Universidade Estadual de Campinas
Instituto de Matemática, Estatística e Computação Científica

MS428 Programação Linear

Algoritmo Primal Simplex

Método das Duas Fases

Matheus Feres Turcheti 241727
José Ricardo de Aguiar Coelho 251222
Alan Fachini Belleza 193398

Campinas

1 O Método das Duas Fases

O algoritmo primal simplex que implementamos resolve o problema de otimização linear

$$min\ f(x) = c^T \cdot x$$
$$s.a. \begin{cases} A \cdot x = b \\ x \geq 0 \end{cases}$$

em que $c, x \in \mathbb{R}^n,\ b \in \mathbb{R}^m,$ e $A \in \mathbb{R}^{m \times n}$.

A solução deste problema é obtida ao fazer iterações simplex (às quais vamos nos referir como fase II e estão resumidas na seção 2.2.2). Para iniciar tais iterações, porém, é suposto que é conhecida uma matriz $B \in \mathbb{R}^{m \times m}$ não-singular (ou seja, uma base para $\mathbb{R}^m$) formada a partir de colunas (não necessariamente consecutivas) de A. Como tal partição $A = \begin{bmatrix} B & N \end{bmatrix}$ pode não ser óbvia, é necessária uma fase preliminar (fase I) às iterações simplex, a qual será responsável por encontrar tal partição.

Esse problema, de se encontrar uma partição básica de A, pode ser resolvido de duas maneiras distintas. Uma delas mal se configura numa fase distinta das iterações simplex, pois consiste em adicionar variáveis artificiais com custo altíssimo $M \in \mathbb{R}$ (comparados ao c original) na função objetivo f. Este método é chamado **big-M** e funciona pois, durante as iterações simplex, são "expulsas" da base as colunas que se associam a variáveis mais custosas. Ou seja, o PL se torna

$$min\ f(x) = c^T \cdot x + \sum_{j=1}^{m} M \cdot u_j$$
$$s.a. \begin{cases} \begin{bmatrix} A & I_m \end{bmatrix} \cdot \begin{bmatrix} x \\ u \end{bmatrix} = b \\ x \geq 0 \end{cases}$$

E assim é possível iniciar as iterações simplex usando $B = I_m$ e $N = A$.

O outro método, com uma fase I devidamente separada da fase II, é o método que escolhemos utilizar em nossa implementação. Este é conhecido como **Método das Duas Fases**, e consiste em resolver um problema auxiliar ao original e, com a última base B do problema auxiliar, iniciar as iterações simplex. Este problema auxiliar possui solução conhecida $w^* = 0$, e serve apenas para encontrar uma base em A. Caso a solução final seja distinta da solução conhecida, significa que não foi possível encontrar B excusivamente como um conjunto das colunas de A e, por isso, o problema original é infactível.

Como nos primeiros passos de uma iteração simplex é preciso encontrar uma solução $\hat{x}_B \geq 0$ de $B \cdot x_B = b$, chamada solução básica factível, precisamos forçar que $b \geq 0$. Para isso multiplicamos algumas restrições (i.e. linhas de A e de b) por -1. Isso não altera fundamentalmente o problema, mas permite que a solução do sistema seja sempre não-negativa, sendo assim possível iniciar iterações simplex usando $B = I_m$ em

$$min\ w(x, u) = \sum_{j=1}^{m} u_j$$

$$s.a. \begin{cases} A \cdot x + I_m \cdot u = b \\ x \geq 0, u \geq 0 \end{cases}$$

Na primeira iteração obtém-se $\hat{x}_B \equiv \hat{u} \geq 0$ solução de $B \cdot x_B = b \iff I_m \cdot u = b \iff \hat{u} = b \geq 0$, e nas iterações subsequentes $x_B \neq u$ e $B \neq I_m$, mas $\hat{x}_B \geq 0$.

Ao final das iterações é esperado que $w^* \equiv w(x, u^*) = 0$, ou seja, $u^* = \mathbf{0}$ e, por consequência, B não contém nenhuma coluna de I_m, uma vez que todas as variáveis artificiais u_j estão em N. Assim podemos iniciar as iterações simplex do problema original com B, que contém apenas colunas de A.

