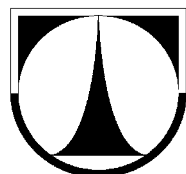
Technická univerzita v Liberci

Fakulta strojní



DIPLOMOVÁ PRÁCE

Problém obchodního cestujícího a jeho aplikace v GIS

Liberec 2008 Pavel Kříž

Technická univerzita v Liberci

Fakulta strojní

Studijní obor : 3902T021 Automatizované systémy řízení ve strojírenství

Zaměření : Automatizace inženýrských prací

Katedra aplikované kybernetiky

Problém obchodního cestujícího a jeho aplikace v GIS

Traveling salesman problem and its application in GIS

Pavel Kříž

Vedoucí diplomové práce : prof. Ing. Vladimír Věchet, CSc.

Konzultant diplomové práce: Ing. Jan Váša

ANOTACE

Technická univerzita v Liberci

Fakulta strojní

Katedra aplikované kybernetiky

Studijní obor :	3902T021 Automatizované systémy řízení ve strojírenství
Studijní zaměření :	Automatizace inženýrských prací
Diplomant :	Pavel Kříž
Téma práce :	Problém obchodního cestujícího a jeho aplikace v GIS
Theme of work :	Traveling salesman problem and its application in GIS
Rok obhajoby DP :	2008
Vedoucí DP :	prof. Ing. Vladimír Věchet, CSc.
Konzultant DP :	Ing. Jan Váša

Stručný výtah :

Cílem této diplomové práce je implementace algoritmů pro řešení problému obchodního cestujícího v jazyce C. Tyto algoritmy budou použity v aplikaci, která umožní uživateli výběr křižovatek ze seznamu, jejich zobrazení na silniční síti České republiky a dle zvoleného způsobu výpočtu zjistí nejkratší Hamiltonovskou kružnici. Aplikace je psána v jazyce C#.

Abstract:

The aim of the diploma work is implementation of algorithms for solving the traveling salesman problem in C programming language. These algorithms will be used in the application, which enables the customer to choose crossroads from the list and their depiction in the Czech Republic highway net and according to chosen method of calculation he ascertains the shortest Hamilton circle. Application is written in language C#.

Prohlášení o využití diplomové práce

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č.121/2000 o právu autorském, zejména § 60 (školní dílo) a § 35 (o nevýdělečném užití díla k vnitřní potřebě školy).

Beru na vědomí , že TUL má právo na uzavření licenční smlouvy o užití mé práce a prohlašuji, že **souhlasím** s případným užitím mé práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

V Liberci 3.5.2008

……………………….

Pavel Kříž

Místopřísežné prohlášení

„Místopřísežně prohlašuji, že jsem diplomovou práci vypracoval samostatně s použitím uvedené literatury.“

V Liberci 3.5.2008

………………………

Pavel Kříž

Obsah

1. Úvod **9**

2. Problém obchodního cestujícího (TSP) **10**

2.1. Struktura uložení a přístup k datům 12

2.1.1. Vstupní data *12*

2.1.2. Struktura uložení grafu *13*

2.1.3. Datová struktura zásobník *15*

2.1.4. Datová struktura seznam *17*

2.2. Nejkratší cesta 20

2.2.1. Modifikovaný Dijkstrův algoritmus *21*

2.2.2. Prohledávání s návratem *22*

2.2.3. Popis výpočtu v jazyce C *23*

2.3. Úplný graf 24

2.3.1. Postup výpočtu *24*

2.3.2. Popis výpočtu v jazyce C *25*

2.4. Metody řešení TSP 26

2.4.1. 1-strom a metoda penalizace *26*

2.4.2. Metoda postupného vkládání vrcholu *29*

2.4.3. Metoda nejvzdálenějšího vkládání vrcholu *31*

3. Popis GIS aplikace psané v C# .NET **33**

3.1. Vzhled aplikace 34

3.1.1. Hlavní okno *34*

3.1.2. Okno pro zobrazení výsledků *38*

3.2. Struktura uložení a přístup k datům 39

3.2.1. Vstupní data *39*

3.2.2. Struktura uložení grafu *41*

3.2.3. Načtení dat z externích knihoven *43*

3.3. Zobrazení vstupních dat 44

3.3.1. Převod souřadnic *44*

3.3.2. Vykreslení mapy, vygenerované cesty a vybraných křižovatek *45*

3.4. Uživatelské rozhraní 47

3.4.1. Výběr křižovatek ... 47
3.4.2. Výběr metody výpočtu a vlastní výpočet ... 47
3.4.3. Zvětšení (zmenšení) mapy ... 48
4. Závěr ... 49
Použitá literatura ... 51
Přílohy na CD:
Příloha 1 - Zdrojové texty v jazyce C.
Příloha 2 - Zdrojové texty v jazyce C#.
Příloha 3 - GIS aplikace.

1. Úvod

Cílem práce je vytvořit pro vstupní data algoritmy pro řešení problému obchodního cestujícího v jazyce C a zobrazit řešení v geografickém informačním systému (dále jen GIS), který bude implementován v aplikaci psané v jazyce C#. Tato práce je vypracována na základě zadání firmy Tranis s.r.o, která se zabývá tvorbou dopravního softwaru.

Asi každý z nás se snažil při cestě nějakým dopravním prostředkem najít nejkratší možnou cestu z určitého místa *A* na silniční síti do cílového místa *B*. Tuto cestu nalézt není v dnešní době žádný problém. Může se použít nějaké navigační zařízení nebo se nechá vyhledat na internetu. Co když ale bude zapotřebí vyjet z určitého místa *A*, projet několika *n* ($n > 1$) místy na silniční síti a vrátit se zpět do výchozího místa *A*?

Tato úloha se nazývá problém obchodního cestujícího (z angl. traveling salesman problem – dále jen TSP) a řeší se pomocí teorie grafů. Bude řešena v programovacím jazyce C. Aby se mohla volit místa na silniční síti a prezentovat výsledky, bude vypracována GIS aplikace v jazyce C# .NET (dále jen aplikace). Tato práce je tedy rozdělena na dvě části. Jedna je řešení problému obchodního cestujícího v jazyce C a druhá je vypracování aplikace.

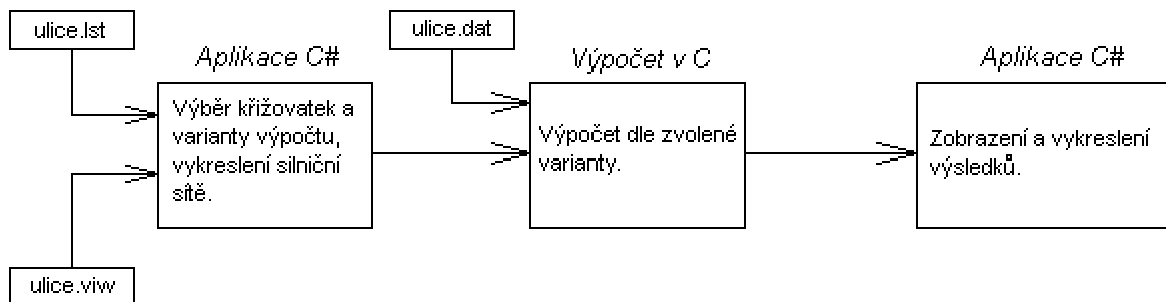
Vstupní data jsou z databáze silnic firmy Tranis s.r.o. Obsahují informace o silniční síti České republiky, ale nejsou úplná. Vyskytují se zde pouze silnice 1. třídy. Jsou uloženy ve třech binárních souborech *ulice.dat*, *ulice.viw* a *ulice.lst*. Soubory *ulice.viw* a *ulice.lst* jsou použity v aplikaci (viz. kapitola 3.2.1.) a soubor *ulice.dat* slouží k výpočtu v jazyce C (viz. kapitola 2.1.1.). Schéma předávání dat je na obr.2.1.

Pro řešení TSP existuje více algoritmů. V této práci jsou použity algoritmy, které se nazývají 1-strom a metoda penalizace, metoda postupného vkládání a metoda nejvzdálenějšího vkládání. Pro dosažení cíle této práce jsou dostačující.

2. Problém obchodního cestujícího (TSP)

Problém obchodního cestujícího se může popsat takto: obchodní cestující má za úkol navštívit v libovolném pořadí *n* měst a vrátit se zpět tak, aby jeho trasa byla co nejkratší. Přitom se předpokládá, že vzdálenosti mezi všemi dvojicemi měst jsou předem známé a symetrické (v obou směrech je vzdálenost stejná). Místo měst se bude pracovat s křižovatkami.

Vstupními daty pro výpočet v jazyce C je soubor *ulice.dat* a data z aplikace (viz. obr. 2.1). V souboru jsou uloženy informace o silniční síti. Data z aplikace jsou identifikační čísla křižovatek, pro které budeme hledat řešení.



Obr. 2.1: Schéma postupu řešení a předávání dat.

V teorii grafů jde o nalezení nejkratší hamiltonovské kružnice v úplném grafu, jehož hrany jsou ohodnoceny délkami. Úplný graf je takový graf, kde každé jeho dva vrcholy jsou spojeny hranou. Má tu výhodu, že v něm existuje hamiltonovská kružnice a tím i řešení problému obchodního cestujícího. Tento graf je třeba nejprve vytvořit a to tak, že pro každé dvě identifikační čísla křižovatek na vstupu se vyhledá nejkratší cesta. Proto je zvolen tento postup výpočtu:

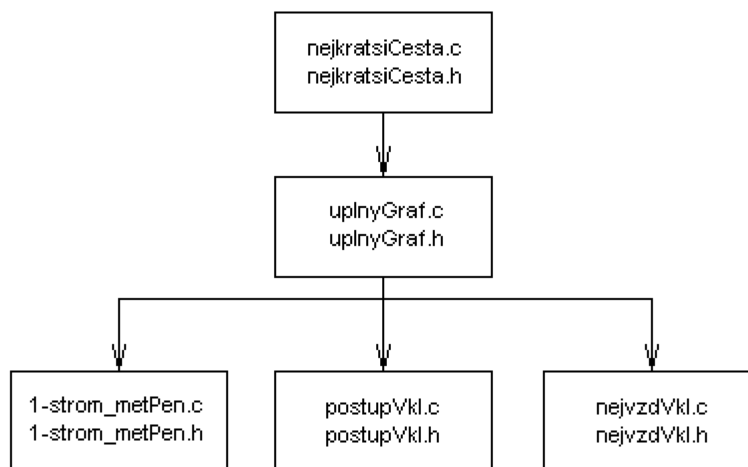
- nalezení nejkratší cesty mezi dvěma křižovatkami,
- nalezení úplného grafu,
- implementace algoritmů pro řešení TSP

Algoritmy pro nalezení nejkratší cesty a úplného grafu slouží jako příprava vstupních hodnot pro řešení TSP. Výpočet pomocí těchto algoritmů je časově náročný. Algoritmy pro

řešení TSP se ale bez těchto výpočtů neobejdou, proto jsou započítány do celkové doby výpočtu problému obchodního cestujícího. Z toho plyne, že každá varianta řešení si nejprve nalezne úplný graf a pak teprve řeší problém TSP.

Všechny tyto algoritmy jsou implementovány v jazyce C. Aby se mohly využít v aplikaci, jsou z nich vytvořeny dynamické knihovny. Každá varianta je jedna knihovna. Zdrojové texty jsou vypracovány použitím Microsoft Visual C++ 2008 Express Edition, který je volně stažitelný z internetových stránek firmy Microsoft (viz. int. adresa [9]).

Aby bylo možné používat algoritmy pro nalezení úplného grafu pro každou variantu výpočtu, je zdrojový text v jazyce C rozdělen do několika modulů. Pro nalezení nejkratší cesty mezi dvěma křižovatkami je použit modul *nejkratsiCesta.c*, pro nalezení úplného grafu je použit modul *uplnyGraf.c*, pro řešení TSP metodou 1-strom a metoda penalizace je použit modul *1-strom_metPen.c*, pro řešení TSP metodou postupného vkládání je použit modul *postupVkl.c* a pro řešení TSP metodou nejvzdálenějšího vkládání je použit modul *nejvzdVkl.c*. Ke každému modulu je vytvořen hlavičkový soubor, v němž jsou uloženy deklarace funkcí a definice použitých struktur a který má stejný název, ale příponu *.h*. Závislost modulů je na obr. 2.2. Z této závislosti je vidět, že pokud se vytváří dynamická knihovna např. pro metodu postupného vkládaní, musí se použít moduly *nejkratsiCesta*, *uplnyGraf* a *postupVkl* (s příponou *.c* i *.h*). Všechny zdrojové texty psané v jazyce C jsou uloženy na přiloženém CD v příloze 1.



Obr.2.2: Skládání a závislost modulů.

