Implementação do Simplex de Duas Fases na linguagem de programação Python[1]

Gabriel Coutinho Chaves
nUSP 15111760
gabriel.coutinho.chaves@usp.br

Resumo

Relatório do trabalho final da disciplina SME0211 - Otimização Linear, requisitado pela Profa. Marina Andretta.

1 Introdução

Dentre as opções oferecidas para a elaboração do trabalho, optou-se pela implementação do algoritmo simplex, a qual foi realizada utilizando a linguagem Python[1]. Mais especificamente, foi implementado o Simplex de Duas Fases para a determinação de uma solução básica viável inicial; o Tableau Completo para o cálculo em cada uma das iterações; e a Regra de Bland para determinar a mundança de base.

Para os testes computacionais, foram usados problemas de otimização linear do livro Pesquisa Operacional[2] e da base de dados Netlib[3]. Fora isso, o solver GLPK[4] foi utilizado para comparar os resultados dos problemas testados.

1.1 Justificativa das Escolhadas Adotadas

Optou-se pelo Simplex de Duas Fases em razão deste tipo de implementação eximir da necessidade de que uma base viável inicial fosse fornecida como *input* do algoritmo. Já o Tableau Completo e a Regra de Bland foram escolhidos por possuirem uma maior simplicidade de implementação em relação a outros tipos de realizações simplex.

1.2 Bibliotecas Utilizadas

Foi feito uso extensivo da biblioteca NumPy[5], especialmente das **N-dimensional arrays** para a maior parte da manipulação de vetores e matrizes durante a execução do algoritmo. Além disso foram utilizados os métodos: **numpy.concatenate** (para concatenar arrays), **numpy.dot** (opera o produto interno entre dois arrays), **numpy.identity** (que cria uma matriz identidade nxn), **numpy.nanargmin** (que retorna o menor valor numérico e ignora os *not a number* — **numpy.nan**), **numpy.where** (que retorna todos os indices em uma array em que certa sentença é verdadeira), **numpy.any** e **numpy.all** (respectivamente, para verificar se algum ou todos os elementos de um array satisfazem certa condição).

Também foram utilizadas as bibliotecas PuLp[6], para leitura e manipulação de problemas de programação linear em arquivos no formato MPS[7], e *time*[1] para o cálculo do tempo de solução.

1.3 Funções Preliminares

Antes do algoritmo propriamente dito, foi necessária a criação de funções preliminares para o funcionamento do simplex: as abstrações *pivot* e *criar_tableau*.

A função *pivot* recebe como argumentos uma array, uma linha i e uma coluna j e retorna uma matrix (array) obtida a partir de operações de soma e multiplicação por escalar das linhas de modo que o elemento da i-ésima linha e j-ésima coluna (o pivô) seja igual a 1 e os demais elementos da j-ésima coluna sejam nulos. A função *pivot* é usada para atualizar o *tableau*.

Já a função *criar_tableau* recebe o custo atual, o custo reduzido, o vetor RHS (*Right Hand Side* — o vetor dos valores à direita das restrições) — e a matrix ($B^{-1}A$) e concatena esses arrays retornando a matriz correspondente a determinado estado do tableau completo — com o custo atual como elemento $a_{0,0}$ seguido dos custos reduzidos nas demais colunas da linha 0 e das variáveis básicas na coluna 0, e as linhas e colunas restantes referentes a matriz $B^{-1}A$

2 Algoritmo Simplex

O algoritmo simplex foi implementado como uma função que recebe como argumentos o vetor dos custos *c*, a matriz dos coeficientes *A* e o vetor RHS *b* de um problema de otimização linear **na forma padrão**, além de um coeficiente p opcional de arrendondamento (por padrão 1) para lidar com erros numéricos e um parametro m que determina os *outputs*. Logo no início é criada uma variável i correspondente ao número de iterações do algoritmo, a qual é atribuido inicialmente o valor 0. Ademais usa-se o método *time.time* para guardar o tempo inicial.

A primeira operação realizada checa se as matrizes têm tamanhos compátiveis, isto é, se o número de colunas de *A* é igual ao tamanho de *c* e o número de linhas de *A* é igual ao tamanho de *b* (condição para que o produto matricial $Ax = b$ exista). Se o tamanho dos arrays for incompatível o algoritmo devolve uma mensagem que denuncia a incompatibilidade e é terminado.

Caso os arrays sejam compatíveis, é iniciada a Primeira Fase do Simplex, que consiste na resolução de um problema auxiliar para determinar uma base inicial viável.

