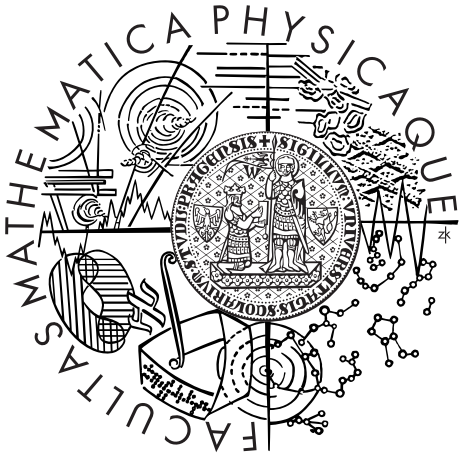
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Marek Vandas

Rozvrhování zdrojů na letišti jako časově omezený rozvrhovací problém

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Zuzana Reitermanová

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2012

Poděkování:
Děkuji Mgr. Zuzaně Reitermanové za ochotu, trpělivost a podnětné připomínky při zpracování bakalářské práce. Dále děkuji své rodině za podporu, kterou mi při studiu projevuje.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Rozvrhování na letišti jako časově omezený rozvrhovací problém

Autor: Marek Vandas

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Zuzana Reitermanová, Katedra teoretické informatiky a matematické logiky MFF UK

Abstrakt: V této práci jsou identifikovány požadavky na bezpečný pozemní letištní provoz, který se skládá z přidělování ranvejí, taxi operací a přidělování bran. Je ukázáno, jak je možné tento problém modelovat jako problém splňování omezujících podmínek a vyřešit jej pomocí technik rozvrhování. Součástí práce je aplikace, která tyto techniky ilustruje a umožňuje následnou vizualizaci výsledků. Pro účely této aplikace byl vytvořen obecný řešič podmínek, který je snadno rozšiřitelný a využitelný i na jiný druh problémů. Další výhodou tohoto řešiče je, že umožňuje snadnou změnu prohledávací strategie.

Klíčová slova: rozvrhování, omezující podmínky, letištní provoz

Title: Airport scheduling as time constrained resource scheduling problem

Author: Marek Vandas

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Zuzana Reitermanová, Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis identifies constraints for safe ground airport operations. These operations consist of runway assignment, taxi operations planning and gate scheduling. The aim of this thesis is to show how this problem can be formulated as constraint satisfaction problem and then solved as a scheduling problem. Based on this model, an application that ilustrates these concepts is designed and implemented. This application enables a visualisation of results. An extendable constraint solver was implemented for the purpose of this application. This solver can be used to solve problems from other domains as well and also enables easy change of search strategy.

Keywords: airport scheduling, constraint satisfaction problem, scheduling problem

Obsah

Úvod

V posledních letech se každoročně zvyšuje provoz na letišťích přibližně o 5 procent. Toto klade velké požadavky na vytíženost existujících letišť a má za následky zpoždění mnoha letů. Rozšiřování letišť je velice nákladné, a proto je potřeba dostupné zdroje využívat co možná nejlepším způsobem. Ovšem množství omezujících podmínek a vzájemných interakcí jednotlivých zdrojů představuje pro lidského kontrolora velkou zátěž, tudíž je vhodné navrhnout pomocný počítačový nástroj, který by pro zadaný problém navrhoval vhodné řešení.

Tato práce se zabývá modelováním a rozvrhováním pozemních operací na leťišti. Pozemní operace zahrnují přílety, odlety, taxislužby a přidělování bran. Vzhledem k náhodné povaze požadavků na tyto operace se rozvhování týká pouze následného časového intervalu zhruba 5 - 20 minut. Nejdůležitějším zdrojem na letišti jsou ranveje, které jsou využívány pro přílety, odlety i taxi služby. Optimalizace jejich využití je tedy klíčovým prvkem k dobrému řešení.

Problém je modelován jako rozvrhovací problém s využitím techniky splňování omezujících podmínek. Programování s omezujícími podmínkami (CP - constraint programming) je přístup k řešení především kombinatorických problémů pomocí popisu tohoto problému v řeči proměnných a podmínek mezi těmito proměnnými. Má velmi silné teoretické základy. Mezi velké výhody CP patří oddělení popisu problému od jeho vyřešení nebo možnost vysokoúrovňového zapsání problému. Jednou z nejúspěšnějších aplikací CP je právě rozvrhování.

Z důvodu složitosti skutečného provozu na letištích si tato práce neklade za cíl vytvoření aplikace, která by mohla být plnohodnotně využita na letišti. Cílem této práce je identifikovat důležité body, které pro letištní provoz platí, a ukázat, jak lze tyto body modelovat pomocí splňování omezujících podmínek. Poté je pro zjednodušenou variantu tohoto problému implementován nástroj, jehož cílem je nalézt v omezeném čase co nejlepší rozvrh. Vstupem je sekvence pozemních letištních operací se zadanou požadovanou množinou zdrojů. K řešení problému splňování podmínek je využíván pro účely této práce implementovaný řešič systémů podmínek. Je navržen tak, aby byl snadno rozšiřitelný. Již v jeho

současné podobě je možné jej využít k řešení i jiných než rozvrhovacích problémů.

Motivací pro vznik této práce byla má touha proniknout lépe do problematiky CP a zjistit, jaké existují techniky v automatickém rozvrhování.

Tato práce je rozdělena do několika kapitol. V první kapitole uvedeme čtenáře do problematiky programování s omezujícími podmínkami. Shrneme zde základní vlastnosti podmínek a nastíníme, jak lze systémy podmínek řešit. V druhé kapitole definujeme, co je rozvrhovací problém a jak lze representovat pomocí omezujících podmínek. Ve třetí kapitole bude rozebírán provoz na letišti. Budou zde diskutovány požadavky na pozemní provoz na letišti a z nich vyplývající navržený CP model. Na závěr této kapitoly je popsán vytvořený algoritmus řešící zadaný problém. Na jeho základě je odvozen heuristický algoritmus, jehož cílem je nalézt co možná nejlepší řešení v nejkratší době. K práci jsou připojeny přílohy, ve kterých lze nalézt programátorskou a uživatelskou dokumentaci, popis vstupních souborů a popis obsahu přiloženého CD.

Kapitola 1

Programování s omezujícími podmínkami

Programování s omezujícími podmínkami (CP - constraint programming) [1] je přístup k řešení problémů pomocí popisu tohoto problému v řeči proměnných a podmínek mezi těmito proměnnými. Jejich deklarativní charakter dovoluje oddělit popis problému od jeho vyřešení a nabízí lidsky srozumitelný modelovací jazyk. Omezující podmínky našly uplatnění v řadě praktických aplikací. Tyto oblasti zahrnují zpracování přirozeného jazyka, počítačovou grafiku nebo databázové systémy. Těžištěm využití omezujících podmínek jsou však oblasti plánování a rozvrhování [3].

V této kapitole budou nejdříve definovány základní prvky CP. Poté bude presentován přehled základních řešících technik s důrazem na systematické prohledávání zahrnující metody konzistence. Algoritmy popsané v této kapitole (algoritmus 1 – 4) byly implemetovány a jejich realizace je popsána v příloze A.

1.1 Úvod do problematiky

Základními prvky CP jsou proměnné a podmínky. Každá proměnná má množinu hodnot, kterých může nabývat. Tato množina se nazývá *doména* proměnné. Podmínky omezují možné kombinace hodnot jednotlivých proměnných. Podmínkou může být libovolná závislost hodnot proměnných. Tedy například tvrzení, že dvě hodnoty musí být různé nebo tvrzení, že součet hodnot musí nabývat hodnotu 0. Podmínky je možné větvit pomocí klasických logických formulí – konjukce, disjunkce nebo ekvivalence. Nabízejí tedy atraktivní modelovací jazyk se silnými vyjadřovacími prostředky.

Systémy založené na CP můžeme rozdělit do dvou kategorií. V první katego-

rii jsou problémy s proměnnými s konečnými doménami. Tento druh problémů se nazývá CSP (Constraint Satisfaction Problem). Druhou kategorii tvoří problémy s nekonečnými doménami. Příkladem nekonečné domény může být interval reálných čísel. Tyto systémy je buď možné pomocí diskretizace domén převést na CSP, nebo je řešit pomocí numerických metod. Tato práce se zabývá pouze problémy s konečnými doménami. Čas, který se jeví jako spojitá veličina, je pro účely práce diskretizován do uniformních časových intervalů.

Formální definice CSP lze najít třeba v [1] a vyjádřit takto:

Definice 1 *CSP je trojice* (X, D, C)*, kde* $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$ *je konečná množina neznámých,* $\boldsymbol{D} = (D_1, D_2, \ldots, D_n)$ *je konečná množina odpovídajících domén a* $\boldsymbol{C} = (C_1, C_2, \ldots, C_m)$ *je konečná množina podmínek. Podmínka je relace na podmnožině* $C_l \subseteq D_{i_1} \times \ldots \times D_{i_k}$*, která definuje hodnoty proměnných* $x_{i_1}, \ldots, x_{i_k}$ *splňující* C_l*. Úplné ohodnocení proměnných je n-tice hodnot* $(v_1, v_2, \ldots, v_n)$ *proměnných* $(x_1, x_2, \ldots, x_n)$*, kde* $v_i \in D_i$*. Řešením CSP je úplné ohodnocení proměnných, při kterém jsou všechny podmínky splněny.*

Příklad 1: Jako jednoduchý příklad si představme CSP problém se dvěmi proměnnými x a y, kde $x \in \{0, 1, 2, 3\}$ a $y \in \{1, 2, 3\}$ a s podmínkami $c_1 : x+y = 3$ a $c_2 : y > 1$. První podmínku lze vyjádřit pomocí kartézského součinu domén jako $c_1 = \{(0, 3), (1, 2)\}$, druhou podmínku jako $c_2 = \{(2), (3)\}$. Řešením tohoto problému je například ohodnocení $x = 1, y = 2$. Ohodnocení, která nesplňují některou podmínku, se nazývají nekonzistentní.

1.2 Řešení problému splňování podmínek

Při řešení CSP můžeme požadovat libovolné úplné konzistentní ohodnocení, všechna úplná konzistentní ohodnocení nebo jedno optimální. Poslední kategorie problémů se nazývá CSOP (Constraint Satisfaction Objective Programming). CSP jako takové je NP-úplné [4]. Z tohoto důvodu nám často v optimalizačních problémech stačí pouze dobré řešení, neboť nalezení optimálního by trvalo velice dlouho. Existují dva základní přístupy k řešení zadaného CSP. Jsou jimi lokální a nebo systematické prohledávání.

1.2.1 Lokální prohledávání

Východiskem lokálního prohledávání je plné ohodnocení proměnných. Lokální prohledávání poté pracuje v iteracích. V každé iteraci máme úplné ohodnocení proměnných, je tedy snadné otestovat každou podmínku, zda je, nebo není splněna.

Pokud jsou všechny podmínky splněny, můžeme skončit, neboť jsme našli řešení. V opačném případě určitým způsobem změníme úplné ohodnocení na jiné, takzvané *sousední ohodnocení*. Toto sousední ohodnocení se může například lišit pouze v hodnotě jedné proměnné - vybereme tedy jednu z proměných v porušených podmínkách a změníme její hodnotu. Tím jsme se opět dostali do úvodní fáze algoritmu s plným ohodnocením proměnných. Tyto kroky opakujeme, dokud nenalezneme řešení, nebo nedosáhneme maximálního počtu iterací. Existují různé varianty tohoto algoritmu. Jmenujme třeba metodu největšího stoupání (Hill Climbing), minimalizace konfliktů nebo tabu seznam [1]. Algoritmy lokálního prohledávání bývají často založeny na náhodném, stochastickém prohledávání . Tyto algoritmy nacházejí uplatnění v mnoha praktických aplikacích, kde je vyžadováno nalezení nějakého, i když ne úplně konzistentního řešení. Nevýhodou těchto algoritmů je, že nezaručují nalezení řešení (v případě, že nějaké existuje). Nejsou tedy *úplné*. Proto se zaměříme na metody systematického prohledávání.

1.2.2 Systematické prohledávání

Druhým typem algoritmů jsou algoritmy založené na systematickém prohledávání. Jednou z možností algoritmů takovéhoto typu je metoda generuj a testuj (generate and test [13]) – systematicky generuj všechna možná řešení a pro každé kompletní ohodnocení proměnných otestuj, jestli jsou všechny podmínky splněny. Efektivnost této varianty je velmi slabá, a proto bývá nahrazována metodou backtrackingu (prohledávání s navracením). Postupně ohodnocujeme proměnné. V každém kroku máme tedy některé proměnné ohodnoceny a ostatní mají zatím neurčenou hodnotu. Po každém ohodnocení další proměnné otestujeme platnost podmínek, pro které platí že, obsahují právě ohodnocenou proměnnou a zároveň že všechny ostatní proměnné zahrnuté v těchto podmínkách již mají známou hodnotu. Pokud některá podmínka není splněna, vrátíme se k poslední ohodnocované proměnné se zatím nevyzkoušenou hodnotou a zkusíme tuto hodnotu. Toto prohledávání je vždy efektivnější než metoda generuj a testuj z toho důvodu, že pokud je nalezena nekonzistentnost, není už existující ohodnocení dále zbytečně rozšiřováno a je změněno. Algoritmus 1 shrnuje metodu tohoto řešení.

Nevýhodou tohoto algoritmu je takzvaný trashing neboli opakování stejných chyb. Zjištění nekonzistence nastává až po úplném ohodnocení všech proměnných zúčastněných v podmínce. Po zjištění, že žádná hodnota v doméně proměnné není konzistentní, se algoritmus vrací k předešlé proměnné a zkouší pro ní jiné volby. Pokud ovšem konflikt způsobuje proměnná ohodnocovaná dříve, je tato práce zbytečná a dochází k opakovanému zjištění již zjištěného konfliktu.

Algoritmus 1: Label(X, A, C)

Input: X : neohodnocené proměnné, A : částečné ohodnocení proměnných,
C : podmínky

Output: Nalezne řešení CSP nebo vrátí, že řešení neexistuje

```
1 if X == ∅ then
2     return A
3 x ← vyber zatím neohodnocenou proměnnou z X
4 for každou hodnotu h v doméně proměnné x do
5     if A ∪ {x/h} je konzistentní s plně ohodnocenými podmínkami C then
6         R ← Label(X − {x}, A ∪ {x/h}, C)
7         if R ≠ fail then
8             return R
9 return fail
```

Algoritmus, který se za pomocí nesplněných podmínek pokusí vrátit k posledně ohodnocované proměnné, jež se účastnila v nějaké nesplněné podmínce, se nazývá backjumping [1].

Další nevýhodou tohoto algoritmu je příliš pozdní detekce nekonzistencí, nesplnění podmínky je odhaleno až po úplném ohodnocení zúčastněných proměnných. K odstranění tohoto problému jsou používány metody konzistence.

1.2.3 Metody konzistence

Hlavní myšlenkou metod konzistence je možnost využití podmínek nikoliv pouze k otestování platnosti řešení, ale také k redukování prohledávaného prostoru. Každé volání funkce Label obdržuje CSP problém, ve kterém jsou oproti původnímu problému změněny domény proměnných (daných ohodnocením některých proměnných). Tyto změny je ovšem možné pomocí podmínek promítnout i do domén ostatních proměnných. Podmínky jsou tedy aktivně využívány k odstranění hodnot z domén proměnných, dedukci nových podmínek a nalezení nekonzistencí. Tento proces aktivního používání podmínek se nazývá *propagace podmínek*. Množina hodnot, která není odstraněna tímto procesem, se nazývá *aktuální doména* proměnné. Propagace podmínek bývá opakována, dokud se domény proměnných nestabilizují nebo se některá z domén nevyprázdní. První stav se nazývá *fixpoint*. Pokud se některá doména vyprázdnila, žádné řešení s dosavadním ohodnocením neexistuje, a je tedy třeba se vrátit a zkusit jiné ohod-

nocení.

Příklad 2: Uvažme, jak toto lze použít v příkladu 1 (strana 6). Z první podmínky můžeme vyčíst, že x nemůže nabývat hodnotu 3. Můžeme tedy tuto dedukci promítnout do aktuální domény proměnné x. Více omezit domény pomocí samotné první podmínky nelze. Z druhé podmínky můžeme vyčíst, že y může nabývat pouze hodnot $\{2, 3\}$. Toto je promítnuto do aktuální domény proměnné y. Tato podmínka je *vyřešena* (resolved) - při již libovolné kombinaci hodnot je splněna. Jelikož se změnila doména y můžeme opět zkontrolovat, jestli pomocí první podmínky nelze omezit domény jejích proměnných. Je patrné, že s aktuálními doménami $x \in \{0, 1, 2\}, y \in \{2, 3\}$ proměnná x nemůže nabývat hodnotu 2. Finálně tedy omezíme domény na $x \in \{0, 1\}, y \in \{2, 3\}$. Dosáhli jsme tudíž stabilního stavu, ve kterém už nemůžeme udělat další redukci domén, aniž bychom změnili množinu řešení.

V literatuře existují různé druhy konzistence [1]. Definujme zde tedy základní druh konzistence.

Definice 2 *Mějme podmínku c s proměnnými $x_1, \ldots, x_n$ a aktuálními doménami $dom(x_i)$, $i = 1, \ldots, n$. O c řekneme, že je zobecněně hranově konzistentní (GAC - generalized arc consistent) právě tehdy když pro každou proměnnou x_i a každou hodnoutou $v_i \in dom(x_i)$ existují hodnoty $v_1, v_2, \ldots v_{i-1}, v_{i+1} \ldots, v_n$ z $dom(x_1)$, $dom(x_2)$, $\ldots dom(x_{i-1})$, $dom(x_{i+1})$, $\ldots$, $dom(x_n)$, že $c(v_1, v_2, \ldots, v_n)$ platí.*

Speciální případ této definice pro binární podmínky se nazývá *hranová konzistence* (AC - arc consistency). Je patrné, že při odstraňení hodnot porušujících GAC konzistenci z aktuálních domén proměnných dostaneme ekvivalentní CSP problém (ekvivalentní vzhledem k množině řešení). Algoritmy hledající nekonzistentní hodnoty vzhledem k určité podmínce se nazývají *filtrační algoritmy* pro podmínku. Dalšími druhy konzistence jsou například konzistence vrcholová (NC - node consistency), okrajová (BC - bound consistency) a po cestě (PC - path consistency). NC odpovídá zahrnutí unárních podmínek přímo do domén proměnných. BC je slabší než AC - v něm pouze požadujeme, aby hodnoty na okraji domény proměnné x - nejmenší hodnoty $lb(x)$ a největší hodnoty $ub(x)$ byly konzistentní vzhledem k definici GAC. Rozdíl významu těchto dvou definic je patrný z následujícího příkladu: $x \in \{1, 2, 3\}$, $y \in \{2\}$ a podmínka $x \neq y$. Zatímco je podmínka BC konzistentní, AC konzistentní není. V PC uvažujeme více podmínek najednou, a proto je silnější než třeba AC. Ovšem dosažení vyššího stupně konzistence je více výpočetně náročné. Proto bývá nejčastěji používáno pouze AC nebo BC. Pro podrobnější diskuzi o konzistenčních technikách viz [1].

Metody konzistence nám tedy přináší jakési odvozování aktuálních domén proměnných z již udělaných rozhodnutí. Obecně nezaručují nalezení řešení ani že v případě dosažení konzistence řešení existuje, ale omezují velikost prohledávaného prostoru. Je tedy očekáváno, že řešení bude nalezeno dříve. Prohledávací algoritmy jsou často implementovány tak, že po celou dobu udržují AC. Tento způsob se nazývá *MAC* (maintaining arc consistency). Na začátku je vytvořena pro všechny podmínky AC. Poté jsou po každém ohodnocení proměnné spuštěny filtrační algoritmy pro podmínky obsahující danou proměnnou. Často je pro každou podmínku dosažení určitého stupně konzistence různě náročné, je ovšem pro jednotlivé podmínky možné mít různé druhy konzistence. Na následujících řádcích bude vysvětlen základní princip udržování konzistence a poté rozšířen pro složitější případy.

Jakým způsobem lze tedy dosáhnout určité konzistence? Pro jednoduchost uvažujme problém pouze s binárními podmínkami, pro které chceme dosáhout AC. Přiložené algoritmy Revise (algoritmus 2), MakeACConsistent (algoritmus 3) a MakeInitialyConsistent (algoritmus 4) ilustrují, jak lze dosáhout AC. Označme c^k_{ij} určitý filtrační algoritmus pro podmínku c^k, který z domény proměnné x_i redukuje doménu proměnné x_j. Algoritmus Revise s pomocí filtrační procedury odebere z domény proměnné x_j všechny nekonzistentní hodnoty s aktuální doménou x_i a vrátí hodnotu, zda byla nějak doména změněna. Propagační algoritmy jsou typicky směrové - z domény jedné proměnné dedukují hodnoty jiné proměnné. Ovšem podmínky směrové nejsou. Proto pro binární podmínku máme zpravidla dvě filtrační procedury, a je tedy třeba použít algoritmus Revise ve dvojicích. Opakování algoritmu Revise(c^k_{ij}, x_i, x_j), Revise(c^k_{ji}, x_j, x_i), které je prováděno, dokud alespoň jedno volání vrací, že byla doména změněna, udělá podmínku c^k AC konzistentní. Vstupem Algoritmus MakeACConsistent je fronta podmínek (filtračních algoritmů), která obsahují proměnné s doménami, jež je potřeba překontrolovat k dosažení AC. V případě inicializace jsou to všechny filtrační algoritmy (algoritmus MakeInitialyConsistent), jinak jsou to filtrační algoritmy podmínek obsahující právě ohodnocenou proměnnou. Pokud dojde ke změně domény proměnné, může dojít k porušení AC pro jinou podmínku, a je tedy třeba znovu zkontrolovat všechny podmínky obsahující danou proměnnou (řádek 7). Tento algoritmus se v literatuře nazývá AC3. Existují i jiné verze AC1, AC2, AC4, AC5, atd. [13]. Ovšem v praxi je AC3 nejpoužívanější.

