Specifikace problému

Zadání

Firma „Pan Zmrzlík, syn a vnukové“ rozváží mrazírenskými vozy zmrzlinu ve střední Evropě (Německo, ČR, Slovensko, Polsko, Maďarsko). Obsluhuje celkem 3000 zmrzlinářství všech velikostí. Všechny zmrzliny jsou na místo určení centrálně rozváženy z jednoho místa v ČR (centrála). Všechny mrazírenské vozy jsou stejného typu a mohou převážet maximálně 10 tun zmrzliny. Firma převáží náklad pouze v kontejnerech, každý kontejner váží i s nákladem 1 tunu, k zákazníkovi se převážejí pouze zcela naplněné kontejnery, na cestě od zákazníka se převáží kontejnery prázdné. Hmotnost kontejneru je zanedbatelná vzhledem k hmotnosti nákladu.

Každé zmrzlinářství je spojeno přímou silnicí maximálně s 500 jinými zmrzlinářství. Vykládání/nakládání každého kontejneru trvá 15 minut. Zmrzlinářství přijímají zmrzlinu každý den v čase 6-18 hod. Z každého zmrzlinářství může přijít maximálně jedna objednávka denně, přičemž možný počet objednaných kontejnerů se pohybuje v rozmezí jedna až pět.

Počet mrazírenských vozů považujte za neomezený. Průměrná rychlost mrazírenského vozu je 70 km/hod, považujte ji za konstantní na všech silnicích. Pokud nestihnete objednávku zmrzliny doručit, tak ji zamítnete. Přijatou objednávku musíte doručit. Počet kontejnerů objednávaných ze zmrzlinářství odpovídá rovnoměrnému rozdělení. Databázi zmrzlinářství si stáhněte nebo vygenerujte (při generování používejte rozumné číselné hodnoty). Jízdu mrazírenského vozu můžete kdykoli přeplánovat. Mrazírenský vůz se po vyložení všech kontejnerů vrací nejkratší cestou zpět do centrály.

Náklady na přepravu jsou následující:

- fixní náklady – 5 Kč/km
- přeprava zmrzliny = 1 Kč/km/kontejner
- vyložení plného kontejneru = 100,- Kč
- naložení prázdného kontejneru = zdarma
- čekání – 150 Kč/hod

Základním požadavkem firmy „Pan Zmrzlík, syn a vnukové“ je

- obsloužit všechny zákazníky, pokud je to časově zvládnutelné
- minimalizovat přepravní náklady přepočtené na dopravu jednoho kontejneru

Dodatky

Připravte rozumná vstupní data (zmrzlinářství, vzdálenosti mezi nimi odpovídají rozměrům obsluhovaného území) a uložte je ve vhodném formátu, zvolte a implementujte vhodnou datovou strukturu(y) pro reprezentaci vstupních dat, důsledně zvažujte paměťovou náročnost zvolené struktury a časovou náročnost algoritmů pro následovné vyhledání optimálních cest.

Zvolte vhodné algoritmy a proveďte diskrétní simulaci v rámci pěti dnů (respektujte základní požadavek optimalizace rozvozu): U X zmrzlinářství jsou objednávky známé již v 0:00 hodin prvního dne, v tuto dobu se začíná rozvážet), další požadavky na dovoz kontejnerů přicházejí až do pátého dne 12:00 s exponenciálním pravděpodobnostním rozdělením se střední hodnotou intervalu mezi příchody T = 600 s (simulaci proveďte pro hodnoty X= 50, X=150 a X=300); průběh simulace (všechny důležité hodnoty) zapisujte na obrazovku a do souboru, simulaci umožněte ve vhodných okamžicích přerušit.

Vytvořte prostředí pro snadnou obsluhu programu (menu, ošetření vstupů) - nemusí být grafické umožněte zadání požadavku na dovoz kontejnerů z klávesnice. Umožněte aktuální sledování polohy a stavu nákladu libovolného mrazírenského vozu (např. kde je na cestě, kolik zbývá složit kontejnerů, atd.) a aktuálního stavu obsluhy daného zmrzlinářství.

