FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

Semestrální práce z předmětu
Programovací techniky

Camel Warehouse Management System

Autoři:

Jakub Křižanovský
A21B0192P
krizanoj@students.zcu.cz

Stanislav Kafara
A21B0160P
skafara@students.zcu.cz

Obsah

Zadání

Zásobovací společnost *Necháme to bloudovi s. r. o.* se specializuje na přepravu zboží do saharských oáz. Její majitel Harpagon Dromedár je však vyhlášená držgrešle, a tak nejraději využívá jako přepravní prostředky velbloudy, kteří nejsou nároční na údržbu a provoz. Přepravované zboží je uskladněno ve speciálních koších, které jsou přichycovány na velbloudí hrby. Pro svoji živnost Harpagon využívá jak velbloudy jednohrbé, známé též jako dromedáry, tak velboudy dvouhrbé, kteří jsou někdy označováni jako drabaři. V poslední době se však starému Harpagonovi moc nedaří a spousta zvířat mu v poušti, částečně i vlivem změny klimatu, uhynula, což je pro něj citelná finanční rána, která mu dělá vrásky na čele. Rozhodl se tedy, že je čas dát prostor moderním technologiím, a proto si chce nechat vytvořit software, který mu pomůže rozplánovat přepravu všeho poptávaného zboží do oáz tak, aby nepřišel o nějaké další zvíře, dodržel závazky a neztratil klientelu, maximálně využil nosnosti zvířat a zároveň zvířata zbytečně neunavil delší cestou, než kterou opravdu musí jít. Tyhle požadavky můžeme jednodušše označit jako snahu o minimalizaci „ceny“ přepravy. Vytvořme pro Harpagona simulační program, který mu pomůže naplánovat přepravu, známe-li:

- počet skladů S,
- každý sklad bude definován pomocí:
  - kartézských souřadnic skladu x_s a y_s,
  - počtu košů k_s, které jsou do skladu vždy po uplynutí doby t_s doplněny. Na začátku simulace předpokládejte, že došlo k doplnění skladů, tj. ve skladu je k_s košů,
  - doby t_n, která udává, jak dlouho trvá daný typ koše na velblouda naložit/vyložit (každý sklad může používat jiný typ koše, se kterým může být různě obtížná manipulace),
- počet oáz O,
- kartézské souřadnice každé oázy x_o a y_o,
- počet přímých cest v mapě C,
- seznam cest, přičemž každá cesta je definována indexy i a j, označujících místa (oáza, sklad) v mapě, mezi kterými existuje přímé propojení, a platí: $i \in \{1,...,S,S+1,...S+O\}, j \in \{1,...,S,S+1,...S+O\}$, tj. sklady jsou na indexech od 1 do S a oázy na indexech od $S+1$ do $S+O$. Pozn. pokud existuje propojení z místa i do místa j, pak existuje i propojení z místa j do místa i,
- počet druhů velbloudů D,
- informace o každém druhu velblouda, kterými jsou:
  - slovní označení druhu velblouda, které bude uvedeno jako jeden řetězec neobsahující bílé znaky,
  - minimální v_{min} a maximální v_{max} rychlost, kterou se může daný druh velblouda pohybovat, přičemž jedinec se pohybuje konstantní rychlostí (obecně reálné číslo), která mu bude vygenerována v daném rozmezí pomocí rovnoměrného rozdělení,
  - minimální d_{min} a maximální d_{max} vzdálenost, kterou může daný druh velblouda překonat na jedno napojení, přičemž pro jedince je tato doba opět konstantní a jedná se o reálné číslo vygenerované v daném rozmezí pomocí normálního rozdělení se střední hodnotou $\mu = (d_{min} + d_{max})/2$ a směrodatnou odchylkou $\sigma = (d_{max} - d_{min})/4$, (pozn. velbloudi se mohou napít pouze v oázách a nebo ve skladech, jinde není voda k dispozici),
  - doba t_d, udávající kolik času daný druh velblouda potřebuje k tomu, aby se napil,
  - počet košů k_d udávající maximální zatížení daného druhu velblouda,
  - hodnota p_d udávající poměrné zastoupení daného druhu velblouda ve stádě, hodnota je zadána jako desetinné číslo z intervalu < 0, 1 >, přičemž platí $\sum_{d=1}^{D} p_d = 1.0$,
- počet požadavků k obsloužení P,
- každý požadavek bude popsán pomocí:
  - času příchodu požadavku t_z (pozn. požadavek přichází doopravdy až v čase t_z, tzn. nemůže se stát, že by jeho obsluha začala dříve, a že by v době příchodu požadavku byl náklad již na cestě),
  - indexu oázy $o_p \in \{1, \ldots, O\}$, do které má být požadavek doručen,
  - množství košů k_p, které oáza požaduje,
  - doby t_p udávající, za jak dlouho po příchodu požadavku musí být koše doručeny (tj. nejpozději v čase $t_z + t_p$ musí být koše vyloženy v oáze).