Proceduru Revise je možné přizpůsobit pro každou podmínku a tím pro jednotlivé podmínky dosáhnout různých stupňů konzistence. Koncept fronty filtračních algoritmů je také snadno rozšiřitelný pro podmínky s více proměnnými.

Algoritmus 2: Revise(c_{ij}^k, x_i, x_j)

Input: c_{ij}^k : filtrační algoritmus pro podmínku c^k pro redukci domény proměnné x_j z domény proměnné x_i

Output: vrátí true, pokud se doména proměnné x_j změnila, jinak vrátí false

```
1  deleted ← false
2  for each value v_j ∈ domain(x_j) do
3      toDelete ← true
4      for each value v_i ∈ domain(x_i) do
5          if c_ij^k(v_i, v_j) je splněna then
6              toDelete ← false
7              break
8      if toDelete then
9          domain(x_j) ← domain(x_j) \ {v_j}
10         deleted ← true
11 return deleted
```

Algoritmus 3: MakeACConsistent(Q, X, C)

Input: Q : fronta podmínek, které je třeba udělat AC, X : množina proměnných, C : p

Output: Každou podmínku udělá AC nebo vyprázdní doménu některé proměnné a skončí tedy neúspěchem

```
1  while not Q empty do
2      c_ij^k ← Q.front
3      Q.popFront
4      if Revise( c_ij^k, x_i, x_j ) then
5          if domain(x_j) == ∅ then
6              return fail
7          Q ← Q ∪ {c_jk ∈ C}
                   /* přidáme do fronty filtrační algoritmy, které
           odvozují hodnoty domén z hodnot domény x_j */
```

Algoritmus 4: MakeInitialyConsistent(X, C)

Input: X : množina proměnných, C : množina podmínek

Output: vytvoří konzistenci pro všechny vstupní podmínky

1 $Q \leftarrow \{c_{ij}^k \in C\}$

2 makeACConsistent(Q)

Ve frontě jsou filtrační algoritmy pro jednotlivé podmínky. Jakmile se změní hodnota domény některé proměnné, jsou všechny filtrační algoritmy, které odvozují z domény této proměnné hodnoty ostatních domén, přidány do fronty ke zkontrolování. Výsledek nezávisí na tom, v jakém pořadí se filtrační algorimy provádějí, lze je tedy libovolně uspořádat. Bývají tudíž uspořádány od časově nejméně nákladných k nejvíce nákladným. Následkem tohoto uspořádání je, že pokud se některá doména vyprázdní z důvodů rychlejšího filtračního algoritmu, není třeba vůbec provádět výpočetně náročnejší operaci. Více o praktické implementaci těchto algoritmů je možné nalézt v příloze A, kde se věnujeme implementaci řešiče podmínek, který je součástí této práce.

1.2.4 Prohledávací strategie

V algoritmu Label (algoritmus 1) jsou dva body, které určují celkovou efektivitu hledání řešení. Prvním z těchto bodů je výběr proměnné na řádku 3 a druhým je výběr hodnoty pro tuto proměnnou na řádku 4. Je patrné, že při optimálních volbách bude řešení nalezeno bez navracení a pro špatné volby nemusí být v rozumném časovém intervalu vůbec žádné řešení nalezeno. Ohodnocování proměnných může být statické a nebo dynamické. Statické je určeno na začátku algoritmu a v průběhu se nemění. Dynamické ohodnocování je založeno na informaci, která se v průběhu algoritmu mění - například velikost aktuální domény proměnné. Typicky je pořadí ohodnocování proměnných a výběr hodnot závislý na konkrétním problému, a je proto nutné definovat vlastní prohledávací strategii. Existují ovšem všobecně platné heuristiky, které v praxi potvrdily svoji účinnost. Výběr proměnné se typicky řídí principem *fail-first* tedy vyber proměnnou, kterou se bude nejhůře ohodnocovat. Tento princip říká, že pokud toto ohodnocení nepovede k řešení, je dobré to zjistit co nejdříve. Proměnná, která je špatně ohodnotitelná, může být například proměnná s nejmenší aktuální doménou (takzvaná DOM heuristika) nebo proměnná, která je svázána nejvíce podmínkami (takzvaná DEG heuristika). Výběr hodnoty se naopak řídí heuristikou *suceed-first* - metodou prvního úspěchu. Tato heuristika říká: vyber tu hodnotu, která nejspíše povede k

řešení. Vzhledem k tomu, že pro vybranou proměnnou bude třeba ozkoušet stejně všechny hodnoty, snaž se tedy vybrat hodnotu, která nejspíše povede k cíli.

Na prohledávání s využitím konzistenčních technik se lze také dívat i jiným způsobem. Po počátečním vytvoření konzistence je nalezen fixpoint, tedy bod, kdy se aktuální domény proměnných stabilizovaly. V tento okamžik je třeba k vyřešení udělat nějakou akci, která omezí aktuální domény proměnných. Do této chvíle jsme uvažovali pouze akci ohodnocení proměnné, ovšem toto není nic jiného než přidání podmínky, že proměnná se rovná konkrétní hodnotě. Je tedy možné rozšířit akci na přidání libovolné podmínky a touto podmínkou omezit současný problém. Sekvencí přidávaných podmínek je možné vést prohledávací strategie. Je tedy například možné místo větvení pomocí disjunkcí ve tvaru $x = v_1 \vee x = v_2 \ldots \vee x = v_n$ větvit například pomocí bisekce intervalů - tedy omezení $x \leq v_i \vee v_i < x$. V přiloženém řešiči je možné použít libovolnou z těchto strategií.

Praktické problémy mají typicky tak velké prostory možných ohodnocení, že není možné je všechny prohledat. Bývají proto často prohledávany pomocí neúplných prohledávacích strategií vycházejících z algoritmů prezentovaných na předchozích řádcích. Tyto algoritmy jsou někdy také nazývány *nesystematické*. Mezi zástupce těchto algoritmů patří LDS (limited discrepancy search), DDS (depth bounded discrepancy search), DBS (depth bounded search) a další. Těmito metodami se budeme zabývat v kapitole 3, ve které budeme popisovat řešení rozvrhovacího problému.

1.2.5 Objektivní funkce

Pro optimalizační problémy je do modelu přidaná proměnná z, která je rovna hodnotě objektivní funkce. Bez újmy na obecnosti uvažujme minimalizační problém. Jednou z možných variant hledání optimálního řešení je převést tento problém na sérii rozhodovacích variant problému [2]. Vždy nějak omezíme hodnotu z a budeme se ptát, jestli řešení stále existuje. Můžeme třeba iterovat dosazení hodnot proměnné z od nejmenší po největší nebo naopak. Pro úspěch optimalizačních problémů je důležité, aby se hodnota objektivní funkce co možná nejsilněji propagovala do domén proměnných a tím omezovala prohledávaný prostor. Jakmile je nalezeno nějaké řešení, je do modelu přidána podmínka, že každé další řešení musí být lepší než současné nalezené. Propagace podmínek nám zaručí, že podstromy, ve kterých se nemůže nalézat lepší řešení, nejsou prohledávány, a tudíž jsou akceptována pouze řešení, ve kterých je hodnota objektivní funkce lepší. Tento druh prohledávání je metoda mezí a větví (branch and bound) [13]. Úspěch tohoto algoritmu závisí na včasném nalezení dobrého řešení (tedy prohledávací heuristi-

ce). Často je nalezeno dobré řešení brzy, ovšem dlouho trvá důkaz optimálnosti. Naštěstí často stačí nalézt dostatečně dobré řešení a nikoliv nejlepší.

Kapitola 2

Rozvrhovací problém

2.1 Co je rozvrhování?

Rozvrhování [2] bývá definováno jako přiřazení zdrojů a časů aktivitám tak, aby bylo dosaženo určitých cílů a všechny podmínky byly splněny. Každá aktivita ke svému vykonání potřebuje určitý(é) zdroj(e) a trvá nějakou dobu. Zdroje zpravidla mívají omezenou kapacitu, která nesmí být v žadném časovém okamžiku překročena.

Vstupem rozvrhovacího problému je tedy množina aktivit (někdy nazývaných operacemi) s požadavky na potřebné zdroje. Aktivitou myslíme nějakou akci, která je potřeba vykonat. Může jí být nějaká část výrobního procesu nebo třeba provedení přednášky ve škole. Zdrojem může být například místo, kde se daná aktivita provede, nebo pracovník, který jí provádí. Takový zdroj tedy mívá omezený počet aktivit, které může najednou provést. Tento počet nazýváme *kapacita* zdroje. Mezi aktivitami můžou být další podmínky. Asi nejčastější bývají časové. Ty mohou například určovat, že aktivita může začít až po skončení nějaké jiné aktivity nebo že musí začít ihned po jiné. Pro každou aktivitu může být také definován nejdřívější časový okamžik, kdy může začít, a také nejpozdější čas, do kterého musí skončit. Aktivity mohou být *nepřerušitelné* (non-preemtive scheduling) a *přerušitelné* (preemtive scheduling). Nepřerušitelné aktivity musí být od svého startovního do svého koncového času prováděny na potřebných zdrojích. Přerušitelné aktivity je možné po nějaké době provádění pozastavit a nechat provádět jinou aktivitu, než jsou dokončeny. Nepřerušitelné aktivity mají *dobu zpracování* - minimální čas, po který musí být vykonávány. Od okamžiku startu do okamžiku konce blokují určitou kapacitu zdroje. Přerušitelné aktivity během svého vykonávání potřebují určitou *energii* zdroje. Jsou dokončeny poté, co energie alokovaná zdroji je rovna energii potřebné k jejich vykonání. Podle povahy

zdrojů rozdělujeme problémy na *disjunktivní* (disjunctive scheduling) a *kumulativní* (cumulative scheduling) [2]. V disjunktivních problémech se vyskytují pouze zdroje kapacity 1, často nazývané *stroje* (machines, unary resource). Tyto zdroje mohou provádět pouze jednu aktivitu v čase. V kumulativních problémech je možné, aby zdroj prováděl více aktivit najednou.

Rozvrhem potom rozumíme přiřazení přesných startovních a koncových časů aktivitám tak, aby v žádném časovém okamžiku nebyly překročeny kapacity zdrojů, aktivity byly přiřazeny ke zdrojům a rovněž všechny další podmínky byly splněny. Mezi tyto další podmínky může patřit třeba požadavek, že v nějaký časový okamžik je zdroj nedostupný.

Pro automatický rozvrhovací systém je také důležité definovat objektivní funkci, pomocí které by bylo možné porovnávat nalezené rozvrhy a určovat, který z nich je lepší. Obvyklou objektivní funkcí bývá délka celého rozvrhu (makespan). Ovšem existují i jiné objektivní funkce. Například vážený součet aktivit, které končí později, než je požadováno v zadání. Případně přidání *slabých* (soft constraints) podmínek určujících peference pro požadovaný rozvrh. Objektivní funkcí potom může být maximalizace počtu splněných preferenčních podmínek.

V literatuře existují různé druhy rozvrhování, nejčastěji zkoumaným je plánování výroby (job-shop scheduling) [9]. Existují různé varianty tohoto problému. Často přidávanými speciálními podmínkami bývají potřebné časy na převezení výrobků z jedné výrobny do jiné, takzvané setup časy. Dalším rozvrhovacím problémem je školní rozvrh. Ten je například možné převést na obarvování grafu.

2.2 Jak je možné popsat rozvrhování jako CSP?

Vzhledem k problému rozvrhování pozemních operací na letišti se budeme v této sekci zabývat pouze nepřerušitelnými kumulativními aktivitami s jednotkovými požadavky na zdroje. Čas budeme uvažovat v oboru přirozených čísel. Vždy je možné nalézt dostatečně jemné dělení. Označme A množinu všech aktivit, R množinu všech zdrojů. Vstupem jsou nejdřívější startovní časy, nejpozdější koncové časy a nejkratší prováděcí časy jednotlivých aktivit - hodnoty $r_i, d_i, p_i, i \in A$. Každá aktivita i navíc vyžaduje r_{ik} jednotek zdroje $k \in R$. Tyto hodnoty jsou v našem případě tedy buď jedna nebo nula.

Pro každou aktivitu a je možné vytvořit dvě proměnné s_a a e_a označující startovní a koncový čas aktivity. Aktivita je vykonávána v intervalu $[s_a, e_a)$. Označme množinu aktivit vykonávaných v čase t jako $active(t)$. Dodatečné podmínky určující pořadí prováděných aktivit (takzvané *precedenční podmínky*) označme

P. Pomocí těchto podmínek je například možné určit, že některá aktivita musí skončit dříve, než jiná aktivita začne. Poté lze rozvrhovací problém zapsat jako:

Definice 3

$$
\begin{aligned}
& s_i \in [r_i, d_i - p_i], && i \in A && (1)\\
& e_i \in [d_i - p_i, d_i], && i \in A && (2)\\
& e_i - s_i \geq p_i, && i \in A && (3)\\
& s_i + l_{ij} \leq s_j, && l_{i,j} \in P_s && (4)\\
& e_i + l_{ij} \leq s_j, && l_{ij} \in P_e && (5)\\
& \textstyle\sum_{i \in active(t)} r_{ik} \leq capacity_k && \forall t \in N, \forall k \in R && (6)
\end{aligned}
$$

Podmínky (1) a (2) přímo vycházejí ze zadání. Podmínky (3) zaručují, že aktivity budou prováděny alespoň po dobu svých minimálních prováděcích časů. Podmínky (4) a (5) jsou precedenčními podmínkami. Pokud bychom například chtěli říci, že aktivita a musí skončit dříve, než začne aktivita b, bude v podmínkách P_e hodnota l_{ab} rovna 0. Podmínky (6) nám zaručují, že v žádném časovém okamžiku není překročena kapacita zdroje. Právě podmínky zdrojů dělají rozvrhovací problém obtížným. Pro propagování těchto podmínek je potřeba speciálních filtračních algoritmů, které jsou předmětem aktivního výzkumu [7], [4]. Některé z existujících algoritmů byly implementovány v rámci této práce. Ostatní podmínky jsou pouze lineárního tvaru a jako takové snadno a efektivně propagovatelné.

Tato definice se trochu liší od klasické definice objevující se v literatuře [5], [6]. Je upravena pro pozdější lepší využití pro definici problému na letišti. Pokud vypustíme podmínky (5), omezíme hodnoty l_{ij} pouze na nezáporné hodnoty a u podmínky (2) nahradíme nerovnosti rovnostmi, obdržíme klasickou instanci výrobního problému (job-shop). Pokud hodnoty l_{ij} mohou nabývat i záporných hodnot, jedná se o takzvané rozvrhování s časovými okny (Scheduling with Time Windows).

Kapitola 3

Aplikace rozvrhování na letišti - model a řešení

V této kapitole ukážeme, jak lze využít metody rozvrhování pro koordinaci pozemních operací na zjednodušeném modelu letiště. Pozemní operace byly vybrány z toho důvodu, že jsou klíčovou oblastí pro fungování letiště a jejich dobré využití má velký vliv na ostatní součásti letištního provozu.

3.1 Současný stav plánování na letištích a existující přístupy

Tento odstavec vychází z velké části z [12]. V současné době jsou za plánování pozemních operací zodpovědní nejméně tři lidé - předletový manažer, taxi manažer a manažer ranvejí. Tento úkol je i za normálních podmínek obtížný. Musí brát do úvahy velké množství omezujících podmínek, které zaručují bezpečný plán (rozvrh). Každý z nich se snaží navrhnout optimální plán pro sféru svého vlivu. Manažer ranvejí, jakožto poslední v řadě, už má velmi malé možnosti, jak něco změnit. Přitom, jak plyne téměř z každého článku o letištích, je ravej klíčovou oblastí pro celkovou optimalitu plánu. Tento poznatek je založen na tom, že na ranveji dochází k velké řadě separačních podmínek. Tyto podmínky jsou dané například váhovou kategorií letadel nebo separačními vzdálenostmi ve vzduchu. Další z důvodů je ten, že ranveje se často kříží ať již s taxi cestami, tak s jinými ranvejemi. V této práci tedy v první řadě navrhneme sekvenci letadel na ranveji a teprve poté se je snažíme od jejich stanovišť dopravit včas na daná startovní místa. Tímto spojíme povinnosti taxi manažera s manažerem ranvejí a budeme očekávat, že celková efektivita tímto vzroste.

Literatury o rozvrhování na letišti existuje celá řada. V práci [5] je řešen

problém přidělování bran jednotlivým letadlům pomocí CSP. Problém je zde převeden na rozvrhovací problém a jsou porovnána řešení s dosud používanými expertními (tedy pravidlovými a nikoliv optimalizačními) systémy. V práci [17] je řešen problém alokace ranveje jako optimalizační problém využívající vícestupňovou optimalizaci založenou na numerických metodách. Nejbližší metoda řešení je uvedena v [10]. Na ní se budeme v průběhu kapitoly odkazovat a poukazovat na podobné nebo rozdílné rysy řešení. Metody uvedené v [10] byly vyzkoušeny a osvědčeny v praxi například na pražském letišti v Ruzyni.

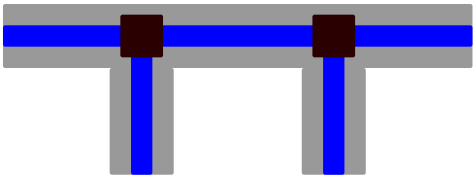
3.2 Zadání problému

Nechť jsou zadané operace, které mají být vykonány. Tyto operace zahrnují přílet, odlet, taxi z místa příletu do místa parkování (bran), taxi z místa parkování do místa odletu a taxi z jednoho místa parkování na jiné místo parkování. Každá z těchto operací má zadaný zdroj, který je potřeba k jejímu vykonání. Základní entitou je let. Ten zahrnuje akce jednoho letadla. Každé letadlo do systému vstupuje v určitém stavu a končí v určitém stavu. Vstupní stav je buď parkování nebo přílet. Výstupním stavem je opět buď parkování nebo odlet. Schématicky je sekvence možných operací znázorněna takto (v závorce je vždy uvedeno, který typ zdroje ke svému vykonání operace potřebuje):

- přílet (ranvej) → taxi z místa příletu (taxi) → parkování (brána)
- parkování (brána) → taxi z místa parkování (taxi) → odlet (ranvej)
- parkování (brána) → taxi z místa parkování (taxi) → parkování (brána)

Jelikož je provoz na letišti velice komplikovaný a dynamický proces, pro účely této práce jsou zvoleny určité zjednodušující předpoklady.

- Letadla jsou až na svoji váhovou kategorii nerozlišitelná - tento předpoklad není nikterak silný, protože na výsledném modelu by nic nezměnil, pouze by udělal vstupní soubory složitější. Má za následek, že se všechna letadla pohybují stejně rychle.
- Pohyby letadel jsou deterministické - tedy víme, jak dlouhou dobu jim bude která akce trvat. Tím, že doby jednotlivých akcí pro rozvrhování trochu nadhodnotíme, můžeme zajistit dostatečně velkou časovou rezervu pro provedení akce.



Obrázek 3.1: Schématické znázornění povrchu letiště

- Pro přílety a odlety bývá přidělována takzvaná SID (Standard Instrument Departure) – trasa, kterou mají použít při počáteční (závěrečné) fázi letu. Tato trasa pro přílety začíná v takzvaném TMA (Terminal Manoeuvring Area) a končí na ranveji. Tyto cesty se mohou křížit. Pro práci uvažujeme, že z jednoho startovacího (cílového) místa vede pouze jedna SID a ty se nekříží. Řešení pro tento problém je v principu stejné jako řešení taxislužeb, které je vysvětleno v části věnované modelování povrchu letiště. Jedním z důvodů pro nezahrnutí modelování těchto skutečností je obtížnost sehnání těchto map.

Druhy podmínek, které omezují možné rozvrhy jsou následující:

1. Během vzletu nebo příletu nesmí být žádná část ranveje využívána pro taxi operace nebo jiné vzlety či přílety.
2. Letadlo musí mít v každém okamžiku přidělenou nějakou pozici.
3. Letadla se nesmí na taxi cestách předjíždět ani míjet v protisměru.
4. Separační podmínky z důvodu mixování příletů a odletů na stejné ranveji.
5. Separační podmínky pro rozvrhnutí dvou letů různých váhových kategorií na stejné ranveji.

