FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Semestrální práce

Nástroj pro řešení úloh lineárního programování

Michal Malík

PLZEŇ

2025

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Semestrální práce

Nástroj pro řešení úloh lineárního programování

Michal Malík

PLZEŇ 2025

Citace v seznamu literatury:
MALÍK, Michal. *Nástroj pro řešení úloh lineárního programování*. Plzeň, 2025. Semestrální práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

Obsah

Zadání 1

Jako téma své semestrální práce jsem si zvolil 3. zadání, navržení nástroje pro řešení úloh lineárního programování, od Ing. Františka Pártla.

1.1 Lineární programování

Lineární programování je subdisciplína matematického programování, která slouží k určení minima nebo maxima lineární funkce určitých proměnných. K tomu využívá soustavu rovnic, představující různá omezení daných proměnných.

K lepšímu vysvětlení si to pojďme ukázat na příkladu. Představme si, že máme výrobní linku. Na té se vyrábí dva produkty, X a Y. Produkt X se vyrábí jednu hodinu, zatímco produkt Y dvě. Výrobní linka jede standardně osm hodin denně. Pro oba výrobky probíhá i kontrola kvality. Celkově můžeme kvalitu ověřovat 6 hodin, kde spotřebujeme 2 hodiny na kontrole výrobku X a jednu při kontrole Y. Oba výrobky produkují dvě jednotky. Teď si pojďme ukázat, jak vytěžit maximum.

Účelová funkce bude vypadat následovně

$$z = 2x + 2y$$

kde proměnné x a y reprezentují výrobky X a Y. Teď ještě musíme vypsat potřebné rovnice k dosažení výsledku

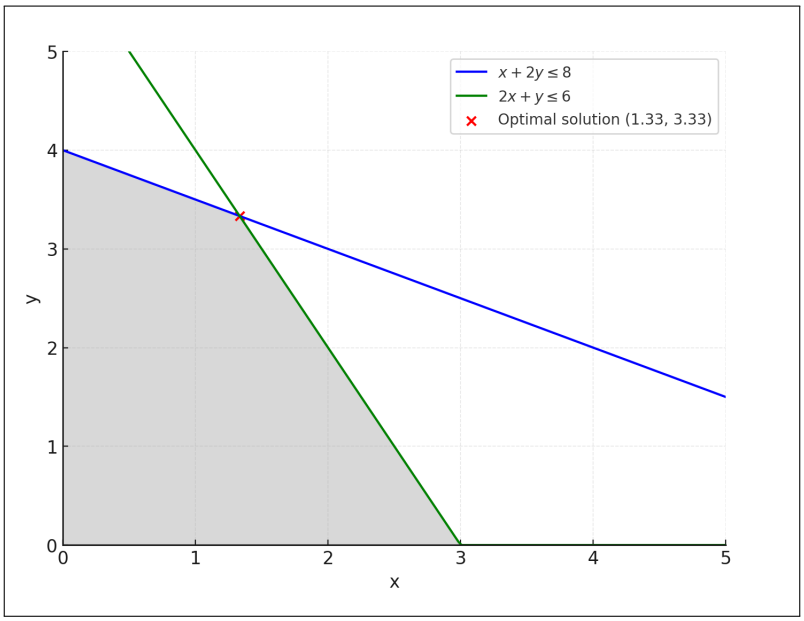
$$x + 2y <= 8$$

$$2x + y <= 6$$

a na závěr musíme nadefinovat omezení pro jednotlivé proměnné. Ty nechceme mít záporné.

$$x >= 0$$

$$y >= 0$$



Obrázek 1.1: Grafické znázornění úlohy lineárního programování

Když si všechny tyto výrazy spojíme dohromady, dostaneme graf, který může vypadat nějak takto.

Ve výsledku vidíme, že naše řešení nemusí být vždy celočíselné. Ze zadání semestrální práce víme, že se tímto nemusíme zabývat, jelikož celočíselné lineární programování je NP-těžké a s tím se v tuto chvíli nechceme zdržovat.