Za úspěšně ukončenou simulace se považuje moment, kdy jsou všechny požadavky obsloužené a všichni velbloudi se vrátili do svých domovských skladů. V případě, že se některý požadavek nepodaří (z jakéhokoli důvodu) obsloužit včas, pak simulace skončila neúspěchem, o čemž bude program informovat příslušným výpisem (viz níže).

V rámci výpisů použijte jednu z následujících variant (formát je závazný, indexace od jedné):

- Příchod požadavku:

  Cas: <t>, Pozadavek: <p>, Oaza: <o>, Pocet kosu: <k>, Deadline: <t+t_p>

- Ve skladu se začíná připravovat velbloud na cestu:

  Cas: <t>, Velbloud: <v>, Sklad: <s>, Nalozeno kosu: <k>, Odchod v: <t+k·t_n>

- Velbloud došel do oázy, kde bude něco vykládat (pozn. výpis nikde v průběhu nebude odřádkován, časovou rezervou t_r je myšlen rozdíl mezi časem, kdy má být náklad nejpozději vyložen a časem, kdy k vyložení opravdu došlo):

  Cas: <t>, Velbloud: <v>, Oaza: <o>, Vylozeno kosu: <k>, Vylozeno v: <t+k·t_n>, Casova rezerva: <t_r>

- Velbloud došel do oázy/skladu, kde se musí napít před další cestou:

  Cas: <t>, Velbloud: <v>, Oaza: <o>, Ziznivy <druh>, Pokracovani mozne v: <t+t_d> nebo:

  Cas: <t>, Velbloud: <v>, Sklad: <s>, Ziznivy <druh>, Pokracovani mozne v: <t+t_d>

- Velbloud prochází oázou, ale nemá zde žádný zvláštní úkol:
- Cas: <t>, Velbloud: <v>, Oaza: <o>, Kuk na velblouda
- Velbloud dokončil cestu a vrátil se do skladu:
- Cas: <t>, Velbloud: <v>, Navrat do skladu: <s>
- Požadavek se nepodařilo (z jakéhokoli důvodu) obsloužit včas:
- Cas: <t>, Oaza: <o>, Vsichni vymreli, Harpagon zkrachoval, Konec simulace

Výstup Vašeho programu bude do standardního výstupu a bude vypadat například následovně:

```
...
Cas: 12, Pozadavek: 2, Oaza: 2, Pocet kosu: 3, Deadline: 30
Cas: 12, Velbloud: 5, Sklad: 3, Nalozeno kosu: 3, Odchod v: 18
Cas: 21, Velbloud: 5, Oaza: 1, Ziznivy dromedar, Pokracovani mozne v: 22
Cas: 23, Velbloud: 5, Oaza: 10, Kuk na velblouda
Cas: 24, Velbloud: 5, Oaza: 2, Vylozeno kosu: 3, Vylozeno v: 30, Casova rezerva: 0
...
```

Čas ve výpisu bude zaokrouhlen dle pravidel zaokrouhlení na celé číslo.

