

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky



**FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI**

NÁSTROJ PRO ŘEŠENÍ ÚLOH
LINEÁRNÍHO PROGRAMOVÁNÍ
PROGRAMOVÁNÍ V JAZYCE C
KIV/PC

Eduard Schödl
15.12.2024

Obsah

| | | |
|----------|--|-----------|
| 1 | Zadání | 3 |
| 2 | Analýza úlohy | 3 |
| 2.1 | Čtení vstupů ze souboru | 3 |
| 2.2 | Analýza dat | 4 |
| 2.3 | Metoda pro řešení problému | 4 |
| 2.4 | Vybrané řešení | 5 |
| 3 | Popis implementace | 5 |
| 3.1 | Moduly | 6 |
| 3.2 | Použité datové struktury | 6 |
| 3.3 | Dvoufázový Simplexový algoritmus | 6 |
| 3.3.1 | Fáze 1 – Hledání základního přípustného řešení | 7 |
| 3.3.2 | Fáze 2 – Řešení původního problému | 7 |
| 3.4 | Přenositelnost | 8 |
| 3.4.1 | Přenositelnost v makefile | 8 |
| 4 | Uživatelská příručka | 9 |
| 4.1 | Přeložení programu -Windows | 9 |
| 4.2 | Přeložení programu -Linux | 10 |
| 4.3 | Spuštění programu | 11 |
| 4.4 | Chybové hlášky | 11 |
| 4.4.1 | Fatální chyby | 11 |
| 4.4.2 | Upozornění | 12 |
| 4.5 | Formát vstupního souboru | 12 |
| 4.6 | Ukázka běhu | 13 |
| 4.6.1 | Příklad s optimálním řešením | 13 |
| 4.6.2 | Příklad s neexistujícím řešením | 13 |
| 4.6.3 | Příklad s chybou ve vstupním souboru | 15 |
| 5 | Závěr | 15 |
| | Zdroje | 16 |

1 Zadání

Cílem této semestrální práce je vytvořit program v jazyce C, který bude řešit problémy lineárního programování. Program musí splňovat následující požadavky:

- **Načítání vstupních dat:** Program musí být schopen načíst data ze vstupního souboru ve specifikovaném formátu, který obsahuje sekce jako *Maximize/Minimize*, *Subject To*, *Generals*, *Bounds* a *End*.
- **Validace vstupu:** Musí být zajištěna kontrola syntaxe a struktury vstupních dat. Program má detekovat chyby, jako jsou neplatné znaky, chybějící sekce nebo nesprávné matematické výrazy.
- **Výpočet optimálního řešení:** Program má implementovat vhodný algoritmus pro nalezení optimálního řešení lineárního problému.
- **Výstup výsledků:** Výsledky budou zobrazeny buď v příkazové řádce, nebo uloženy do výstupního souboru v přehledném formátu.
- **Přenositelnost:** Program musí být přenosný mezi operačními systémy Windows a Linux, přičemž jeho sestavení bude zajištěno pomocí makefile.

Podrobné zadání je sepsáno v dokumentu zveřejněném cvičícím [2].

2 Analýza úlohy

2.1 Čtení vstupů ze souboru

Nejdříve je nutné načíst data ze vstupního souboru, aby bylo s čím pracovat. Nabízejí se dvě možnosti, jak toho dosáhnout. Obě využívají již vestavěné funkce pro práci se soubory a to čtení po řádkách.

Na začátku je potřeba ověřit, zda daný soubor vůbec existuje. Toho lze dosáhnout jeho otevřením a následné kontroly návratové hodnoty dané funkce. Stejný výsledek poskytuje funkce **access**, která vrátí stejný závěr.

První z nich je přečíst řádek a ihned ho zpracovat a tak pokračovat až do konce souboru. Toto však přineslo několik problémů jako je nemožnost ukončení programu dříve z důvodu syntaktické chyby v souboru nebo neznalost velikosti dat ve smyslu velikosti matice, kterou se bude problém řešit.

Druhá možnost je přečíst všechny řádky najednou a uložit si je do pole, nebo do vytvořené struktury, kde budou jednotlivé řádky rozděleny podle sekce, ke které patří. To umožňuje pohodlnější operace, jsou předem známe velikosti dat a tím usnadňují práci s nimi. Negativem je opakované čtení týchž dat, jednou pro uložení a po druhé pro zpracování.