Do aplikace psané v jazyce C# se tyto algoritmy vkládají pomocí dynamických knihoven. Dynamická knihovna se vytvoří tak, že se vytvoří nový projekt *File→New→Project*

(klávesová zkratka *Ctrl+Shift+N* nebo ikona). Objeví se nabídka šablon. Typ projektu je *Win32* ve složce *Visual C++ Projects*, šablona je *Win32 Console Project*. Dále se zvolí umístění projektu, napíše název a stiskne *OK*. Objeví se průvodce *Win32 Application*, kde se na kartě *Application Settings* zvolí typ aplikace *DLL*. Jako další volba se označí *Empty project*, stiskne tlačítko *Finish* a objeví se nový prázdný projekt. Moduly s koncovkou *.c* se vkládají do složky *Source Files* v okně *Solution Explorer* stisknutím pravého tlačítka myši nad touto složkou a výběrem v kontextovém menu *Add→Add Existing Item…*. Moduly s koncovkou *.h* se vkládají stejným způsobem do složky *Header Files* také v okně *Solution Explorer*. Funkce, která má být v dynamické knihovně volána, musí být označena modifikátorem *__declspec(dllexport)* a nesmí chybět hlavní funkce *DllMain*. Nyní je možné projekt sestavit *Build→Build Solution* (klávesová zkratka *Ctrl+Shift+B*). Tím se vytvořil soubor s příponou *.dll*, který je uložen ve složce daného projektu. Tento soubor se pak vloží do projektu aplikace psané v jazyce C#.

Při popisu algoritmů bude místo označení silniční sítě používáno označení graf a místo označení silnice označení hrana. Nejprve je ale popsána struktura uložení a přístup k datům.

2.1. Struktura uložení a přístup k datům

V této kapitole jsou popsána vstupní data a navrhnuta vhodná struktura uložení těchto dat v paměti při výpočtu. Dále jsou zde popsány ostatní datové struktury použité pro výpočet a jejich vlastnosti.

2.1.1. Vstupní data

Soubor *ulice.dat* (obr. 2.3) slouží k výpočtu nejkratší Hamiltonovské cesty v jazyce C. Obsahuje pole silnic o velikosti 231930. Každá silnice je vyjádřena jejím identifikačním číslem (id silnice), identifikačním číslem počáteční křižovatky, indexem koncové křižovatky a velikostí silnice udávané v metrech. Index koncové křižovatky je taková hodnota silnice, kde je tato křižovatka uložena jako počáteční. Toto pole je setříděno podle identifikačního čísla počáteční křižovatky od nejmenší hodnoty k největší. Data jsou v souboru uložena v binárním tvaru.

id silnice

id počáteční křižovatky

index koncové křižovatky

velikost silnice

```
...
711234673,60167665,231387,18.190100
705020173,60172327,133802,5.074690
570337745,61364995,32922,30.248500
574633790,70341319,84745,95.765300
574631776,59117941 7,84647,84.828400
...
```

Obr. 2.3: Část dat ze souboru ulice.dat v textovém zobrazení.

2.1.2. Struktura uložení grafu

Pro uložení grafu jsou použity dvě struktury. Jedna je určena pro uložení vstupních dat do paměti a k výpočtu vzdálenosti jednotlivých hran od zvoleného počátečního vrcholu. Tyto vlastnosti při dalších výpočtech nebudou stačit, jelikož bude nutné ukládat grafy, které budou obsahovat hrany se zjištěnou cestou. Druhá struktura má podobné vlastnosti, ale navíc umí uložit zjištěnou cestu.

První struktura *hrana1* slouží k popisu jedné silnice. Pomocí této struktury jsou ukládána data ze souboru *ulice.dat* do paměti počítače. To zajišťuje funkce *initGraf*. Z toho plyne, že se struktura skládá z identifikačního čísla (dále jen id) silnice (*idObjektu*), id počáteční křižovatky (*idKrizPocatek*), indexu koncové křižovatky (*idKrizKonec*) a velikosti silnice (*delkaHrany*). Navíc obsahuje proměnou *vzdalenost*, do níž se při výpočtu modifikovaným Dijkstrovým algoritmem ukládá aktuální vzdálenost od zvoleného počátečního vrcholu.

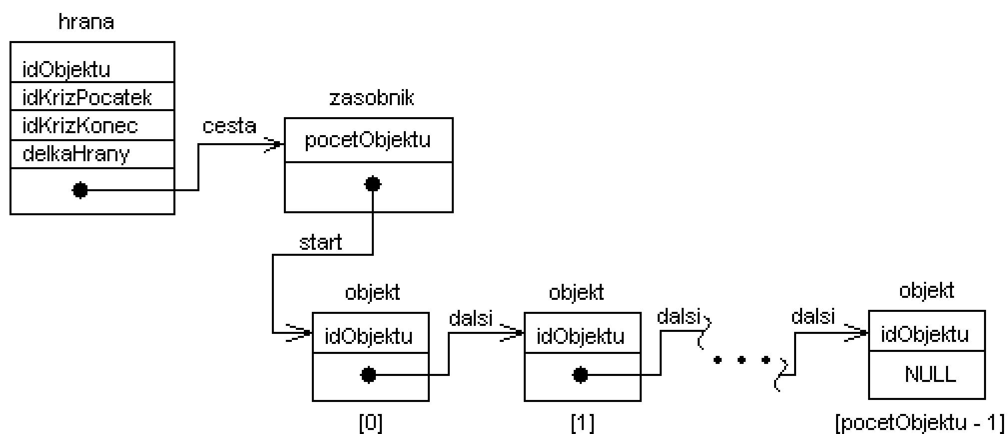
```
typedef struct hrana1
{
      int idObjektu;
      int idKrizPocatek;
      int idKrizKonec;
      double delkaHrany;
      double vzdalenost;
}HRANA1;
```

HRANA1 *initGraf()

Tato funkce se snaží otevřít soubor *ulice.dat*. Pokud jej neotevře, ukončí program, pokud jej otevře, alokuje paměť pro načtení dat. Při neúspěšné alokaci se ukončí program. Data se ukládají do pole struktur *HRANA1*. Funkce vrací ukazatel na toto pole. Hodnota *vzdalenost* se nastaví na "nekonečno". "Nekonečno" je v tomto případě dostatečně velké číslo, které je zaručeně větší než součet všech velikostí hran grafu. Velikost nekonečna je nastavena na hodnotu 100000000.

Druhá struktura je nazvána *hrana*, oproti struktuře *hrana1* místo proměnné *vzdalenost* obsahuje zjištěnou cestu. Ta je vyjádřena jako ukazatel (*cesta*) na datovou strukturu zásobník (*zasobnik*). Zásobník je popsán v kapitole 2.1.3. Uložení struktury *hrana* v paměti počítače je na obr.2.4.

```
typedef struct hrana
{
      int idObjektu;
      int idKrizPocatek;
      int idKrizKonec;
      double delkaHrany;
      ZASOBNIK *cesta;
}HRANA;
```



Obr. 2.4: Uložení struktury hrana v paměti počítače.

Funkce *initGraf()* je definována v modulu *nejkratsiCesta.c* a struktury *hrana* a *hrana1* jsou uloženy v modulu *nejkratsiCesta.h*.

2.1.3. Datová struktura zásobník

Zásobník je taková posloupnost prvků, ke které se přistupuje pouze na jednom jejím konci, který se nazývá vrchol. Obsah zásobníku lze měnit pouze na tomto konci pomocí operace přidání prvku na vrchol zásobníku a operace odebrání prvku z vrcholu zásobníku. Bude použit pro uložení nalezené cesty (viz. kapitola 2.2.2.) a k ukládání množiny bodů při hledání minimální kostry (viz. kapitola 2.4.1.).

K vytvoření zásobníku je použita struktura, v níž je uložen počet hran (*pocetObjektu*) uložených v cestě a ukazatel (*start*) na první hranu cesty (*objekt*). Struktura *objekt* obsahuje id uložené hrany (*idObjektu*) a ukazatel (*dalsi*) na další strukturu *objekt* (viz. obr.2.4).

```
typedef struct zasobnik
{
      int pocetObjektu;
      OBJEKT *start;
}ZASOBNIK;


typedef struct objekt
{
      int idObjektu;
      struct objekt *dalsi;
}OBJEKT;
```

Mimo operace vytvoření zásobníku *initZasobnik*, je implementována operace přidání nového prvku na vrchol zásobníku *pridejDoZasobniku*. Jelikož není potřeba odebírat prvek z vrcholu zásobníku, není vytvořena operace odebrání prvku z vrcholu zásobníku.

ZASOBNIK *initZasobnik ()

Vytvoří nový zásobník a vrátí na něj ukazatel. V případě neúspěšné alokace paměti, funkce ukončí program. Hodnota počet objektů je nastavena na *0* a ukazatel na vrchol je nastaven na *NULL*.

OBJEKT *vytvorObjekt (int idObj)

Vytvoří nový objekt a vrátí na něj ukazatel. Jestli se nepodaří alokovat paměť, funkce ukončí program, jinak se ohodnotí *idObjektu* hodnotou atributu této funkce *idObj* a ukazatel na další prvek je nastaven na *NULL*.

void pridejDoZasobniku (ZASOBNIK *z, int idObjektu)

Funkce vytvoří nový prvek pomocí funkce *vytvorObjekt* a vloží do něj hodnotu *idObjektu*. Pak tento prvek vloží na vrchol zásobníku (*z*) a inkrementuje hodnotu *pocetObjektu* v zásobníku.

void uvolniZasobnik (ZASOBNIK *z)

Tato procedura uvolní všechny prvky zásobníku (z) a pak samotný zásobník z paměti.

void uvolniObjekt (OBJEKT *o)

Procedura uvolní z paměti strukturu *objekt*, na kterou ukazuje ukazatel *o*.

Při hledání minimální kostry je potřeba vytvořit nové operace se zásobníkem. Zda daný zásobník obsahuje určitý vrchol, nám zjistí funkce *mnozinaObsahujeVrchol*. Ještě je potřeba vytvořit jednu speciální funkci pro nalezení takové hrany daného grafu, jejíž počáteční vrchol leží v množině uložené v zásobníku a jejíž koncový vrchol v této množině neleží. Funkce se nazývá *minCenaZMnA*.

int mnozinaObsahujeVrchol (ZASOBNIK *mn, int v)

Zjišťuje, zda množina vrcholů v zásobníku *mn* obsahuje vrchol *v*. Jestli ano, vrací jeho hodnotu, jinak vrací *0*.

HRANA *minCenaZMnA (ZASOBNIK *mn_A, HRANA *g)

Nalezne hranu v grafu *g* s nejmenší délkou (cenou), která vychází z nějakého vrcholu z množiny *mn_A*, ale v této množině nekončí. Funkce alokuje paměť pro strukturu *hrana*, kam se uloží nalezená hrana a vrátí ukazatel na tuto hranu. Pokud se nepodaří alokovat paměť, funkce ukončí program.

Všechny tyto funkce a procedury jsou implementovány v modulu *nejkratsiCesta.c* a struktury *zasobnik* a *objekt* jsou uloženy v souboru *nejkratsiCesta.h*.

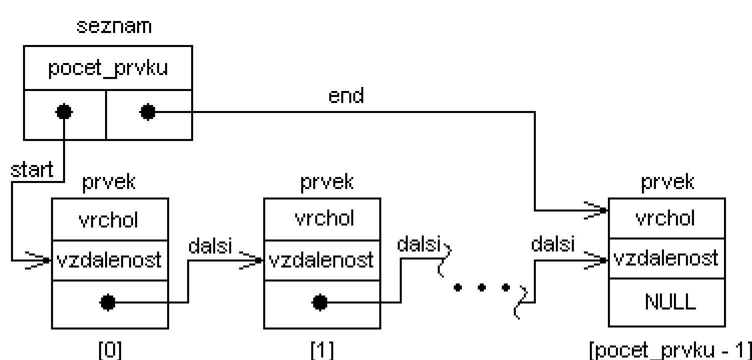
2.1.4. Datová struktura seznam

Seznam je posloupnost prvků, u které lze aplikovat přidání a odebraní prvku na libovolném místě této posloupnosti. Pro tento výpočet jsou tyto vlastnosti poněkud upraveny. Datová struktura seznam je použita ve výpočtu modifikovaným Dijkstrovým algoritmem (viz. kapitola 2.2.1.) a ve výpočtu metodou nejvzdálenějšího vkládání (viz. kapitola 2.4.3.). Seznam ve výpočtu modifikovaným Dijkstrovým algoritmem se liší od seznamu ve výpočtu metodou nejvzdálenějšího vkládání.

Seznam ve výpočtu modifikovaným Dijkstrovým algoritmem je vytvořen pomocí struktury, která obsahuje hodnotu počet prvků v seznamu a ukazatel na počáteční a na koncový prvek seznamu. Prvek je struktura, která obsahuje vrchol, vzdálenost a ukazatel na další prvek. Co vrchol a vzdálenost znamenají, je vysvětleno v kapitole 2.2.1.

```
typedef struct seznam
{
  PRVEK *start,*end;
  int pocet_prvku;
}SEZNAM;

typedef struct prvek
{
  int vrchol;
  double vzdalenost;
  struct prvek *dalsi;
}PRVEK;
```



Obr. 2.5: Uložení struktury seznam v paměti počítače.

Ze seznamu je nutné vybírat prvek s nejmenší vzdáleností, proto operace přidání prvku do seznamu je provedena tak, aby byl seznam setříděn podle vzdálenosti od nejmenší po největší. Ukazatel na počátek tedy ukazuje na prvek s nejnižší hodnotou vzdálenosti ze všech prvků

v seznamu a ukazatel na konec ukazuje na prvek s největší hodnotou vzdálenosti ze všech prvků v seznamu. Operace odebrání prvku ze seznamu je pak provedena odebráním počátečního prvku. Pro výpočet je nutné přidat operaci, která zjistí, zda seznam obsahuje daný vrchol nebo ne. Pokud seznam daný vrchol obsahuje, ale jeho vzdálenost je větší než vzdálenost daného vrcholu, odstraní jej ze seznamu a oznámí, že seznam daný vrchol neobsahuje.

