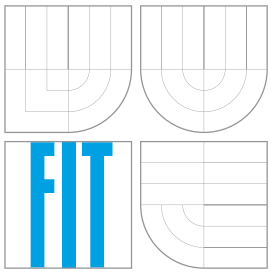
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PLÁNOVÁNÍ A ROZVRHOVÁNÍ
PLANNING AND SCHEDULING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LUKÁŠ HEFKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. VLADIMÍR JANOUŠEK, Ph.D.
SUPERVISOR

BRNO 2009

Plánování a rozvrhování

Zadání diplomové práce

1. Prostudujte problematiku plánování a rozvrhování. Zaměřte se na aplikaci genetických algoritmů v tvorbě rozvrhu přidělování zdrojů aktivitám.
2. Navrhněte vhodný příklad pro demonstraci problematiky plánováni a rozvrhování. Pro reprezentaci plánů použijte vysokoúrovňové Petriho sítě.
3. Realizujte systém pro tvorbu rozvrhu s využitím genetických algoritmů. Využijte existující nástroje pro modelování Petriho sítěmi a pro genetickou optimalizaci.
4. Proveďte vyhodnocení dosažených výsledků.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Diplomová práce se zabývá problematikou optimalizace plánování a rozvrhování. K tomu se využívá genetických algoritmů inspirovaných evolučním vývojem. Součástí práce je seznámení s problémem plánování a rozvrhování, genetickými algoritmy a Petriho sítěmi. Těchto znalostí bylo využito k vytvoření aplikace, která by s využitím genetických algoritmů dovedla řešit plánovací problémy a výsledné plány pak reprezentovala Časovou Petriho sítí. V závěru práce jsou prezentovány dosažené výsledky a příklady oblasti využití.

Abstract

This thesis deals with optimization problems of planning and scheduling. There are using genetic algorithms which are inspired by evolution process. Main work is familiar with the problem of planning and scheduling, genetic algorithm and Petri nets. This knowledge was used to create applications that would with the use of genetic algorithms was able to solve planning problems and the resulting plans would be represented the Time Petri Net. In conclusion of the this thesis are presented obtained results and examples of field use.

Klíčová slova

plánování, rozvrhování, Job Shop, genetické algoritmy, simulace plánů, Petriho sítě, Časové Petriho sítě, PNML

Keywords

planning, scheduling, Job Shop, genetic algorithms, simulations plan, Petri Nets, Time Petri Net, PNML

Citace

Lukáš Hefka: Plánování a rozvrhování, diplomová práce, Brno, FIT VUT v Brně, 2009

Plánování a rozvrhování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Vladimíra Janouška, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Lukáš Hefka

25. května 2009

Poděkování

Chtěl bych tímto poděkovat vedoucímu práce Ing. Vladimíru Janouškovi, Ph.D., za jeho cenné připomínky a trpělivost, se kterou práci četl.

Obsah

Kapitola 1

Úvod

Tato práce si klade za cíl seznámení s problematikou *plánování* a *rozvrhování*, *genetickými algoritmy* a teorií *Časových Petriho sítí*. Zaměřuje se na aplikaci genetických algoritmů při tvorbě rozvrhů a přidělování zdrojů aktivitám. Součástí této práce je návrh a implementace aplikace pro řešení plánovacích problémů za pomoci genetických algoritmů. Aplikace bude pro zadaný problém hledat co nejoptimálnější plán. Ten bude následně reprezentován v podobě Časové Petriho sítě a popsán jazykem `PNML`.

V kapitole 2 jsou uvedeny základní pojmy z oblasti plánování a rozvrhování. Jsou zde také popsána kritéria pro hledání plánů a obecné charakteristiky úloh a zdrojů. Kapitola také obsahuje možné typy rozvrhovacích úloh a možnosti jejich výsledné reprezentace. V závěru kapitoly je popsán nejrozšířenější rozvrhovací problém Job Shop. Kapitola 3 uvádí do problematiky genetických algoritmů. Definuje základní pojmy a vysvětluje samotný princip GA. Uvádí metody selekce, typy křížení a typy mutací. Kapitola také představuje neznámější typy genetických algoritmů. Poslední teoretickou kapitolou je kapitola 4 jejímž cílem je seznámení se základními koncepty Petriho sítí. Jsou zde popsány formální definice potřebné pro úvod do studia Petriho sítí. Kapitola se také věnuje rozšířené třídě klasických Petriho sítí - Časové Petriho sítě. Závěr kapitoly uvádí principy modelování sdílených zdrojů.

Následující kapitoly popisují návrh a implementaci dané aplikace. Postup při tvorbě návrhu je zachycen v kapitole 5. Je zde popsána architektura aplikace a návrh struktury pro reprezentaci problému, který bude aplikací řešen. Je také navržen objektový model genetického algoritmu a sepsány požadavky, které musí splňovat. Implementaci tohoto genetického algoritmu dokumentuje kapitola 6. Popisuje kódování problému, implementovaný operátor křížení a mutace. Protože genetický algoritmus potřebuje při ohodnocování jedinců stochasticky simulovat plány, byl zde implementován simulátor plánů. Jeho algoritmus je popsán v kapitole 7. Výsledné plány a jejich transformaci do Časových Petriho sítí popisuji v kapitole 8. Představuji zde také prvky standardizovaného jazyka PNML, do kterého jsou plány převáděny. V závěru této kapitoly je navržen obecný layout pro rozložení prvků Petriho sítě tak, aby byla pro nejrůznější plány zobrazena dle určitých pravidel a aby tak

byla dobře čitelná.

Testování vytvořené aplikace a shrnutí dosažených výsledků je v kapitole 9. Protože popisované plánovací problémy byly spíše obecnějšího charakteru, v kapitole 10 popisuji praktické využití aplikace na konkrétních oblastech využití. Shrnutí celé práce a závěry autora se věnuje poslední kapitola 11.

Kapitola 2

Plánování a rozvrhování

Plánování – postup vytváření plánů. Plánem rozumíme posloupnost akcí, které je potřeba seřadit tak, aby systém, řídící se tímto plánem, dostal se z nějakého svého počátečního stavu do stavu konečného.

Rozvrhování – postup vytváření časových rozvrhů. Rozvrhem rozumíme posloupnost akcí, které musíme seřadit tak, aby se systém podléhající plánování dostal z nějakého svého počátečního stavu do stavu konečného s ohledem na uspořádání na časové ose.

Obrázek 2.1: Proces plánování a rozvrhování

Mezi typické plánovací problémy patří například plánování výroby a výrobních postupů v továrnách, tvorba školních rozvrhů, plánování rozvrhů pracovních směn nebo plánování náročných výpočtů. Tato práce se nezaměřuje na žádné zde uvedené konkrétní plánování. Věnuje se obecnému plánování, které lze aplikovat na více typů plánovacích problémů.

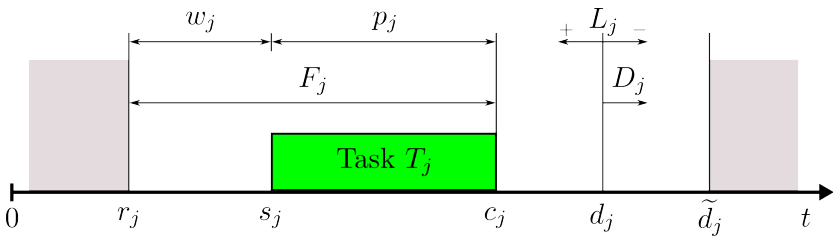
2.1 Základní pojmy

Tato část je věnována seznámení se základními stavebními kameny plánování a rozvrhování. Podrobněji se o těchto pojmech lze dočíst v [1] [2] [3].

2.1.1 Charakteristiky úloh

Úloha je objekt jenž plánujeme. Úloha je charakterizována vnitřní strukturou a svými vlastnostmi. Operace nebo také podúloha je dílčí částí celé úlohy. Úloha se skládá z jedné nebo více operací, které mohou být prováděny na jednom nebo více zdrojích.

Všechny úlohy mají vlastní charakteristiku, která je většinou dána statickými a dynamickými parametry, které vstupují do procesu rozvrhování. Statické parametry jsou takové parametry úlohy, jež známe na počátku rozvrhovacího procesu. Dynamické parametry jsou známy až v průběhu rozvrhovacího procesu, někdy dokonce až na samém konci. Úloha zahrnuje také vlastní strukturu, která ovlivňuje, jak se daná úloha na zdrojích zpracuje.



Obrázek 2.2: Parametry úlohy

Možné parametry úloh:

- *Doba trvání* (processing time) p_j
- *Začátek vykonávání* (start time) s_j [1].
- *Konec vykonávání* (completition time) c_j
- *Okamžik disponability* (release time) r_j: Nejmenší čas, kdy je úloha připravena k provádění.
- *Okamžik požadovaného dokončení* (due date) d_j: Hodnota c_j by měla být menší než tato hodnota.
- *Poslední okamžik dokončení* (deadline) $\widetilde{d}_j$: Hodnota c_j musí být menší než tento čas. Pokud se c_j nevejde do daného limitu, pak není řešení dosažitelné
- *Precedenční vazby na ostatní úlohy* (precedence constraints): Často dochází k situacím, kdy máme danou posloupnost úloh, které se musí provádět v předepsaném pořadí.
- *Stroj* (dedicated processor): Jeden nebo více strojů, na který musí běžet úloha.
- *Priorita* (priority): Důležitost úlohy vzhledem k úlohám jiným.
- *Latence* (latency) L_j: $L_j = c_j - d_j$
- *Doba čekání* (waiting time) w_j: $w_j = s_j - r_j$

[1] U preemptivního rozvrhování může být začátků vykonávání více. Může nastat situace, kdy se provádění úlohy přeruší jinou úlohou a okamžik pokračování můžeme považovat za okamžik začátku vykonávání

- *Překročení vymezeného času* (tardiness) D_j: $D_j = max\{c_j - d_j, 0\}$
- *Doba dokončení* (flow time, response time) F_j: $F_j = c_j - r_j$

Význam uvedených parametrů také ilustruje obrázek 2.2. Tyto parametry nevytváří kompletní výčet. V praxi se můžeme setkat pouze se zlomkem výše uvedených parametrů.

2.1.2 Charakteristiky zdrojů

Zdroj je jednotka, na které může být vykonávána úloha. Zdroje rozlišujeme dle vykonávání úloh. Prvním typem jsou *unární zdroje* (unary resources), které mohou vykonávat v čase pouze jednu úlohu. Druhým typem jsou zdroje s *jistou kapacitou prováděných úloh* (limited capacity resources). U zdrojů, které mají kapacitu větší než 1, se můžeme setkat s tzv. „dávkovým“ zpracováním úloh (batch precessing). Zde všechny úlohy začínají ve stejný okamžik a dokud se nepřipraví celá dávka úloh, tak se čeká na další úlohy do dávky. Pod pojmem zdroje nemusí být myšlen pouze stroj z výrobní linky nebo počítač. Zdroje mohou být i následující:

- Energie
- Peníze
- Lidská práce
- Nástroje
- Roboti a stroje

2.1.3 Definice rozvrhování

Hlavním úkolem rozvrhování je alokace množiny úloh $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ na množinu dostupných zdrojů $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ za dodržení všech omezujících podmínek [10].

Při rozvrhování v praxi se můžeme setkat s rozvrhováním *statickým*, kde množina úloh, která se bude provádět, je známá předem. Tento typ se využívá převážně ve výrobních procesech (továrny, dílny). Dalším typem je rozvrhování dynamické. Zde množina úloh předem známá není, ale vyvíjí se v čase. Tento typ rozvrhování je používán např. v robotice.

Výsledné řešení rozvrhování se většinou posuzuje vhodnou ohodnocovací funkcí. Těmi mohou být například funkce $f(c_1, c_2, ..., c_n)$, které vyjadřují:

- *Nejpozději dokončenou úlohu* (makespan):

$$c_{\max} = \max_{j=1}^{n}\{c_j\} \tag{2.1}$$

- *Celkový součet všech dokončených úloh* (total flow time):

$$\sum_{j=1}^{n} c_j \tag{2.2}$$

- *Vážený celkový součet všech dokončených úloh* (weighted total flow time):

$$\sum_{j=1}^{n} w_j c_j \tag{2.3}$$

- *Maximální latenci*:

$$L_{\max} = \max_{j=1}^{n}\{L_j\} \tag{2.4}$$

- *Součet překročení vymezeného času* (tardiness):

$$\sum_{j=1}^{n} D_j \tag{2.5}$$

- *Vážený součet překročení vymezeného času*:

$$\sum_{j=1}^{n} w_j D_j \tag{2.6}$$

- *Vážený součet jednotkových penalizací*:

$$\sum_{j=1}^{n} w_j U_j \tag{2.7}$$

  kde penalizace U_j je definována jako:

$$U_j = \begin{cases} 0 & \text{pro } c_j < d_j \\ 1 & \text{jinak} \end{cases} \tag{2.8}$$

Podle daného problému je potřeba zvolit vhodnou ohodnocovací funkci. Ne u všech problémů se totiž může hodit nejmenší doba provádění. Je proto důležité si před každou implementací promyslet, kterou z funkcí zvolit.[2]

[2]Pro složitější rozvrhovací problémy se může využít i více ohodnocovacích funkcí současně. Omezující kritéria mohou být nejrůznější. Můžeme například požadovat nejkratší dobu provádění a současně nejméně penalizací.

2.2 Typy rozvrhovacích úloh

Rozvrhovací problémy se dělí na několik skupin, podle složitosti systému nebo na základě povahy úlohy. Jednou z těchto skupin je **plánování na jednom zdroji**. To je taková situace, kdy se všechny úlohy zpracovávají právě na jednom zdroji. Rozvrh pak definuje přiřazení těchto úloh na tento zdroj v čase. **Plánování na více zdrojích** přestavuje situaci, kdy existuje více paralelních zdrojů, které mohou jednotlivé úlohy zpracovávat. Rozvrhem u této skupiny je pak přiřazení jednotlivých úloh na dané zdroje v čase. U této skupiny úloh však vzniká několik omezení. Například zda je daný zdroj vhodný pro zpracování úlohy, precedence jednotlivých úloh nebo časovou provázanost. Další skupinou jsou **Multi-operační problémy (shop)**. Ty představují takové plánování úloh, kdy se jedná úloha postupně zpracovává na více zdrojích (skládá se z více operací). U této skupiny existují tyto varianty:

- **Flow shop** - je taková situace, kdy se každá úloha musí provádět na všech zdrojích ve stejném pořadí
- **Flexible Flow Shop** - zobecnění předchozího případu. Je zde rozdělení na fáze, kde každé fázi přísluší paralelní zdroj. Úloha musí projít všemi fázemi ve stejném pořadí.
- **Job shop** - úloha zde musí být prováděna na zdrojích podle předem daného pořadí
- **Open shop** - provádění úlohy nemusí probíhat na všech zdrojích

2.3 Grahamova klasifikace

V praxi se můžeme setkat s velkým množstvím nejrůznějších rozvrhovacích problémů. To nás přivádí k myšlenkám zavedení standardních notací. Grahamova notace [1] je v současné době velmi rozšířená a zaštiťuje velké množství rozvrhovacích problémů. Pokládám proto za vhodné tuto notaci alespoň ve stručnosti zmínit.

Základem klasifikace je trojice:

$$\alpha \mid \beta \mid \gamma$$

První položka $\alpha = \{\alpha_1, \alpha_2\}$ popisuje zdroje. α_1 označuje typ zdrojů (paralelní, dedikované atd.). α_2 udává počet zdrojů.
Druhý prvek β nese informace o vlastnostech problému jako takového. Popisuje omezení aplikovaná na úlohy. Dále například počet úloh, preempci, přídavné zdroje, relaci následnosti, dobu připravenosti, deadline, omezení počtu úloh).
Poslední prvek γ udává optimalizační kritéria. Konkrétně tyto: $C_{\max}$ - celková doba vykonávání, $\overline{F}$ - střední průběžný čas, $\overline{F}_w$ - střední vážený průměrný čas a $L_{\max}$ - maximální

opoždění.

Příklady takto klasifikovaných problému můžou být následující:

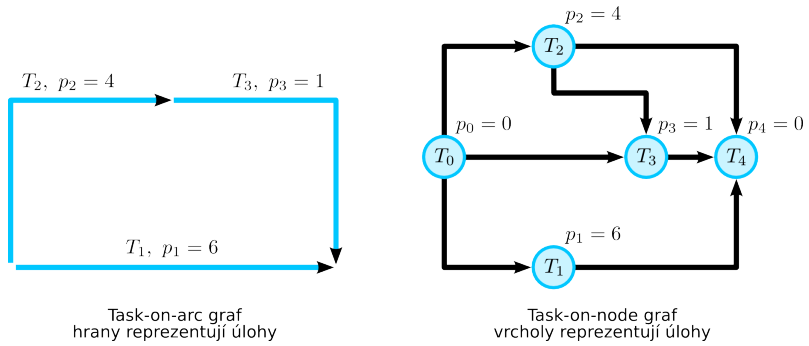
- $P3 \mid prec \mid C_{\max}$: montáž kola
- $P_m \mid r_j \mid \sigma w_j C_j$: paralelní stroje

2.4 Grafická reprezentace problémů

Aby se zadané problémy daly snáze pochopit, je někdy výhodnější je zobrazit graficky pomocí diagramů. Zadání problémů se nejčastěji reprezentují pomocí tzv. *task-on-node* nebo *task-on-arc* grafů. U prvního typu grafu se úlohy zakreslují na vrcholech a u druhého typu na hranách grafu. Pro ilustraci uvažme příklad zadán následovně:

$$\mathcal{T} = \{T_1, T_2, T_3\},\ \mathcal{P} = \{6, 4, 1\},\ T_2 \rightarrow T_3$$

Výsledné řešení problému vystihuje pro oba dva typy grafů obrázek 2.3. U grafů typu *task-on-node* se přidávají navíc dva uzly, které reprezentují nové úlohy. Ty mají funkci počáteční resp. koncové úlohy. Využívají se k vymezení časového úseku od počátku skutečné úlohy do konce poslední skutečné úlohy.