Hlavní problémy při čtení souboru zahrnují:

- ověření existence souboru,
- syntaktickou správnost dat,
- výkon a složitost zpracování.

Nabízí se dvě metody čtení dat:

1. Čtení souboru **po řádcích** a jejich okamžité zpracování.
2. Čtení **všech řádků najednou** a jejich uložení do pole/struktury.

Tabulka 1 shrnuje porovnání obou metod:

Tabulka 1: Porovnání metod čtení vstupního souboru

| Kritérium | Čtení po řádcích | Čtení všech řádků |
|------------------------|---------------------------------|-------------------------|
| Složitost implementace | Jednoduchá, průběžné zpracování | Vyšší, ukládání do pole |
| Detekce chyb | Obtížná, postupná | Snazší, kontrola předem |
| Paměťová náročnost | Nižší | Vyšší |
| Flexibilita zpracování | Omezená | Vysoká |

2.2 Analýza dat

Načtená vstupní data mohou obsahovat chyby, které nemusí způsobit předčasný konec aplikace, ale mohou zkreslit výpočty a dát tak špatný výsledný závěr o optimálním řešení problému. Je nezbytné, aby taková data byla nejprve validována.

V souboru se mohou vyskytovat chyby syntaktické a strukturální, jejichž přítomnost není žádoucí, a proto je nutné je detekovat a ukončit program s patřičným chybovým kódem a zprávou. Tyto chyby zahrnují špatně postavenou strukturu vstupního souboru, nekorektně formulované matematické výrazy nebo nepovolené znaky. [5]

Možné problémy zahrnují:

- **Syntaktické chyby:** neuzavřené závorky, špatná posloupnost operátorů.
- **Strukturální chyby:** nesprávné nebo chybějící sekce, špatné pořadí pořadí (obsah následující po sekci *End*).

Příklad chyby:

$$(x_1 + 2x_2 - \leq 5 \quad (\text{nekorektní parita závorek, znaménko před operátorem}) \quad (1)$$

V případě špatné struktury je možným řešením kontrolovat data již při načítání. Tím se zamezí například obsahu následujícím po sekci *End*, kde nic jiného již nesmí být nebo chybějícím datům v sekcích, kde jsou naopak vyžadována. Nelze však zajistit korektní názvy sekcí, nebo názvy, které nejsou sekcemi. Ty mohou být považovány za informace patřící k již zpracovávaným.

Vstupní soubor obsahuje obecně nerovnice používané pro řešení problému. Tyto nerovnice mohou být formou jak jednoduchých, tak složitých výrazů, u nichž je potřeba provést úpravy pro zjednodušení. Nejdříve je nutné, aby byl výraz správně zkonstruovaný, odpovídající závorky, korektní posloupnost operátorů. Možností je kontrolovat znak po znaku, ukládat informace o daném výrazu například v polích sloužících jako zásobník a tím zajistit korektní odstranění závorek, správně vyhodnocení násobení a záporného znaménka před závorkou. Neupravené výrazy mohou vést k nesprávnému sestavení matice a tím k chybnému výpočtu.

Příklad zjednodušení výrazu:

$$3x_1 - 2x_2 + 4x_3 - (5x_1 - 2x_2) \leq 10 \quad \Rightarrow \quad -2x_1 + 4x_3 \leq 10. \quad (2)$$

Tento proces zahrnuje:

- Kontrolu výrazů znak po znaku.
- Použití zásobníku pro správné párování závorek a vyhodnocení operátorů.

2.3 Metoda pro řešení problému

Po kontrole, zda jsou data správná a připravena k dalšímu zpracování, přichází na řadu samotný algoritmus pro řešení problémů lineárního programování.

Nabízí se mnoho metod pro řešení problémů lineárního programování. Mohou to být grafické metody, metoda vnitřních bodů nebo Simplexová metoda a její varianty. Liší se však složitostí implementace. Grafická metoda umí pracovat pouze se dvěma proměnnými a tudíž není vhodná. Nejlepší

volbou tak je Simplexová metoda, přesněji její Two-phase varianta. Ta řeší problémy obecně a hodí se jak na jednoduchá, tak i složitá zadání a je poměrně jednoduchá na realizaci. Navíc je schopna řešit problémy s umělými proměnnými.

Metody pro řešení problému lineárního programování:

- **Grafická metoda:** Použitelná pouze pro dvě proměnné.
- **Metoda vnitřních bodů:** Rychlá, ale složitá na implementaci.
- **Simplexová metoda (Two-phase):** Univerzální a vhodná i pro problémy s umělými proměnnými.
- ...