1.2 Zpracování vstupního souboru

Další důležitou fází této práce je zpracování vstupního souboru do potřebného finálního produktu, který potřebuje sestrojit k výpočtu úlohy lineárního programování. Vstup se skládá z pěti sekcí. Maximize/Minimize, Bounds, Subject to, Generals a End. Musíme zařídit a ohlídat, aby náš program prošel skrze všechny tyto sekce a rozpoznal správnost dat, které vstupní soubor ukládá. Vstupní soubor může obsahovat i komentáře začínající znakem "\".

1.2.1 Maximize/Minimize

Z této sekce získáme, jestli se jedná o maximalizaci, či minimalizaci a zároveň nám předá účelovou funkci, jejíž koeficienty poslouží k výpočtu.

1.2.2 Bounds

Sekce 'Bounds' v sobě ukládá informace o omezenosti jednotlivých proměnných. Tím myslíme spodní a horní hranice konkrétní proměnné.

1.2.3 Subject to

Zde získáme koeficienty soustavy rovnic všech omezujících funkcí. Musíme zde dávat pozor na správné užívání logických operátorů (<=, >=, =) a ukládat si i pravé strany všech těchto rovnic.

1.2.4 Generals

V části 'Generals' dostaneme výpis všech proměnných. Náš program si bude muset poradit s tím, jestli všechny proměnné, které zde najdeme, jsou využívány, či jestli v seznamu žádné proměnné nechybí.

1.2.5 End

Jak už název napovídá, tato sekce hlásí konec vstupního souboru. Na pořadí všech sekcí, kromě sekce 'End' nezáleží, ale tato sekce se vždy musí objevovat na konci souboru s vstupními daty.

Analýza úlohy 2

2.1 Jednoduché problémy

Nejdříve jsem začal s tím nejlehčím, rozpracováním běhu programu. Jak bude muset kód za sebou zpracovávat jednoduché úkoly, a kde všude se může uživatel při spuštění programu splést.

Jako první věc, na kterou jsem se začal soustředit, je vstup od uživatele. Uživatel má možnost si vstup buď nechat vypsat do konzole, či k přiloženému souboru pro zapsání výstupních dat, pomocí použití příkazů '-o' nebo '–output'. Uživatel pomocí spouštícího příkazu i zadává název souboru se vstupními daty. Posloupnost příkazů je libovolná, ale vždy platí, že po příkazu pro zápis výsledku do souboru je nutné vždy hned za ním uvést název daného souboru.

Když jsme u těch souborů, vzal jsem jako nutnost i ověřování existence vstupního souboru a i správnosti cesty k výstupnímu souboru. Výstupní soubor nemusí nutně existovat, jelikož knihovna pro otevírání souborů v jazyce ANSCII C je schopna vytvořit soubor pro zápis, pokud jej nenajde. Jediný způsob, kterým může zápis selhat bývá neexistence kořenových složek daného souboru.

2.2 Zpracování vstupu

Pro správnou analýzu této pod-úlohy bylo potřeba si načrtnout jak vstupní data mohou vypadat.

Zdrojový kód 2.1: Ukázka vstupních dat v textovém souboru

```
\ uloha z uvodu zadani
Subject To \ poradi neni fixni
vyroba: x + 2y <= 8
kontrola: 2 *x + 1 * y <= 6
Maximize
2x+ 2 * y \ ucelova funkce
Generals
y x z
Bounds
0 <= x
0 <= y
End
```

Jak jsme si říkali v první kapitole, důležité je zde projít skrze všechny sekce. Pokud se tak nestane, program vyhodí chybovou hlášku 'Syntax Error!' Syntaxové chyby se zde můžou vyskytnou skoro kdykoliv, proto jsem zvolil takovou cestu, která podrobně vždy připraví každý řádek předtím, než s ním začne pracovat. Přípravou zamýšlíme například přeskočení komentářů, zbytečných mezer či tabulátorů na začátku textu. Také si rovnou ověřím jestli dává řádka smysl z matematického hlediska a nebo jestli neobsahuje nějaké nesmyslné znaky.

2.2.1 Zpracování infixového výrazu

Další důležitou částí při zpracování je získat koeficienty účelové funkce a ze soustavy funkcí omezení. Ze soustavy rovnic dále také musíme získat pravé strany a operátory, které ovlivňují množinu hodnot výsledku.