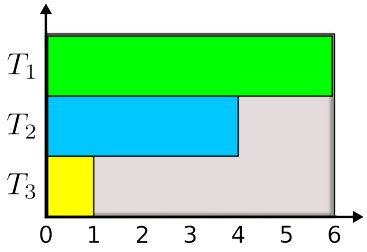


Obrázek 2.3: *Task-on-arc* a *task-on-node* grafy

U grafické reprezentace výsledného rozvrhu se velmi často používají *Ganttovy diagramy*. Velmi využívané jsou také v oblasti manažerské, kde slouží k projektovému plánování. Jedná se o sloupcový graf, který vyjadřuje úplnou informaci o jednotlivých úlohách (počátek, konec, precedence, zdroj). Příklad řešení našeho problému je ilustrován Ganttovým diagramem na obrázku 2.4.

2.5 Job shop scheduling

Job shop scheduling [5] v českém překladu rozvrhování zakázkové výroby je typ rozvrhování, kde daný problém může být popsán následovně. Existuje množina m strojů a n



Obrázek 2.4: Ganttův diagram

prací. Každá práce má dáno uspořádání strojů, tzn., že práce jsou složeny z jednotlivých úkolů, které jsou určeny požadavkem na stroj a procesním časem na daném stroji.

Mějme dány tři konečné množiny:

- úloh $J = \{J_1, J_2, ..., J_n\}$
- strojů $M = \{M_1, M_2, ..., M_m\}$
- operací $O = \{o_1, o_2, ..., o_N\}$

Každá z úloh je složena z daného, obecně nestejného počtu operací. Množina O je množina všech operací všech úloh. Je zapotřebí provést všechny úlohy při splnění těchto omezení:

- pro každou operaci je potřeba jednoznačně přiřazený stroj
- na jednom stroji je možno provádět pouze jednu operaci jedné úlohy
- pořadí operací úloh nelze měnit
- provádění operace nelze přerušit
- mezi operacemi různých není precedenční omezení

Vykonání jedné úlohy tedy znamená provedení všech jejich operací. Ty musí být provedeny v zadaném pořadí a na daných strojích. Rozvrhem pak nazýváme jakékoli proveditelné pořadí π operací na všech strojích z M. Řešení problému je nalezení optimálního pořadí operací na daných strojích. Výsledné řešení může vycházet z nejrůznějších cílů. Může to být např. minimalizace celkové doby zpracování, minimalizace ztrát souvisejících s nesplněním prací v požadovaných termínech, minimalizace čekání strojů a pod.

Nechť je dáno toto označení:

n	počet úloh
N	celkový počet operací

$B(i)$	množina bezprostředních předchůdců operace i
m	počet strojů
$O(k)$	množina operací přiřazených stroji k
p_j	doba provádění operace j
z_j	nejdříve možný termín zahájení operace j
M	horní mez celkové doby trvání všech operací
$C_{\max}$	celková doba trvání všech úloh
$x_{j,l}$	proměnná pro popis precedenčního vztahu mezi operacemi $j, l \in O^k$
	$x_{j,l} = 1$ operace j se provede před operací l
	$x_{j,l} = 0$ operace l se provede před operací j

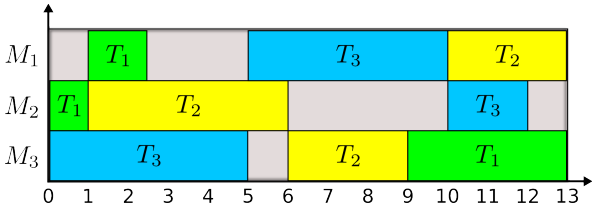
Problém optimalizace rozvrhu, kde se snažíme minimalizovat $C_{\max}$, lze pak matematicky definovat takto:

$$C_{\max} \geq z_i + p_i, \qquad i = 1, 2, ..., n \tag{1}$$
$$z_j \geq z_j + p_i, \qquad j = 1, 2, ..., N;\ i \in B(j) \tag{2}$$
$$z_l \geq z_j + p_j x_{j,l} - M(1 - x_{j,l}), \qquad k = 1, 2, ..., m;\ j, l \in O(k) \tag{3}$$
$$z_j \geq z_l + p_l(1 - x_{j,l}) - M x_{j,l}, \qquad k = 1, 2, ..., m;\ j, l \in O(k) \tag{4}$$
$$x_{j,l} \in \{0, 1\}, \qquad k = 1, 2, ..., m;\ j, l \in O(k) \tag{5}$$
$$C_{\max} \geq 0,\ z_j \geq 0 \qquad j = 1, 2, ..., N \tag{6}$$

Omezení (2) udává, že operace každého úkolu se provádějí v předem určeném pořadí. Podmínky (3)-(5) říkají, že v určitém čase může být stroj obsazen pouze jednou operací. Libovolné přípustné řešení, které splňuje podmínky (1)-(6) je proveditelné a označujeme jej jako rozvrh.

Příklad 2.5.1 *Mějme zadánu instanci problému, která obsahuje úlohy T_1,T_2 a T_3 ((i, j) označme operaci úlohy T_j na stroji i):*

- $p_{(1,1)} = 2$, $p_{(2,1)} = 1$, $p_{(3,1)} = 4$ s pořadím $(2, 1)$,$(1, 1)$,$(3, 1)$
- $p_{(1,2)} = 2$, $p_{(2,2)} = 1$, $p_{(3,2)} = 4$ s pořadím $(3, 2)$,$(1, 2)$,$(2, 2)$
- $p_{(1,3)} = 2$, $p_{(2,3)} = 1$, $p_{(3,3)} = 4$ s pořadím $(2, 3)$,$(3, 3)$,$(1, 3)$



Obrázek 2.5: Optimální řešení Job shop problému dle kritéria makespan (Ganttův diagram)

Optimální řešení tohoto problému je zobrazeno pomocí Ganttova diagramu na obrázku 2.5. Výpočet celkové ceny tohoto řešení je následující:

$$C_{\max} = \max(start_{(3,1)} + p_{(3,1)},\ start_{(2,2)} + p_{(2,2)},\ start_{(1,3)} + p_{(1,3)}) = \max(13, 13, 12) = 13$$

Kapitola 3

Genetické algoritmy

Člověk se v životě často dostane do situace, kdy potřebuje řešit složité problémy. Snaží se pak nalézt řešení, které není ledajaké, ale jistým způsobem co nejvíce optimální, případně nejoptimálnější. Množina problémů u kterých se snažíme nalézt optimum na jisté množině potencionálních řešení se nazývá optimalizační problémy. Množina možných řešení se také někdy označuje jako **stavový prostor**. Řešení daného problému se pak transformuje na procházení tímto stavovým prostorem. Abychom určili kvalitu jednotlivých řešení ve stavovém prostoru, ohodnocujeme je na základě kriteriální ztrátové funkce. Řada problémů obsahuje značný počet potencionálních řešení (NP-úplné problémy), u kterých se nedá v rozumném čase vyhodnotit optimum. Proto se hledají cesty k urychlení tohoto hledání. Jednou z těchto cest jsou Genetické algoritmy.

Na otázku vývoje člověka přinesl v druhé polovině 19. století Charles Darwin ucelenou a obhajitelnou teorii evoluce. Tato teorie odpovídá na otázku vývoje, který stojí na chybách při replikaci a přírodním výběru, který zvýhodňuje vlastnosti jedinců lépe přizpůsobených okolí a penalizuje ty vlastnosti, které jedince omezují. Kvůli těmto chybám vzniká spousta odlišných vlastností při replikace. Tedy jedinci, kteří jsou rychlejší, inteligentnější a silnější mají daleko větší šance v různorodém prostředí než jedinci pomalejší, méně inteligentní a slabší.

Genetické algoritmy [11] [7] [8] se inspirují biologickými disciplínami a používají výrazové prostředky z těchto oborů. Pro označení jednotlivce v populaci využíváme pojmu genotyp, chromozom, struktura nebo řetězec. Gen může nabývat několika hodnot, které v celku mohou vyjadřovat určitou vlastnost jako např. barvu očí. Těmto hodnotám se říká alely. Každý z chromozomů je reprezentantem možného řešení problému. Formát chromozomu a jeho význam je označován jako fenotyp.

Pro prohledávání stavového prostoru existuje více přístupů. Procházení obvykle vyžaduje vyvážení dvou požadavků:

- exploitation - prohledávání ve slibných oblastech prostoru
- expploration - nejúplnější průchod stavovým prostorem

Technikám, které důkladně prohledávají okolí nejlepšího řešení říkáme **Gradientní metody**. Mezi takové metody patří například **Hillclimbing**. Jejich nevýhodou je, že úplně ignorují zbytek stavového prostoru. Opak těchto metod jsou metody náhodné. Ty procházejí prostor ze všech stran, ale ignorují jeho slibné oblasti. Genetické algoritmy kombinují vlastnosti těchto dvou způsobů. Na základě parametrů, které jim můžeme nastavit se zaměřují jak na slibné oblasti, tak na co největší část stavového prostoru. To z těchto algoritmů dělá něco ojedinělého.

GA se vyznačují těmito vlastnostmi:

- lze je aplikovat na velmi široké spektrum úloh
- prochází nejrůznější stavové prostory (spojité, multimodální, nehladké a pod.)
- řešení lze hledat i pro více kritérií
- využití i u dynamických optimalizací
- umí nalézt více optimálních řešení

Aby bylo možné vyřešit daný problém pomocí GA je potřeba splnit tyto požadavky:

- vhodně zvolit reprezentaci problému - zakódování do chromozomu
- navrhnout způsob vytvoření počáteční populace chromozomů
- implementovat fitness funkci, která bude bude ohodnocovat jedince
- definovat genetické operátory
- zvolit vhodné parametry jako velikost populace, pravděp. křížení a pod.

Celý proces genetického algoritmu je ilustrován na obrázku 3.1.

3.1 Stavební bloky a teorie schémat

V populaci lze pozorovat mezi stejně dobrými jedinci určitou podobnost vyjádřenou stejnými bity na určitých pozicích. Ostatní pozice chromozomu nemusí v podobnosti jedinců hrát žádnou roli. Pozicím, které mezi stejně dobrými jedinci udávají podobnost říkáme stavební bloky.

Velká část modelů GA staví na takzvané teorii schémat [4]. Schéma je podobnostní šablona definovaná nad abecedou $\{0, 1, *\}$. Symbol „*“ na pozici ve schématu udává, že na této pozici se může vyskytovat libovolný symbol z množiny $\{0, 1\}$. Počet pevně stanovených pozic ve schématu H se nazývá řádem schématu $o(H)$. Vzdálenost pevných hodnot

Algoritmus 1 Pseudokód genetického algoritmu

```
t := 0;
initpopulation P(t);
evaluate P (t);
while not finished do
  t := t + 1;
  P' := selectpar P (t);
  recombine P' (t);
  mutate P' (t);
  evaluate P' (t);
  P := survive P,P' (t);
end while
```

ve schématu se označuje jako délka schématu $\delta(H)$. Většinou je účelnější sledovat šíření schémat než šíření jednotlivých jedinců.

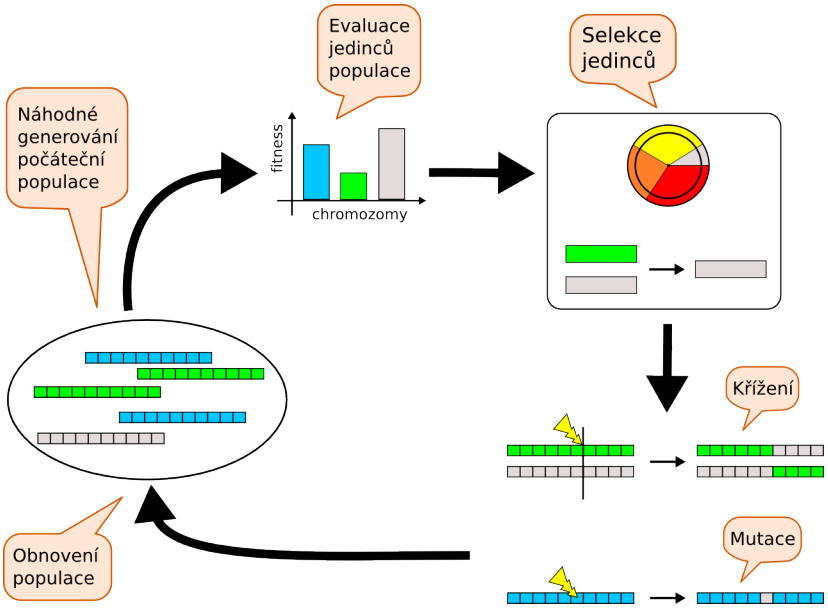
Protože genetické operátory přímo ovlivňují schémata chromozomu, je výhodné zjistit, jak moc schémata ovlivňují. To se dá vyjádřit pomocí tzv. **schéma-teorému**, který je dán vztahem:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\overline{f}} \cdot \left[1 - P_C \cdot \frac{\delta(H)}{L-1} - P_M \cdot o(H)\right] \tag{3.1}$$

kde uvedené symboly mají následující význam:

$m(H,t)$	počet řetězců v populaci odpovídajících schématu H v čase t
$f(H)$	kvalita schématu
$\overline{f}$	střední kvalita populace
P_C	pravděpodobnost křížení
P_M	pravděpodobnost mutace
$\delta(H)$	definující délka schématu
L	délka schématu
$o(H)$	řád schématu

Pokud provedeme rozbor schéma-teorému můžeme vypozorovat, že stavební bloky chromozomů jsou tvořeny krátkými nadprůměrnými schématy nízkého řádu. Schémata jsou krátká z toho důvodu, aby s co největší pravděpodobností nebyla porušena křížením a nízkého řádu, aby s co největší pravděpodobností přežila uplatnění mutace.



Obrázek 3.1: Proces genetického algoritmu

3.2 Selekce

Selekční operátor se inspiruje Darwinovou teorií o přirozeném výběru jedinců. Realizuje model přežití nejsilnějšího jedince. Z populace vybírá ty nejlepší jedince, ze kterých se pak rekombinací tvoří potomci noví. Aby řešení nešlo od samého počátku pouze jedním směrem a nedostali jsme se tak do lokálního mimina je výhodné do nové populace vzít i některé nejhorší jedince. To zaručují pravděpodobnostní mechanizmy při výběru jedinců, kdy se chromozomu přiřadí pravděpodobnost jeho přežití na základě hodnoty z fitness funkce. Můžeme tak vyjádřit *Selekční intenzitu* nebo také *Selekční tlak* tímto vztahem:

$$I = \frac{\overline{m^*} - \overline{M}}{\overline{\sigma}} \tag{3.2}$$

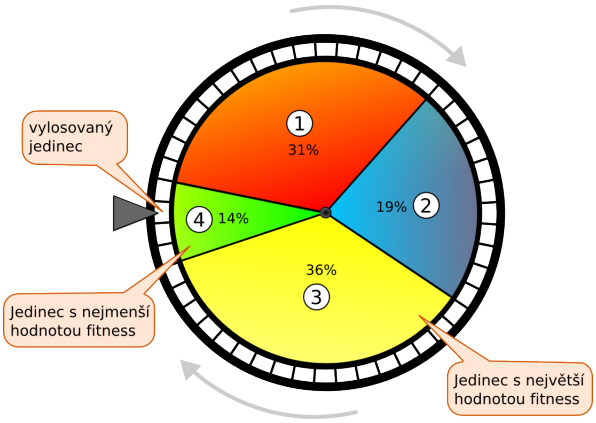
kde je význam symbolů následující:

$\overline{M}$	průměrná fitness hodnota v populaci před selekcí
$\overline{m^*}$	průměrná fitness hodnota po selekci
$\overline{\sigma}$	rozptyl fitness hodnot před selekcí

Rychlost konvergence algoritmu závisí na selekčním tlaku. Neboli čím větší je selekční tlak, tím rychleji algoritmus konverguje.

3.2.1 Ruletová selekce

Ruletová selekce (Roulette-wheel selection) vybírá jedince na základě pravděpodobnosti, která je přímo úměrná jeho kvalitě. Celý proces výběru se dá představit na ruletovém kole, které roztočíme a vybereme toho, na kterém se zastaví kulička. Jedinci jsou na ruletovém kole rozmístěni tak, že čím více je jedinec kvalitnější, tím více zabírá na kole políček. Toto kolo je ilustrováno na obrázku 3.2 I když je toto selekční schéma jedním z nejužívanějších,



Obrázek 3.2: Ruletový výběr jedinců

má i pár nevýhod. Má problém se škálováním, je vhodný pouze pro maximalizační problémy a také vyžaduje velkou populaci.

3.2.2 Pořadová selekce

Pořadová selekce (Rank selection) je modifikací předchozího selekčního schématu. Chromozomy jsou zde seřazeny v posloupnosti dle jejich kvality. Nad tímto uspořádáním následně probíhá ruletová selekce. Tato metoda řeší některé problémy ruletové selekce jako např. problém škálování, vzorkování u menších populací a také zvládá dobře nejen maximalizační, ale i minimalizační problémy.

3.2.3 Turnajová selekce

Turnajová selekce (Tournament selection) je opačným přístupem než předchozí dvě metody. Pro vytvoření nové generace se ze staré generace vezme n chromozomů a mezi nimi se uspořádá turnaj. Do nové generace tak jdou nejlepší jedinci z n-tice. Hodnotou n se dá jednoduše ovlivňovat selekční intenzita. Čím větší je hodnota n, tím větší je selekční tlak.

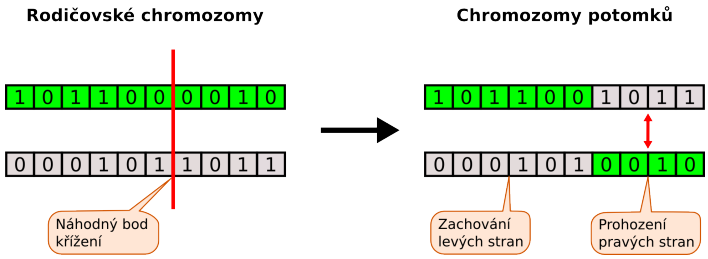
3.3 Křížení

Operátor křížení (crossing-over) je jeden z nejdůležitějších operátorů v evoluci populace. Pro genetické algoritmy má velký přínos při výměně informací mezi jedinci. Přesto však

někteří tvrdí, že tento operátor rozbíjí stavební bloky a tak jej uplatňují se stejně malou pravděpodobností jako mutaci. Existuje velké množství variant křížení, přičemž základem je náhodný výběr dvojic jedinců z populace. Mezi nimi následně dochází k výměně genové informace. Tato rekombinace se ale neprovádí na celou populaci, nýbrž pouze na nějakou část (např. 60%). Zbytek jedinců zůstává bez nějaké výměny informací.