Tabulka 2 shrnuje srovnání metod:

Tabulka 2: Porovnání metod pro řešení LP problémů

| Metoda | Výhody | Nevýhody | Vhodnost |
|-----------------------|-------------------------|-----------------------------|--------------|
| Grafická metoda | Jednoduchá | Pouze pro 2 proměnné | Nevhodná |
| Metoda vnitřních bodů | Rychlá | Složitá implementace | Omezená |
| Two-phase Simplex | Flexibilní, univerzální | Střední výpočetní náročnost | Nejvhodnější |

2.4 Vybrané řešení

Za finální řešení bylo zvoleno:

- **Čtení všech řádků najednou:** umožňuje snadnou validaci dat.
- **Analýza znak po znaku:** zajišťuje syntaktickou správnost.
- Zvolená metoda je **Two-phase Simplex**, která je schopna:
 - řešit problémy obecného tvaru,
 - zvládnout umělé proměnné,
 - snadno se implementuje.

Tímto postupem je zajištěna správnost dat, jejich validace a optimalizované řešení úlohy.

3 Popis implementace

Implementace projektu byla rozdělena do jednotlivých částí pro zajištění přehlednosti a snadné údržby kódu. Každý modul řeší specifickou oblast programu a je umístěn v odpovídajícím adresáři.

Na začátku probíhá kontrola argumentů příkazové řádky. Program vyžaduje, aby byl zadán alespoň vstupní soubor. Poté jsou zadané cesty k souborům zpracovány. Pokud je specifikován i výstupní soubor, jeho cesta je rovněž ověřena. Aplikace se pokusí soubory otevřít, a pokud se to nepodaří, uživatele na problém upozorní a ukončí vykonávání.

Pokud byly všechny kroky úspěšné, vytvoří se struktura pro uložení dat a vstupní soubor se do ní načte. Program pokračuje zpracováním jednotlivých sekcí a vytvořením simplexové tabulky. V průběhu zpracování probíhá validace syntaxe a struktury dat, aby byla zajištěna jejich správnost a konzistence.

Po úspěšném zpracování všech vstupů začne samotný Simplexový algoritmus, který se pokusí najít optimální řešení zadaného problému. Výsledky jsou následně vypsány do příkazové řádky nebo do specifikovaného výstupního souboru.

3.1 Moduly

- **main.c** - Obsahuje hlavní funkci programu, která zajišťuje inicializaci, volání jednotlivých modulů a koordinaci fází zpracování problému lineárního programování.
- **file.c** - Obstarává práci spojenou se soubory. Jejich získání z příkazové řádky, otevření a zápis.
- **parse.c** - Modul, který se stará o práci s výrazy a řetězci. Obsahuje funkce například pro odstranění mezer, získání koeficientů z výrazu nebo jejich zjednodušení.
- **section_buffer.c** - Zajišťuje vytvoření struktury uchováající jednotlivé řádky vstupu, ukládání řádek a její uvolnění.
- **Moduly jednotlivých sekcí LP souboru** - Moduly starající se o zpracování jednotlivých sekcí obsažených ve vstupním souboru. Patří sem moduly *bounds.c*, *generals.c*, *objectives.c*, *subject_to.c*.
- **validate.c** - Zajišťuje správnost vstupních dat.
- **lp.c** - Samotná implementace veškerých potřebných funkcí pro práci se Simplexovou metodou.
- **memory_manager.c** - Obsahuje obalové funkce nad funkcemi pro správu práce s pamětí. Dodává jim funkčnost počítání alokace a uvolnění.

3.2 Použité datové struktury

- **Section_Buffers** - Struktura, která ukládá řádky pro jednotlivé sekce a jejich počet.
- **Simplex_Tableau** - Struktura reprezentující matici pro tabulku Simplexové metody. Dále obsahuje typ optimalizace a počet řádků a sloupců.
- **General_vars** - Struktura uchováající názvy rozhodovacích proměnných ze sekce *Generals*.
- **Bounds** - Struktura obsahuje meze pro jednotlivé proměnné. Výchozí hodnotou je $\langle 0, \infty \rangle$.
- **Term** - Struktura sloužící pro uložení termínů matematického výrazu.