SEZNAM *initSeznam ()

Vytvoří nový seznam a vrátí na něj ukazatel. Pokud se nepodaří alokovat paměť, funkce ukončí program. Počáteční i koncový ukazatel je nastaven na *NULL* a počet prvků je nula.

void pridejDoSeznamu(SEZNAM *s,int vrchol,double vzdalenost)

Funkce vytvoří nový prvek s hodnotami *vrchol* a *vzdalenost* a tento prvek umístí do seznamu (*s*) podle vzdálenosti. Pokud seznam neobsahuje žádný prvek, vloží se nový prvek do tohoto seznamu a oba ukazatele se na něj nasměrují. Jinak se nový prvek vloží na takové místo seznamu, aby platilo, že předchozí vrchol má vzdálenost menší a následující vrchol má vzdálenost větší nebo rovnu než vzdálenost vloženého prvku.

PRVEK *vytvorPrvek(int vrchol, double vzdalenost)

Vytvoří nový prvek a vrátí na něj ukazatel. Pokud se nepodaří alokovat paměť, funkce ukončí program. Hodnoty vrchol a vzdálenost se nastaví podle hodnot argumentů této funkce. Ukazatel na další prvek se nastaví na *NULL*.

PRVEK *odeberZeSeznamu(SEZNAM *s)

Tato funkce odebere ze seznamu (*s*) počáteční prvek a vrátí na něj ukazatel. Pokud seznam neobsahuje žádný prvek, funkce vrací *NULL*.

int obsahujeVrchol (SEZNAM *s, int vrchol, double vzdalenost)

Zjistí, zda daný seznam (*s*) obsahuje vrchol zadaný jako argument funkce. Pokud ano, vrací funkce hodnotu tohoto vrcholu. Pokud jej seznam obsahuje, ale jeho vzdálenost v seznamu je větší než vzdálenost daného vrcholu, funkce jej odstraní ze seznamu a vrátí *0*. V případě, že seznam vůbec daný vrchol neobsahuje, vrací funkce *0*.

void uvolniSeznam (SEZNAM *s)

Tato procedura uvolní všechny prvky seznamu (*s*) a pak samotný seznam z paměti.

Všechny tyto funkce a procedury jsou implementovány v modulu *nejkratsiCesta.c* a struktury *seznam* a *prvek* jsou uloženy v souboru *nejkratsiCesta.h*.

Seznam ve výpočtu metodou nejvzdálenějšího vkládání je podobný seznamu použitém ve výpočtu modifikovaným Dijkstrovým algoritmem. Je také vytvořen pomocí struktury (*szm*), která obsahuje hodnotu počet prvků v seznamu a ukazatel na počáteční a na koncový prvek seznamu. Prvek (*prvk*) je struktura, která obsahuje vrchol, ukazatel na strukturu *HRANA* a ukazatel na další prvek. Co vrchol a ukazatel na strukturu *HRANA* znamenají, je vysvětleno v kapitole 2.4.3.

```
typedef struct prvk
{
	int vrchol;
	HRANA *min1;
	struct prvk *dalsi;
}PRVK;

typedef struct szm
{
	PRVK *start,*end;
	int pocet_prvku;
}SZM;
```

Tento seznam není potřeba vytvářet setříděný, proto operace přidání prvku do seznamu je provedena tak, že se každý nový prvek přidá na počátek tohoto seznamu. Operace odebrání prvku ze seznamu odebere takový prvek, který má ze všech prvků v seznamu největší délku hrany.

SZM *initSzm ()

Vytvoří nový seznam a vrátí na něj ukazatel. Pokud se nepodaří alokovat paměť, funkce ukončí program. Počáteční i koncový ukazatel je nastaven na *NULL* a počet prvků je nula.

PRVK *vytvorPrvk(int vrchol, HRANA *min1)

Vytvoří nový prvek a vrátí na něj ukazatel. Pokud se nepodaří alokovat paměť, funkce ukončí program. Hodnota *vrchol* a ukazatel *min1* se nastaví podle hodnot argumentů této funkce. Ukazatel na další prvek se nastaví na *NULL*.

void pridejDoSzmu(SZM *s,int vrchol, HRANA *min1)

Funkce vytvoří nový prvek s hodnotami *vrchol* a *min1* a tento prvek umístí do seznamu (*s*) na začátek.

PRVK *odeberZeSzmu(SZM *s)

Tato funkce odebere ze seznamu (*s*) takový prvek, který má největší velikost hrany ze všech prvků v seznamu, a vrátí na něj ukazatel. Pokud seznam neobsahuje žádný prvek, funkce vrací *NULL*.

Tyto funkce a procedury jsou implementovány v modulu *nejvzdVkl.c* a struktury *szm* a *prvk* jsou uloženy v souboru *nejvzdVkl.h*.

2.2. Nejkratší cesta

Nejdůležitější částí z hlediska rychlosti výpočtu je výpočet nejkratší cesty. Bude totiž potřeba pro nalezení úplného grafu (viz. kapitola 2.3). Nejkratší cesta se bude hledat mezi dvěma body grafu.

Pro výpočet nejmenší vzdálenosti a pro zjištění nejkratší cesty mezi dvěma body jsou použity dvě funkce. První počítá nejmenší vzdálenost mezi dvěma body grafu pomocí tzv. modifikovaného Dijkstrova algoritmu. Druhá funkce vyhledá pomocí algoritmu prohledávání s návratem (angl. backtracking) nejkratší cestu. Prohledávání s návratem je rekurzivní metoda.

K nalezení nejkratší cesty modifikovaným Dijkstrovým algoritmem je využito prohledávání do šířky a trojúhelníková nerovnost. Pro každý vrchol x se bude uchovávat hodnota $U(x)$, která bude rovna délce nejkratší dosud nalezené cesty z počátečního vrcholu r do vrcholu x. Jestliže se žádná cesta z r do x dosud nenašla, bude $U(x) = \infty$. Kdyby hodnoty U byly skutečnými vzdálenostmi vrcholů grafu od vrcholu r, musela by pro každou hranu grafu (x, y) platit trojúhelníková nerovnost

$$U(y) \leq U(x) + a(x, y), \qquad (2.6)$$

kde *a(x, y)* je velikost hrany *(x, y)*. Jestliže neplatí, znamená to, že *U(y)* není délkou nejkratší cesty z *r* do *y*, protože přes vrchol *x* vede do *y* cesta kratší. Proto v takovém případě se hodnota *U(y)* sníží provedením příkazu

$$U(y) = U(x) + a(x, y)\ . \qquad (2.7)$$

U prohledávání s návratem bude prohledáván strom řešení (získaný pomocí modifikovaného Dijkstrova algoritmu) do hloubky. Aby toto prohledávání bylo efektivnější, budou se prohledávat pouze větve, u nichž je naděje, že mohou být prodlouženy na hledané řešení. Je k tomu využito rekurzivní volání funkce *backtracking*.

2.2.1. Modifikovaný Dijkstrův algoritmus

Vstupem pro tento algoritmus je silniční síť (graf) popsaná pomocí silnic (hran). Tyto silnice jsou ohodnoceny svými velikostmi, tudíž ohodnocení těchto hran není záporné. Dále je znám výchozí vrchol, odkud se hledá nejkratší cesta, a koncový vrchol, kde tato cesta končí. Výstupem tohoto algoritmu jsou pro každý koncový vrchol hran grafu na cestě z počátečního do koncového vrcholu hodnoty vzdálenosti koncových vrcholů hran grafu od počátku. V tomto algoritmu je použita datová struktura seznam (viz. kapitola 2.1.4.), která bude ukládat potřebné vrcholy a jejich vzdálenost od počátku. Její výhodou je, že bude uchovávat tento seznam setříděný podle této vzdálenosti od nejmenší po největší. Pokud počáteční vrchol hrany není v seznamu, tato hrana splňuje nerovnost (2.6).

Před vlastním výpočtem se musí nastavit vzdálenosti všech vrcholů od počátku na nekonečno a do seznamu vložit počáteční vrchol, jehož vzdálenost je nastavena na nulu. Nyní se může popsat modifikovaný Dijkstrův algoritmus:

1. Dokud není seznam prázdný, odebere se ze seznamu první vrchol a označí se *x*. Pokud je tento odebraný vrchol zároveň koncovým, výpočet končí.
2. Pro všechny hrany (*e*) vycházející z vrcholu *x* se provede krok 3 a po zpracování všech hran se pokračuje podle kroku 1.
3. Koncový vrchol hrany *e* se označí *y*. Jestliže platí $U(x) + a(x, y) < U(y)$, pak se provede $U(y) = U(x) + a(x, y)$ a navíc, pokud vrchol *y* neleží v seznamu, vloží se do něj podle vzdálenosti.

Tímto způsobem se ohodnotili potřebné vzdálenosti vrcholů od počátku a tím se získal tzv. strom řešení.

Časový odhad tohoto algoritmu je $O(n^2+m)$, kde n je počet vrcholů a m je počet hran grafu. Tento odhad je řád funkce, reprezentující závislost času na počtu vrcholů a hran. Použitím programátorských triků pro výběr minima (použitím datové struktury halda, anglicky heap) lze časový odhad zlepšit na $O((m+n)\log n)$.

2.2.2. Prohledávání s návratem

Vstupem pro tuto rekurzivní funkci je výchozí vrchol *r*, koncový vrchol *c*, graf a strom řešení. Jako výstup je velikost cesty a posloupnost hran, které tvoří hledanou cestu z výchozího do koncového vrcholu. Pro uložení výstupních hodnot je použita strukturu *hrana*, která je schopna ukládat cestu i její velikost. Cesta se ukládá do zásobníku, který je popsán v kapitole 2.1.3. Pomocná proměnná je index koncového vrcholu v grafu (*iKV*) a index hrany s minimální vzdáleností od počátku (*iMin*).

Jelikož ve vstupním grafu je uložen koncový vrchol každé hrany jako index (viz. 2.1.1.), je použita funkce *najdiIndex*, která je schopna vyhledat index hrany v grafu pro zvolený vrchol.

Popis algoritmu:

1. Nalezne se index koncového vrcholu *c* v grafu a uloží se do *iKV*. Dále *iMin* je nastaveno na hodnotu *iKV*.
2. Nastavení ukončovací podmínky. Pokud počáteční vrchol hrany s indexem *iKV* se nerovná výchozímu vrcholu *r*, pokračuje se krokem 3, jinak výpočet končí.
3. Pro všechny hrany grafu se provede: jestliže koncový vrchol *i*-té hrany je roven *iKV* a zároveň vzdálenost této hrany od počátku je menší než vzdálenost hrany s indexem *iMin*, nastaví se hodnota *iMin* na *i*.
4. Přidání hrany s indexem *iMin* do zásobníku.
5. Zavolání rekurzivní metody *backtracking*, ale místo koncového vrcholu *c* se vloží počáteční vrchol hrany s indexem *iMin*.
6. Nastavení vzdálenosti koncového vrcholu hrany s indexem *iMin* od počátku do proměnné *delkaHrany* ve struktuře *hrana*

2.2.3. Popis výpočtu v jazyce C

K výpočtu nejkratší cesty a vzdálenosti slouží tyto procedury a funkce. Zdrojové kódy jsou uvedeny v souborech *nejkratsiCesta.c* a *nejkratsiCesta.h*.

int najdiIndex(int vrchol, HRANA1 *h)

Vyhledá pozici argumentu *vrchol* v poli hran grafu (*h*). Je to pozice, kde se tento vrchol nachází jako počáteční vrchol hrany.

void initVzdalenosti (HRANA1 *h)

Procedura nastaví všechny vzdálenosti koncových vrcholů hran grafu od počátku na "nekonečno". "Nekonečno" je v tomto případě dostatečně velké číslo, které je zaručeně větší než součet všech velikostí hran grafu. Velikost nekonečna je nastavena na hodnotu 100000000.

void modifDijkstraDvaVrcholy (int pocatecniVrchol, int koncovyVrchol, HRANA1 *hrana)

Tato procedura slouží k nastavení vzdálenosti vrcholů grafu od počátečního vrcholu, dokud nenalezne vzdálenost od počátečního vrcholu ke koncovému. Využívá k tomu modifikovaný Dijkstrův algoritmus.

První argument procedury je identifikační číslo počátečního vrcholu, druhý je id koncového vrcholu a třetí je ukazatel na graf. Jako výstup procedury je nastavení vzdáleností jednotlivých hran grafu.

void backtracking (int iterace,int pocatecniVrchol,int koncovyVrchol,int idObj,HRANA *v, HRANA1 *h)

Tato rekurzivní procedura pomocí prohledávání do hloubky zjišťuje nejkratší cestu mezi počátečním a koncovým vrcholem grafu. Prohledávání začíná od koncového vrcholu dokud se nenalezne počáteční vrchol. Jako výstup je nová hrana (*v*), která obsahuje id této hrany, id počátečního a koncového vrcholu, zjištěnou cestu a vypočítanou délku této cesty.

První argument slouží k inicializaci 5. argumentu *v*. Jelikož se tato procedura volá rekurzivně dokud není splněna ukončovací podmínka a argument *v* je třeba inicializovat pouze jednou, při prvním volání této procedury je hodnota *iterace* nastavena na *0* a v proceduře je vložena podmínka: *inicializuj* když je *iterace == 0*. Při dalším volání této procedury je iterace inkrementována. Druhý a třetí argument je id počátečního a koncového

vrcholu. Čtvrtý argument je id nově vzniklé hrany (*v*). Pátý argument je výstup této procedury, je to ukazatel na novou hranu. Z důvodu rekurze je předáván odkazem. Posledním argumentem je ukazatel na graf.