3.3.1 Jednobodové

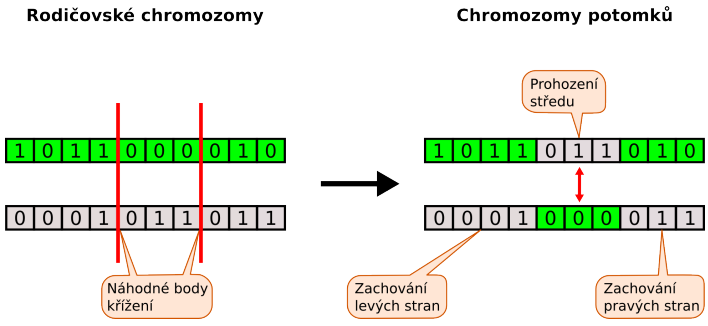
Patří k nejjednodušším a také nejvíce používaným metodám křížení. Ze dvou náhodně zvolených rodičů vznikají dva noví potomci. U rodičů se na vybraném místě zvolí bod. Levé části se potomkům zanechají a pravé části se prohodí. Podrobněji to ilustruje obrázek 3.3. Bod ve kterém se rodiče budou křížit se volí náhodně.



Obrázek 3.3: Jednobodové křížení

3.3.2 Dvoubodové

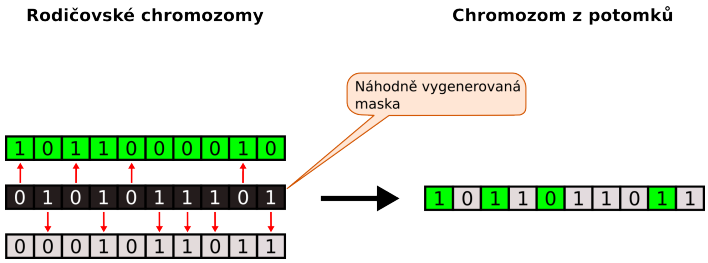
Dvoubodové křížení pracuje na stejném principu jako jednobodové. S tím rozdílem, že se zde náhodně vybírají dva body u rodičů, které rozdělí chromozomy na tři křížící oblasti. Levá a pravá oblast se potomkům zachovává a prostřední oblast se zamění. I zde ze dvou rodičů vznikají dva noví potomci. Ilustrace na obrázku 3.4.



Obrázek 3.4: Dvoubodové křížení

3.3.3 Uniformní

Uniformní křížení nevyužívá bodů na základě kterých prohazuje oblasti, ale vygeneruje náhodně masku o stejné velikosti jako chromozomy. Hodnota masky určuje ze kterého rodiče se použije hodnota pro danou pozici. Pokud je hodnota v masce 0 - použije se hodnota z prvního rodiče. Pro hodnotu 1 se použije hodnota z druhého rodiče. Ilustrativně je to popsáno na obrázku 3.5.



Obrázek 3.5: Uniformní křížení

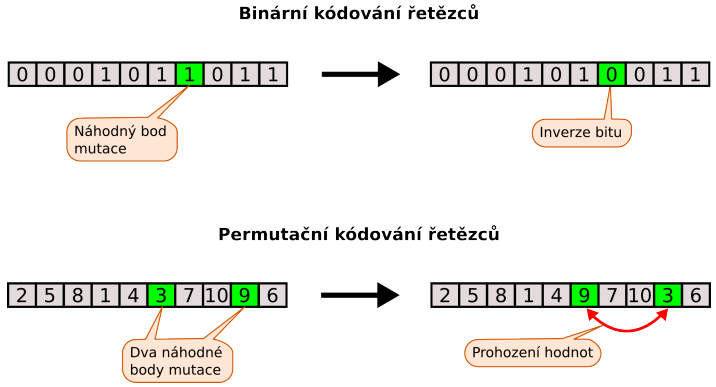
Tento typ křížení je někdy kritizován z důvodu rozvracení kódu (teorie stavebních bloků). Oproti tomu tento princip může do populace přinést různorodost, která je někdy žádoucí. Velmi dobře se také brání předčasné konvergenci algoritmu. Je vhodný pro vícerozměrné funkce s mnoha lokálními extrémy.

3.4 Mutace

Tento reprodukční operátor je i přes malou četnost výskytu velmi významný. Klasicky modifikuje (vytváří mutanty) genů s velmi malou pravděpodobností. Tato pravděpodobnost je obvykle mezi hodnotami 0.0005 a 0.01. Mutace přináší do populace nové informace. Pro příliš velkou pravděpodobnost mutace se může vývoj populace stát nestabilní. Naopak pro malou pravděpodobnost nevznikají nové informace a řešení může směřovat pouze jedním směrem. To může vést do lokálního extrému. Existují implementace, kdy se míra mutace mění dynamicky v závislosti na konvergenci populace.

U binárního kódování se mutace provádí tak, že se náhodně zvolí jedna pozice v chromozomu a její hodnota se invertuje. U permutačního kódování řetězců se náhodně vyberou dvě pozice v chromozomu a ty se jednoduše zamění. Tyto postupy jsou znázorněny na obrázku 3.6.

Řada výzkumů ukázala, že u malých populací je výhodné nastavit větší pravděpodobnost mutace a stejně tak opačně. Tato teorie se dá vysvětlit tak, že u malých populací mutace pomáhá procházet větší stavový prostor. U velkých populací se toto zaručuje samotnou velikostí populace.



Obrázek 3.6: Mutace pro binární a permutační kódování

3.5 Ohodnocovací funkce

Ohodnocovací funkce je jeden z nejdůležitějších elementů, který má pod správou uživatel. Důležitý proto, že řídí celý proces evoluce. Z toho důvodu by se jí měla při implementaci věnovat značná pozornost. Ohodnocovací funkce určuje kvalitu jednotlivých jedinců. Tato kvalita je přímo úměrná schopnosti přežití v okolním prostředí. Rozlišujeme dva typy ohodnocovacích funkcí.

Nejpoužívanějším typem je *Fitness funkce* f. Tato funkce představuje stavový prostor ve kterém se snažíme nalézt maximum. Chromozom představuje jeden bod v tomto prostoru. Hledáme tedy nejlepší chromozom. $f(i)$ udává fitness hodnotu i-tého chromozomu. Pro řešení, která jsou nepřijatelná lze použít penalizační funkci g.

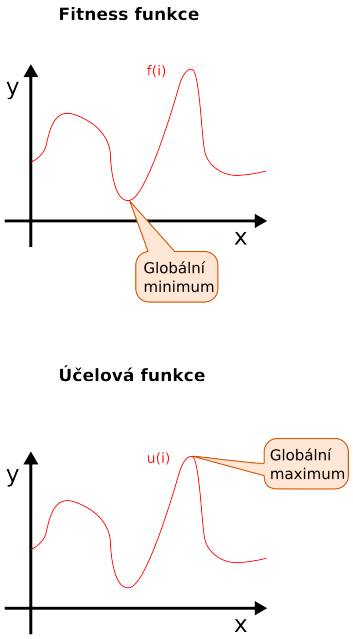
Dalším typem je *Účelová funkce* u. Jedná se o invertovanou fitness funkci. Používá se tehdy pokud ve stavovém prostoru hledáme minimum. Dá se využít pro řešení problémů jako např. Knapsack nebo TSP.

Grafické srovnání těchto dvou ohodnocovacích funkcí je na obrázku 3.7

3.6 Typy GA

Jednoduchý GA (Simple GA) – tento algoritmus využívá nepřekrývající se populace. Do každé generace algoritmus vytvoří úplně novou populaci jedinců. Pro terminální podmínku můžeme využít konvergenci populace. Protože dosažení úplné konvergence je velmi náročné, populace se považuje za zkonvergovanou, pokud rozptyl klesne pod určitou mez. Dá se ovšem použít i jiných terminálních podmínek jako počet generací apod.

GA se stálým stavem (Steady-State GA) – tento typ algoritmu používá překrývající populace. Uživatel tedy musí specifikovat jak velká část jedinců se bude nahrazovat. Dále je zde potřeba specifikovat, jak se budou volit jedinci k nahrazení. Tedy vybrat



Obrázek 3.7: Fitness a účelová funkce

nahrazovací strategii. Lze použít stejných terminálních podmínek jako u předchozího typu GA.

Inkrementální GA – využívá podobného principu jako GA se stálým stavem. Opět používá překrývající se populace. Tento typ GA se ale vyznačuje velmi malým překryvem. Obvykle se v každé generaci vytvoří jeden nebo dva noví jedinci. Je zde potřeba opět vybrat nahrazovací strategii. Nejčastěji jsou v populaci nahrazování nejhorší jedinci.

Kapitola 4

Petriho sítě

Pojem Petriho sítě označuje širokou třídu matematických modelů, kterými lze popisovat řídící toky a také informační závislosti uvnitř systémů. Jedná se tedy o matematickou reprezentaci diskrétních distribuovaných systémů. Petriho sítě poprvé vznikly v roce 1962 v disertační práci německého matematika C.A.Petriho. Petri v práci prezentoval nové koncepty popisu závislostí podmínek a událostí v modelovaném systému. Od té doby se Petriho sítě začaly postupně rozvíjet a dnešní době už existuje několik tříd těchto sítí. V této práci se budu soustředit především na Časové Petriho sítě, které jsou využity pro reprezentaci plánů.

Oblastí, ve kterých se s Petriho sítěmi můžeme setkat, stále přibývá. Nejčastěji je to ale při návrhu, modelování a analýze paralelních a distribuovaných systémů, ale také v telekomunikacích či administrativě.

4.1 Základní koncepty

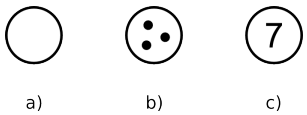
Stavebními kameny Petriho sítí jsou **místa** (places), **přechody** (transitions). Místo představuje podmínku a přechod je událost. Přechod je událost, která má vstupní podmínky a po jejím proběhnutí začnou platit podmínky jiné. Místa a přechody se propojují **hranami** (arcs). Grafickou reprezentací místa je kružnice a přechod je zobrazen obdélníkem, případně úsečkou viz obrázek 4.1.

Obrázek 4.1: a) grafická reprezentace místa, b) grafická reprezentace přechodu ve dvou variantách

Modelovací schopnost Petriho sítí je obecně větší než konečných automatů. U konečných automatů je množina stavů modelovaného systému dána stavy automatu. Petriho sítě

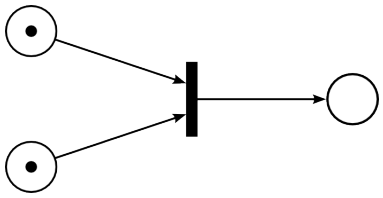
udávají stav značením jednotlivých míst (tzv. značení sítě).

Místa jsou značena nezápornou celočíselnou hodnotou udávající počet značek (tokens) v místě. Tento počet je graficky znázorněn pomocí vyplněných kroužků o stejném počtu jako je počet značek v místě. Při větším množství značek se počet reprezentuje číslem viz. obrázek 4.2.

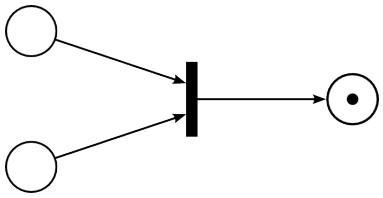


Obrázek 4.2: a) místo bez značek, b) místo se třemi značkami, c) místo se sedmi značkami

Petriho síť tedy vzniká propojením míst a přechodů. Pokud např. místem modelujeme logickou podmínku, pak podmínka platí, pokud je v místě značka. V opačném případě podmínka neplatí. Daný přechod je proveditelný, pokud jsou splněny všechny jeho podmínky. Provedení přechodu znamená, že se odebere značení ze vstupních míst a je přidáno do míst výstupních. Na obrázku 4.3 je síť před provedením přechodu a na obrázku 4.4 po provedení přechodu.



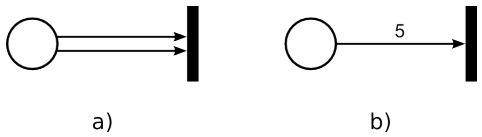
Obrázek 4.3: Síť před provedením přechodu - splněny vstupní podmínky



Obrázek 4.4: Sít po provedení přechodu

Případu, kdy místa mají význam logických dvouhodnotových podmínek využívá nejjednodušší třída Petriho sítí C/E Petriho sítě. (Condition/Event Petri Nets). Zobecněním C/E Petriho sítí vzniká v dnešní době velmi rozšířená třída Petriho sítí. Místa sítě jsou interpretována jako parciální stav systému, který se specifikuje nezáporným celočíselným značením. Provedení události se tedy formuluje na minimální počet značek ve vstupních podmínkách přechodu. Tento minimální počet se reprezentuje vícenásobnou hranou. Pro

větší počet hran se při grafické reprezentaci častěji zobrazuje nad hranou hodnota udávající její váhu viz. obrázek 4.5



Obrázek 4.5: a) násobná hrana, b) ohodnocení hrany

4.2 Formální definice

Tato kapitola obsahuje základní matematické definice potřebné pro úvod do studia Petriho sítí. Více se lze dočíst v [13].

- **Definice 4.2.1** *Síť.*

  Trojici $N = (P, T, F)$ nazýváme sítí, jestliže

  1. P a T jsou disjunktní množiny a
  2. $F \subseteq (P \times T) \cup (T \times P)$ je binární relace

  Množina P se nazývá *množinou míst* sítě N, množina T *množinou přechodů* sítě N a relace F *tokovou relací* sítě N.

  Grafem sítě nazýváme bipartitní orientovaný graf, který vznikne grafovou reprezentací relace F. Množina $P \cup T$ je množinou vrcholů grafu sítě.

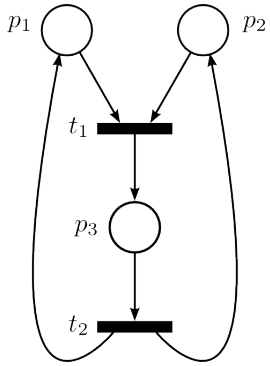
- **Příklad 4.2.1**

  Síť N zobrazená na obrázku 4.6
  má formální zápis $N = (P, T, F)$, kde
  $P = \{p_1, p_2, p_3\}$
  $T = \{t_1, t_2\}$
  $F = \{\langle p_1, t_1\rangle, \langle p_2, t_1\rangle, \langle t_1, p_3\rangle, \langle p_3, t_2\rangle, \langle t_2, p_1\rangle, \langle t_2, p_2\rangle\}$
  .

- **Definice 4.2.2** *Petriho síť.*

  Šestici $N = (P, T, F, W, K, M_0)$ nazýváme P/T Petriho sítí (Place/Transition Petri Net) jestliže

  1. (P, T, F) je konečná síť
  2. $W : F \to \mathbb{N}\backslash 0$ je ohodnocení hran grafu sítě určující *váhu* každé hrany



Obrázek 4.6: Příklad Petriho sítě N

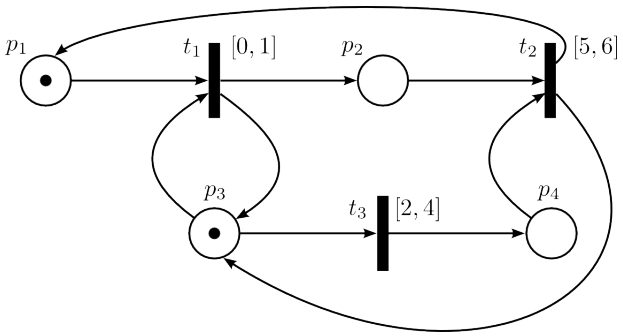
3. $K : P \to \mathbb{N} \cup \{\omega\}$ je zobrazení určující *kapacitu* každého místa
4. $M_0 : P \to \mathbb{N} \cup \{\omega\}$ je *počáteční značení* míst Petriho sítě takové, že $\forall p \in P : M_0(p) \leq K(p)$

Množina P se nazývá *množinou míst* sítě N, množina T *množinou přechodů* sítě N a relace F *tokovou relací* sítě N.

Grafem sítě nazýváme bipartitní orientovaný graf, který vznikne grafovou reprezentací relace F. Množina $P \cup T$ je množinou vrcholů grafu sítě.

4.3 Časové Petriho sítě

Třída Časových Petriho sítí je rozšířením klasických Petriho sítí o možnost popisu časových vztahů mezi operacemi v modelovaném systému [13].



Obrázek 4.7: Ukázka Časové Petriho sítě

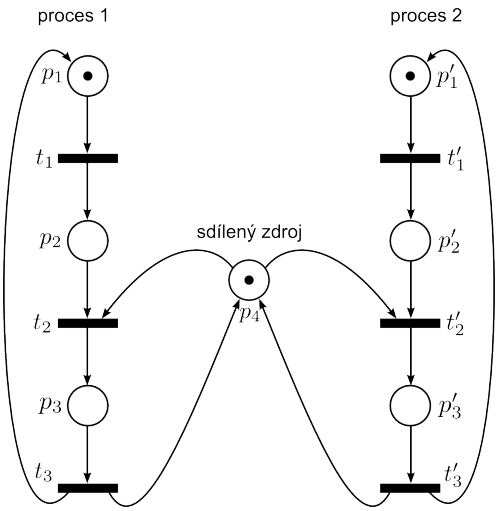
Každý přechod t časové Petriho sítě je atribuován dvěma nezápornými reálnými čísly a a b, $a \leq b$ reprezentujícími relativní časové hodnoty vztažené k okamžiku, kdy se stává přechod t proveditelným.

Hodnota a značí minimální čas, který musí uplynout, aby přechod t mohl být proveden a hodnota b je maximální čas, po který je přechod t proveditelný, aniž byl proveden. Tedy, stane-li se přechod t proveditelným v čase τ, pak může být proveden v časovém intervalu $\langle \tau + a, \tau + b \rangle$, pokud ovšem v tomto intervalu nedošlo ke změně značení, která vylučuje provedení přechodu t. Grafická reprezentace sítě se nijak nemění od klasických Petriho sítí. Jedinou změnou je, že se k přechodům přidává popisná informace udávající minimální a maximální čas, ve kterém může být daný přechod proveden. Tato informace se zapisuje v hranatých závorkách. Ukázka časové sítě je ilustrována na obrázku 4.7.