Proveďte simulaci a vygenerujte následující statistiky (uložte je do vhodných souborů):

- přehled všech objednávek během 5 dnů s časem příchodu objednávky, zmrzlinářstvím, které objednávku poslalo, počtem objednávaných kontejnerů a statutem objednávky (přijata/nepřijata, splněna/nesplněna), na konci uveďte souhrnná čísla (celkový počet objednávek, celkový počet objednaných kontejnerů,…)
- přehled uskutečněných jízd s podrobným rozpisem trasy, počtu přepravovaných kontejnerů a podrobným rozpisem finančních nákladů, na konci uveďte souhrnná čísla
- přehled obsluhy každého zmrzlinářství, tj. počet kontejnerů zmrzlinářstvím objednaných a přehled mrazírenských vozů, která ve zmrzlinářství vykládala/nakládala kontejnery (na konci uveďte souhrnná čísla)

Vytvořte dokumentační komentáře ve zdrojovém textu programu a vygenerujte programovou dokumentaci (Javadoc), vytvořte kvalitní dále rozšiřitelný kód - kontrola dle webové aplikace http://liks.fav.zcu.cz/jj.

Analýza problému

Ze zadání vyplývá, že budeme řešit grafový problém, konkrétně problém okružních jízd. Tento problém lze řešit několika způsoby.

Můžeme vysílat vozy pro jednotlivé objednávky samotné, leckdy poloprázdné. Tuto možnost jsme zavrhli z důvodu nesplnění jednoho ze základních požadavků v zadání – minimalizace ceny dopravy zmrzliny. Další možností by bylo čekat, jestli nepřijde objednávka z města, které již leží na trase některého z vozů čekajících na výjezd. To by byla o něco lepší možnost, ale pravděpodobnost, že objednávka z takového místa přijde než vůz vyjedeme, nám přišla příliš malá. Nakonec jsme zvolili jako ideální Clarke–Wrightův algoritmus pro okružní jízdy, který sdružuje trasy, na kterých se ušetří nejvíce kilometrů.

S volbou algoritmu pro výpočet okružních jízd se pojí i použité datové struktury a potřebná vstupní data.

Vstupní data

Na začátku jsme měli pouze 3001 měst ze střední Evropy, která jsme si stáhli včetně GPS souřadnic. Pomocí jednoduchého programu jsme si vygenerovali cesty mezi městy a to tak, aby do každého města vedlo minimálně 450 cest a cesty mezi městy nebyly delší než 550 km. Maximální počet cest z města se pohyboval kolem 650, což v průměru vedlo na 522 cest z města.

Graf načítáme z textového souboru, kdy načteme nejprve názvy všech měst (a jejich id: 0 – 3000) a poté ke každému městu jeho sousedy. Ti jsou uloženi v textovém souboru v řádce odpovídající městu, pro které sousedy načítáme, a reprezentováni svým id.

Protože Clarke-Wrightův algoritmus často využívá nejkratších cesty pro své výpočty, rozhodli jsme se při spuštění rovnou načítat i matici nejmenších vzdáleností a nejkratších cest. Obě jsme předem spočítali na námi vygenerovaném grafu pomocí Floyd-Warshallova algoritmu. Sice nám tímto vzrostla paměťová spotřeba o další dvě, poměrně velké, matice, ale nemusíme několikrát během programu počítat nejkratší vzdálenosti ani posléze rekonstruovat nejkratší cesty. Dosáhneme tak lepší časové složitosti.

Reprezentace grafu

Graf lze v paměti ukládat prakticky dvěma možnými způsoby. Maticí sousednosti nebo seznamem sousednosti. Protože už musíme mít po celou dobu programu alokované dvě matice 3001x3001 nebyla by matice sousednosti příliš dobrou volbou. Vlastně jsme o ní neuvažovali i z důvodu poměrně řídkého grafu: na 3001x3001 „políček“ matice máme pouze 3001x500 cest.

Pro reprezentaci grafu jsem tedy použili trochu upravený seznam sousednosti. Třída představující graf obsahuje pole jednotlivých vrcholů (měst). Každé město kromě svého názvu a id obsahuje navíc seznam všech svých sousedů. Tento seznam je implementován množinou, ve které je klíčem id sousedního města a hodnotou pak vzdálenost obou měst.

Diskrétní čas

Protože máme simulovat v diskrétním čase, museli jsme implementovat další datovou strukturu, která by měla na starost čas.