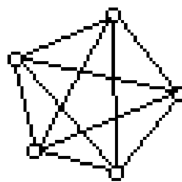
HRANA nejkratsiCesta (int pocVrchol, int koncVrchol, int idObj, HRANA1 *h)

Funkce pomocí dvou předchozích procedur vypočítá nejkratší vzdálenost a nalezne nejkratší cestu mezi počátečním a koncovým vrcholem grafu. Výsledek je vrácen jako jedna nová hrana, v případě špatné alokace paměti pro novou hranu funkce vrací 1.

První a druhý argument je počáteční a koncový vrchol hledané cesty, třetí argument je id nové hrany a poslední je ukazatel na graf.

2.3. Úplný graf

Úplný graf je takový prostý neorientovaný graf bez cyklů, jehož každé dva různé vrcholy jsou spojeny hranou (viz. obr.2.8). Výhodou tohoto grafu je existence hamiltonovské kružnice a tím i řešení problému obchodního cestujícího. Pro další výpočet je zapotřebí vytvořit úplný orientovaný graf, který se získá tak, že z každé neorientované hrany se vytvoří dvě orientované hrany opačného směru. Tento graf také zaručuje existenci hamiltonovské kružnice.



Obr. 2.8: Úplný graf.

2.3.1. Postup výpočtu

Vstupem pro tento výpočet je pole hodnot vrcholů *poleId* a velikost tohoto pole *pocetVrcholu*. Počet vrcholů *n* by měl být minimálně tři. To je zaručeno v aplikaci, která předává data pro výpočet v jazyce C. Výstupem je pro daný počet vrcholů úplný orientovaný

graf uložený pomocí pole struktur *hrana*. Velikost tohoto pole je dvojnásobek hodnoty, kterou vrací funkce *zjistiPocetHran*. Tato funkce počítá počet hran *p* grafu podle vztahu:

$$p = (n-1)+(n-2)+...+1 = n(n-1)/2, \qquad (2.9)$$

Pro zadané pole vrcholů o velikosti *n* se musí najít nejkratší cesta mezi každými dvěma různými vrcholy tohoto pole. Postup nalezení úplného grafu:

1. Čtení grafu (*h*) ze souboru *ulice.dat* pomocí funkce *initGraf*.
2. Pro všechny vrcholy z *poleId* s indexem *i* z intervalu (*i = 0, 1, ..., n-2*) se provede krok 3.
3. Pro všechny vrcholy z *poleId* s indexem *j* z intervalu (*j = i+1, i+2, ..., n-1*) se provede krok 4.
4. Pomocí funkce *nejkratsiCesta* se vyhledá hrana nového grafu a uloží se pomocí struktury *hrana*. Jako počáteční vrchol (*pocVrchol*) se použije vrchol z *poleId* s indexem *i* a jako koncový (*koncVrchol*) použijeme vrchol z *poleId* s indexem *j*. Nalezne se hrana opačného směru.

Aby se hrana opačného směru v kroku 4 nemusela znovu počítat funkcí *nejkratsiCesta*, překopíruje se původní hrana do této hrany, ale zamění se počáteční vrchol za koncový a koncový za počáteční. Změní se *idObjektu*. Ukazatel *cesta* a hodnota *delkaHrany* zůstane stejná.

2.3.2. Popis výpočtu v jazyce C

K nalezení úplného grafu slouží tyto procedury a funkce. Zdrojové kódy jsou uvedeny v souborech *uplnyGraf.c* a *uplnyGraf.h*.

int zjistiPocetHran (int n)

Počítá počet hran úplného grafu a vrací výsledek. Jediným argumentem je počet vrcholů úplného grafu. Funkce počítá podle vzorce (2.9).

HRANA *uplnyGraf (int *poleId, int pocetVrcholu)

Funkce vyhledá úplný orientovaný podgraf vstupního grafu a vrátí na něj ukazatel. První argument je ukazatel na pole id křižovatek, což je pole vrcholů hledaného úplného grafu. Druhý argument nás informuje o počtu vrcholů v tomto poli.

Nejprve se inicializuje vstupní graf (funkce *initGraf()*), spočítá se podle počtu vrcholů počet hran (funkce *zjistiPocetHran (int n)*) a podle počtu hran se alokuje paměť pro nový úplný graf. Jelikož máme orientovaný úplný graf, počet hran je dvojnásobný. Pokud je alokace neúspěšná, funkce vrací *1*. Dále se spočítají všechny hrany nového úplného grafu. Funkce vrací ukazatel na nový úplný graf, který je srovnaný podle id počátečního vrcholu.

2.4. Metody řešení TSP

V této kapitole jsou popsány všechny vypracované metody pro řešení TSP. Jsou to tyto tři algoritmy:

- 1-strom a metoda penalizace
- metoda postupného vkládání vrcholu
- metoda nejvzdálenějšího vkládání vrcholu

První metoda se snaží najít optimální řešení. Další dvě metody patří mezi tzv. heuristické algoritmy, které si nekladou za cíl nalézt přesné řešení, ale pouze nějaké vhodné přiblížení v použitelném čase.

2.4.1. 1-strom a metoda penalizace

Toto je jedna z variant řešení problému obchodního cestujícího. K řešení využívá tzv. nejlevnější 1-strom a metodu penalizace vrcholů.

Nejlevnější 1-strom je nejlevnější faktor H daného grafu G, který má stejný počet vrcholů a hran a vrchol 1 v něm má stupeň $d_H(1) = 2$. Jinak řečeno, nejlevnější 1-strom daného grafu o n vrcholech se dostane tak, že se vezme podgraf indukovaný množinou vrcholů $\{2,\dots,n\}$, najde se nejlevnější kostra tohoto podgrafu a přidá se k ní vrchol 1 spolu s nejlevnějšími

dvěma hranami z tohoto vrcholu. Nejlevnější 1-strom tedy obsahuje přesně jednu kružnici a tato kružnice prochází vrcholem 1. Nejlevnější kostra grafu *G* je taková kostra, která má nejmenší cenu, tj. nejmenší součet ohodnocení hran, mezi všemi kostrami grafu *G*.

Penalizace vrcholů spočívá v tom, že se každému vrcholu *x* grafu *G* přiřadí číslo $p(x)$, tzv. *penále*. Každé hraně grafu, která spojuje vrcholy *i* a *j*, se nyní změní délka z hodnoty $a(i, j)$ na hodnotu $a(i, j)+p(i)+p(j)$. Penále se zpravidla odvozuje ze stupňů vrcholů v grafu. Má-li vrchol vysoký stupeň, dostane kladné penále, tím se prodraží hrany z jeho okolí a je naděje, že se stupeň sníží. Má-li naopak vrchol stupeň 1, dostane záporné penále, ceny hran v jeho okolí klesnou a stupeň vrcholu se může zvětšit. Nejjednodušší pravidlo radí zvětšovat dosavadní penále vrcholu *v* o hodnotu $k(d(v) - 2)$, kde konstanta *k* nemá být příliš velká, může být i rovna jedné. V tomto výpočtu je použita konstanta $k = 10$. K penalizaci vrcholů je použita funkce *penalizace*.

K nalezení nejlevnějšího 1-stromu je použita funkce *jedna_strom*. Vstupem pro tuto funkci je souvislý graf *G* a ohodnocení jeho hran. Výstupem je nejlevnější 1-strom *faktorL*. Pomocná proměnná je zásobník (viz. kapitola 2.1.3.) vrcholů *v_mnA*, do kterého jsou vkládány vrcholy, jež jsou v grafu *faktorL* již spojeny hranou. Postup nalezení 1-stromu:

1. Vytvoření podgrafu *g_copy* indukovaného množinou vrcholů $\{2,\dots,n\}$.
2. Inicializace diskrétního grafu *faktorL* s množinou vrcholů $V(G)$.
3. Inicializace prázdné množiny vrcholů *v_mnA* a vložení do této množiny libovolného vrcholu z množiny $\{2,\dots,n\}$.
4. Dokud není *faktorL* stromem (funkce *jeStromem*), provede se krok 5.
5. Do grafu *faktorL* se vloží (funkce *pridejHranuDoKostry*) taková hrana grafu *g_copy*, která má nejmenší velikost z hran, které začínají v množině *v_mnA* a zároveň které v této množině nekončí (*minCenaZMnA*). Koncový vrchol této hrany se vloží do množiny *v_mnA* (*pridejDoZasobniku*).
6. Do grafu *faktorL* se přidá 1. vrchol a dvě nejkratší hrany vedoucí z tohoto vrcholu.

K hledání nejkratší hamiltonovké kružnice pomocí 1-stromu a metody penalizace slouží funkce *hK_1_strom_metPen*. Vstupem je úplný graf *G*. Výstupem je podgraf grafu *G*, který je nejkratší hamiltonvskou kružnicí. Tato metoda nezaručuje nalezení této kružnice, proto po určitém počtu cyklů vrátí dosud nalezené řešení. Nyní se může popsat postup:

1. Vytvoření kopie *g_copy* grafu *G*, se kterou se bude dále počítat, aby se neztratili původní délky hran.
2. Pomocí funkce *jedna_strom* se nalezne nejlevnější 1-strom v grafu *g_copy* a označí se *kostra*.
3. Dokud *kostra* není kružnice (každý vrchol je 2. stupně), provede se krok 4.
4. Provede se penalizace vrcholů (*penalizace*) v grafu *g_copy* a nalezne se pro něj nejlevnější 1-strom. Pokud krok 4 má více jak 250000 iterací, první hrana grafu *kostra* se označí identifikačním číslem (idObjektu) -2 a ukončí se tento cyklus.
5. Nastavení původních tj. nepenalizovaných délek hran v grafu *kostra*.

K tomuto výpočtu slouží tyto procedury a funkce. Zdrojové kódy jsou uvedeny v souborech *1-strom_metPen.c* a *1-strom_metPen.h*.

HRANA *jedna_strom (HRANA *g, int pocet_cest, int *poleId, int pocet_vrcholu)

Funkce vytvoří 1-strom z úplného grafu o *n* vrcholech a vrátí na něj ukazatel. Využívá k tomu postup hledání minimální kostry.

První argument je ukazatel na úplný graf, ve kterém se hledá 1-strom, druhý argument informuje o počtu hran úplného grafu, třetí je ukazatel na pole id křižovatek (vrcholy úplného grafu) a čtvrtý informuje o počtu vrcholů v tomto poli.

Funkce si nejprve vytvoří úplný graf bez prvního vrcholu (poleId[0]) a bez všech hran, které v tomto vrcholu začínají nebo končí. V tomto upraveném úplném grafu nalezne minimální kostru a pak k této kostře vloží odebraný první vrchol a dvě nejmenší hrany vycházející z tohoto vrcholu.

void penalizace (HRANA *strom, HRANA *g, int pocet_vrcholu, int pocet_cest,int *poleId)

Tato procedura spočítá pro všechny vrcholy minimální kostry (*strom*) penalizaci a ke každé hraně v grafu přičte penalizaci krajních vrcholů. Pokud se nepodaří alokovat paměť pro pomocné proměnné, ukončí se program.

HRANA *hK_1_strom_metPen (HRANA *g, int pocet_cest, int pocet_vrcholu, int *poleId)

Tato funkce najde nejkratší hamiltonovskou kružnici procházející vrcholy *poleId* v grafu *g* pomoci 1-stromu a metody penalizace. V této funkci se vytváří kopie grafu *g* a při neúspěšné alokaci paměti pro tuto kopii vrací funkce 1.

HRANA *jedna_strom_metPen (int *poleId, int pocetVrcholu)

Tato funkce pro dané pole vrcholů vypočítá úplný graf a předá jej jako parametr funkci *hK_1_strom_metPen*. Vrací nejkratší hamiltonovskou kružnici.

int jeStromem (HRANA *fL, int pocet_vrcholu)

Zjistí, zda graf *fL* je stromem. Pokud ano, vrací 1, jinak vrací 0.

void pridejHranuDoKostry (HRANA *h, HRANA *k)

Překopíruje hranu *h* do grafu *k* za poslední uloženou hranu.

int jeKruznice (HRANA *h, int pocet_vrcholu, int *poleId)

Kontroluje, zda jsou všechny vrcholy grafu *h* stupně 2. Pokud ano, vrací 1, pokud ne, vrací 0.

int zjistiIndexVrcholu (int *poleId, int pocetVrcholu, int vrchol)

Vrátí index vrcholu *vrchol* v poli vrcholů *poleId*. Pokud jej nenalezne, vrací -1.

2.4.2. Metoda postupného vkládání vrcholu

Další variantou řešení problému obchodního cestujícího je metoda postupného vkládání vrcholů, která patří mezi heuristické algoritmy.

Začne se s triviální kružnicí v nějakém vrcholu a do této kružnice se budou postupně zařazovat další vrcholy, dokud se nenalezne kružnice hamiltonovská. K zařazení do kružnice se vybírá libovolný vrchol, který se vloží do kružnice mezi sousední dva vrcholy tak, aby se kružnice prodloužila co nejméně.

Vstupem je graf *uG*. K nalezení triviální kružnice je použita funkce *trivialKruznice*, která nalezne v grafu *uG* tři hrany tvořící triviální kružnici a nasměruje na ně ukazatele. Dále je použita datová struktura seznam *s*, v níž se nacházejí vrcholy grafu, které neleží v kružnici. Tento seznam je stejný jako seznam použitý ve výpočtu modifikovaným Dijkstrovým algoritmem. Výstupem je podgraf grafu *uG*, který je nejkratší hamiltonvskou kružnicí.