Minimální požadavky

- Funkcionalita uvedená v zadání výše je **nutnou podmínkou pro finální odevzdání práce**, tj. simulační program při finálním odevzdání musí splňovat veškeré požadavky/funkcionalitu výše popsanou, **jinak bude práce ohodnocena 0 body**.
- V případě, že bude program poskytovat výše popsanou přepravu, ale **řešení bude silně neefektivní**, bude uplatněna bodová **penalizace až 20 bodů**.

- **Finální odevzdání práce** v podobě **.ZIP souboru** (obsahujícím zdrojové kódy + přeložené soubory + dokumentace) bude nahráno **na portál**, a to **dva celé dny před stanoveným termínem** předvedení práce, tj.:
    - studenti, kteří mají **termín předvedení** stanoven na **pondělí**, nahrají finální verzi práce na portál **nejpozději v pátek ve 23:59**,
    - studenti, kteří mají **termín předvedení** stanoven na **čtvrtek**, nahrají finální verzi práce na portál **nejpozději v pondělí ve 23:59**.

  Při nedodržení tohoto termínu bude uplatňována **bodová penalizace 30 bodů**, navíc studentům nemusí být umožněno předvedení práce ve smluveném termínu.

- Seznamte se se strukturou vstupních dat (polohou skladů a oáz, informacemi o cestách, velbloudech, požadavcích ...) a načtěte je do svého programu. Formát souborů je popsán přímo v záhlaví vstupního souboru tutorial.txt **(5 bodů)**.
- Navrhněte a implementujte vhodné datové struktury pro reprezentaci vstupních dat, důsledně zvažujte časovou a paměťovou náročnost algoritmů pracujících s danými strukturami **(10 bodů)**.
- Proveďte základní simulaci jedné obslužné trasy včetně návratu velblouda do skladu. Vypište celkový počet doručených košů > 0 a celkový počet obsloužených požadavků > 0. Trasa velblouda musí být smysluplná. **(10 bodů)**.

Další požadavky

- Vytvořte prostředí pro snadnou obsluhu programu (menu, ošetření vstupů včetně kontroly vstupních dat) - nemusí být grafické, během simulace umožněte manuální zadání nového požadavku na zásobování některé oázy či odstranění některého existujícího **(5 bodů)**.
- Umožněte sledování (za běhu simulace) aktuálního stavu přepravy. Program bude možné pozastavit, vypsat stav přepravy, krokovat vpřed a nechat doběhnout do konce, podobně jako je tomu v debuggeru **(5 bodů)**.
- Proveďte celkovou simulaci a vygenerujte do souborů následující statistiky (v průběhu simulace ukládejte data do vhodných datových struktur, po jejím skončení je uložte ve vhodném formátu do vhodně zvolených souborů) **(10 bodů)**:
    - přehled jednotlivých velbloudů - základní údaje o velbloudovi (druh; id domovského skladu; rychlost; max. vzdálenost, kterou urazí na jedno napojení), uskutečněné trasy (čas, kdy opustil sklad; kudy šel; kolik toho vezl; kam a kdy doručoval zboží; kde a kdy se zastavil na napojení; kdy se vrátil do svého domovského skladu), jak dlouho za celou dobu simulace odpočíval (tj. byl ve skladu a čekal na přiřazení požadavku) a celkovou vzdálenost, kterou ušel,
    - přehled jednotlivých oáz - čas a velikost vzniklého požadavku; kdy musel být nejpozději doručen; kdy byl skutečně doručen; ze kterého skladu a kterým velbloudem byl obsloužen,
    - přehled jednotlivých skladů - časy, kdy došlo k doplnění skladu; kolik košů v té době ve skladu zbývalo a kolik jich je k dispozici po doplnění,
    - délka trvání celé simulace, celková doba odpočinku všech použitých velbloudů, celková ušlá vzdálenost, kolik velbloudů od jednotlivých druhů bylo použito. Nemá-li úloha řešení, vypište, kdy a kde došlo k problému.