Jednotlivé události (výjezd vozu, vyložení zmrzliny, návrat vozu do depa), které mají nastat řadíme do prioritní fronty vzestupně podle jejich času dokončení. Během programu už jen stačí vybírat vždy prvek na vrcholu fronty, který má nejmenší čas dokončení, a globální aktuální čas nastavit na jeho hodnotu. Protože jsou ve frontě prvky seřazeny od nejmenšího po největší nemělo by se stát, že budeme v čase skákat zpět. Čas samozřejmě nastavuje i nově vygenerovaná objednávka.

Algoritmus pro plánování tras

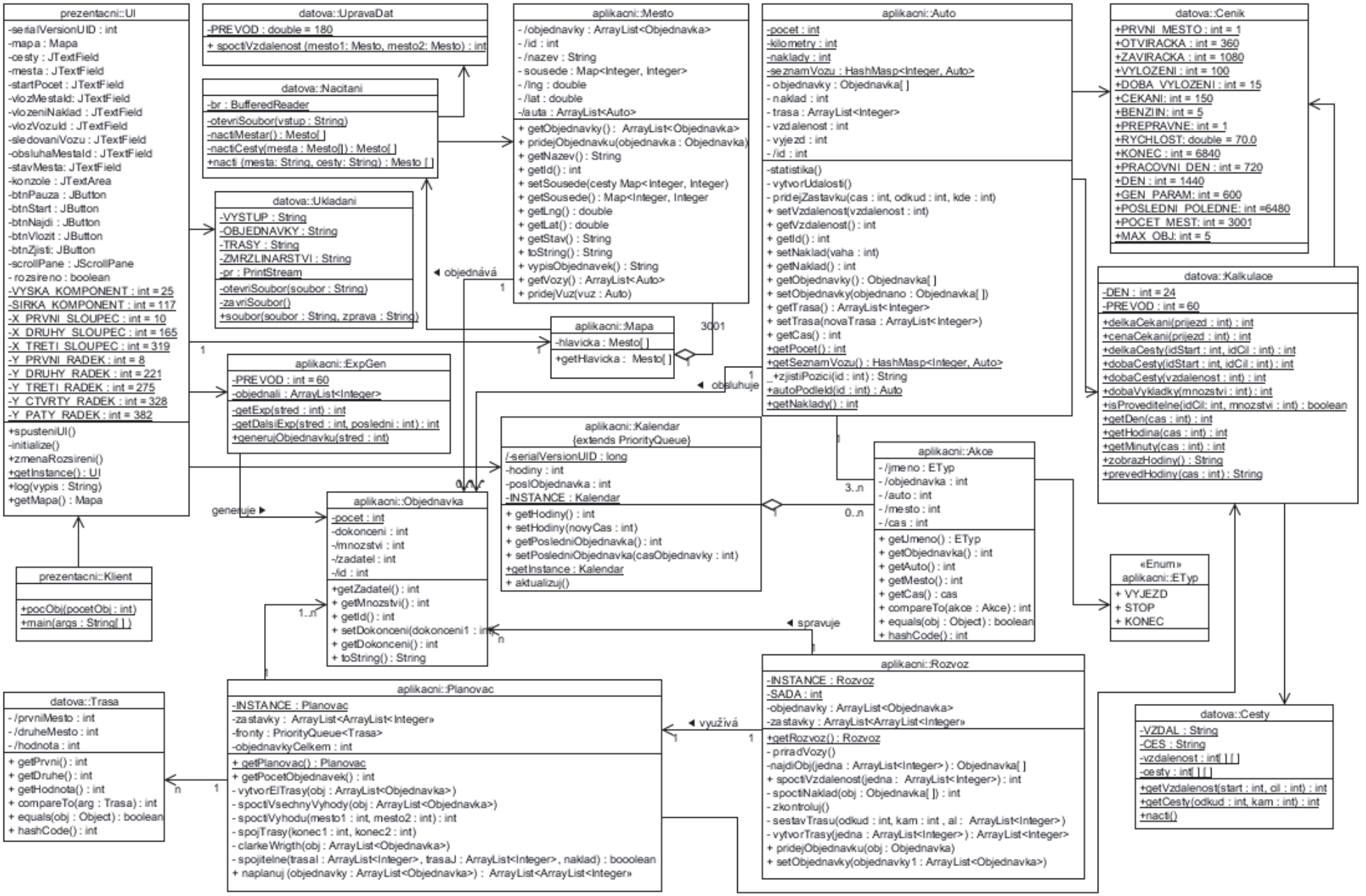
Jak už bylo zmíněno výše, pro plánování tras jednotlivých vozů jsme použili Clarke-Wrightův algoritmus. Jedná se o heuristiku řešící problém okružních jízd.