Postup výpočtu:

1. Vytvoření triviální kružnice *p_kr* a seznamu *s*, který obsahuje vrcholy grafu *uG* a neobsahuje vrcholy triviální kružnice.
2. Dokud seznam není prázdný, provede se krok 3.
3. Odebere se vrchol ze seznamu a najdou se takové dvě hrany spojující tento vrchol s dvěma sousedními vrcholy v kružnici *p_kr* tak, aby se kružnice prodloužila co nejméně. Odebere se hrana spojující tyto sousední vrcholy z *p_kr* a přidají se právě nalezené dvě hrany.

K tomuto výpočtu slouží tyto procedury a funkce. Zdrojové kódy jsou uvedeny v souborech *postupVkl.c* a *postupVkl.h*.

HRANA **trivialKruznice (HRANA *uG, int pocetVrcholu)

Funkce nalezne triviální kružnici (kružnice o 3 vrcholech a 3 hranách) v úplném grafu (*uG*) a vrací ukazatel na ukazatele na jednotlivé hrany teto kružnice.

HRANA *najdiHranu (HRANA *uG, int idPoc, int idKon)

Funkce nalezne hranu v grafu *uG* o daném počátečním (*idPoc*) a koncovém (*idKon*) vrcholu a vrátí její adresu.

HRANA *hamiltonKruznicePV (HRANA *uG, int pocetCest, int pocetVrcholu, int *poleId)

Funkce nalezne nejkratší hamiltonovskou kružnici v úplném grafu (*uG*) metodou postupného vkládání a vrátí na ni ukazatel.

HRANA *postupVkl (int *poleId, int pocetVrcholu)

Tato funkce pro dané pole vrcholů *poleId* vypočítá úplný graf a předá jej jako parametr funkci *hamiltonKruznicePV*. Vrací nejkratší hamiltonovskou kružnici.

2.4.3. Metoda nejvzdálenějšího vkládání vrcholu

Dalším heuristickým algoritmem pro řešení problému obchodního cestujícího je metoda nejvzdálenějšího vkládání vrcholů. Je podobná metodě postupného vkládání. Liší se způsobem výběru vrcholu.

Začne se s triviální kružnicí v nějakém vrcholu a do této kružnice se budou postupně zařazovat další vrcholy, dokud se nenalezne kružnice hamiltonovská. K zařazení do kružnice se vybírá vrchol, který je od kružnice nejvíce vzdálen (odtud název metody). Přesněji, existuje-li kružnice *K*, která prochází vrcholy $v_1,\dots,v_k$, pak se vybere vrchol *v*, pro který je výraz $min\{a(v, v_i) \mid i = 1,\dots,k\}$ největší. Vrchol *v* se pak zařadí do kružnice mezi takové dva sousední vrcholy tak, aby se tím kružnice prodloužila co nejméně.

Vstupem je graf *uG*. K nalezení triviální kružnice je použita funkce *trivialKruznice*, která nalezne v grafu *uG* tři hrany tvořící triviální kružnici a nasměruje na ně ukazatele. Dále je použita datová struktura seznam *s*, v níž se nacházejí vrcholy grafu, které neleží v kružnici. Výstupem je podgraf grafu *uG*, který je nejkratší hamiltonvskou kružnicí.

Postup výpočtu:

1. Vytvoření triviální kružnice *p_kr* a seznamu *s*, který obsahuje vrcholy grafu *uG* a neobsahuje vrcholy triviální kružnice.
2. Dokud seznam není prázdný, provede se krok 3.
3. Pro všechny vrcholy v seznamu se vyhledá nejkratší hrana spojující tento vrchol s kružnicí a uloží se do seznamu k danému vrcholu. Odebere se ze seznamu takový vrchol, který má největší velikost hrany ze všech prvků v seznamu. Najdou se takové dvě hrany spojující tento vrchol s dvěma sousedními vrcholy v kružnici *p_kr* tak, aby se kružnice prodloužila co nejméně. Odebere se hrana spojující tyto sousední vrcholy z *p_kr* a přidají se právě nalezené dvě hrany.

K tomuto výpočtu slouží tyto procedury a funkce. Zdrojové kódy jsou uvedeny v souborech *nejvzdVkl.c* a *nejvzdVkl.h*. Funkce *trivialKruznice* a *najdiHranu* jsou stejné jako funkce se stejným názvem v kapitole 2.5.2, proto zde nejsou popsány.

void hledejMinHranu (HRANA *uG, SZM *s, HRANA **p_kr, int pocetVrcholu)

Pro všechny vrcholy, které se nenachází v kružnici, vyhledá nejkratší hranu spojující tento vrchol s kružnicí a uloží ji do seznamu k danému vrcholu.

HRANA *hamiltonKruzniceNV (HRANA *uG, int pocetCest, int pocetVrcholu, int *poleId)

Funkce nalezne nejkratší hamiltonovskou kružnici v úplném grafu (*uG*) metodou nejvzdálenějšího vkládání a vrátí na ni ukazatel.

HRANA *nejvzdVkl (int *poleId, int pocetVrcholu)

Tato funkce pro dané pole vrcholů *poleId* vypočítá úplný graf a předá jej jako parametr funkci *hamiltonKruznicePV*. Vrací nejkratší hamiltonovskou kružnici.

3. Popis GIS aplikace psané v C# .NET

Tato aplikace slouží k zadání vstupních hodnot pro řešení problému obchodního cestujícího a k prezentaci výsledků. Z obr. 2.1 je vidět, co by měla aplikace poskytnout uživateli. Měla by být navržena tak, aby se uživateli zobrazila silniční síť a on si v ní mohl vybrat křižovatky, kterými chce projet. Dále by měl uživatel mít možnost vybrat si variantu výpočtu. Po spuštění výpočtu by aplikace měla předat potřebná data algoritmům psaným v jazyce C. Po dokončení výpočtu by se měl zobrazit výsledek a měla by se vykreslit vypočítaná cesta.

Aplikace je vypracována použitím Microsoft Visual C# 2008 Expres Edition, který je volně stažitelný z internetových stránek firmy Microsoft (viz. int. adresa [9]). K překladu zdrojových textů je využita platforma .NET.

K vytvoření aplikace je potřeba nejprve vytvořit nový projekt *File→New Project* (klávesová zkratka *Ctrl+Shift+N* nebo ikona). Objeví se nabídka šablon. Pro tuto práci je použita šablona *Windows Forms Application*. Jméno projektu je zvoleno TSP (z angl. traveling salesman problem). V tomto projektu se vygeneroval nový jmenný prostor (angl. namespace) nazvaný také TSP, kde budou definovány všechny třídy, struktury a výčtové typy. Nově vytvořený projekt obsahuje přednastavené soubory. Mimo jiných to jsou *Form1.cs* a *Program.cs*. Soubor *Form1.cs* je přejmenován na *MainForm.cs*.

Aplikace je vytvořena následujícím postupem:

- Vytvoření vzhledu aplikace, nastavení vlastností a událostí ovládacích prvků.
- Vytvoření metod pro načtení vstupních dat ze souborů a struktur pro uložení grafu.
- Vykreslení vstupních dat.
- Uživatelské rozhraní

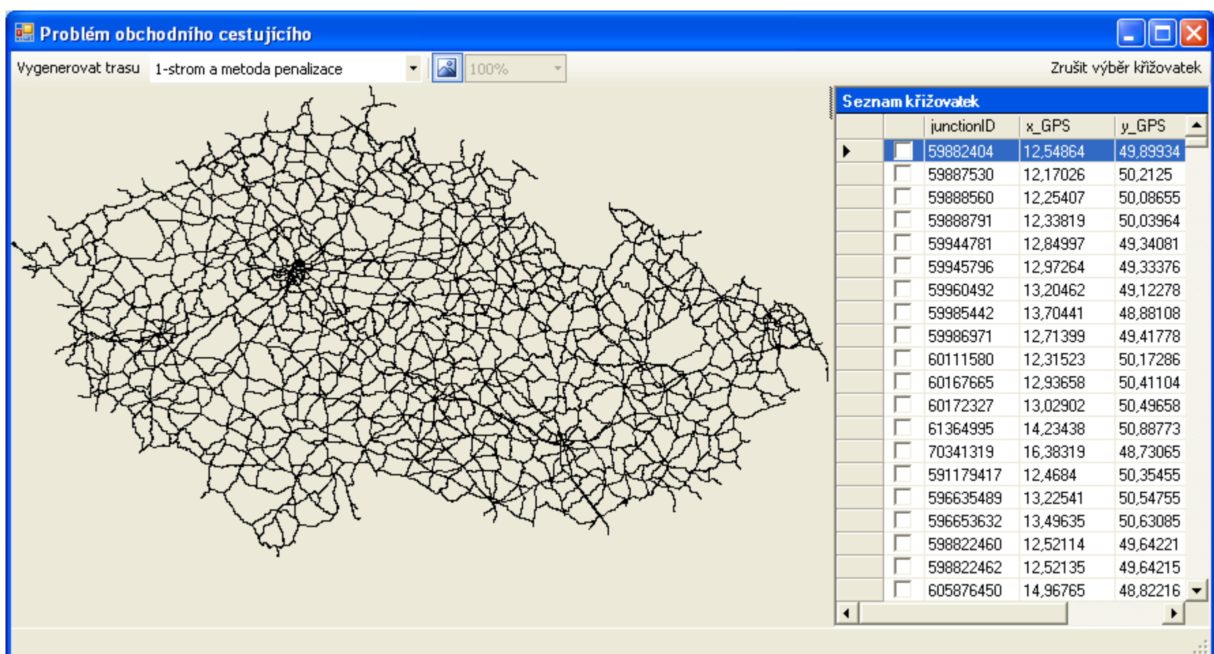
Všechny zdrojové texty GIS aplikace psané v C# .NET jsou uloženy na přiloženém CD v příloze 2. Spustitelný soubor aplikace, dynamické knihovny a všechna vstupní data jsou uložena v příloze 3.

3.1. Vzhled aplikace

Tato kapitola popisuje návrh (angl. design) vzhledu a popis ovládacích prvků. Aplikace se skládá ze dvou oken. Obě okna jsou vytvořena pomocí vizuálního návrháře. Jedno je hlavní (*MainForm*) a druhé (*CalculateForm*) slouží pouze k zobrazení výsledků.

3.1.1. Hlavní okno

Podle požadavků na aplikaci je vytvořeno hlavní okno (*MainForm*), které je schopno zobrazit silniční síť a seznam křižovatek. Aplikace nabízí ovládací prvky pro komunikaci s uživatelem (viz. obr. 3.1).

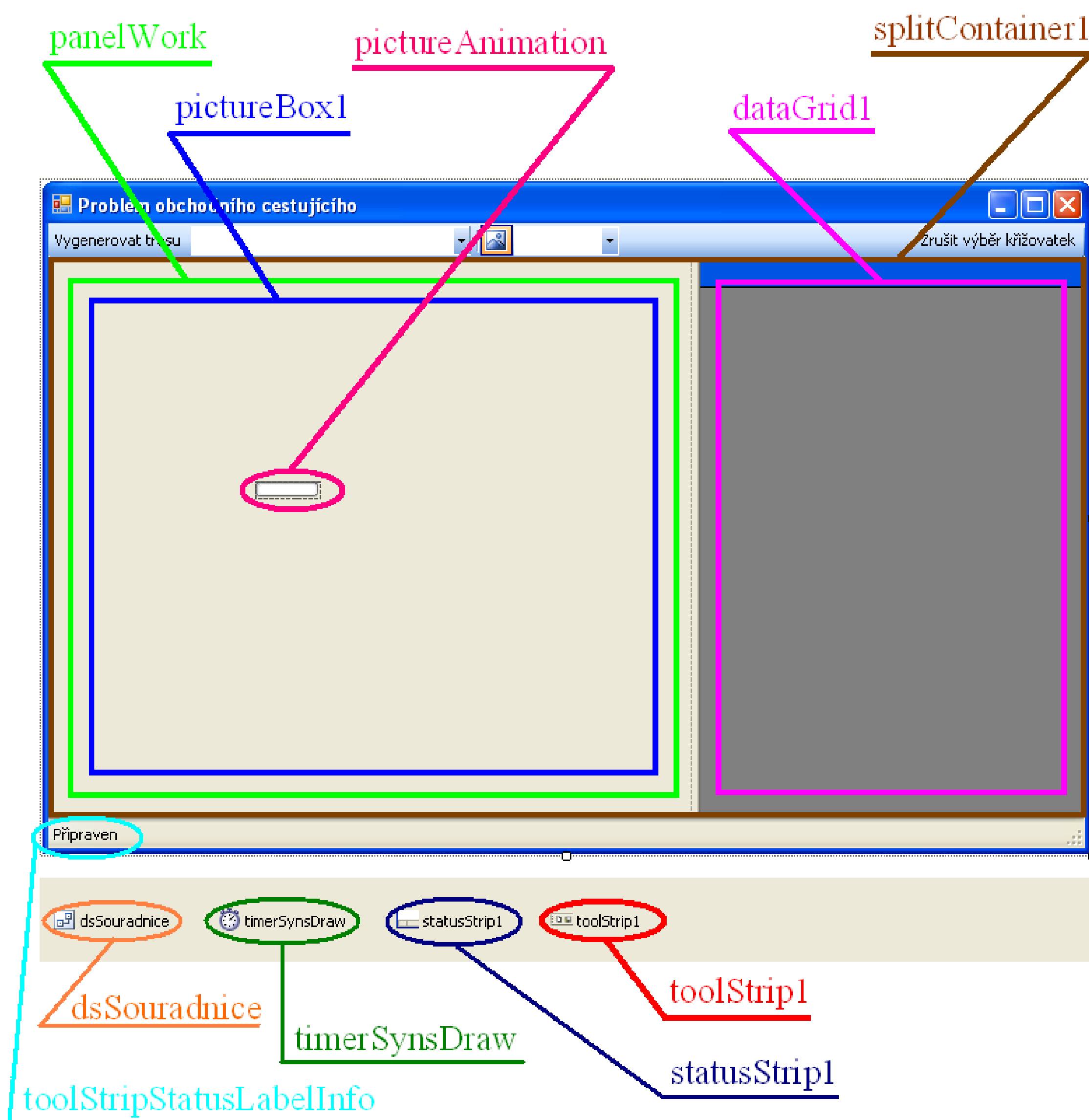


Obr. 3.1: Vzhled aplikace.