Pro vyjádření všech těchto podmínek je potřeba vytvořit model povrchu letiště.

3.3 Vytvořený model povrchu letiště

Povrch letiště je modelovaný jako graf, kde vrcholy odpovídají křižovatkám a hrany odpovídají spojnicím těchto křižovatek. Schématicky je toto znázorněno na obrázku 3.1. Černé obdélníky odpovídají křižovatkám (vrholům grafu) a modré jejich spojnicím. Hrany i vrcholy odpovídají zdrojům. Tyto zdroje budeme nazývat *křižovatkové* a *hranové* zdroje. Křižovatky jsou zdroji kapacity 1. Tím zajišťují, že se letadla na jejich pozici nemohou srazit. Křižovatkové zdroje můžeme nazývat logickými zdroji, protože nerepresentují pohyb letadel, ale pouze zajišťují potřebné

rozestupy mezi letadly. K pohybu letadla je potřeba využít hranových zdrojů. Ty mají určitou délku representující vzdálenost na letišti. Tato délka je podle rychlosti letadla převedena do minimálního časového okamžiku, který je pro letadlo potřeba k překonání této vzdálenosti (nejkratší době vykonání). Označme doby, které jsou třeba k překonání vzdáleností mezi křižovatkou i a j, jako $t_{i,j}$. Ze vzdálenosti cesty mezi dvěmi křižovatkami je také odvozena kapacita zdroje - neboli počet letadel, která mohou jet za sebou po stejné hraně. Pohyby letadel jsou koordinovány buď pomocí radia nebo v dnešní době již častěji světelnými signály. Pomocí těchto technik je možné letadla dopravovat z jednoho místa na druhé a před křižovatkami je zastavovat.

Po zvolení této representace letiště je možné definovat, jakým způsobem jsou akce letadla implementovány.

3.3.1 Pohyb po taxicestách

Pohyb letadla po povrchu z určité pozice do jiné pozice je sekvence požadavků na zdroje podél cesty z původního místa do cílového. Skládá se tedy ze střídajících se křižovatkových a hranových zdrojů. Označme start této akce jako proměnnou s_0. Poté pro každou křižovatku po cestě vytvoříme proměnnou určující, kdy letadlo touto křižovatkou projíždí. Označme tyto proměnné $s_1, \ldots, s_n$. Letadlo nemůže cestu z křižovatky i na křižovatku $i+1$ urazit rychleji, než je minimální doba potřebná k překonání hranového zdroje mezi křižovatkou i a $i+1$ - hodnota $t_{i,i+1}$. Z tohoto důvodu zahrneme do systému podmínky (I), že $s_i + t_{i,i+1} \leq s_{i+1}$.

Cestu tedy budeme representovat jako sekvenci aktivit $a_1, \ldots a_{n-1}$, kde každá využívá hranový zdroj $i, i+1$ v intervalu $[s_i, s_{i+1}]$. Jako minimální separační doba na křižovatce bude zvolena konstanta c. Ke svému pohybu letadlo potřebuje také pro všechna $i \in \{1, 2, \ldots, n\}$ křižovatkové zdroje v době $[s_i - c/2, s_i + c/2]$. Pokud letadlo projíždí křižovatkou plnou rychlostí a není omezováno žádným dalším provozem, nemají křižovatky žádný vliv na rychlost pohybu. V případě, že je letadlo donuceno čekat před křižovatkou (například protože je zabrána pro přistávání), křižovatkové zdroje zaručují dodržování bezpečných vzdáleností a simulují pomalejší zrychlování letadla do taxi rychlosti. Po dobu čekání nám podmínky (I) stále vynucují, že je zdroj v této době používán.

Aby se letadla nemohla navzájem předjíždět, je pro každé dvě aktivity prováděné na jednom zdroji vynuceno, že pokud jedna aktivita začala dříve než druhá, musí i dříve skončit - tedy pro každé dvě aktivity (a, b) na zdroji mezi křižovatkami i, j je dána podmínka $s_{a_i} < s_{b_i} \Rightarrow s_{b_j} < s_{b_j}$. Dalším požadavkem je, že se letadla nesmí míjet v protisměru. Proto jsou pro každý hranový zdroj i, j přidány

následující podmínky. Pro všechny aktivity a, b v množině aktivit pro hranový zdroj i, j musí být alespoň jedna z následujících podmínek splněna.

1. $a == b$
2. aktivita a používá hranový zdroj i, j ve směru $i \rightarrow j$ a aktivita b používá hranový zdroj i, j ve směru $i \rightarrow j$
3. aktivita a používá hranový zdroj i, j ve směru $i \rightarrow j$ a aktivita b používá hranový zdroj i,j ve směru $j \rightarrow i$ a $s_{a_j} <= s_{b_j} \vee s_{b_i} <= s_{a_i}$

Důsledkem podmínky 3 je, že buď aktivita a musí skončit před aktivitou b, nebo aktivita b musí skončit před aktivitou a. To zaručuje, že není možné, aby dvě aktivity, které využívají zdroje v opačném směru, využívaly tento zdroj ve stejném časovém okažiku – tedy aby se letadla jedoucí proti sobě na zdroji potkala. Tato podmínka se v literatuře nazývá *disjunktivní podmínkou* [2].

Po zvolení této representace bylo zjistěno, že v článku [10] je zvolena velmi podobná representace. Rozdíl v řešení je hlavně v tom, že v jejich případě se pohyb letadla používající hranový zdroj skládá ze dvou aktivit. První aktivita je stejná jako v našem případě - zajišťuje pohyb letadla po hranovém zdroji. Druhá je aktivita blokování – blokuje celou kapacitu zdroje v opačném směru a tím zajišťuje, že zdroj nemohou letadla používat zároveň v obou směrech. Z toho ovšem plyne, že ztratíme možnost, aby více letadel mohlo jet za sebou po stejném zdroji, ačkoliv to byl jeden z předpokladů v článku. Tento aspekt není v článku dostatečně detailně vysvětlen. V práci jsou pro vyjádření této skutečnosti využity disjunktivní podmínky.

3.3.2 Ostatní aktivity

Aktivita vzletu nebo příletu je definovaná jako jedna aktivita se startovním časem s a koncovým časem e. Ranvej je representováná jako množina hranových a křižovatkových zdrojů na ní se nacházejících. Pro vzlet nebo přílet je potřeba na určitou dobu zablokovat všechny tyto zdroje. To je provedeno tak, že aktivita vyžaduje po celou dobu intervalu $[s, e)$ plnou kapacitu od všech zdrojů na dané ranveji. Standardní doba je jeden a půl minuty [12]. Přílety a odlety mohou vždy začínat po půl minutě. Těmto časovým okamžikům se říká *sloty*. Každé letadlo patří do určité váhové kategorie. Rozlišujeme dvě váhové kategorie. Letadla malá a velká. V situaci, kdy po sobě vzlétají nebo přistávají dvě letadla různých váhových kategorií a druhé z nich je v nižšší váhové kategorii, je vždy potřeba zvětšit klasický rozestup na vyšší, například na dvě minuty. To je dáno

tím, že turbulence vzniklé po těžším letadle nejsou bezpečné pro lehčí letadlo. Tato podmínka je v modelu representována jednoduchou lineární nerovností mezi starty různých letů v závislosti na váhové kategorii.

Vzhledem k přijatým předpokladům se letadly ve vzduchu se nezabýváme. Proto jsou separační podmínky z důvodu mixování příletů a odletů na stejné ranveji fixovány na konstantní hodnotu (například 5 minut). Minimálně tento časový interval musí uplynout mezi naplánováním dvou aktivit těchto různých druhů. Tato podmínka je v modelu representována lineární podmínkou mezi starty jednotlivých aktivit požadujících stejné ranveje v závislosti na druhu aktivity.

Možný model pro řešení pohybu letadel ve vzduchu je téměř ekvivalentní modelování taxislužeb po povrchu. Místa, kde by se trasy protínaly, by měla stejný význam jako křižovatky v taxislužbách. Celá trasa by poté byla zdrojem s kapacitou 1. Separační podmínky na závěru trasy by byly vynuceny metodou křižovatkových zdrojů.

Brána je zdroj kapacity jedna. Do času stráveného v bráně jsou započítávány přípravy, než se letadlo dostane na taxicesty. Zde začíná na křižovatce, která už spadá pod rozvrhování taxicest.

K dosažení požadavku, že letadlo musí mít v každém okamžiku přidělenou nějakou pozici, stačí přidat do systému podmínky, že konec jedné aktivity letadla se rovná začátku jiné. Tím letadlo v každém okamžiku využívá některý ze zdrojů, který jednoznačně určuje jeho pozici.

3.4 Požadavky na aplikaci

Požadavky na pozemní operace nejsou nikdy známy dlouho dopředu, proto jsou rozvrhovány vždy pouze v krátkém nebo středně dlouhém časovém horizontu - ve výhledovém okně 5 až 20 minut. Uživatel aplikace zadá množinu letů, které je potřeba rozvrhnout. Každý z letů je jednou ze sekvencí, které byly uvedeny v části 3.2. Uživatel také zadá, které zdroje (taxicesty, ranveje, brány) mají být k vykonání aktivit použity. Uživatel navíc může zadat požadavek, že některý let na ranveji musí začínat dříve než jiný, aby mohl ovlivnit výsledný rozvrh. Každá z jednotlivých aktivit letů může mít časové okno, ve kterém musí být provedena – tedy kdy nejdříve může začít a kdy nejpozději musí skončit. Pokud toto okno není zadáno, je považované za flexibilní a automaticky upraveno na hodnoty 0 až horizont. Časová okna jsou důležitá hlavně pro přílety a pro odlety. Pro odlety to jsou takzvané CFMU (Central Flow Mangement Unit) - hodnoty přidělované jednotlivým letům na základě centrálního registru v Bruselu. Jsou přidělovány

centrálně z jednoho místa, aby zaručovaly, že jednotlivé letové sektory v Evropě nejsou přehlcené letadly.

Hlavním požadavkem na výstup systému je, aby každý vygenerovaný rozvrh byl *bezpečný*, tedy aby splňoval všechny podmínky definované výše. Dalším požadavkem je, aby výsledky produkoval pokud možno rychle. Čas hledání proto je konfigurovatelný. Navíc je možné program kdykoliv zastavit a vrátit zatím nejlepší nalezený výsledek. Pokud je zadaný problém neřešitelný nebo není nalezeno žádné řešení v zadaném čase, nebude se program snažit naplánovat co nejvíce aktivit, ale oznámí uživateli, že problém nelze vyřešit. Důvodem je, že pro automatický systém je těžké rozpoznat, které aktivity je možné zpozdit a které nikoliv, protože neznáme všechny uživatelovy preference. Proto rozhodnutí, kterou z aktivit opozdit nebo odstranit, je necháno na uživateli. Uživatel z výstupu pozná, zda se problém nepodařilo vyřešit, či zda je neřešitelný. Musíme také určit kritéria, podle kterých budeme rozvrhy poměřovat. Základním kritériem bude čas rozvrhu, tedy maximum z koncových časů všech aktivit. Abychom zatěžovali povrch letiště co nejméně, bude druhotnou objektivní funkcí globální doba potřebná k taxi operacím - tedy součet doby trvání všech taxioperací. Vždy budeme nejprve preferovat první kritérium a až poté druhé.

3.5 Přístup k řešení

3.5.1 Prohledávací strategie

Jelikož je potřeba, aby odpovědi systému byly co nejrychlejší, není využíváno úplného systematického prohledávání. Na druhou stranu požadavek, že každé vrácené řešení musí být bezpečné a splňovat všechny podmínky, má za důsledek, že se klasické metody lokálního prohledávání nejeví jako dobrý kandidát. Zaměříme se tedy na metody nesystematického prohledávání. Bude vysvětleno, jak tyto metody fungují, a jak je možné jich využít v tomto případě.

Budeme využívat variantu stromové prohledávací metody. Tato je v optimalizačním systému splňování podmínek charakterizována několika aspekty.

- Větvící strategie s propagováním omezujících podmínek je aplikována v každém vrcholu prohledávaného stromu. Propagace podmínek omezuje prohledávaný prostor a zabraňuje prohledávání prostorů, kde nemůže být nalezeno řešení. Redukuje tedy startovní a koncové časy aktivit. Na druhou stranu složité propagační procedury trvají určitý čas a nemusí vždy vyvážit své přínosy.

- Větvící strategie je charakterizována druhem rozhodnutí, která jsou udělána v každém kroku. Mohou být různého typu. Pro rozvrhovací problémy je typické, že postupně staví rozvrh od počátečního času do koncového času. V každém okamžiku mají částečný rozvrh, který se snaží rozšírit do úplného. V disjunktivních problémech bývá například rozhodnuto, která z aktivit na zdroji bude provedena dříve. Další možnou strategií je bisekce startovních a koncových časů. Obecnou strategií je vybrat aktivitu z dosud nerozvrhnutých aktivit a pokusit se jí rozvrhout co nejdříve nebo jí o nějakou dobu pozdržet.

- Heuristická funkce – určuje, kterou z možností vyzkoušet dříve.

- Procedura mezí, která na základě již nalezených řešení zmenšuje prohledávací prostor a zabraňuje pokračování v prohledávání z vrcholů, jejichž podstromy nemohou obsahovat lepší řešení než nejlepší dosud nalezené.

Jedná se o optimalizační problém, a proto budou všechny metody využívat metodu větví a mezí (branch and bound). Pro objektivní funkci je vytvořena proměnná, která je rovna její hodnotě. Pak je tuto metodu snadné zabudovat přímo do systému podmínek, protože se jedná o jednoduchou lineární nerovnost. Propagování podmínek již automaticky zařídí vše potřebné. V našem případě optimalizujeme dvě proměnné - celkový konec operací (proměnná *makespan*) a součet taxi časů (proměnná $globalTaxiTime$). Jedná se o *multikriteriální optimalizaci*. Bylo ovšem zvoleno jednoduché porovnání dvou výsledků. Je preferován vždy celkový konec operací. To znamené, že dvojice $(99, 300)$ je lepší než $(100, 200)$. První číslo ve dvojici representuje celkový konec operací, druhé součet taxi časů. Dosud nejlepší nalezené řešení je propagováno do hodnot objektivní funkce ve dvou krocích:

1. Vždy omezme hodnotu maximálního konce operací na menší nebo rovnu dosud nejlepší nalezené – po každém nalezení řešení s hodnotou m je do systému podmínek vždy přidána podmínka $makespan \leq m$, hodnotu součtu taxičasů pro řešení m označme t.

2. Pokud nemůže být hodnota maximálního konce operací lepší než dosud nalezená, omezme hodnotu součtu taxi časů na menší než nejlepší dosud nalezenou pro nalezenou hodnotu maximálního konce operací – po každém nalezení řešení je do systému podmínek přidána podmínka $makespan = m \rightarrow globalTaxiTime < t$.

Pvní bod zaručuje, že hodnota *makespan* nemůže být větší než dosud nejlepší nalezená (m). Pokud může být hledaná hodnota menší než nejlepší řešení dosud nalezené, není druhý bod nijak propagován. Až v momentě, kdy je dokázáno, že řešení s lepší hodnotou proměnné *makespan* nelze nalézt, je hodnota pro globální součet taxi časů omezena – musí být lepší než dosud nejlepší nalezená pro hodnotu m.

Pro propagaci podmínek v každém vrcholu prohledávacího stromu byly implementovány a použity standardní algoritmy pro propagování aritmetických podmínek. Pro propagování zdrojových podmínek byly použity a implementovány algoritmy *edge-finding, not-first/not-last, propagation-precedense, isOverloaded* [7] pro zdroje kapacity jedna a pro kumulativní zdroje byly použity a implementovány algoritmy *calculateUB, calculateLB* navržené v [4]. Tato práce se popisem těchto algoritmů nezabývá, protože je mimo rozsah této práce. Tyto algoritmy jsou dobře zdokumentovány v literatuře, ze které byly převzaty.

3.5.2 Větvící strategie

V této části bude nejdříve ukázán základní navržený algoritmus pro řešení zadaného problému. Poté bude na základě tohoto algoritmu vytvořen heuristický algoritmus, který dovoluje většinou nalézt poměrně dobré řešení v krátkém čase.

Nejdříve bude zopakováno a upřesněno co je vstupem a cílem. Vstupem jsou pro všechny operace letů proměnné s, e určující startovní a koncové časy letadel na jednotlivých zdrojích. Například pro letadlo procházející systémem způsobem – přílet → taxi z místa příletu → parkování – máme proměnné start příletu, konec příletu. Konec příletu je roven počátku taxi cesty. Každé křižovatce j na taxi cestě náleží proměnná t_j, která určuje čas, kdy křižovatkou letadlo projíždí. Mezi dvěmi po sobě jdoucími křižovatkami po cestě se letadlo přepravuje po hranovém zdroji. Na tomto hranovém zdroji může strávit více času, než kolik je minimální doba potřebná na přecestování této vzdálenosti. Toto zaručuje, že letadla na sebe mohou čekat. Okamžikem, kdy letadlo je na poslední křižovatce cesty – nejbližší křižovatce k místě brány – už začíná aktivita parkování. Začátek aktivity je roven době na křižovatce a využívá zdroj brány. Cílem je určit hodnoty všech startovních a koncových časů všech aktivit tak, aby byly splněny všechny výše definované omezující podmínky.

K řešení systému podmínek byla použita verze algoritmu 1. K utvoření prohledávací strategie je nejdůležitější uspořádat výběr ohodnocování proměnných – v algoritmu 1 řádek 3. Následně vybrat metodu pro výběr hodnot pro proměnnou vybranou v předešlém kroce – v algoritmu 1 řádek 4.

Pro všechny aktivity stačí ohodnotit startovní časy, protože startovní čas jedné aktivity je roven koncovému času jiné aktivity – tím jsou tedy i koncové časy ohodnoceny. Proměnné je možné uspořádat do skupin podle typů aktivit. Tyto skupiny zahrnují proměnné definující starty a přílety na ranveji a proměnné definující taxi cesty. Proměnné taxi cest je dále možné dělit na proměnné pro taxi cesty pro příjezd k ranveji před vzletem, proměnné pro taxi cesty po přistání a proměnné pro taxi cesty pro přejezd mezi dvěmi parkovacími místy.

Proměnné definující starty a přílety jsou klíčové, protože do velké míry určují hodnoty ostatních proměnných a navíc jsou nejvíce omezené. Jsou tedy dle heuristiky *fail-first* ohodnoceny nejdříve. Výběr jejich hodnot je od nejmenší hodnoty v doméně po největší, protože hledáme rozvrh, který minimalizuje celkový koncový čas.

Po přidělení startovních a příletových časů (slotů) všem letadlům je potřeba dopravit letadla přes síť taxi cest na potřebné pozice. Letadlo, které potřebuje příjezd k ranveji před vzletem, by mělo vyjíždět co nejpozději, aby se dostalo na startovní pozici právě včas. Oproti tomu letadla po příletu a letadla potřebující přejezd mezi dvěmi parkovacími místy vyjíždí co nejdříve, aby byly co nejdříve v cíli. Tedy hodnoty proměnných pro taxi cesty pro příjezd k ranveji před vzletem budou tedy ohodnocovány od největší hodnoty v jejich doméně k nejmenší hodnotě. Ostatní hodnoty proměnných pro taxi cesty jsou ohodnocovány od nejmenší po největší hodnotu. Výběr uspřádání proměnných pro ohodnocení je následující: V každém kroku je vybrána jedna zatím nerozvržená taxi cesta (odpovídající jednomu letadlu). Je vybrána taková, která má nejmenší nejdřívější koncový čas. Pokud má více taxi cest stejný nejmenší nejdřívější koncový čas, je vybrána ta, u které má tento koncový čas menší velikost domény. Následně budou proměnné definující tuto taxicestu ohodnocované v uspořádání dané touto taxicestou (podle pořadí křižovatek, kterými letadlo projíždí). Vždy je tedy vybrána taxicesta a letadlo je po ní navedeno do cílové pozice. Pokud neexistuje nerozvržená taxicesta, jsou všechna letadla ve svém cíli a řešení bylo nalezeno. Tento algoritmus lze shrnout takto:

Algoritmus 5 je *úplný*. Pokud tedy existuje řešení, nalezne jej. Pro praktické využití je ovšem příliš pomalý a není schopen nalézt alespoň ňejaké řešení v krátké době. Proto byly zkoumány alternativy jak jej zrychlit. První alternativou jsou metody nesystematického prohledávání.

Abychom našli nějaké řešení, jsou používány heuristiky (výběr hodnoty pro proměnnou), jež nás vedou do míst prohledávaného prostoru, která pravděpodobně budou obsahovat nejlepší řešení. Heuristika ovšem není vždy pravdivá – může se

Algoritmus 5: Schedule

Input: Proměnné pro starty a přílety letadel, proměnné pro taxioperace

Output: Nalezne řešení zadaného rozvrhovacího problému, pokud existuje

1. Zkoušej všechna ohodnocení pro startovní časy definující starty a přílety – jako proměnnou pro ohodocování vyber tu, která má nejmenší hodnotu v doméně (může být rozvrhuta nejdříve). Pro vybranou proměnnou zkoušej postupně hodnoty z její domény od nejmenší po největší. Pro každé takové ohodnocení pokračuj krokem 2.

