FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

Semestrální práce z KIV/PC

Nástroj pro řešení úloh lineárního programování

Tomáš Vítek
A21B0316P
twitty@students.zcu.cz

4. 1. 2025

Obsah

1 Zadání

Naprogramujte v jazyce ANSI C přenositelnou **konzolovou aplikaci**, která bude řešit úlohy lineárního programování zadané ve zjednodušeném formátu LP.

Program bude spouštěn příkazem lp.exe2s kombinací následujících argumentů – výrazy v lome-ných závorkách (<>), resp. hranatých závorkách ([]) označují povinné, resp. nepovinné argumenty:

<input-file>	Soubor s popisem úlohy ve formátu LP. V případě, že uživatel zadá neexistu- jící soubor, program vypíše chybové hlášení `"Input file not found!\n"`a vrátí hodnotu 1.
-o <path>	Výstupní soubor s řešením úlohy. Pokud umístění neexistuje, bude vypsáno hlášení `"Invalid output destination!\n"` a program skončí s návrato-vou hodnotou 2. V případě, že uživatel tento přepínač nezadá, bude výsledek optimalizace vypsán na obrazovku. Do tohoto souboru neuvádějte chybová hlášení.
–output <path>	Stejné jako v případě přepínače -o. Použití obou přepínačů -o a --output není chybou, program pak bude akceptovat poslední zadanou hodnotu.

V případě nalezení konečného optimálního řešení úlohy program vrátí hodnotu EXIT_SUCCESS. Chybové stavy týkající se zpracování vstupních souborů nebo samotného algoritmu optimalizace jsou popsány v dalších sekcích.

Hotovou práci odevzdejte v jediném archivu typu ZIP prostřednictvím automatického odevzdávacího a validačního systému. Postupujte podle instrukcí uvedených na webu předmětu. Archiv nechť obsahuje všechny zdrojové soubory potřebné k přeložení programu, **Makefile** pro Windows i Linux (pro překlad v Linuxu připravte soubor pojmenovaný `Makefile` a pro Windows Makefile.win) a dokumentaci ve formátu PDF vytvořenou v typografickém systému TEX(LATEX). Bude-li některá z částí chybět, kontrolní skript Vaši práci odmítne.

Specifikace vyhodnocovaných výrazů

Pro zachycení optimalizačního modelu bude program používat redukovanou a zobecněnou verzi formátu LP. Vstupní soubory mohou obsahovat následující sekce:

`Maximaze/Minimaze` — Výraz uvozující řádek se zápisem optimalizované účelové funkce. Pro-gram musí být schopen zpracovat standardní operátory +, -, *, = nebo závorky (), [] a {}. Oproti originální verzi ovšem nevyžadujeme, aby jednotlivé operandy a operátory byly v matematických výrazech striktněodděleny mezerou (to platí i v ostatních sekcích souboru). Názvy pro-měnných tedy dříve uvedené operátory a závorky obsahovat nesmí. Při násobení není nutné použít operátor *, například "2.5z"značí *2,5 krát* z, zatímco "z2"je pouze název proměnné.

`Subject To` — Sekce obsahující seznam podmínek ve formátu `"<název>: <výraz>"`. Navíc oproti účelové funkci mohou podmínky obsahovat porovnávacíoperátory `<, >, <= a >=`.

`Bounds` — Omezení hodnot rozhodovacích proměnných. V této sekci jsou povoleny pouze porovnávací operátory uvedené výše.

`Generals` — Obsahuje seznam použitých rozhodovacích proměnných oddělených znaménkem mezery. Pokud je v souboru nalezena proměnná, která v této sekci není uvedena, program skončí s chybovou hláškou `"Unknown variable'<j>'!\n"`, kde `<j>` je neznámá proměnná, a návratovou hodnotou 10. Pokud sekce obsahuje nepoužitou rozhodovací proměnnou `<n>`, program vypíše pouze varování `"Warning: unused variable '<n>'!\n"`.

`End` — Uvozuje konec souboru, tzn. že se vyskytuje vždy jako poslední a sekce uvedené za ním jsou syntaktickou chybou.