- Vytvořte generátor vlastních dat. Generátor bude generovat vstupní data pomocí rovnoměrného rozdělení, přičemž volte vhodně rozsah hodnot pro jednotlivé veličiny. U seznamu cest se vyhněte duplikátům. Data budou generována do souboru (nebudou přímo použita programem) o stejném formátu jako již dodané vstupní soubory. Při odevzdání přiložte jeden dataset s řešitelnou úlohou a jeden dataset, kdy nebude možné obsloužit všechny požadavky včas. **(5 bodů)**.
- Vytvořte dokumentační komentáře ve zdrojovém textu programu a vygenerujte programovou dokumentaci (Javadoc) **(10 bodů)**.
- Vytvořte kvalitní dále rozšiřitelný kód - pro kontrolu použijte softwarový nástroj PMD (více na http://www.kiv.zcu.cz/~herout/pruzkumy/pmd/pmd.html), soubor s pravidly pmdrules.xml najdete na portálu v podmenu Samostatná práce **(10 bodů)**
    - mínus 1 bod za vážnější chybu, při 6 a více chybách nutno opravit,
    - mínus 2 body za 10 a více drobných chyb.

- **V rámci strukturované dokumentace (celkově 20 bodů):**
  - připojte zadání **(1 bod)**,
  - popište analýzu problému **(5 bodů)**,
  - popište návrh programu (např. jednoduchý UML diagram) **(5 bodů)**,
  - uživatelskou dokumentaci **(5 bodů)**,
  - zhodnoťte celou práci a vytvořte závěr **(2 body)**,
  - uveďte přínos jednotlivých členů týmu (včetně detailnějšího rozboru, za které části byli jednotliví členové zodpovědní) k výslednému produktu **(2 body)**.

Analýza problému

Pro úplnou a efektivní implementaci programu je nutno zvážit, jak načíst vstupní data, jak reprezentovat načtená a další simulační data, jak bude simulace řízena, jak rozhodovat o tom, jaké sklady budou distribuovat zboží požadované oázami, jak uchovávat záznamy o dění simulace pro generování statistik a jaké bude rozhraní pro komunikaci programu s uživatelem.

Načítání vstupních dat

Prvním problémem je načítání vstupních dat. Vstupní data mají jednoznačně danou strukturu, není tedy problém soubor předzpracovat tak, že nebude obsahovat komentáře a nadbytečné bílé znaky a pak hodnotu po hodnotě předat programu.

Reprezentace dat

Druhým problémem je reprezentace dat. Načtená data, např. sklady, oázy, typy velbloudů, …, která se v průběhu simulace nemění, můžeme jednoduše uložit do pole jako objekty mající příslušné atributy a metody, jakým akcím jsou zodpovědné. Další simulační data, jako velbloudi patřící určitému skladu, které se v průběhu mění, bude vhodné ukládat do dynamických datových struktur, jako je např. dynamické pole nebo hashovací tabulka.

Řízení simulace

Dalším problémem je, jak bude simulace řízena. Nedává velký smysl, aby simulace byla krokována po konstantním kroku, proto bude krokována po událostech.

Událostmi budou např. přijetí požadavku od oázy, vyslání velblouda na cestu, průchod velblouda oázou a jiné. Události se budou postupně zpracovávat podle jejich času nastání, tzn. V čase jejich nastání se budou vykonávat jim přiřazené akce.

Prioritní fronta takových událostí se nabízí jako vhodné řešení. Události v ní budou řazeny podle času jejich nastání.