2. Ze všech možných taxicest vyber tu, která má nejmenší nejdřívější koncový čas a není zatím rozvržena. Pokud má více taxi cest stejný nejmenší nejdřívější koncový čas, je vybrána ta, u které má tento koncový čas menší velikost domény. Tato taxi cesta definuje sekvenci $t_1, t_2, \ldots, t_n$ časových okamžiků, které jsou časy průjezdů letadla křižovatkami. Pokud jsou všechny taxicesty rozvrženy, nahlaš řešení a pokračuj zatím nezkoušenou alternativou.

3. Pro vybranou taxicestu postupně ohodnocuj proměnné v pořadí $t_1, t_2, \ldots, t_n$, pro každou takovou proměnnou postupně zkoušej dosazovat hodnoty z její domény od největší po nejmenší v případě taxi cest vedoucích na ranvej, od nejmenší po největší jinak. Po ohodnocení všech proměnných $t_1, t_2, \ldots, t_n$ pokračuj krokem 2.

mýlit. Proto jednou z myšlenek nesystematického prohledávání je, že omezíme počet možných zmýlení. Heuristika nám v každém dalším kroku všechny akce ohodnotí podle toho, která z nich se dle ní jeví jako lepší. Poté je možné tyto akce lineárně uspořádat. Nejdříve vyzkoušíme ty hodnoty, které nám heuristika doporučila jako lepší. Každou hodnotu, která není jako první navržená heuristikou, nazveme *odchýlením* neboli *diskrepancí*, mýlkou. Z tohoto schématu vychází například LDS (Limited Discrepancy Search) [14]. LDS určuje, že se během jednoho prohledávání z kořene do listu dovolí heuristice pouze omezeněkrát mýlit (proto limited). Prohledávání je postupně restartováno se zvyšujícím se počtem možných diskrepancí. Je zastaveno, pokud se počet mýlek rovná určené maximální hodnotě. Dalším z algoritmů vycházejících z podobného schématu je DDS (Depth-bounded Discrepancy Search) [15]. V tomto algoritmu je opět počet možných zmýlení postupně zvyšováno až na maximální hranici. DDS vychází z pozorování, že většina heuristik udělá chyby spíše ze začátku prohledávání než později (protože později už má heuristika více informací). Je tedy omezena možnost dělat chyby pouze do určité hloubky prohledávacího stromu. Všechny hodnoty pod touto hranicí už musí být nejlepšími akcemi doporučenými heuristikou. Při testování se jako lepší alternativa projevil algoritmus DDS. Z tohoto důvodu je ve všech algoritmech zde presentovaných a používajících nesystematického prohledávání využito tohoto algoritmu.

Propojení těchto algoritmů s algoritmem 5 je následující. První fáze (bod 1) algoritmu 5 je nechána beze změny. V druhé fázi je využito nesystematického prohledávání. Za jednu diskrepanci je považováno nevybrání nejlepší hodnoty v doméně proměnné – tedy pokud proměnná měla doménu $[0, 1, 2]$ a vybírali jsme hodnoty od nejmenší po největší, není vybrání hodnoty 0 žádnou diskrepancí. Vybrání hodnoty 1 je jednou diskrepancí a vybrání hodnoty 2 představuje dvě diskrepance. Ačkoliv se nalezení řešení touto metodou podařilo zrychlit, nebyl stále čas odpovědi dostatečně rychlý. Tento algoritmus je ovšem při dostatečném maximálním počtu diskrepancí stále úplný.

Z důvodů rychlosti nalezení řešení nakonec vznikla heuristická metoda, která sice není úplná a negarantuje nalezení optimálního řešení, ale na testovaných příkladech v případě existence řešení nalezla poměrně dobré řešení brzy. První krok algoritmu je opět stejný. Další kroky vycházejí ze skutečnosti, že v algoritmu 5 je třetí krok (postupné ohodnocování proměnných) nejnáročnější. Z tohoto důvodu jej spojí do jednoho kroku – všem proměnným na jedné taxi cestě je v závislosti na tom, jestli je ohodnocujeme od nejmenší nebo od největší hodnoty, dosazena nejmenší (respektive největší) hodnota v doméně dané proměnné. Vždy

je tedy ohodnocena celá taxi cesta najednou. Protože je ale potřeba někdy taxi cestu zpozdit, je v kroku 2 vždy rozhodnuto buď rozvrhout taxi cestu nejdříve (nejpozději) jak je to možné, nebo ji označit jako pozdrženou. Tato taxicesta je pozdržená dokud se nezmění hodnota některé z domén proměnných v ní obsažená (změní se například rozvržením jiné taxi cesty využívající stejné zdroje). V druhém kroku algoritmu 5 jsou pro výběr taxi cesty uvažovány pouze ty, které nejsou pozdržené. Pokud je tato množina prázdná a existují nějaké nerozvržené taxi cesty, algoritmus se vrací k předešlé volbě a zkusí jinou alternativu.

Tuto metodu opět zkombinujeme s neúplným prohledáváním. Diskrepancí je v tomto případě označení taxicesty jako pozdržené.

Abychom zachovali úplnost algoritmu, je úplná i heuristická metoda zkombinována. Nejdříve je spuštěna heuristická metoda a po jejím skončení je spuštěna úplná metoda. Výhodou tohoto přístupu je i to, že pokud heuristická metoda nalezne řešení, je prohledávaný prostor omezen pomocí metody větví a mezí a je tedy pro úplnou metodu snažší jej prohledat.

Závěr

Hlavními výsledky této práce jsou:

- Návrh a implementace řešiče podmínek.
- Vytvoření a popsání modelu pro bezpečné rozvrhování pozemního provozu na letišti pomocí technik programování s omezujícími podmínkami.
- Vytvoření a posání prohledávací strategie pro vytvořený model.
- Implementace aplikace ilustrující předchozí body.

V práci byl popsán a zdůvodněn model pro bezpečné rozvrhování pozemního provozu na letišti a navržen algoritmus, jež takto zadaný problém řeší. Pro názornou vizualizaci nalezených řešení byla vytvořena aplikace, která simuluje nalezený rozvrh jako pohyb letadel po mapě letiště. V této aplikaci je možné kromě simulace výsledků zadávat vstupní problémy a následně je řešit. Aplikace byla důsledně rozdělena do presentační a výpočetní části. Je tedy možné obě dvě části nezávisle na sobě měnit a použít výpočetní část jako funkci z příkazového řádku. Aplikace je napsána v jazyku C++ a je možné ji spustit na operačních systémech Windows a Linux.

Aplikace ke svému výpočtu využívá pro účely práce implementovaný řešič podmínek, který je možné nezávisle využít i v jiných projektech k modelování problémů splňování podmínek. Implementování tohoto řešiče se ukázalo jako nejobtížnější úkol. Plnohodnotný řešič podmínek byl implementován z toho důvodu, že původně vytvořené doménově omezené řešiče se ukázaly jako špatně konfigurovatelné a těžkopádné.

V práci bylo pro složitost letištního provozu nutné zvolit zjednodušující předpoklady. Tyto předpoklady zahrnují, že pohyb letadel je deterministický, letadla jsou až na svoji váhovou kategorii nerozlišitelná a že modelování příletů a odletů ve vzduchu je zjednodušeno.

Byl navržen a implementován neúplný heuristický algoritmus, jehož cílem je v omezeném čase nalézt co nejlepší řešení. Pro zachování úplnosti výsledné

prohledávací strategie je po skončení tohoto algoritmu spuštěn algoritmus úplný, který je založený na obecném schématu řešení omezujících podmínek. Uživatel může prohledávání kdykoliv zastavit a obdržet zatím nejlepší nalezené řešení (pokud nějaké bylo nalezeno).

Pro praktickou využitelnost by bylo vhodné udělat aplikaci více interaktivní. Jednou z možností je vytvořit snažší zadávání vstupních údajů a poskytovat uživateli více informací při nenalezení řešení. V budoucnosti by bylo zajímavé porovnat výsledky dosažené pomocí jiných řešičů podmínek s výsledky dosaženými se zde implementovaným. Dalším důležitým bodem by mohlo být zahrnutí alternativních taxi cest pro pohyb letadel po ploše letiště.

Příloha A

Struktura aplikace a detaily implementace

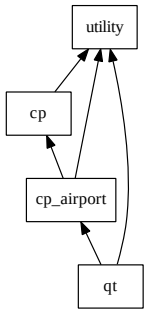
A.1 Rozdělení aplikace

Při návrhu aplikace bylo třeba udělat několik důležitých rozhodnutí. Následující body tato rozhodnutí shrnují.

- Vybrat programovací jazyk, ve kterém bude aplikace naprogramována
- Vybrat knihovnu pro práci s grafickým uživatelským rozhraním
- Nalézt a vybrat existujicí soubory, které by popisovaly strukturu letiště, a z nich vytvořit vstupy pro rozvrhovací problém
- Rozdělit aplikaci tak, aby byly jednotlivé části co nejméně na sobě závislé
- Navrhout a implementovat řešič podmínek

Protože řešení problémů splňování podmínek je výpočetně velmi náročné, byl jako programovací jazyk zvolen jazyk C++. C++ ovšem neobsahuje ve svých standardních knihovnách žádnou podporu pro grafické uživatelské rozhraní. Volba nakonec padla na framework Qt [20]. Hlavními důvody jsou následující skutečnosti: Qt je velmi dobře dokumentováno, je multiplatformní, dobře se s ním pracuje a má víceméně volnou licenci pro komerční i nekomerční produkty. Jedinou podmínkou volného používání je, že v produktech musí být řečeno, že využívaji Qt. Toto jsou hlavní body, které určují osatní části řešení. Popisu vstupních souborů se věnujeme v příloze B. Prozatím řekněme, že základem jsou xml soubory.

Práce byla rozdělena do několika balíčků. Na obrázku A.1 je znázorněno toto rozdělení. Nyní budou několika slovy popsány jednotlivé části:



Obrázek A.1: Rozdělení aplikace - šipka určuje, jak jsou části na sobě závislé

- **utility** – je pouze knihovnou víceméně nezávislých tříd, které jsou využívány ostatními částmi. Je zde definována výjimka, od které dědí všechny ostatní výjimky používané v aplikaci. Pomocí maker R_ASSERT (runtime) a D_ASSERT (debug) je možné v aplikaci zadat jednoduché požadavky na podmínky, které by měly platit. Pokud neplatí, je k chybovému hlášení připojena zpráva o souboru a řádce, kde k chybě došlo. Jsou zde definovány chytré ukazatele (smart pointers [19]) a jednoduché kontejnery jako matice, jednoduchý spojový seznam, setříděný vektor a podobně. Třídy jsou ve jmenném prostoru stde.

- **cp** – zde je implementován obecný řešič podmínek. Hlavními entitami jsou třídy Model a Solver (řešič). Do modelu jsou přidávány proměnné a podmínky. Model je poté přečten řešičem. Prohledávání je provedeno pomocí použití cílů. Celý modul je objektově založený, a proto je snadné jej rozšiřovat – přidávat podmínky a definovat vlastní prohledávací strategie. Všechny třídy a funkce jsou ve jmenném prostoru cp.

- **cp_airport** – tato část vykonává přepis textového vstupu rozvrhovacího problému do programu splňujícího omezující podmínky. Vstup se skládá z popisu operací, které je potřeba vykonat a z popisu zdrojů, které jsou na letišti. Poté tento problém řeší pomocí definování prohledávací strategie. Pro komunikaci s ostatní částí programu je potřeba pouze nastavit vstupní a výstupní textové proudy pomocí několika funkcí. Výstupním souborem rozvhování je xml soubor.

- **qt** – toto je presentační vrstva určená hlavně ke komunikaci s uživatelem a zobrazování mapy, nalezených výsledků a následné simulace těchto výsledků. Pro simulaci výsledků jsou vytvořeny třídy provádějící animace grafických objektů. Pro účely vstupních soborů jsou definovány třídy, které čtou vstupní xml sobory a vytvářejí z nich representaci mapy – vektorové grafické ob-

jekty nebo animace. Ke čtení xml souborů je použito rozhraní SAX, které nabízí Qt framework. V neposlední řadě je část tohoto balíčku použita k extrahování informací z xml souborů, které jsou po kombinaci se zadanými informacemi uživatelem použity k vytvoření souboru popisu zdrojů. Tomuto se blíže věnujeme v části B.

Presentační vrstva je důsledně oddělena od výpočetní. Je tedy možné obě dvě vrstvy poměrně nezávisle měnit. Balíčky utility, cp a cp_airport využívají pouze standardní knihovny C++ [1] a jsou napsány tak, že vyhovují standardu ISO 98. Díky komunikaci pouze pomocí vstupních a výstupních proudů je možné využít cp_airport bez nainstalovaného Qt jako program z příkazového řádku. Dále lze využít samotný řešič podmínek v jiných projektech bez velkých obtíží.

Všechny hlavičkové soubory jsou v adresáři *app/include*. Všechny implementační soubory jsou v adresáři *app/src*. Jsou následně rozděleny do adresářů se jmény balíčků, ve kterých se nachází. V rámci balíčků jsou děleny již dle potřeby. Balíčky, u kterých je možné samostatné využití - cp a cp_airport - mají v adresáři *app/include/cp* respektive *app/include/cp_airport* hlavní hlavičkový soubor. Tento soubor se jmenuje stejně jako balíček a definuje všechny potřebné symboly pro využití balíčku.

Pro sestavení aplikace je použit popis potřebných souborů pomocí projektových souborů Qt. Součástí instalace Qt je utility qmake, která vytvoří z projektových souborů makefile skript pro sestavení. V hlavním adresáři aplikace (*app*) jsou projektové soubory pro samostatně spustitelné aplikace – pro hlavní aplikaci app, samostatný rozvrhovač pro spuštění z příkazového řádku cp_airport a pro aplikaci tests, ve které jsou implementovány unit testy pro balíček cp. V rámci každého balíčku je vytvořen samostatný projektový soubor umístěný v adresáři *app/src/jmeno_balicku*.

Jak obrázek A.1 napovídá, budeme procházet jednotlivé balíčky od spoda nahoru, tedy nejdříve definujeme balíčky, které jsou následnými balíčky využívány.

A.2 Balíček utility

Balíček utility je závislý pouze na třídách ze standadní knihovny. Všechny třídy v něm definované jsou ve jmenném prostoru stde. Pro účely testování je zde implementován jednoduchý systém pro tvorbu unit testů - třída **Test**. Aplikace používá chytré ukazatele, ty jsou definovány v souboru *SharedPtr.hpp*. V adresáři

[1] jedinou výjimkou jsou chytré ukazatele, které byly do jazyka C++ přidány až v technickém reportu 1 z roku 2003, který je ovšem již na většině překladačů implementován

containers jsou definovány jednoduché nezávislé třídy, které je možné v aplikaci používat - jedná se o jednoduchý spojový seznam (**stde::SingleLinkedList**), setříděný vektor (**stde::SortedVector**), matici (**stde::matrix**) nebo funkcionální seznam (**stde::ImutableList**). V neposlední řadě je zde definován logovací systém v adresáři *Logger*. Balíček se skládá pouze ze dvou implementačních souborů *src/utility/test.cpp* a *src/utility/log.cpp*, ostatní třídy jsou zpravidla šablonami (template). Pro hlavní aplikaci je důležitý pouze implementační soubor *log.cpp*. Logovací systém a jeho využití je blíže vysvětleno v sekci A.3.8. Je zde definována třída

stde::MyException, která je odvozena od třídy **std::exception**. Tato třída je předkem pro všechny výjimky v aplikaci používané.

A.3 Balíček cp

V této části ukážeme, jak je koncipován vytvořený řešič, jak je možné jej využít a jakým způsobem může být rozšířen. Vytvořený řešič je snadno použitelný a je možné ho použít pro velké množství aplikací. Je pravděpodobné, že v současné době nedosahuje takových výsledků jako existující řešiče Choco [24], Gecode [22] nebo komerční ILOG Solver [23]. Je to dáno hlavně tím, že jsou vyvíjeny delší dobu, a tudíž jsou lépe optimalizovány a ozkoušeny. V budoucnosti bude řešič dále rozvíjet do podoby srovnatelné s těmito řešiči.

O technikách CP existuje velké množství teoretické literatury. O praktické konstrukci řešičů je ovšem pojednáno v literatuře méně.

V následující části nejdříve ukážeme jednoduchý příklad použití vytvořeného řešiče, který bude ilustrovat jakým způsobem se s CP v procedurálních jazycích programuje. Zároveň budou na tomto příkladě vysvětleny základní prvky řešiče. Poté identifikujeme základní požadavky, které je potřeba při návrhu uvážit, a řekneme, jak jsou implementovány. Na závěr uvedeme přehled implementovaných podmínek.

A.3.1 Můj první CP program

Jako příklad jsem zvolil jednoduchý problém. Je zadáno 8 písmen S, E, N, D, M, O, R, Y. Každé z těchto písmen musí být číslicí v intervalu 0 až 9. Navíc musí platit aritmetická rovnice SEND + MORE = MONEY. Každé slovo tedy representuje číslo složené z písmen. Jak je obvyklé, čísla nesmí začínat nulou - písmena na začátku slov - S a M - nesmí být nula. Navíc je přidána podmínka, že všechna písmena jsou navzájem různá. Tento problém lze vyjádřit pomocí řešiče následovně:

```
1   #include <iostream>
2   #include "cp/cp.hpp"
3
4   int main(){
5       cp::Model model;
6
7       cp::IntVariable S(&model, 1, 9,"S"), E(&model, 0, 9,"E");
8       cp::IntVariable N(&model, 0, 9,"N"), D(&model, 0, 9,"D");
9       cp::IntVariable M(&model, 1, 9,"M"), O(&model, 0, 9,"O");
10      cp::IntVariable R(&model, 0, 9,"R"), Y(&model, 0, 9,"Y");
11
12      cp::IntVariable send( &model, 0, cp::INT_MAX_VALUE, cp::BOUND, "send" );
13      cp::IntVariable more( &model, 0, cp::INT_MAX_VALUE, cp::BOUND, "more" );
14      cp::IntVariable money( &model, 0, cp::INT_MAX_VALUE, cp::BOUND, "money" );
15
16      model.add( send  ==              1000*S + 100*E + 10*N + D );
17      model.add( more  ==              1000*M + 100*O + 10*R + E );
18      model.add( money == 10000*M + 1000*O + 100*N + 10*E + Y );
19
20      model.add( send + more == money );
21
22      cp::IntVariable vars [] = { S, E, N, D, M, O, R, Y };
23
24      cp::allDifferent( &model, vars, vars + 8 );
25
26      std::vector< cp::IntVariable > varsVec( vars, vars + 8 );
27      try{
28            cp::Solver s( model );
29            s.addGoal(  new cp::Labeling( varsVec ) );
30            while( s.nextSolution() ){
31                 s.addGoal( new cp::PrintSelectedVariables( varsVec, std::cout ) );
32                 s.nextSolution();
33            }
34       }catch( std::exception & e ){
35            std::cerr << e.what() << std::endl;
36            return 1;
37       }
38       return 0;
39   }
```

Výstupem tohoto programu je:

```
S in {9}
E in {5}
N in {6}
D in {7}
```

```
M in {1}
O in {0}
R in {8}
Y in {2}
```

Na řádce dva je vložen hlavní soubor celého balíčku. Všechny funkce a třídy, které obsahuje, jsou definovány ve jmenném prostoru cp. Na řádce 5 je vytvořen model, který representuje deklarativní charakter úlohy. Do modelu jsou přidávány proměnné a podmínky. Konstruktory třídy representující proměnné mají jako parametr model, do kterého jsou automaticky přidány. Dalšími parametry jsou hodnoty okrajů intervalu pro doménu a informace, jak má být doména interně representována. Jako volitelný parametr mají jméno proměnné. Proměnné jsou vytvořeny na řádcích 7 - 14.

Do modelu jsou poté přidány podmínky, které musí mezi proměnnými platit. Zde se jedná o aritmetické podmínky vycházející ze zadání úlohy – řádky 16 - 20. Jako poslední je do modelu přidána podmínka, že všechny proměnné musí být různé – řádek 24. Tato funkce má jako parametr ukazatel na model a vytvoří podmínky nerovností[2].

Po vytvoření modelu je model načten řešičem - třída **cp::Solver** - řádek 28. Toto načtení se pokusí vytvořit konzistence požadované podmínkami. Pokud již v této fázi zjistíme, že je model neřešitelný, je vygenerována výjimka. Tato výjimka značí, že se nepodařilo vytvořit počáteční konzistenci. Všechny výjimky používané v práci dědí (nepřímo) od třídy std::exception, proto je kód obalen try - catch blokem. Pokud se podařilo načtení modelu, byl dosažen fixpoint. V tuto chvíli je potřeba definovat prohledávací strategii, která bude řešiči říkat, jaké akce si přejeme udělat. Tyto akce jsou definovány jako cíle. Cíle je možné větvit pomocí and a or. Zde je řešiči zadán jednoduchý cíl, který implementuje algoritmus 1 - postupně ohodnocuje proměnné, které má jako parametr. Pro každou proměnnou zkusí postupně všechny hodnoty. Výběr právě ohodnocované proměnné se řídí heuristikou DOM. Hodnoty jsou postupně zkoušeny od nejmenší po největší. Po zadání cíle spustíme hledání - řádek 30. Tato while smyčka postupně hledá všechna řešení. Pokud bylo řešení nalezeno, je pomocí dalšího cíle vypsáno - řádky 31, 32. Koncept cílů je obdobný [23].