Až na návěští `End` není pořadí jednotlivých sekcí fixní. Na výskyt neplatných operátorů, neznámých sekcí a jiných problémů program reaguje vypsáním chybového hlášení `"Syntax error!\n"` a skončí s návratovou hodnotou 11. Komentáře v souboru jsou uvozeny znakem "\". Ukázku vstupního souboru si můžete prohlédnout v konzolovém rozhraní 1 na straně 4.

Optimalizační algoritmus

Při analýze úlohy jistě narazíte na problémy degenerovaných úloh a jiné, které budeme pro jed-noduchost ignorovat. Algoritmus hledání optimálního řešení úlohy lineárního programování může tedy teoreticky skončit následovně.

1. **Nalezení konečného optimálního řešení** V takovém případě program vypíše optimální hodnoty rozhodovacích proměnných a skončís návratovou hodnotou `EXIT_SUCCESS`. Optimálních řešení může být více, s čímž validační skript počítá.
2. **Úloha je neomezená** Účelová funkce může nabývat libovolně velkých hodnot, aniž by porušila některou z omezují-cích podmínek, tj. optimum je v nekonečnu. Program na tuto skutečnost upozorní chybovým hlášením `"Objective function is unbounded.\n"` a skončí s návratovou hodnotou 20.
3. **Neexistence přípustného řešení** Soustava omezení nemá žádnou společnou přípustnou oblast, tj. neexistuje žádný bod, kterýby vyhovoval všem omezením současně – úloha je nesplnitelná. V takovém případě program vypíše chybovou hláškou `"No feasible solution exists.\n"` a vrátí hodnotu 21.

2 Analýza úlohy

2.1 Zadání

Zadání úlohy vyžaduje naprogramování konzolové přenositelné aplikace v jazyce **ANSI C**. Jejím obsahem bude implementace nástroje pro řešení úloh lineárního programovní.

2.2 Lineární úlohy

Lineární úloha je matematický problém, který lze formulovat pomocí lineární funkce nebo rovnice. V těchto úlohách jsou rozhodujícími faktory proměnné, které jsou v lineární závislosti na jiných proměnných. V běžné podobě se lineární úlohy vyskytují jako optimalizační problémy, kde je cílem maximalizovat nebo minimalizovat určitou funkci (např. zisk, náklady) za podmínek (omezeních), které jsou vyjádřeny jako lineární rovnice nebo nerovnice.

Formulace lineární úlohy

Obecná formulace lineární úlohy zahrnuje:

- **Cílová funkce**: Funkce, kterou chceme maximalizovat nebo minimalizovat. Obvykle se vyjadřuje jako lineární kombinace proměnných, např.:
$$Z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$$
kde $c_1, c_2, \ldots, c_n$ jsou koeficienty, a $x_1, x_2, \ldots, x_n$ jsou rozhodovací proměnné.
- **Omezení**: Podmínky, které musí být splněny, jsou vyjádřeny jako soustava lineárních rovnic nebo nerovnic. Například:
$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \geq b_2$$
kde a_{ij} jsou koeficienty a $b_1, b_2, \ldots, b_m$ jsou pravé strany rovnic nebo nerovnic.

– **Meze**: V některých úlohách je potřeba, aby proměnné byly nezáporné, tj. $x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0$.

Lineární úlohy se obvykle řeší pomocí **simplexové metody**, která je algoritmem pro hledání optimálního řešení v diskrétní množině. Existují také další metody, jako **grafické řešení** (pro úlohy se dvěma proměnnými) nebo použití specializovaných softwarových nástrojů, například **LP solvers** (linear programming solvers).

2.3 Simpexový alogritmus

Simplexový algoritmus je jedním z nejpopulárnějších a nejefektivnějších algoritmů pro řešení lineárních optimalizačních úloh. Tento algoritmus je iterativní metodou, která začíná v jednom z vrcholů konvexní množiny řešení a postupně se přesouvá k sousedním vrcholům. Proces pokračuje, dokud nenalezne optimální řešení nebo se nepotvrdí, že úloha nemá řešení. Simplexový algoritmus vždy dosáhne optimálního řešení v jednom z vrcholů, pokud existuje.