3.3 Dvoufázový Simplexový algoritmus

Dvoufázový Simplexový algoritmus implementovaný v programu je rozdělen do dvou fází, které umožňují řešení problémů, kde počáteční řešení není přímo známo nebo kde jsou přítomny rovnosti a nerovnosti. Tyto nerovnosti jsou nejdříve převedeny na rovnosti úpravami:

1. Nerovnosti typu \leq

Pro nerovnosti, které mají tvar:

$$x_1 + x_2 \leq 5 \quad (3)$$

je přidána kladná pomocná proměnná (tzv. slack proměnná) s_1 , aby se dosáhlo rovnosti:

$$x_1 + x_2 + s_1 = 5, \quad s_1 \geq 0 \quad (4)$$

2. Nerovnosti typu \geq

Pro nerovnosti, které mají tvar:

$$x_1 + x_2 \geq 5 \quad (5)$$

je přidána záporná pomocná proměnná (tzv. surplus proměnná) s_1 a umělá proměnná y_1 pro dosažení rovnosti:

$$x_1 + x_2 - s_1 + y_1 = 5, \quad s_1 \geq 0, \quad y_1 \geq 0 \quad (6)$$

Umělá proměnná y_1 je zavedená, aby bylo možné najít počáteční přípustné řešení během fáze 1 algoritmu.

3. Rovnosti typu =

Rovnice ve tvaru:

$$x_1 + x_2 = 5 \quad (7)$$

zůstávají nezměněny, avšak v tomto případě je přidána pouze umělá proměnná y_1 :

$$x_1 + x_2 + y_1 = 5, \quad y_1 \geq 0 \quad (8)$$

Tím se zajistí, že rovnost bude splněna během první fáze algoritmu.

Následuje podrobný popis jednotlivých fází:

3.3.1 Fáze 1 – Hledání základního přípustného řešení

Tato fáze se zaměřuje na nalezení základního přípustného řešení splňujícího všechna omezení. Pokud takové řešení neexistuje, algoritmus končí s informací, že problém je neřešitelný.

1. Přidání umělých proměnných

U rovností a nerovností typu „ \geq “ a „ $=$ “ je nutné přidat umělé proměnné, aby bylo zajištěno počáteční přípustné řešení. Například:

$$x_1 + x_2 \geq 5 \quad \Rightarrow \quad x_1 + x_2 - y_1 = 5, \quad y_1 \geq 0 \quad (9)$$

2. Definice pomocné cílové funkce

Pomocná cílová funkce minimalizuje součet všech umělých proměnných:

$$\min z = y_1 + y_2 + \dots \quad (10)$$

3. Iterace v Simplexové tabulce

Algoritmus iterativně provádí pivotování (tj. úpravy tabulky) s cílem minimalizovat hodnotu pomocné cílové funkce z . Pokud $z > 0$ na konci této fáze, problém nemá přípustné řešení.

4. Odstranění umělých proměnných

Pokud $z = 0$, všechny umělé proměnné jsou odstraněny z tabulky, což znamená, že základní přípustné řešení bylo nalezeno. Tím je připraveno řešení původního problému.

3.3.2 Fáze 2 – Řešení původního problému

Po nalezení základního přípustného řešení v první fázi algoritmus přechází k řešení původního problému.

1. Nastavení původní cílové funkce

Do Simplexové tabulky je nyní zavedena původní cílová funkce (například maximalizace $z = c_1x_1 + c_2x_2 + \dots$).

2. Optimalizace cílové funkce

Algoritmus iterativně provádí pivotování, aby optimalizoval hodnotu cílové funkce:

- **Výběr pivotního sloupce** – Hledá se nejlepší sloupec pro zlepšení cílové funkce. Nejmenší hodnota pro maximalizaci, největší pro minimalizaci.
- **Výběr pivotního řádku** – Hledá se řádek s nejmenším podílem pravé strany a pivotního sloupce. Záporné hodnoty se ignorují.
- **Úprava tabulky** – Probíhá pomocí Gaussovy eliminace.

3. Konečné řešení

Po optimalizaci cílové funkce algoritmus kontroluje, zda nalezené řešení splňuje všechna omezení (například $x_i \geq 0$). Pokud je cílová funkce neomezená, algoritmus vypíše upozornění a skončí.

3.4 Přenositelnost

Jelikož bylo součástí zadání zajistit přenositelnost programu alespoň pro systémy **Windows** a **Linux**, bylo nezbytné implementovat podmíněné připojování knihoven a redefinice funkcí pomocí preprocesorových direktiv. Každý systém má vlastní implementaci některých funkcí a knihoven, proto je nezbytné program přizpůsobit tak, aby běžel na obou platformách beze změny zdrojového kódu.