2.1 Fase I do Simplex

Nessa etapa, pretende-se produzir um *tableau* para o problema auxiliar. Cria-se então uma matriz $A\_auxiliar$ concatenando uma identidade mxm (em que m é o número de restrições do problema) às colunas de A; e um vetor $c\_auxiliar$ de tamanho $m + n$ (em que n é o número de variáveis do problema original), tal que as n primeiras entradas são nulas e as outras m entradas são iguais a 1. A partir daí, calculam-se o custo atual e custo reduzido para a base B (que nesse caso é sempre a identidade mxm formada pelas m últimas colunas de $A\_auxiliar$). Também é criado uma array B com os índices das colunas na base. Como a base inicial para o problema é sempre as m últimas colunas da matrix $A\_auxiliar$, B inicialmente é composto pelos inteiros no intervalo $[n, n + m]$.

2.2 Iteração do Simplex

Com o tableau para o problema auxiliar formado, pode-se iniciar as iterações do algoritmo, processo o qual será o mesmo para a resolução do problema original na Fase II do Simplex e, por essa razão, optou-se por abstrair essa parte do código numa função denotada *iteracao_simplex*.

A função *iteracao_simplex* recebe como argumentos o *tableau* (como uma array), a base B (correspondente ao estado do *tableau*), o número atual de iterações i e o coeficiente p de arredondamento. Dentro da função é iniciado um loop *while* que verifica se algum dos elementos da primeira linha do *tableau*, a partir da segunda coluna, é menor que 0, isto é, o loop continua somente se algum dos custos reduzidos for negativo. Nessa etapa, o arredondamento é importante pra garantir a convergência, então a constante p de arredondamento é utilizada.

Dentro desse loop while, usa-se o método np.where pra procurar o primeiro elemento negativo da primeira linha do *tableau* (a partir do 2ª elemento, ou seja, desconsideranto o custo atual), que será a variável que vai "entrar na base" e guarda-a na variável $index\_j$. Em seguida, cria-se uma cópia dessa j-ésima coluna (a partir da 2º linha, para ignorar a linha dos custos reduzidos) na qual é atribuido a todos os valores não positivos o tipo **np.nan**, subsequentemente dividindo element-wise (elemento por elemento) a primeira coluna do *tableau* por essa cópia.

No array obtido dessa divisão, procura-se, usando o método **np.argmin**, o menor valor. Como está sendo usada a Regra de Bland, se houver empate (tiver dois ou mais elementos que sejam os menores

do array), toma-se o primeiro valor (de menor indice) e guarda-se seu índice na variável $index\_i$, correspondente à variável que "sai da base".

Com esses dois índices, remove-se a variável $index\_i$ na lista B, dos índices básicos, e a substitui pelo $index\_j$. Em seguida chama-se a função *pivot* que então recebe como argumentos o *tableau* atual, o índice que entra na base ($index\_j$) e o que sai da base ($index\_i$) e é operado o pivotamento. O resultado se torna o novo tableau atualizado para a nova base B. Por fim, o número de iterações i é acrescido de 1 e uma iteração é encerrada.

O processo continua até que a condição do *loop while* não seja mais verdadeira, isto é, os custos reduzidos sejam todos não negativos. Quando isto acontecer, a função retorna o último *tableau* atualizado, a base B correspondente e o número de iterações i. Assim, é obtida a solução do problema auxiliar e, a partir dela, pode-se obter uma base viável para o problema original.

2.3 Obtenção da Base Inicial Viável

Ao tableau resultante da função *iteracao_simplex* aplicada no problema auxiliar é checado se o custo reduzido, o elemento da coluna 1 e da linha 1 do tableau, é nulo. Caso seja não nulo, o algoritmo informa que o problema é inviável e encerra. Do contrário, o problema é comprovado viável.

Em seguida, checa-se existência de variáveis artificiais na base B resultante da função *iteracao_simplex*, procurando na lista B índices maiores que n, o número total de variáveis do problema original. Caso hajam, checa-se a linha correspondente a variável básica do índice encontrado. Se a linha for toda nula até a n-ésima coluna (em que n é o número de variáveis do problema original), então a matriz A original tinha linhas linearmente dependentes e tal linha é descartada, bem como o índice é retirado da base B, que fica com um elemento a menos (isso para cada variável artificial encontrada).

Caso a linha não seja toda nula, toma-se o primeiro elemento não nulo para entrar na base. É chamada a função *iteração_simplex* que recebe como argumentos o *tableau* atual, o índice da variável artificial e o índice desse primeiro elemento não nulo. Realiza esse processo até que todas as variáveis artificais saiam da base B.

Assim, toma-se a primeira coluna (a partir da segunda linha, desconsiderando o custo atual) que contém as variáveis básicas e forma uma solução básica viável x_B.