Caso $w(x, u^*) \neq 0$, siginifica que há pelo menos uma coluna de I_m em B, ou seja, não é possível determinar uma partição básica de A. Esse caso, como já mencionado, aponta a infactibilidade do problema original.

2 Algoritmo Implementado

Escolhemos a linguagem Python e utilizamos de 5 *scripts* para implementar o algoritmo primal simplex, os quais serão descritos nas seções abaixo.

Antes, demarcamos que os arquivos *.gitattributes* e *.gitignore* foram usados apenas para fim de organizar o repositório `git` que utilizamos para versionar código. Além disso, o arquivo *README.md* contém também uma explicação do projeto (análogo, mas mais breve do que as seções abaixo, pois não repete comentários nem documentação contida no código fonte, e está em inglês).

2.1 Álgebra Linear

Para manejar os vetores e matrizes, utilizamos a biblioteca NumPy (que é referida como `np`) e a extendemos em *primal_simplex/utils.py* para deixar representações de vetores (coluna) e matrizes mais explícitas no código, além de permitir a implementação de métodos adicionais. Tais métodos adicionais merecem explicação, uma vez que performam papéis importantes no *solver*

- `__new__` é o método que permite tratar `matrix` e `vector` como tipos de dados por si mesmos
- `__call__` é o método que permite particionar facilmente uma matrix M e um vetor v usando a sintaxe de funções `M(columns)` para uma lista de índices `columns`; similarmente, `v(elements)` para uma lista de posições `elements`.
- `extented_by` retorna uma nova instância do objeto do qual é chamado e, como o nome sugere, esta instância contém mais elementos do que o original. No caso das matrizes, o método recebe uma outra matriz com o mesmo número de linhas que a original e adiciona as colunas da especificada à direita da original, isto é, `M = A.extended_by(B)` equivale a $M = \begin{bmatrix} A & B \end{bmatrix}$. Para vetores, os elementos do argumento do método são adicionados abaixo dos do original, isto é `u = v.extended_by(w)` equivale a $u = \begin{bmatrix} v \\ w \end{bmatrix}$.
- `index` é um método exclusivo dos vetores e retorna o índice da primeira ocorrência do parâmetro no vetor (isso mimica um método que existe nativamente nas `lists`, mas não nos `np.ndarrays`). Ou seja se $d = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$, então `d.index(1)` será `0`, uma vez que Python começa a contagem do 0.

As demais funções deste módulo de utilidades, o qual será referido como `ut`, são basicamente *wrappers* das contrapartes originais da biblioteca NumPy. Também queremos destacar como realizamos os produtos internos e como resolvemos os sistemas lineares que o método requer.

Os produtos internos são feitos com o operador nativo de multiplicação de matrizes de Python (o qual funciona para a nossa extensão de NumPy *arrays*) representado por `@`, ou seja $\alpha = u^T \cdot v$ equivale a `alpha = u.T @ v`, sendo que `.T` representa a transposta (logo `u.T` é como representamos vetores linha), e $u = M \cdot v$ equivale a `u = M @ v`.

Já os sistemas lineares são resolvidos usando o método `linalg.solve` do NumPy (para o qual implementamos o *wrapper* `solve_system`), recebendo uma matriz como primeiro argumento e um vetor como segundo, isto é $x : A \cdot x = b$ equivale a `x = solve_system(A, b)`.

2.2 *Solver*

O *solver* de fato é a classe `linear_problem` que está em *primal_ simplex/primal_ simplex.py*, sendo possível importá-la para fora do diretório graças a *__init__.py*.

A classe funciona como uma cápsula para armazenar todos os dados relevantes do problema linear (PL) em questão. Para iniciar o problema, é necessário criar um objeto informando o vetor de custos c, o vetor de recursos b, e a matriz de coeficientes A (todos estes em formato de `list` nativas de Python). A classe dispõe de apenas um método público, chamado `solve`, o qual, como o nome sugere, realiza iterações simplex até obter uma resposta ou chegar ao número máximo de iterações especificado. A classe também dispõe de atributos que explicitam qual é o problema e qual a sua solução:

- `c`, `b`, `A` são, como de se esperar, os dados originais do problema, exceto pela necessidade de multiplicar restrições por -1 a fim de obter um vetor `b` não negativo (o que inclui multiplicar linhas de `A` também)
- `basic` é uma lista dos índices das variáveis básicas do problema
- `nonbasic` é uma lista dos índices das variáveis não-básicas do problema
- `decision_var` é um vetor equivalendo à solução ótima x^* do problema; caso o problema seja ilimitado ou infactível, este atributo será `None`
- `optimal_value` é o valor ótimo da função objetivo $f(x^*) = z^*$; caso o problema seja ilitimado, terá valor $-\infty$ (representado por `-np.inf`); e caso o problema seja infactível, atribuímos $+\infty$ apenas para representar que não foi possível minimizar a função objetivo (mas poderíamos ter usado `None` e teríamos o mesmo efeito)

Enfatizamos que estes são atributos que estão expostos ao escopo público através do decorator `property` e que, por isso, não podem ser alterados fora do escopo da classe após a criação de um PL, apenas visualizados.

Notamos também o papel interessante do atributo privado `_message` e do *dunder method* `__repr__`, os quais servem para representar "publicamente" o PL sempre que uma instância for "printada" (ou seja, eles exibem qual a situação do método).

Para o exemplo genérico

```
from primal_simplex import linear_problem as PL
c = [ ... ]
b = [ ... ]
A = [ [ ... ], ..., [ ... ] ]
max_iterations = ...
```

```
problem = PL(c, b, A)
problem.solve(max_iterations)
print(problem)
```

Temos cinco saídas possíveis:

1. `This problem has not been solved yet.` Realisticamente essa saída é impossível, pois o método `solve` foi chamado, mas esta é a mensagem que exibimos caso o programador tenha esquecido de chamá-lo

2. `This problem is infeasible.` Saída exibida após finalizar a fase I sem sucesso, isto é, não foi possível encontrar uma partição básica factível e, por isso, o *solver* nem inicia e fase II e já provê tal infomação

3. `This problem has no finite solution.` Esta será a mensagem caso o PL em questão seja ilimitado

4. `The optimal solution is x* = x with f(x*) = z, found in P iterations.` Caso o PL tenha solução ótima finita, será exibida uma lista `x` com os valores das variáveis de decisão separadas por vírgula, e o valor ótimo `z` todos com duas casas decimais (a fim de legibilidade, mas os atributos contém os valores "cheios"), além disso exibimos um número `P` de iterações simplex que foram necessárias na fase II para chegar a tal solução

5. `K are not enough iterations.` Caso o número máximo de iterações (aqui `K`) providenciado seja baixo demais e a solução ótima não for encontrada, esta informação será exibida

2.2.1 `solve`

O método `solve` não vai muito além do que já ficou implicitamente explicado. É um método em alto nível em relação ao problema, que apenas condensa em si as etapas que devem ser feitas até atingir um critério de parada.

Isto é, o método primeiramente tenta determinar uma base e realiza iterações simplex. Caso a base não seja encontrada, a segunda saída acima é exibida. Caso contrário, uma das três últimas saídas é exibida, a depender dos valores que serão atribuídos a `_decision_var` e `_optimal_value`, isto é, a depender do resultado das iterações simplex: encontrada uma solução (finita ou infinita) ou número insuficiente de iterações. Destacamos que seu único parâmetro é o número de iterações que o usuário deseja realizar ao máximo, sendo que, se não especificado um número, o máximo de iterações será 1000.

2.2.2 `_simplex`

Este método privado tem por objetivo traduzir para linguagem de código a fase II do algoritmo apresentado na aula 10. Em essência o passo a passo é o mesmo, mas destacamos aqui nosso racional:

1. Particionamos A e c em básicos (B, c_B) e não básicos (N, c_N)

2. Encontramos uma solução básica factível resolvendo o sistema linear $B \cdot x_B = b$ (`x_B = ut.solve_system(B, self.b)`)

3. Encontramos o vetor multiplicador simplex λ resolvendo o sistema $B^T \cdot \lambda = c_B$ (`_lambda = ut.solve_system(B.T, c_B)`)