Podmíněné připojování knihoven je řešeno pomocí direktiv `#ifdef`, `#elif` a `#endif`. Program detekuje cílovou platformu prostřednictvím standardních maker:

- `_WIN32` - Definováno na platformě Windows
- `__linux__` - Definováno na platformě Linux

Podle těchto maker jsou zahrnuty odpovídající knihovny a definice funkcí. Ukázkový kód:

```
#ifdef _WIN32
    #define strcasecmp _stricmp
    #define LINE_BREAK "\n"
#elif __linux__ #ifdef _WIN32
    #include <inttypes.h>
    #include <unistd.h>
    #define __int64 int64_t
    #define _close close
    #define _read read
    #define _lseek64 lseek64
    #define _O_RDONLY O_RDONLY
    #define _open open
    #define _lseeki64 lseek64
    #define _lseek lseek
    #include <strings.h>
    #define LINE_BREAK "\r"
#endif #elif __linux__
```

Obrázek 1: Příklad podmíněného připojení knihoven a mapování funkcí.

3.4.1 Přenositelnost v makefile

Pro zajištění přenositelnosti při kompilaci pomocí překladače `gcc` byl upraven soubor `makefile`. V rámci něj jsou automaticky detekovány podmínky cílového systému a použity odpovídající překladačové příkazy.

Soubor obsahuje cíl pro vyčištění projektu od objektových souborů a spustitelného souboru. Systémy však používají rozdílné příkazy a zápis cest v adresářích. Soubor však na názkladě cílového systému vybere správné příkazy pro mazání a v případě **Windows**, nahradí znaky `/` za zpětné.

Ukázka:

```
ifeq ($(OS),Windows_NT)
    RM = del /Q
else
    RM = rm -f
endif
```

Obrázek 2: Podmíněný příkaz mazání.

```
clean:
ifeq ($(OS),Windows_NT)
    $(RM) $(subst /,\\,$(OBJ)) $(subst /,\\,$(EXE))
else
    $(RM) $(OBJ) $(EXE)
endif
```

Obrázek 3: Modifikace cest pro Windows.

4 Uživatelská příručka

4.1 Přeložení programu - Windows

Pro přeložení programu na platformě Windows je nezbytné mít nainstalovaný překladač GCC, například prostřednictvím MinGW, nebo použít nástroje Microsoft Visual C/C++. Program je připraven k překladač pomocí souboru `makefile` nebo `makefile.win`, který obsahuje pravidla pro sestavení.

Postup překladač pro GCC:

1. Stáhněte a nainstalujte MinGW (pokud není již nainstalováno). Ujistěte se, že je `gcc` a `make` přidán do systémové proměnné `PATH`.
2. Otevřete příkazový řádek (CMD nebo PowerShell).
3. Přejděte do složky s programem příkazem:

```
cd cesta\k\souborum
```

4. Spusťte příkaz pro sestavení programu:

```
mingw32-make
```

Tento příkaz automaticky přeloží všechny zdrojové soubory a vytvoří spustitelný soubor `lp.exe`.

5. Pokud chcete projekt vyčistit od dočasných nebo zkompileovaných souborů, použijte příkaz:

```
mingw32-make clean
```

Tento příkaz odstraní všechny soubory `.o` a výstupní spustitelné soubory.

Postup překladač pro `cl` (Microsoft Visual C/C++):

1. Ujistěte se, že máte nainstalováno Microsoft Visual Studio a příkazový řádek nástroje `Developer Command Prompt for Visual Studio`.
2. Otevřete `Developer Command Prompt for Visual Studio`. Tento příkazový řádek automaticky nastaví všechny potřebné proměnné prostředí pro překladač `cl` a nástroj `nmake`.
3. Přejděte do složky s programem příkazem:

```
cd cesta\k\souborum
```

4. Spusťte příkaz pro sestavení programu pomocí `makefile.win`:

```
nmake /f makefile.win
```

Tento příkaz použije překladač `cl`, aby přeložil všechny zdrojové soubory a vytvořil spustitelný soubor `lp.exe`.

5. Pokud chcete projekt vyčistit od dočasných nebo zkompileovaných souborů, použijte příkaz:

```
nmake /f makefile.win clean
```

Tento příkaz odstraní všechny soubory `.obj` a výstupní spustitelné soubory.