2.2.1.1 Algortimus Shunting Yard

Algoritmus 'Shunting Yard' vytvořil Edsger W. Dijkstra a slouží k převodu infixového zápisu matematické rovnice na zápis postfixový. Například $2 * x + 5y$ převedu na zápis $2x * 5y * +$. K jeho vytvoření je potřeba fronta, do které se bude zapisovat postfixový zápis a zásobník pro matematické operátory.

Infixový zápis procházím vždy po znaku. Pokud narazím na číslo nebo název proměnné, tak daný prvek vždy ihned vložím do fronty. Mezitím na zásobník vkládáme matematické operátory, ty se liší svoji vahou. Operátory +- mají váhu 1, * a / mají váhu 2 a největší váhu mají závorky. Ze zásobníku vyndavám operátory a v kládám je do fronty pouze ve 2 případech. Za 1. pokud je váha nově přidaného

operátoru menší než na operátoru na vrcholu zásobníku. A za 2. pokud přijde otevření závorky, a poté na zásobník přijdou další operátory, tak při uzavření závorky se všechny operátory mezi otevřením a zavřením přesunou do fronty. Samozřejmě se zásobník vyprázdní i na konci, pokud v něm něco zbude.

2.2.1.2 Vyjmutí koeficientů

K závěru využívám objekt slovníku, kde ke každé proměnné mám uloženou její hodnotu koeficientu. Samozřejmě je zde potřeba nezapomínat i samostatná čísla. Když procházíme postfixový zápis, vždy když narazíme na proměnnou či číslo, vytvoříme novou instanci ve slovníku. Jindy, když narazíme na operátor, tak vezmeme dva prvky ze slovníku, které jsou z levé strany nejblíže k operátoru. Zároveň i smažeme jejich existenci z fronty. Poté vytvoříme novou sdruženou instanci způsobem, který je daný operátorem z postfixového zápisu.

Na závěr by nám měl zbýt jen jeden objekt slovníku, který v sobě má vždy pojmenovanou proměnnou a její koeficient.

2.2.1.3 Pravá strana

Pravé strany získáme při rozdělení textu. Jediné platné operátory, podle kterých můžeme rozdělit rovnici jsou <=, >= a =. Pravá strana je taková hodnota, která nám po rozdělení zbyde v pravé části rovnice.

2.2.2 Spodní a horní omezení

V sekci 'Bounds' se můžeme vyskytnout s různým způsobem zápisu, proto je potřeba si jej ohlídat. Pomocí operátorů <= a >= zjistíme, jaké ohraničení pro danou funkci existuje. Pokud není z dat vstupního souboru řečeno, proměnná má obor hodnot neomezený.

2.2.3 Proměnné

Po zamyšlení jsem zjistil, že není spolehlivé ze sekce 'Generals' ukládat hlavní proměnné. Ty vybírám spíše z účelové funkce, protože zde se musí všechny proměnné vyskytovat. Tyto přídavné proměnné slouží spíše na okrasu a na kontrolu správnosti seznamu. Pokud ze vstupu dostaneme proměnnou navíc, programu to nevadí. Vypíše pouze varování. Ale pokud v seznamu z 'Generals' chybí proměnná, která se vyskytuje v účelové funkci, tak aplikace vyhodí chybovou hlášku 'Warning: unused variable '<n>'!', kde <n> reprezentuje název proměnné.

2.3 Vyřešení problému Lineárního programování

Na závěr jsem musel zjistit jak vlastně vypočítat zadaný problém. Po chvilce hledání jsem přišel na algoritmus, který pracuje s tzv. **Simplexovo tabulkou**.