Princip fungování

Simplexový algoritmus hledá optimální řešení v diskrétní množině vrcholů polyhedronu definovaného omezeními lineární úlohy. Každý krok algoritmu je zaměřen na zlepšení hodnoty cílové funkce, dokud není dosaženo optimálního řešení.

Algoritmus pracuje s tzv. **simplexovou tabulkou**, která je sestavena z koeficientů cílové funkce, omezení a proměnných. Na každém kroku algoritmus provádí výpočty, které vedou k pohybu mezi vrcholy polyhedronu směrem k optimálnímu bodu.

Kroky simplexového algoritmu

– **Inicializace**: Nejprve se sestaví počáteční simplexová tabulka, která obsahuje informace o koeficientech lineární úlohy. Tato tabulka představuje počáteční řešení, které není nutně optimální.
– **Výběr vstupní proměnné**: Zvolí se proměnná, která nejvíce zlepšuje hodnotu cílové funkce. Tato proměnná je přidána do báze.

– **Výběr výstupní proměnné**: Určí se proměnná, která musí být odstraněna z báze, aby byla zachována konzistence s omezeními. Tento krok zaručuje, že zůstáváme v prostoru povolených řešení.
– **Aktualizace simplexové tabulky**: Po výběru proměnných se aktualizují hodnoty v simplexové tabulce podle pravidel algoritmu. Tato tabulka se používá pro výpočet nových hodnot proměnných v dalším kroku.
– **Opakování**: Tento proces se opakuje, dokud všechny koeficienty v řádku cílové funkce nebudou nenegativní, což indikuje dosažení optimálního řešení.

Výhody a nevýhody

Simplexový algoritmus je velmi efektivní v praxi a obvykle najde optimální řešení v relativně krátkém čase. I když teoreticky může mít exponenciální složitost, v reálných úlohách je výkon obvykle velmi dobrý. Nicméně, v extrémních případech může algoritmus selhat a zůstat u suboptimálních řešení.

Jedním z možných problémů je, že simplexový algoritmus může někdy projít cyklem, což znamená, že se vrátí zpět na stejný bod bez pokroku. Tento problém je možné vyřešit pomocí různých modifikací, jako je pravidlo Bland nebo použití *dual simplex method*. ss

3 Popis implementace

Projekt je implementován v jazyce *ANSI C* za použití standartních knihoven *<stdio.h>, <stdlib.h>, <ctype.h>, <limits.h>* a *<string.h>*. Projekt je rozdělen do jedenácti hlavičkových a dvanácti *.c* souborů. Každý *.c* soubor má s výjimkou *Main.c* souboru vlastní stejnojmený .h soubor.

V souborech *Description, Matrix, Bounds, Logger a Result* se nachází implementace stejnojmenných struktur a funkcí k jejich alokaci, inicializaci, měnění obsahu jejich hodnot, deinicializaci a dealokaci.

Soubory *FileReader* obsahují implementaci načítání vstupních údajů od uživatele, ať už z příkazového řádku, tak ze souboru. Výpis informací do souborů a do příkazového řádku obsahují soubory *ResultWriter*.

Global slouží k deklaraci a inicializaci globálních proměnných dostupných v souborech *Main* a *Exit*. Jinde v kódu se tyto struktury nepoužívají a neměla by do nich být tato třída includována. Jako globální proměnné jsou vytvořeny potřebné struktury, které je potřeba dynamicky alokoat a dealokovat. O to se stará třída *Exit*. Je tak možné na jednom místě dealokovat dynamicky alokovanou paměť a předejít tak ztrátě paměti.

Implementace výpočtové části se nachází v třídě *MathCalculations*. Ta obsahuje metody pro práci s maticí, včetně simplexového algoritmu. Třída *Tool* obsahuje doplňkové funkce k matematickým a jiným procesům, jejich použití se předpokládá na více místech v kódu.

3.1 Struktury

Aplikace obsahuje pět struktur, *popis úlohy, logger, matici, meze a výsledky*.