Rozdělování požadavků

Hlavním problémem práce je rozdělování přijatých požadavků jednotlivým skladům a velbloudům. Do úvahy musíme vzít vhodnost skladů a velbloudů pro obsluhování požadavků a počítání doručovacích tras.

Sklady jsou nejvhodnější k obsluhování požadavku, pokud z nich vedou do oázy nejkratší trasy, velbloudi ve skladech jsou schopni je přejít a zároveň skladům neschází velbloudi, resp. jich schází nejmenší počet potřebných k doručení požadavku.

Pro výpočet tras, tj. cest v grafu s uzly reprezentovanými sklady a oázami, můžeme použít dva přístupy, a sice předpočítání tras nebo počítání tras v čase potřeby nebo jejich kombinaci. Pro předpočítání všech možných tras je vhodný Floyd-Warshall algoritmus [1] s časovou složitostí $O(n^3)$ a paměťovou náročností $O(n^2)$, kde n je počet uzlů grafu. Pro vypočtení trasy v čase potřeby je vhodný Dijkstrův algoritmus [1] s časovou složitostí

$O(m \cdot \log(n))$, kde m je počet hran orientovaného grafu. Implementujeme oba algoritmy a po načtení vstupních dat rozhodneme, jaký algoritmus bude v simulaci používán. Floyd-Warshall algoritmus je vhodný pro husté grafy, grafy s malým počtem uzlů a mnoho požadavků na cesty mezi uzly. Naopak Dijkstrův algoritmus použijeme, pokud graf bude řídký nebo bude mít příliš uzlů na to, aby se Floyd-Warshall matice dala v přijatelném čase vypočíst nebo se vůbec vešla do operační paměti.

Neznáme způsob, jak by bylo možné zohlednit veškeré optimalizované proměnné algoritmem s rozumnou časovou náročností, proto naše řešení nebude optimální, ale nastavením parametrů by se mělo optimu alespoň přibližovat.

Nemůžeme také předvídat, jaké budou vznikat požadavky, takže není jednoznačné, jaké sklady prioritizovat ve výběru obsluhy požadavku a také nelze říct, jestli by bylo lepší např. čekat s několika splnitelnými požadavky se snahou doručování v čase deadline a minimalizovat tak počet velbloudů, když by nakonec přišel požadavek, který by obsloužit nešel a požadavků by se celkem obsloužilo méně, než snažit se požadavek obsloužit hned po jeho přijetí, proto budeme požadavky obsluhovat v čase jejich vzniku.

Generování statistik

Dalším problémem je uchovávání záznamů o dění simulace pro generování statistik. Záznamy o pití velbloudů, jejich tras, …, mohou být uchovávány v hashovací tabulce a dynamickém poli a při ukončení simulace zapsány do souborů.