2.3.1 Simplexův algoritmus

Simplexův algoritmus je algoritmus, který poprvé popsán George Dantzigem. Slouží k efektivnímu prohledávání základních řešení úloh lineárního programování. Pomocí hodnot které dostanu z účelové funkci a soustavy rovnic funkcí omezení vytvořím stejnojmennou tabulku. Z příkladu, který jsme si popisovali v první kapitole, by jsme si mohli představit něco takového:

	C	2	2	0	0	
B	C_B	x	y	S_1	S_2	Rh
S_1	0	1	2	1	0	8
S_2	0	2	1	0	1	6
	Z	0	0	0	0	
	C - Z	2	2	0	0	

Tabulka 2.1: Simplexova tabulka

Jak vidíte, je zde spousta řádků a sloupců. První řádek C obsahuje koeficienty účelové funkce. Proměnné x a y už známe. Ale co S_1, S_2 ? To jsou takzvané '**Slack**' **proměnné**. Pomáhají nám s výpočtem. Sloupce B a C_B jsou **bázové sloupce**. Ty využíváme k určení pivotního prvku pro úpravu tabulky a zároveň mají v sobě uložené finální řešení. Pokud na konci algoritmu zůstane Slack proměnná v matici, tak alespoň jedna hodnota proměnné je nulová.

K určení pivotního prvku ještě používáme řádky Z a $C - Z$. $C - Z$ slouží zároveň i k určení konce algoritmu, pokud všechny jeho hodnoty jsou nulové nebo záporné, algoritmus skončil.

2.3.1.1 Pivotní prvek

Pivotní prvek je takový prvek, podle něhož se iteruje zbytek tabulky. Iterujeme takovým způsobem, že potřebujeme, aby sloupec, který obsahuje pivotní prvek, měl na řádku, kromě pivotního prvku, samé 0.

Vyběr pivota provádíme tak, že nejdříve podle řádku $C - Z$ vybereme ten sloupec, kde je hodnota řádku největší nenulová kladná. Poté vybíráme ten řádek, jehož hodnota je kladné nenulové číslo a zároveň dělitelem pravé strany a rozdíl

mezi pravou stranou a prvkem daného řádku z pivotního sloupce je nejmenší. Pokud ani jeden z možných prvků řádků v pivotním sloupci není kladné nenulové číslo, znamená to, že řešení je nekonečné.

2.3.2 Big M Tableau

Tabulka, o které jsme doteď mluvili, slouží pouze při operátorech <=. Pokud chceme řešit rovnice typu >= a =, musíme ji rozšířit na **'Big M Tableau'**. Tato rozšířená tabulka využívá kromě Slack proměnných také **'Artificial' proměnné**. Ty jsou na řádku C reprezentovány zápornou konstantou M. M znázorňuje hodně velké číslo, něco na způsob ∞.

Pokud rovnice obsahuje operátor <= nebo =, tak $-M$ nahrazuje hodnoty Slack proměnných v bázovém sloupci. Například kdybychom v našem příkladě přeměnili jednu z rovnic na

$$x + y >= 6$$

tak tabulka změní svůj vzhled takto: Pokud Artificial proměnné neopustí na konci

	C	2	2	0	0	-M	-M	
B	C_B	x	y	S_1	S_2	A_1	A_2	Rh
S_1	0	1	2	1	0	0	0	8
A_2	-M	2	1	0	-1	0	1	6
	Z	-2M	-M	0	M	0	-M	
	C - Z	2 + 2M	2 + M	0	-M	0	0	

Tabulka 2.2: Big M Tableau

výpočtu bázový sloupec, znamená to, že příklad nemá řešení.

2.3.3 Z Maximize na Minimize

Pokud se stane, že máme hledat minimalizaci účelové funkce, způsob řešení je skoro podobný. Jediné, co je potřeba změnit jsou znaménka u koeficientů účelové funkce. Jestli se vyskytuje záporná hodnota na pravé straně v soustavě rovnic omezujících funkcí, je třeba v celé rovnici také zaměnit znaménko a v případě nerovnosti, otočit operátor.

Implementace 3

V této kapitole se podíváme na kódovou stránku programu, jeho implementaci a pomocné knihovny, které jsme při řešení problému vytvořili.

3.1 Hlavní část

Hlavní část programu, která se nachází v souboru **main.c** by měla být lehce čitelná a sama osobě toho moc nedělat. Jediná samostatná práce, co **main.c** dělá, je ohlídání si správnosti použitých argumentů a zjišťovaní chybových kódů, pokud se někde vyskytnou. To můžeme, například, vidět po zavolání funkce *lpp_solve*, která má na starosti vyřešit úlohu lineárního programování přes Simplexovu tabulku. Funkce vrací datový typ integer, který se porovná s chybovým kódem. Třeba když úloha nemá řešení či je neomezená.