[2]Jedná se pouze o příhodnou funkci, která vloží pro každou dvojici proměnných do modelu jednoduchou podmínku nerovnosti. Pro účely práce nemělo smysl implementovat známou globální podmínku all different, která propaguje lépe

A.3.2 Přehled zvolené implementace řešení

Z předchozího příkladu je možné vyčíst určité požadavky na implementaci a vyvětlit, proč je dobré zvolit právě takovéto řešení.

Implementace je rozdělena do modelovací části a řešící části. Je to dáno tím, že během prohledávání dochází k různým akcím. Tyto akce je nutné nějak koordinovat - například po změně domény proměnné propagovat tuto změnu do dalších domén. Během návratu je ovšem nutné tyto změny vrátit. Navíc je potřeba udržovat určité axiomy, které při prohledávání budou platit – jedním z těchto axiomů je, že doména proměnné nesmí být prázdná. Pro zjednodušení splnění těchto požadavků je sférou vlivu uživatele pouze modelovací oblast, pomocí které komunikuje s řešičem. Řešič se stará o dodržování všech axiomů a o aktuálnost dat. Uživatel pomocí rozšiřování modelu bude vést prohledávání. Všechny modelové objekty jsou od okamžiku svého vytvoření neměnné (imutable). Jejich hlavním úkolem je definovat, jaké objekty z řešící oblasti mají být vytvořeny při jejich načtení řešičem. Model je prvotně vytvořen a poté v konstruktoru řešiče načten. Po úspěšném načtení modelu je možné se řešiče zeptat na hodnoty aktuálních domén. Jako klíč pro dotaz jsou použity modelové proměnné. Kroky vedoucí k nalezení řešení lze definovat pomocí cílů. Cíly může být cokoliv a lze je pomocí logických spojek větvit do složitějších cílů. Tím lze vlastně vytvářet cíle, které se skládají z dalších cílu. Výsledkem je podobná vyjadřovací schopnost jako v neprocedurálních programovacích jazycích (například v prologu). Každý cíl má při provádění dva parametry. Prvním parametrem je Model, do kterého je možné přidat podmínky (rozšířit jej) a tím vést prohledávání. Druhým parametrem je třída, které je možné využít k dotazování na aktuální hodnoty domén.

Koncept cílů byl inspirován [23]. Nevýhodou striktního oddělení modelovací a řešící oblasti je nutnost neustále se ptát na aktuální domény proměnných. Na druhou stranu toto oddělení umožňuje vést prohledávání libovolným způsobem - vždy je přítomný model a do něj můžeme přidávat jakékoliv podmínky. Tato vlastnost bývá ve většině dokumentací řešičů opomíjena[3] a nahrazována pouze sekvencí ohodnocování proměnných. Zde je tedy implementována jako defaultní a dle mého názoru vytváří snažší pohled na prohledávání. Navíc by toto schéma mělo být snadno rozšiřitelné na vícevláknové prohledávání.

Zde je vhodné ještě zmínit otázku správy paměti. Ostatní systémy podmínek napsané v C++ využívají pro zvětšení efektivity vlastní implementaci správy paměti[4]. Vytvoření takovéhoto systému je především z důvodu portability obtížné.

[3] toto jsem nenašel například v dokumentaci [24]

[4] Že se jedná o nezanedbatelnou část, bylo experimentálně ověřeno. Systém jsem testoval

Proto nebyl implementován. Během vytváření modelu a následného prohledávání vzniká mnoho objektů na haldě. Je tedy potřeba určit, jak a kdy budou odalokovány. Pro tyto účely jsou hojně využívány chytré ukazatele, které vhodně řeší tyto problémy. Většina funkcí tedy vrací chytré ukazatele, které jsou typedefem pro pointer na daný objekt (například **cp::IntVariable**).

Jak již bylo řečeno, všechny funkce a třídy zde definované se nacházeji ve jmenném prostoru cp. Mnoho tříd ovšem není pro běžného uživatele důležitých. Proto jsou definovány ve jmenném prostoru cp::detail. V hlavičkových souborech, u kterých se nepřepokládá, že by je měl uživatel vkládat do svého řešení, není striktně dodržováno rozdělení na implementační a hlavičkové soubory. V současné době je nutné napsat třídy, které jsou šablonami (template), v hlavičkových souborech. Je ovšem striktně dodržováno, aby hlavičkové soubory tříd identifikovaných jako veřejné byly pouze deklarativní a nebyly v nich tedy žádné definice. Jména tříd začínají velkými písmeny. Typicky jsou deklarované v souborech, které se jmenují stejně jako třída. Jména funkcí začínají malým písmenem. Privátní datové položky tříd jsou zakončeny podtržítkem (např. data_). V každém víceslovném jménu začína každé další slovo velkým písmenem (tedy IntVariable, getContext). Ve většině případů končí privátní funkce stejně jako privátní položky podtržítkem. Často je využívána tzv. template metoda, která definuje rozhraní pomocí nevirtuálních metod a skutečná práce je provedena v privátních virtuálních metodách. Tyto metody se zpravidla jmenují stejně jako funkce z rozhraní, akorát končí podtržítkem. Tyto metody je poté potřeba u podtříd implementovat. Výhodou tohoto návrhu je například možnost obalení kódu, který je prováděn virtuální metodou, testy na předpoklady. Nyní se budeme věnovat jednotlivým klíčovým částem balíčku.

A.3.3 Model

Všechny modelové objekty dědí od třídy **cp::MObject**. Třída **cp::Model** je potom jednoduchá třída, která drží seznam těchto objektů. Do tohoto seznamu je možné přidávat pouze na začátek - tedy před hlavu seznamu. Protože bude nutné pracovat s rozdílovými seznamy, byl seznam implementován podobným způsobem jako ve funkcionálních jazycích – seznam mající data a držící další seznam (poslední drží prázdný seznam). Výhodou této representace je snadné manipulování s modely, možnost jejich laciného kopírování (kopíruje se jeden chytrý ukazatel) a

s použitím alokátoru TCMalloc (volně šiřitelného softwaru navženého a využívaného googlem [28]). Tento alokátor bez jakýchkoliv úprav pro účely aplikace (například zahrnutí backtrackingu) zrychlil program asi o 15 procent.

snadné provádění rozdílů. Tento funkcionální seznam je definován jako template třída v balíčku utility (třída **stde::ImutableList** v adresáři *utility/containers*). Model má jako jedinou svoji datovou položku právě tento seznam a nabízí jednoduché operace. K operacím nad modelovými objekty je využíván návrhový vzor visitor [16]. Tohoto vzoru je využito k načtení modelu i k načítání rozdílů dvou modelů - konkrétní visitor je podtřídou **cp::MVisitor**.

Modelové proměnné

K modelování jsou potřeba proměnné. Tyto proměnné jsou třídou **cp::MIntVariable**, která dědí od **cp::MObject**. Řešící proměnné, na které je modelová proměnná převedena, se liší pouze v representaci svých domén a svými (potencionálními) jmény. Representace domén jsou v tuto chvíli implementovány tři - doména enumerovaná, doména representovaná okraji a konstantní doména. Parametrem konstruktoru je celočíselná hodnota, která určuje, který typ domény má být pro řešící proměnnou vytvořen. V současnosti je pro jednotlivé representace typů domén definován enum s hodnotami **cp::BOUND**, **cp::ENUM** a **cp::INT_INSTANTIATED**. Celočíselná hodnota byla zvolena proto, aby bylo možné representaci domén rozšiřovat - k vytvoření domén je používán návrhový vzor factory method [16]. Je tedy možné registrovat novou representaci pomocí třídy **cp::SIntDomainFactory**. Pokud není druh representace zadán, je defaultně vytvořena proměnná s enumerovanou doménou. Dalšímy parametry pro tvorbu proměnné jsou hranice jejího intervalu nebo výčet hodnot. K vytvoření proměnné můžeme také využít funkcí *makeIntVar*, které nabízejí větší možnosti ve volbě parametrů. Doména proměnné musí být z intervalu **- cp::INT_MAX_VALUE** až **cp::INT_MAX_VALUE**. Proměnné s boolovskou hodnotou jsou representovány stejným způsobem jako proměnné s celočíselnou doménou. Musí být vytvořeny podobným způsobem pomocí funkcí *cp::makeBoolVar*. Všechny tyto funkce jsou deklarovány v souboru *cp.hpp*. Jak **cp::IntVariable**, tak **cp::BoolVariable** je pouze *typedef* za chytrý ukazatel na třídu **cp::MIntVariable**. Modelové proměnné mají také ten účel, že fungují jako klíč v dotazech na aktuální hodnoty domén v řešeném problému.

Modelové podmínky

Po vytvoření proměnných jsou do modelu přidány podmínky. Všechny podmínky je možné do modelu přidat jako volání funkcí. Například podmínka $a = b * c$ je do modelu m přidána jako volání $cp :: times(\&m, a, b, c)$. Všechny takovéto funkce jsou definovány v souboru *cp.hpp*. Parametrem je také model, do kterého se

podmínky automaticky přidají. Modelová podmínka má pouze ten účel, aby si zapamatovala modelové proměnné a definovala, jak bude vytvořena řešící podmínka. Modelové podmínky využívají návrhového vzoru abstract factory [16]. Na základě parametrů, s kterými jsou vytvořeny, jsou jim během načítání modelu poskytnuty řešící proměnné.

Některé podmínky lze také do modelu přidat jako aritmetický výraz. Podmínku zvolenou jako příklad lze přidat jako volání $m.add(a == b * c)$. Podmínka je poté rozložena na syntaktický strom a postupně jsou tvořeny mezivýsledky - je tedy nejdřív přidána pomocná proměnná, která je rovna výrazu $b * c$, a poté je vytvořena podmínka *cp::eq*, která zaručuje, že se mezivýsledek rovná a. Z hlediska efektivity je tedy důležité rozlišovat, kdy dochází k vytváření mezivýsledků a kdy toto nenastává – podle toho lze využívat buď dané funkce, nebo více názornou representaci pomocí výrazů. Některé z existujících řešičů dělají analýzu, jak dané výrazy rozložit, aby byl výsledek co nejefektivnější. Toto je však daleko za hranicí této práce. Výrazy byly implementovány pomocí tzv. *expression templates* [29]. Mezi jejich výhody patří, že se syntaktický strom vytváří během překladu. Bývají využívány v knihovnách pro manipulaci s maticemi a vektory. Ne všechny výrazy byly implementovány. Byly implementovány operátory $\geq, \leq, +, *, ==, ||$ v souboru *Expression.hpp*. Tyto operátory je snadné rozšířit pro další aritmetické operace.

Důležitým konceptem v programování s omezujícími podmínkami jsou také tzv. *rozšířené podmínky* (reified constraints). Ty dovolují definovat podmínky složené z dalších podmínek. Tedy například podmínky typu $x \leq y + 4 \leftrightarrow y > 5$. Takovýmto podmínkám se také někdy říká *meta podmínky*. Musí být schopné propagovat jak svojí pravdivou hodnotu, tak svůj opak. Pro někeré podmínky není úplně jasné, jak by se měl jejich opak propagovat, a tudíž jsou zpravidla vytvořeny jen pro nějaké podmínky. Pro účely práce stačilo jako rozšířenou podmínku udělat podmínku rovnosti, menší než a podmínku nebo. Byl zvolen koncept, že tyto podmínky mají jako hodnotu řídící boolovskou proměnnou, která určuje, jak se má daná podmínka propagovat. Tato hodnota je tedy vždy nula, jedna nebo je neznáma (může nabývat obou hodnot). V případě neznámé hodnoty jsou obě varianty pouze kontrolovány, zda o některé z nich již není dokázáno, že musí platit. Pokud je toto dokázáno, je řídící proměnná změněna na danou hodnotu a dále propagujeme zjištěnou variantu. Tato representace dovoluje používat podmínky jako výrazy. Je tedy snadné vytvořit ekvivalenci ($model.add((x <= y) == (x == z))$ - $x \leq y$ je ekvivalentní $x = y$) nebo implikaci ($model.add((x <= y) <= (x == z))$ - $x \leq y$ implikuje $x = y$).

- $lt(x, int\ c)$ - vytvoří podmínku $x < c$, kde x je celočíselná proměnná a c je konstanta, zařizuje NC konzistenci
- $leq(x, int\ c)$ - vytvoří podmínku $x \leq c$, kde x je celočíselná proměnná a c je konstanta, zařizuje NC konzistenci
- $geq(x, int\ c)$- vytvoří podmínku $x \geq c$, kde x je celočíselná proměnná a c je konstanta, zařizuje NC konzistenci
- $gt(x, int\ c)$ - vytvoří podmínku $x > c$, kde x je celočíselná proměnná a c je konstanta, zařizuje NC konzistenci
- $eq(x, int\ c)$ - vytvoří podmínku $x = c$, kde x je celočíselná proměnná a c je konstanta, zařizuje NC konzistenci
- $neq(x, int\ c)$ - vytvoří podmínku $x \neq c$, kde x je celočíselná proměnná a c je konstanta, pro enumerované proměnné zařizuje NC konzistenci, pro okrajové BC konzistenci
- $eq(x, y)$ - vytvoří podmínku $x = y$, kde x jsou celočíselné proměnné, zařizuje AC konzistenci
- $neq(x, y)$ - vytvoří podmínku $x \neq y$, kde x, y jsou celočíselné proměnné, pokud jsou obě proměnné enumerované, zařizuje AC konzistenci, jinak BC konzistenci
- $leq(x, y, int\ \ c)$ - vytvoří podmínku $x \leq y + c$, kde x, y jsou celočíselné proměnné, c je konstanta, zařizuje BC konzistenci
- $times(z, x, y)$ - vytvoří podmínku $z = x * y$, kde x, y, z jsou celočíselné proměnné, konzistence nelze zařadit - slabší než BC
- $allDifferent(x_1, \ldots, x_n)$ - vytvoří $n * (n-1)/2$ podmínek neq
- $lleq(x_1, \ldots, x_n, c_1, \ldots, c_n, int\ \ c)$ - vytvoří podmínku $\sum_i c_i * x_i \leq b$, kde $x_1, \ldots, x_n$ jsou celočíselné proměnné, $c_1, \ldots, c_n, b$ jsou konstanty, zařizuje BC konzistenci
- $equation(x_1, \ldots, x_n, c_1, \ldots, c_n, int\ b)$ - vytvoří podmínku $\sum_i c_i * x_i = b$, kde $x_1, \ldots, x_n$ jsou celočíselné proměnné, $c_1, \ldots, c_n, b$ jsou konstanty, zařizuje BC konzistenci

- $domainConstraint(x, b_1, \ldots, b_n, v_1, ...v_n)$ - vytvoří podmínku $x = v_i \leftrightarrow b_i = 1$, kde x je celočíselná proměnná, $b_1, \ldots, b_n$ jsou boolovské proměnné, $v_1, \ldots, v_n$ jsou konstanty, zařizuje AC konzistenci

- $feasiblePair(x, y, CouplesTable)$ - vytvoří podmínku $\exists(i,j)|(i,j) \in CouplesTable \wedge (x,y) = (i,j)$, , zařizuje AC konzistenci

- $min(x, x_1, \ldots, x_n)$ - vytvoří podmínku $x = min(x_1, \ldots, x_n)$ kde $x, x_1, \ldots, x_n$ jsou celočíselné proměnné, zařizuje BC konzistenci

- $leq\_meta(x, y, z, int\ c)$- vytvoří podmínku $x \leftrightarrow (y \leq z + c)$, kde x je bool proměnná, y, z jsou celočíselné proměnné

- $or\_meta(x, y, z)$- vytvoří podmínku $x \leftrightarrow (y \vee z)$, kde x, y, z jsou bool proměnné

- $eq\_meta(x, y, z)$- vytvoří podmínku $x \leftrightarrow (y = z)$, kde x je bool proměnná, y, z jsou celočíselné proměnné

- $cumulative(s_1, \ldots, s_n, e_1, \ldots, e_n, height_1, \ldots, height_n, capacity)$ - kumulativní podmínka - aktivity se startovními časy s_i a koncovými časy e_i, požadují $height_i$ kapacity zdroje musí využívat stejný zdroj s kapacitou *capacity*.

- $disjunctive(s_1, \ldots, s_n, e_1, \ldots, e_n)$ - disjunktivní podmínka - aktivity se startovními časy s_i a koncovými časy e_i, požadují jednu jednotku unárního zdroje.

Z propagačních algoritmů jsou nejzajímavější algoritmy pro propagování kumulativní a disjunktivní podmínky. Popis těchto algoritmů je složitý, a proto zde pouze zmíním odkazy na literaturu, odkud byly převzaty a podle kterých byly implementovány. Pro kumulativní podmínku byl implementován algoritmus navržený v [4]. Je to algoritmus s časovou složitostí $O(n^2 * sz(heights))$, kde $sz(heights)$ označuje počet rozdílných hodnot požadovaných kapacit zdroje. Pro propagování disjunktivní podmínky byly převzaty algoritmy, které byly vyvinuty na půdě mff. Jejich popis je součástí disertační práce [7]. Jedná se o algoritmy *not-last, not-first, edge-finding, detectable-precedense, isOverloaded*. Tyto algoritmy využívají speciální datovou strukturu **ThetaTree** nebo **ThetaLambdaTree** - jedná se o binární stromy, které dovolují dělat potřebné operace rychle. Jsou implementovány v souborech v adresáři *app/cp/integer/constraints/task_filtering_algorithms*. Propagování těchto podmínek je defaultně nastaveno tak, jak je navrženo v práci, ze které jsou převzaty. Složitost těchto algoritmů je $O(n * log(n))$.

A.3.4 Cíle

Pro definování prohledávací strategie je použit koncept cílů. Tyto cíle dědí od třídy **cp::Goal**. Třída **cp::Goal** má ryze virtuální metodu *execute*. Tato funkce má jako první parametr ukazatel na model, do kterého je možné přidat další podmínky a tím vést prohledávání. Druhým parametrem je ukazatel na třídu *cp::Problem*. Ta má aktuální informaci o stavu domén a je skrz ní možné tuto informaci zjistit - pro danou modelovou proměnnou je možné získat náhled na aktuální proměnnou. Tento náhled je konstantní a není možné pomocí něj nějak proměnnou měnit. Je možné z něj ovšem zjistit užitečné informace jako velikost domény, nejnižší a nejvyšší hodnotu v doméně nebo iterovat pomocí iterátorů přes všechny hodnoty v doméně. Existují dále takzvané meta cíle, tyto cíle jsou buď cíl and nebo cíl or. Pomocí těchto metacílů je možné větvit cíle. Nejjednodušším vysvětlením bude názorná ukázka na příkladu. Bude zde ukázáno, jak je definován cíl **Labeling** použitý v prvním CP programu.

```
1   class SetMinValue : public Goal{
2      IntVariable var_;
3   public:
4      SetMinValue( const IntVariable & toVariable ) : var_(toVariable){}
5      virtual GoalPtrType execute( Model * modelToRefine, Problem * problem ){
6         int val = problem->variable( var_ )->getLB();
7         eq( modelToRefine, var_, val );
8         return success();
9      }
10  };
11
12  class RemoveMinValue : public Goal{
13     IntVariable var_;
14  public:
15     RemoveMinValue(const IntVariable & toVariable) : var_(toVariable){}
16     virtual GoalPtrType execute( Model * modelToRefine, Problem * problem ){
17        int val = problem->variable( var_ )->getLB();
18        neq( modelToRefine, var_, val );
19        return success();
20     }
21  };
22
23
24  class Labeling : public Goal{
25     std::vector<IntVariable> & vars_;
26  public:
27     Labeling ( std::vector< IntVariable > & vars ) : vars_(vars)
```

```
28        { }
29        virtual GoalPtrType execute(Model *, Problem * problem ){
30           int index = -1;
31           size_t minDom = 2 * INT_MAX_VALUE + 1;
32           for( size_t i = 0, N = vars_.size(); i < N; ++i ) {
33              SIntVariable v = problem->variable(vars_[i]);
34              if ( !v->isInstantiated() && v->size() < minDom ){
35                 minDom = v->size();
36                 index = (int)i;
37              }
38           }
39           if (index == -1)
40              return success();
41           return GoalPtrType(
42              new AndGoal(
43                 new OrGoal( new SetMinValue( vars_[index] ),
44                             new RemoveMinValue( vars_[index] ) ),
45                 new Labeling(*this))
46           );
47        }
48     };
49
```

Konstruktor cílu **Labeling** má jako parametr vektor proměnných. Cílem je ohodnotit všechny tyto proměnné. Pokaždé, když jsou všechny proměnné ohodnoceny, byl cíl **Labeling** splněn.