Ovládací prvky jsou vkládány z nástrojové sady (angl. Toolbox). Hlavní okno obsahuje tyto ovládací prvky (viz. obr. 3.2):

- ToolStrip - *toolStrip1*
- SplitContainer - *splitContainer1*
- Panel - *panelWork*

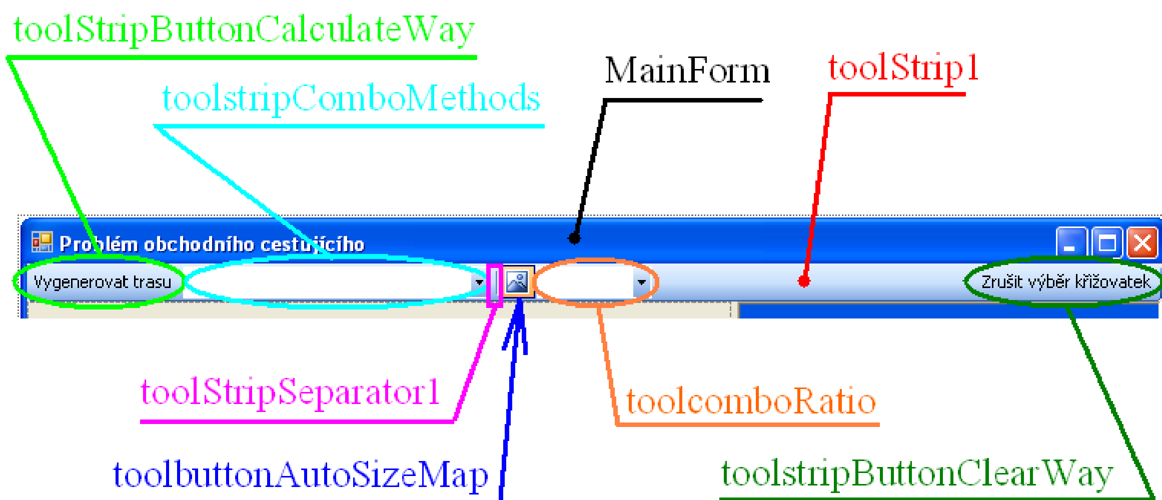
- PictureBox - *pictureBox1*
- PictureBox - *pictureAnimation*
- DataGrid - *dataGrid1*
- DataSet - *dsSouradnice*
- StatusStrip - *statusStrip1*
    - ToolStripStatusLabel - *toolStripStatusLabelInfo*
- Timer - *timerSynsDraw*



Obr. 3.2: Ovládací prvky hlavního okna.

toolStrip1: tento panel nástrojů slouží k zobrazení ovládacích prvků, které budou sloužit ke komunikaci uživatele s aplikací. Obsahuje tyto ovládací prvky (viz. obr. 3.3):

- ToolStripButton - *toolStripButtonCalculateWay*
- ToolStripComboBox - *toolstripComboMethods*
- ToolStripSeparator - *toolStripSeparator1*
- ToolStripButton - *toolbuttonAutoSizeMap*
- ToolStripComboBox - *toolcomboRatio*
- ToolStripButton - *toolstripButtonClearWay*



Obr. 3.3: Ovládací prvky panelu nástrojů *toolStrip1*.

toolStripButtonCalculateWay: po stisknutí tohoto tlačítka se vyvolá událost *toolStripButtonCalculateWay_Click*, v níž se zahájí výpočet.

toolstripComboMethods: slouží k výběru metody výpočtu. Uživatel si může zvolit mezi metodami:

- 1-strom a metoda penalizace
- Metoda postupného vkládání
- Metoda nejvzdálenějšího vkládání
- Nejkratší cesta

toolStripSeparator1: slouží jako oddělovač ovládacích prvků.

toolbuttonAutoSizeMap: silniční síť může být přizpůsobena ovládacímu prvku pictureBox nebo si může zachovávat poměr svých stran. Přepínání mezi těmito stavy zajišťuje toto tlačítko vyvoláním události *toolbuttonAutoSizeMap_Click*.

toolcomboRatio: slouží k nastavení velikosti zvětšení (zoom). Uživatel volí mezi hodnotami: 50%, 100%, 200%, 400%, 800%. Pokud uživatel změní hodnotu zvětšení, vyvolá se událost *toolcomboRatio_SelectedIndexChanged*.

toolstripButtonClearWay: po stisknutí tohoto tlačítka se vyvolá událost *toolstripButtonClearWay_Click*, ve které se zruší výběr křižovatek.

splitContainer1: obsahuje dva panely *splitContainer1.Panel1* a *splitContainer1.Panel2*. Slouží ke změně velikosti ovládacích prvků umístěných v těchto panelech.

panelWork: je „pracovní plocha“, která obsluhuje posuvné lišty, když je velikost mapy větší než velikost této plochy.

pictureBox1: ovládací prvek, ve kterém se zobrazuje silniční síť. Pokaždé, když je nutné překreslit plátno tohoto ovládacího prvku, vyvolá se událost *pictureBox1_Paint*.

pictureAnimation: informuje uživatele o přepočítávání silniční sítě.

dataGrid1: slouží k zobrazení a k výběru jednotlivých křižovatek. Při stisknutí tlačítka myši nad tímto ovládacím prvkem se vyvolá událost *dataGrid1_Click*.

dsSouradnice: sada dat, která slouží k uložení hodnot ze souboru *ulice.lst*. V této sadě je vytvořena tabulka *dataTableList* s pěti sloupci: *datacolmark*, *datacoljunctionID*, *datacolx_gps*, *datacoly_gps*, *datacoly_c*.

statusStrip1: panel, který zobrazuje stav aplikace pomocí *toolStripStatusLabelInfo*.

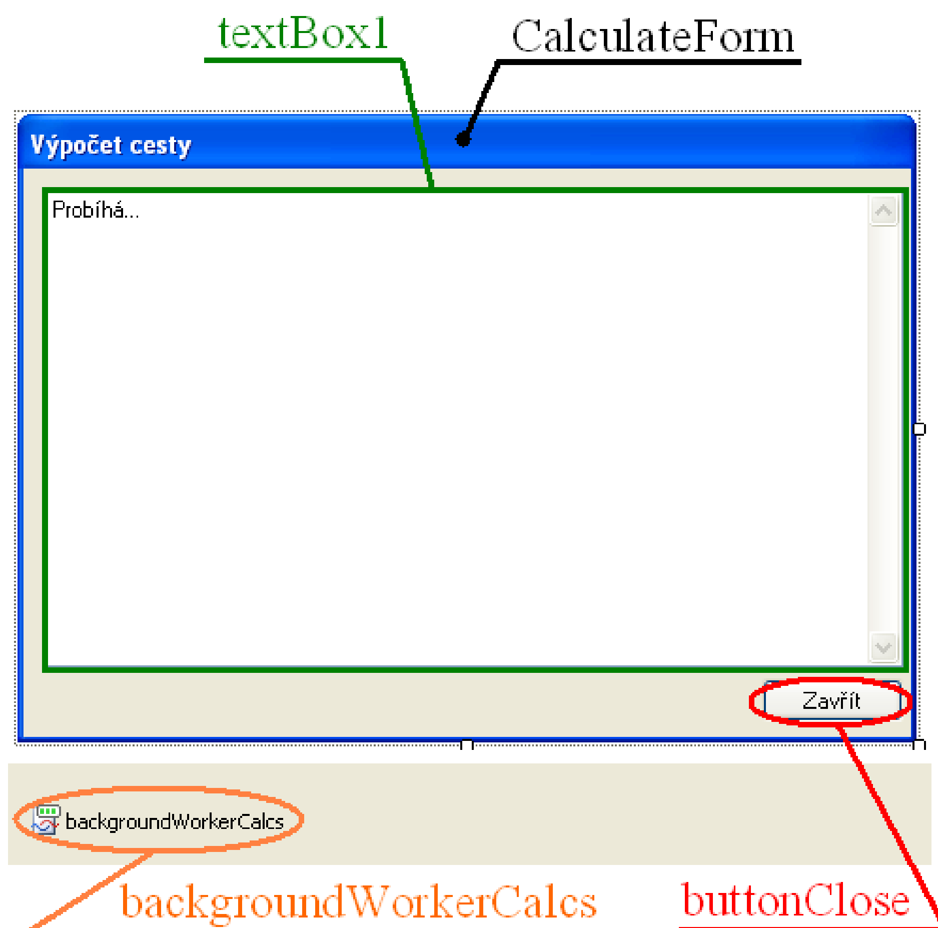
toolStripStatusLabelInfo: štítek informující o stavu aplikace.

timerSynsDraw: slouží pro synchronizaci vykreslení mapy s událostmi windows. Událost *timerSyncDraw_Tick* je vyvolána až po ukončení probíhajících systémových operací v hlavním vlákně aplikace.

Vlastnosti a události těchto ovládacích prvků jsou nastaveny v okně Properties (z angl. vlastnosti), které nabízí vizuální návrhář. Automaticky vygenerovaný zdrojový text je uložen v souboru *MainForm.Designer.cs*. Prázdné metody událostí se vytvořily v souboru *MainForm.cs*.

3.1.2. Okno pro zobrazení výsledků

Okno pro zobrazení výsledků (*CalculateForm*) slouží pouze k zobrazení vypočítaných hodnot a doby výpočtu zvolené varianty.



Obr. 3.4: Ovládací prvky okna pro zobrazení výsledků.

Obsahuje ovládací prvky (viz. obr. 3.4):

- Buton – *buttonClose:* slouží k uzavření okna.
- TextBox – *textBox1:* slouží k zobrazení výsledků.
- BackgroundWorker – *backgroundWorkerCalcs:* slouží ke spuštění výpočtu v samostatném vlákně. Při spuštění výpočtu se vyvolá událost *backgroundWorkerCalcs_DoWork* a při ukončení výpočtu se vyvolá událost *backgroundWorkerCalcs_RunWorkerCompleted*.

Prázdné metody událostí se vytvořily v souboru *CalculateForm.cs* a automaticky vygenerovaný zdrojový text je uložen v souboru *CalculateForm.Designer.cs*.

3.2. Struktura uložení a přístup k datům

V této kapitole jsou popsána vstupní data a jejich načtení do paměti. Pro tato data je navrhnuta vhodná struktura uložení v paměti při výpočtu. Dále je popsáno, jak jsou načteny výsledky výpočtů z externích knihoven.

3.2.1. Vstupní data

Soubor *ulice.lst* (obr. 3.6) slouží k výběru křižovatek podle souřadnic celosvětového polohového systému (z angl. global position system – dále jen GPS). Počet křižovatek v souboru je 112118. Každá křižovatka je v souboru popsána jejím identifikačním číslem, reálnými (GPS) souřadnicemi a transformovanými souřadnicemi. Vztah mezi reálnými a transformovanými souřadnicemi je:

$$\begin{aligned} x_t &= x_r \\ y_t &= \log\left[tg\left(\frac{y_r \cdot \pi}{180 \cdot 2} + \frac{\pi}{4} \right) \cdot \frac{180}{\pi} \right] \end{aligned} \qquad (3.5)$$

kde x_t (y_t) je transformovaná souřadnice x (y) a x_r (y_r) je reálná souřadnice x (y). Křižovatky jsou setříděny podle id křižovatky od nejmenší hodnoty k největší.

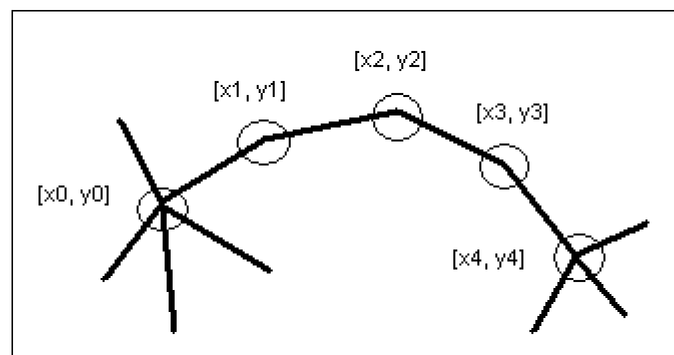
.

id křižovatky | reálná i transformovaná souřadnice x | reálná souřadnice y | transformovaná souřadnice y

...
60172327,13.029020,50.496580,58.684450
61364995,14.234380,50.887730,59.301910
70341319,16.383190,48.730650,55.958586
591179417,12.468400,50.354550,58.461510
596635489,13.225410,50.547550,58.764619
...