4.4 Modelování sdílených zdrojů

Petriho sítěmi lze modelovat systémy se sdílenými zdroji. Princip modelování sdílených zdrojů bude nejlépe patrný na příkladu 4.4.1.

- **Příklad 4.4.1**



Obrázek 4.8: Příklad Petriho sítě se sdíleným zdrojem

Příklad ilustruje použití sdíleného zdroje dvěma procesy. Každý z procesů je složen ze tří událostí a tří přechodů. Sdílený zdroj obsahuje jednu značku, což znamená, že zdroj může v jednu chvíli využívat pouze jeden proces. Daný proces si značku ze sdíleného zdroje odebere a po provedení událostí ji následně vrátí. Pokud jsou přechody t_3 a t'_3 časové, tak představují onen fakt používání zdroje. Velkým problémem u sdílených zdrojů je tzv. *uváznutí*, někdy také označované jako *deadlock*, což znamená, že žádný z procesů nemůže pokračovat a žádný přechod sítě není v tomto okamžiku proveditelný.

Kapitola 5

Návrh programu

Hlavním cílem, který jsem si kladl při návrhu, bylo vytvoření programu pro řešení co nejobecnější množiny problémů plánování a rozvrhování. Protože úplně všechny problémy z této oblasti není možné jednou aplikací pokrýt, soustředil jsem se proto na skupinu problémů, která je v této oblasti nejrozšířenější a tou jsou Shop problémy. Vycházel jsem ze základního typu Job Shop (viz. kapitola *Job Shop scheduling*, 2.5), který jsem rozšířil do obecnější podoby. Z toho důvodu jsem aplikaci dal jméno *Job Shop Solver*. Dalším důležitým požadavkem na navrhovaný program byla snadná rozšiřitelnost na větší skupinu plánovacích problémů, případně snadná úprava kritérií. Protože je program navrhován jako konzolová aplikace, může být žádoucí jej využívat v aplikacích s GUI [1]. Snažil jsem se tedy program navrhnout tak, aby bylo napojení na GUI co nejjednodušší. Tím by mohlo být využití aplikace širší a mohla by být využita v praxi u nejrůznějších plánovacích problémů.

Úkolem programu bude najít nejlepší plány k zadanému problému. Pro specifikaci zadání jsem navrhl strukturu (viz. kapitola *Reprezentace problému*, 5.2), která bude daný problém reprezentovat. Aplikace bude pro hledání nejlepších plánů využívat genetických algoritmů (viz. kapitola *Genetické algoritmy*, 3). Hledání bude možno provádět pro nejrůznější kritéria, jakými jsou např. nejkratší celková doba plánu nebo nejmenší časová prodleva na zdrojích. Výsledný nalezený plán úlohy bude v poslední fázi reprezentován Časovou Petriho sítí. Ta bude popsána a přehledně vymodelována v jazyce PNML (viz. kapitola *Transformace plánů do Petriho sítě*, 8), který je v současné době standardem pro popis Petriho sítí.

5.1 Architektura aplikace

Objektový model aplikace je koncipován do několika oddělených modulů. Tyto moduly představují hlavní bloky, které jsou díky interfaceovému návrhu snadno nahraditelné za bloky s jinou implementací.

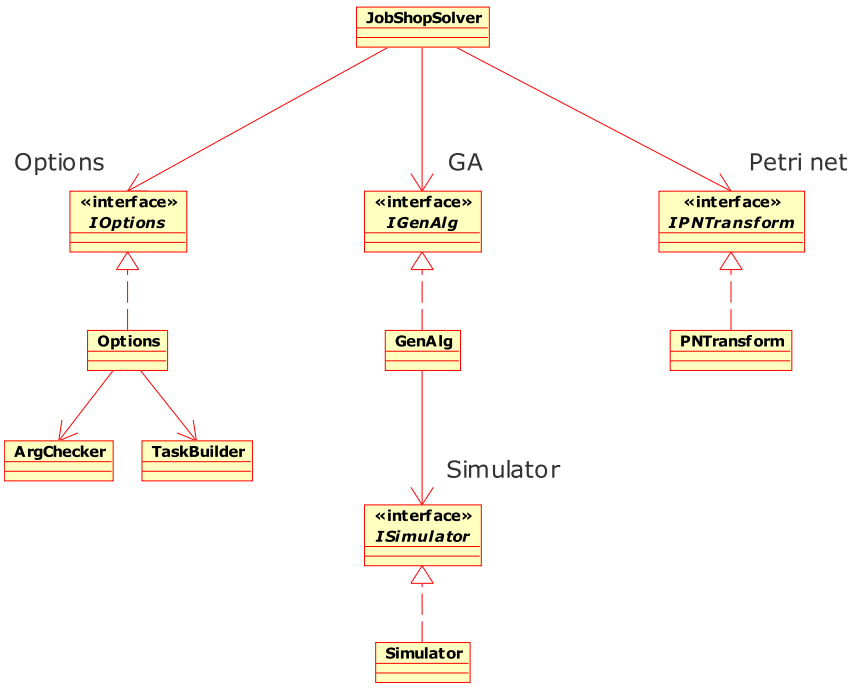
Hlavní objektový model aplikace je zobrazen na obrázku 5.1. Představuje pouze základní

[1] GUI - Graphical User Interface, je uživatelské rozhraní, které umožňuje ovládat počítač pomocí interaktivních grafických prvků.

strukturu s bázovými třídami aplikace. Jednotlivé moduly jsou popisovány v kapitolách níže. Model je tedy složen z těchto modulů:

- `Options`
- `GA`
- `Simulator`
- `Petri net`

Aplikace musí v prvním kroku získat zadanou úlohu od uživatele. O to se bude starat modul `Options`, který zpracuje parametry na standardním vstupu a následně načte zadání úlohy, které je uloženo v textovém souboru v podobě určité struktury (viz. kapitola *Reprezentace problému*, 5.2). Modul se bude starat nejen o kontrolu správné syntaxe této struktury, ale také její sémantiky. V poslední fázi se vytvoří reprezentace zadané úlohy v podobě objektu, který bude předáván do dalších modulů.



Obrázek 5.1: Hlavní objektový model rozdělený na bloky

Hlavní činnost programu bude probíhat v modulu `GA`. Ten bude zahrnovat genetický algoritmus, který se bude snažit pro zadanou úlohu hledat pomocí evoluce nejlepší plán. Základem genetických algoritmů je ohodnocování jednotlivých jedinců, v tomto případě nalezených plánů. Protože plány mohou obsahovat operace, které nemají přesně stanovenou délku provádění, ale mohou ji mít v podobě intervalu, mají plány charakter stochastického

modelu. Bude proto zapotřebí provádět na jednotlivými plány simulaci, ve které se budou náhodně volit délky operace v daném intervalu. O simulaci plánu se bude starat modul `Simulator`.

V poslední fázi se bude nejlepší nalezený plán pro zadanou úlohu reprezentovat v podobě Časové Petriho sítě. Petriho síť bude modelována ve značkovacím jazyce PNML. Bude zapotřebí vytvořit obecný layout, pomocí kterého se pro jakýkoli plán zobrazí Petriho síť v přehledné a čitelné podobě. Tato transformace plánu do podoby Petriho sítě bude prováděna modulem `Petri net`.

5.2 Reprezentace problému

Pro reprezentaci problému je zapotřebí vytvořit obecnou strukturu, ve které by bylo možné danou úlohu popsat. Navrhl jsem proto strukturu pokrývající a současně rozšiřující plánovací úlohu typu Job Shop. Rozšíření Job Shop úlohy spočívá v časovém intervalu jednotlivých operací. Klasická úloha Job Shop počítá pouze s konstantní délkou operace. Toto rozšíření umožňuje popisovat problémy s určitou nepřesností, která může být v určitých případech žádoucí. Na druhou stranu komplikuje řešení takových problémů [2]. Ukázku takového zadání problému je možno vidět na příkladu níže.

Příklad 5.2.1 *Ukázka struktury zadání na jednoduchém problému*

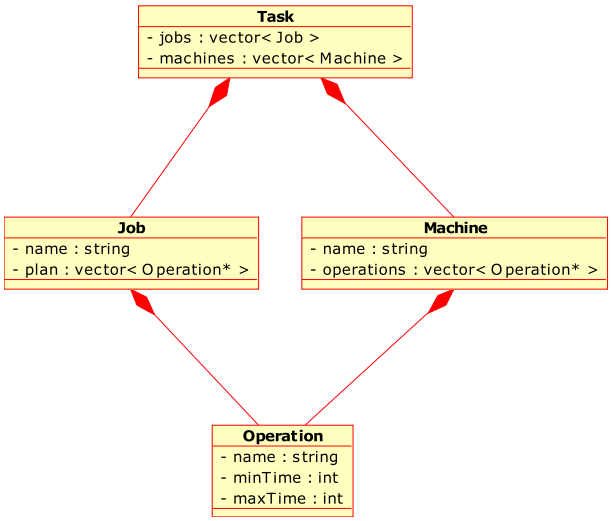
Job: A, B
Operations: $1 - 5$
Machines: X, Y
Process plans:
$A \rightarrow (2, 1)$
$B \rightarrow (3, 4, 5)$
Utilization:
$X \rightarrow \{1, 3, 5\}$
$Y \rightarrow \{2, 3, 4\}$
Duration:
$1: \ 3 - 4$
$2: \ 1 - 1$
$3: \ 5 - 6$
$4: \ 2 - 2$
$5: \ 7 - 9$

Struktura je složena ze šesti prvků, které umožňují přesný popis dané úlohy. Tento popis částečně odpovídá obecnému zadáním Job Shop problému a využívá tak jeho pojmů.

[2] Z důvodu rozšíření o časový interval operací je nutno zavést do řešení úloh simulaci.

Problém se skládá z několika úloh uvedených v parametru `Job`, ty zase z operací, které je potřeba na strojích provést, aby byla úloha splněna. Celkový počet operací na všech úlohách vyjadřuje parametr `Operations`[3]. Struktura dále obsahuje seznam strojů (někdy označovaných jako zdrojů), na kterých jsou vykonávány operace jednotlivých úloh. Výčet strojů je uveden v parametru `Machines`. Pro každou z úloh se definuje tzv. procesní plán, který udává z jakých operací jsou úlohy složeny a v jakém pořadí musí být prováděny. Plány jsou definovány v parametru `Process plans`. Dalším parametrem je `Utilization`, který definuje schopnosti jednotlivých strojů, neboli které operace na nich lze provádět. Posledním parametrem je `Duration` vymezující časový interval pro jednotlivé operace.

Pro tuto strukturu v textovém formátu jsem vytvořil objektový model, který ji bude reprezentovat v paměti počítače. Model návrhu je na obrázku 5.2.



Obrázek 5.2: Objektová reprezentace problému

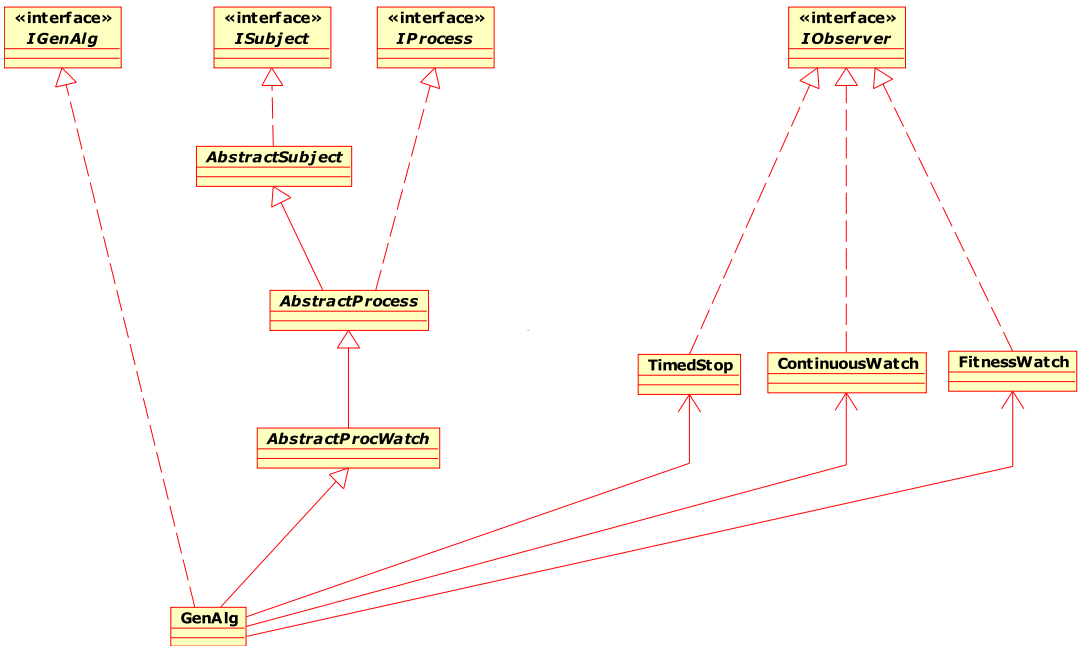
Zadání celého problému by tak bylo uloženo v jediném objektu třídy `Task`, přes který by se přistupovalo ke všem dalším položkám. Tato třída obsahuje dva vektory[4], jeden pro úlohy a druhý pro stroje. Úlohy a stroje jsou navrženy jako samostatné třídy. Třída `Job`, která by představovala úlohu, by uchovávala její název a její procesní plán v podobě vektoru obsahující ukazatele na operace. Stroje by byly objekty třídy `Machine`. Ta by uchovávala název stroje a vektor ukazatelů na operace, které by bylo možno na stroji provádět. Operace jsou taktéž zapouzdřeny ve zvlášť třídě `Operation` a jejími členy jsou název a minimální a maximální doba provádění.

[3]Parametr `Operations` byl zaveden především pro kontrolu sémantiky zadání.
[4]Pojmem *vektor* je zde myšleno vektor z knihovny STL.

5.3 Model Genetického algoritmu

Modul s genetickým algoritmem bude tvořit nejvýznamnější část programu a tak bylo žádoucí mu věnovat větší pozornost. Pro řešení úloh genetickým algoritmem budu využívat knihovny `GAlib` což je C++ knihovna obsahující komponenty pro vytváření genetických algoritmů. Obsahuje nástroje pro použití genetických algoritmů k optimalizacím v různých C++ programech. Ty mohou využívat libovolnou reprezentaci a genetické operátory. Knihovna `GAlib` je podporována na různých UNIX platformách (Linux, SGI, MacOSX, Sun, HP, DEC, IBM) stejně dobře jako na Windows, případně MacOS. Domovské stránky této knihovny jsou na adrese `http://lancet.mit.edu/ga/`.

Při návrhu jsem se soustředil na abstrakci v modelu, která by umožňovala snadnou záměnu knihovny `GAlib` za jinou. Návrh se také odvíjel od mého požadavku umět genetický algoritmus pozastavit, spustit případně restartovat. Tato funkcionalita by mohla být následně využívána při napojení na GUI. Posledním požadavkem bylo využít návrhového vzoru `Observer` [6], který by umožňoval vytvářet prvky pro sledování genetického algoritmu. Objektový model se znázorněnou hierarchií dědičnosti je vidět na obrázku 5.3.



Obrázek 5.3: Objektový model Genetického algoritmu

Bázovou třídou genetického algoritmu je třída `GenAlg`. Ta implementuje rozhraní[5], které je současně také rozhraním modulu `GA`. Třída `GenAlg` je navržena tak, že ji lze použít jako bázovou třídu pro řešení nejrůznějších optimalizačních problémů. Důležitými prvky jsou observery. Právě pro ně jsem přizpůsobil celý návrh objektového modelu. Observery

[5]Rozhraní neboli *interface* je v jazyce C++ definováno jako čistě abstraktní třída (*pure virtual*).

jsou objekty, které mohou sledovat aktuální stav průběhu genetického algoritmu a také reagovat na konkrétní změny tohoto stavu. Například mohou vypisovat informace o průběhu prohledávání a nebo na základě stanovených podmínek mohou prohledávání samy zastavit.

Důležitou třídou, ze které třída `GenAlg` dědí, je třída `AbstractProcess`. Tato třída implementuje rozhraní `IProcess`, které umožňuje už výše zmiňovanou funkcionalitu zastavení, spuštění nebo restart genetického algoritmu. Třída implementuje také pro observery důležité rozhraní `ISubject`. Toto rozhraní slouží k připojování jednotlivých observerů ke genetickému algoritmu.

Návrh observerů byl navržen tak, aby byly pokud možno nezávislé na bázové třídě genetického algoritmu. Observery, které jsem navrhl se budou starat o průběžné vypisování informací na výstup (kterým může být logovací soubor případně standardní výstup) a nebo o případné ukončení prohledávání při platnosti uživatelem nastavených podmínek. Celkem jsem navrhl tři observery stručně popsané v tabulce 5.1.

Třída	**Popis**
`TimedStop`	Zastaví prohledávání pokud uběhne daný časový limit.
`ContinuousWatch`	Průběžně vypisuje informace o aktuálním stavu prohledávání.
`FitnessWatch`	Pokud je nalezeno řešení s lepší hodnou fitness, vypíše jej.

Tabulka 5.1: Seznam observerů

Kapitola 6

Implementace Genetického algoritmu

Jak už jsem zmínil v návrhu, genetický algoritmus bude mít za úkol pro uživatelem zadanou úlohu, najít co nejlepší plán. Protože problém hledání takového plánu je označován jako NP-úplný, nebude jeho nalezení úplně snadné. Genetické algoritmy se ve srovnání se slepým prohledáváním dokáží i s těmito problémy poměrně dobře vyrovnat.

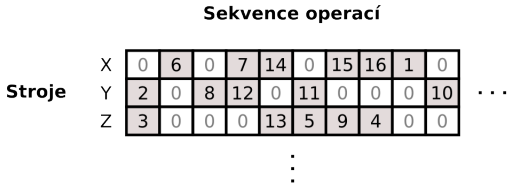
Genetický algoritmus ale pro svou činnost musí mít definovány určité prvky, které jsou pro nejrůznější optimalizační úlohy jedinečné. Základem je definice kódování problému. Neboli jak bude problém reprezentován v podobě genomu. Dalšími důležitými prvky jsou genetické operátory a v neposlední řadě ohodnocovací funkce (*fitness funkce*), která velkou mírou ovlivňuje hledání. Právě implementací těchto prvků genetického algoritmu pro problém plánování se věnuje celá tato kapitola.

