VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MODERNÍ PLÁNOVACÍ ALGORITMY

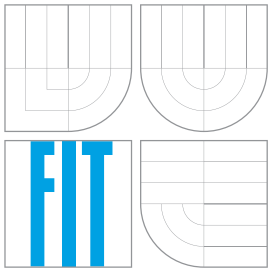
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR BINKO

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MODERNÍ PLÁNOVACÍ ALGORITMY

MODERN PLANNING ALGORITHMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR BINKO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. FRANTIŠEK ZBOŘIL, Ph.D.

BRNO 2010

Abstrakt

Tato práce popisuje algoritmy graphplan, satplan a real-time adaptive A*. Na implementaci těchto algoritmů je otestována jejich funkčnost a předpokládané vlastnosti (real-time výpočet, paralelismus), v netriviálních doménách. Graphplan a satplan jsou testovány v doménách block-world, tire-worl a bulldozer. Výsledky těchto algoritmů jsou porovnány a vykresleny do grafu. Real-time adaptive A* je testován v doméně tire-world. Dosažené výsledky jsou srovnány s klasickým A* a jsou zhodnoceny výhody a nevýhody těchto algoritmů.

Abstract

This work describes graphplan, satplan and real-time adaptive A* planning algorithms. Through implementation of these algorithms, functionality and assumed attributes (real-time calculation, parallelism) are tested. These tests take place in nontrivial domains. Graphplan and satplan algorithms were tested in block-world, tire-world and bulldozer domains. Results of these tests were compared and displayed in graphs. Real-time adaptive A* algorithm was tested in tire-world domain. Results of these tests were compared with classic A* algorithm. Advantages and disadvantages of these algorithms are also described in this work.

Klíčová slova

Graphplan, satplan, A*, real-time adaptive A*, plánování, jazyk PDDL, plánovací úlohy, plánovací grafy, relace mutex, výroková logika, program blackbox, strips.

Keywords

Graphplan, satplan, A*, real-time adaptive A*, planning, PDDL language, planning tasks, planning graphs, mutex relation, propositional logic, blackbox program, strips.

Citace

Petr Binko: Moderní plánovací algoritmy, diplomová práce, Brno, FIT VUT v Brně, 2010

Moderní plánovací algoritmy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně, pod vedením pana Ing. Františka Zbořila Ph.D.

.....
Petr Binko
26. května 2010

Poděkování

Děkuji svému vedoucímu Ing. Františku Zbořilovi, Ph.D. za odborné rady a konzultace, které mi pomohly dokončit tuto práci.

Obsah

Kapitola 1

Úvod

Plánovaní z pohledu umělé inteligence, může být chápáno jako proces rozhodování o tom, jaké akce budou provedeny. Představme si inteligentního robota. Robot je výpočetní mechanizmus, který reaguje na vstupy skrze senzory a tím pozoruje prostředí a vytváří si reprezentaci svého bezprostředního okolí. Aby byl robot užitečný, musí být schopen provádět nějaké akce. Na své okolí tedy působí skrze efektory, které mu umožňují pohyb a interakci s předměty v jeho těsné blízkosti.

S určitou abstrakcí lze říci, že robot je mechanizmus, který mapuje svoje pozorování, které získal skrze senzory, na akce, které jsou vykonány efektory. Plánování je proces rozhodování, potřebný k tomu, aby byla vytvořena nějaká sekvence akcí na základě, nějaké sekvence pozorování. Čím více je komplikované prostředí a úkoly, které má robot vykonat, tím více musí být robot inteligentní. Aby se dalo mluvit o opravdové inteligenci, musí být robot schopen plánovat své akce i v prostředích, které se svou složitostí blíží nebo rovnají reálnému prostředí.

Takový robot zatím neexistuje. Většina robotů v dnešní době, provádí úkony, ke kterým není zapotřebí opravdové inteligence, jako transport předmětů z jednoho místa na druhé v prostředích, které jsou předvídatelné a předem známé. Největším problémem při vykonávání úkonů v prostředích, které nejsou předem známé, je v současné době spolehlivá reprezentace dat ze senzorů a ovládání základních pohybových prvků robota. Dříve než bude tento problém vyřešen, nemá smysl dávat robotům složité úlohy. Až bude tento problém vyřešen, budou potřeba silné techniky reprezentace a plánování aby byla úroveň inteligence robotů dostatečně vysoká.

Překážky pro vytvoření opravdu inteligentní bytosti jsou těsně spjaty s vnímáním a reprezentací znalostí týkajících se světa. Reálný svět je velmi komplikovaný jak ve své fyzice a geometrii tak i ve svých sociálních aspektech. Reprezentace těchto všech znalostí, pomocí logických a symbolických prostředků, kterých se v umělé inteligenci používá, by mohla být nedostačující a komplikovaná. Tento problém zůstává předmětem mnoha vědeckých debat.

Plánování je v současné době použitelné pouze v omezených prostředích, ve kterých je snadné identifikovat atomická fakta a popsat chování celého prostředí. Tyto předpoklady nejlépe splňují systémy, které jsou celé vytvořeny člověkem nebo systémy na které je možné nahlížet s potřebnou mírou abstrakce.

Cílem této práce, je vysvětlení principů, v současné době nejzajímavějších, plánovacích algoritmů. Dále, na jejich implementaci vyzkoušet funkčnost a použitelnost v daných (netriviálních) doménách.

V první části práce, jsou vysvětleny základní teoretické vlastnosti algoritmů graphplan, satplan a real-time adaptive A*. V druhé části jsou testovány jednotlivé implementace

těchto algoritmů. Real-time adaptive A* je testován v doméně tile-world. Výkon algoritmu je porovnáván s klasickým A* a to po stránce délky vyhledáváni, ceny nalezené cesty a počtu dosažených cílů v daném časovém limitu.

Graphplan je nejdříve testován na vytvořené implementaci v prostředí block-world s více rameny. Poté je graphplan společně se satplanem testován v programu blackbox a výsledky obou algoritmů jsou porovnány grafem.

Konec práce obsahuje zamyšlení nad vývojem a trendy plánovacích algoritmů. Celkový přínos práce je zhodnocen v závěru.

Kapitola 2

Historie a vývoj plánování

Kořeny plánování leží částečně v řešení problémů skrze prohledávání stavového prostoru a techniky tomu podobné jako třeba redukce problému na podproblémy. Takovými úlohami se již zabývali vědci v 70-tých letech minulého století. Plánování bylo již od svých počátků hnáno dopředu, hlavně potřebami robotiky. První významný plánovací systém byl STRIPS (Fikes a Nilsson, 1971), který ilustroval interakci mezi těmito hlavními vlivy. STRIPS byl navržen jako plánovací komponenta softwaru pro projekt Shakey robota. V roce 1993 se Fikes a Nilsson opět zmiňují o STRIPS projektu a podávají přehled jeho možného využití v soudobých plánovacích úlohách.

Po několik let vládl v poli plánování terminologický zmatek. Někteří autoři (Genesereth and Nilsson, 1987) používali výraz lineární k vyjádření toho, čemu my říkáme úplně uspořádaný a nelineární pro částečně uspořádaný. Sacerdoti (1975) používal výraz lineární ve spojení s vlastností, kterou nazýváme neprokládaný. S danou množinou podcílů, může neprokládaný plánovač najít plány k řešení každého podcíle, ale tyto plány může kombinovat pouze tak, že všechny kroky jednoho plánu vloží před, nebo za kroky plánu jiného. Mnoho prvotních plánovačů 70-tých let byly neprokládané a tudíž neúplné, což znamená, že nenašli vždy řešení, pokud existovalo. Toto bylo umocněno sussmanovou anomalií nelezené během experimentů se systémem HACKER. Ten představil myšlenku chránění podcílů a byl prvotním příkladem učení plánu.

Plánování pomocí regrese, ve které jsou kroky úplně uspořádaného plánu uspořádány tak, aby se zamezilo konfliktům mezi podcíly bylo představeno Waldingerem (1975) a také použito Warrenovým (1974) WARPLANEM. WARPLAN stojí za povšimnutí také kvůli tomu, že to byl první plánovač napsaný v logickém jazyce prolog, a je jedním z nejlepších příkladů obdivuhodné úspory které lze někdy dosáhnout použitím logického programování (WARPLAN má pouze okolo 100 řádků, což je pouze zlomek velikosti, které dosahovali soudobé plánovače). INTERPLAN (Tate, 1975) také povoloval libovolné prokládání kroků plánu tak aby se obešla Sussmanova anomálie a jí podobné problémy.

Vytvoření částečně uspořádaných plánu, (později nazývané síť úkolů - task network), mělo raženou cestu plánovačem NOAH (Sacerdoti, 1975, 1977) a bylo důkladně prozkoumáno v systému NONLIN (Tate, 1977). NONLIN byl také první plánovač, který využíval explicitní algoritmus pro rozhodovaní splnitelnosti či nesplnitelnosti podmínek v různých částech částečně specifikovaného plánu.

TWEAK (Chapman, 1987) formalizuje generický, částečně uspořádaný plánovací systém. Chapman poskytl detailní analýzu, včetně důkazů úplnosti různých formulací plánovacího problému a problémů s ním spjatých.

V roce 1991 McAllister a Rosenblitt navrhli plánovač, který byl implementován algo-

ritmem SNLP (Soderland a Weld, 1991). Na algoritmu SNLP byl založen další algoritmus POP (partial order planner)[6].

Mezi nejnovější algortmy patří GRAPHPLAN (Avrim Blum a Merrick Furst, 1995), který využívá nové metody plánovacích grafů, aby snížil prostorovou náročnost algoritmu. Dalším algoritmem je SATPLAN (vyvýjený 1997-2003), který převádí problém plánování na SAT problem.

Kapitola 3

Graphplan

Je algoritmu pro automatizované plánování vyvinutý v roce 1995 (Avrim Blum, Merric Furst) [3]. Graphplan má na vstupu plánovací problém, který je zapsaný ve STRIPS a na výstupu sekvenci operací, které po vykonání vedou do koncového stavu. Jméno graphplan vzniklo díky použití nové techniky zvané plánovací graf (planning graph), která redukuje množství prostoru, který je nutné prohledat v grafu stavového prostoru. V takovém grafu jsou uzly možné stavy a hrany představují dostupnost stavu pomocí určitých akcí.

3.1 Plánovací grafy

Plánovací graf se skládá ze sekvence úrovní, které odpovídají časovým krokům plánu. Každá úroveň obsahuje množinu akcí a množinu stavů, které mohou být pravdivé v daném časovém kroku [4]. To závisí na akcích, které se provedly v předchozím kroku. Takže na začátku máme základní fakty, které definují počáteční stav plánu a tyto fakta jsou rozvinuta do grafu. Akce v jednotlivých úrovních mohou být vzájemně exkluzivní, to znamená, že provedení jedné akce znemožňuje provedení jiné akce. Pro každý stav C se přidá perzistentní akce s předpokladem C a efektem C. Platnost stavu C v kroku k=1 je stejná jako platnost stavu C v kroku k+1. Tudíž se zachová stav C do dalšího kroku nezměněný.

