Práctica 2

Implementación del algoritmo del Simplex Primal

Pedro López Sancha, Emilia Vayssier

November 14, 2018

En este trabajo se ha implementado el algoritmo del Simplex Primal en dos lenguajes de programación distintos: C++ y MATLAB.
Se programaron dos versiones del algoritmo que difieren en la forma de elegir de la variable no básica de entrada a la base. La primera versión utiliza la Regla de Bland, mientras que la segunda utiliza la Regla del coste reducido más grande. Ambas versiones constan de las Fases I y II.
Se resolvieron los problemas planteados utilizando las dos versiones y se presenta una comparación del rendimiento. Los resultados vienen acompañados de la actualización de los valores a cada iteración.

1 Implementación en C++

A pesar de que se recomendó emplear MATLAB como lenguaje de programación, hemos decidido implementar el algoritmo primeramente en C++ como divertimento, y también como práctica para otras asignaturas como Algoritmia.
Para ello, primeramente ha sido necesario crear una clase Matriz que permita trabajar con este tipo de objetos de manera cómoda. Posteriormente, tras depurar el código de la clase Matriz y tras una gran cantidad de pruebas, hemos conseguido desarrollar el Simplex primal, y hacer que este funcione correctamente.

1.1 Clase Matriz

La clase Matriz se ha implementado siguiendo el paradigma de la programación orientada a objetos. En esta sección se discutirán brevemente las funcionalidades básicas de la clase Matriz que se emplean en el algoritmo del Símplex.

- La matriz internamente se guarda como un vector de vectores. Es por este motivo por el que se emplea el contenedor *vector* de la librería estándar de C++. Esto además permite acceder al elemento a_{ij} con una notación relativamente natural como es $matriz[i][j]$.
- Internamente se guardan las dimensiones de la matriz. Si en cualquier momento se accede a una posición que no pertenece a la matriz, las funciones de la clase devuelven 0.
- Las funciones que incorpora la clase Matriz para las operaciones básicas de matrices son:
  - Producto de una matriz por un escalar: *scalarProduct* que recibe como parámetros una matriz y un escalar, y devuelve el producto de ambos.
  - Suma de matrices: *matrixSum* que recibe como parámetros dos matrices y devuelve su suma.
  - Producto de matrices: *matrixProduct* que recibe como parámetros dos matrices y devuelve su producto.
  - Matriz transpuesta: *transpose*, que recibe como parámetro una matriz y devuelve su transpuesta.
  - Determinante: *determinant* que recibe como parámetro una matriz y calcula el determinante usando menores. Cabe decir que es una manera ineficiente de calcularlo, por ello el tiempo de ejecución para matrices 10×10 como las proporcionadas en datos es del orden de 15 segundos. Por lo tanto, las funciones adjunta y inversa no se ejecutan para matrices grandes.
  - Matriz adjunta: *adjoint*, que recibe como parámetro una matriz y devuelve su matriz de adjuntos.
  - Matriz inversa: inverse, que recibe como parámetro una matriz y calcula la inversa.
- Las funciones para la modificación de matrices son:
  - Inserción de fila y columna: *insertRow* y *insertCol*, que reciben como parámetros un vector de reales y la fila o columna en la cual se debe insertar.
  - Inserción de matriz en fila y columna: *insertMatrixInRow* y *insertMatrixInCol*, que reciben como parámetros una matriz de las dimensiones adecuadas y la fila o columna en la cual se debe insertar.

  - Reemplazo de fila y columna: *replaceRow* y *replaceCol*, que reciben como parámetros un vector y la fila o columna por la cual se debe reemplazar.
  - Redondeo de matriz: *roundMatrix*, que permite redondear todos los valores reales a su entero más cercano si están a una distancia de este inferior o igual a la suministrada como parámetro. Esta función es útil ya que en operaciones entre matrices grandes, el cálculo de números reales pierde precisión.