4.2 Přeložení programu - Linux

Na operačních systémech typu Linux je k přeložení programu nutné mít nainstalovaný `gcc` a nástroj `make`, které jsou většinou dostupné prostřednictvím systémového správce balíčků. Program je připraven k překladač pomocí souboru `makefile`, který obsahuje pravidla pro sestavení.

Postup překladač:

1. Ověřte instalaci `gcc` a `make` pomocí příkazu:

```
gcc --version  
make --version
```

Pokud některý z nástrojů není nainstalován, nainstalujte je pomocí balíčkovacího systému, například:

```
sudo apt install gcc make
```

2. Otevřete terminál a přesuňte se do adresáře s programem:

```
cd cesta/k/souborum
```

3. Spusťte příkaz:

```
make
```

Tento příkaz přeloží program a vytvoří spustitelný soubor `lp.exe`.

4. Chcete-li vyčistit projekt od zkompileovaných souborů, použijte:

```
make clean
```

Tento příkaz odstraní všechny dočasné a výstupní soubory vytvořené během kompilace.

4.3 Spuštění programu

Po úspěšném přeložení lze program spustit prostřednictvím příkazové řádky. Program vyžaduje jako povinný argument cestu k vstupnímu souboru obsahujícímu definici problému lineárního programování. Jako druhý volitelný argument je, pomocí přepínače `-o` nebo `--output`, předána cesta k výstupnímu souboru, kam bude výsledek programu uložen. V případě jeho neuvedení se výsledek vypíše přímo do příkazové řádky.

Příklad spuštění programu:

```
lp.exe input.lp -o output.txt
```

Tento příkaz spustí program, načte data z `input.lp` a zapíše výsledky do souboru `output.txt`. Pokud není uveden argument `-o`, výsledky budou vypsány do příkazové řádky.

4.4 Chybové hlášky

Program jednoduchým způsobem komunikuje s uživatelem o svém stavu. V případě, kdy nastane fatální chyba, je uživateli uživateli vypsána chybová hláška a běh programu je ukončen. Pokud ale chyba není podstatná pro fungování aplikace, tak je uživatel pouze upozorněn a program pokračuje dále.

4.4.1 Fatální chyby

Fatální chyby jsou takové, které znemožňují pokračování programu, například syntaktické chyby ve vstupních datech, nesprávná struktura souboru nebo chyba spojená s alokací paměti.

Tabulka 3: Fatální chyby programu

| Chyba | Popis | Chybová hláška |
|------------------------------|---|-----------------------------|
| Neexistující vstupní soubor | Žadáný vstupní soubor nebyl nalezen. | Input file not found! |
| Neexistující výstupní soubor | Žadáný výstupní soubor neexistuje. | Invalid output destination! |
| Neznámá proměnná | Proměnná použitá ve výrazech není specifikována v sekci <i>Generals</i> . | Unknown variable '<j>' |
| Syntaktické chyby | Výskyt neplatných operátorů, neznámých sekcí a jiné problémy. | Syntax error! |

4.4.2 Upozornění

Tyto chyby uživatele upozorní, ale neblokují další zpracování. V programu se nachází pouze upozornění na nepoužitou proměnnou.

Tabulka 4: Upozornění programu

| Chyba | Popis | Chybová hláška |
|--------------------|---|---------------------------------|
| Nepoužitá proměnná | Proměnná ze sekce <i>Generals</i> není použita ve výrazech. | Warning: unused variable '<n>'! |

4.5 Formát vstupního souboru

Vstupní soubor musí obsahovat definici problému lineárního programování ve specifickém formátu. Každá sekce je označena klíčovým slovem následovaným daty:

- **Maximize / Minimize:** Definuje cílovou funkci, například:

```
Maximize
z = 2x + 3y
```

- **Subject To:** Obsahuje omezení ve formě nerovnic:

```
Subject To
x + y <= 10
2x + 3y >= 5
```

- **Bounds:** Definuje hranice pro proměnné:

```
Bounds
0 <= x <= 5
y free
```

- **Generals:** Seznam používaných proměnných:

```
Generals
x y
```

- **End:** Označuje konec souboru.