3.1.1 Popis úlohy

Popis úlohy je reprezentován strukturou description.

```
struct description{
    char* objective;
    char** constraints;
    size_t numConstraints;
    char** bounds;
    size_t numBounds;
    char** generals;
    size_t numGenerals;
    int maxOrMin;
};
```

Struktura uchovává informace o všech informací dostupných ze zadání úlohy. Informace uchovává přesně tak, jak je byly zapsány ve vstupním souboru. Obsahuje tedy informace o účelové funkci, maximalizaci či minimalizaci, mezích a omezujících rovnicích. Informace z této struktury jsou následně používány a převedeny do maticové podoby.

3.1.2 Matice

Matice je popsána strukturou matrix. Matici využívá algoritmus k řešení lineárních optimalizačních úloh, kde každá buňka matici představuje koeficienty nebo hodnoty související s rozhodovacím procesem.

```
struct matrix {
    float** board;
    size_t numRows;
    size_t numCols;
};
```

Matice má stanovený počet řádek a sloupců. Hodnoty uvnitř jsou datového typu float.

3.1.3 Meze

Součástí úlohy jsou i omezení pro jednotlivá neznámá, který bude program hledat. Meze jsou definovány vlastní strukturou bounds.

```
struct bounds {
    float* lowerBounds;
```

```
    int* lowerEquals;
    float* upperBounds;
    int* upperEquals;
    int numBounds;
};
```

Pokud není horní mez zadáním stanovena je ji přidělena hodnota INT_MAX, pokud není stanovena dolní mez je ji přidělena hodnota INT_MIN. Vedle toho struktura rozlišuje i otevřenost a uzavřenost stran intervalu. Pokud je *lowerEquals* rovno 0, je dolní mez intervalu otvřená, pokud je rovno 1 jedná se o uzavřenou dolní mez intervalu.

3.1.4 Výsledky

Výsledek simplexového algoritmus je uložen do struktury result.

```
struct result {
    char** variables;
    size_t numVariables;
    float* results;
};
```

Struktura obsahuje počet neznámých, jejich názvy a hodnoty. Je používána pro uložení a výpis výsledků.

3.1.5 Logger

Struktura logger v sobě uchovává název výstupního souboru.

```
 struct logger {
    char* outputFileName;
};
```

Pokud je název souboru zadán, bude do něj vypsán výstup. Pokud je název souboru NULL, je výstup vypsán na obrazovku. Logger má aktuálně omezené využití, ale v případě rozšíření, by mohl vypisovat loggovací zprávy o aktuálním stavu programu.

3.2 Důležité funkcionality

Aplikace sestává z desítek různých funkcí, ty nejdůležitější vypadají takto.

3.2.1 Main funkce

Vstupní bod aplikace, skládající se z načtení vstupních dat, jak z příkazové řádky tak ze vstupního souboru, potřebných kontrol dat, vypočítání vhodného řešení a následným vypsáním výsledků.

```
1  int main(int argc, char const *argv[])
2  {
3      if(argc < 2) {
4          printf("Input file not found!\n");
5          return 1;
6      }
7      readFromFile(&description1, getInputFileName(argv, argc));
8      saveOutputFile(argc, argv, &logger1);
9      assignmentCheck(description1);
10     prepareForCalculations(description1, &matrix1, &bounds1,
           &result1);
11     simplexMethod(description1, matrix1, &result1);
12     printOutResults(result1, logger1, description1, matrix1,
           bounds1);
13     freeStructures();
14     return 0;
15 }
```