Příklad:

$Start : Have(pizza)$
$Finish : Have(pizza) \wedge Eaten(pizza)$

$Op(ACTION : Eat(pizza),$
$PRECOND : Have(pizza),$
$EFFECT : Eaten(pizza) \wedge \neg Have(pizza))$

$Op(ACTION : Buy(pizza),$
$PRECOND : \neg Have(pizza),$
$EFFECT : Have(pizza))$

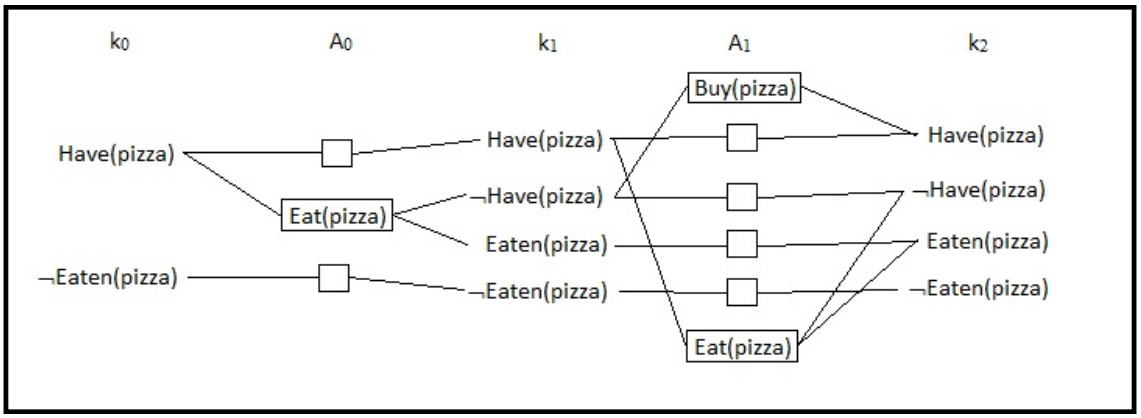
Počáteční stav je, že máme pizzu. Chceme dosáhnout cílového stavu a to tak, abychom měli pizzu a zároveň snědli pizzu. Jsou definovány dvě operace.

- 1) Sníst pizzu, která má předpoklad, že nějakou pizzu máme a výsledkem je, že ne-

máme pizzu a snědli jsme pizzu.

- 2) Koupit pizzu, která má předpoklad, že nemáme pizzu a výsledkem je, že máme pizzu.

Nyní si ukážeme, jak by vypadal graf pro tento příklad. Začneme v kroku k_0. V tomto počátečním stavu máme následující dva fakty: Have(pizza),¬Eaten(pizza). K přechodu do dalšího kroku, musíme provést nějakou akci A_0. Podle předpokladů akcí je možné provést akci Eat(pizza) plus perzistentní akce, které nezmění fakta. Z ¬Eaten(pizza) nevede, kromě perzistentní akce, žádná jiná, protože ¬Eaten(pizza) není předpoklad pro žádnou akci.



Obrázek 3.1: Příklad plánovacího grafu.

Vidíme, že A_0 znázorňuje všechny akce, které lze v kroku k_0 vykonat (Buy(pizza) nemůžeme provést, protože má předpoklad ¬Have(pizza)). V k_1 jsou všechny stavy, do kterých se můžeme dostat avšak je jasné, že nemůžeme dosáhnout všech stavů zároveň. Provedení jedné akce znemožňuje provedení druhé. Například pokud provedeme akci Eat(pizza), která má za výsledek ¬Have(pizza), nemůžeme provést perzistentní akci z Have(pizza), protože ta má za následek have(pizza) což je konflikt. Zároveň také výsledek akce Eat(pizza) neguje předpoklad pro perzistentní akci z Have(pizza). Takže v plánovacích grafech se udržuje informace o tom, které akce a stavy jsou vzájemně exkluzivní.

V dalším kroku příkladu již můžeme použít akci Buy(pizza), protože máme předpoklad ¬Have(pizza). V A_1 jsou nyní všechny akce, které lze použít k přechodu do kroku S_2. Nyní už je zřejmé, co je cílem plánovacích grafů. Vidíme, že k dosažení chtěného cíle, tudíž Have(pizza) ∧ Eaten(pizza) je zapotřebí, aby tyto dva stavy nebyly vzájemně exkluzivní. Takže je pochopitelné, že cíle nelze v tomto příkladě dosáhnout jednou akcí, protože v kroku k_1 jsou tyto dva stavy v konfliktu. V kroku k_2 však již ne, tudíž provedení akce Eat(pizza) v prvním kroku a provedení akce Buy(pizza) v druhém kroku vede ke stavu Have(pizza), který již není vzájemně exkluzivní se stavem Eaten(Pizza). Nyní je také vidět, proč potřebujeme perzistentní akce, protože Eaten(pizza) by v kroku k_2 nešlo získat tak, aby nebylo v konfliktu s Have(pizza). Tudíž perzistentní akce říká: zachovaly jsme si eaten(pizza) z k_1 do k_2. Dosažení takového kroku, kde budou všechny stavy, definující cílový stav, nekonfliktní je nutná podmínka pro nalezení plánu, ale není dostačující. Dále je také důležité, že pokud graf dosáhne v nějakém kroku k toho, že dva stavy nejsou vzájemně konfliktní, tyto stavy poté zůstávají nekonfliktní ve všech krocích větších než k, což je pochopitelné, protože víme, že existují perzistentní akce. Zároveň také, pokud jsou nějaké stavy v kroku

k v konfliktu, existuje možnost, že se stanou nekonfliktní v některém z kroků větších než k. Z toho plyne, že pokud udržujeme nějakou množinu konfliktních stavů, tak se s rostoucím krokem může pouze zmenšovat, nikoliv narůstat. Z toho vyplívá, že hloubka grafu je konečná. Pokud dojdeme na konec grafu a požadované cílové stavy jsou stále konfliktní, můžeme tvrdit, že k dosažení cíle neexistuje plán.

3.2 Mutex akce a stavy

Akce, které jsou v konfliktu, se nazývají mutex akce. Relace mutex je mezi dvěmi akcemi pokud:

- 1) Výsledek jedné akce, neguje výsledek akce druhé.
- 2) Jeden z výsledků první akce je negací předpokladu druhé akce.
- 3) Jeden z předpokladů první akce je vzájemně exkluzivní s předpokladem akce druhé.

Stavy, které jsou v konfliktu, se nazývají mutex stavy. Relace mutex je mezi dvěma stavy pokud:

- 1) První stav neguje druhý.
- 2) Oba stavy jsou dosaženy akcemi, které jsou v relaci mutex (nekonzistentní podpora).

3.3 Hlavní myšlenka graphplanu

Pseudokód:

```
function graphPlan(problem)
     graph <- planning-graph(problem)
     goals <- goal[problem]
     do
if goal nejsou v˜relaci mutex v˜poslední aktuální úrovni grafu
          then do
               solution <- Extract-solution(graph)
               if solution != failure then return solution
               else if no-solution-possible(graph)
                    then return failure
          graf <- expand-graph(graph, problem)
```

Máme vytvořený plánovací graf a zadané cílové stavy (graph, goals). Graphplan algoritmus po každém rozšíření grafu o další úroveň zkontroluje, jestli jsou cíle v relaci mutex. Pokud nejsou, existuje možnost, že lze najít plán. Algoritmus se tedy pokusí extrahovat plán a to tak, že se snaží jít zpětně z dosažených cílů a pokouší se najít množinu na sobě nezávislých akcí, které povedou k cíly [4]. Pokud uspěje, vrátí plán, pokud ne, plán neexistuje a algoritmus vrací failure. Pokud cíle byly v relaci mutex, rozšíří graf o další úroveň a cyklus se opakuje.

3.4 Meze graphplanu

- 1) Počet stavů monotónně vzrůstá. Jakmile se stav objeví v grafu, zůstává zde napořád. Nové akce mohou přinést nové stavy, ale to nemůže pokračovat do nekonečna, protože množina stavů je konečná.
- 2) Počet akcí monotónně narůstá. Některé akce, které nešlo použít, protože množina stavů neobsahovala potřebné předpoklady, se mohou stát použitelnými s příchodem nových stavů. Ale opět je množina akcí konečná tudíž počet akcí nemůže růst do nekonečna.
- 3) Počet mutexů monotónně klesá. Něco co není v relaci mutex se nemůže stát mutexem.

Tyto tři vlastnosti zaručují, že se graphplan ukončí.

Kapitola 4

Real time adaptive A*

Agenti pohybující se v reálném čase, například postavy v počítačových hrách, se musejí pohybovat plynule, a tudíž je zapotřebí vyhledávání v reálném čase [5]. Real-time heuristiky vyhledávacích metod, nacházejí pouze začátek trajektorie z aktuálního stavu do cílového stavu. Omezují svoje vyhledávání na malou část stavového prostoru, která je dosažitelná z aktuálního stavu pomocí malého počtu vykonaných akcí. Takový prostor se nazývá lokální prohledávaný prostor - local search space. Agenti si vymezí lokální prohledávaný prostor, prohledají ho, rozhodnou se jak se v něm pohybovat a provedou jednu nebo více akcí po výsledné trajektorii. Agent opakuje tento proces, dokud nedosáhne cílového stavu. Real-time vyhledávací heuristika tudíž neplánuje celou cestu do cílového stavu, což často vede ke kratší době vyhledávání, ale také k větší ceně cesty. Co je však nejdůležitější, tak real-time vyhledávací heuristiky mohou splnit těžké realtimové požadavky ve velkých stavových prostorech, protože velikost jejich lokálních prohledávaných prostorů (jejich rozhledů - look-aheads) jsou nezávislé na velikosti stavového prostoru a proto mohou zůstat malé. Aby se vyhledávání směřovalo k cíly a aby nedocházelo k cyklení, heuristiky se sjednocují se stavy a aktualizují se mezi jednotlivými vyhledáváními, což si vezme velký kus vyhledávacího času v každé epizodě. Algoritmus RTAA* je contract anytime metoda, může si tedy vybrat svůj lokální prohledávaný prostor, aktualizovat heuristiky všech stavů uvnitř lokálního prohledávaného prostoru a to velmi rychle. Tato rychlost aktualizace heuristik společně s většími prohledávanými prostory, překompenzuje fakt, že metoda RTAA* využívá trochu méně informovanou heuristiku.

4.1 Hlavní myšlenka

Hlavní myšlenka je jednoduchá ale efektivní [5]. Předpokládejme, že je potřeba provést několik vyhledávání algoritmem A* s konzistentní heuristikou ve stejném stavovém prostoru a se stejnými cílovými stavy a s odlišnými počátečními stavy. Adaptive A* dělá heuristiku více informovanou po každém vyhledávání, aby se urychlili budoucí vyhledávání.

A* je algoritmus využívaný k hledání cesty s minimální cenou, ve stavovém prostoru nebo v grafech [9]. Pro každý stav s, uživatel dodá heuristiku $h(s)$, která odhaduje vzdálenost cíle v daném stavu (tedy cenu cesty z tohoto stavu k cíly). Heuristika musí být konzistentní. Pro každý stav potkaný během hledání, A* udržuje dvě hodnoty: nejmenší cenu $g[s]$ doposud nalezené cesty z počátečního stavu s_{curr} do stavu s (který je zpočátku nekonečno) a odhad vzdálenosti $f[s] = g[s] + h[s]$ z počátečního stavu s_{curr} přes stav s do cílového stavu. A* poté pracuje následovně: udržuje prioritní frontu nazývanou open, která

z inicializace obsahuje pouze stav s_{curr}. A* vybere stav s nejmenší hodnotou f z prioritní fronty, pokud je s cílový stav, tak algoritmus končí. Jinak expanduje stav, to znamená, že aktualizuje hodnotu g každého následníka stavu s a poté, tyto následovníky vloží do fronty open. Poté proces opakuje. Po skončení je hodnota g každého expandovaného stavu s rovna vzdálenosti z počátečního stavu s_{curr} do stavu s.