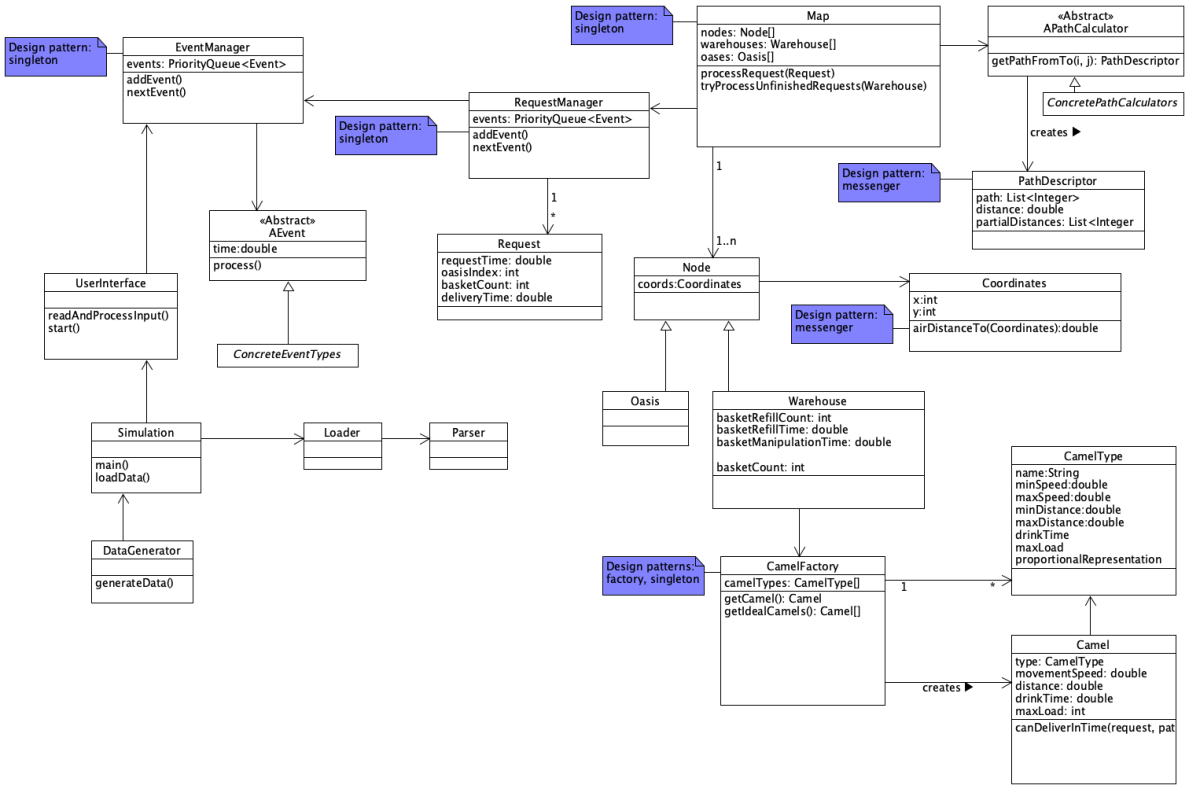
Uživatelské rozhraní

Dalším problémem je interakce programu s uživatelem. Rozhraní může být textové, jelikož příkazy budou jednoduché. Příkaz se předá k vyhodnocení a na základě něj se vykoná příslušná akce simulace.

Návrh programu

Zjednodušený UML diagram

Jedná se o zjednodušenou verzi UML diagramu programu. Snažil jsem se zachytit ty nejpodstatnější atributy a metody tak aby diagram zůstal přehledný.



Obrázek 1: UML diagram

Řešení simulačního času

Při analýze problému jsme zjistili, že čas by nebylo vhodné přičítat po konstantních intervalech, ale lepším řešení by bylo mít frontu určitých událostí, které se mají v určitý čas stát.

Problém řeší třída EventManager, která si uchovává události (abstraktní třída AEvent) a při po dokončení zpracování jedné události, vždy skočí na další událost a nastaví aktuální simulační čas na čas, kdy se má událost stát. Tyto eventy uchovává v prioritní frontě seřazené podle času, aby šla vždy co nejrychleji vybrat první položka a položky šly také relativně rychle přidávat.

Výpočet trasy

Pro výpočet trasy je použito několik algoritmů. Hlavními z nich jsou Floyd-Warshallův algoritmus a Dijkstrův algoritmus. Floyd-Warshallův algoritmus si všechny trasy napočítá v preprocessingu a pak už je získání nejkratší trasy velice rychlé. Hodí se tedy pro husté grafy. Dijkstrův algoritmus oproti tomu vše počítá za běhu programu, ale zase nepočítá trasy, které pak ani nakonec nebudou využity. Hodí se spíše pro řídké grafy.