3.2.2 Načítání vstupních dat

Načítání dat ze vstupního souboru se děje v jednom while cyklu, prvně je zjištěno příslušné návěstí a dle toho uložena hodnota. Metoda obsahuje i příslušné kontroly formátu vstupního souboru.

```
1
2  int readFromFile(struct description** desc, const char*
       fileName) {
3      FILE* file; char line[LINE_LEN];
4      int inConstraints, inBounds, inGenerals, inType;
5      if (!fileName || !desc) {
```

```
6           ext(1, "Input file not found!\n");
7       }
8       file = fopen(fileName, "r");
9       if (!file) {
10          ext(1, "Input file not found!\n");
11      }
12      inBounds = 0, inConstraints = 0, inGenerals = 0, inType =
            0;
13      descriptionAllocate(desc, NULL, NULL, 0, NULL, 0, NULL, 0,
            0);
14
15      while (fgets(line, LINE_LEN, file)) {
16          trimLine(line);
17          removeComment(line);
18          if (line[0] == '\0' || line[0] == '\\') continue;
19          if (strncmp(line, "Maximize", 8) == 0 || strncmp(line,
                "Minimize", 8) == 0) {
20              handleMaximizeMinimize(&inConstraints, &inBounds,
                    &inGenerals, &inType, desc, line);
21          } else if (strncmp(line, "Subject To", 10) == 0) {
22              handleSubjectTo(&inConstraints, &inBounds,
                    &inGenerals, &inType);
23          } else if (strncmp(line, "Bounds", 6) == 0) {
24              handleBounds(&inConstraints, &inBounds,
                    &inGenerals, &inType);
25          } else if (strncmp(line, "Generals", 8) == 0) {
26              handleGenerals(&inConstraints, &inBounds,
                    &inGenerals, &inType);
27          } else if (strncmp(line, "End", 3) == 0) {
28              break;
29          } else if (inConstraints) {
30              if(processConstraints(desc, line) == EXIT_FAILURE)
                    {
31                  fclose(file);
32                  ext(11, "Syntax error!\n");
33              }
34          } else if (inBounds) {
35              processBounds(desc, line);
36          } else if (inGenerals) {
37              processGenerals(desc, line);
38          } else if (inType) {
39              processObjective(desc, line);
40          } else {
```

```
41              fclose(file);
42              ext(11, "Syntax error!\n");
43          }
44      }
45      fclose(file);
46      return EXIT_SUCCESS;
47  }
```

3.2.3 Převod řetězce na čísla

Jednou z nejdůležitějších úloh programu je správně rozpoznat a vyhodnotit zadaný řetězec obsahující matematický výraz, rovnici či funkci. Jednotlivé části výrazu jsou nejprve rozpoznány a uloženy do zásobníku čísel, připadně do zásobniku dalších prvků (závorek, operátorů atp.). Poté je do výrazu dosazena jednička pro proměnnou, kterou hledáme, pro ostatní proměnné je dosazena nula. Výsledek této operace je pak zapsán na odpovídající pozici matice. Po sestrojení matice se pracuje již jen s ní.

```
1   float getGeneralsCoeficient(const char* expression, const
        char* token) {
2       if (!expression || !token) {
3           return INT_MIN;
4       }
5       char* newExpression = (char*)malloc(strlen(expression) * 2
            + 1);
6       if (!newExpression) {
7           return INT_MIN;
8       }
9       newExpression[0] = '\0';
10      const char* p = expression;
11      while (*p) {
12          if (isalpha(*p) || *p == '_') {
13              char varName[128];
14              size_t varLen = 0;
15              while (isalnum(*p) || *p == '_') {
16                  varName[varLen++] = *p;
17                  p++;
18              }
19              varName[varLen] = '\0';
20              if (strcmp(varName, token) == 0) {
21                  strcat(newExpression, "1");
```

```
22              } else {
23                  strcat(newExpression, "0");
24              }
25          } else {
26              strncat(newExpression, p, 1);
27              p++;
28          }
29      }
30
31      removeRightSide(newExpression);
32      float result = (float) evaluateExpression(newExpression);
33      free(newExpression);
34      return result;
35  }
```

3.2.4 Simplexový algoritmus

Simplexový algoritmus je nejdůležitější metodou programu. Z již sestrojené simplexové tabulky v podobě matice, která mu je předána jako parametr, získá potřebné hodnoty proměnných.

Výpočet se odehrává ve while cyklu, dokud je možné hledat stále lepší řešení. Na základě pivotního řádku a sloupce se vybere pivot, kterým jsou poděleny zbylé řádky.