Nyní vysvětlíme, jak se může heuristika upravit tak, aby byla více informovaná po každém hledání, čímž by se urychlila budoucí hledání. Předpokládejme, že s je stav, který byl expandován během vyhledávání algoritmem A*. Můžeme získat přijatelný odhad (nenadhodnocený) jeho vzdálenosti k cíly $gd[s]$ následovně: vzdálenost z počátečního stavu s_{curr} do libovolného cílového stavu přes stav s je rovna vzdálenosti z počátečního stavu s_{curr} do stavu s plus cílová vzdálenost $gd[s]$ stavu s. Očividně vzdálenost nemůže být menší než $gd[s_{curr}]$. Proto není cílová vzdálenost $gd[s]$ stavu s menší než vzdálenost $gd[s_{curr}]$ (hodnota $f[s']$ cílového stavu s', který měl být expandován, když byla A* ukončena) mínus vzdálenost z počátečního stavu s_{curr} do stavu s (g hodnota $g[s]$ stavu s když je A* ukončena).

$$g[s] + gd[s] \geq gd[s_{curr}]$$
$$gd[s] \geq gd[s_{curr}] - g[s]$$
$$gd[s] \geq f[s'] - g[s]$$

Následovně, $f[s'] - g[s]$ poskytuje případný odhad vzdálenosti k cíly $gd[s]$ stavu s a může být vypočítána velmi rychle. Více informované heuristiky mohou být získaný výpočtem a přiřazením tohoto rozdílu každému stavu, který byl expandován během vyhledávání algoritmem A* a tudíž je v seznamu closed, když A* skončí (stavy v open nejsou aktualizovány, protože vzdálenost z počátečního stavu do těchto stavů může být menší než jejich g hodnota, když A* skončí). Těchto výpočtů bylo využito k vývoji nového real-time vyhledávání s heuristikou, nazývaného real-time adaptive A* (RTAA*).

4.2 Pseudokód a princip algoritmu

Procedura: $Real - time - adaptive - astar()$ //

```
[1] while (scurr != GOAL) do
[2]  lookahead = libovolně zvoleny integer > 0
[3]   astar();
[4]  if s' = FAILURE then
[5]    return FAILURE
[6]  for all s~in CLOSED do
[7]    h[s] = g[s'] + h[s'] - g[s]
[8]  movements = libovolný integer > 0
[9]  while (scurr != s' AND movements > 0) do
[10]  a = akce z~A(s) na minimální trajektorii z~scurr do s'
[11]  scurr = succ(scurr,a);
[12]  movements++;
[13]  libovolně krát do
[14]  inkrementuj libovolný c[s,a] kde s~in S~a a in A(s)
[15]  pokud některá inkrementovaná c[s,a] je na trajektorii    minimální cenou
z~scurr do s'
[16] then break;
[17]  return SUCCESS;
```

Proměnné označené [uživatel] musejí být inicializovány předtím než je RTAA* zavolána. S_{curr} musí být nastaven na počáteční stav agenta, c na počáteční cenu akce a h na počáteční heuristiku, která musí být konzistentní pro počáteční cenu c, tudíž musí splňovat $h[s] = 0$ pro všechny cílové stavy s a $h[s] \leq h[succ(s, a)] + c[s, a]$ pro všechny necílové stavy s a akce a, které v nich mohou být vykonány.

Proměnné označené [A*] jsou aktualizovány během volání astar() [3], který provádí (dopředné) vyhledávání A* s aktuální heuristikou aktuálního stavu, směrem k cílovému stavu, dokud jej nedosáhne nebo lookahead stavy nejsou všechny expandovány. Po tomto vyhledávání potřebujeme, aby stav s' byl ten stav, který se měl expandovat než A* skončil. Potřebujeme, aby stav s' byl roven FAILURE pokud vyhledávání skončilo z důvodu prázdného seznamu open. V tomto případě neexistuje cesta s konečnou cenou z aktuálního stavu do libovolného cílového stavu a RTAA* v tomto případě vrací FAILURE [5]. Dále potřebujeme, aby seznam CLOSED obsahoval stavy, expandované během A* vyhledávání a aby hodnoty $g[s]$ byly definovány pro všechny generované stavy s, včetně všech expandovaných stavů. Definujeme hodnoty $f[s] = g[s] + h[s]$ pro stavy s. Expandované stavy z lokálního prohledávaného prostoru aktualizují svoje heuristiky pomocí funkce:

$$h[s] = f[s'] - g[s] = g[s'] + h[s'] - g[s] \text{ [6-7]}.$$

Heuristiky ostatních stavů zůstávají nezměněny. RTAA* pote provede akce podél trajektorie nalezené pomocí A* dokud nedorazí do stavu s' (nebo ekvivalentně nedosáhne stavu, který nebyl expandovaný nebo lokální prohledávaný prostor nedosáhne svých hranic), všechny akce pohybu byly vykonány (movements ≤ 0) nebo cena akce na trajektorii vzroste [9-16]. Poté se proces opakuje, dokud není dosaženo cílového stavu. V tom případě procedura vrací success [17].

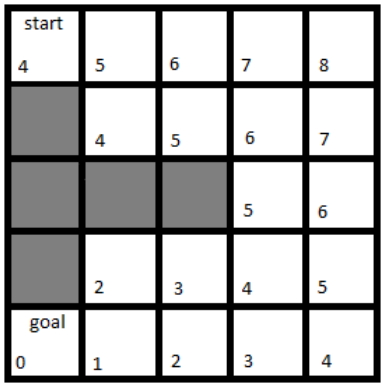
Hodnoty proměnných lookahead a movements určuje chování celého algoritmu. Například RTAA* provede jedno hledání A* z počátečního stavu do koncového a poté přemístí agenta po nalezené trajektorii, pokud si vždy zvolí za lookahead a movements nekonečno a cena akcí nevzrůstá.

4.3 Aplikace RTAA*

Real-time vyhledávací metody s heuristikou jsou často používány jako alternativy k tradičním vyhledávacím metodám. Avšak RTAA* používáme k vyhledávání cesty k cíly v neznámem terénu, v reálném čase. Například postavy v počítačových hrách většinou neznají terén, ale automaticky ho pozorují v nějakém daném rozsahu (lookahead) a pamatují si objevený terén pro budoucí využití. Aby bylo snadné tyto agenty ovládat, uživatel může kliknout kamkoliv do terénu (známého/neznámého) a agent se na danou pozici autonomně přesune. Pokud agent během pohybu zjistí, že jeho aktuální trajektorie je blokována překážkou, musí vytvořit nový plán. Jednotlivá vyhledávání musejí být rychlá, aby se agenti pohybovali plynule i na slabších procesorech. Navíc ostatní části počítačové hry, taktéž využívají procesor a počet agentů, kteří potřebují opakovaně prohledávat prostor, může být vyšší. Proto existuje časový limit pro každé jednotlivé vyhledávání A*. Abychom mohli algoritmus aplikovat, diskretizujeme terén do buněk, které jsou nebo nejsou zablokované, což je běžná praxe v kontextu s počítačovými hrami. Agenti zpočátku nevědí, které buňky jsou blokované a které ne a využívají následující strategii z robotiky: předpokládají, že buňky nejsou zablokované, dokud nenarazí na buňku, která je zablokovaná (objev překážky). Vždy vědí, na které (nezablokované) buňce se nacházejí. Pozorují stav buněk ve svém čtyř

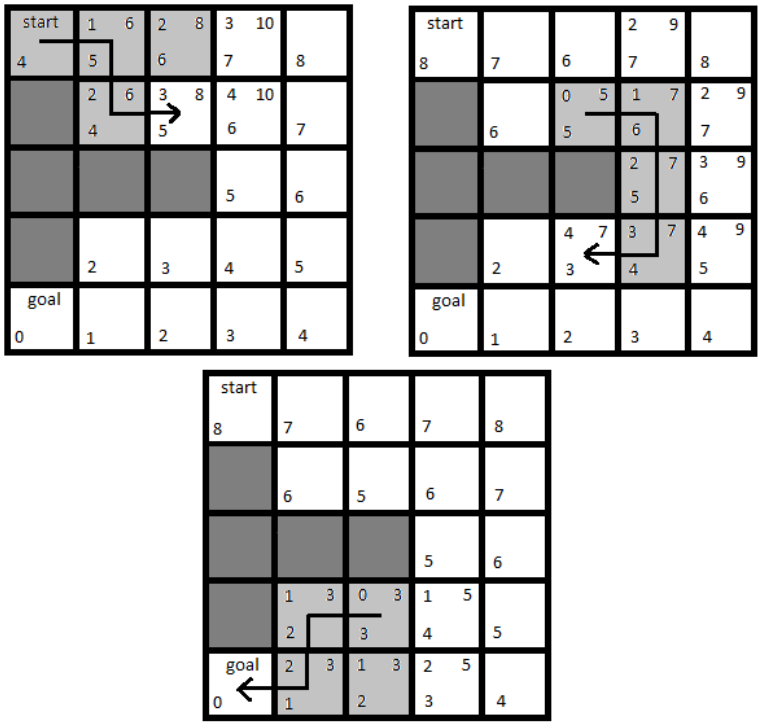
okolí a zvednou cenu akce, jejímž provedením by se dostaly na blokovanou buňku, v rozmezí od jedné do nekonečna a potom se přesunou na libovolnou buňku (nezablokovanou) s cenou jedna. Proto používáme manhattonské vzdálenosti, které konzistentní heuristika odhaduje ze vzdálenosti k cíly. Úkolem agenta je přejít do daného cílového stavu, o kterém předpokládáme, že je dosažitelný.

4.4 Ilustrace



Obrázek 4.1: Příklad scény 5x5 na hledání cesty

Obrázek zobrazuje jednoduchou navigaci od startu k cíly v neznámém terénu, který použijeme k demonstraci chování RTAA*. Šedé buňky jsou blokované. Všechny buňky mají svou počáteční heuristiku v levém dolním rohu. Nové vyhledávání (nová epizoda) začne, když cena akce, na aktuální trajektorii vzroste a přeruší spojení mezi buňkami se stejnou hodnotou f v prospěch buněk s větší hodnotou g a zbývající spojení v následujícím pořadí od nejvyšší po nejnižší prioritu: vpravo, dolů, vlevo a nahoru (systematické rušení spojení je snadnější k demonstraci než náhodné). Černý puntík označuje agenta a čáry značí trajektorii. Buňky, které agent označil, jako blokované jsou tmavě šedé. Generované buňky mají svojí g-hodnotu v horním levém rohu a f-hodnotu v pravém horním rohu. Expandované buňky jsou světle šedé a mají svou aktualizovanou heuristiku v pravém dolním rohu, což usnadňuje pochopení příkladu, protože na první pohled, je lze porovnat s heuristikou před aktualizací.