Zdrojový kód 3.1: Ukázka chybovníku pro funkci *lpp_solve*

```
if(result) {
    switch (result) {
        case 20:
            printf("Objective function is unbounded.\n");
            lpp_dealloc(&LPPSolver);
            return 20;
            break;

        case 21:
            printf("No feasible solution exists.\n" );
            lpp_dealloc(&LPPSolver);
            return 21;
            break;

        case 3:
            printf("Allocation failed.\n");
            lpp_dealloc(&LPPSolver);
            return 3;
            break;

        default:
            break;
    }
}
```

Pokud program skončí v pořadku, končí návratovou hodnotou 0 (**EXIT_SUCCESS**).

3.2 LPFile

Přesuňme se teď na mnou vytvořené knihovny. **lpfile.h** je základním stavebním kamenem našeho souboru. Nachází se zde jak část s načítáním vstupních dat, která k pomoci využívá další knihovnu **shunt.c** a druhou část, řešící. Řešící část využívá zase knihovny **simplex.h**. Ale o obou těchto knihovnách si povíme až déle.

3.2.1 Struktura úlohy lineárního programování

Zdrojový kód 3.2: Struktura *LPProblem*

```
1  struct LPProblem {
2      int num_vars;                                  /**< Počet
       proměnných. */
3      int num_constraints;                           /**< Počet
       omezení. */
4
5      double objective[MAX_VARS];                    /**<
       Koeficienty účelové funkce. */
6      double constraints[MAX_ROWS][MAX_VARS];        /**<
       Koeficienty omezovacích funkcí. */
7      double rhs[MAX_VARS];                          /**< Pravé
       strany omezovacích funkcí. */
8
9      double lower_bounds[MAX_VARS];                 /**< Spodní
       hranice hodnot. */
10     double upper_bounds[MAX_VARS];                 /**< Horní
       hranice hodnot. */
11
12     char vars[MAX_VARS][LINE_LENGTH];              /**< Jména
       použitých proměnných. */
13     char b_column_vars[MAX_ROWS][LINE_LENGTH];     /**< Jména
       proměnných v bázovém sloupci. */
14     char operators[MAX_VARS][LINE_LENGTH];         /**< Použíte
       operátory omezovacích funkcí. */
15 };
```

Tato struktura v sobě ukládá všechny důležité informace, které potřebujeme k výpočtu úlohy.

Je doprovázena funkcemi, které slouží pro její alokaci a inicializaci, ale také pro její dealokaci a uvolnění paměti. Má v sobě uložené všechny koeficienty, počet omezení i použitých proměnných a ukládá v sobě jedinou část simplexové tabulky, a to bázový sloupec.

3.2.2 lpp_load

První největší a nejdůležitější funkcí je *lpp_load*. Ta slouží k rozkouskování všech sekcí vstupního souboru, ověření správnosti syntaxe a převedení všech dat do struktury *LPProblem*. Procházím zde každý řádek po jednom, vždy ho připravím pomocí pomocné funkce *lpp_prepare_str*. Poté porovnám řádek pomocí *strcmp* s názvy sektorů. Pokud řádek je nějaký ze sektorů, ukládám si jeho označení, abych věděl v jakém sektoru se teď nacházím. Pokud je titulek Maximize nebo Minimize, musím si pamatovat, který z nich to je, abych kdyžtak mohl provést následné úpravy

Pokud nenarazím na žádný z titulů, přesunu se na switch, který odkazuje na jednotlivé sektory, a co v nich je potřeba udělat. Pokud se dostanu buď do maximalizujícího sektoru či sektoru s omezujícími funkcemi, zavolám si pomocnou funkci *lpp_check_line*, která zjistí, jestli je funkce zapsaná správně z matematického hlediska.

Na závěr porovnám proměnné ze sekce 'Generals' a s proměnnými, které jsem našel v účelové funkci a zjistím jestli se někde nevyskytuje chyba. Poté co je všechno hotovo, mi nezbývá nic jiného než na alokovat a inicializovat novou strukturu úlohy lineárního programování.