6.1 Kódování problému

Vhodné zakódování informací do chromozomu může významně přispět k efektivnosti hledání požadovaného řešení. Špatným kódováním můžeme výrazně navyšovat prohledávaný prostor. Proto je vhodné této části věnovat zvýšenou pozornost. Způsobů jakými lze problémy kódovat může být vícero (celočíselně, binárně, v podobě stromu, řetězce). Volba kódování pak souvisí s charakterem daného problému, jaké informace jsou pro nás klíčové, které genetické operátory se nad jedinci budou aplikovat a pod.

Pro zakódování problému jsem zvolil dvourozměrné pole s celými čísly. To bude reprezentovat chromozom nějakého řešení. Řádky pole budou budou konkrétní stroje a jednotlivé sloupce pak budou operace, které se na nich budou provádět. Hodnoty, které se v polích mohou vyskytovat musí být v intervalu $0 - N$. Nula představuje prázdné místo plánu a hodnota od 1 do N pak konkrétní číslo operace. Pořadí operací v poli udává uspořádání v jakém budou na strojích prováděny. Toto pořadí je závislé i na operacích na jiných strojích. Ukázka zakódování je vidět na obrázku 6.1. Můžeme zde vidět, že například operace 8 je

závislá na operaci 6 a nemůže být proto provedena dříve. Dále můžeme vyčíst, že operacím 14 a 13 nebrání nic, aby byly na strojích provedeny paralelně. Je důležité upozornit na fakt, že pořadí neudává faktické provádění na strojích, ale pouze naznačuje časové pořadí začátku operace. Chromozom tedy není přímým obrazem výsledného prováděcího plánu. Ten je z chromozomu vytvořen dle časového intervalu operací až při samotné simulaci (více v kapitole *Simulátor plánů*, 7).



Obrázek 6.1: Ukázka kódování problému

Protože je problematické pracovat s různě velkými chromozomy, bylo zapotřebí vymezit pevnou velikost pole. Výšku pole udává počet strojů a šířka je dána počtem všech operací. To z toho důvodu, aby byl prostor i pro takové plány, kde by se na strojích neprováděly paralelně žádné operace.

Při inicializaci populace se vytvoří chromozomy dané velikosti a ty jsou naplněny operacemi[1] tak, aby operace ležely na řádcích stroje, který je umí zpracovávat. Rozmístění operací na stroji je náhodné a po celé šířce pole. Prázdná místa jsou vyplněna nulami. Pokud je operace proveditelná na více než jednom stroji, je stroj vybrán náhodně.

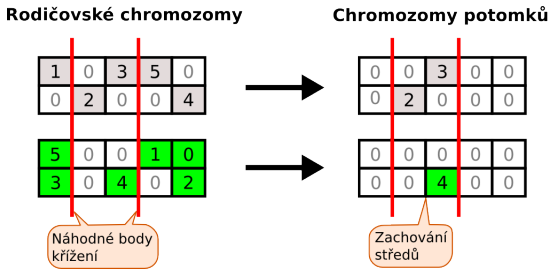
6.2 Operátor křížení

Operátor pro křížení jedinců tvořil implementačně nejsložitější část genetického algoritmu. Snažil jsem se o zachovávání schémat a také, aby výsledné chromozomy byly korektní (aby např. žádná operace nebyla na více strojích a pod.). Navrhl jsem tedy dvoubodové křížení pro celá čísla, které se nejvíce podobá křížící metodě `Order 1 crossover (OX)`. Tuto metodu bylo zapotřebí upravit tak, aby byla použitelná pro řešení tohoto problému. Při křížení se uvažuje fakt, že potomci mají inicializované geny chromozomů na hodnotu 0. Křížení se skládá ze tří částí, které se budu snažit ilustrovat obrázky.

V první části křížení se náhodně vygenerují dva body v rozsahu šířky chromozomu. Na základě těchto dvou bodů se budou do chromozomu potomka brát geny buď z 1. nebo 2. rodiče. Pro jednoduchost zavedu pojmy jako vnitřní a vnější oblast. Do vnitřní oblasti patří geny mezi náhodně vygenerovanými body a do vnější ty ostatní. V této první části křížení se do potomků přenesou pouze vnitřní oblasti a to tak, že pro potomka 1 se použije vnitřní oblast rodiče 1 a pro potomka 2 se použije vnitřní oblast rodiče 2, tak jako to znázorňuje

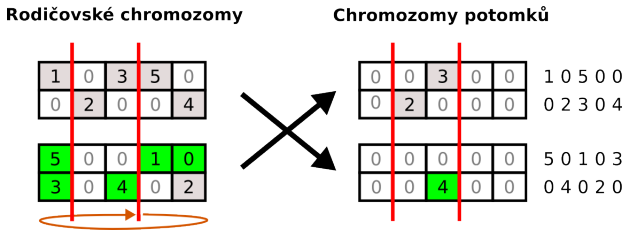
[1] Pojmem *operace* je zde myšlen její identifikátor.

obrázek 6.2.



Obrázek 6.2: Křížení část 1 - zachování středů mezi body

Ve druhé části křížení se vytváří seznamy genů, počínaje genem za druhým náhodným bodem a konče genem před prvním náhodným bodem. Geny jsou brány cyklicky, tedy za posledním genem chromozomu následuje gen první. Seznamy jsou vytvořeny pro všechny řádky chromozomů a budou využívány pro doplňování do potomků, které je popsáno v třetí části. Seznamy jsou přiřazeny k potomkům do kříže, stejně jak je zobrazeno na obrázku 6.3. Tedy pro potomka 1 se použijí seznamy rodiče 2 a naopak.

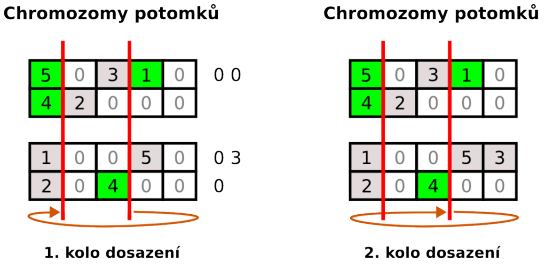


Obrázek 6.3: Křížení část 2 - vytvoření seznamů genů pro doplňování

Poslední třetí část křížení doplňuje vytvořené seznamy do potomků. Skládá se z $1 - N$ kol dle aktuálního rozložení genů. Doplňování se provádí dokud nejsou všechny seznamy prázdné a je prováděno tak, že se bere ze seznamu vždy první prvek zleva, který je po zpracování[2] smazán. Doplňování seznamů na ve dvou kolech ilustruje obrázek 6.3. V tomto případě jsou dvě kola dostačující - seznamy se ve druhém kole vyprázdní.

V 1. kole se seznamy doplňují pouze do vnějších oblastí, počínaje první genem za druhým náhodným bodem. Pokud je prvkem seznamu číslo, které už v chromozomu potomka existuje, je toto číslo ignorováno a následně odstraněno ze seznamu. V kole 2-N je doplňování prováděno nad celým chromozomem, počínaje prvním genem za druhým náhodným bodem. Doplňuje se pouze do genů s nulovou hodnotou. Nyní se už nedoplňují prvky seznamu i s nulovou hodnotou jako tomu bylo v 1. kole, ale jsou rovnou ignorovány. I zde ale platí

[2]Zpracování prvku seznamu je myšleno jeho ignorování nebo dosazení do chromozomu potomka



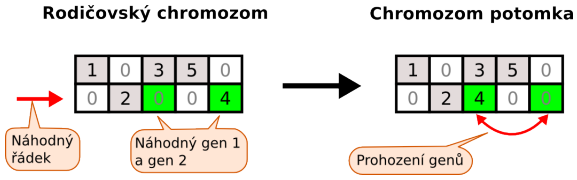
Obrázek 6.4: Křížení část 3 - doplňování seznamů genů v kolech

pravidlo, že pokud prvek seznamu už v chromozomu existuje, tak je ignorován.

6.3 Operátor mutace

Operátor pro mutaci jedinců je v genetických algoritmech využíván vždy jen v malém množství, nicméně se stává u spousty problémů dobrým nástrojem pro zvýšení diverzity populace. Do populace přináší nové informace, které mohou zabránit uváznutí v lokálním extrému. Princip mutace u celočíselných hodnot většinou spočívá ve vzájemné záměně dvou náhodně vybraných genů. Mutace, kterou jsem navrhl, je ve své podstatě jednoduchá, nicméně obsahuje několik pravidel, která zabraňují vytvoření nekorektního chromozomu.

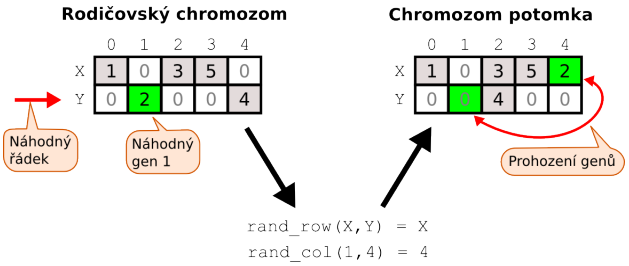
Nejprve se ve dvourozměrném poli chromozomu vybere náhodně řádek (stroj) na kterém se bude mutace provádět. Na řádku se náhodně zvolí gen a pokud je jeho hodnota rovna nule, je zaměněn s náhodně vybraným genem na stejném řádku. Tuto situaci ilustruje obrázek 6.5.



Obrázek 6.5: Jednodušší případ mutace - první náhodně vybraný gen je roven nule

V případě, že je jeho hodnota větší než nula, je mutace komplikovanější. Protože hodnota genu představuje číslo operace, a ta může běžet i na více strojích, je zde snaha ji při mutaci přesunout na náhodně zvolený stroj. Proto je pro tuto náhodně zvolenou operaci získán seznam strojů, na kterém může být operace prováděna. Ze seznamu se náhodně vybere jeden stroj a zjistí se k němu odpovídající řádek v chromozomu. Na nově vybraném řádku se může provést záměna pouze s nulovým genem. Pokud na něm nějaké nulové geny existují, je náhodně vybrán jeden z nich a geny mohou být zaměněny. Tento případ mutace osvětluje obrázek 6.6. Za povšimnutí stojí fakt, že v chromozomu po provedení mutace vznikne sloupec obsahující pouze nulové geny. Takový sloupec nemá žádný vliv na korekt-

nost chromozomu. Při vytváření procesního plánu (viz. kapitola *Simulátor plánů*, 7) se takový sloupec zcela ignoruje.



Obrázek 6.6: Složitější případ mutace - první náhodně vybraný gen představuje operaci proveditelnou jak na stroji X, tak na stroji Y

Kapitola 7

Simulátor plánů

Genetické algoritmy potřebují pro svou činnost kromě definovaných genetických operátorů také ohodnocovací funkci. Ta určuje, jak jsou daní jedinci kvalitní. Chromozomy v této aplikaci představují v určité formě plány. Aby byl plán korektní, musí splňovat určitá kritéria, která vychází ze zadání konkrétního problému. Konkrétněji, musí být dodrženo pořadí operací v jednotlivých úlohách. Kvalita chromozomu je v první části ohodnocování dána počtem chyb v nedodržení pořadí operací. Tím je genetický algoritmus nucen hledat takové plány, které jsou korektní. Pokud některý chromozom neobsahuje ani jednu chybu, je přistoupeno k druhé části ohodnocování a tou je simulace plánů. Simulace je zde potřeba z toho důvodu, že jednotlivé operace plánu nemají konstantní délku provádění, ale mají ji danou časovým intervalem. Interval udává minimální a maximální dobu provedení operace. Bylo tedy zapotřebí vytvořit simulátor, který by jednotlivé chromozomy převáděl do simulačního modelu a ten pak stochasticky simuloval. Při simulaci by se měřily nejkratší časy provádění plánu a také prodleva[1] na strojích.

Před samotnou simulací je zapotřebí převést chromozom do simulačního modelu. Ten je tvořen kalendáři provádění [9] pro jednotlivé stroje (řádky chromozomu). Kalendáře obsahují prvky, které jsou dány trojicí v následujícím formátu:

$$\langle op1, tStart, tEnd\rangle, \langle op2, tStart, tEnd\rangle, ...\langle opN, tStart, tEnd\rangle.$$

Algoritmus simulace prochází chromozom po sloupcích zleva doprava. Pro každý gen se zjistí jeho hodnota. Je-li nulová, algoritmus pokračuje následujícím genem, je-li větší než nula jedná se o gen číslem operace. Pokud je kalendář stroje na kterém operace leží prázdný, nastaví se počáteční čas $startTime$ na nulu. V jiném případě na koncový čas poslední položky kalendáře. V dalším kroku je potřeba nalézt dobu, kdy může být operace proveditelná. To je taková doba, kdy předchozí operace v plánu úkolu je v daném čase už provedena (pokud taková operace existuje). Po nalezení času, kdy může být operace provedena, se může do kalendáře přidat nová položka s touto operací. Její čas začátku provádění

[1] *Prodleva* na strojích je doba, kdy nejsou stroje využívány

bude nastaven na hodnotu *startTime* a čas konce provádění na hodnotu *startTime* + náhodně zvolená hodnota v intervalu minimální a maximální doby provádění této operace. Pseudokód pro simulaci plánu je zapsán níže.

Algoritmus 2 Pseudokód simulace plánu v podobě chromozomu

```
for col = 0 to width do
  for row = 0 to height do
    op = chromozom[row][col];

    if op = 0 then
      continue;
    end if

    if calendar[row].empty() then
      startTime = 0;
    else
      startTime = calendar[row].back().tEnd;
    end if

    while not isExecutableInTime(op, startTime) do
      startTime = startTime + 1;
    end while
    newItem.name = op;
    newItem.tStart = startTime;
    newItem.tEnd = startTime + rand(min,max);
    calendar[row].add(newItem);
  end for
end for
```

Příklad takového chromozomu, který se bude simulovat je na obrázku 7.1. Tento chromozom byl hledán genetickým algoritmem dle následujícího procesního plánu:

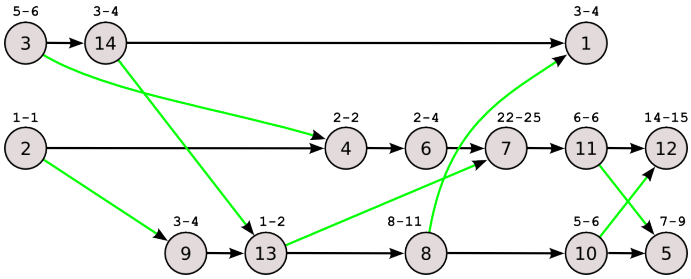
Process plans: $A \rightarrow (2, 9, 8, 1)$ $B \rightarrow (3, 4, 6, 11, 5)$ $C \rightarrow (14, 13, 7, 10, 12)$

X	3	14	0	0	0	0	0	0	0	0	1	0	0
Y	2	0	0	0	0	0	4	0	6	7	0	11	12	0
Z	0	0	0	9	13	0	0	0	8	0	0	10	5	0

Obrázek 7.1: Příklad chromozomu, který se bude simulovat

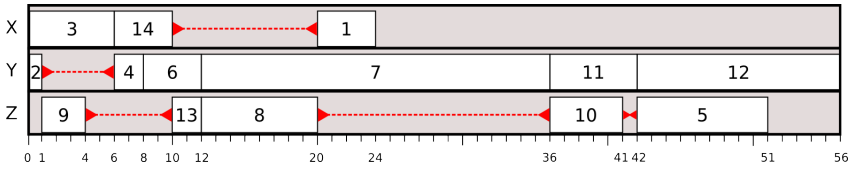
Počáteční čas jednotlivých operací není tedy závislý jen na dokončení předchozí operace na stejném stroji, ale také na předchůdcích v procesním plánu. Na druhou stranu závislost

mezi jednotlivými úlohami nevzniká žádná. To vychází z definice Job Shop problému. Závislost operací reprezentovanou grafem je možno vidět na obrázku 7.2.



Obrázek 7.2: Plán chromozomu reprezentovaný grafem s vyznačením všech závislostí

Závislosti v grafu představují ten fakt, že daná operace nesmí začít dokud nejsou předchozí operace dokončeny. Závislosti v grafu zobrazené vodorovnou černou hranou jsou závislosti pouze na daném stroji. Hrany znázorněné zeleně, představují vazbu danou plánem úlohy. Tyto hrany (závislosti) tvoří největší problém při hledání nejlepšího plánu. Jsou zdrojem zpoždění na strojích. Aby stroj provedl svou následující operaci, musí čekat na stroje jiné. V grafu lze tuto situaci ukázat například na operaci s číslem 1, která by mohla být provedena hned za operací 14. Stroj však musí čekat na dokončení úlohy s čísle 8. Na stroji tím vzniká velká prodleva, která může být v některých případech nežádoucí. Simulace tedy vytvoří kompletní časový harmonogram na jednotlivých strojích. Pro tento příklad by mohlo provedení jedné simulace být znázorněno pomocí Ganttova diagramu na obrázku 7.3.



Obrázek 7.3: Plán chromozomu reprezentovaný Ganttovým diagramem

Z takto simulovaného modelu je potřeba zjistit *délku provedení celého plánu* (makespan) a také *dobu nevyužívání strojů* (latence). Celkovou dobu lze zjisti dle poslední dokončené úlohy. Na obrázku 7.3 je to operace s číslem 12. Výpočet celkové délky plánu je dán vztahem:

$$c_{\max} = \max_{j=1}^{n}\{c_j\} \tag{7.1}$$

kde c_j je doba dokončení j-té operace a n je celkový počet operací. Pro tuto simulaci je hodnota $c_{\max} = 56$. Latence stroje je v této aplikaci počítána od doby, kdy stroj už provedl první operaci do doby provedení poslední operace. Na obrázku 7.3 jsou latence znázorněny

červenou čárkovanou čarou. Pro stroj X je latence pouze doba mezi operacemi 14 a 1. Celková latence je dána vzorcem:

$$L = \sum_{j=1}^{n} l_j \tag{7.2}$$

kde l_j je latence j-tého stroje a n značí celkový počet strojů. V tomto příkladu je hodnota $L = 38$.