Metoda *Goal::execute* vrací další cíl, který má být proveden - takto může pomocí and a or cílů být cíl rozložen na menší podcíle. Existují dvě speciální hodnoty cílů a to *success* a *fail*. Hodnota *success* znamená, že cíl byl splněn, tudíž už nepotřebuje provádět další akce. Hodnota *fail* znamená, že cíl selhal - je potřeba návratu k poslednímu cílu or a zkusit jinou variantu. Cíl and je splněn, pokud jsou splněny oba cíle, z kterých je složen. Cíl or je splněn, pokud je splněn alespoň jeden z cílů, ze kterých je složen.

V metodě *Labeling::execute* je nejdříve na řádcích 30 - 38 vybrána proměnná, která bude ohodnocována jako další (pomocí heuristiky DOM). Pokud jsou všechny proměnné ohodnoceny (instanciovány), cíl je splněn (vrátí hodnotu *success*). V opačném případě je potřeba vytvořit nové cíle, které naplní hlavní cíl **Labeling**. Vybraná proměnná má v doméně alespoň dvě hodnoty – je tedy potřeba udělat nějakou volbu, abychom zmenšili tuto doménu. Pomocí cíle or je vytvořen *choice point* neboli bod výběru – buď se proměnná rovná nejmenší hodnotě v doméně, nebo se jí nerovná. Tyto možnosti jsou definované jako další možné cíle (řádky 43, 44). Toto ale ke splnění cíle **Labeling** nestačí, proto je tento cíl zřetězen meta

cílem and s novým cílem **Labeling**. Takto je vlastně definována rekurze – po splnění cíle dosazení nebo vyjmutí proměnné z domény pokračuj v ohodnocování cílem **Labeling**. Jednoduché cíle **SetMinValue** a **RemoveMinValue** poté ilustrují, jak je možné daný model rozšiřovat. Metody v těchto cílech používané jsou stejné jako při modelování problému. Řádky 6, 17, 33 a 34 ilustrují, jak lze získat informace o aktuálních hodnotách domén proměnných.

Všechny cíle jsou vytvořeny pomocí operátoru new. Ovšem kdy a jestli vůbec bude cíl proveden, není v době tvorby cíle známo (nějaký cíl před ním může selhat). Proto metoda *execute* vrací **cp::GoalPtrType**, což je typedef za chytrý ukazatel na **cp::Goal**. Z charakteristiky cílů vyplývá, že se musí chovat jako čistá matematická funkce - tedy nemají mít měnitelný vnitřní stav a každé volání má být závislé pouze na parametrech, protože není jisté, kolikrát budou díky předchozím bodům výběru volány. Proto je vhodné vždy pro rekurzi vytvořit nový stav s pozměněným stavem daným udělanou volbou. Cíle jsou vždy vyhodnocovány odleva doprava (cíl $a \wedge b \wedge (c \vee d)$ bude postupně proveden jako a, b, c, návrat, d).

Existuje více předdefinovaných cílů. Například zobecněním cíle **Labeling** vznikne cíl **cp::DfsLabeling**. Tento cíl implementuje obecné prohledávání jak je definováno v algoritmu 1. Při konstrukci tohoto cíle jsou jako parametry specifikovány heuristiky pro výběr proměnné a pro výběr hodnoty. Tyto heuristiky implementují metodu *select*, která vrací buď proměnnou, nebo hodnotu, která má být vyzkoušena jako první. Tyto heuristiky implementují rozhraní **IVariableSelector** pro výběr proměnné a **IValueSelector** pro výběr hodnoty. Opět existují některé základní předdefinované. Očekává se, že uživatel z nich bude vytvářet podtřídy, které budou vhodné pro řešení specifického problému.

Kromě meta cílů existují i speciální cíle určující tvar prohledávaného stromu – podtřídy třídy **cp::StrategyGoal**. Pomocí nich je definované nesystematické prohledávání jako je Lds nebo Dds. Tyto cíle mají tu vlastnost, že jsou propagovány do potomků (vytvořených podcílů), dokud potomci nemají vlastní strategii. Okamžiky volby definují binární strom. Strategie definuje, kterým směrem se vydáme - v každém okamžiku volby se tedy zeptáme strategie, jestli dovoluje jít doleva (první parametr or cíle) nebo doprava (druhý parametr or cíle). Tyto speciální cíle je možné vytvořit a aplikovat na jiný cíl pomocí funkce *cp::apply(cp:: Goal *, cp::StrategyGoal *)*.

Všechny tyto cíle jsou definovány v hlavičkovém souboru *cp/search/Search.hpp* a implementovány v souboru *src/cp/search.cpp*, kde je možné také vypozorovat více podrobností o implementaci cílů.

A.3.5 Řešič

Řešič je třída **cp::Solver**. Jeho veřejné rozhraní lze popsat takto:

```
Solver::Solver( const Model & model );
/*< přetransformuje model do řešících objektů a pokusí se vytvořit počáteční
 konzistenci, pokud se konzistenci nepovede vytvořit, je vyhozena výjimka
dědící od třídy cp::Exception. */

void Solver::setOptimize( BoundingFunctionPtrType boundingFunction );
/*< pro účely branch and bound algoritmu je možné instalovat funkci, kterou
 zavolá po každém provedení backtrackingu. Tato funkce vrací cíl,
representující omezení hodnoty objektivní funkce. Tento cíl bude proveden dříve,
 něž bude proveden or cíl ke kterému jsme se navraceli. Je na uživateli
pamatovat si nejlepší dosud nalezenou hodnotu. Pro tyto účely je možné využít
předdefinovanou třídu cp::Optimize. */

void Solver::setTimeLimit(double timeLimitInSeconds);
/*< Nastaví omezující limit na dobu prohledávání */

void Solver::printInformation( std::ostream & oss );
/*< Vypíše statistiky prohledávání, dobu od vytvoření, počet selhání,
počet cílů... */

void Solver::addGoal( Goal * goal );
/*< Určí cíl který se má splnit. */

Problem * Solver::problem() const;
/*< Vrátí strukturu representující aktuální stav, je možné ji využít
k získání informací o aktuálních doménách*/

bool Solver::nextSolution();
/*< Začne prohledávat s definovaným cílem. Pokud splní všechny
 cíle vrátí true, jinak false. Po opakovaném zavolání po splnění
všech cílů se vrátí k nejbližšímu nezkoušenému bodu výběru a
 zde zkusí druhou alternativu. Před voláním musí uživatel zadat cíl.*/
```

Řešič - implementace prohledávání

Řešič je implementován pouze jako interpretr cílů. Zajišťování doménových informací má na starosti třída **cp::detail::SProblem** (podtřída třídy **cp::Problem**). Tato třída má pro účely řešiče pouze tři metody:

```
bool SProblem::propagate( const Model \& refinedModel )
//< načte nový model a vrátí hodnotu zda se podařilo nalézt konzistenci
```

```
void SProblem::increaseLevel()
//< zvýší úroveň prohledávání při narazení na or cíl

void SProblem::decreaseLevel()
//< vrátí o jednu úroveň zpět datové struktury
```

Řešič má interní zásobník (nazvěme ho *or zásobník*), na kterém jsou prohledávané vrcholy. Pokaždé, když narazí na cíl or, vytvoří nový vrchol a dá ho na tento zásobník. Toto zahrnuje i operaci *SProblem::increaseLevel*. Každý takovýto vrchol má vlastní zásobník, na kterém jsou prováděny and cíle pro daný vrchol (nazvěme jej *and zásobník*). Pro dosažení cíle je nutné splnit všechny cíle v and zásobnících od vrcholu or zásobníku do jeho spodu. Během provádění cíle může být současný model rozšířen. Proto je po provedení každého cíle tento rozšířený model načten a propagován pomocí funkce *SProblem::propagate()*. Pokud se nepodařilo vytvořit konzistenci, je potřeba návratu - pro navrácení datových struktur do původního stavu je zavolána funkce *SProblem::decreaseLevel()*, odstraněn vrchol z vrchu or zásobníku a je pokračováno v prohledávání s uloženým alternativním cílem.

Řešič - paměť při navracení

Protože během prohledávání dochází v programu k různým změnám - ať již stavu podmínek, stavu domén, nebo stavu problému, je nutné vytvořit subsystém, který se bude starat o aktuálnost dat v paměti a během navracení tato data vrátí do původního stavu. Jsou obecně možná dvě schémata. První, takzvané kopírovací, si zapamatuje všechna data, která se můžou změnit. Během návratu všechna tato data obnoví. Druhé schémata představuje zapamatovatování sii pouze těch dat, která byla v dané úrovni prohledávání změněna, a při návratu obnovení pouze těchto dat. První schéma, ačkoliv jednodušší, se nezdálo být dostatečně efektivní. Proto byl vytvořen systém založený na druhém schématu. Pro tyto účely bylo vytvořeno rozhraní **cp::detail::Backtrackable** a od něj byla odvozena částečně implementační část **cp::detail::BacktrackableEntity**. Každý **cp::Problem** má právě jednu třídu **cp::BacktrackContext** (jeden řešič má jeden problém a ten má jeden **BacktrackContext**). U této třídy se při změně dat registrují datové struktury. Tato registrace znamená, že při návratu na tuto úroveň potřebují tyto třídy obnovit svá data. Funkcí, kterou se registrují je *BacktrackableEntity::registerForBacktrack()*. Každá entita si musí pamatovat, jak si data změnila a jaké akce mají být při návratu provedeny. Rozhraní **Backtrackable** definuje virtuální funkci *Backtrackable::backtrackToLevel_(level)*. Po zavolání této funkce

se datová strukura vrátí do stavu při příchodu na tuto úroveň (například provede opak všech akcí pod touto úrovní). Třída **cp::detail::BacktrackManager** udržuje pro jednotlivé úrovně seznam toho, které datové struktury byly změněny. Poté při návratu na nižší úroveň zavolá pro všechny zaregistrované datové struktury na této úrovni funkci *backtrackToLevel*. Tím jsou datové struktury obnoveny.

Existující paměťové třídy jsou:

1. **cp::detail::BacktrackableInt**
2. **cp::detail::BacktrackableEBool**
3. **cp::detail::BacktrackableSet** (třída fungující jako std::set)
4. **cp::detail::BacktrackableMap** (třída fungující jako std::map)
5. **cp::detail::BacktrackableVector**

Všechny podmínky a domény mohou vytvořit libovolnou z těchto datových struktur pro uložení svých stavů. Tato datová struktura sama udržuje aktuálnost dat. Například třída **BacktrackableInt** má dvě metody *bool set(int)* a *int get()*. Metoda *get* vrací aktuální hodnotu. Metoda *set* nastaví aktuální hodnotu na hodnotu parametru. Pokud je tato hodnota jiná než předešlá, zapamatuje si předešlou hodnotu. Návratová hodnota značí, zda došlo ke změně hodnoty. Celý tento podsystém je v adresáři *cp/memory*. Interně využívají paměťové třídy template třídu **cp::detail::BacktrackStoredValues**. Tato třída udržuje seznam hodnot s úrovní.

Po navržení a implementování tohoto systému bylo zjištěno, že v ostatních řešičích bývá pro větší efektivitu jedno velké paměťové pole, které je využíváno ostatními datovými strukturami. (Přináší to výhody ve zlepšení cacheovaní nebo ve zmenšení paměťových nároků.) V této implementaci jsou jednotlivé struktury nezávislé a každá má vlastní paměť. V budoucnosti bude proto tento systém přebudovat a zefektivněn touto cestou.

A.3.6 Domény a proměnné

Jak již bylo řečeno, aplikace byla rozdělena na řešící a modelovací část. Pro účely získávání aktuálních informací o proměnných byl vytvořen náhled na řešící proměnné. Tento náhled je implementován třídou **cp::detail::ISIntVariable**. Ukazatel na tuto třídu je dosažitelný jako typedef ve jmenném prostoru cp - **cp::SIntVariable**. Tato třída definuje základní operace s konstantní proměnnou – je možné zjistit velikost domény a iterovat pomocí **ISIntVariable::const_iterator**

přes hodnoty v doméně. Dále je možné zjisťovat jestli je instanciovaná, jestli obsahuje určitou hodnotu nebo další podrobnosti o proměnné. Pro podmínky je ovšem potřeba dalších operací dovolujících měnit stav proměnné. Proto je od třídy **ISIntVariable** odvozena třída **cp::detail::SIntVariable** umožňující i tyto operace. Veškeré funkce proměnné delegují aktuální práci do funkcí domény - třídy **cp::detail:SIntDomain**. Od této třídy jsou odvozeny konkrétní representace domén - **cp::detail::BoundIntDomain**, **cp::detail::EnumIntDomain**, **cp::detail::ConstIntDomain**, **cp::detail::BoolDomain**, **cp::detail::ConstBoolDomain**.

A.3.7 Propagace podmínek

V algoritmu MakeACConsistent (algoritmus 3) je nastíněno, jak vypadá propagace podmínek. Řešící podmínka během své konstrukce dostane jako parametr seznam svých proměnných. Je potomkem třídy **cp::detail::SConstraint**. Tato třída definuje ryze virtuální funkci *PropagationResult awake_()*. V této funkci je provedena propagace podmínky poprvé. K propojení domén proměnných s podmínkami je využito návrhového vzoru observer [16]. Ve funkci *awake_* se může podmínka registrovat jako posluchač ke specifickým změnám v doméně proměnné. Rozlišujeme tyto změny - změna spodní hranice, změna horní hranice, instanciace a odstranění hodnoty uprostřed. Kódy těchto změn jsou definovány v **cp::detail::PropagationEventNumbers**. Registrace k proměnné je provedena pomocí jedné z následujících funkcí.

```
template <typename ListenerT>
EventHandler cp::detail::SIntVariable::attach(
   ListenerT* object,PropagationResult (ListenerT::*member)(PropagationEventArgs),
   int eventNumber, int priority)

template <typename ListenerT>
void cp::detail::SIntVariable::attach(
          EventHandler handler, ListenerT* object,
          PropagationResult (ListenerT::*member)(PropagationEventArgs),
          int eventNumber, int priority )
```

Každé spojení má jednoznačný identifikátor - **EventHandler**. Tento identifikátor jednoznačně určuje jednu filtrační proceduru. Podmínka může vytvořit libovolné množství filtračních procedur. Často má ovšem podmínka pro více proměnných jednu filtrační proceduru, kterou je při změně libovolné z nich potřeba provést. Proto existuje také druhá varianta této metody s již zadaným identifikátorem. Důvody jednoznačného identifikátoru budou vysvětleny níže.

Filtrační procedury jsou funkce se signaturou

```
PropagationResult ListenerT::any\_method(PropagationEventArgs)
```

Jsou definované na libovolném objektu typu ListenerT a mají libovolné jméno. Je pouze potřeba, aby jako argument měly třídu **PropagationEventArgs** a vracely strukturu **PropagationResult**.

Filtrační procedura se zajímá jen o určité změny v doméně proměnné. Druhy těchto změn jsou třetím parametrem funkce *attach*. Jak již bylo řečeno v části 1.2.3, je možné pro zvětšení efektivity libovolně propagační procedury uspořádat. Jak mají být uspořádány, je definováno pomocí čtvrtého parametru.

Důvody pro jednoznačný identifikátor jsou dva. Pokud filtrační procedura zjistí, že je již vyřešena (resolved), není potřeba, aby dále dostávala informace o změnách v doméně proměnné. Může se tedy odhlásit. K tomu slouží následující funkce:

```
cp::detail::SIntVariable::detach(EventHandler)
```

Podmínky se tedy můžou dynamicky přihlašovat a odhlašovat jako posluchači k proměnným. Například podmínka min využívá tohoto principu. Navratitelné datové struktury zařídí, aby po návratu byly spoje správně navázány. Základem globálního propagačního algoritmu je fronta filtračních procedur. Pokud je proměnné změněna doména, jsou do této fronty přidány všechny filtrační procedury, které byly k dané změně přihlášeny. Ovšem pokud některá z těchto procedur změní tu samou proměnnou stejným způsobem, situace se opakuje. Jenomže v tento okamžik již může procedura být ve frontě (je naplánovaná k proběhnutí, ale zatím neproběhla). Zbytečně by ve frontě byla vícekrát. Toto je druhá motivace pro jednoznačný identifikátor spojení, pomocí nějž jsou identifikovány procedury, které ve frontě již jsou.

Návratová hodnota filtrační procedury je **cp::detail::PropagationResult**. Jedná se o strukturu obalující enum. Může nabývat hodnoty **PropagationResult::Ok, PropagationResult::Update, PropagationResult::Fail, PropagationResult::EnviromentChange**. Poslední dvě jmenované mají stejný význam. Ok je vráceno filtrační procedurou, pokud odfiltrovala všechny hodnoty, které odfiltrovat chtěla a není ji již třeba do okamžiku změny jejích proměnných naplánovat. Update je hodnota vrácená filtrační procedurou, pokud si přeje být i po návratu spuštěna znovu - v případě nalezené změny pro proměnnou tímto umožňuje, aby jiné méně časově náročné procedury mohly nejdříve provést svoji práci. Jinými slovy není po spuštění z fronty naplánovaných filtračních procedur vyjmuta.

Podmínky mohou měnit stav domén pomocí funkcí definovaných ve třídě **cp::detail::SIntVariable**. Jsou jimi *setLessEqualThan, setMoreEqualThan, removeFromDomain, restrictDomain*. Všechny tyto funkce mají dva parametry – celočíselnou hodnotu a ukazatel na třídu, která tuto operaci požaduje (toto je z lo-

govacích důvodů vyvětlených v části věnované ladění systému podmínek). Vracejí opět hodnotu **PropagationResult**, která určuje, jestli se daná operace povedla. Pokud nikoli – je vrácena hodnota Fail, pokud došlo ke změně - je vrácena hodnota Update, pokud operace nic nezměnila – je vrácena hodnota Ok. Proměnné, či lépe řečeno domény proměnných, zajišťují informování všech podmínek k dané změně napojených – tedy naplánování běhu jejich filtračních algoritmů.

Pro chybové stavy jako je vyprázdnění domény byla zvolena cesta návratových kódů. Je tedy nutné, aby je podmínky respektovaly a chovaly se korektně. Pokud jim je během filtrační procedury vrácena z proměnné hodnota Fail, musí ihned tuto hodnotu vrátit vyšší úrovni jako výsledek. Například systém choco pro toto využívá cestu výjimek. Protože však během prohledávání k vyprázdnění domén zpravidla dochází často, není cesta výjimek šetrná - jsou notoricky drahé a nevystihují daný problém (výjimka by neměla být častá).

Nyní již zbývá vysvětlit pouze parametr filtračních algoritmů – třídu **PropagationEventArgs**. Tato třída má tři datové položky, které dovolují filtračním procedurám zjistit důvod, proč jsou volány. Datovými položkami jsou - proměnná která byla změněna, hodnota kódu, který definuje danou změnu, a číslo poslední verze domény. První dvě hodnoty jsou zřejmé. Třetí hodnota dovoluje získat iterátor na hodnoty, které byly z domény proměnné vyjmuty od posledního spuštění tohoto filtračního algoritmu. Tyto informace jsou zásadní pro možnost vytvoření algoritmů na základě AC4. Iterátory je možné získat pomocí volání *deltaBegin(domainVersion)*, *deltaEnd(domainVersion)* na třídě **cp::detail::SIntConstraint**.

Všechny třídy implementující navazování spojení a frontu filtračních algoritmů jsou v adresáři *cp/propagation*. Fronta filtračních algoritmů (třída **PropagatorsEventsQueue**) je seřazena podle priority filtračních procedur a poté podle identifikátorů. Samotná propagovací smyčka je definována ve třídě **Propagator**. Třída **PropagationEventsManager** zajišťuje pro domény proměnných navazování a rušení spojů a také přidání všech napojených filtračních algoritmů do fronty. Třída **PropagationEvent** zastupuje jeden druh změny, která může pro proměnnou nastat, a drží seznam všech napojených fiiltračních algoritmů.

A.3.8 Ladění systému podmínek

Jednou z nevýhod systému podmínek je jejich obtížná pozorovatelnost, tedy zjišťování, k jakým propagacím dochází a k jakým nikoliv. Z tohoto důvodu vzniknul také logovací systém, ve kterém je možné tyto informace nalézt. Tento

logovací systém je definovaný v souboru utility/logger/log.hpp a includován v souboru cp/Log.hpp do balíčku cp. Definuje dvě makra **LOG** a **LOG_MASK**. Makro **LOG** se chová jako standartní výstupní proud (std::ostream). Makro **LOG_MASK** řídí, jestli se vytváří logovací soubor. Tento soubor se jmenuje **cp_out.txt** Pokud je **LOG_MASK** definováno jako 0, k žádnému logování nedochází. Díky tomu je možné po odladění programu logovací systém vypnout a standardní překladače odstraní z kódu tato volání.

Pro účely ladění jsou všechny řešící třídy odvozeny od třídy **PrintableInterface**, která definuje funkce k vypsání hodnot. Do ladícího souboru jsou psány informace o provedení cílů - jak se změnil model, změny hodnot domén a která podmínka byla důvodem této změny. Dalšími informacemi jsou aktuální hodnoty domén nebo aktuální hloubka prohledávání.

Ladící soubor sice obsahuje mnoho potřebných informací, ale není v současné podobě příliš přehledný. V budoucnosti je záměrem vytvořit jednoduchý program, pomocí kterého by z tohoto logovacího souboru bylo možné vytvořit grafickou representaci a tu poté zobrazovat (například nakreslení grafu prohledávání nebo vypsání přidaných podmínek a aktuálních domén).