Po nalezení ideálního řešení jsou pomocí metody *getResult* přečteny a zapsány výsedky do struktury *result*.

```
1   void simplexMethod(struct description* description, struct
        matrix* matrix, struct result** result) {
2       int columnIndex, rowIndex, index;
3       float divider, multiplier;
4       float coeficients[matrix->numCols - 1];
5       getAllCoeficients(matrix, coeficients);
6       while(doNextIteration(matrix) == 1) {
7           columnIndex = findPivotFromObject(matrix);
8           rowIndex = findPivotRowConstraint(matrix, columnIndex);
9           divider = matrix->board[rowIndex][columnIndex];
10          divideTheRow(matrix, rowIndex, divider);
11          for (index = 0; index < matrix->numRows; index++) {
12              if(index == rowIndex) {
13                  continue;
14              }
```

```
15              if(index == matrix->numRows - 1) {
16                  objectiveFunctionCalculation(description,
                        matrix, coeficients);
17              }
18              multiplier=
19              getMultiplier(matrix->board[rowIndex]
                    [columnIndex],
                    matrix->board[index][columnIndex]);
20              substractRow(matrix, multiplier, rowIndex, index);
21          }
22      }
23      getResult(matrix, result);
24  }
```

4 Uživatelská příručka

4.1 Překlad

Aplikace je pouze konzolová a je ji nutné nejprve přeložit, o což se stará soubor **Makefile**. Překlad se spouští příkazem `make -f Makefile` na Linuxu, poté již lze aplikaci spustit. Pro spuštění na Windows je potřeba použít příkaz `make -f Makefile.win`

4.2 Spuštění

Aplikaci lze spusit příkazem `.\lp.exe`. Aplikace pro svůj běh potřebuje být spuštěna alespoň se vstupním souborem pommocí parametru *-input*. Navíc může být spuštěna i s parametry *-o* či *–output* pro zápis výsledků do výstupního souboru.

Pro windows je spuštění stejné jako pro linux.

4.3 Výstup

Výstupem programu jsou textové řeztěce obsahující informace v předepsaném formátu ohledně řešení zadané lineární úlohy. Výstup může být zobrazen buď do konzole či textového souboru.

```
Warning: unused variable 'another_unused_variable'!
x = 0.0000
y = 1.0000
z = 0.0000
another_unused_variable = 0.0000
```

Obrázek 4.1: Grafický výstup programu

5 Závěr

I přes všechny slepé cesty, a že jsem se po nich s nadšením vydal, se mi podařilo naprogramovat aplikaci, která splňuje požadavky zadání.

Projekt byl otestován notebooku HP ProBook s 8GB RAM a 1.6 GHz, program se ukázal jako velmi rychlý, kdy i pro větší počet proměnných dokončil výpočty čase okolo jedné sekundy.

Čas [s]	**Proměnné [n]**	**Rovnice [n]**
0.003	2	2
0.007	2	3
0.840	3	3
0.044	4	4
0.090	4	6

Tabulka 5.1: Délka běhu programu

Čas běhu programu je více ovlivněn složitostí zadaných matematických výrazů než počtem proměnných či rovnic. V případě č. 3 bylo použito rovnic s vícero závorkami. Vyhodnocení těchto výrazů je slabým místem programu a zároveň kandidátem na další vylepšení.

Samotný simplexový algoritmus je velmi rychlý.

6 Zdroje

Internetové zdroje

Ryjáček, J. (2025). TGD2: Teoretické základy operačního výzkumu. Západočeská univerzita. Dostupné z: http://najada.fav.zcu.cz/~ryjacek/students/ps/TGD2.pdf,
Wikipedia contributors. (2025). Simplex algorithm. Wikipedia. Dostupné z: https://en.wikipedia.org/wiki/Simplex_algorithm,
Kckurzy.cz. (2025). Kckurzy.cz (simplexová tabulka, operační výzkum) [Video]. YouTube. Dostupné z: https://www.youtube.com/watch?v=wubr8nbi8AI.

7 Seznam vyobrazení

Tabulky

Obrázky