- Otras funciones relevantes son:
  - Matriz identidad: *identity*, que retorna una matriz identidad de la dimensión dada.
  - Información: *info* y *print*, que imprimen por pantalla información sobre el objeto.

1.2 Implementación del Simplex Primal

El algoritmo del símplex se ha implementado mediante dos archivos, el archivo de cabecera *Simplex.h* que contiene las declaraciones de las funciones, y el archivo de implementación *Simplex.cc*, que contiene las definiciones de dichas funciones. Se ha eligido esta metodología ya que es la recomendad por la comunidad de programación.

1.2.1 Fases del Símplex

Se han implementado las dos fases del Símplex, de modo que el programa recibe las matrices que representan el problema y este resuelve de manera autónoma la fase 1 y, si determina que es factible, continúa con la fase 2. Cada fase tiene su propia función, como se verá en el próximo apartado. Primeramente, mostramos brevemente las operaciones de cada fase:

- Fase 1: primeramente, a partir de la matrices de coeficientes A, inserta la matriz identidad correspondiente a las variables artificiales de fase 1. También crea los vectores de coeficientes de la función objetivo de fase 1, así como el vector de variables básicas y no básicas. Por último, iguala la solución básica factible inicial X_b al vector de términos independientes b y calcula el valor de la función objetivo. A continuación, comienza a iterar el algoritmo. Una vez acaba, si el óptimo de la función objetivo es $z = 0$, o un valor cercano por error de aproximación, determina que el problema es factible. En tal caso, retorna la base óptima primal y su inversa.

- Fase 2: a partir del vector de variables básicas y no básicas, crea los vectores de coeficientes de la función objetivo así como la matriz no básica A_n. Calcula el valor de la función objetivo en la base actual y, en lugar de calcular la inversa de la base, recibe la de fase 1. Seguidamente comienza a iterar el Símplex.

Ambas funciones tienen variables que determinan si estamos en el óptimo, si el problema es ilimitado o si es degenerado. El algoritmo itera mientras ninguna de estas condiciones se cumpla. En el momento que una de estas condiciones se da, para su ejecución y dice qué ha sucedido. Las funciones se llaman desde el *main* del programa. Entre las llamadas a ambas y, en caso de que el problema sea factible, se comprueba primeramente si han quedado variables artificiales en la base. En caso afirmativo, para la ejecución del programa. Si no, revisa el vector de variables no básicas y elimina aquellas que sean artificiales.

1.2.2 Funciones en la implementación

Cada paso del algoritmo tiene su propia función. De este modo, el código es más escalable, estructurado, legible y facilita la comprensión. Las funciones declaradas en el archivo de implementación són:

- Cálculo de costes reducidos: *computeReducedCosts*, que calcula el vector de costes reducidos. También determina si la solución básica factible actual es óptima, y en caso de que no lo sea, devuelve el índice q de la variable no básica que debe entrar en la base y su posición en el vector de variables no básicas. En los argumentos de la función, se especifica si empleamos o no la regla de Bland y también en qué fase del Simplex estamos.

- Cálculo de la dirección básica factible: *computeBasicFeasibleDirection*, que calcula la dirección básica factible en caso de que no estemos en el óptimo. También determina si nos encontramos ante un problema ilimitado buscando una componente negativa del vector.

- Cálculo de la longitud de paso: *computeTheta*, que calcula la longitud de paso a lo largo de la dirección básica factible. Se ejecuta tan solo en caso de que el problema no sea ilimitado.

- Actualización de matrices: *updateMatrices*, que actualiza los vectores de coeficientes de variables básicas c_b y no básicas c_n, la matriz A_n de coeficientes de variables no básicas, y la inversa de la base B^{-1}. Esta última se calcula con la actualización de la inversa, en lugar de calcular la nueva matriz de base B y posteriormente calcular la inversa.