4. Calculamos os custos relativos $\hat{c}_N$ usando o próprio vetor `c_N` fazendo, para cada posição não-básica `j`, `c_N[j] -= _lambda.T @ N[:, j]` o que equivale a $\hat{c}_{Nj} = c_{Nj} - \lambda^T \cdot a_{Nj}$

5. Checamos se o menor custo relativo `c_Nk` $\equiv \hat{c}_{Nk}$ é não-negativo; se o for, paramos a iteração e retornamos a solução ótima dada por `x_B` $\equiv x_B$ (importante notar que isso significa que no caso de infinitas soluções exibiremos apenas a primeira encontrada), caso contrário guardamos o índice não-básico da variável de menor custo relativo, pois ela deverá entrar na base

6. Encontramos a direção simplex y resolvendo $B \cdot y = a_{Nk}$ (`y = ut.solve(B, a_Nk)` com `a_Nk` sendo a k-ésima coluna não-básica de A, aquela cujo custo relativo é o menor desta iteração), e chechamos se todos os elementos em y são não-positivos usando `np.all`: caso sejam concluímos que o problema é ilimitado, caso contrário prosseguimos

7. Calculamos o tamanho de passo ϵ iterando em `y` até encontrar a menor razão entre uma variável básica factível e uma coordenada positiva da direção simplex, salvamos o índice de tais posições dos vetores envolvidos e, por fim, realizamos a atualização das colunas básicas e não básicas para a próxima iteração

2.2.3 `_find_base`

Este método privado é, como o nome sugere, responsável pela fase I do método de duas fases do algorimo primal simplex, a qual encontra uma base para iniciar as iterações simplex da fase II.

Em nossa implementação, sempre adicionamos uma matriz identidade $m \times m$ e realizamos iterações simplex até encontrar o valor ótimo de uma função cujos únicos custos não-nulos estão associados às colunas da matriz que adicionamos. Isto é, resolvemos o PL

$$min\ w(x,u) = \sum_{j=1}^{m} u_j$$

$$s.a. \begin{cases} \hat{A} \cdot x + I_m \cdot u = \hat{b} \\ x \geq 0, u \geq 0 \end{cases}$$

sendo $\hat{A}$ e $\hat{b}$ a matriz de coeficientes e o vetor de recursos obtidos ao forçar que $b \geq 0$; e sendo u o vetor de variáveis artificiais criadas.

Novamente, o problema original permanece, em essência, inalterado ao multiplicarmos linhas de A por -1, por isso escolhemos manter tais modificações na matriz de coeficientes e no vetor de recursos originais (bem como citamos ao elencar os atributos da classe).

Dentro deste método já realizamos a checagem do valor ótimo $w(x, u^*) \equiv$ `w`, desta forma o que é retornado tem o mesmo valor-verdade que `w == 0`. Ou seja, se $w(x, u^*) \neq 0$, então retornamos `False`, significando que o PL original é infactível (pois não retiramos todas as variáveis artificiais u_j da base); caso contrário retornamos `True`, indicando que temos uma partição básica que aponta apenas para as colunas de A (ou $\hat{A}$, equivalentes a esse ponto).

3 Resultados

Para utilizar o código é preciso, primeiramente, possuir um interpretador Python instalado (com versão 3.10 ou superior a fim de não aparecerem problemas com os *type hints* que usamos nos parâmetros das funções e métodos) e também a biblioteca NumPy instalada em uma versão compatível com a especificada em *requirements.txt* (recomendamos usar um *virtual environment* a fim de não criar conflitos entre versões para outros projetos). Satisfeitas essas duas dependências, o código pode ser testado de duas formas disintas.

3.1 Entrada e Saída

A primeira forma é interativa, em uma interface de linha de comando (CLI), a qual inicia ao rodar o *script main.py* em um terminal. As entradas esperadas seguem o formato abaixo

```
Number of constraints: m = ?
Number of variables: n = ?
Costs vector: c = ? ? ... ?
Resources vector: b = ? ? ... ?
Matrix A = [
? ? ... ?
.
.
.
? ? ... ?
]

Specify the maximum number of iterations (0 means the default 1000): ?
```

Em que os pontos de interrogação representam as entradas do usuário (repare que os valores nos vetores e na matriz são separados por espaços e que os vetores são representados como linha apenas aqui para fins de facilitar o uso da CLI). Verifique os *screenshots* no apêndice para exemplos reais.