Program po načtení dat spočítá hustotu grafu podle vzorce:

$$D = \frac{|E|}{|V|(|V|-1)}$$

kde $|E|$ je počet hran a $|V|$ je počet vrcholů grafu. Podle této hustoty se pak algoritmus rozhodne, který z těchto dvou algoritmů použije. Floyd-Warshallův algoritmus také není použit, pokud by výsledná distanční matice byla příliš velká a nevešla by se do paměti.

Experimentovali jsme také s algoritmem A*, který při počítání trasy využívá heuristickou funkci, ale nakonec jsme ho nevyužili.

Dále jsme si také v datech všimli, že některé grafy mají tvar „hvězdy“, kde je jeden centrální vrchol uprostřed a všechny cesty vedou z něj do všech ostatních vrcholů. Takovéto grafy program detekuje a pak už je počítání trasy velice jednoduché, protože trasa odkudkoliv kamkoliv vždy povede pouze přes střed. Tato detekce významně zrychlí celý algoritmus, protože pak výpočet trasy probíhá v konstantním čase.

Algoritmus distribuce

Pro distribuci jsme zvolili greedy algoritmus, který vždy preferuje co nejrychlejší možné doručení koše. Zde je pseudokódem zjednodušeně popsaný algoritmus.

Map.processRequest(request):

```
Seřaď sklady podle vzdušné vzdálenosti od oázy od nejbližšího po nejvzdálenější;
for(sklad : top 10 seřazených skladů, které mají koše):
	pathDescriptor <- spočítej nejkratší vzdálenost pomocí pathCalculatoru;

	basketAmount <- min(počet zbýv. košů requestu, počet košů ve skladu);

	if(pathDescriptor lze projít && požadavek lze stihnout):
		warehouse.distrubute(request, basketAmount, pathDescriptor);
		request.basketsRemaining -= basketAmount;
		uber basketAmount ve skladu;
		if(basketsRemaining == 0):
			return;

Přidej request do unfinished;
```

Warehouse.distribute(request, basketAmount, pathDescriptor):

```
//Zkus použít velbloudy ve skladu
while(basketAmount > 0 && máme další velbloudy ve skladu):
	camel <- další velbloud ve skladu;
	load <- min(basketAmount, max zátěž camela);
	if(camel dokáže doručit požadevek danou cestou s danou náloží load včas):
		odečti od zásob košů ve skladu load;
		basketAmount -= load;
		přidej CamelPrepareEvent;
		odeber velblouda z velbloudů ve skladu;

//Generuj velbloudy, a ty používej
while(basketAmount > 0):
	camel <- vygeneruj nového velblouda;
	nastav velbloudovi domov na tento sklad;
	load <- min(basketAmount, max zátěž camela);
	if(camel dokáže doručit požadevek danou cestou s danou náloží load včas):
		odečti od zásob košů ve skladu load;
		basketAmount -= load;
		přidej CamelPrepareEvent;
	else:
		přidej velblouda do velbloudů ve skladu;
```

Map.tryProcessUnfinishedRequests(warehouse):

```
for(request : unfinishedRequests):
	if(warehouse nema kose):
		return;
	if(oaza request je příliš daleko vzdušnou čarou od skladu na včasné doručení):
		continue;

	basketAmount <- min(počet zbýv. košů requestu, počet košů ve warehouse);
```

```
pathDescriptor <- spočítej nejkratší vzdálenost pomocí pathCalculatoru;

if(pathDescriptor lze projít && požadavek lze stihnout):
	warehouse.distrubute(request, basketAmount, pathDescriptor);
	request.basketsRemaining -= basketAmount;
	uber basketAmount ve skladu;
	if(basketsRemaining == 0):
		odeber request z unfinishedRequests;
		continue;
```

Uživatelská dokumentace

Spuštění programu

Program lze spustit z příkazové řádky pomocí dávkového souboru run.sh / run.cmd.