- Actualización de variables: *updateVariables*, que actualiza la función objetivo, las componentes de la solución básica factible actual, y el vector de variables básicas y no básicas.

- Degeneración de la base: *isDegenerate*, que busca alguna variable con valor 0 en el vector la solución básica factible actual.

- Información: *printIteration*, que imprime la información relevante sobre la última iteración.

1.3 Errores de la implementación

Como ya se ha mencionado anteriormente, en cálculos entre matrices de dimensiones del orden de 10×10, la pérdida de precisión en operaciones entre números reales es significativa. Por esta razón, a medida que el algoritmo itera, las matrices con las que opera se alejan más de las que deberían realmente ser.
A continuación se muestra un ejemplo que ilustra lo mencionado.

```
Phase I
        Iteration 1:    q = 1   rq = -202       p = 8   B(p) = 29       Theta* = 2.56322        Z = 2384.23
        Iteration 2:    q = 2   rq = -253.816   p = 2   B(p) = 23       Theta* = 0.42425        Z = 2276.55
        Iteration 3:    q = 3   rq = -373.869   p = 3   B(p) = 24       Theta* = 0.975162       Z = 1911.97
        Iteration 4:    q = 4   rq = -848.986   p = 1   B(p) = 22       Theta* = 0.158557       Z = 1777.35
        Iteration 5:    q = 5   rq = -325.568   p = 9   B(p) = 30       Theta* = 0.269883       Z = 1689.49
        Iteration 6:    q = 6   rq = -715.174   p = 5   B(p) = 26       Theta* = 0.4805         Z = 1345.85
        Iteration 7:    q = 8   rq = -271.827   p = 6   B(p) = 27       Theta* = 2.22432        Z = 741.218
        Iteration 8:    q = 7   rq = -135.811   p = 2   B(p) = 2        Theta* = 2.83475        Z = 356.226
        Iteration 9:    q = 11  rq = -431.24    p = 7   B(p) = 28       Theta* = 0.201306       Z = 269.414
        Iteration 10:   q = 9   rq = -608.332   p = 0   B(p) = 21       Theta* = 0.238134       Z = 124.55
        Iteration 11:   q = 2   rq = -76.2144   p = 4   B(p) = 25       Theta* = 1.6342 Z = -6.06112e-014
        Optimal Not unlimited   No Degenerate
        B = {9,4,7,3,2,6,8,11,1,5}      N = {29,23,24,22,30,26,25,27,21,10,28,12,13,14,15,16,17,18,19,20}
        Basic feasible solution found, iteration 12
Phase II
        Iteration 1:    q = 12  rq = -149.747   p = 7   B(p) = 11       Theta* = 3.62458        Z = -88.1924
        Iteration 2:    q = 10  rq = -128.59    p = 1   B(p) = 4        Theta* = 0.441805       Z = -145.004
        Iteration 3:    q = 13  rq = -101.527   p = 1   B(p) = 10       Theta* = 0.555777       Z = -201.43
        Iteration 4:    q = 14  rq = -109.127   p = 0   B(p) = 9        Theta* = 0.657269       Z = -273.156
        Iteration 5:    q = 11  rq = -11.9723   p = 2   B(p) = 7        Theta* = 0.169718       Z = -275.188
        Iteration 6:    q = 18  rq = -2.85322   p = 6   B(p) = 8        Theta* = 6.57399        Z = -293.945
        Iteration 7:    q = 7   rq = -58.5947   p = 2   B(p) = 11       Theta* = 0.146013       Z = -302.501
        Iteration 8:    q = 19  rq = -0.108209  p = 6   B(p) = 18       Theta* = 43.2102        Z = -307.176
        Iteration 9:    q = 4   rq = -27.3037   p = 7   B(p) = 12       Theta* = 0.519841       Z = -321.37
        Optimal Not unlimited   No Degenerate
        Basic feasible solution found, iteration 10     Z* = -321.37
```