Protože provedení jediné simulace nad plánem chromozomu mnoho nevypovídá, je potřeba provádět simulací několik. V každé simulaci je počítána celková doba provedení plánu a také celková latence. Z těchto hodnot se po dokončení všech simulací počítá vážený průměr. Ohodnocovací funkce genetického algoritmu pak jako svůj výsledek vrací buď celkovou délku provedení operace nebo prodlevu na strojích. To je dáno vstupním parametrem uživatele aplikace.

Kapitola 8

Transformace plánů do Petriho sítě

Cílem aplikace je pro zadanou úlohu najít nejlepší plán, který bude reprezentován Časovou Petriho sítí. Rozhodl jsem se, že pro reprezentaci Petriho sítě zvolím standardizovaný jazyk `PNML` [12]. Díky tomuto otevřenému formátu může být výstup této aplikace využitelný i v jiných nástrojích (např. pro simulaci, vizualizaci a pod.). Protože výsledné plány mohou mít nejrůznější podobu, bylo zapotřebí vytvořit obecný layout sítě, pomocí kterého by se jakýkoli plán dal transformovat do přehledné a dobře čitelné Petriho sítě. Vygenerované Petriho sítě v podobně PNML souboru lze zobrazit libovolným prohlížeči PNML. V této práci využívám jednoduchého opensource programu `PNML view`. Tento nástroj umožňuje prohlížení PNML souborů, inteligentně umisťuje popisky u objektů a umožňuje export do formátů jako PDF, EPS, SVG nebo JPEG. Domovská stránka projektu je na adrese `http://www.vanwal.nl/pnmlview`.

8.1 Jazyk PNML

`PNML` (Petri Net Markup Language) je jazyk navržený k popisu Petriho sítí založený na jazyku `XML`. Současná verze jazyka je k dispozici na referenčních stránkách `http://www.pnml.org`.

Jazyk PNML umožňuje v jednom dokumentu popsat i více Petriho sítí. Popisuje Petriho síť na základě objektů, které reprezentují její grafovou strukturu. Protože se jedná o jazyk XML, jsou těmito objekty XML elementy. Pro popis sítě jsou tedy k dispozici tyto objekty: `place`, `transition` a `arc`. V jazyce PNML existují i další objekty používané pro popis struktury sítě. Těmito objekty jsou: `pages`, `reference places` a `reference transitions`. Každý z výše uvedených objektů má svůj identifikátor, který se využívá při odkazování na objekt. K popisu vlastností objektů se využívá tzv. `labels`, přičemž každý objekt může takových popisků obsahovat i více. Nejčastěji se využívají pro označení váhy hrany, přidání jména k objektu nebo pro nastavení počtu značek v místě.

Protože se Petriho sítě popsané jazykem PNML reprezentují i graficky, je zapotřebí definovat rozložení a vzhled jednotlivých objektů. Pro místo a přechod se definuje umístění

na osách x a y. Zobrazení hrany je pak dáno počátečním místem (resp. přechodem) a koncovým přechodem (resp. místem). U hran je navíc možno definovat seznam bodů, kterými hrana povede. Komentáře a objekty mají svůj tzv. referenční bod ke kterému je pozice prvku vztažena. V případě objektu to většinou bývá střed grafické reprezentace objektu. U komentáře levý spodní roh a u hrany je to pak střed první části hrany. Většina vizualizátorů PNML se těmito pravidly moc neřídí a snaží ze komentáře rozmisťovat inteligentně tak, aby se s ničím nepřekrývaly.

Protože může být někdy potřeba rozdělit Petriho síť na několik částí, umožňuje jazyk PNML síť rozdělit na tzv. stránky (pages). Stránka představuje objekt, který je složen z dalších objektů. Aby bylo možné stránky mezi sebou propojovat, jsou zde použity tzv. referenční uzly. Při návrhu jazyka PNML bylo také myšleno na potřeby zápisu speciálních informací pro nejrůznější nástroje, které s jazykem pracují. Tyto informace se zapisují do částí pojmenovaných `tool specific information`.

8.2 Základní prvky jazyka PNML

V této části se pokusím ukázat popis základních prvků jazyka PNML. Především těch prvků, které jsou využívány v aplikaci. Jak už bylo uvedeno v předchozí kapitole, prvky Petriho sítě jsou reprezentovány pomocí XML elementů a atributů. Proto k jednotlivým prvkům ukáži i jejich zápis v podobě XML.

8.2.1 Síť

Začnu prvkem, který tvoří základ v PNML dokumentu a tím je **síť** (net). Ta je složena ze stránek (pages) a objektů popsaných níže. Tak jako každý objekt obsahuje unikátní identifikátor *id* a také *type* specifikující typ Petriho sítě, která je modelována. Každá síť taká může mít svůj název v elementu *name*. Níže pak následují ony prvky sítě. Příklad takové sítě může být následující:

```
<net id="pn1" type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
    <name>
        <text>Plan from JobShop Solver</text>
    </name>
    <place id="p1">
        ...
    </place>
    ...
</net>
```

8.2.2 Místo

Dalším popisovaným prvkem je **místo** (place). Na příkladu níže má místo identifikátor *p1*. Popisek zobrazovaný u místa je *„condition 1“* a je umístěn po pravé straně. Z elementu

initialMarking lze také vyčíst, že místo obsahuje 3 značky (tokeny). V síti ve které je toto místo umístěno se zobrazí na pozici danou v elementu *position*.

```
<place id="p1">
    <name>
        <text>condition 1</text>
        <graphics>
            <offset x="30" y="0" />
        </graphics>
    </name>
    <initialMarking>
        <text>3</text>
    </initialMarking>
    <graphics>
        <position x="250" y="500" />
    </graphics>
</place>
```

8.2.3 Přechod

Přechody (transition) mají velmi podobnou strukturu jako místa. Pouze neobsahují elementy *initialMarking*. Příklad přechodu je uveden níže:

```
<transition id="t1">
    <name>
        ...
    </name>
    <graphics>
        ...
    </graphics>
</transition>
```

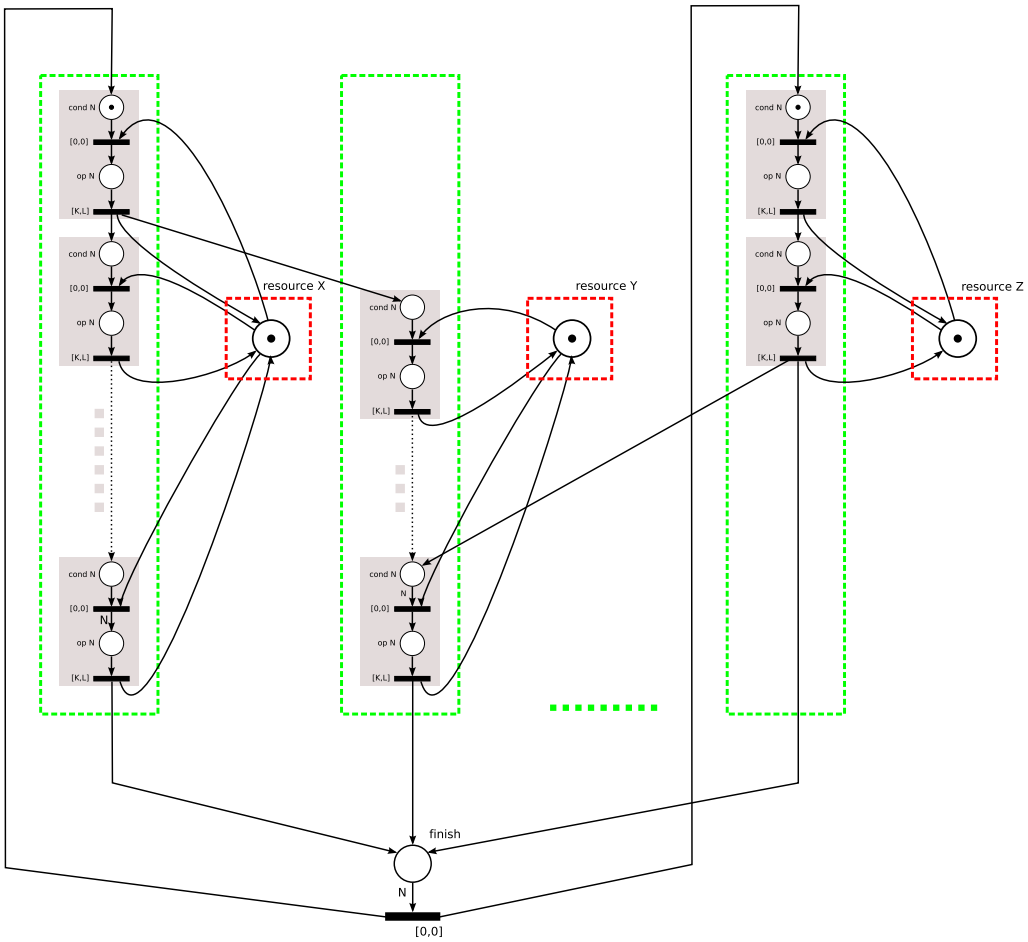
8.2.4 Hrana

Posledním zde uváděným prvkem jazyka PNML je **hrana** (arc). Ta využívá identifikátorů míst a přechodů a to tak, že je na základě nich propojuje. Hrana na příkladu níže vede z místa *p1* do přechodu *t1*. Stejně tak jako u předchozí prvků lze specifikovat její název a jeho umístění. Váha hrany, která je v tomto případě 1 se zapisuje do elementu *inscription*.

```
<arc id="a1" source="p1" target="t1">
    <name>
        ...
    </name>
    <inscription>
        <text>1</text>
        <graphics>
            <offset x="30" y="0" />
        </graphics>
    </inscription>
</arc>
```

8.3 Layout sítě

Hlavní motivace pro vytvoření obecného layoutu byla schopnost zobrazovat Petriho sítě dle určitých pravidel a také z důvodu lepší čitelnosti. Aby bylo možné jednoduše modelovat prvky Petriho sítě, bylo zapotřebí vytvořit třídu pro lepší práci s XML elementy jazyka PNML. Tato třída, kterou jsem pojmenoval `PNMLBuilder` tvoří určité API pro práci s prvky PNML popsané v předchozí kapitole. Při vytváření modelu Petriho sítě se uživatel této třídy nemusí vůbec starat o XML elementy. Takto vytvořená abstrakce umožňuje snadnější rozmísťování, propojování a nastavování prvků Petriho sítě. Navržený layout s ukázkou rozmístěním několika operací je vidět na obrázku 8.1.



Obrázek 8.1: Layout pro rozložení prvků Petriho Sítě

Základem layoutu jsou bloky operací znázorněné zelenou čárkovanou čarou. Každý takový blok představuje operace, které mají společní stroj. Stroje jsou tedy svázány s těmito zelenými bloky a to tak, že pro každý náleží jeden stroj. Jednotlivé operace jsou znázorněny šedými bloky. Operace je v Petriho síti reprezentována dvěma místy a dvěma přechody.

První místo představuje podmínku, která musí být platná (musí obsahovat tolik značek, kolik do ní vede hran), aby byla operace proveditelná. Za touto podmínkou následuje časový přechod, který je proveditelný vždy v nulovém čase. Provádění operace vyjadřuje poslední přechod šedého bloku, kde je uveden časový interval provádění operace.

Rozmístění operací v zeleném bloku je provedeno tak, aby hrany představující závislosti mezi zelenými bloky vedly vždy odshora dolů. Přestože se tak celková Petriho síť může na výšku podstatně zvětšit, značně zlepšuje čitelnost a také lze částečně vidět časová souslednost jednotlivých operací. Každá operace si na počátku vezme značku ze stroje a po provedení ji vrátí. Operace, jejichž čas provádění je ve výsledném plánu nulový, jsou zobrazeny na samém vrcholu zeleného bloku. Protože tyto operace začínají celý proces provádění plánu, musí obsahovat jednu značku. Poslední operace zelených bloků vedou hranou do koncového místa a přechodu. Odtud se pak značky vrací do počátečních operací.

Kapitola 9

Testování a výsledky

V první části bych chtěl prezentovat optimální parametry genetického algoritmu pro řešení problému plánování Job Shop. V další části pak ukáži, jak může pro různě zadané problémy probíhat hledání optimálního plánu. Uváděné testy byly prováděny na procesoru *Intel Core 2 Duo* s frekvencí procesoru *1660 MHz* a pamětí *2048 MB* typu *DDR2*.

9.1 Parametry GA

Aby bylo možné plánovací problémy řešit v rozumném čase, bylo zapotřebí nalézt optimální genetický algoritmus a také jeho parametry. Jak už jsem uvedl v kapitole 5.3, využíval jsem k řešení problému knihovnu `GALib`, která umožňuje výběr mezi několika algoritmy (Simple, Steady-State, apod.). Abych si ale snížil počet možných variant, vyhledal jsem si existující implementace genetické optimalizace plánování (Job Shop) a zjistil, že nejčastěji se využívá algoritmu `Steady-State`. Pro další testování jsem se tedy soustředil především na tento algoritmus.

Testování různých parametrů probíhalo na hledání problému o 2 strojích a 20 operacích. Cílem bylo pro každou kombinaci parametru změřit dobu nalezení prvního plánu a ve kterém kole GA byl nalezen. Ve výsledcích uvedených v tabulce 9.1 jsem nejmenší časy (do 20s) a nejmenší kola (do 3) nalezení plánu zvýraznil pro přehlednost tučně. Z dat je patrné, že pro populaci 10 vychází výsledky nejlépe. Bohužel se pak ukázalo, že takto veliká populace nedostačuje kvůli špatné konvergenci. Pokud se tedy ignorují všechny výsledky s velikostí populace 10, zůstávají zde stále dobré výsledky pro hodnotu křížení 0.5 a velikost populace 50. Tyto výsledky jsem znázornil červeně. Po provedení několika dalších experimentů jsem tyto parametry drobně upravil a jejich výsledná podoba je v tabulce 9.2.

9.2 Závislost hledání na zadané úloze

V této části se pokusím popsat, jak se mění složitost hledání plánu na zadání úlohy. Tuto závislost budu demonstrovat na dvou kritériích, kterými jsou počet operací a počet strojů

Křížení	Populace	Mutace	Doba nalezení	Kolo nalezení
0.2	10	0.2	**4s**	**3**
		0.5	**2s**	**2**
		0.8	**6s**	**3**
	25	0.2	**6s**	**3**
		0.5	**4s**	**2**
		0.8	1m 32s	21
	50	0.2	3m 15s	30
		0.5	3m 55s	28
		0.8	-	-
0.5	10	0.2	**2s**	**2**
		0.5	**5s**	**2**
		0.8	52s	**2**
	25	0.2	**6s**	**2**
		0.5	1m 21s	16
		0.8	1m 10s	11
	50	0.2	**9s**	**2**
		0.5	**18s**	**2**
		0.8	**7s**	**2**
0.8	10	0.2	43s	14
		0.5	35s	11
		0.8	**7s**	**3**
	25	0.2	3m 14s	31
		0.5	1m 23s	13
		0.8	2m 10s	19
	50	0.2	4m 45s	28
		0.5	-	-
		0.8	8m 46s	31

Tabulka 9.1: Srovnání času nalezení prvního plánu na různých parametrech GA

používaných v úloze. Při testování jsem nastavoval běh genetického algoritmu na několik kol. Běh algoritmu se po každém kole jakoby restartuje a generátor náhodných čísel se inicializuje novou hodnotou. Testování probíhalo při následujících parametrech genetického algoritmu: *30 000 generací*, *50 jedinců populace*, *pravděpodobnost mutace 1.0* a *pravděpodobnost křížení 0.4*. Jednotlivé testy jsem opakoval a výsledná hodnota pak tvořila aritmetický průměr těchto testů.

Aby bylo srovnání jednotlivých závislostí přehlednější, ukáži výsledná data formou tabulek. Tabulky pro každý uvedený počet operací/strojů udávají počet korektních plánů ze všech generací. Tento údaj tvoří průměr a je vyjádřen v procentech, aby byl lépe porovnatelný mezi jinými daty. Dále je uveden počet plánů, to je počet nalezených korektních plánů vzhledem k zadání úlohy. Mezi těmito plány se vybírá jeden nejlepší. Pro každý počet operací/strojů také uvádím dobu nalezení prvního plánu, kolo běhu GA ve kterém byl plán nalezen a také celkový procesorový čas. Nalezení prvního (korektního) plánu je pro samotný průběh GA velmi klíčové. Do populace se tím dostane korektní jedinec a díky

Parametr	Hodnota/Typ
Typ algoritmu	GASteadyStateGA
Selektor	GAUniformSelector
Počet generací	30000
Velikost populace	50
Pravděp. mutace	1.0
Pravděp. křížení	0.4
Pravděp. nahrazení	0.1

Tabulka 9.2: Nejlepší nalezené parametry GA

němu je pak nalezení dalších korektních plánů pro GA velmi snadné. Jednoduše řečeno, genetický algoritmus se v první fázi snaží do populace nagenerovat nějaké korektní plány. V další fázi už se pouze tyto plány drobně upravují nebo kříží mezi sebou, tak aby byl nalezen ten nejlepší plán. Tato druhá fáze je obecně časově náročnější z toho důvodu, že jsou v populaci korektní plány, které je zapotřebí simulovat. Simulace plánů neboli zjišťování její celkové doby provádění (případně prodlevy) na strojích není i přes nejrůznější optimalizační úpravy úplně rychlé. Pro představu, při tomto testování se musí simulace provádět $10kol * 30000gen * 50pop = 15\ 000\ 000$ krát. Což při časové náročnosti simulace není zrovna zanedbatelné číslo.

Závislost počtu operací v zadání problému na hledání plánů genetickým algoritmem je vidět v tabulce 9.3. Při tomto testování byl zvolen *počet kol na hodnotu 10*. V tabulce je patrné, že pro menší počet operací bylo procento korektních plánů v populaci větší. Doba potřebná pro nalezení prvního plánu roste s přibývajícím počtem operací exponenciálně. To může být patrné na grafu 9.1 vlevo. Počet kol pro nalezení prvního plánu s počtem operací roste a dalo by se říct, že je úměrný k době nalezení prvního plánu. Celková doba provádění plánu je hodně závislá na nalezení prvních plánů. Pokud se tak stane při samotném počátku hledání je časová složitost kvůli prováděné simulace o poznání větší.