A.3.9 Přehled souborů a jejich organizace, kompilace, unit testy

Implementační sobory balíčku jsou:

- *src/cp/constraints.cpp*
- *src/cp/cp.cpp*
- *src/cp/model.cpp*
- *src/cp/problem.cpp*
- *src/cp/search.cpp*
- *src/cp/solver.cpp*

Kromě těchto souborů se balíček skládá z několika desítek hlavičkových souborů. Všechny soubory jsou v adresáři include/cp. Tento adresář obsahuje následující podadresáře:

- *bool* - definice a implementace boolovských podmínek a domén
- *constraints* - definice základních podmínek pro odvozování dalších tříd

- *domain* - definice základních tříd využívaných doménami
- *integer* - definice a implementace celočíselných domén a celočíselných podmínek
- *memory* - definice a implementace paměťového subsystému
- *numeric* - definice a implementace numerických podmínek (rovností, nerovností)
- *propagation* - definice a implementace propagační fronty a posluchačů
- *search* - definice prohledávacích strategií a cílů

Součástí balíčku jsou unit testy. Nachází se v adresáři *app/src/cp/tests*. Je možné vytvořit aplikaci, která je všechny spustí. Tato aplikace je definována v adresáři *app* v projektovém souboru *tests.pro*. Překlad a spuštění je totožné s překladem hlavní aplikace, které je popsané v části C. Pouze místo souboru*app.pro* je použit soubor *tests.pro*. Součástí těchto testů je řešení sudoku, testy paměťových tříd, výrazů nebo problému výroby (tento test pochází ze staršího vývojového období). V souboru s testem výrazů je možné se seznámit se základními prvky modelování.

A.4 Balíček cp_airport

Tento balíček je mostem mezi GUI aplikací a výpočetním jádrem - balíčkem cp. Hlavní hlavičkový soubor *include/cp_airport/cp_airport.hpp* definuje pouze tyto funkce a třídy.

```
class Database{
protected:
    Database();
    Database & operator=( const Database & other );
public:
    virtual int intValue( const std::string & key ) const = 0;
    virtual const std::string & strValue( const std::string & key ) const = 0;
};


const Database & database();

void solve( std::istream & input );
/*<! ze vstupního proudu načte zadání rozvrhovacího problému
```

```
  (soupis požadovaných operací) a pokusí se jej vyřešit, je očekáváno
  že byly nastaveny výstupní proudy a načtena databáze. Při syntaktické chybě
 ve vstupním souboru je vyhozena výjímka s informací o řádce na které je chyba  */


struct IOStream{
        virtual ~IOStream(){}
    virtual void print( const std::string & data ) = 0;
};
/*< Rozhraní definující výstupní proudy.*/


void setInfoOutput( IOStream * toStrem );
/*< Nastaví výstupní proud pro vypisování objektivní
funkce postupně nacházených řešení. */

void setOutput( IOStream * toStream );
/*< Nastaví výstupní proud pro řešení. */

bool endSearch();
/*< Pro účely GUI aplikace je možné asynchronně přerušit prohledávání.
Jestli se má prohledávání přerušit je zjištěno touto funkcí. Tato funkce
 není definována v tomto balíčku. */
```

Toto je veřejné rozhraní zde popisovaného balíčku. Z pohledu tohoto balíčku nezáleží na tom, jestli řešení vypisuje na standartní výstup nebo na grafický ovládací prvek.

Tím, že jsme takto definovali veřejné rozhraní a jedná se o poměrně malý balíček, není nutné dodržovat všechny zásady zapouzdření. (Zapouzdření v rámci balíčku již bylo definováno veřejným hlavičkovým souborem.) Proto pro jednodušší manipulaci s daty v některých případech nedodržuje striktní dělení dat na privátní a veřejná.

Uvnitř balíčku jsou definovány třídy **Config, Flight, ResourceManager, StringParser**. Třída **Config** rozšiřuje třídu *Database*. Během jejího vytvoření je načten konfigurační soubor, z kterého lze určit informace o potřebných datech, jako je umístění souboru se scénou nebo souboru s popisem zdrojů pro rozvrhovací systém nebo velikost zvolené jednotky pro dělení času do intervalů. Tato třída tedy definuje prostředí celého programu - existuje právě jedna instance v celém programu.

Před použitím hlavní metody *solve* musí být nastaveny výstupní datové proudy. Po zavolání metody *solve* je pokaždé vytvořena třída **ResourceManager**, která dynamicky vytváří potřebné zdroje podle požadavků letů ze statické da-

tabáze zdrojů během načítání vstupního souboru v metodě *readModel*. Zde je převeden vstupní soubor do modelu proměnných a podmínek (třídy **cp::Model**). Model je vytvářen přesně tak, jak byl popsán v kapitole 3. Pro každý vstupní let je vytvořena instance třídy **Flight**, která zastupuje průchod jednoho letadla systémem. Pokud je během převádění vstupu na model zjištěna syntaktická chyba, je vygenerována výjimka s chybovým hlášením o tomto problému. Tato výjimka obsahuje také informaci o řádce, na které k problému došlo. Všechny tyto funkce jsou definované v souboru *src/cp_airport/airport.cpp*.

Po úspěšném vytvoření modelu je v privátní funkci search (obsažené v souboru *src/cp_airport/airportSearch.cpp*) vytvořena v práci popsaná prohledávací strategie, která je kombinací heuristické metody (třída **HeuristicSearch**) a úplné strategie (třída **FullSearch** - implementace algoritmu 5). Poté je vytvořena instance třídy *cp::Solver* a je spuštěno prohledávání. Implementací heuristiky pro výběr proměnné zastupující start na ranveji je třída **VariableWithLowestLBValueSelector**. Implementací omezující strategie popsané v 3.5.1 (procedura větví a mezí) je třída **MultiOptimize**.

Protože požadujeme, aby bylo možné kdykoliv prohledávání zastavit, je pomocí funkce *endSearch* při každém návratu kontrolováno, zda nemá prohledávání být ukončeno. Toto je implementováno v omezující funkci pro branch and bound algoritmus. Pokud bylo zjištěno, že je požadováno ukončení hledání, je vygenerována výjimka, která ihned ukončuje prohledávání. V rámci balíčku je odchycena a jako výsledek prohledávání je vráceno dosavadní nejlepší nalezené řešení (pokud nějaké existuje).

V adresáři *app* na přiloženém CD je projektový soubor *cp_airport.pro*, který popisuje soubory potřebné k sestavení rozvrhovače jako programu z příkazové řádky. Pro sestavení a spuštění jsou potřeba udělat stejné kroky jako pro hlavní aplikaci, které jsou popsané v C. Jediným rozdílem je jiný název projektového souboru a vytvořeného adresáře. Instalace Qt není potřebná pro běh výsledné aplikace, ale umožňuje nezávisle na platformě provést stejné kroky k sestavení (utility qmake).

A.5 Balíček qt

Velká část balíčku qt se zabývá čtením vstupních souborů z formátu xml, k jeho čtení je použito metody SAX. Pro snadné extrahování dat z těchto souborů byla pro každý formát implementována hiearchie tříd, která svojí strukturou popisuje zadaný xml soubor. Všechny tyto třídy jsou potomkem třídy **Xml::Parser**.

Při načítání souboru letištní scény je nejdříve nutné vytvořit projekci na 2D plátno, protože pozice objektů v souboru scény jsou zapsané pomocí zeměpisných šířek a zeměpisných délek. Pro účely práce byla vybrána jednoduchá metoda, ve které je zanedbáno zakřivení Země. Jelikož se vždy jedná o plochu jednoho letiště – tedy poměrně malou plochu, nejsou pozorovatelné nepřesnosti. Načtení scény letiště probíhá ve dvou krocích.V prvním je soubor skenován pro pozice vrcholů v něm obsažených, na závěr této fáze je vytvořena projekce – třída **guia_Map2DPojection**. V druhém kroku jsou již na základě vytvořené projekce do scény přidávány objekty obsažené ve vstupním souboru. Na závěr je trochu vylepšen vzhled letiště, pomocí vlastního nakreslení obrysů cest.

Pro účely animace byla vytvořena abstraktní třída **Animation**, od které jsou pro pohyb letadel odvozeny třídy **RotationAnimation** a **PathAnimation**. Celou animovací smyčku zajišťuje třída **AnimationEngine**. Pro koordinaci zobrazování a posouvání času s aktuálním stavem **AnimationEngine** byla vytvořena třída **guia_ClockMediator**. Definice a implementace těchto tříd je v souborech *guia_animation.cpp, guia_Animation.hpp*

Protože výpočet rozvrhu může být dlouhotrvající operace, je prováděn ve vlastním vlákně. Definice a implementace tohoto vlákna je v souborech *guia_SearchThread.hpp*, *guia_search.cpp*. Ke komunikaci s vláknem obsluhy událostí je využito standardních technik pomocí mutexů.

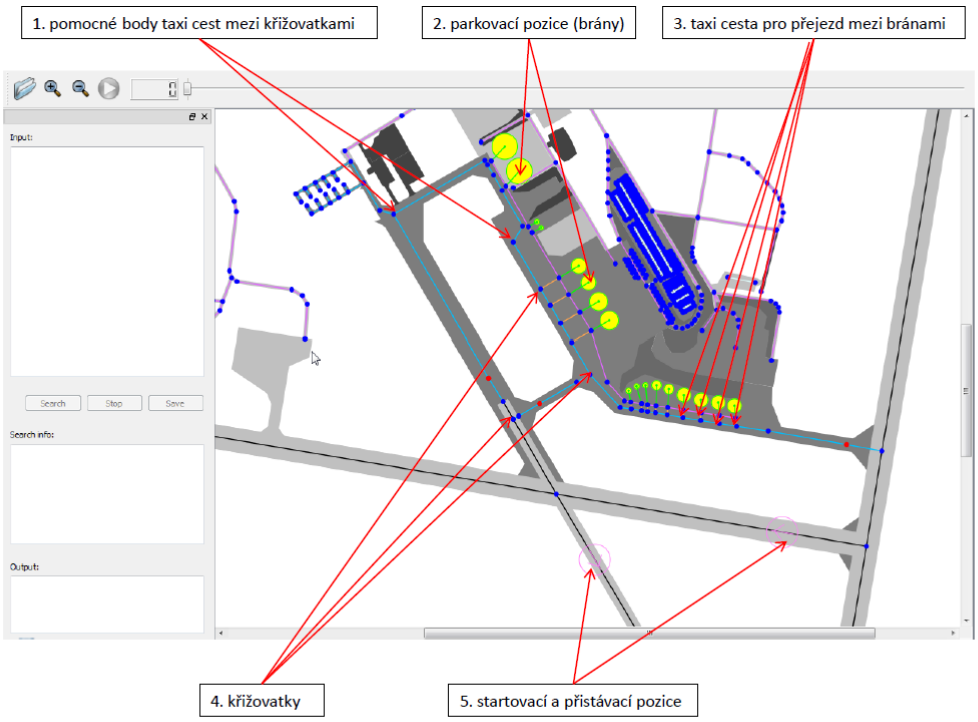
Příloha B

Vstupní letištní soubory

Jako soubory se strukturou letiště pro zobrazování byly vybrány soubory z hry Microsoft Flight Simulator [26]. Tyto soubory jsou pro účely této hry uložené v binárním souboru ve speciálním formátu bgl. Ovšem před překladem do tohoto formátu jsou representovány jako xml soubory se strukturou popsanou v [25]. Vstupním souborem pro zobrazování je tedy tento xml soubor. Existují programy, které dovolují editovat takováto letiště [27]. Hra má poměrně velkou fanouškovskou základnu a tito fanoušci vytvářejí bgl soubory s různými letišti. Tyto soubory pak vystavují na internetu, kde je nabízejí k volnému stažení. Velkou nevýhodou formátu bgl je, že existují různé verze a tento formát není dobře dokumentován. Ovšem ani toto nebylo pro fanoušky hry překážkou a vytvořili programy, které dokážou převést bgl soubory zpět do xml formátu - například BGLAnalyze, Bgl2Xml. Tyto programy převedou vstupní bgl soubory zpět do xml. Touto cestou tedy bylo získáno ukázkové letiště. Dále tento soubor budeme nazývat *bgl.xml* souborem.

Tento výběr není z důvodů špatné dokumentace optimální. Byly zkoušeny i jiné, volnější alternativy – soubory z Open Street Map, Airplane X. Ovšem žádný z těchto pokusů nevypadal dostatečně dobře při zobrazení nebo měl jiné znatelné nevýhody. Výhodou zvoleného postupu je, že letiště jsou vytvořena fanoušky pro letectví a jsou tudíž věrohodná – odpovídají skutečným parametrům letišť.

Pro výpočet rozvhu potřebuje subsystém cp_airport celkově dva vstupní soubory – soubor s popisem operací a soubor s popisem zdrojů. Pro simulaci výsledků je kromě souboru bgl.xml definující scénu potřeba soubor s informacemi o tom, jak mají být letadla využívající zdroje animována – tedy soubor s propojením scény se zdroji.



Obrázek B.1: Scéna letiště

B.1 Soubor popisující strukturu letiště

Vstupním souborem pro zobrazení scény je soubor s formátem bgl.xml. Po načtení tohoto souboru je vytvořena scéna letiště. Obrázek B.1 zobrazuje tuto scénu. Kostra scény se skládá z bodů a spojnic těchto bodů. Tyto body jsou zpravidla nakresleny tmavě modrou barvou. Spojnice těchto bodů representují cesty na letišti. Cesty, které jsou používány pouze pro taxi operace, jsou zobrazeny světle modrou barvou. Cesty, které jsou používány jak pro taxi operace, tak pro vzlety a přílety, jsou zobrazeny černou barvou. Žluté kruhy representují parkovací pozice. Kružnice se šipkou uvnitř, která je ve směru ranveje, representuje startovací (přistávací) pozici.

Pro definování modelu povrchu je klíčovým bodem určit, které z vrcholů representují křižovatky. Toto nebylo možné extrahovat ze vstupního souboru přímo (protože například startovní pozice nejsou zahrnuty v kostře scény), a tudíž je potřeba určit křižovatky ručně. Na obrázku B.1 jsou s popiskem 4 zobrazeny některé z těchto křižovatek. Křižovatkou je místo křížení výjezdu z parkovací pozice s taxi cestou (na obrázku zobrazeno šipkou od popisku 4, která je nejblíže k parkovací pozici) nebo křížení více než jedné taxi cesty (ostatní šipky od popisku 4) nebo startovací (přistávací) pozice. Ne všechny body na povrchu jsou

křižovatkami. Šipka od popisku 1 na obrázku B.1 ukazuje, že některé body povrchu nejsou křižovatkami.

Taxi cesta z jednoho místa na jiné místo je tedy definována posloupností bodů na povrchu. Tyto body vždy začínají a končí křižovatkou. Pro rozvrhování jsou důležité pouze křižovatky, protože ty dělí jednotlivé části taxi cesty na hranové zdroje. Ovšem pro zobrazení výsledků rozvrhování jsou potřeba i pomocné body mezi dvěmi křižovatkami.

Ve vstupním souboru bgl.xml jsou jednotlivé body povrchu representovány jako vrcholy se souřadnicemi zeměpisné délky a zeměpisné šířky. Každý z těchto vrcholů má jednoznačný identifikátor, který je použit k popisu taxicest, parkovacích pozic a ranvejí. Každé parkovací místo má svoje jméno – například gate1, gate2, parking11. Ranveje mají standardně jména podle úhlu, kam míří – tedy například ranvej r06 míří ve směru 60 stupňů. Startovací pozice mají jména podle své ranveje – startovací pozice pro ranvej r06 je s06, při použití ranveje z druhé strany jsou to r24 a s24 (o 180 stupňů více). Jména jednotlivých bodů, ranvejí, parkovacích míst a dalších informací je ve scéně možné získat pomocí podržení myši nad daným objektem.

B.2 Formát souboru s popisem operací

Klíčovým souborem je soubor s popisem operací. Uživatel zadává a mění právě tyto soubory. Proto je důležité, aby byl co nejjednodušší. Pro ostatní soubory jednoduchost není tak klíčovým požadavkem, protože jsou pro každé letiště vytvořeny pouze jednou a dále používány beze změn. Tento výpis ukazuje formát vstupního souboru s popisem operací:

```
1    #horizont rozvrhování - všechny aktivity mají skončit před touto hodnotou
2    horizont 120
3
4    #omezení doby hledání výsledku na 10 vteřin
5   time_limit 10
6
7  # definice letu s malým letadlem, nejdříve potřebuje ranvej k přistávání
8  # v době 30 až 100
9  # poté z místa příletu taxicestu do místa parkování parking2, poté
10 # je až do konce zaparkován na místě parking2
11  flight let1 Small
12  runway r06l 30 100
13  taxi taxi_s06_parking2_2
14  parking parking2
15
```

```
16 # definice letu s velkým letadlem, začíná v bráně 1,
17 # potřebuje se dopravit do místa startu s_06  pomocí taxicesty
18 # taxi_gate1_s06_1, z koncového místa taxicesty odlétá na ranveji r06
19  flight let2 Large
20  parking gate1
21  taxi taxi_gate1_s06_1
22  runway r06
23
24 # definice letu s malým letadlem, začíná v bráně 2,
25 # potřebuje se dopravit do místa startu s_06  pomocí taxicesty
26 # taxi_gate2_s06_1, z koncového místa taxicesty odlétá na ranveji r06
27  flight let3 Small
28  parking gate2
29  taxi taxi_gate2_s06_1
30  runway r06
31
32 # podmínka, že má let3 využít ranvej dříve než let2
33  let3 << let2
34
35 # Let 4 pouze potřebuje přejet z místa parkování gate3
36 # do místa parkování parking1 za využití taxicesty taxi_gate3_parking1_1
37  flight let4 Large
38  parking gate3
39  tow taxi_gate3_parking1_1
40  parking parking1
```

Nejdůležitější informace jsou obsaženy v komentářových řádcích výpisu. Nyní bude vysvětleno hlavně značení. Názvy zdrojů jsou jednoslovné tedy například gate 2 je nazvána jako gate2. Všechny ranveje začínají písmenkem r a dále pokračují dvojicí celočíselných hodnot. Pokud je ranvej využívána k přistávání, končí písmenkem l (land). Každá ranvej má body startu. Tyto body začínají písmenkem s a pokračují číslem ranveje (ranvej r06 má startovní bod s06). Taxicesty začínají slovem taxi a následně pokračují informací, odkud vedou a který bod je koncem cesty. Nakonec je v jejich názvu zadaná hodnota, kolikátá nejkratší cesta má být použita (obvykle jsou dvě možnosti). Jednotlivé hodnoty jsou odděleny podtržítkem (tedy taxi_gate3_parking1_1 je příkaz použij nejkratší cestu mezi parkovací pozicí gate3 a parking1). Podmínka seřazení letů na ranveji určuje, že jedno letadlo musí začít používat ranvej dříve než druhé. Tato podmínka má význam především pro dvě letadla využívající stejnou ranvej. Klíčové slovo tow znamená přejezd mezi dvěmi parkovacími místy. Za každou aktivitou mohou být zadány nejdřívější startovní a nejpozdější koncové časy pro danou aktivitu. Pokud nejsou zadány, je nejdřívější startovní čas aktivity nastaven na 0 a nejpozdější koncový

čas nastaven na horizont.

Sekvence možných posloupností operací je definována v části 3.2. V ukázkovém vstupu jsou všechny tyto sekvence použity. V souborech je možné vytvářet komentářové řádky pomocí znaku # a nechávat volné řádky. Znak # musí být na komentářovém řádku jako první hodnota. Prvními dvěmi nekomentářovými hodnotami v souboru musí být horizont plánování a časový limit na prohledávání. Na jejich pořadí nezáleží.

Čas byl rozdělen na intervaly po 10 vteřinách – vstupní informace kromě časového limitu jsou v tomto formátu. Všechny netextové hodnoty jsou celočíselné. Pro změnu defaultního intervalu 10 vteřin je potřeba změnit hodnotu v konfiguračním souboru a provést další akce, které jsou popsány v části B.6.

Zvolenou připonou vstupního formátu je přípona .int.txt. Příklady vstupních souborů jsou na přiloženém CD v adresáři *app/files/CYXS/examples*.