Após inserir o número máximo de iterações, será apresentada a saída do *solver* conforme explicada na seção 2.2. Por fim, será apresentada a opção de resolver outro problema sem interromper a execução ou de finalizar o programa (esta última sendo a opção padrão).

3.2 Testes

A segunda forma de testar nossa implementação é rodando um *script test.py*, no qual colocamos várias entradas no corpo do arquivo e, para cada uma delas, chamamos o *solver* para exibir a solução.

Este método também pode ser usado em detrimento da CLI, caso seja exaustivo demais ter que manualmente inserir problemas similares repetidas vezes.

4 Referências

1. Slides referentes às Aulas 11 e 12

2. Algortimo disponibilizado na Aula 10

5 Apêndice: Exemplos

```
└─❯ python main.py
Number of constraints: m = 3
Number of variables: n = 5
Costs vector: c = -1 0 0 0 0
Resources vector: b = 6 4 2
Matrix A = [
1 2 1 0 0
1 -1 0 1 0
0 1 0 0 1
]

Specify the maximum number of iterations (0 means the default 1000):

 The optimal solution is x* = [4.67, 0.67, 0.0, 0.0, 1.33] with f(x*) = -4.67, found in 1 iterations

Do you want to solve another problem?(y/N) y


Number of constraints: m = 3
Number of variables: n = 5
Costs vector: c = 2 1 0 0 0
Resources vector: b = 5 4 20
Matrix A = [
1 6.5 -1 0 0
2 1 0 -1 0
5 4 0 0 1
]

Specify the maximum number of iterations (0 means the default 1000):

 The optimal solution is x* = [1.75, 0.5, 0.0, 0.0, 9.25] with f(x*) = 4.0, found in 2 iterations

Do you want to solve another problem?(y/N)
```

Figura 1: Exemplos 1 e 2 de *test.py*

```
❯ python main.py
Number of constraints: m = 2
Number of variables: n = 4
Costs vector: c = -2 -2 0 0
Resources vector: b = -1 2
Matrix A = [
1 -1 -1 0
-.5 1 0 1
]

Specify the maximum number of iterations (0 means the default 1000):

 This problem has no finite solution.

Do you want to solve another problem?(y/N) y


Number of constraints: m = 2
Number of variables: n = 5
Costs vector: c = -3 -2 -1 0 0
Resources vector: b = 3 6
Matrix A = [
3 -3 2 1 0
-1 2 1 0 1
]

Specify the maximum number of iterations (0 means the default 1000):

 The optimal solution is x* = [8.0, 7.0, 0.0, 0.0, 0.0] with f(x*) = -38.0, found in 2 iterations

Do you want to solve another problem?(y/N) y


Number of constraints: m = 3
Number of variables: n = 5
Costs vector: c = -1 -3 0 0 0
Resources vector: b = 12 4 6
Matrix A = [
-3 4 1 0 0
1 -1 0 1 0
1 1 0 0 1
]

Specify the maximum number of iterations (0 means the default 1000): 10

 The optimal solution is x* = [1.71, 4.29, 0.0, 6.57, 0.0] with f(x*) = -14.57, found in 1 iterations

Do you want to solve another problem?(y/N)
```

Figura 2: Exemplos 11, 9 e 7 de *test.py*

```
======= RESTART: C:\Users\Usuari\Downloads\primal-simplex-master\main.py =======
Number of constraints: m = 2
Number of variables: n = 4
Costs vector: c = -3 4 0 0
Resources vector: b = 4 18
Matrix A = [
1 1 1 0
2 3 0 -1
]

Now specify the maximum number of iterations (0 means the default): 1000


This problem is infeasible


Do you want to solve another problem?(y/N) y
Number of constraints: m = 4
Number of variables: n = 6
Costs vector: c = -2 -2 0 0 0 0
Resources vector: b = -24 21 6 3
Matrix A = [
8 -3 -1 0 0 0
3 5 0 -1 0 0
3 -4 0 0 -1 0
1 0 0 0 0 1
]

Now specify the maximum number of iterations (0 means the default): 1000


This problem is infeasible
```

Figura 3: Exemplos de problemas infactíveis