Obr. 3.6: Část dat ze souboru ulice.lst v textovém zobrazení.

Data ze souboru *ulice.lst* jsou načtena v metodě *loadGridData()*, která je definovaná v hlavní třídě aplikace *MainForm*, do objektu třídy *DataTable* nazvaném *datatableList*. Třída *DataTable* slouží k reprezentaci jedné tabulky dat v paměti.



Obr. 3.7: Polyline ze čtyř elementů a pěti krajních bodů.

K zobrazení silniční sítě v aplikaci slouží data uložená v souboru *ulice.viw* (obr. 3.8). Zobrazení silnic se provádí pomocí polyline (obr. 3.7), které se skládají z různého počtu elementů. Elementy na sebe navazují. Z toho plyne, že polyline o *n* elementech má *n+1* krajních bodů. Každý element je popsán svými krajními body. Silnice je tedy v souboru

ulice.viw popsána pomocí identifikačního čísla (id silnice), počtu krajních bodů elementů (počet souřadnic) a souřadnic krajních bodů elementů *x* a *y*. Souřadnice jsou transformované. Počet takto popsaných silnic je 115965. Silnice v tomto souboru jsou setříděny podle id od nejmenší hodnoty k největší.

id silnice
počet souřadnic
souřadnice x
souřadnice y

...
565773133,2,14.579440,58.132039,14.579750,58.132491,
565773289,5,14.577750,58.128559,14.577800,58.128996,14.577880,58.129293,14.578030,58.129589,14.578960,58.131259,
565773304,2,14.577720,58.128294,14.577750,58.128559,
...

Obr. 3.8: Část dat ze souboru ulice.viw v textovém zobrazení.

Data ze souboru *ulice.viw* jsou načtena v metodě *loadMapData()*, která je definovaná v hlavní třídě aplikace *MainForm*, do objektu třídy *MapLinesCollection* nazvaném *mapLines*. Tato třída reprezentuje sbírku klíčů a hodnot. Jako klíč je použito id silnice a hodnota je zde objekt *line* třídy *lineObject*. V tomto objektu je uložena jedna polyline.

3.2.2. Struktura uložení grafu

Protože návratová hodnota funkcí pro výpočet TSP v jazyce C je typu ukazatel na strukturu *hrana*, která je definovaná v jazyce C (viz. kap. 2.1.2.), musí se definovat tato struktura i v této aplikaci. Definují se i struktury *zasobnik* a *objekt*. Jedna z proměnných struktury *hrana* totiž ukazuje na *zasobnik*, který obsahuje proměnnou ukazující na *objekt*.

Struktury v aplikaci mají stejné proměnné jako struktury v C, liší se ale v definici. Musí být označeny atributem *structLayout*, aby se mohla předat tato struktura funkci jazyka C. Jako parametr je předán výčet *LayoutKind.Sequential*, který zajistí, že se proměnné nacházejí jedna

za druhou s minimální velikostí balíku (více v [8]). Dále je nastaven modifikátor přístupu *internal*, což značí, že přístup k této struktuře je omezen na aktuální projekt. V těchto strukturách jsou ukazatele, jazyk C# je ale v zabezpečeném kódu nepodporuje, proto je použit modifikátor nezabezpečeného kódu *unsafe*. Aby tento kód mohl být zkompilován, musí se ve vlastnostech projektu (*Project→TSP Properties…→Build*) zaškrtnout políčko *Allow unsafe code*.

```
[StructLayout(LayoutKind.Sequential)]
internal unsafe struct objekt
{
    public int idObjektu;
    public objekt *dalsi;
}

[StructLayout(LayoutKind.Sequential)]
internal unsafe struct zasobnik
{
    public int pocetObjektu;
    public objekt *start;
}

[StructLayout(LayoutKind.Sequential)]
internal unsafe struct hrana
{
    public int idObjektu;
    public int idKrizPocatek;
    public int idKrizKonec;
    public double delkaHrany;
    public zasobnik *cesta;
}
```

Pro práci v aplikaci jsou tyto struktury nevyhovující, protože by se muselo pracovat v nezabezpečeném kódu. Proto jsou vytvořeny třídy *HranaCS* a *KruzniceCS*.

```
internal class HranaCS
{
    public HranaCS(){
        zasobnik = new List<int>();
    }
    public int IdObjektu;
    public int IdKrizPocatek;
    public int IdKrizKonec;
    public double DelkaHrany;
    public List<int> Zasobnik{
        get{
            return zasobnik;
        }
    }

    private List<int> zasobnik;
}
```

První slouží k uložení jedné hrany grafu. Obsahuje soukromou (private) proměnnou *zasobnik*, což je seznam hodnot typu integer, veřejný (public) konstruktor, který inicializuje *zasobnik*, a veřejné proměnné *IdObjektu*, *IdKrizPocatek*, *IdKrizKonec*, *DelkaHrany*. Ještě je implementována vlastnost *Zasobnik*, jež je pouze pro čtení (*get*).

Účelem třídy *KruzniceCS* je uložení celého grafu. Graf je popsán hranami, proto je vytvořena soukromá proměnná *hrany*, což je seznam objektů *HranaCS*. Dále je vytvořen konstruktor, který inicializuje proměnnou *hrany*, a vlastnost *Hrany* pouze pro čtení. Pro zjištění celkové délky nalezené kružnice je zde veřejná proměnná *CelkovaDelka*.

Definice těchto struktur je v souboru *Calcs.cs*.

```
internal class KruzniceCS
{
    public KruzniceCS()
    {
        hrany = new List<HranaCS>();
    }

    internal List<HranaCS> Hrany
    {
        get
        {
            return hrany;
        }
    }
    public double CelkovaDelka;

    private List<HranaCS> hrany;
}
```

3.2.3. Načtení dat z externích knihoven

Externí knihovny jsou vytvořeny v jazyce C. Jsou použity tyto dynamické knihovny *postupVkl.dll*, *nejvzdVkl.dll* a *1-strom_metPen.dll*. Každá knihovna je jedna metoda výpočtu.

Nejprve je vytvořena rodičovská abstraktní (abstract) třída *Calcs*, která definuje abstraktní vlastnosti *MinPoints* a *MaxPoints* a abstraktní metodu *CalculateWay()*. Dále definuje statickou metodu *GetModul()* a metodu s nezabezpečeným kódem *getKruzniceFromHrana*. Abstraktní vlastnosti a metody jsou deklarovány v děděných třídách. Metoda *GetModul()* podle zvoleného typu výpočtu vytváří výpočetní modul a metoda *getKruzniceFromHrana()* transformuje data uložená v nezabezpečeném poli struktur *hrana* do zabezpečené třídy *KruzniceCS*.

Načtení dat z externí knihovny *postupVkl.dll* probíhá v děděné třídě *postupVklCalcs* třídy *Calcs*. Pomocí atributu *DllImport* je importována tato knihovna a její externí funkce *postupVkl()*, která metodou postupného vkládání vypočítá nejkratší hamiltonovskou kružnici a vrátí výsledek jako ukazatel na pole struktur *hrana*. Vlastnost *MinPoints* vrací hodnotu 3 a vlastnost *MaxPoints* hodnotu 12. Tyto hodnoty jsou minimální a maximální počet křižovatek, které si uživatel pro tuto metodu výpočtu může zvolit. Metoda *CalculateWay()* zavolá funkci *postupVkl()* z externí knihovny a výsledek v nezabezpečeném kódu přetransformuje pomocí metody *getKrzniceFromHrana()* do zabezpečené třídy *KruzniceCS*.

Načtení dat z externí knihovny *1-strom_metPen.dll* a *nejvzdVkl.dll* probíhá stejně. Děděná třída se jmenuje *jednaStromCalcs* (příp. *nejvzdVklCalcs*). Zdrojový text je uložen v souboru *Calcs.cs*. Toto uspořádání tříd umožňuje jednoduché rozšíření o další externí knihovny.

3.3. Zobrazení vstupních dat

V této kapitole je popsáno vytvoření GIS. Je zde popsán převod GPS souřadnic do souřadnic vykreslované plochy, vykreslení základní mapy v samostatném vlákně, vykreslení vybraných křižovatek a vygenerované trasy v mapě.

3.3.1. Převod souřadnic

Před samotným zobrazením vstupních dat se musí GPS souřadnice převést do souřadnic vykreslované plochy. To vykonávají metody ve struktuře *MapPoint*, která je definovaná v souboru *MapPoint.cs*. V tomto souboru jsou ještě definovány dvě třídy (*RectangleGps* a *RectangleMap*), které jsou v této struktuře použity.

Třída *RectangleGps* slouží k popisu obdélníkového výřezu v souřadnicích GPS a třída *RectangleMap* slouží k popisu obdélníkového výřezu v mapě. Obě třídy mají soukromé proměnné *x1*, *y1*, *x2*, *y2*, *width* a *height*, kde *x1*, *y1* jsou souřadnice levého horního rohu a *x2*, *y2* jsou souřadnice pravého dolního rohu obdélníkového výřezu. Proměnné *width* a *height* jsou šířka a výška tohoto výřezu. K těmto proměnným jsou vytvořeny vlastnosti *X1*, *Y1*, *X2*, *Y2*, *Width* a *Height*. Soukromé metody *updateWidth()* a *updateHeight()* přepočítávají šířku a výšku obdélníkového výřezu. Veřejná metoda *Set()* nastaví hodnoty proměnných podle vstupních parametrů.

Nyní je možno popsat strukturu *MapPoint*. Je to struktura, která popisuje souřadnice jednoho bodu v GPS. Obsahuje statické konstanty typu desetinného čísla (decimal) *minX*, *minY*, *maxX*, *maxY* a *rateXY*, kde *minX* (12.17026) a *minY* (55.70526) jsou nejmenší souřadnice x a y a *maxX* (18.76673) a *maxY* (59.51573) jsou největší souřadnice x a y ze vstupních dat GPS. Hodnota *rateXY* je poměr mezi velikostmi stran. Je dána vztahem:

$$rateXY = \frac{\max X - \min X}{\max Y - \min Y}. \tag{3.9}$$

Souřadnice bodu jsou uloženy v proměnných *x* a *y*. Mají vlastnosti *X* a *Y*. Dále se zde vyskytuje instance třídy *RectangleGps* označená *rectGps* a instance třídy *RectangleMap* označená *rectMap*. Statická metoda *SetRectangles* provádí nastavení výřezu v mapě. Převod souřadnice x provádí metoda *GetXCoord*, která vrací hodnotu danou vztahem:

$$xCoord = (\mathrm{x - rectGps.X1})\,\frac{\mathrm{rectMap.Width}}{\mathrm{rectGps.Width}}. \tag{3.10}$$

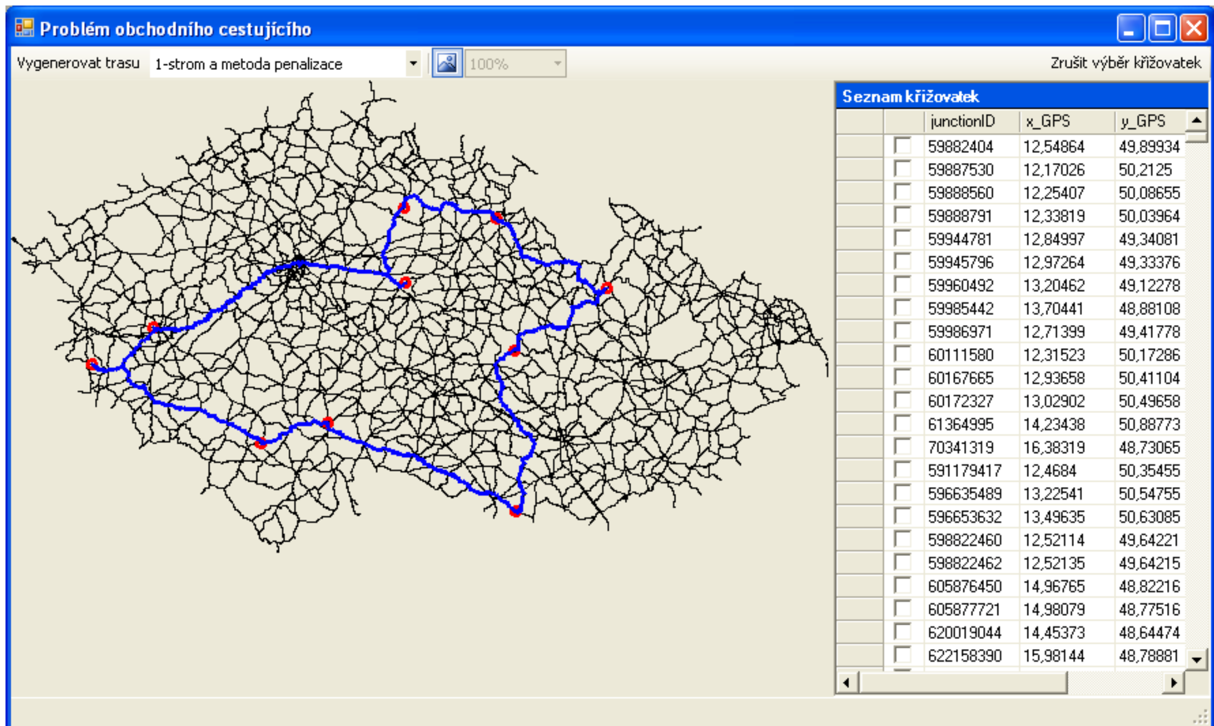
Převod souřadnice y provádí metoda *GetYCoord*, která vrací hodnotu danou vztahem:

$$yCoord = (1 - (\mathrm{y - rectGps.Y1}))\,\frac{\mathrm{rectMap.Height}}{\mathrm{rectGps.Height}}. \tag{3.11}$$

3.3.2. Vykreslení mapy, vygenerované cesty a vybraných křižovatek

Mapa silnic se vykresluje pomocí bitové mapy (*Bitmap*) do ovládacího prvku *pictureBox1*. Vygenerovaná cesta a vybrané křižovatky se vykreslují, když je vyvolaná událost *pictureBox1_Paint*, na plátno ovládacího prvku *pictureBox1*.