3.2.3 lpp_solve

Druhá velká funkce je řešitel naší úlohy, ten pomocí knihovny **simplex.h** vytvoří všechny potřebné řádky a provádí nad simplexovou tabulkou úpravy pomocí pivotního prvku. Výsledky přepisuje v samotné struktuře *LPProblem*, konkrétně v atributu pole pravých stran *rhs*. Při povídání o hlavní části programu, jsme si už povídali co funkce vrací a jakých chybných hodnot může nabývat.

Pokud chceme zjistit výsledek můžeme zvolit dvě různé funkce, *lpp_write* a *lpp_print*. Záleží jestli chceme výsledek vypsat do terminálu nebo do vstupního souboru. Ke správnému vypsání slouží jak sloupec pravých stran, tak i bázový sloupec, protože jestli se nějaká ze základních proměnných v bázovém sloupci nevyskytuje, znamená to, že její hodnota je nulová.

3.3 Simplex

Pomocná knihovna **simplex.h** je naprosto jednoduchá na pochopení, má 3 základní úlohy. Připravit veškeré sloupce a řádky, pomoct najít pivotní prvek v simplexově tabulce a ohlídat, kdy je algoritmus dokončený.

Zdrojový kód 3.3: Kontrola nalezení optimálního řešení

```
1  int simplex_check_optimal_solution(double c_z_row[], const
       int n) {
2      int i;
3      if (c_z_row == NULL || n <= 0) {
4          return 0;
5      }
6      for (i = 0; i < n; i++) {
7          if (c_z_row[i] > EPSILON) {
8              return 1;
9          }
10     }
11     return 0;
12 }
```

Všechny řádky a sloupce připravují hlavně pomocí dat, uložených ve struktuře lineárního programování. Abych přesně věděl, kolik místa potřebuju využívám uložených hodnot *num_vars* a *num_constraints*.

3.4 Shunt

Poslední knihovna **shunt.h** je rozdělená na dvě části. První část je zaměřená na převod infixového zápisu na zápis postfixový. Druhá se zaměřuje na vytažení koeficientů z postfixové fronty. V této knihovně se vyskytuje nebezpečná manipulace s proměnnými objektů zásobníku a fronty, které jsem si vědom. Bohužel jsem to během vytváření práce nechal být a k následné opravě bezpečnějším způsobem jsem se nedostal.

Zdrojový kód 3.4: Globální proměnné zásobníku a fronty

```
1 /* Zásobník pro operátory */
2 char stack[MAX_STACK_SIZE];
3 int stack_top = -1;
4
5 /* Fronta pro výstupní postfix výraz */
6 char queue[MAX_QUEUE_SIZE][50];
7 int queue_rear = -1;
```

3.4.1 Infix na postfix

Infixový zápis převezmeme jako parametr do metody *shunt_get_postfix* společně s prázdným ukazovatelem na řetězec, do kterého uložíme postfix. Každý znak projede skrz kontrolu, abychom zjistili o jaký znak se jedná.

Pokud se jedná o číslo, či písmeno z abecedy, ty se spojují do proměnné *var_name*, pokaždé záleží na pořadí, kterým číslice a písmena chodí. Pokud do prázdné proměnné přijde jako první číslo, předpokládá se, že celá tato proměnná bude nějaký větší číselný celek, na druhou stranu, když začneme u písmene, budeme zacházet s *var_name* jako jménem konkrétní proměnné.

Zároveň nám mohou dorazit různé operátory, či závorky. Při analýze jsme si říkali, že každý operátor má svoji váhu, tu získáváme pomocí *shunt_precedence*, která vezme přijatý operátor a vrátí celočíselnou hodnotu odpovídající jeho váze. Důležité je dávat pozor i na implicitní násobení, které může vzniknout infixovým zápisem.

Zdrojový kód 3.5: Ukázka ověření implicitního násobení

```
/* Implicitní násobení (např. '2x') */
    if (strcmp(last_char, var_name) == 0 && infix[i - 1] == '
    ␣' && isdigit(ch) && strcmp(last_char, "") != 0) {
        shunt_push('*');
        shunt_enqueue(var_name);
        strncpy(last_char, var_name, var_index);
        var_index = 0;
}
```

Ke zpracování nám slouží už výše zmíněné zásobník a fronta, na které přidáváme hodnoty pomocí vnořených metod *shunt_push* a *shunt_enqueue*. Jak jsem zmínil, oba tyto objekty jsou globální a proto je důležité řešit vždy pouze jeden postfix v daný moment. Až s ním skončíme, můžeme řešit další infixové rovnice.