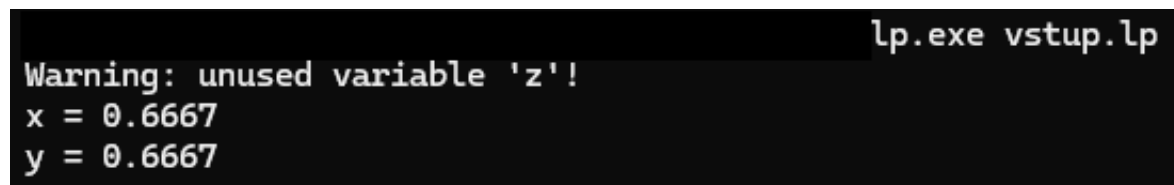
4.6 Ukázka běhu

4.6.1 Příklad s optimálním řešením

```
1 Maximize
2   x + y
3
4 Subject To
5   c1: x + 2*y <= 2
6   c2: 2x + y <= 2
7
8 Bounds
9   x >= 0
10  y >= 0
11
12 Generals
13   x y z
14
15 End
```

Listing 1: Vstupní soubor s optimálním řešením.

Po úspěšném zpracování vstupních dat a nalezení optimálního řešení je výstup programu následující:



```
Warning: unused variable 'z'!
x = 0.6667
y = 0.6667
```

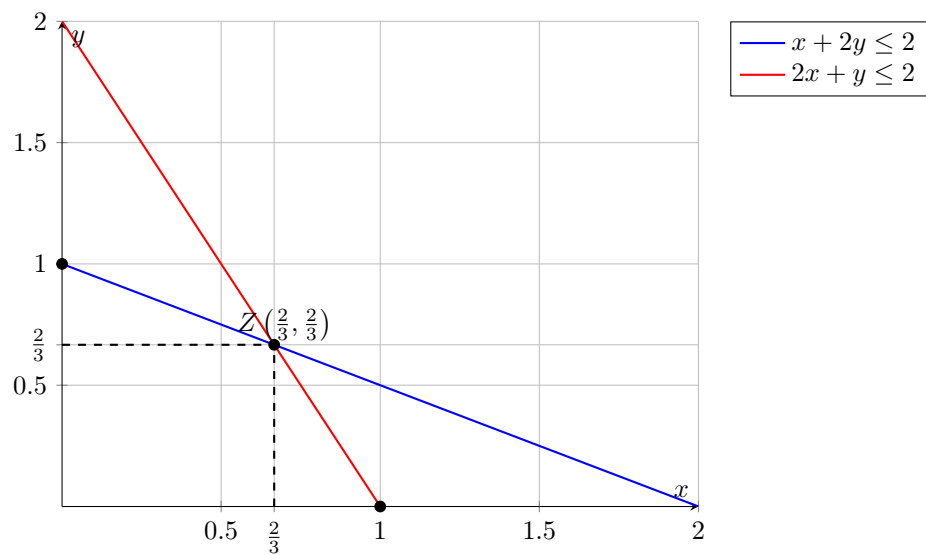
Obrázek 4: Výsledek běhu programu s optimálním řešením a nepoužitou proměnnou.

Optimální řešení je v bodě $Z = (\frac{2}{3}, \frac{2}{3})$, kde je hodnota $Z = x + y$ maximalizována.

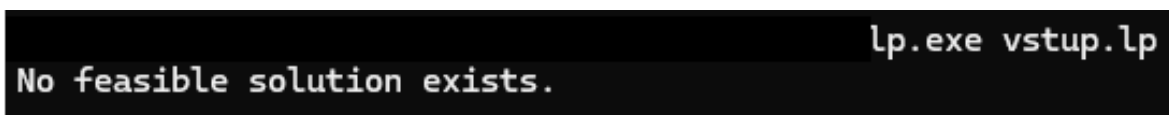
4.6.2 Příklad s neexistujícím řešením

```
1 Minimize
2   x - y
3
4 Subject To
5   -5x - (-y - 3x) >= 2
6   x - 2*y >= 2
7
8 Bounds
9   x >= 0
10  y free
11
12 Generals
13   x y
14
15 End
```