Podle [1] má algoritmus časovou složitost $O(n^2 \log n)$. Algoritmus spočítá cesty do všech měst a zpět zvlášť. Poté spojuje ty trasy, na kterých se ušetří nejvíce kilometru, ale dodávka nepřekročí maximální náklad jednoho auta. Pseudokód opět viz [1].

Design

Použili jsme třívrstvou architekturu, tj. rozdělení programu na 3 vrstvy: prezentační, aplikační a datovou. Prezentační vrstva komunikuje s uživatelem, aplikační se stará o samotný běh programu a datová obsahuje třídy pro načítání/ukládání dat a knihovní třídy obsahující konstanty.

Detailnější popis na následujícím diagramu tříd:


prezentacni::UI
-serialVersionUID : int
-mapa : Mapa
-cesty : JTextField
-mesta : JTextField
-startPocet : JTextField
-vlozMestaId: JTextField
-vlozeniNaklad : JTextField
-vlozVozuId : JTextField
-sledovaniVozu : JTextField
-obsluhaMestaId : JTextField
-stavMesta: JTextField
-konzole : JTextArea
-btnPauza : JButton
-btnStart : JButton
-btnNajdi : JButton
-btnVlozit : JButton
-btnZjisti: JButton
-scrollPane : JScrollPane
- rozsireno : boolean
-VYSKA_KOMPONENT : int = 25
-SIRKA_KOMPONENT : int = 117
-X_PRVNI_SLOUPEC : int = 10
-X_DRUHY_SLOUPEC : int = 165
-X_TRETI_SLOUPEC : int = 319
-Y_PRVNI_RADEK : int = 8
-Y_DRUHY_RADEK : int = 221
-Y_TRETI_RADEK : int = 275
-Y_CTVRTY_RADEK : int = 328
-Y_PATY_RADEK : int = 382
+spusteniUI()
-initialize()
+zmenaRozsireni()
+getInstance() : UI
+log(vypis : String)
+getMapa() : Mapa
datova::UpravaDat
-PREVOD : double = 180
+ spoctiVzdalenost (mesto1: Mesto, mesto2: Mesto) : int
datova::Nacitani
-br : BufferedReader
-otevriSoubor(vstup : String)
-nactiMestar() : Mesto[ ]
-nactiCesty(mesta : Mesto[]) : Mesto[ ]
+nacti (mesta: String, cesty: String) : Mesto [ ]
datova::Ukladani
-VYSTUP : String
-OBJEDNAVKY : String
-TRASY : String
-ZMRZLINARSTVI : String
-pr : PrintStream
-otevriSoubor(soubor : String)
-zavriSoubor()
+soubor(soubor : String, zprava : String)
aplikacni::Mesto
- /objednavky : ArrayList<Objednavka>
- /id : int
- /nazev : String
- sousede : Map<Integer, Integer>
- /lng : double
- /lat : double
-/auta : ArrayList<Auto>
+ getObjednavky() : ArrayList<Objednavka>
+ pridejObjednavku(objednavka : Objednavka)
+ getNazev() : String
+ getId() : int
+ setSousede(cesty Map< Integer, Integer)
+ getSousede() : Map<Integer, Integer
+ getLng() : double
+ getLat() : double
+ getStav() : String
+ toString() : String
+ vypisObjednavek() : String
+ getVozy() : ArrayList<Auto>
+ pridejVuz(vuz : Auto)
objednává
1
aplikacni::Mapa
-hlavicka : Mesto[ ]
+getHlavicka : Mesto[ ]
3001
1
obsluhuje
aplikacni::ExpGen
-PREVOD : int = 60
-objednali : ArrayList<Integer>
-getExp(stred : int) : int
-getDalsiExp(stred : int, posledni : int) : int
+generujObjednavku(stred : int)
generuje
aplikacni::Auto
-pocet : int
-kilometry : int
-naklady : int
-seznamVozu : HashMasp<Integer, Auto>
- objednavky : Objednavka[ ]
- naklad : int
- trasa : ArrayList<Integer>
- vzdalenost : int
- vyjezd : int
- /id : int
-statistika()
- vytvorUdalosti()
- pridejZastavku(cas : int, odkud : int, kde : int)
+ setVzdalenost(vzdalenost : int)
+ getVzdalenost() : int
+ getId() : int
+ setNaklad(vaha : int)
+ getNaklad() : int
+ getObjednavky() : Objednavka[ ]
+ setObjednavky(objednano : Objednavka[ ])
+ getTrasa() : ArrayList<Integer>
+ setTrasa(novaTrasa : ArrayList<Integer>)
+ getCas() : int
+getPocet() : int
+getSeznamVozu() : HashMasp<Integer, Auto>
_+zjistiPozici(id : int) : String