Počet operací	Korektních plánů v pop.	Počet plánů	Doba nalezení prvního plánu	Kolo nalezení prvního plánu	Celkový čas
5	56%	5	0.00s	1	4m 20s
10	37%	6	0.06s	1	15m 29s
15	32%	5	11.37s	2	21m 52s
20	25%	16	16.02s	2	26m 15s
25	12%	20	21.73s	2	21m 23s
30	1%	23	60.75s	3	23m 29s
35	1%	27	91.31s	5	28m 12s

Tabulka 9.3: Závislost počtu operací v zadání problému na hledání plánů

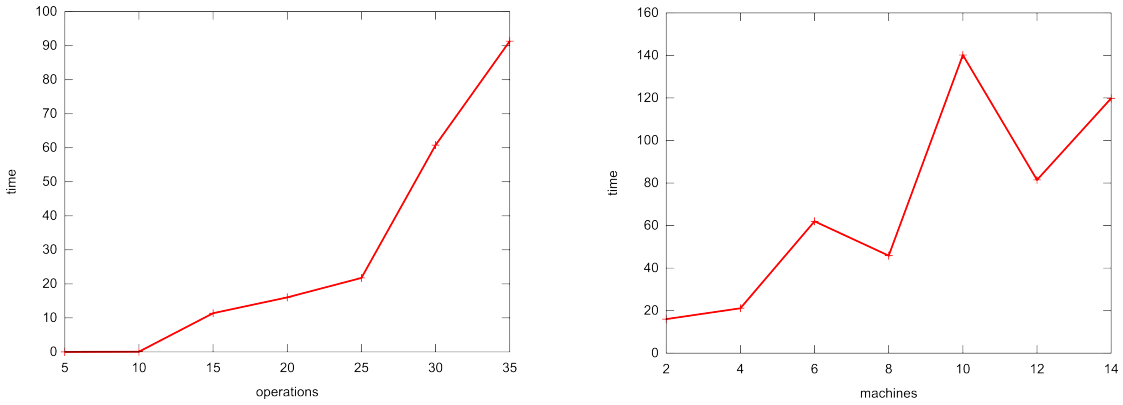
Tabulka 9.4 zobrazuje závislost počtu strojů na samotném hledání plánů genetickým algoritmem. Testování probíhalo na *3 kolech* GA. Z dat je možné vyčíst, že procento ko-

rektních plánů v populaci téměř nijak nesouvisí s počte strojů, tak jako tomu bylo u operací. Doba nalezení prvního plánu není tolik úměrná počtu strojů. To lze vidět na obrázku 9.1 vpravo. Celková doba hledání se s počtem strojů poměrně značně zvětšuje.

Počet strojů	**Korektních plánů v pop.**	**Počet plánů**	**Doba nalezení prvního plánu**	**Kolo nalezení prvního plánu**	**Celkový čas**
2	19%	13	16.02s	2	6m 37s
4	30%	11	21.11s	2	32m 23s
6	18%	6	62.00s	3	24m 10s
8	19%	5	45.87s	2	29m 15s
10	5%	2	140.19s	3	12m 32s
12	32%	1	81.44s	3	1h 4m 11s
14	23%	5	119.83s	2	1h 0m 16s

Tabulka 9.4: Závislost počtu strojů v zadání problému na hledání plánů

Po závěrečném srovnání lze říci, že doba potřebná pro nalezení prvního plánu roste více s přibývajícím počtem operací než s počtem strojů. Je to dáno tím, že větší počet strojů dává genetickému algoritmu určitou volnost při hledání plánu. Zajímavým faktem taky je, že s přibývajícím počtem operací roste počet nalezených plánů a u strojů je tomu přesně naopak. Důvodem toho je, že s počtem operací se chromozom zvětšuje do šířky a s počtem strojů do výšky. Implementovaný operátor křížení své dva náhodné body (řezy) generuje vertikálně. To způsobuje, že s rostoucím počtem operací vzniká lepší diverzita populace a tak je vetší šance nalezení více plánů.



Obrázek 9.1: Graf nalezení prvního plánů v závislosti na počtu operací (vlevo) a na počtu strojů (vpravo)

Kapitola 10

Příklady využití

Protože jsem aplikaci psal spíše obecněji, uvedu zde některé její praktická využití. Přestože jsou příklady modelového charakteru, snažil jsem se, aby se co nejvíce blížily praxi.

10.1 Oblast plánování výroby

- **Zadání**

  Stolářská firma se rozhodne zaměřit se na výrobu pěti hlavních produktů, kterými jsou:

  - čelo postele
  - vchodové dveře
  - pracovní stůl
  - podstavec tv
  - rám zrcadla

  Výroba těchto produktů bude prováděna ve firmě současně a budou k ní využívány dostupné stroje. Firma má k dispozici *formátovací pilu*, *pásovou pilu*, *spodní frézu*, *vodorovnou vrtačku* a *pásovou brusku*. Každý z výrobků potřebuje pouze některé stroje a jejich pořadí je pevně dáno výrobním procesem.

  Pro výrobu *čela postele* je nejprve zapotřebí použít formátovací pilu, následně vodorovnou vrtačku a v poslední fázi pásovou pilu. Vyrobení *vchodových dveří* vyžaduje použití pásové pily, vodorovné vrtačky, spodní frézy a nakonec pásové brusky. Výroba *pracovního stolu* využívá pouze spodní frézu a následně formátovací pilu. *Podstavec tv* je nejprve opracováván na vodorovné vrtačce, dále na pásové pile a v poslední fázi je použita formátovací pila. Poslední výrobek - *rám zrcadla* - je zpracováván na spodní fréze, pásové pile, pásové brusce a nakonec na vodorovné vrtačce.

  Pro přehlednost jsou tyto informace uvedeny v tabulce 10.1. Jednotlivé operace na daných strojích jsou zde očíslovány jednoznačným identifikátorem. Tabulka je také

doplněna o časový odhad zapsaný intervalem pro jednotlivé úkoly. Ten je zapsán v hranatých závorkách a časovou jednotkou jsou hodiny.

Výrobek	Vodor. vrtačka	Formát. pila	Pásová pila	Spodní fréza	Pásová bruska
čelo postele	1 [4-5]	2 [6-8]	3 [4-6]		
vchodové dveře	4 [3-3]		5 [11-15]	6 [8-8]	7 [1-3]
pracovní stůl		8 [9-9]		9 [5-6]	
podstavec tv	10 [1-2]	11 [3-4]	12 [1-2]		
rám zrcadla	13 [2-2]		14 [9-11]	15 [1-3]	16 [4-7]

Tabulka 10.1: Očíslování jednotlivých úkolů, které jsou pro konkrétní výrobek prováděny na daném stroji

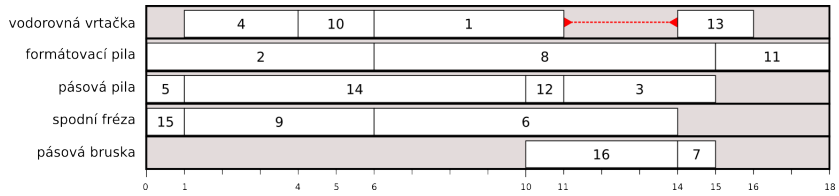
Cílem je najít takové uspořádání úkolů na strojích, aby prodleva (kdy jsou stroje nečinné) byla co nejmenší.

- **Řešení**

  Aby mohl být tento problém aplikací řešen, je potřeba jej nejprve přepsat do požadované formy. Vytvořený soubor se zadáním tohoto problému je následující:

```
jobs        = celo_postele vchodove_dvere pracovni_stul podstavec_tv ram_zrcadla
operations  = 16
machines    = vodor_vrtacka format_pila pasova_pila spodni_freza pasova_bruska
plans       = celo_postele:(2,1,3) vchodove_dvere:(5,4,6,7) pracovni_stul:(9,8)
              podstavec_tv:(10,12,11) ram_zrcadla:(15,14,16,13)
utilization = vodor_vrtacka:{1,4,10,13} format_pila:{2,8,11} pasova_pila:{3,5,12,14}
              spodni_freza:{6,9,15} pasova_bruska:{7,16}
duration    = 1:4-5 2:6-8 3:4-6 4:3-3 5:1-2 6:8-8 7:1-3 8:9-9 9:5-6 10:1-2 11:3-4
              12:1-2 13:2-2 14:9-11 15:1-3 16:4-7
```

  Nalezený plán zobrazený Ganttovým diagramem je na obrázku 10.1. Prodleva v plánu je pouze na jednom stroji a v obrázku je znázorněna červenou čárkovanou čarou.



Obrázek 10.1: Výsledný plán v Ganttovém diagramu

Aplikace pro tento úkol nalezla optimální plán, který je reprezentován Časovou Petriho sítí A.2. V logu, který je uveden v příloze A jsou pak vidět výsledná data, jak je vypisuje aplikace. Nalezený plán má dobu prodlevy 3 hodiny, což vzhledem k celkovému času provádění na všech strojích, který činí 81 hodin, je dostačující výsledek.

10.2 Oblast managementu projektů

- **Zadání**

  Mějme menší firmu zabývající se vývojem SW. V současné době jsou ve firmě méně nebo více rozpracovány 3 projekty. Na těchto projektech pracují zaměstnanci Karel, Michal a Tomáš. Jednotlivé projekty se skládají z úkolů, které je potřeba provést, aby byl projekt dokončen. Na dokončení všech úkolů má firma vyhrazeno 12 pracovních dní při pracovní době 8 hod/den.

  Projekty a jejich úkoly s odhadovanou dobou v hodinách jsou uvedeny v tabulce 10.2. Každý úkol má své číslo pro jednoznačnou identifikaci.

Projekt 1		Projekt 2		Projekt 3	
Úkol	Čas	Úkol	Čas	Úkol	Čas
1. Analýza	6-9	7. Úprava designu	7-8	10. Optimalizace	9-9
2. Návrh GUI	8-8	8. Úprava DB	2-3	11. Testování	16-18
3. Kostra aplikace	2-3	9. Testování	11-12	12. Dokumentace	23-25
4. 1/2 implementace	18-22			13. Instalace	3-5
5. 2/2 implementace	24-27				
6. Testování	20-21				

Tabulka 10.2: Jednotlivé úkoly projektů s uvedeným odhadem času

  Protože ne každý zaměstnanec se hodí na libovolnou práci, je třeba specifikovat schopnosti jednotlivých zaměstnanců. `Karel` má největší zkušenosti s *testováním*, *implementací* a *psaním dokumentace*. `Michal` je vhodný adept na práci s *grafikou* a také na *psaní dokumentací*. `Tomáš` především provádí *analýzy*, *implementaci*, *optimalizaci* a také *instalace u zákazníků*.

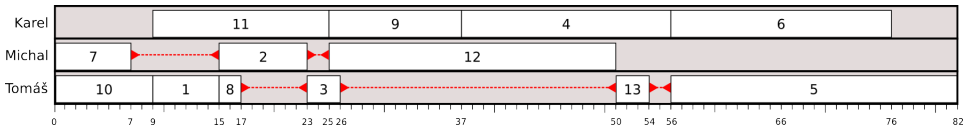
  Cílem je nalézt optimální rozdělení úkolů mezi dané tři pracovníky v časovém pořadí tak, aby všechny tři projekty byly dokončeny do požadovaných 12 pracovních dnů.

- **Řešení**

  Zadání přepíšeme do struktury srozumitelné aplikaci. Vstupní soubor obsahující zadání problému bude následující:

```
jobs         = Projekt1 Projekt2 Projekt3
operations   = 13
machines     = Karel Michal Tomas
plans        = Projekt1:(1,2,3,4,5) Projekt2:(7,8,9) Projekt3:(10,11,12,13)
utilization = Karel:{3,4,5,6,8,9,11,12} Michal:{2,7,12} Tomas:{1,3,4,5,8,10,13}
duration     = 1:6-9 2:8-8 3:2-3 4:18-22 5:24-27 6:20-21 7:7-8 8:2-3 9:11-12
               10:9-9 11:16-18 12:23-25 13:3-5
```

Pro tento vstup aplikace nalezla výsledný plán, který je transformován do Časované Petriho sítě ve formátu PNML. Na obrázku B.2 je pak grafické zobrazení této Časové Petriho sítě. Pro vizualizaci byl použit nástroj `PNML view`. Nalezený plán reprezentovaný Ganttovým diagramem je na obrázku 10.2. V příloze B je také výstupní výpis aplikace se zobrazenými výsledky. Jak je z uvedeného výpisu patrné, minimální doba potřebná pro provedení nalezeného plánu je *78 hodin* a maximální doba je *82 hodin*. To tedy splňuje zadané kritérium, že projekty musí být splněny do 12 pracovních dnů, které odpovídají 96 hodinám. Doba hledání výsledného plánu byla 18s.



Obrázek 10.2: Výsledný plán v Ganttovém diagramu

Kapitola 11

Závěr

Práce se zabývala problematikou plánování a rozvrhování, seznamuje s principy genetických algoritmů a také uvádí do teorie Petriho sítí se zaměřením na Časové Petriho sítě.

Na základě získaných znalostí byla vytvořena aplikace `Job Shop Solver`. Ta byla napsána v jazyce C++ a je určena pro linuxové operační systémy. Aplikace je schopná hledat nejlepší plány pro zadané problémy. Soustředil jsem se především na řešení nejpoužívanějšího rozvrhovacího problému typu Job Shop. Ten jsem upravil do ještě obecnější podoby přidáním časových intervalů pro dobu jednotlivých operací. Tím se ale hledání plánů poměrně značně zkomplikovalo a bylo tak potřeba vytvořit simulátor, který by v GA při ohodnocování jedinců nalezené plány stochasticky simuloval. Po dokončení hledání se nejlepší nalezený plán převádí do Časové Petriho sítě, která je popsána za pomoci standardizovaného jazyka PNML. Pro vizualizaci Petriho sítí popsaných v jazyce PNML existuje řada nástrojů. V této práci jsem využíval jednoduchého nástroje `PNML view`. Přestože bylo při implementaci využito knihovny `GALib` a také ve velké míře knihovny `Boost`, počet řádků zdrojového textu přesáhl 7 tisíc. V závěru práce je také provedeno porovnání doby hledání na různých parametrech genetického algoritmu a také je zde porovnána závislost hledání na různě zadaných plánovacích problémech.

Další vývoj aplikace by mohl spočívat v rozšiřování schopnosti zvládat více typů plánovacích problémů a také větší variability v zadání. S tímto rozšířením počítá i současně navržený objektový model aplikace. Aplikace by v současné době mohla sloužit jako jádro většího plánovacích systému zaměřeného na konkrétní oblast využití (např. plánování výroby nebo v oblasti managementu).

Literatura

[1] Blazewicz, J.: *Scheduling in Computer and Manufacturing Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996, ISBN 0387614966.

[2] Brucker, P.: Complex scheduling problems. *Zeitschrift Oper. Res*, ročník 30, 1999: str. 99.

[3] Cigler, J.: *Programování s omezujícími podmínkami v Scheduling Toolboxu*. Diplomová práce, České vysoké učení technické v Praze, 2007.

[4] Deb, K.; Schmidt, M.; Stidsen, T.: *Evolutionary Algorithms for Engineering Applications*. Springer-Verlag Berlin, 2001, ISBN 3540620214.

[5] Friedman, M.: Job Shop Scheduling. 1997.

[6] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1997, ISBN 0-201-63361-2.

[7] Josef, S.: Aplikované evoluční algoritmy. 2006.

[8] Kvasnička, V.; Pospíchal, J.; Tiňo, P.: *Evoluční algoritmy*. STU Bratislava, 2000, ISBN 80-227-1377-5.

[9] Peringer, P.: Simulační nástroje a techniky. `https://www.fit.vutbr.cz/study/courses/SNT/public/Prednasky/SNT.pdf`, 2009.

[10] Pinedo, M.: *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, druhé vydání.

[11] Pošík, P.: Genetické algoritmy. `http://www.volny.cz/posa/skola/pga/theory/ga-theory.htm`, 2002.

[12] Wikipedia: Petri Net Markup Language. `http://en.wikipedia.org/wiki/Petri_Net_Markup_Language`, 2008.

[13] Češka, M.; Marek, V.; Novosad, P.; aj.: *Petriho sítě*. 2006, ISBN 80-85867-35-4.