Figure 1: problema 1 del conjunto 45 con regla de Bland

```
Phase I
        Iteration 1:    q = 7   rq = -409       p = 1   B(p) = 22       Theta* = 0.616162       Z = 2649.99
        Iteration 2:    q = 10  rq = -590.818   p = 2   B(p) = 23       Theta* = 0.247991       Z = 2503.47
        Iteration 3:    q = 9   rq = -678.651   p = 5   B(p) = 26       Theta* = 1.02529        Z = 1807.66
        Iteration 4:    q = 14  rq = -569.325   p = 9   B(p) = 30       Theta* = 0.763408       Z = 1373.03
        Iteration 5:    q = 6   rq = -606.655   p = 3   B(p) = 24       Theta* = 0.239007       Z = 1228.04
        Iteration 6:    q = 13  rq = -583.057   p = 6   B(p) = 27       Theta* = 0.405983       Z = 991.325
        Iteration 7:    q = 11  rq = -697.292   p = 3   B(p) = 6        Theta* = 0.729863       Z = 482.398
        Iteration 8:    q = 5   rq = -1002.86   p = 0   B(p) = 21       Theta* = 0.195187       Z = 286.653
        Iteration 9:    q = 3   rq = -417.187   p = 7   B(p) = 28       Theta* = 0.516764       Z = 71.0655
        Iteration 10:   q = 1   rq = -140.949   p = 8   B(p) = 29       Theta* = 0.248542       Z = 36.0338
        Iteration 11:   q = 4   rq = -73.6344   p = 4   B(p) = 25       Theta* = 0.489361       Z = 3.18183e-013
        Optimal Not unlimited   No Degenerate
        B = {5,7,10,11,4,9,13,3,1,14}   N = {29,2,28,25,21,24,22,8,26,23,6,12,27,30,15,16,17,18,19,20}
        Basic feasible solution found, iteration 12
Phase II
        Iteration 1:    q = 12  rq = -75.5435           p = 8   B(p) = 1        Theta* = 0.31296        Z = 34.1644
        Iteration 2:    q = 15  rq = -1.08217           p = 4   B(p) = 4        Theta* = 26.545         Z = 5.43829
        Iteration 3:    q = 11  rq = -1.13687e-013      p = 3   B(p) = 11       Theta* = 0.853432       Z = 5.43829
        Rn = 284.173    259.095 5.68434e-014            201.879 66.9088         0.30626 2.66045 0.910129        2.82346 3.42611
        Optimal Not unlimited   No Degenerate
        Basic feasible solution found, iteration 4      Z* = 5.43829
```

Figure 2: problema 1 del conjunto 45 con coste reducido más negativo

Como se puede observar, aplicando regla de Bland el algoritmo da el mismo óptimo que la solución proporcionada en el conjunto de datos. Sin embargo, aplicando la regla del coste reducido más negativo, llega un momento en el que el vector de costes es no negativo y el algoritmo concluye que es óptimo. Como se puede ver, hay un valor muy cercano a cero, que podría ser negativo. En tal caso, el algoritmo habría seguido iterando.

1.4 Posibles mejoras

Como se ha mostrado, la implementación en C++ no funciona completamente, y creemos que es debido a los errores de cálculo que tiene el propio lenguaje. La solución a este problema está fuera de nuestros alcance. No obstante, en caso de descubrir o implementar una librería que esté libre de este tipo de errores, podríamos incluir las siguientes mejoras:

- En caso de que encuentre degeneración y de estar usando la regla del coste reducido más negativo, no parar la ejecución del algoritmo, sino crear una lista con todas las bases por las que el algoritmo ha pasado a partir de la detección de degeneración. Así, tras cada iteración, podríamos comprobar si la base actual ya ha sido visitada. En caso afirmativo, probablemente indicaría que el Símplex está ciclando, y por tanto el algoritmo no asegura su convergencia.

2 Implementación en MATLAB