+autoPodleId(id : int) : Auto
+getNaklady() : int
datova::Cenik
+PRVNI_MESTO : int = 1
+OTVIRACKA : int = 360
+ZAVIRACKA : int = 1080
+VYLOZENI : int = 100
+DOBA_VYLOZENI : int = 15
+CEKANI: int = 150
+BENZIN: int = 5
+PREPRAVNE: int = 1
+RYCHLOST: double = 70.0
+KONEC : int = 6840
+PRACOVNI_DEN : int = 720
+DEN : int = 1440
+GEN_PARAM: int = 600
+POSLEDNI_POLEDNE: int =6480
+POCET_MEST: int = 3001
+MAX_OBJ: int = 5
datova::Kalkulace
-DEN : int = 24
-PREVOD : int = 60
+delkaCekani(prijezd : int) : int
+cenaCekani(prijezd : int) : int
+delkaCesty(idStart : int, idCil : int) : int
+dobaCesty(idStart : int, idCil : int) : int
+dobaCesty(vzdalenost : int) : int
+dobaVykladky(mnozstvi : int) : int
+isProveditelne(idCil: int, mnozstvi : int) : boolean
+getDen(cas : int) : int
+getHodina(cas : int) : int
+getMinuty(cas : int) : int
+zobrazHodiny() : String
+prevedHodiny(cas : int) : String
aplikacni::Kalendar
{extends PriorityQueue}
/-serialVersionUID : long
-hodiny : int
-posliObjednavka : int
-INSTANCE : Kalendar
+ getHodiny() : int
+ setHodiny(novyCas : int)
+ getPosledniObjednavka() : int
+ setPosledniObjednavka(casObjednavky : int)
+getInstance : Kalendar
+ aktualizuj()
3..n
0..n
aplikacni::Akce
- /jmeno : ETyp
- /objednavka : int
- /auto : int
- /mesto : int
- /cas : int
+ getJmeno() : ETyp
+ getObjednavka() : int
+ getAuto() : int
+ getMesto() : int
+ getCas() : cas
+ compareTo(akce : Akce) : int
+ equals(obj : Object) : boolean
+ hashCode() : int
«Enum»
aplikacni::ETyp
+ VYJEZD
+ STOP
+ KONEC
aplikacni::Objednavka
-pocet : int
-dokonceni : int
-/mnozstvi : int
-/zadatel : int
-/id : int
+getZadatel() : int
+ getMnozstvi() : int
+ getId() : int
+ setDokonceni(dokonceni1 : int)
+ getDokonceni() : int
+ toString() : String
1..n
n
spravuje
prezentacni::Klient
+pocObj(pocetObj : int)
+main(args : String[ ])
aplikacni::Rozvoz
-INSTANCE : Rozvoz
-SADA : int
-objednavky : ArrayList<Objednavka>
-zastavky : ArrayList<ArrayList<Integer>>
+getRozvoz() : Rozvoz
- priradVozy()
-najdiObj(jedna : ArrayList<Integer>) : Objednavka[ ]
+ spoctiVzdalenost(jedna : ArrayList<Integer>) : int
- spoctiNaklad(obj : Objednavka[ ]) : int
- zkontroluj()
- sestavTrasu(odkud : int, kam : int , al : ArrayList<Integer>)
- vytvorTrasy(jedna : ArrayList<Integer>) : ArrayList<Integer>
+ pridejObjednavku(obj : Objednavka)
+ setObjednavky(objednavky1 : ArrayList<Objednavka>)
využívá
aplikacni::Planovac
-INSTANCE : Planovac
-zastavky : ArrayList<ArrayList< Integer>>
-fronty : PriorityQueue<Trasa>
-objednavkyCelkem : int
+ getPlanovac() : Planovac
+ getPocetObjednavek() : int
- vytvorElTrasy(obj : ArrayList<Objednavka>)
- spoctiVsechnyVyhody(obj : ArrayList<Objednavka>)
- spoctiVyhodu(mesto1 : int, mesto2 : int) : int
- spojTrasy(konec1 : int, konec2 : int)
- clarkeWrigth(obj : ArrayList<Objednavka>)
- spojitelne(trasaI : ArrayList<Integer>, trasaJ : ArrayList<Integer>, naklad) : booolean
+ naplanuj (objednavky : ArrayList<Objednavka>) : ArrayList<ArrayList<Integer>>
datova::Trasa
- /prvniMesto : int
- /druheMesto : int
- /hodnota : int
+ getPrvni() : int
+ getDruhe() : int
+ getHodnota() : int
+ compareTo(arg : Trasa) : int
+ equals(obj : Object) : boolean
+ hashCode() : int
n
1
datova::Cesty
-VZDAL : String
-CES : String
-vzdalenost : int[ ] [ ]
-cesty : int[ ] [ ]
+getVzdalenost(start : int, cil : int) : int
+getCesty(odkud : int, kam : int) : int
+nacti()