B.3 Formát výstupního souboru rozvrhování

Výstupem rozvrhovacího subsystému (cp_airport) je xml soubor. Jeho formát bude ukázán v následujícím výpise.

```
<Result>
<Schedule value="feasible">
   <Flight name="let4">
      <Resource type="parking" name="gate3">
        <Sub from="0" to="5"/>
      </Resource>
      <Resource type="taxi" name="taxi_gate3_parking1_1">
        <Sub from="5" to="6"/>
        <Sub from="6" to="13"/>
      </Resource>
      <Resource type="parking" name="parking1">
        <Sub from="13" to="15"/>
      </Resource>
   </Flight>
   <Flight name="let5">
      <Resource type="parking" name="parking2">
        <Sub from="0" to="2"/>
      </Resource>
      <Resource type="taxi" name="taxi_parking2_gate3_1">
        <Sub from="2" to="3"/>
        <Sub from="3" to="6"/>
        <Sub from="6" to="11"/>
      </Resource>
```

```
      <Resource type="parking" name="gate3">
        <Sub from="11" to="15"/>
      </Resource>
   </Flight>
</Schedule>
</Result>
```

Hlavním elementem je Schedule, jehož atribut value může mít jednu z následujících tří hodnot:

- feasible - podařilo se v zadaném časovém intervalu rozvrh najít a hodnoty uvnitř element Schedule definují nejlepší nalezený výsledek
- unfeasible - daný problém je neřešitelný - žádný rozvrh s požadovanými vlastnostmi neexistuje
- unknown - v zadaném čase se nepodařilo ani nalézt řešení, ani dokázat, že je problém neřešitelný

Pokud byl rozvrh nalezen, obsahuje element Schedule pro každý vstupní let právě jeden element Flight. Ten má jako potomky jednotlivé zdroje využívané daným letem. Každý tento zdroj má jeden nebo více potomků – elementů Sub. Více potomků má pouze pro taxi zdroje – každý z nich definuje seznam časů strávených na jednom hranovém zdroji, z kterého se taxi cesta skládá. Jsou uspořádány v pořadí pohybu letadla. Jinými slovy, hodnoty atributů to elementů Sub definují časy projíždění jednotlivými křižovatkami. Pro ostatní zdroje obsahuje element Resource pouze jednu informaci – dobu strávenou na zdroji (ranveji, nebo parkovací pozici).

Zvolenou připonou výstupního formátu je přípona .out.xml.

Příklady vstupních a k nim odpovídajících výstupních souborů jsou na přiloženém CD v adresáři *app/files/CYXS/examples*.

B.4 Formát souboru propojujícího zdroje na letišti se scénou

Tento soubor je pro aplikaci klíčovým interním souborem. Je potřeba jej vytvořit pouze jednou pro každé letiště a poté je aplikací neustále využíván. Definuje, jaké zdroje jsou na letišti přítomny. Pomocí indexů bodů povrchu letiště jsou tyto zdroje popsány a provázány se scénou pro vytvoření animací výsledných rozvrhů. Z tohoto souboru je generován textový soubor pro popis zdrojů pro rozvrhovací subsystém. Jedná se o xml soubor s velice jednoduchou strukturou:

```
<Resources>
        <!-- parking -->
        <Resource name="gate4" type="parking"
                path="p1:543:j529"
        />
        <!-- runways -->
        <Resource name="r06" type="runway"
                path="s06:j11:s24:j10"
                from="s06"
                to="s24"
        />

        <!--  taxiways  -->
   <Resource name="taxi_parking2_parking1_1"
             type="taxi" from="parking2" to="parking1"
             path="j531:j530"/>
</resources>
```

Jak již bylo řečeno, každý bod na povrchu má jednoznačný index. K některým z těchto indexů je pro účely tohoto souboru přidán prefix, který určuje význam těchto bodů. Body, které jsou parkovacími pozicemi mají prefix *p*. Body, které jsou startovacími pozicemi, mají prefix *s* a body, které jsou křižovatkami, mají prefix *j*. Pomocné body pro taxi cesty nemají prefix žádný. Každý element Resource obsahuje několik atributů – povinnými atributy jsou jméno zdroje (name), typ zdroje (type) a cesta zdroje (path). V případě typu zdroje parking a taxi určuje atribut path body povrchu, po kterých se letadlo pohybuje. V případě typu taxi je význam zřejmý. V případě typu parking se jedná o cestu z parkovacího bodu na první křižovatku. V případě typu zdroje runway je v cestě zdroje sekvence křižovatkových zdrojů, která určuje hranové a křižovatkové zdroje, které jsou na ranveji. Atributy from a to určují, jakým způsobem má být animován přílet (odlet) letadla.

Každý zdroj, který je na letišti, má záznam se svým jménem v tomto souboru. Z těchto informací jsou vypočítány na základě vstupního souboru se scénou letiště vzdálenosti mezi křižovatkami a z nich poté odvozeny minimální doby přejezdů letadel po hranových zdrojích. Následně je možné vygenerovat soubor s popisem zdrojů pro rozvrhovací subsystém. Tyto informace jsou také využity k následné simulaci nalezených řešení (pro pohyb letadel po povrchu letiště).

Celý soubor je možné nalézt na přiloženém CD v adresáři *app/files/CYXS* pod názvem *scene_resources.xml*.

B.5 Formát souboru popisujícího zdroje na letišti pro rozvrhovací subsystém

Tento soubor je možné na základě konfiguračního souboru (B.6) a souboru popsaného v předchozí časti (části B.4) vygenerovat pomocí hlavní aplikace – je vygenerován, pokud je hlavní aplikace spuštěna s argumentem `-generate`. V části B.1 bylo řečeno, jakým způsobem jsou definovány taxicesty – a sice jako sekvence bodů na povrchu. Tyto body mají jednoznačný index. Pro rozvrhovací subsystém je potřeba vytvořit soupis zdrojů, které si uloží do své databáze a z něj následně interpretuje uživatelovy požadavky na používané zdroje. Pro tyto účely byl přímočaře z representace vytvořen tento formát. Následující výpis je krátký výňatek z tohoto souboru:

```
#type     name           aliases          time   capacity
parking   gate4          0                2      1
resource  j534           j534             4      1
resource  j14j15         j14s15:s15j14    5      2
runway    r19l           j16:j11:s19:j14  0      0
taxi      taxi_s24_s06_1 s24:j11:s06      0      0
```

Nejedná se o xml soubor, protože by pro jeho zpracování bylo třeba Qt, čemuž jsme se chtěli vyhnout (aby byl výsledný program použitelný i bez nainstalovaného Qt jako program z příkazového řádku). Soubor je rozdělen do sloupců. První sloupec určuje typ popisovaného zdroje. Druhý sloupec určuje jméno zdroje. Na základě typu zdroje jsou různě interpretovány poslední dva sloupce. Pokud je ve sloupci hodnota 0, znamená to, že daný záznam pro tento sloupec hodnotu nemá. Pro oddělení hodnot v jednotlivých sloupcích je použito dvojtečky.

Pokud je typ zdroje resource, jedná se buď o křižovatkový zdroj, nebo o hranový zdroj. Hranový zdroj (spojnice dvou křižovatek) je pojmenován jako sekvence indexů bodů povrchu, přes které vede. V případě hranového zdroje, je potřeba vytvořit jeden zdroj, který je možné používat v obou směrech. Proto ve sloupečku aliases jsou obě dvě varianty názvu tohoto zdroje oddělené dvojtečkou. Následují sloupečky času, který potřebují letadla k překonání vzdálenosti mezi dvěmi koncovými křižovatkami hranového zdroje a dále sloupečky počtů letadel, která mohou na zdroji jet za sebou (kapacita zdroje). V případě, že se jedná o křižovatkový zdroj, je situace jednodušší. Na základě poze jediné hodnoty ve sloupečku aliases je vytvořen jedien zdroj s kapacitou 1.

Pokud je typ zdroje runway, je ve sloupečku aliases vypsána množina zdrojů, které je potřeba během přistání nebo vzletu blokovat.

Pokud je typ zdroje taxiway, je ve sloupečku aliases vypsána sekvence zdrojů,

kterou je potřeba využít při použití této taxicesty.

Hodnota zdroje parking zaručuje vytvoření jednoho zdroje s jednotkovou kapacitou.

Celý soubor se nachází na přiloženém CD v adresáři *app/files/CYXS* pod názvem *scene_resources.txt*.

B.6 Konfigurační soubor

Konfigurační soubor slouží k zadání měnitelných parametrů pro rozvrhování. Všechny časové hodnoty jsou zadané v sekundách. Jsou jimi velikost časové jednotky (proměnná unit), doba pro blokování křižovatky (proměnná junction_time), minimální separační doba pro vzlet nebo přílet lehčího letadla po těžším (proměnná big_small) a proměnná určující separační časy na stejné ranveji při mixování příletů a odletů (proměnná runway_operation_mix_time).

Formát tohoto souboru je klíč hodnota. Tedy například:

```
unit 10
```

Po změně některé z těchto hodnot je potřeba spustit aplikaci s parametrem `-generate`, který na základě hodnot v tomto konfiguračním souboru vygeneruje nový soubor s popisem zdrojů pro rozrvhovací submodul.

Součástí konfiguračního souboru je také proměnná ICAO (International Civil Aviation Organization - jednoznačný identifikátor letiště) definující, které letiště se má načíst. V současné době je implementováno pouze jedno letiště – letiště Prince-George v Kanadě. Pro rozšíření na další letiště je potřeba vytvořit v adresáři *app/files* adresář s názvem identifikátoru icao a v něm vytvořit nebo vygenerovat soubory popsané výše a pojmenovat je stejnými jmény jako v implementovaném letišti. Poté pomocí změny hodnoty proměnné icao bude automaticky načítáno toto letiště.

Konfigurační soubor je možné nalézt na přiloženém CD v adresáři *app* pod názvem *config.cfg*.

Příloha C

Uživatelská dokumentace

V této části je popsána instalace a ovládání vzniklé aplikace, která slouží k ilustrování zvoleného přístupu.

C.1 Instalace a spuštění

Instalaci a spuštění aplikace na operačním systému Linux je možné shrnout v následujících krocích:

1. Pro vytvoření a spuštění aplikace je potřeba mít nainstalované grafické knihovny Qt [20] ve verzi 4.7 nebo vyšší. Instalační soubory této knihovny je možné nalézt na internetové adrese [21].

2. Zkopírujte adresář *app* z přiloženého CD do lokálního adresáře, ve kterém chcete mít aplikaci nainstalovanou.

3. V příkazové řádce se přesuňte do lokálního adresáře z předchozího kroku.

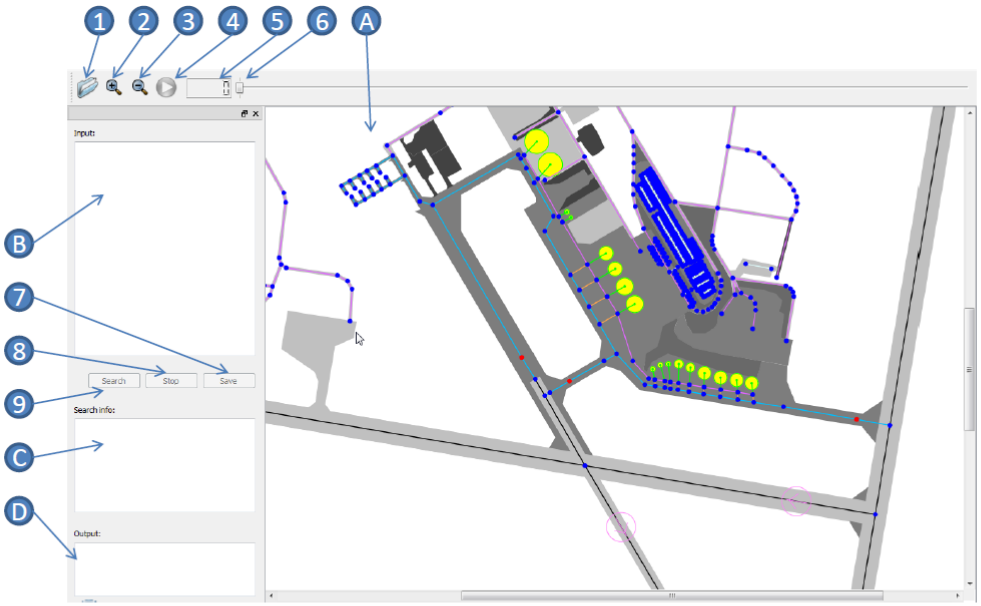
4. Zadejte příkazy `mkdir app_bin` a `cd app_bin`.

5. Zadejte příkaz `qmake ../app/app.pro`. Tento příkaz vytvoří makefile skript v závislosti na konfiguraci Qt vytvořené v prvním kroku.

6. Zadejte příkaz `make`.

7. Zadejte příkaz `./app`.

Nejčastějším překladačem na operačním systému Windows je Microsoft Visual Studio, proto bude popsána instalace pro něj. Kroky 1 - 5 jsou identické s instalací na Linuxu. Pro krok 6 je místo make použita utilita nmake - obdoba make pro



Obrázek C.1: Hlavní okno aplikace

Windows. Cesta k ní obvykle nebývá v proměnné PATH. Je možné využít Microsoft Visual Studio Command Prompt, který cestu k tomuto programu nastaví (po spuštění Visual Studia je možné Microsoft Visual Studio Command Prompt spustit pomocí záložky tools). Příkaz v 6. kroku je `nmake release`. Příkaz pro 7. krok je `"release/app.exe"`.

Aplikace předpokládá, že je spouštěna z adresáře, který je hiearchicky na stejné úrovni jako adresář *app* (tedy například z vytvořeného *app_bin*). Dále aplikace předpokládá, že jsou v proměnné PATH cesty k dll knihovnám, které byly vytvořeny v 1. kroce instalace – tento požadavek je automaticky nakonfigurován při instalování Qt.

Po úspěšném provedení 7. kroku se objeví hlavní okno aplikace.

C.2 Popis uživatelského prostředí a ovládání aplikace

Na obrázku C.1 je zobrazeno hlavní okno aplikace. V oblasti A je zobrazena scéna letiště. Popis objektů, které je možné nalézt v této scéně, je popsán v části B.1. Tuto scénu je možné pomocí tlačítka 2 přibližovat (klávesovou zkratkou je

písmeno `i` nebo `+`), případně pomocí tlačítka 3 oddalovat (klávesová zkratka je písmeno `o` nebo `-`). Obsahem této sekce je návod, jak ovládat tuto aplikaci.

C.2.1 Zadání a řešení vstupního rozvrhovacího problému

Pro zadání rozvrhovacího problému slouží v hlavním okně aplikace oblast B, do které je možné tento problém ve formátu popsaném v části B.2 zadat. Je možné také vstupní problém pomocí tlačítka 1 načíst ze souboru. Po jeho stisknutí se objeví okno se standardním dialogem pro otevírání souborů. Je nastaven filtr pro příponu vstupních souborů na .in.txt. Adresář, ze kterého jsou nejprve nabídnuty soubory, je adresář se vzorovými zadáními.

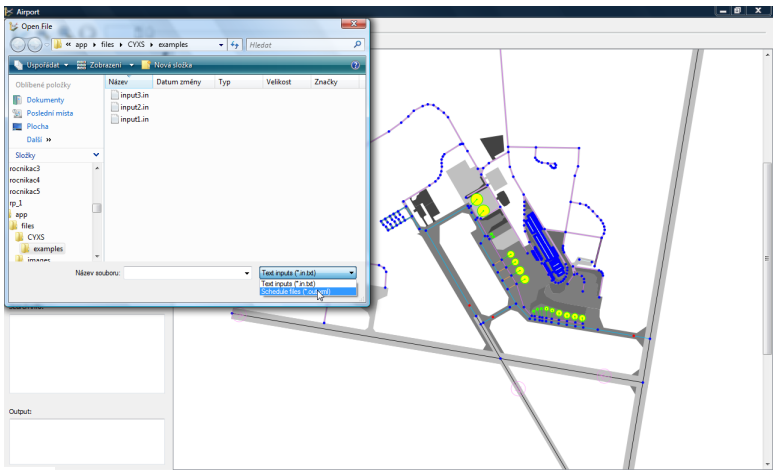
Po zadání problému je umožněno zahájení prohledávání. Tlačítkem 9 je prohledávání spuštěno. Oblast C slouží k zobrazování informací o prohledávání. Jsou zde postupně vypisovány nalezené hodnoty řešení a na závěr je zobrazena statistika prohledávání. Pokud došlo při zadání problému k syntaktické chybě, není hledání započato a do této oblasti je vypsáno chybové hlášení s řádkou, kde k chybě ve vstupním souboru došlo. Během prohledávání je možné pomocí tlačítka 8 kdykoliv řešící algoritmus přerušit. Algoritmus vrátí buď nejlepší dosud nalezené řešení, nebo informaci, že se žádné řešení zatím nepodařilo nalézt. Toto řešení je zobrazeno v oblasti D a je ve formátu popsaném v části B.3. Pokud bylo řešení nalezeno, je automaticky vytvořena animace nalezeného řešení. Spouštěním a ovládáním této animace se zabývá následující část. Výsledek je možné pomocí tlačítka 7 uložit do výstupního souboru. Tyto soubory mají příponu .out.xml.

Velikosti jednotlivých oblastí lze pomocí myši standardním způsobem měnit.

C.2.2 Simulace výsledků rozvrhování

Pro simulaci výsledků je nejdříve nutné získat řešení rozvrhovacího problému. Buď je možné toto řešení získat cestou popsanou v minulé části, nebo je možné načíst již uložené řešení. Pro načtení uloženého řešení slouží tlačítko 1. Ve filtru pro přípony souborů je potřeba nastavit přípony řešení – .out.xml. Obrázek C.2 ilustruje tuto akci. Po získání řešení rozvrhovacího problému je automaticky vytvořena simulace, která jej provádí. Spuštění této simulace je provedeno pomocí tlačítka 4. Toto tlačítko je také možné použít k pozastavení simulace. Pomocí stisku a posunování tlačítka 6 je možné posouvat časovou osu. V okénku 5 se zobrazuje aktuální hodnota času v simulaci.

Jako základní jednotka času bylo zvoleno 10 vteřin reálného času. Změna tohoto nastavení je možná provést v konfiguračním souboru, jehož umístění a



Obrázek C.2: Zvolení fitru na výstupní soubory

obsah je popsán v části B.6.

Příloha D

Obsah přiloženého CD

K bakalářské práci je přiloženo CD, které obsahuje tuto práci v digitální podobě, zdrojové kódy aplikace a vygenerovanou programátorskou dokumentaci.

Literatura

[1] Francesca Rossi,Peter van Beek, Toby Walsh, et al, *Handbook of Constraint Programming* 1^{th} Edition.

[2] Philippe Baptiste, Claude Le Pape, Wim Nuijten, *Constraint-Based Scheduling Applying Constraint Programming to Scheduling Problems*, Kluwer Academic Publishers, 2001

[3] Malik Ghallab, Dana Nau, Paolo Traverso, *Automated planning Theory and Practise*, Morgan Kaufmann, 2004

[4] Wilhelmus Petronella Maria Nuijten, *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*, PhD Thesis Eindhoveh University of Technology, 1994

[5] Ulrich Dorndorf, *Project Scheduling with Time Windows - From Theory to Applications*, Physica-Verlag Heidelberg New York, 2002

[6] Klaus Neumann, Christoph Schwindt, Jürgen Zimmermann, *Project Scheduling with Time Windows and Scarce Resourcess*, Springer-Verlag Berlin Heidelberg NewYork 2003

[7] Petr Vilím, *Global constraints in scheduling*, PhD thesis, Charles University in Prague - Faculty of Mathematics and Physics, 2007

[8] Francis R. Carr, *Robust Decision-Support Tools for Airport Surface Traffics*, PhD Thesis, Massachusetts Institute of Technology, 2004

[9] Kai-Pei Chen, Marvin S. Lee, P. Simin Pulat, Scott A. Moses *The shifting bottleneck procedure for job-shops with parallel machines*, Int. J. Industrial and Systems Engineering, Vol. 1, Nos. 1/2, 2006

[10] H.H. Hesselink, S. Paul *Planning aircraft movements on airports with constraint satisfaction*, 1998

[11] P. van Leeuwen, H.H. Hesselink and J.H.T. Rohlingl *Scheduling aircraft using constraint satisfaction*, 2002

[12] Pim van Leeuwen, Nicolas van Hanxleden Houwert *Scheduling aircraft using constraint relaxation*, 1998

[13] Roman Barták, *On-line Guide to Constraint Programming*, http://kti.mff.cuni.cz/ bartak/constraints/, 1998

[14] W.D. Harvey and M.L. Ginsberg, *Limited Discrepancy Search*, in Proceedings of IJCAI95, pages 607-613, 1995

[15] Toby Walsh, *Depth-bounded Discrepancy Search*, in Proceedings of IJCAI95, pages 607-613, 1995

[16] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994

[17] Ioannis D. Anagnostakis, John-Paul Clarke, *A Multi-objective, Decomposition-Based Algorithm, Design Methology and its Application to Runway Operations Planning*

[18] John Lakos, *Large-Scale C++ Software Design*, Addison Wesley, 1996

[19] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley, 2001

[20] *Framework Qt*, http://qt.nokia.com/products/

[21] *Framework Qt - instalační soubory*, http://qt.nokia.com/products/http://qt.nokia.com/downloads

[22] *Gecode*, http://www.gecode.org/

[23] *Ilog solver*, http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/

[24] *Choco solver*, http://www.emn.fr/z-info/choco-solver/

[25] *Popis xml formátu pro načítání letiště*, http://msdn.microsoft.com/en-us/library/cc526978.aspx

[26] *Microsoft Flight Simulator X*, http://www.microsoft.com/games/flightsimulatorx/

[27] *Airport Facilator X*, http://www.flight1.com/products.asp?product=afxv1

[28] *Thread-Caching Malloc*,
http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[29] *Expression templates*,
http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Expression-template