Obrázek 4.2: Postup řešení problému hledání cesty, pomocí RTAA* s lookahead = 4

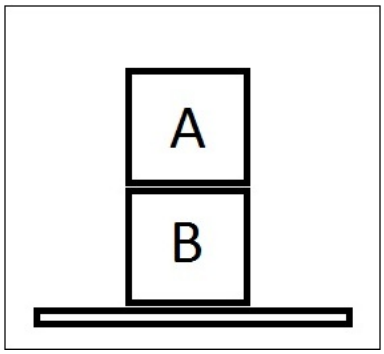
Kapitola 5

Satplan

Satplan je metoda automatizovaného plánování. Převádí plánovací problém na SAT problém, který je poté řešen pomocí metod pro stanovení splnitelnosti jako je DPLL algoritmus nebo WalkSAT. Splnitelnost je problém rozhodnutí, zda dané proměnné boolovské formule, mohou být přiřazeny takovým způsobem, aby byl výsledek formule TRUE. Zároveň je důležité rozhodnout, zda takové přiřazení neexistuje, což by implikovalo, že funkce vyjádřena danou formulí je FALSE pro všechny možné přiřazení. V tomto případě bychom řekli, že funkce je nesplnitelná, jinak je splnitelná. Aby se zdůraznila binární povaha tohoto problému, často se nazývá jako boolovská nebo výroková splnitelnost (boolean, propositional satisfiability) zkráceně SAT.

5.1 Princip plánování s výrokovou logikou

Uveďme si příklad z prostředí block world. To je klasická úloha plánovaní, kdy na stole máme několik kostek, které buď leží na sobě, nebo přímo na desce stolu. Cílem je přerovnat kostky do daného rozložení a to tak, že nemůžeme přemisťovat více kostek zároveň a můžeme požít jen povolené akce (položit kostku na stůl, položit kostku na jinou kostku). Mějme dvě kostky v následujícím rozložení [5.1]:



Obrázek 5.1: - Příklad scény z domény block-world

Počáteční stav bude definován takto:

$$Ontable(B)^0 \wedge On(B, A)^0$$

Aby byla definice pro výrokovou logiku úplná a dala se použít, je nutné explicitně přidat negace daných formulí a tím vznikne následující formule, která přesně popisuje počáteční stav.

$$\text{Počáteční stav: } Ontable(B)^0 \wedge On(B,A)^0 \wedge \neg Ontable(A)^0 \wedge \neg On(A,B)^0$$

Vycházíme z toho, že když je B na stole, tak nemůže být na A a ekvivalentně, když je A na B, nemůže být na stole. Jako další krok, se vytvoří množina následníků, což není nic jiného než reakce na různé akce.

$$\text{Axiomy následníka: } On(A,B)^1 <=> \\ (Ontable(B)^0 \wedge Ontable(A)^0 \wedge Move(B,A)^0) \vee (On(A,B)^0 \wedge \neg Move(B,Table)^0)$$

Tyto axiomy nám vlastně říkají, jaký musel být stav a provedená akce v kroku k-1, abychom dosáhli požadovaného stavu v kroku k.

Po nastavené počátečního stavu a nastavení axiomů následníka reprezentujeme cíl jako množinu klauzulí.

$$\text{Cíl: } OnTable(A)^1 \wedge On(A,B)^1$$

Pokud bychom chtěli zapsat, že se pokoušíme vyřešit problém plánování v kroku k=1 napsali bychom:

$$[Ontable(A)^1 \wedge On(A,B)^1 \wedge \text{ počáteční stav } \wedge \text{ axiomy následníka }] -> SAT?$$

Což znamená, že pokud je celá formule splnitelná, potom existuje přiřazení hodnot axiomům tak, že jsme schopni odvodit cíl řešení. SAT nám říká, jaké se mají použít axiomy následníka, aby byl odvozený cíl. Pokud formule není splnitelná, potom neexistuje plán takový, aby dosáhl cílového stavu v prvním kroku [4].

Axiomy předpokladů akcí

Některé akce mohou mít předpoklady. Pokud například chceme přesunout B na A, předpoklad musí být, že A neleží na B.

$$Move(B,A)^0 \wedge \neg On(B,A)^0$$

Axiomy exkluzivity akcí

Tyto axiomy říkají, které akce jsou v relaci mutex. Pokud máme akce, přenes B na A a přenes B na stůl, tak se vzájemně vylučují (nemůžeme v jednom kroku přenést stejnou kostku na dvě různá místa). Zapíšeme tedy před konjunkci těchto dvou axiomů negaci, čímž jednoznačně určíme, že tyto dvě akce jsou vzájemně exkluzivní. Tudíž pro všechny dvojice akcí, které tvoří mutex, musíme přidat axiom exkluzivity.

$$\neg(Move(B,A)^0 \wedge Move(B,Table)^0)$$

Axiomy stavových omezení

Slouží například k vyjádření faktu, že dva bloky nemohou být na jednom a tom samém bloku.

$$\forall p,x,y,t x \neq y => \neg(On(p,x)^t \wedge On(p,y)^t)$$

Takže abychom vyřešili plánovací problém, musíme reprezentovat počáteční množinu stavů, axiomy předpokladů akcí, axiomy exkluzivity akcí a axiomy stavových omezení. Poté tento problém převést v daném kroku na SAT problém a zjistit zda je řešitelný.

5.2 Funkce satplanu

Pseudokód:

```
Function SATplan(problem,Tmax)
     for T=0 to Tmax do
          cnf, mapping <- Trans-to-SAT(problem,T)
          assignment <- SAT-Solver(cnf)
          if assignment is not NULL then
               return Extract-Solution(assignment,mapping)
return failure
```

Velký rozdíl oproti graphplanu je, že musíme zadat ukončující podmínku ve smyslu maximálního počtu kroků. K tomu slouží hodnota Tmax . Tudíž satplan na rozdíl od graphplanu nezastaví bez ukončující podmínky. V prvním kroku se problém plánování převede na SAT problém. Poté se zjišťuje, jestli existuje přiřazení (assignment), které by řešilo SAT problem. Pokud SAT-solver nalezne řešení, je vráceno, pokud ne, cyklus pokračuje dalším krokem. Pokud proběhne celý cyklus for, aniž by bylo nalezeno řešení, algoritmu vrací failure. Je důležité zmínit, že SAT-solver nevrací plán ale pouze přiřazení. Z tohoto přiřazení se musí plán extrahovat.

Kapitola 6

Real time adaptive A* implementace a výsledky

V této kapitole bude popsána implementace programu a dosažené výsledky algoritmu RTAA*.

6.1 Implementace

Program se skládá ze třech tříd. Hlavní třídou je RTAAstar, která obsahuje pouze jednu veřejnou metodu, která se nazývá RealTimeAdaptiveAstar(). Tato metoda vyžaduje čtyři parametry.

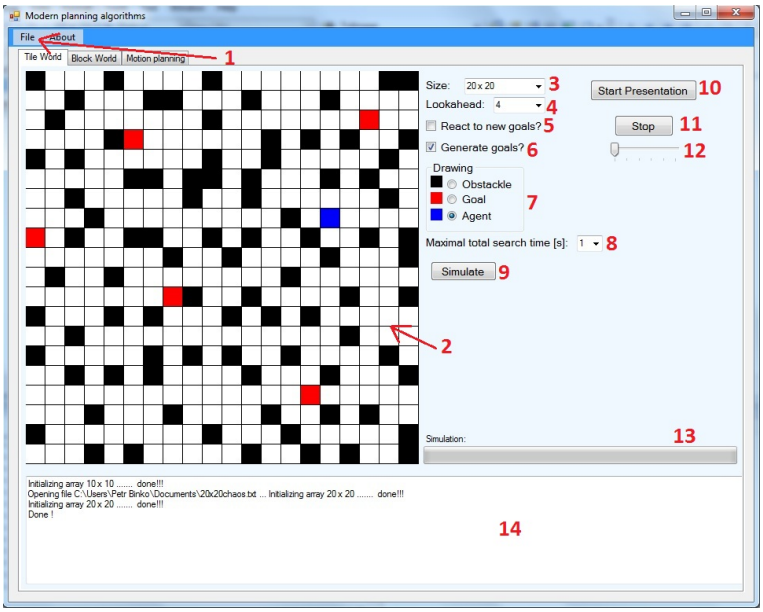
- 1. Dvourozměrné pole typu tileWorldCell. Tato struktura je nadefinována jako jedno políčko mřížky. Jedna buňka obsahuje informace o heuristice, ceně, pozici, pozici buňky, ze které se na dané políčko vkročilo a dále obsahuje informaci o tom, zda je buňka blokována, jestli je cílem nebo se na ní nachází agent.
- 2. Startovní pozici vyhledávání (pozice agenta) zadanou jako bod (Point - struktura C♯).
- 3. Cílovou pozici vyhledávání zadanou jako bod.
- 4. Hodnotu integer, udávající lookahead - rozhled se kterým ma algoritmus pracovat.

Metoda RealTimeAdaptiveAstar() poté vypočte celou cestu a vrací jí jako množinu buněk v Listu. List je speciální struktura jazyka C♯, která má vlastnosti jednorozměrného pole, ale navíc má již implementované metody jako například: add(), remove(), clear() atd. Cesta poté odpovídá posloupnosti buněk od nejnižšího indexu k nejvyššímu.

Druhou nejdůležitější třídou je TileWorldCore. Tato třída v sobě udržuje dvourozměrné pole, které odpovídá scéně. Obsahuje metody pro práci se scénou jako je přidávání a odebírání blokovaných buněk a cílů, testování regulérnosti scény (zda je v ní alespoň jeden cíl a agent), metody pro výběr cílů a náhodné přidávání nových cílů.

Poslední třídou je Form1. Ta se stará o grafický výstup a uživatelské prostředí. Obsahuje metody obsluhy událostí všech komponent prostředí, jako jsou tlačítka, combo boxy nebo progress bar. Důležitou součástí jsou metody pro simulační a prezentační mód programu a metody pro ukládání a načítání scény.

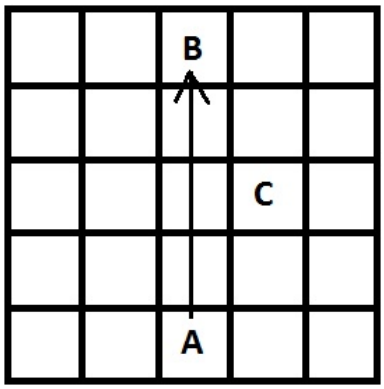
6.2 Popis uživatelského prostředí



Obrázek 6.1: Uživatelské prostředí programu

- 1. Tlačítko File - obsahuje dvě tlačítka. Save pro uložení aktuální scény a Load pro načtení vybrané scény.
- 2. Scéna - Jedná se o grafickou reprezentaci dvourozměrného pole typu tileWorldCell z třídy TileWorldCore.
- 3. Combo box, kterým lze měnit velikost (hustotu) mřížky. Možnosti jsou: 5x5, 10x10, 20x20, 50x50, 100x100.
- 4. Lookahead - nastavení rozhledu algoritmu. Možnosti jsou 1-100 a inffinity. Přičemž hodnota inffinity (odpovídá maximální hodnotě 32-bitového integeru) znamená, že se algoritmus bude chovat jako klasický A*.
- 5. Nastavení chování agenta. Nastavuje, zda má agent při své cestě k cíly reagovat na nově objevené cíle. Například pokud agent půjde z políčka A do políčka B (obrázek 6.2)a cestou se vedle něj na políčku C objeví nový cíl, agent přeruší svou cestu do B a přeplánuje s novým cílem v C. Pokud je tato vlastnost vypnuta, agent si políčka C nevšímá, dokud nedosáhne stavu B a C pak může být zvoleno jako nový cíl.