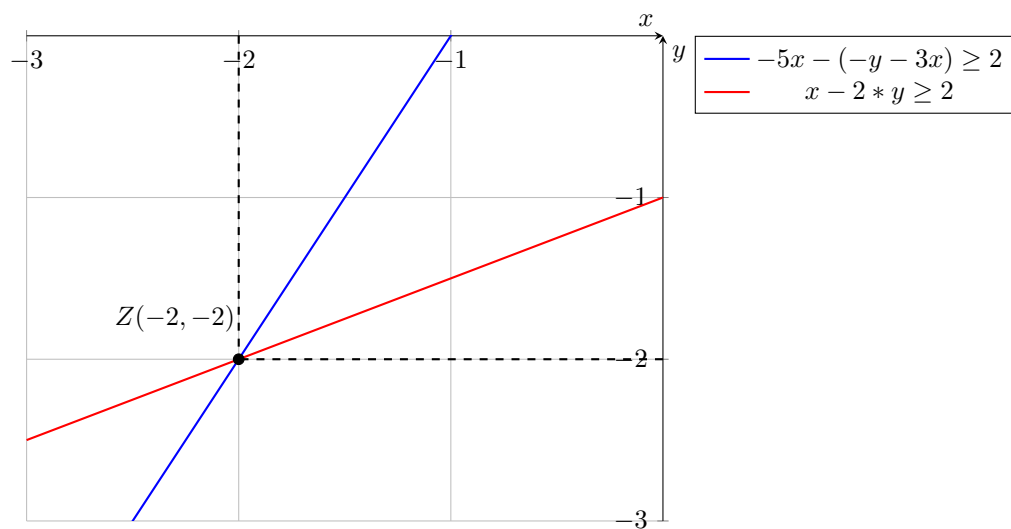
Listing 2: Vstupní soubor s neexistujícím řešením.



Obrázek 5: Vizualizace řešení LP problému s optimálním řešením.



Obrázek 6: Výsledek běhu programu s neexistujícím řešením.



Obrázek 7: Vizualizace řešení LP problému s neexistujícím řešením.

Po úspěšném zpracování vstupních dat a nalezení optimálního řešení je výstup programu následující:

4.6.3 Příklad s chybou ve vstupním souboru

```
1  Maximize
2      {(2x + y)          \neparitní závorky
3
4  Subject To
5      x +* 2y <= 1      \špatné pořadí operátorů
6
7  Bounds
8      x >= 0
9      5 <= y <= 80
10
11  Generals
12      x y
13
14  End
```

Listing 3: Vstupní soubor s chybou.



Obrázek 8: Výsledek běhu programu se syntaktickou chybou.

5 Závěr

V rámci této semestrální práce byl vytvořen program schopný řešit problémy lineárního programování pomocí dvoufázové Simplexové metody. Program byl implementován v jazyce C. Vstupní data jsou zpracována ze souboru a validována, čímž je zajištěno, že program pracuje pouze s korektními daty. Výsledky jsou následně vypsány do příkazové řádky nebo do výstupního souboru.

Program splňuje zadání a jeho implementace byla úspěšně dokončena. Různá vstupní data byla otestována a výsledky byly porovnány s jinými nástroji pro řešení problémů lineárního programování.

Během implementace však nastaly některé výzvy. Nejvýznamnějším problémem bylo kontrolování syntaxe vstupních dat. Validace byla komplikována různými možnostmi, které uživatel mohl zadat, a bylo nutné navrhnout univerzální a robustní řešení. Další výzvou byla implementace samotné Simplexové metody. Přestože je dobře zdokumentována, zdroje se v některých krocích lišily, což vyžadovalo důkladné studium a ověření správnosti zvoleného přístupu.

Do budoucna by bylo vhodné zlepšit robustnost validace vstupních dat a zoptimalizovat výpočet řešení problému lineárního programování. Dalším krokem by mohlo být rozšíření funkcionality o vizualizaci výsledků, což by usnadnilo použití programu a lepší orientaci.

Zdroje

- [1] Vijayasri Iyer. *Simplex Method: The Easy Way*. 2021. URL: <https://vijayasriiyer.medium.com/simplex-method-the-easy-way-f19e61095ac7>.
- [2] František Pártl. *Zadání semestrální práce - Nástroj pro řešení úloh lineárního programování*. 2024. URL: <https://www.kiv.zcu.cz/studies/predmety/pc/data/works/sw2024-03.pdf>.
- [3] Z. Ryjáček. *Teorie grafů a diskrétní optimalizace 2*. 2001. URL: <http://najada.fav.zcu.cz/~ryjacek/students/ps/TGD2.pdf>.
- [4] *Simplex method calculator*. URL: <https://www.emathhelp.net/calculators/linear-programming/simplex-method-calculator/>.
- [5] *Specifikace formátu LP*. URL: <https://docs.gurobi.com/projects/optimizer/en/current/reference/fileformats.html>.