Základní třída pro vykreslování nějakého objektu se jmenuje *DrawObject*. Je to abstraktní třída a obsahuje abstraktní metodu *Draw()*. Specifická třída pro vykreslení cesty *LineObject* je odděděná od třídy *DrawObject*. Obsahuje seznam bodů, které se v metodě *Draw()* přepočítají do souřadnic vykreslované plochy a mezi kterými se vykreslí čára (viz. obr.3.7). Definice těchto tříd je v souboru *Structs.cs*.



Obr. 3.12: Vzhled aplikace s vykreslenou mapou, vypočítanou cestou a zvolenými křižovatkami.

O vykreslení mapy v samostatném vlákně se stará třída *Painter* popsaná v souboru *Painter.cs*. Veřejná metoda *DrawMap()* provádí spuštění vlákna pro vykreslení mapy a v tomto novém vlákně volá soukromou metodu *doWork()*, která v cyklu asynchronně kreslí všechny silnice (čáry) v mapě. To je provedeno metodou *asyncDrawLine()*, která používá konstrukci *Invoke* pro synchronizaci vláken a která pro instanci třídy *LineObject* (*line*) volá metodu *Draw()*.

Vygenerovaná cesta se vykresluje metodou *drawWay()* (definovaná ve třídě *MainForm*), která pro všechny silnice v nalezené cestě volá metodu *Draw()* z třídy *LineObject*.

K vykreslení vybraných křižovatek je použita metoda *drawSelection()*, ve které se pro každý označený řádek v ovládacím prvku *dataGrid1* volá metoda *drawGridRow()*. Ta uloží do bodu (*point*) třídy *MapPoint* pro daný řádek souřadnice GPS a zavolá metodu *drawMapPoint()*. Tato metoda pomocí metod pro převod GPS souřadnic do souřadnic vykreslované plochy vykreslí kružnici na plátno.

3.4. Uživatelské rozhraní

Pro komunikaci uživatele s aplikací slouží uživatelské rozhraní. Jsou to takové ovládací prvky, pomocí kterých může uživatel ovládat aplikaci. Uživatel má možnost vybrat si křižovatky, variantu výpočtu, zvětšení (zmenšení) mapy a samozřejmě pro zvolené možnosti může spustit výpočet.

3.4.1. Výběr křižovatek

Uživatel aplikace si vybírá několik křižovatek, které se hned zobrazují na mapě (viz. kap. 3.3.2.). Vybírá je pomocí označovacích políček, které mají dva stavy. Tato políčka a potřebné hodnoty křižovatek jsou zobrazeny v ovládacím prvku *dataGrid1*.

V události *OnShown* jsou načtena data pro výběr křižovatek z datové tabulky (*dataTableList*) do ovládacího prvku *dataGrid1* a jsou automaticky zobrazena při načítání aplikace. K ovládání zobrazení různých typů hodnot v tomto ovládacím prvku slouží třídy popsané v souboru *CustomColumnStyles.cs*, který je stažen z internetových stránek [12].

Stisknutím myši nad ovládacím prvkem *dataGrid1* se vyvolá událost *dataGrid1_Click*, ve které se kontroluje, nad jakým řádkem a nad jakým sloupcem se myš stiskla. Pokud se stiskla nad sloupcem pro označování (*mark*), tak v případě neoznačené křižovatky ji označí a přidá do seznamu stisknutých křižovatek a v opačném případě ji odznačí a odebere ze seznamu stisknutých křižovatek. Nakonec tato událost vyvolá překreslení plátna ovládacího prvku *pictureBox1*.

3.4.2. Výběr metody výpočtu a vlastní výpočet

Z důvodu možnosti porovnání výpočetních metod je umožněn výběr v ovládacím prvku *toolstripComboMethods*. Výpočet se spouští stisknutím tlačítka *toolStripButtonCalculateWay*. Oba tyto ovládací prvky se nachází na panelu nástrojů.

Po stisknutí tlačítka pro výpočet se vyvolá událost *toolStripButtonCalculateWay_Click*, v níž se zjistí varianta výpočtu, otevře nový formulář (*CalculateForm*) a spustí výpočet. Pokud není počet označených křižovatek v daném rozmezí (viz. kap. 3.2.3.), výpočet se

neprovede a objeví se varovné hlášení. Výpočet je spuštěn metodou *CalculateWay()*, která je definovaná v třídě *CalculateForm* a která volá metodu *RunWorkerAsync()* ovládacího prvku *BackgroundWorker*. Metoda *RunWorkerAsync()* zajišťuje spuštění výpočtu v jiném vlákně vyvoláním události *backgroundWorkerCalcs_DoWork*, v níž se spustí metoda *CalculateWay()* třídy *Calcs* a změří se doba výpočtu. Po skončení této operace se již v hlavním vlákně vyvolá událost *backgroundWorkerCalcs_RunWorkerCompleted*, která zavolá metodu *fillResults()* a která vyvolá událost pro překreslení cesty na mapě (*WayCalculated*). Metoda *fillResults()* zajistí zobrazení vypočtených a naměřených hodnot v ovládacím prvku *textBox1* nového formuláře.

3.4.3. Zvětšení (zmenšení) mapy

Může nastat situace, že zvolené křižovatky jsou příliš u sebe a nelze vizuálně určit jejich polohu, proto je zde možnost zvětšování nebo zmenšování mapy. Rozměr mapy se může nastavit dvěma způsoby. Buď se nechá přizpůsobit zobrazovací ploše (*pictureBox1*) nebo se stanoví počáteční rozměr a ten se zvětšuje nebo zmenšuje. Přechod mezi těmito dvěma stavy se provádí tlačítkem *toolbuttonAutoSizeMap*. Počáteční stav je přizpůsobení počáteční ploše. V obou zobrazeních se zachovává konstantní poměr mezi stranami (*rateXY*).

První způsob nastavení rozměru mapy porovnává poměr stran zobrazovací plochy s poměrem *rateXY* a podle toho vykreslí mapu. U druhého způsobu je stanoven počáteční rozměr mapy. Šířka je stanovena na 600 bodů (pixel) a výška je dána poměrem a šířkou. Toto je velikost mapy 100%. Při změně zvětšení se velikost mapy přepočítává v daném měřítku. Výpočty obou způsobů jsou provedeny v metodě *setPictureSize()*.

Stisknutí tlačítka *toolbuttonAutoSizeMap* vyvolá událost *toolbuttonAutoSizeMap_Click*, v níž se změní viditelnost ovládacího prvku *toolcomboRatio* v opačnou a zavolá se metoda *setPictureSize()*. Pokud je ovládací prvek *toolcomboRatio* viditelný a změní se hodnota měřítka, vyvolá se událost *toolcomboRatio_SelectedIndexChanged*. Ta vloží do proměnné *zoom* novou hodnotu měřítka a zavolá metodu *setPictureSize*.

4. Závěr

Tato práce měla za cíl implementovat algoritmy pro řešení TSP v jazyce C a vytvořit GIS aplikaci v jazyce C# .NET, která umožní prezentovat výsledky, proto je rozdělena na dvě části.

První část se skládá ze čtyř kapitol, ve kterých se popisuje řešení problému obchodního cestujícího (TSP) v silniční síti. Jednotlivé algoritmy jsou psány v programovacím jazyce C. Zdrojové texty těchto algoritmů jsou použity k vytvoření dynamických knihoven, které jsou použity ve druhé části této práce.

V první kapitole jsou popsána vstupní data, která jsou uložena v souboru *ulice.dat*, struktury uložení grafu, pomocí kterých jsou uloženy vstupní a výstupní grafy, a pomocné datové struktury zásobník a seznam.

Další kapitola popisuje hledání nejkratší cesty mezi dvěma křižovatkami pomocí modifikovaného Dijkstrova algoritmu a prohledávání s návratem. Pro tyto algoritmy jsou vytvořeny funkce a procedury v jazyce C.

Algoritmus pro hledání úplného grafu, který využívá hledání nejkratší cesty, je popsán ve třetí kapitole. Dále jsou zde popsány funkce a procedury sloužící k hledání úplného grafu v jazyce C.

Čtvrtá kapitola popisuje algoritmy pro řešení TSP. Jsou jimi 1-strom a metoda penalizace, metoda postupného vkládání a metoda nejvzdálenějšího vkládání. V první metodě řešení je popsán postup hledání tzv. 1-stromu, který využívá hledání nejlevnější kostry, postup penalizace vrcholů a jsou zde popsány funkce pro tyto algoritmy psané v jazyce C. Ve druhé a ve třetí metodě řešení TSP je popsán postup výpočtu a funkce psané v jazyce C. Obě tyto metody patří mezi heuristické algoritmy.

Existuje více algoritmů pro řešení TSP. Vytvořením některých těchto algoritmů by se mohla tato práce nechat rozšířit. Z hlediska rychlosti výpočtu nejvíce záleží na rychlosti nalezení nejkratší cesty mezi dvěma křižovatkami, proto by se mohl tento postup ještě nechat vylepšit např. tím, že by se nehledala cesta v takovém směru, ve kterém je z nějaké podmínky jisté, že tam hledaná cesta nevede. Metoda 1-strom a metoda penalizace nenalezne vždy řešení. V takovém případě by se mohla využít jiná metoda výpočtu, popř. změnit konstanta k nebo změnit počet iterací.

Druhá část se skládá ze čtyř kapitol, v nichž je popsána GIS aplikace psaná v jazyce C#. Tato aplikace slouží k porovnávání metod řešení TSP. Umožňuje zadání vstupních hodnot pro řešení problému obchodního cestujícího a prezentaci výsledků. K výpočtu jsou použity dynamické knihovny napsané v jazyce C, které jsou popsány v první části této práce.

V první kapitole této části je popsán vzhled aplikace, která k zobrazení používá dvě okna. Dále jsou zde popsány všechny ovládací prvky těchto oken. Na obrázcích je vidět jejich rozložení.

V další kapitole jsou popsána vstupní data a jejich načtení do paměti. Pro tato data je navrhnuta vhodná struktura uložení v paměti při výpočtu. Dále je popsáno, jak jsou načteny výsledky výpočtů z externích knihoven.

Vytvoření GIS je popsáno ve třetí kapitole. Je zde popsán převod GPS souřadnic do souřadnic vykreslované plochy, vykreslení základní mapy v samostatném vlákně, vykreslení vybraných křižovatek a vygenerované trasy v mapě.

V poslední kapitole jsou popsány ovládací prvky, pomocí kterých může uživatel ovlivnit dění v aplikaci. Uživatel má možnost vybrat si křižovatky, variantu výpočtu, zvětšení (zmenšení) mapy a samozřejmě pro zvolené možnosti může spustit výpočet.

Aplikace je schopna splnit všechny požadavky uživatele, které potřebuje k řešení TSP. Je možné, že by aplikace mohla spadnout. To je zapříčiněno tím, že pokud se nepodaří v dynamické knihovně alokovat paměť, je zavolána funkce (*exit(1)*), která ukončuje program. Aby bylo rozhraní uživatelsky příjemnější, může se ještě aplikace vylepšit. Např. by se mohlo nastavit vykreslení mapy ve výřezech, křižovatky by se mohly volit přímo pohybem myši na mapě nebo by se mohlo povolit třídění seznamu křižovatek podle jednotlivých sloupců.

Zdrojové texty v jazyce C, zdrojové texty v jazyce C# a vytvořená aplikace jsou uloženy na přiloženém CD.

Použitá literatura

[1] DEMEL, J.: Grafy a jejich aplikace. Praha, Academia 2002

[2] SHARP, J. – JAGGER, J.: Microsoft visual C# .NET krok za krokem. Brno, Mobil Media a.s. 2002

[3] VĚCHET, V.: Algoritmy a datové struktury – zásobníky, fronty a seznamy. Liberec 2004

[4] HEROUT, P. : Učebnice jazyka C. České Budějovice, KOPP 1994

[5] HEROUT, P. : Učebnice jazyka C – 2. díl České Budějovice, KOPP 2006

[6] ELLER, F. : C# začínáme programovat. Praha, Grada Publishing a.s. 2002

[7] ARCHER, T. : Myslíme v jazyku C#. Praha, Grada Publishung, spol. s r. o. 2002

[8] DRAYTON, P. – ALBAHARI, B. – BEWARD, T.: C# v kostce. Praha, Grada Publishing a.s. 2003

Internetové adresy

[9] http://www.microsoft.com/express/download/

[10] http://cs.wikipedia.org/wiki/Teorie_graf%C5%AF

[11] http://www.dotnetcharting.com/thankyou.aspx

[12] http://www.msdnaa.net/Resources/display.aspx?ResID=2260