  C je zvoleno za nový cíl pochopitelně pouze tehdy, nachází-li se blíže k agentovi než stávající cíl B.
- 6. Vypnutím této funkce se zabrání tomu, aby se náhodně generovaly nové cíle. Slouží k demonstraci funkce algoritmu na speciálních případech, kdy nechceme nové cíle.
- 7. Skupina tlačítek, kterými se mění režim úpravy scény. Lze vybrat, zda se mají do scény přidat překážky, cíle nebo agent. Přidávání se provádí kliknutím na příslušné políčko ve scéně.



Obrázek 6.2: Příklad přeplánování

- 8. Lze vybrat celkový čas, který má algoritmus na vyhledávání cest. Po každém vyhledávání cesty se čas zmenšuje, a jakmile dosáhne nuly, je ukončena simulace. Slouží k tomu, abychom mohli otestovat, ke kolika cílům algoritmus stihne vypočítat cestu, v daném časovém limitu.
- 9. Tlačítko, které spouští simulaci.
- 10. Tlačítko, které spustí mód prezentace. V tomto režimu se netestuje rychlost algoritmu, ale pouze se prezentuje jeho funkčnost formou animace. Po stisknutí tohoto tlačítka se po scéně začne agent pohybovat ke svým cílům.
- 11. Zastaví prezentaci.
- 12. Slide bar udávající rychlost animace.
- 13. Progress bar znázorňuje průběh simulace.
- 14. Konzole pro výpisy programu.

6.3 Popis činnosti programu

Po spuštění programu se jako první funkce volá initializeGrid(), ze třídy Form1. Ta si z combo boxu (obrázek 6.1 - číslo 3) zjistí velikost mřížky a hodnotu gridScale. Touto hodnotou se dělí/násobí hodnoty pozice kursoru myši tak, aby se správně převedly do požadovaného měřítka. Do scény se poté vykreslí mřížka a zavolá se funkce initializeCellArray() ze třídy TileWorldCore, která provede inicializaci dvourozměrného pole, podle parametrů šířka a výška (řádky, sloupce).

Uživatel nyní musí do scény přidat agenta a alespoň jeden cíl. Může také přidat překážky jednoduše výběrem příslušného režimu (obrázek 6.1 - číslo 7) a kliknutím na zvolenou buňku mřížky. Když si uživatel nastaví scénu podle svých představ, může buď spustit režim prezentace, nebo simulace.

- 1. Prezentace se spustí tlačítkem start presentation. Tím se spustí timer (časovač), který v daném časovém intervalu volá funkci doStep(). Tato funkce zjistí, zda má agent naplánovanou cestu. Pokud ano, provede posun o jedno políčko po této cestě

a zavolá funkci didNewGoalAperred() ze třídy TileWorldCore. Tato funkce pomocí generátoru pseudonáhodných čísel (typ random v jazyce C♯) rozhodne, zda se objeví nový cíl nebo ne. Pokud ano, přidá jej na náhodnou pozici ve scéně, pokud ne funkce nic neprovede. Pokud má agent zapnuté chování, aby reagoval na nové cíle, podívá se, zda není nový cíl, pokud se objevil, blíže než stávající cíl. Pokud ano, smaže aktuální cestu a přeplánuje s nově přidaným cílem.

Pokud funkce doStep() po svém volání zjistí, že agent nemá naplánovanou cestu, zavolá funkci makeStep() ze třídy TileWorldCore, která zvolí nový cíl (implicitně nejbližší, možné pozdější rozšíření) a naplánuje cestu pomocí RealTimeAdaptiveAstar() ze třídy RTAAstar.

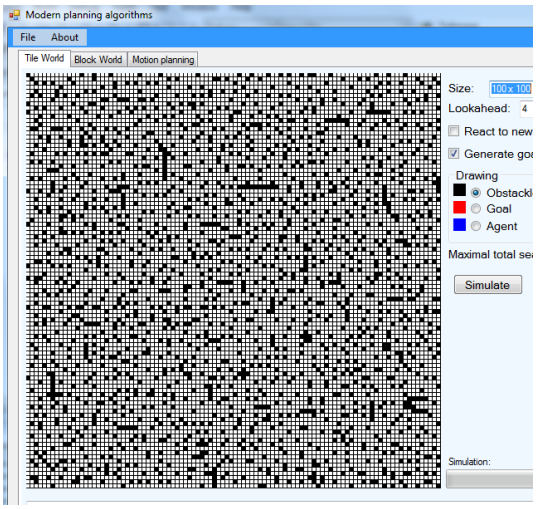
Timer volá funkci doStep() do té doby, dokud není zastaven tlačítkem stop, nebo dokud agent nedosáhne všech možných cílů.

- 2. Pokud uživatel spustí simulaci tlačítkem simulate, zavolá se funkce simulate() ze třídy Form1. Tato funkce se na rozdíl od doStep() nevolá timerem v časových intervalech, ale obsahuje cyklus while, který simuluje chod agenta tak dlouho, dokud se nevyčerpá zvolený maximální čas pro simulaci. Je důležité si uvědomit, že čas simulace 1 sekunda neznamená, že celá simulace bude trvat 1 sekundu. Tento čas je zmenšován pouze o čas samotného vyhledávání. Ostatní činnosti programu se nezapočítávají.

  Režim simulace tedy nemá žádný grafický výstup. Pouze se po dokončení do konzole vypíše počet dosažených cílů, celkový čas všech plánování a celkový počet kroků (posun z políčka na políčko), které agent provedl. Scéna je poté zresetována tak, že překážky zůstávají a všechny zbylé nedosažené cíle včetně agenta jsou odstraněny.

6.4 Experimentální výsledky

V rámci testování se porovnávaly výsledky s různým nastavením lookahead, konkrétně 4, 16, 32 a inffinity tedy A*. Testy probíhaly ve scéně 100x100 s náhodným rozmístěním překážek 6.3 a to po dobu 1 a 2 sekund.



Obrázek 6.3: Scéna, ve které probíhaly testy

	Bez reakce na nové cíle			
Lookahead	4	16	32	A*
Dosažených cílů	3516	2940	2740	2515
Počet kroků	28309	23447	21680	19293
Kroků na jeden cíl	8	7,9	7,9	7,6
	S reakcí na nové cíle			
Lookahead	4	16	32	A*
Dosažených cílů	3504	2948	2733	2460
Počet kroků	27700	23066	20900	18255
Kroků na jeden cíl	7,9	7,8	7,6	7,4

Obrázek 6.4: - Výsledky simulace, po dobu jedné sekundy

	Bez reakce na nové cíle			
Lookahead	4	16	32	A*
Dosažených cílů	6359	5392	4742	4389
Počet kroků	71228	52343	44313	37920
Kroků na jeden cíl	11,2	9,7	9,3	8,6
	S reakcí na nové cíle			
Lookahead	4	16	32	A*
Dosažených cílů	6349	5317	4730	4157
Počet kroků	69400	50240	41500	34837
Kroků na jeden cíl	10,9	9,4	8,7	8,3

Obrázek 6.5: - Výsledky simulace, po dobu dvou sekund

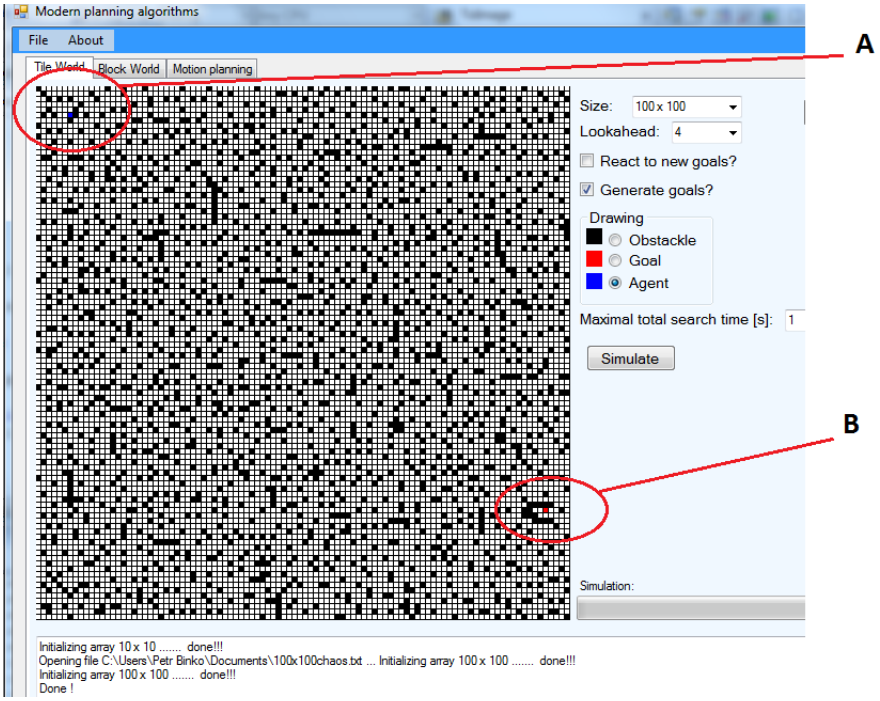
- Simulace 1 sekundu - První tabulka 6.4 udává výsledky bez reakce na nové cíle a druhá s reakcí.
- Simulace 2 sekundy - první tabulka 6.5 udává výsledky bez reakce na nové cíle a druhá s reakcí.

Z výsledků je patrné, že RTAA* v rychlosti výrazně poráží klasický A*. Pokud porovnáme výsledky s rozhledem 4 a A*, vychází, že A* dosáhne v daném časovém limitu zhruba o 30% méně cílů. Na druhou stranu délka nalezených cest je u A* optimální a hlavně v testech, které trvaly 2 sekundy je jednoznačně vidět, že RTAA* s rozhledem 4 nachází mnohem delší cesty.

Pokud se podíváme, jaký vliv má reakce na nové cíle je vidět, že na množství dosažených cílu nemá vliv žádný, ale na dvou vteřinových testech je již zřejmí vliv na délku nalezených cest. U delších simulací by byl určitě ještě výraznější.

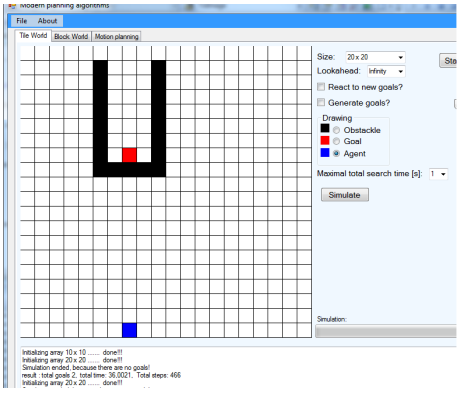
Z pohledu praktického použití například v robotech je nutné na testy nahlížet jinak. Robot s algoritmem RTAA*, by rychle počítal cestu, avšak přesun po této cestě by trval déle než v případě A*. To by však robot mezi jednotlivými vyhledáváními stál na místě a počítal cestu. Pro takovéto porovnání by však bylo nutné provést jinak stavěné testy nebo nejlépe přímo praktické testy s nějakým robotickým zařízením.

