are $\binom{m+n}{m}$ such choices. Thus if SIMPLEX runs more iterations than $\binom{m+n}{m}$ then we already obtained twice the same slack form. ■

Adding a counter to `_simplex` insures `_simplex` terminates.

Proposition C

Let L be an LP $(N, B, \boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}, v)$ where $\boldsymbol{A}$ is an (m, n) matrix. If `_simplex` runs more than $\binom{n+m}{n}$ iterations it does cycle, i.e. it wanders indefinitely along the same finite set of slack forms with same given objective value.

Proof : By lemma B, there are at most as many different slack forms as the number of possible choices of basic sets of variables. There are $\binom{m+n}{m}$ such choices. Thus if SIMPLEX runs more iterations than $\binom{m+n}{m}$ then we already obtained twice the same slack form. ■

Adding a counter to `_simplex` insures `_simplex` terminates.

Remark : This solution is ***not*** an intelligent one. There are many different solutions in practice. We shall implement ***Bland's rule***: each time we choose an index at line 21 and 30 of previous `_simplex`, we choose the smallest possible indexes.

The Simplex Algorithm *Restricted* | No Cycling Version

```python
def _simplex(N, B, T):
    """Restricted simplex algorithm

    Runs simplex algorithm on basic feasible
    linear program in slack form.

    Args:
      N, B (list[int]): lists of non-basic
      and basic variables.
      T (ndarray[float]): numpy array for
      tableau of linear program.
    Output:
      (ndarray[float]) vector tail of which
      is maximal objective value, rest is
      optimal point.
    """
    m = len(B)
    l, margin = None, float('inf')
    aug_var = [i for i in N if T[0, i] < 0]
    while aug_var:
      e = min(augmenting_var)
      for i in range(m):
        if T[i, e] > 0:
           if T[i, -1]/T[i, e] < margin:
             margin = T[i, -1]/T[i,e]
             l = i
      if not l:
        raise Exception("Unbounded LP")
      pivot(N, B, T, e, l)
      aug_var = [i for i in N if T[0, i] < 0]
      return np.array([T[i:-1] for i in range(len(N))]), T[0:-1]
```

That’s it for today !