3.4.2 Vytažení koeficientů

Nejkomplexnější část celého programu se vyskytuje zde. V programovacím jazyce, jako je třeba Python, bychom mohli použít jednoduchou množinu se slovníkem k vytvoření výrazu celé rovnice, která od sebe oddělí název proměnné a její hodnotu. Bohužel jazyk C nemá ze základu tyto instance nadefinované, proto jsem musel experimentovat.

Vytvořil jsem si dvě struktury, *Term* a *Expression*. *Term* je pouhý jeden prvek, kde je uložená hodnota jeho koeficientu a název proměnné, popřípadě hodnota **NULL**, jestli se jedná o samostatné číslo.

Zdrojový kód 3.6: Ukázka struktur *Term* a *Expression*

```
typedef struct {
    char *variable;          /**< Název proměnné. (V případě,
    že je prvek jen samostatné číslo -> NULL) */
    double coefficient;      /**< Číselná hodnota koeficientu.
     */
} Term;

typedef struct {
    Term *terms;         /**< Pole prvků. */
    int size;            /**< Počet prvků v poli. */
    int capacity;        /**< Kapacita pole. */
} Expression;
```

Expression slouží jako úschovna pro takové *Term*, které nejde sloučit dohromady. Například $2 + 7$ dá dohromady *Term* 9, ale $x + y$ dohromady už nijak nesloučíme, místo toho rozšíříme *Expression* o další *Term*.

Celá tato operace probíhá v *postfix_parse_expression*. Ta na začátku vytáhne z globální instance fronty její velikost, a kompletně prochází postfixový zápis. Pokaždé

když narazí na proměnnou nebo číslo, tak vytvoří novou instanci *Expression*, která obsahuje pouze jeden *Term*. Ten si ukládám na speciální zásobník pro typ *Expression*.

Pokud při procházení postfixu narazím na logický operátor, ze zásobníku výrazů vytáhnu dva poslední výrazy a provedu mezi nimi matematickou operaci. Poté na zásobník znova vracím nově konjunkcí vytvořený výraz. Takto to provádíme, do té doby, než projdeme celý postfix a než je zásobník výrazů prázdný. Na závěr vrátíme výraz poslední, který zůstal sám. A to je finální výraz ze kterého můžeme získávat koeficienty našich rovnic.

Uživatelská příručka 4

Ke správnému využití naší kalkulačky si zde vypíšeme základy používání našeho programu. Jak s aplikací zacházet a jak jí spouštět.

4.1 Makefile

V adresáři našeho souboru se setkáme rovnou s dvěma **Makefile** soubory, jeden pro **Win32/64** a druhý pro **UNIX/Linux**. Makefile soubory, jsou speciální soubory pro zlehčení sestavení kompletní projektu vytvořeným v jazyce C.

Přesuňte se teď do linuxového prostředí a pojďme se používat na to, jaké příkazy můžeme používat pro práci s naším Makefile souborem. Zásadní je příkaz do konzole *make*, tím dáváme počítači najevo, že budeme pracovat s Makefilem v místní složce, další příkaz *all*, slouží k přemazání složky s vybudovaným projektem a novým vytvořením. Pokud chceme pouze smazat vybudovaný projekt, používáme příkaz *clean*.

Pokud chceme vyzkoušet program za běhu můžeme zvolit příkazy *run* nebo *runall*. Jediné, v čem se tyto dva příkazy liší, je počet argumentů. Pouhý *run* vypíše výsledek do terminálu, zatímco *runall* obsahuje i argument pro cílový výstupní soubor.

4.2 Spouštěcí argumenty

Už několikrát jsme zmínili, že program může využívat více spouštěcích argumentů. Pokud má argument pouze jeden, předpokládá se že se jedná o soubor se vstupními daty. Pokud chceme nadále specifikovat i soubor pro zápis výstupních informací, je potřeba jej nadefinovat příkazem *-o* nebo *–output*. Za tento příkaz můžeme rovnou uvést adresu k výstupnímu souboru.