6.5 Extrémní případy



Obrázek 6.6: - Extrémní situace A*

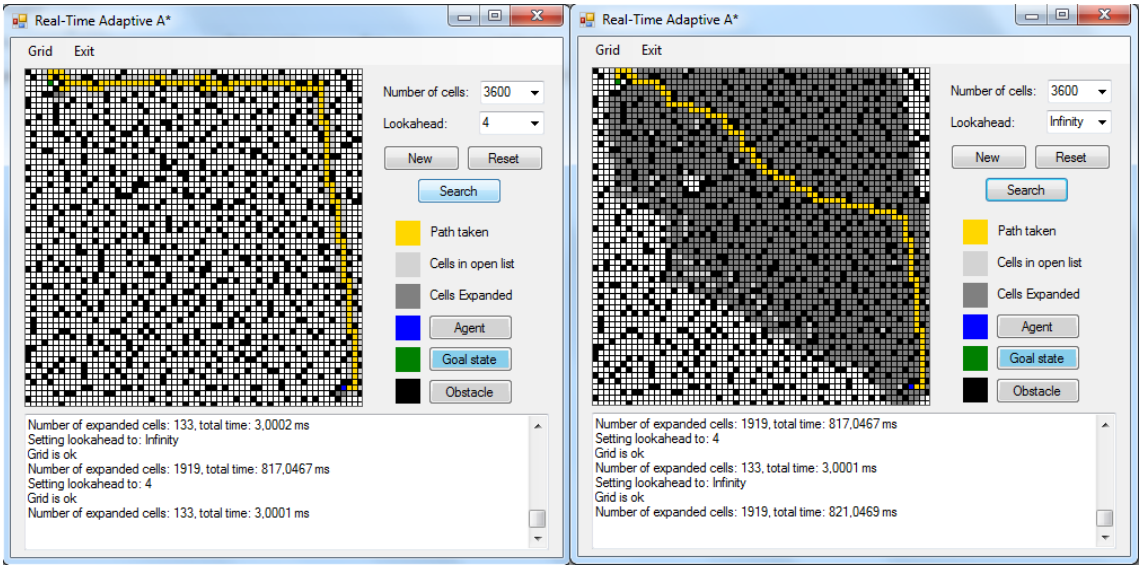
Uvažujme situaci na obrázku 6.6, kde na místě A je agent a jediný cíl je až v oblasti B. Pokud spustíme simulaci s nastavením A*, nalezení cesty k prvnímu možnému cíly bude kvůli velkému množství překážek trvat okolo 3500 milisekund, tedy simulace skončí pouze s jedním dosaženým cílem. Na rozdíl od toho pokud s naprosto stejným cílem a agentem zpustíme RTAA* s rozhledem 4, dosáhneme normálního výsledku okolo 3300 dosažených cílů.



Obrázek 6.7: - Extremní situace RTAA*

Tato skutečnost je způsobena tím, že klasický A* při hledání celé cesty najednou expanduje několikanásobně více stavů než RTAA*. Pro demonstraci využijeme první verzi

implementace RTAA*, která najde cestu k danému cíly a zobrazí expandované stavy. Na obrázku 6.8 vidíme dvě hledání algoritmem RTAA*. Nalevo je hledání s rozhledem 4 a napravo je s rozhledem nekonečno (klasický A*). Je vidět, že zatímco s rozhledem 4, trvá výpočet cesty 3 milisekundy a je expandováno 133 stavů, klasickému A* trvá výpočet 821 milisekund a expanduje 1919 stavů (expandované uzly jsou tmavě šedé).



Obrázek 6.8: Rozdíl v počtu expandovaných uzlů mezi RTAA* s rozhledem 4 a klasickým A*.

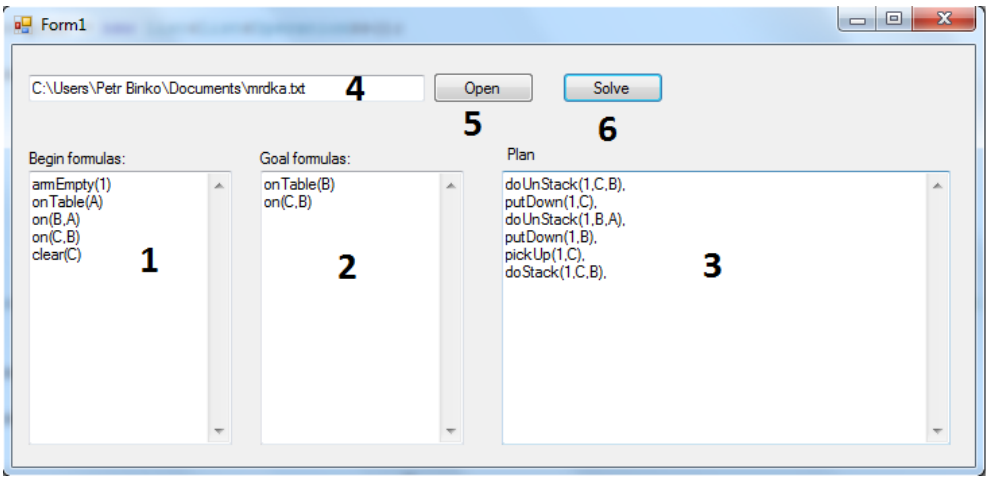
Na druhou stranu uvažujme situaci na 6.7, kde je cíl (červené pole) nepříjemně zakrytý ze třech stran překážkami. Klasický A* nalezne cestu o délce 32, což odpovídá optimální cestě. Naopak RTAA* s rozhledem 4 není schopný takto velikou překážku překonat, během jednoho dílčího vyhledávání a proto nalezne cestu o délce 466 což je extrémně více než A*. Zvýšením rozhledu na 8 však již nalezne cestu délky 180 a s rozhledem 16 délky 74 zhruba v polovičním čase než A*.

Kapitola 7

Graphplan implementace

V této kapitole je popsána implementace algoritmu graphplan, který hledá plán v prostředí block world a to se dvěmi i více rameny. Je zde popsána struktura kódu a chod programu. V závěrečných kapitolách jsou prezentovány dosažené výsledky.

7.1 Ovládání programu



Obrázek 7.1: - Uživatelské prostředí programu

- 1. Textové pole, které obsahuje formule počátečního stavu.
- 2. Textové pole, které obsahuje formule cílového stavu.
- 3. Textové pole, které po vyřešení plánovací úlohy obsahuje seznam operací, které povedou k cíly.
- 4. Cesta ke vstupnímu souboru.
- 5. Tlačítko, po jehož stisknutí uživatel vybere vstupní soubor.
- 6. Tlačítko zpustí algoritmus graphplan.

7.2 Syntax formulí, operací a vstupního souboru

Program načítá textový soubor, který obsahuje slovní zápis počátečního stavu a cílového stavu. Formule a operace mají přesně definovanou syntaxy, kterou je nezbytné dodržovat, aby algoritmus pracoval zprávně. Zároveň je nutné dodržovat malá a velká písmena, protože zápis formulí a operací je case sensitive.
Formule:

- $onTable(< block > x)$ - označuje, že blok x leží na stole.
- $clear(< block > x)$ - označuje, že na bloku x neleží žádný jiný blok.
- $on(< block > x, < block > y)$ - označuje, že na bloku y, leží blok x.
- $armEmpty(< int > arm)$ - označuje, že rameno s pořadovým číslem arm, je prázdné (nedrží žádný blok).
- $holding(< int > arm, < block > x)$ - označuje, že rameno s pořadovým číslem arm, drží blok x.

Při vytváření textového souboru, který obsahuje zadání úlohy, je nezbytné přesně dodržovat syntax formulí a formát souboru.

Nejdříve je na každém řádku zapsána jedna formule počátečního stavu. Poté následuje na samostatném řádku klíčové slovo goal a za ním opět po řádcích jednotlivé formule cílového stavu. Počet volných řádků mezi formulemi nemá na správnost zápisu vliv, avšak mezery přímo ve formuli by zapříčinili nenačtení dané formule.
Příklad zadání úlohy:

```
armEmpty(1)
onTable(A)
onTable(B)
clear(A)
clear(B)

goal
on(A,B)
```

Jednotlivé operace mají také danou syntax, avšak uživatel nikde nemusí žádné operace zadávat či zapisovat. Syntax operací se od klasických příkladů z prostředí block world, které je možné najít na internetu, liší především tím, že u každé operace musí být parametr, který udává číslo ramena, které operaci provádí. Tyto operace jsou čtyři. Každá operace má svoje počáteční podmínky (precondition - formule, které musí být přítomny a nesmějí být v relaci mutex), kladný a záporný efekt, který přidává formule, nebo negace formulí, které popisují výsledky provedení akce.

$Operace : pickUp(< int > arm, < block > x)$
$Precondition : clear(x), onTable(x), armEmpty(arm)$
$effect+ : holding(arm, x)$
$effect- : \neg clear(x), \neg onTable(x), \neg armEmpty(arm)$

$Operace : putDown(< int > arm, < block > x)$
$precondition : holding(arm, x)$
$effect+ : clear(x), onTable(x), armEmpty(arm)$
$effect- : \neg holding(arm, x)$

$Operace : doStack(< int > arm, < block > x, < block > y)$
$precondition : clear(y), holding(arm, x)$
$effect+ : armEmpty(arm), clear(x), on(x, y)$
$effect- : \neg clear(y), \neg holding(arm, x)$

$Operace : doUnStack(< int > arm, < block > x, < block > y)$
$precondition : on(x, y), clear(x), armEmpty(arm)$
$effect+ : holding(arm, x), clear(y)$
$effect- : \neg on(x, y), \neg clear(x), \neg armEmpty(arm)$

Tyto operace označují úkony: zvednutí/položení bloku na stůl, položení bloku na jiný blok a zvednutí bloku z jiného bloku. Z těchto operací je zřejmé, že prostor stolu je neomezený.

7.3 Struktury a implementace

V programu jsou navrženy čtyři základní třídy, které zpřehledňují a usnadňují implementaci algoritmu. První třídou je abstraktní třída Formula, která obsahuje proměnné:

- $< string >$ name - název formule
- $< bool >$ state - stav udává, jestli je formule pravdivá či nikoliv.
- $< List < int >>$ parent - seznam hodnot typu integer, který obsahuje indexy předchůdců dané formule, tedy indexy operací, které mají v pozitivním nebo negativním efektu danou formuli.

Tato třída má metody getParent(), setParent() a abstraktní metodu isInList(), kterou musejí ostatní třídy, které dědí ze třídy Formula přetížit. Tato metoda zjišťuje, zda se formule nachází v daném seznamu, který se metodě zasílá jako parametr. Metoda je abstraktní proto, že jednotlivé formule (on(X,Y), onTable(X), atd.) mají různé proměnné a proto test, zda jsou v nějakém seznamu probíhá u každé formule jinak.

Dalších pět tříd dědí Formula:

```
On
OnTable
Clear
ArmEmpty
Holding
```

Každá třída má, kromě zděděných, také své proměnné, které odpovídají parametrům formule. Například instance třídy On bude vytvořena následovně:

$Onformule = newOn('A', 'B, true);$

Jak je vidět, jednotlivé bloky jsou označovány hodnotami typu char, aby byly zápisy přehlednější a byly ekvivalentní s mnoha příklady, které se dají najít na internetu. Konstruktor třídy On zavolá konstruktor třídy formula (pomocí klíčového slova base), kterému zašle jméno formule a stav, tedy:

$: base('on', true);$

Dále se naplní proměnné třídy On příslušnými hodnotami char.