Uživatelská dokumentace

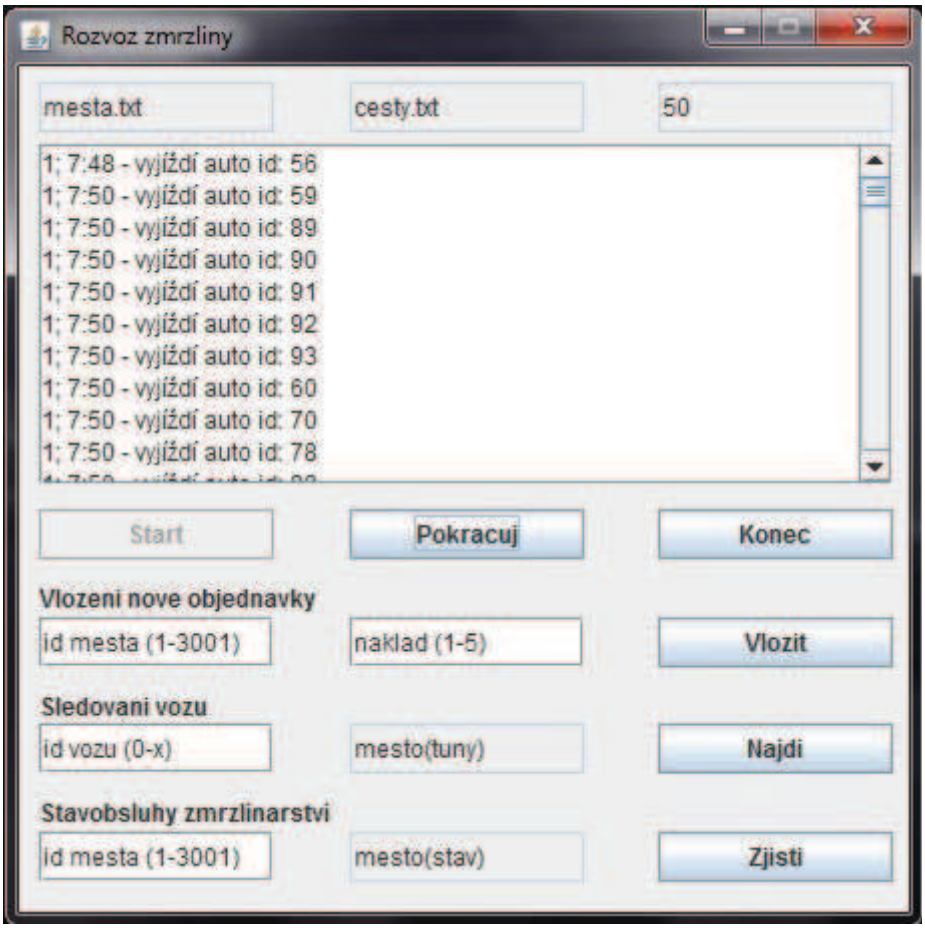
Spuštění programu

Před spuštěním programu zkontrolujte, zda ve složce, kde máte uložený program pro simulaci chodu zmrzlinářství *.jar, se nacházejí i všechny potřebné zdrojové soubory. Jedná se o textové soubory obsahující města a cesty, případě předpočítaných vzdáleností pak ještě soubory s maticí vzdáleností a nejkratších cest.