2.4 Fase II do Simplex

Com uma base B inicial e uma solução básica viável x_B, aproveita-se o *tableau* obtido da Fase I do Simplex (a partir da 2ª linha, ignorando o custo reduzido) para obter a matrix $B^{-1}A$. Assim pode-se calcular o custo atual e o custo reduzido, que completam o necessário para montar o *tableau* para o problema original com a função *criar_tableau*. Em seguida chama-se a função *iteracao_tableau* que recebe como argumentos o *tableau*, a base B viável para o problema original, o número corrente de iterações i, e repete-se o mesmo processo descrito anteriormente.

Ao fim das iterações, a função *simplex* retorna o custo ótimo que é o elemento na linha 1 e coluna 1 do tableau resultante da segunda fase do simplex, e printa a solução básica encontrada, além do tempo de execução e o número i de iterações.

3 Resultados Computacionais

O algoritmo foi testado em 3 problemas livro Pesquisa Operacional e em 6 problemas da base de dados Netlib, que continham um gabarito com solução. O algoritmo se comportou bem até a faixa de 500 variáveis. Para além disso, os resultados não foram satisfatórios. Em comparação com o *solver* GLPK, a precisão das soluções obtidas foi alta. O mesmo não pode ser dito do tempo de solução e de iterações, nos quais o algoritmo foi muito inferior.

Também foram testados 3 problemas inviáveis (ITEST2, ITEST6 e QUAL, todos da base Netlib) para testar se o algoritmo seria capaz de identificar a inviabilidade, o que foi confirmado.

Resultados Computacionais

Problema	Variáveis	Restrições	Iterações	Tempo (s)	Tempo GLPK
Exemplo 2.1	3	3	5	0,0056	0
Exemplo 2.3	6	5	9	0,009	0
Exemplo 2.4	12	7	17	0,06	0
AFIRO	32	28	52	0,014	0
SHARE2B	79	97	491	0,50	0
ADLITTLE	97	57	427	0,28	0
SCAGR7	140	130	403	0,5	0
SC205	203	206	789	1,5	0
SCAGR25	500	472	2647	11	0,01

4 Conversor de MPS

Uma vez que os problemas presentes na database Netlib estavam comprimidos em formato MPS, algumas medidas tiveram que ser tomadas para transformar esses arquivos em inputs que pudessem ser usados pelo algoritmo simplex.

Antes de tudo, foi usado um código em C (disponibilizado no site Netlib) para descomprimir os arquivos e colocá-los em formato MPS. Em seguida fez-se um módulo em Python para converter problemas em MPS para a forma padrão, em arrays do Numpy, de modo que pudessem ser recebidos como argumentos da função *simplex*.

Para tanto, utilizou-se primeiro a biblioteca PuLp para ler e transformar os arquivos MPS em dicionários do Python[1] contendo as chaves: "objective" cotendo informações da função objetivo, como qual o coeficiente multiplicando cada variável; "constraints", contendo informação sobre as restrições, os coeficientes multiplicando cada variável, se são desigualdades e de que tipo etc; "variables", contendo informações das variáveis.

A partir dessas informações no dicionário, pôde-se contar o número de restrições de desigualdade (seja de menor ou igual ou de maior ou igual) e a partir disso criar as variáveis de folga necessárias para por o problema na forma padrão, assim criando um vetor c dos custos, um vetor b das restrições e a matriz A dos coeficientes, todos numpy arrays.

Bibliografia

[1] PYTHON SOFTWARE FOUNDATION. Python Language Site: Documentation, 2023. Página de documentação. Disponível em: <https://www.python.org/doc/>. Acesso em: 21 de nov. de 2023.

[2] ARENALES, M.; ARMENTANO, V. A.; MORABITO, R.; YANASSE, H. H. Pesquisa Operacional. Rio de Janeiro: Campus/elsevier, 2007. 523 p. ISBN 10-85-352-145-1454-2.

[3] NETLIB. Banco de dados. Disponível em: https://www.netlib.org/lp/data/. Acesso em: 21 nov. 2023.

[4] OKI, E. GLPK (GNU Linear Programming Kit), 2012. Disponível em https://www.gnu.org/software/glpk/. Acesso em: 21 nov. 2023.

[5] NUMPY. Documentação da biblioteca, 2023. Página de documentação. Disponível em: <https://numpy.org/doc/stable/>. Acesso em: 21 de nov. de 2023.

[6] PULP: A LINEAR PROGRAMMING TOOLKIT FOR PYTHON, 2023. Página de documentação. Disponível em: <https://coin-or.github.io/pulp/technical/index.html>. Acesso em: 21 de nov. de 2023.

[7] WILLIAMS, H. C. **Advanced linear programming**, by Bruce A. Murtagh. 1981. ISBN 0-07-044095-6