U ostatních tříd reprezentujících jednotlivé formule probíhá proces vytváření stejně, akorát má každá formule jiné parametry.

Tímto systémem je zaručeno, že můžeme vytvořit seznam obecného typu Formula a do něj ukládat jednotlivé formule. Přes proměnou name je pak možné zjistit o jakou konkrétní formuli se jedná a přetypováním z obecného Formula na konkrétní typ lze přistupovat k jednotlivým proměnným.

```
List<Formula> seznam;
if(seznam[i].name == 'on')
{
     ((On)seznam[i]).x = 'A';
    ((On)seznam[i].y = 'B';
}
```

Stejným způsobem je implementován obecný typ Operation, který je abstraktní třídou, která má metody:

- isExecutable - Metoda typu bool, která zjistí, zda je operace s daným seznamem operací proveditelná, tedy, jestli jsou splněny všechny její počáteční podmínky (preconditions).
- effect - metoda přidá do daného seznamu jak pozitivní tak i negativní efekt provedené akce.

Tyto metody jsou ve všech třídách, které reprezentují jednotlivé akce přetíženy. Vytváření jednotlivých operací je ekvivalentní s vytvářením formulí. Opět jsou zde třídy reprezentující jednotlivé akce:

```
PickUp
PutDown
DoStack
DoUnStack
```

Takto navržené třídy ulehčují operace typu: zjisti všechny proveditelné akce v daném kroku. Příklad vytvoření akce a testování zda je s daným seznamem formulí proveditelná.

```
PickUp op = new PickUp(1,'A')
op.isExecutable(List<Formula> seznamFormuli);
```

Další důležitou třídou je Mutex. Jedná se o jednoduchou třídu, která má pouze dvě proměnné a, b, které reprezentují indexy operací nebo formulí, které jsou v relaci mutex.

Nejdůležitější součástí třídy Mutex je metoda isInList(), která zjistí zda se dvojice nevyskytuje v daném seznamu a to bezohledu na pořadí prvků (1,3 = 3,1). Tato metoda je nezbytná, neboť se velmi často stává, že při hledání mutexů algoritmus několikrát nachází stejnou dvojici a tudíž by docházelo k duplikování a zbytečnému nárůstu množiny mutexů.

Poslední třídou je GraphStep. Ta reprezentuje jednu úroveň plánovacího grafu. Obsahuje tři seznamy:

- Seznam formulí
- Seznam operací
- Seznam mutexů

V každé vrstvě je buď naplněn seznam formulí, nebo operací, záleží na tom, jakou vrstvu GraphStep reprezentuje. Seznam mutexů je naplněn vždy, pokud existují nějaké operace nebo formule, které jsou v konfliktu. Celý plánovací graf, je tedy možné reprezentovat seznamem struktur typu GraphStep, přičemž každá formule či operace má seznam indexů (parentů), které znázorňují akce nebo operace, které k nim vedly.

Příklad:
Pokud budeme mít vstupní soubor s formulemi:

```
armEmpty(1)
onTable(A)
on(B,A)
clear(B)
```

Mějme list struktur GraphStep:

```
List<GraphStep> graf;
```

Poté graf[0] bude mít naplněný pouze seznam formulí takto:

```
[0]armEmpty(1)
[1]onTable(A)
[2]on(B,A)
[3]clear(B)
```

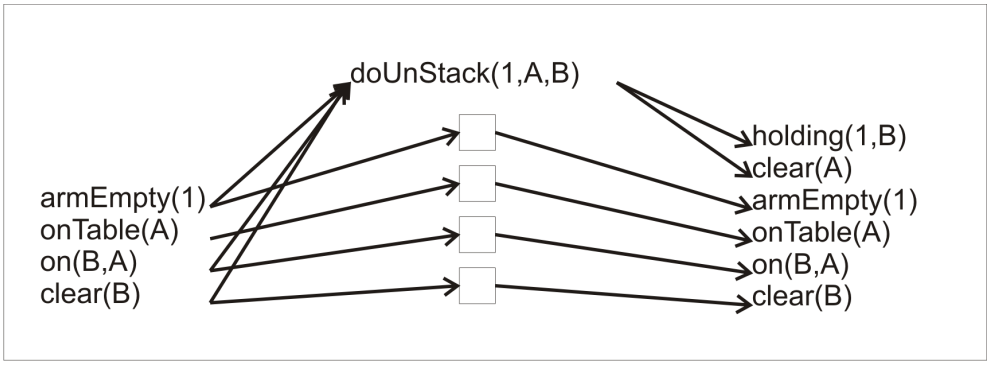
Poté se zjistí proveditelné operace v prvním kroku. Tedy graf[1] bude mít naplněn seznam operací následovně:

```
[0]doUnStack(1,B,A)[2,3,0]
[1]perzistent[0]
[2]perzistent[1]
[3]perzistent[2]
[4]perzistent[3]
```

A graf[2] bude mít opět naplněn pouze seznam formulí a to výsledkem operací z předchozího grafu.

```
[0]holding(1,B)[0]
[1]clear(A)[0]
[2]armEmpty(1)[1]
[3]onTable(A)[2]
[4]on(B,A)[3]
[5]clear(B)[4]
```

Za každou formulí/operací vidíme v hranatých závorkách obsah seznamu parent. Takto strukturovaný seznam odpovídá přesně grafu 7.2, který by měl algoritmus sestavovat. Příklady takovýchto grafů je možné najít na intarnetu.



Obrázek 7.2: - Odpovídající plánovací graf

Program obsahuje další dvě třídy, které nereprezentují struktury, ale obsahují metody, které provádí algoritmus a pracují s uživatelským prostředím.

Třída Form1, obsahuje metody uživatelského prostředí. Jelikož je uživatelské prostředí velice jednoduché jsou zde pouze metody obsluhy tlačítek a metoda na výpis nalezeného plánu.

Poslední a nejdůležitější třídou je graphplanCore. Jedná se o statickou třídu, která obsahuje všechny metody algoritmu graphplan. Třída má čtyři globální seznamy, které se naplní příslušnými metodami při načítaní vstupního souboru. Těmito seznamy jsou:

- Počáteční formule
- Cílové formule
- Názvy všech bloků
- Všechny ramena

Hlavní metoda graphPlanAlgorithm, pak tedy nepřebírá žádné parametry, pouze pracuje s těmito seznamy, najde plán a ten vrátí.

Více o třídě graphplanCore bude vysvětleno v další kapitole zabývající se chodem programu.

7.4 Popis činnosti programu

V této kapitole bude ukázáno, jak program postupuje při řešení zadané úlohy.

Po spuštění programu se objeví uživatelské prostředí. Pomocí tlačítka open, je otevřen zvolený soubor se zadanou úlohou. Do příslušných textových polí se vypíší formule počátečního a koncového stavu, aby mohl uživatel zkontrolovat, zda je vše v pořádku a nedošlo k nějaké nečekané chybě. Formule jsou zároveň naplněny do globálních seznamů třídy graphplanCore, stejně tak jako seznam všech bloků a ramen.

Po stisknutí tlačítka solve, se volá metoda graphplanAlgorithm, která začne sestavování plánovacího grafu.

Pseudokód metody graphplanAlgorithm:

```
while(not end of graph reached)
{
     expand graph;
     resolve mutexes;
     if(goal achived)
     {
         try extract plan
         if(plan exists) {return plan; }
    }
}
```

Sestavování grafu tedy probíhá ve smyčce tak dlouho, dokud se nedojde na konec grafu nebo není nalezen plán. K detekci konce grafu je volána funkce endOf GraphReached(graph, depth), která zjistí, zda mezi posledním a předposledním krokem došlo k nějaké změně. Pokud totiž graf dosáhne svého konce, další expandované vrstvy jsou shodné s poslední.

Detekce dosažení cíle probíhá voláním funkce isGoalAchived(last layer of graph), která zjistí, zda se v posledním kroku vyskytují všechny cílové formule a zda žádná z nich není v relaci mutex s jinou cílovou formulí.

Expandování grafu probíhá ve dvou fázích. Nejdříve se volá metoda getPossibleOperations(), která má za parametry formule posledního kroku grafu a množinu mutexů na těchto formulích. Metoda za pomocí globálních seznamů ramen a bloků zkouší různé kombinace operací a testuje zda je daná operace proveditelná. V druhé fázi, se ke každé formuli z poslední vrstvy grafu vytvoří perzistentní akce, která má za výsledek (efekt), tu samou formuli, která je předpokladem.

V dalším kroku se volá funkce resolveMutexes(), která má za úkol nalézt všechny konflikty mezi operacemi a výslednými formulemi posledního kroku grafu. Metoda hledá konflikty podle pravidel popsaných v teoretické části této práce [1].

Důležitou součástí graphplanu je extrakce plánu. Pro větší rychlost je pro tento problém použita rekurze. Algoritmus extrakce pracuje podle následujícího pseudokódu:

```
bool tryExtract(goals, graph, depth, result)
{
  if(depth == 0) {return true; }
  else
  {
    candidates = generateCandidates(graph, depth, goals);
    if(candidates are valid)
    {
      combinations = generateCombinations(candidates);
      foreach combination c
```

```
      {
       if(tryExtract(new goals(graph, c, depth). graph, depth -2, result)
       {
          result.add(all operations in candidates);
          return true;
       }
      }
    }
    else{ return false; }
  }
} return false;
```

Metodě tryExtract je zaslán graf, aktuální hloubka grafu, aktuální podcíl na dané vrstvě a proměnná result ve které se vrací výsledek.

Pokud se dojde do hloubky nula, znamená to, že extrakce dosáhla od konce na začátek, tudíž musí existovat validní plán. Metoda tedy vrací true. Pokud nebylo dosaženo začátku, vygenerují se kandidáti, což jsou všechny možné operace, které nejsou vzájemně konfliktní a zároveň vedou k dosažení aktuálních cílů. Poté se zkontroluje, zda jsou kandidáti validní, to znamená, zda existuje alespoň jedna kombinace operací taková, aby bylo dosaženo všech formulí aktuálního cíle. Kandidáti jsou totiž pouze operace, které dosahují jednotlivých formulí, ale je nutné zjistit, jestli mohou být pokryty všechny formule najednou. Pokud kandidáti nejsou validní, znamená to, že v předešlých krocích byla zvolena špatná kombinace operací a tudíž se vrací false. Pokud jsou validní, vygenerují se všechny možné kombinace operací. K dosažení jedné formule z cíle může být použita například jedna ze tří možných operací. Je tedy nutné, všechny tyto kombinace otestovat. Je volán cyklus foreach, který pro každou kombinaci rekurzivně volá tryExtract. Cíle však vygeneruje nové a to takové, kterých by bylo nutné dosáhnout s aktuální kombinací operací. To se provádí voláním metody newGoals(). Dále je také v rekurzivním volání snížena hloubka grafu. Pokud toto volání vrací true, znamená to, že na všech nižších úrovních byly nalezeny validní operace a tudíž daná kombinace je správná. Uloží se tedy do seznamu result a vrací se true.

Jak je vidět, tak extrakce najde jeden z možných plánů. Nehledá všechny možné plány.