Soubor *.jar spustíte jednoduše tím, že na něj poklikáte. Pokud ho chcete spustit z příkazové řádky (což není nutné), slouží k tomu příkaz *java -jar jmenosouboru.jar*.

Ovládání programu

Po spuštění programu se zobrazí hlavní menu programu, které můžete vidět na následujícím obrázku. V horní části okna jsou textová pole, kam se zadává cesta ke zdrojovým souborům s grafem. Vedle nich je pole pro nastavení počtu počátečních objednávek.



Vše je nastaveno na implicitní hodnoty, stačí tedy stisknout tlačítko *Start* a začne simulace.

Po načtení mapy se začnou do okna vypisovat jednotlivé události, a to příchod objednávky, její přijetí/zamítnutí, výjezd vozu, vykládání zmrzliny a návrat vozu do depa. Během této

doby je kdykoliv možné simulaci pozastavit stisknutím tlačítka *Pauza*. Znovu stisknutím stejného tlačítka se simulace opět rozběhne.

Při pozastavení programu se zpřístupní textová pole v dolní části obrazovky. Ty slouží k ručnímu nastavení simulace. Můžete přidat objednávku z určitého zmrzlinářství pomocí první volby. Pomocí druhého řádku lze nastavit sledování trasy určitého vozu. Třetí volba slouží ke sledování stavu určitého zmrzlinářství.

Ruční přidání objednávky

Pro přidání objednávky napište do prvního textového pole id města, tj. číslo od 1 do 3000. Do druhého textového pole napište množství objednané zmrzliny, které se smí pohybovat v rozsahu 1-5 tun. Nyní pro vygenerování vaší objednávky stiskněte tlačítko *Vložit*.

Sledování vozu

Pokud chcete sledovat určitý vůz, zapište jeho identifikační číslo do textového pole v druhém řádku. V tomto poli při pozastavení máte aktuální počet vytvořených vozů. Pokud zadáte číslo mimo rozsah, vůz nebude nalezen, protože ještě neexistuje. Po stisku tlačítka *Najdi* se v druhém poli na tomto řádku zobrazí město, ve kterém se vůz nachází a počet tun, které veze.

Sledování stavu zmrzlinářství

Pro sledování stavu určitého města zadejte do textového pole na třetím řádku id města, jehož stav chcete zobrazit (číslo od 1 do 3000). Po stisku tlačítka *Zjisti* se v druhém textovém poli na řádku zobrazí stav města. Pokud město ještě neobjednalo žádnou zmrzlinu, je jeho stav *Neobjednalo*. Pokud byly již všechny jeho objednávky vyřízeny, je jeho stav *Obslouženo*. Pokud čeká na vyřízení nějaké své objednávky, je jeho stav *Čeká*. Když si zmrzlinářství objednalo, ale objednávka ještě nebyla přijata/odmítnuta, má město stav *Objednalo*.

Program můžete kdykoliv ukončit stisknutím tlačítka *Konec*.

Závěr

I přes drobné potíže se nám povedlo vytvořit program splňující zadání, který by měl být dále rozšiřitelný. Protože se jednalo o náš první takto rozsáhlý projekt, navíc první práci v týmu, bylo to dost složité. Přesto jsme se svojí prací spokojeni a myslíme si, že jsme ji zvládli poměrně dobře.

Bohužel až při testování jsme zjistili, že algoritmus pro plánování tras není příliš vhodně zvolený. Kvůli splnění všech podmínek jsme postupně museli ztěžovat kritéria pro spojování cest, a přestože při počátečních pokusech jsme vypravovali auta téměř vždy plná, nyní jezdí průměrně s třetinovým nákladem. Nejspíše jsme tedy měli zvolit jiný postup.

Literatura

[1] M.Battarra, R.Baldacci, D. Vigo: *Clarke and Wright Algorithm*, 5/2007.