Pokud se stane, že v argumentu se bude nacházet více definicí pro více výstupních souborů. Program automaticky převezme pouze ten poslední určený.

4.3 Chybové kódy

Program může skončit spousty způsoby. Zde jsou vypsány všechny kódy a jejich chybové hlášky:

Kód	Chybová hláška
0	EXIT SUCCESFULL
1	Input file not found!
2	Invalid output destination!
3	Alocation failed!
10	Unknown variable '<j>'!
11	Syntax Error!
20	Objective function is unbounded.
21	No feasible solution exists.

Tabulka 4.1: Chybovník

4.4 Správná syntaxe vstupních dat

Ještě si něco málo povíme ke správnosti syntaxi vstupních dat.

Opatrně na používání závorek. Kód si dokáže poradit s těmito druhy: [ ], ( ), { }. Důležité ale je, aby se zachovala jejich synergie. To znamená, že každý druh závorky musí mít svojí otevírací i zavírací část.

Další chyby kterým se na vstupu vyvarujte je nesmyslné psaní matematických rovnic, například používání více logických operátorů za sebou, nesmyslné symboly a znaky nebo špatně umístěnná desetinná místa.

Důležité si ohlídat i přesný počet všech sektorů a nezapomenout na sektor 'End', který vždy musí být na konci. A také správně sepsat použité proměnné do seznamu v sekci 'Generals'.

Závěr 5

V této dokumentaci jsme probrali postup celého vypracování semestrální práce z předmětu **Programování v jazyce C** na téma **Nástroj pro řešení úloh lineárního programování**. Začali jsme s rychlým vysvětlením zadání. Následně jsme na pár stránkách probrali celkovou analýzu problému a jak jej můžeme řešit. Ve třetí kapitole jsme si následně ukázali kódovou reprezentaci řešení našeho problému a na závěr jsme si prošli uživatelskou příručku, abychom mohli konzolovou aplikaci správně používat.

Při řešení zadání bylo největším překvapením, že načítání vstupu od uživatele bylo dvojnásobně náročnější než samotná úloha lineárního programování. To na druhou stranu bylo pěkně zdokumentované téma, které se dalo snadno pochopit, když člověk dával pozor. Téma jsem volil právě kvůli matematické stránce práce.

Největším problémem úplně celého programu bylo oddělení koeficientů proměnných a celých čísel. Ale i tak bych náročnost celé práce nijak nezveličoval. Časová náročnost odpovídala akreditaci předmětu a vývojové prostředí ANSCII C bylo na předmětu naučeno akorát tak, aby to na semestrální práci stačilo.

Tento program prošel všemi testy validátoru pro semestrální práce, takže úlohu beru za splněnou a alespoň dobře vyhotovenou.

Zdroje 6

1. Joshua Emmanuel. (2022, Srpen). *Intro to Simplex Method | Solve LP | Simplex Tableau* [Video]. YouTube. https://www.youtube.com/watch?v=9YKLXFqCy6E
2. Joshua Emmanuel. (2022b, Srpen). *Simplex Method 2 | Big M Tableau | Minimization Problem* [Video]. YouTube. https://www.youtube.com/watch?v=btjxqq-vMOg
3. Joshua Emmanuel. (2023, Srpen). *Special LP cases in Simplex Method | Infeasibility, Alternative solutions, unboundedness, Degeneracy* [Video]. YouTube. https://www.youtube.com/watch?v=FOdgHdLHvL8
4. Wikipedia contributors. (2024, Listopad). *Shunting yard algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Shunting_yard_algorithm
5. *Model File Formats - Gurobi Optimizer Reference Manual*. (n.d.). Gurobi optimization. https://docs.gurobi.com/projects/optimizer/en/current/reference/fileformats/modelformats.htmlformatlp
6. Ryjáček Z. (2016, Únor). *Teorie grafů a diskrétní optimalizace 2.* http://najada.fav.zcu.cz/ ryjacek/students/ps/TGD2.pdf

Seznam obrázků

Seznam tabulek

Seznam výpisů