Uživatelské rozhraní

Uživatelské rozhraní je konzolové a ovládá se pomocí příkazů. Všechny příkazy lze zobrazit příkazem help. U příkazů je vždy prováděna kontrola vstupů, aby do programu nešly zadávat nesmyslné hodnoty. Kdykoliv program čeká na vstup od uživatele, je to uživateli dáno najevo pomocí znaku $.

Možné příkazy

- help – vypíše všechny možné příkazy.
- load <jméno souboru> - načte data z příslušného souboru, který se musí nacházet ve složce data.
- start – spustí simulaci / nechá doběhnout pozastavenou simulaci.
- load_and_start <jméno souboru> - načte data z příslušného souboru a spustí simulaci
- schedule_pause <simulační čas> - simulace se při spuštění v daném simulačním čase pozastaví.
- step [<počet kroků>] – krokuje program – vykoná další událost, která se má stát. Takto se počítají pouze události, které něco vypisují. Pokud je nastaven parametr <počet kroků> krokování opakuje, jinak provede pouze 1 krok.
- time – vypíše aktuální simulační čas.
- list_requests [<stav>] – vypíše všechny požadavky a jejich stav. Pokud je navíc jako parametr uveden stav, vypíše pouze požadavky, které jsou v tomto stavu. U požadavků také vypisuje jejich číslo, aby se s nimi dalo dále pracovat.
- request_info <číslo požadavku> – vypíše informace o požadavku s tímto číslem
- add_request <čas příchodu> <index oázy> <počet košů> <čas na doručení> - přidá nový požadavek se zadanými vlastnostmi. Formát parametrů je stejný jako ve vstupních datech.
- cancel_request <číslo požadavku> – zruší požadavek se zadaným číslem.
- generate – vytvoří ve složce data nový soubor s daty pomocí generátoru dat.
- generate_and_start – vytvoří nový soubor s daty, rovnou ho načte a spustí.
- quit/exit – ukončí program.

Příklad užití

1. Načteme pomocí příkazu load soubor tutorial

   $ load tutorial

2. Vypíšeme požadavky

   $ list_requests

3. Vypíše se nám 8 požadavků s čísly 0-7. Rozhodneme se zrušit požadavek #4

   $ cancel_request 4

4. Rozhodneme se, že budeme chtít v půlce simulaci pozastavit, proto nastavíme předem pauzu na čas 10
   $ schedule_pause 10

5. Simulaci chceme krokovat o 3 kroky vpřed

   $ step 3

6. Simulaci necháme doběhnout do konce

   $ start

Závěr a zhodnocení

Naše řešení splňuje minimální požadavky a další požadavky zadání. Program je navrhnut tak, že by měl být přehledný a dále rozšiřitelný.

Vybírání algoritmu počítání cest v grafu na základě vstupních dat nám zajistilo rychlý běh aplikace na většině vstupních dat. Naše řešení obsluhy požadavků v čase jejich vzniku vede k přílišnému generování velbloudů ve scénářích, kde požadavky vznikají po krátkých časových okamžicích, i přesto že mají dlouhý limit doručení.

V řešení jsme si vystačili s datovými strukturami implementovanými ve standardní knihovně jazyku Java.

Bylo by vhodné si dále promyslet systém rozdělování požadavků a případně implementovat lepší, z časových důvodů už na to ale čas nezbyl.

Rozdělení práce mezi členy v týmu

Jakub Křižanovský:

- Základní návrh tříd
- Implementace Floyd-Warshallova algoritmu
- Uživatelské rozhraní
- Generátor dat
- Časová optimalizace algoritmu

Stanislav Kafara:

- Parser a načítání dat
- Implementace Dijkstrova algoritmu
- Eventy
- Statistiky

Společnými silami:

- Algoritmus distribuce
- Javadoc
- PMD
- Dokumentace

Literatura

[1] Pavel Mautner, Grafové algoritmy 1. Podklady k přednáškám [Online] [Citace 2. 12. 2022] https://www.kiv.zcu.cz/~mautner/Pt/grafove_algoritmy1.pdf