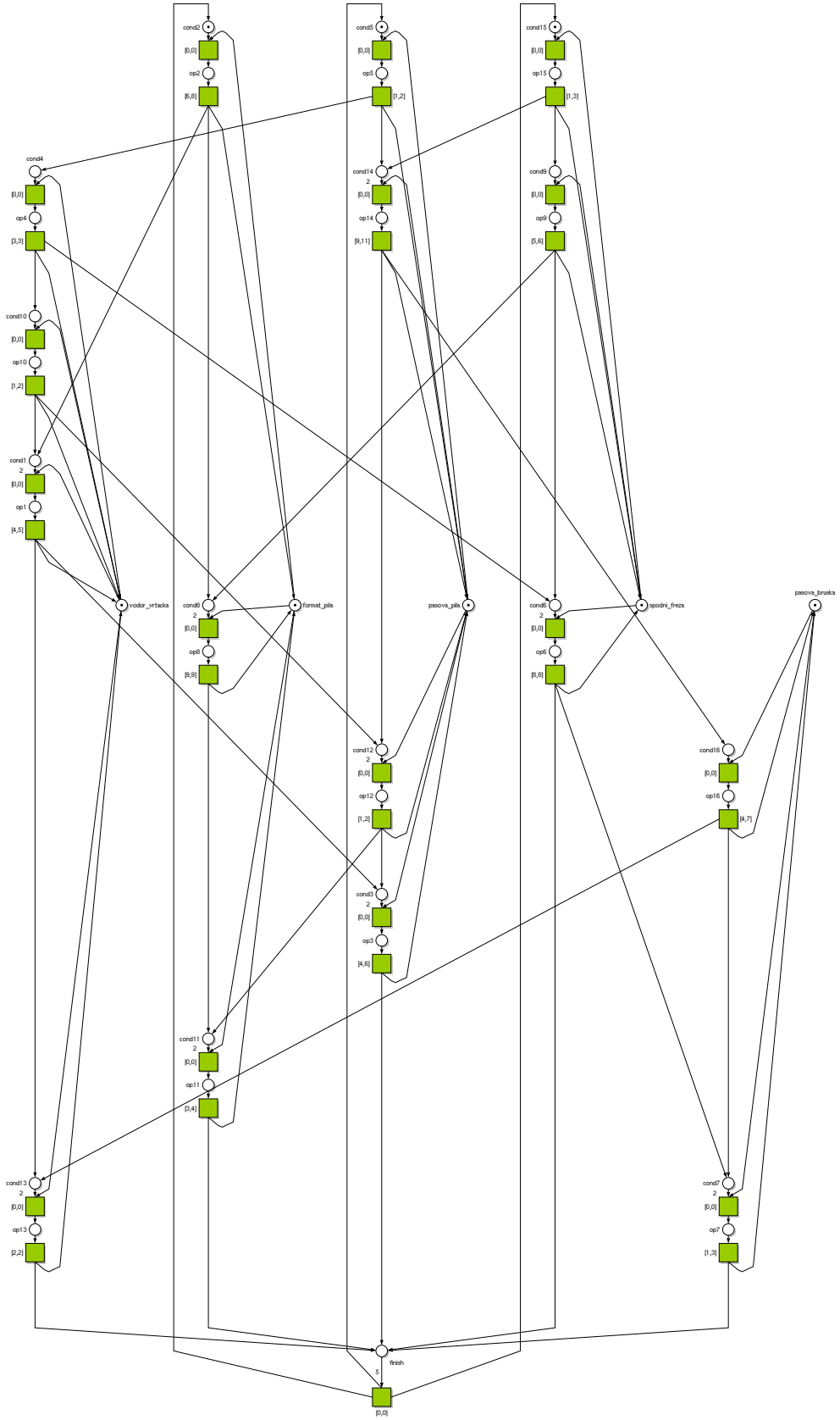
Seznam příloh

Dodatek A

Plánování výroby - výsledky

```
--- 23:01:11.632 jss DEBUG3: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.632 jss DEBUG3: ********************************************************** FINAL RESULTS **********************************************************************************
--- 23:01:11.632 jss DEBUG3: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.632 jss DEBUG3: BEST GENOMES:
--- 23:01:11.632 jss DEBUG2:    Best genome       >>> Fitness:    0.037037; Total time:      19; Idle time:          3;
--- 23:01:11.632 jss DEBUG2:
-----------------------------------------------
| 0  0  4  0  0  0  0  0  0 10  0  0  1 13  0  0|
| 2  0  0  0  0  0  0  0  0  8  0  0  0  0  0 11|
| 5  0  0  0  0  0  0  0 14  0  0  0  0  0  0 12  3|
| 0  0  0 15  0  0  0  0  9  0  0  0  0  0  6  0  0|
| 0  0  0  0  0  0  0  0  0 16  0  0  0  0  0  0  7|
-----------------------------------------------
--- 23:01:11.632 jss DEBUG2:    Best genome       >>> Fitness:   0.0493827; Total time:      20; Idle time:          4;
--- 23:01:11.633 jss DEBUG2:
-----------------------------------------------
| 0  0  0  0  0  4  0  0  0 10  0  0  1 13  0  0|
| 0  0  0  0  0  0  2  0  0  8  0  0  0  0  0 11|
| 0  0  0  0  5  0  0 14  0  0  0  0  0  0 12  3|
| 0  0  0  0  0 15  0  9  0  0  0  0  0  6  0  0|
| 0  0  0  0  0  0  0  0  0  0 16  0  7  0|
-----------------------------------------------
--- 23:01:11.633 jss DEBUG2:    Best genome       >>> Fitness:   0.0617284; Total time:      19; Idle time:          5;
--- 23:01:11.633 jss DEBUG2:
-----------------------------------------------
| 0  0  0  0  0  4  0  0  0 10  0  0  1 13  0  0|
| 2  0  0  0  0  0  0  0  0  8  0  0  0  0  0 11|
| 0  0  0  0  5  0  0 14  0  0  0  0  0  0 12  3|
| 0  0  0 15  0  0  0  9  0  0  0  0  0  6  0  0|
| 0  0  0  0  0  0  0  0 16  0  0  0  0  0  0  7|
-----------------------------------------------
--- 23:01:11.634 jss DEBUG3: Correct plans:      91754      30%
--- 23:01:11.634 jss DEBUG3: Incorrect plans:   208286      69%
--- 23:01:11.634 jss DEBUG3: solutions:              7
--- 23:01:11.634 jss DEBUG3: **********************************************************************************************************************************************************
--- 23:01:11.634 jss INFO  : Plan was found for task:
--- 23:01:11.634 jss DEBUG2: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.634 jss DEBUG2: TASK:
--- 23:01:11.634 jss DEBUG2:                    jobs: celo_postele vchodove_dvere pracovni_stul podstavec_tv ram_zrcadla
--- 23:01:11.634 jss DEBUG2:              operations: 16
--- 23:01:11.634 jss DEBUG2:                machines: vodor_vrtacka format_pila pasova_pila spodni_freza pasova_bruska
--- 23:01:11.634 jss DEBUG2:                   plans: celo_postele:(2,1,3) vchodove_dvere:(5,4,6,7) pracovni_stul:(9,8) podstavec_tv:(10,12,11) ram_zrcadla:(15,14,16,13)
--- 23:01:11.634 jss DEBUG2:             utilization: vodor_vrtacka:{1,4,10,13} format_pila:{2,8,11} pasova_pila:{3,5,12,14} spodni_freza:{6,9,15} pasova_bruska:{7,16}
--- 23:01:11.634 jss DEBUG2:                duration: 1:4-5 10:1-2 11:3-4 12:1-2 13:2-2 14:9-11 15:1-3 16:4-7 2:6-8 3:4-6 4:3-3 5:1-2 6:8-8 7:1-3 8:9-9 9:5-6
--- 23:01:11.634 jss DEBUG2: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.634 jss INFO  : Transforming plan to Petri Net...
--- 23:01:11.635 jss DEBUG2: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.635 jss DEBUG2: SIM MODEL MIN:
--- 23:01:11.636 jss DEBUG3:        Data:
                                        -----------------------------------------------
                                        | 0  0  4  0  0  0  0  0  0 10  0  0  1 13  0  0|
                                        | 2  0  0  0  0  0  0  0  0  8  0  0  0  0  0 11|
                                        | 5  0  0  0  0  0  0 14  0  0  0  0  0  0 12  3|
                                        | 0  0  0 15  0  0  0  9  0  0  0  0  0  6  0  0|
                                        | 0  0  0  0  0  0  0  0  0 16  0  0  0  0  0  0  7|
                                        -----------------------------------------------
--- 23:01:11.636 jss DEBUG3:       Model:
--- 23:01:11.636 jss DEBUG3:                Machine 1:    4|< 1, 4)|  10|< 4, 5)|   1|< 6,10)|  13|<14,16)|
--- 23:01:11.636 jss DEBUG3:                Machine 2:    2|< 0, 6)|   8|< 6,15)|  11|<15,18)|
--- 23:01:11.636 jss DEBUG3:                Machine 3:    5|< 0, 1)|  14|< 1,10)|  12|<10,11)|   3|<11,15)|
--- 23:01:11.636 jss DEBUG3:                Machine 4:   15|< 0, 1)|   9|< 1, 6)|   6|< 6,14)|
--- 23:01:11.636 jss DEBUG3:                Machine 5:   16|<10,14)|   7|<14,15)|
--- 23:01:11.636 jss DEBUG3:  Total time:   18
--- 23:01:11.636 jss DEBUG3:   Idle time:    5
--- 23:01:11.636 jss DEBUG2: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.637 jss DEBUG2: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.637 jss DEBUG2: SIM MODEL MAX:
--- 23:01:11.637 jss DEBUG3:        Data:
                                        -----------------------------------------------
                                        | 0  0  4  0  0  0  0  0  0 10  0  0  1 13  0  0|
                                        | 2  0  0  0  0  0  0  0  0  8  0  0  0  0  0 11|
                                        | 5  0  0  0  0  0  0 14  0  0  0  0  0  0 12  3|
                                        | 0  0  0 15  0  0  0  9  0  0  0  0  0  6  0  0|
                                        | 0  0  0  0  0  0  0  0  0 16  0  0  0  0  0  0  7|
                                        -----------------------------------------------
--- 23:01:11.637 jss DEBUG3:       Model:
--- 23:01:11.638 jss DEBUG3:                Machine 1:    4|< 1, 4)|  10|< 4, 5)|   1|< 6,10)|  13|<17,19)|
--- 23:01:11.638 jss DEBUG3:                Machine 2:    2|< 0, 6)|   8|< 7,16)|  11|<16,20)|
--- 23:01:11.638 jss DEBUG3:                Machine 3:    5|< 0, 1)|  14|< 1,12)|  12|<12,14)|   3|<14,18)|
--- 23:01:11.638 jss DEBUG3:                Machine 4:   15|< 0, 1)|   9|< 1, 7)|   6|< 7,15)|
--- 23:01:11.638 jss DEBUG3:                Machine 5:   16|<12,17)|   7|<17,18)|
--- 23:01:11.638 jss DEBUG3:  Total time:   20
--- 23:01:11.638 jss DEBUG3:   Idle time:    9
--- 23:01:11.638 jss DEBUG2: ----------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:01:11.643 jss INFO  : Saving petri net to file "plan.pnml"...
--- 23:01:11.648 jss INFO  : Ending JobShop Solver...
```

Obrázek A.1: Výstupní výpis aplikace se zobrazením výsledku hledání



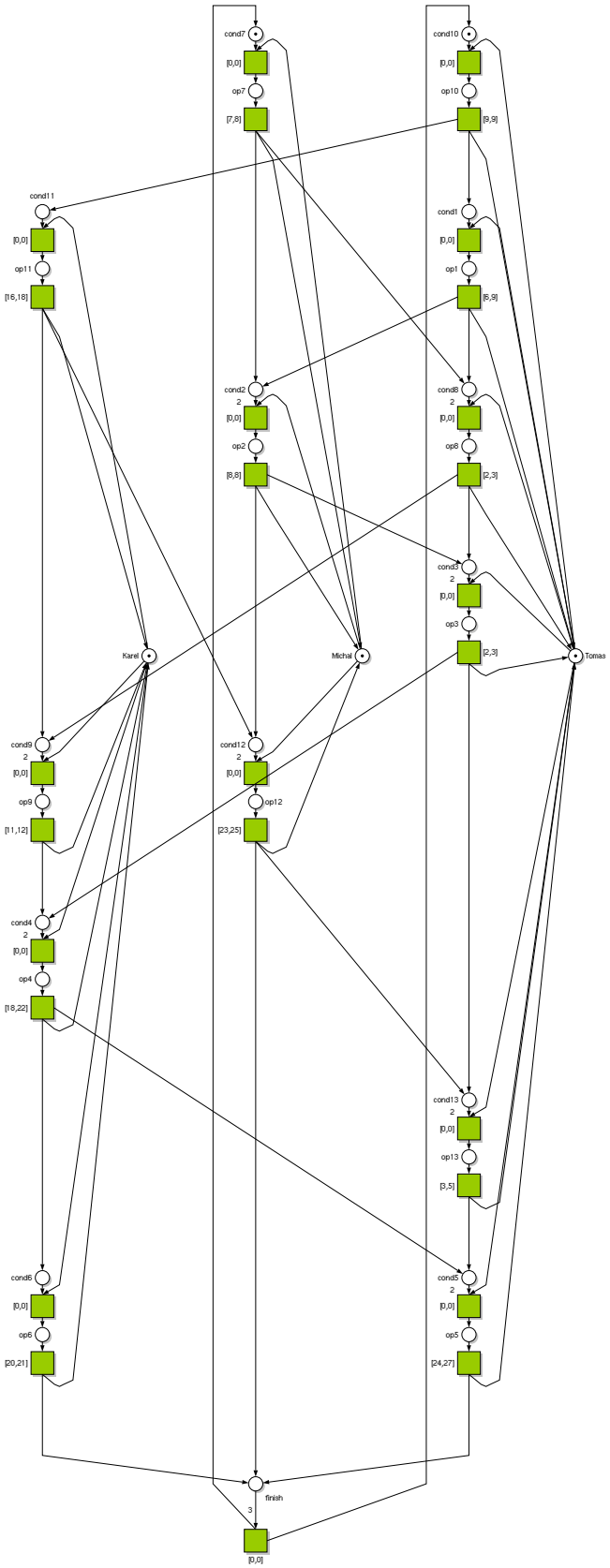
Obrázek A.2: Výsledná Časová Petriho síť

Dodatek B

Projektový management - výsledky

```
--- 23:03:52.136 jss DEBUG3: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.136 jss DEBUG3: ********************************************************** FINAL RESULTS ***********************************************************************************
--- 23:03:52.136 jss DEBUG3: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.136 jss DEBUG3: BEST GENOMES:
--- 23:03:52.136 jss DEBUG2:    Best genome       >>> Fitness:    0.464706; Total time:      79; Idle time:         41;
--- 23:03:52.136 jss DEBUG2:
----------------------------------------
| 0  0  0  0  0  0 11  0  0  9  4  6|
| 0  0  0  0  0  0  7  0  0  2 12  0|
| 0  0  0  0 10  0  1  0  0  8  3 13  5|
----------------------------------------
--- 23:03:52.136 jss DEBUG2:    Best genome       >>> Fitness:    0.511765; Total time:      87; Idle time:         55;
--- 23:03:52.136 jss DEBUG2:
----------------------------------------
| 0  0  0  0  0  0 11  0  0  9  4  6|
| 0  0  0  0  0  0  7  0  0  2 12  0|
| 0  0  0  0  1  0 10  0  0  8  3 13  5|
----------------------------------------
--- 23:03:52.136 jss DEBUG2:    Best genome       >>> Fitness:    0.529412; Total time:      90; Idle time:         60;
--- 23:03:52.136 jss DEBUG2:
----------------------------------------
| 0  0  6  0  0  0 11  0  0  9  4  5|
| 7  0  0  0  0  0  0  0  2  0 12  0|
| 0  0  0  0  1  0 10  0  0  8  3 13  0|
----------------------------------------
--- 23:03:52.136 jss DEBUG3: Correct plans:       7486      24%
--- 23:03:52.137 jss DEBUG3: Incorrect plans:    22554      75%
--- 23:03:52.137 jss DEBUG3: solutions:              5
--- 23:03:52.137 jss DEBUG3: ************************************************************************************************************************************************
--- 23:03:52.137 jss INFO  : Plan was found for task:
--- 23:03:52.137 jss DEBUG2: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.137 jss DEBUG2: TASK:
--- 23:03:52.137 jss DEBUG2:                  jobs: Projekt1 Projekt2 Projekt3
--- 23:03:52.137 jss DEBUG2:            operations: 13
--- 23:03:52.137 jss DEBUG2:              machines: Karel Michal Tomas
--- 23:03:52.137 jss DEBUG2:                 plans: Projekt1:(1,2,3,4,5) Projekt2:(7,8,9) Projekt3:(10,11,12,13)
--- 23:03:52.137 jss DEBUG2:           utilization: Karel:{3,4,5,6,8,9,11,12} Michal:{2,7,12} Tomas:{1,3,4,5,8,10,13}
--- 23:03:52.137 jss DEBUG2:              duration: 1:6-9 10:9-9 11:16-18 12:23-25 13:3-5 2:8-8 3:2-3 4:18-22 5:24-27 6:20-21 7:7-8 8:2-3 9:11-12
--- 23:03:52.137 jss DEBUG2: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.138 jss INFO  : Transforming plan to Petri Net...
--- 23:03:52.139 jss DEBUG2: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.140 jss DEBUG2: SIM MODEL MIN:
--- 23:03:52.140 jss DEBUG3:        Data:
                                    ----------------------------------------
                                    | 0  0  0  0  0  0 11  0  0  9  4  6|
                                    | 0  0  0  0  0  0  7  0  0  2 12  0|
                                    | 0  0  0  0 10  0  1  0  0  8  3 13  5|
                                    ----------------------------------------
--- 23:03:52.140 jss DEBUG3:       Model:
--- 23:03:52.140 jss DEBUG3:               Machine 1:   11|< 9,25)|   9|<25,36)|   4|<36,54)|   6|<54,74)|
--- 23:03:52.140 jss DEBUG3:               Machine 2:    7|< 0, 7)|   2|<15,23)|  12|<25,48)|
--- 23:03:52.140 jss DEBUG3:               Machine 3:   10|< 0, 9)|   1|< 9,15)|   8|<15,17)|   3|<23,25)|  13|<48,51)|   5|<54,78)|
--- 23:03:52.140 jss DEBUG3:  Total time:   78
--- 23:03:52.140 jss DEBUG3:   Idle time:   42
--- 23:03:52.140 jss DEBUG2: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.141 jss DEBUG2: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.141 jss DEBUG2: SIM MODEL MAX:
--- 23:03:52.142 jss DEBUG3:        Data:
                                    ----------------------------------------
                                    | 0  0  0  0  0  0 11  0  0  9  4  6|
                                    | 0  0  0  0  0  0  7  0  0  2 12  0|
                                    | 0  0  0  0 10  0  1  0  0  8  3 13  5|
                                    ----------------------------------------
--- 23:03:52.142 jss DEBUG3:       Model:
--- 23:03:52.142 jss DEBUG3:               Machine 1:   11|< 9,25)|   9|<25,37)|   4|<37,56)|   6|<56,76)|
--- 23:03:52.142 jss DEBUG3:               Machine 2:    7|< 0, 7)|   2|<15,23)|  12|<25,50)|
--- 23:03:52.142 jss DEBUG3:               Machine 3:   10|< 0, 9)|   1|< 9,15)|   8|<15,17)|   3|<23,26)|  13|<50,54)|   5|<56,82)|
--- 23:03:52.142 jss DEBUG3:  Total time:   82
--- 23:03:52.142 jss DEBUG3:   Idle time:   42
--- 23:03:52.142 jss DEBUG2: ------------------------------------------------------------------------------------------------------------------------------------------------------
--- 23:03:52.146 jss INFO  : Saving petri net to file "plan.pnml"...
--- 23:03:52.150 jss INFO  : Ending JobShop Solver...
```

Obrázek B.1: Výstupní výpis aplikace se zobrazením výsledku hledání



Obrázek B.2: Výsledná Časová Petriho síť