Pokud je nalezen plán, vrací se true a výsledek v seznamu result. Graphplan poté buď ukončí činnost, nebo expanduje graf o další úroveň a zkouší znova extrahovat plán. Pokud se dojde na konec grafu bez nalezení plánu, je vypsána chyba a úloha je zřejmě neřešitelná.

7.5 Sussmanova anomálie

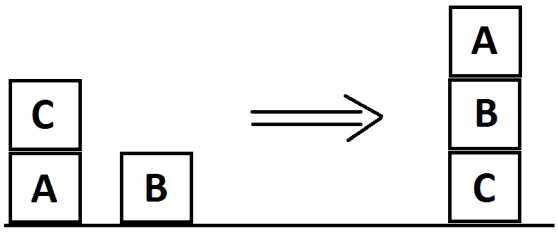
Je problém lineárního plánování [8]. Problém je demonstrován na úloze z prostředí block world. Máme tři bloky v následujícím seskupení: blok B a A leží na stole, blok C leží na A 7.3. Cílem je, aby C leželo na stole a B aby leželo na C a A na B.

Příčinou Sussmanovy anomálie je to, že lineární plánovače rozdělí cíl na podcíle jako: polož A na B a polož B na C. Plánovač dosáhne prvního cíle následovně 7.4.

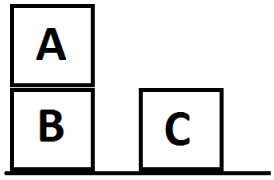
Nicméně nyní nemůže dosáhnout druhého podcíle bez toho, aby nezrušil dosažení prvního podcíle. Stejně tak, pokud nejdříve splní druhý podcíl, přesunutím B na C, dostane se do tohoto stavu 7.5.

Nyní ovšem zase nemůže dosáhnout prvního podcíle aniž by zrušil podcíl druhý.

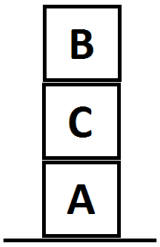
Tato anomálie je již řešitelná pomocí graphplanu. Vyzkoušíme tedy schopnost implementovaného algoritmu řešit Sussmanovu anomálii a to i za pomocí více ramen. Dle používaného



Obrázek 7.3: - Zadání Sussmanovy anomálie



Obrázek 7.4: - Splnění prvního podcíle lineárním plánovačem



Obrázek 7.5: - Splnění druhého podcíle lineárním plánovačem

zápisu nadefinujeme anomálii následovně.

```
armEmpty(1)
onTable(A)
onTable(B)
on(C,A)
clear(B)
clear(C)

goal
on(A,B)
on(B,C)
```

Takto bude vypadat vstupní soubor pro úlohu Sussmanovy anomálie pro řešení jedním

ramenem. Program s takovýmto souborem nalezne plán:

```
doUnstack(1,C,A)
putDown(1,C)
pickUp(1,B)
doStack(1,B,C)
pickUp(1,A)
doStack(1,A,B)
```

Plán má tedy šest kroků. Nyní přidáme do vstupního souboru formuli armEmpty(2). Tudíž budeme hledat plán pro dvě ramena. Program v tomto případě najde následující plán:

```
doUnstack(1,C,A)
putDown(1,C), pickUp(2,B)
pickUp(1,A), doStack(2,B,C)
doStack(1,A,B)
```

Vidíme, že více ramen umožňuje paralelní provedení některých operací a tím snížení počtu kroků na 4. Pokud zvýšíme počet ramen na tři, program vrací úplně stejný plán, protože při řešení neexistuje krok, kde by bylo možné provést paralelně tři operace, tudíž třetí rameno zůstává nevyužité.

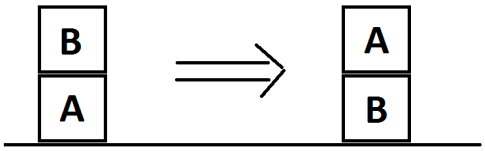
Příklad ukazuje, že program je schopný řešit Sussmanovu anomálií a potvrzuje, že graphplan hledající plán pro více ramen pracuje správně. Na výsledný plánovací graf, sestavený řešením pro dvě ramena, se lze podívat ve vytištěné příloze, založené na konci této práce.

7.6 Srovnání graphplanu a STRIPS plánovače

Strips je lineární plánovač, ve které je pořadí dosažených podcílů lineárně závislé na pořadí akcí, které jsou v plánu vykonány [7].

Definice akcí v plánovači STRIPS je téměř totožná s definicí popsanou v sekci 7.2. Hlavním rozdílem je to, že STRIPS místo záporného efektu (tedy šíření negace formule do dalšího kroku) přímo maže formule z databáze (delete list).

Jak již víme z předchozí sekce, tak lineární plánovače nejsou schopny řešit Sussmanovu anomálii. Vyzkoušíme však, zda by bylo možné navrhnout úlohu block world s více rameny pro plánovač STRIPS.

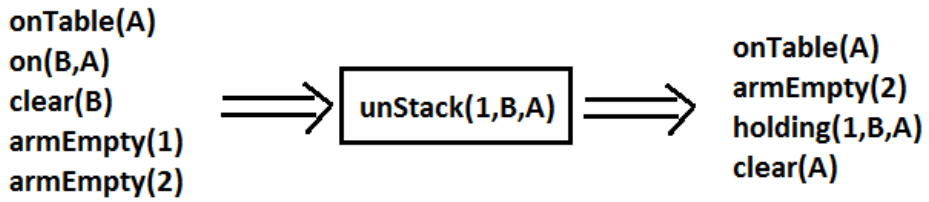


Obrázek 7.6: Příklad 1 - jednoduchá úloha block-world

Operace by opět měli navíc parametr udávající rameno, které akci provádí a formule typu holding a armEmpty, by tento parametr měli také. Dále bychom záporný efekt změnili na delete list. Nyní si na příkladu zkusíme ukázat, jaký by byl nalezený plán.

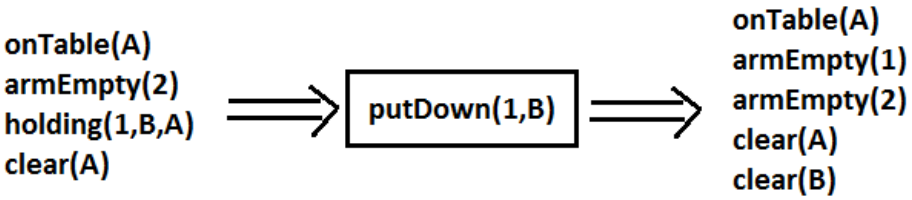
Vezměme jednoduchou úlohu. Máme dva bloky a ty chceme přerovnat v opačném pořadí 7.6.

Plán pro jedno rameno má čtyři kroky. Pokud použijeme implementovaný graphplan se dvěma rameny, kroky budou tři. STRIPS plánovač bude postupovat takto:



Obrázek 7.7: První krok plánovače STRIPS

Takto bude vypadat první krok. Jak je vidět, tak v dalším kroku by bylo možné využít druhého ramene na zvednutí bloku A. Avšak STRIPS pokračuje v dosažení podcíle onTable(B), tedy:



Obrázek 7.8: Druhý krok plánovače STRIPS

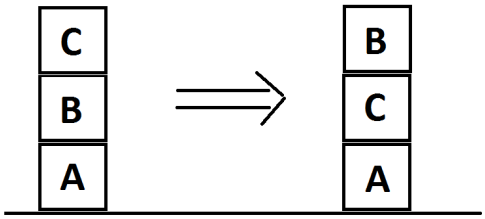
Pokud přeskočíme na konec, tak je zřejmé, že validní plán by v tomto případě byl následující:

```
1. unStack(1,B,A)
2. putDown(1,B)
3. pickUp(1,A)
4. stack(1,A,B)
```

Je vidět, že přidáním druhého ramene jsme v tomto případě dosáhli pouze určitého nedeterminizmu, protože plánovač by si mohl vybrat, jakým ramenem splní daný podcíl. Bylo by tedy možné vytvořit plán:

```
1. unStack(1,B,A)
2. putDown(1,B)
3. pickUp(2,A)
4. stack(2,A,B)
```

Uveďme si ještě jeden složitější příklad. Mějme bloky a cílový stav definován takto:



Obrázek 7.9: Příklad 2 - složitější úloha block-world

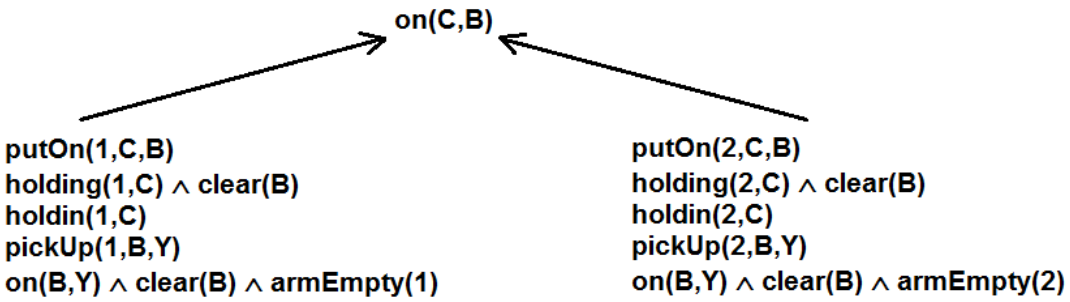
Jedná se tedy o prohození dvou vrchních bloků. Logickým postupem v takovéto situaci, by bylo jedním ramenem zvednout blok C, druhým B a položit je v opačném pořadí. STRIPS však bude mít následující cíl:

$$onTable(A) \wedge on(C, B) \wedge on(B, C)$$

Blok A, je již na stole, proto se pustí do řešení podcíle on(C,B). K tomuto podcíly najde plán:

```
1. unStack(1,C,B)
2. putDown(1,C)
3. unStack(1,B,A)
4. putDown(1,B)
5. pickUp(1,C)
6. stack(1,C,A)
```

Pokud bychom si měli zobrazit zásobník cílů pro řešení podcíle on(C,B), vypadal by nějak takto:

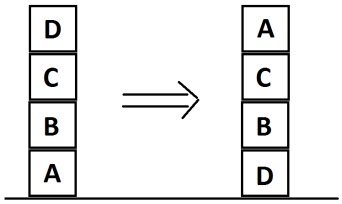


Obrázek 7.10: Zásobník cílů pro řešení podcíle on(C,B).

Vidíme tedy, že přidáním dalšího ramene do úlohy block world, řešené plánovačem STRIPS, nedosáhneme lepšího výsledku. Jakmile planovač započne provádět akce jedním ramenem k dosažení cíle, již s tímto ramenem pracuje dál a druhé vůbec nevyužívá. Závěrem je, že definovat úlohu s více rameny pro STRIPS nemá smysl. Nicméně, jsme tímto příkladem ukázali, čím se graphplan a STRIPS liší a proč je graphplan lepší.

7.7 Testy a výsledky

Otestujeme program na složitějších úlohách. Mějme složitější systém o čtyřech kostkách s cílovým a koncovým stavem definovaným obrázkem 7.11. Snažíme se tedy ve sloupci čtyř bloků prohodit vrchní a spodní blok.



Obrázek 7.11: - První testovaná úloha

Vstupní soubor bude mít tedy obsah:

```
armEmpty(1)
onTable(A)
on(B,A)
on(C,B)
on(D,C)
clear(D)

goal
onTable(D)
on(B,D)
on(C,B)
on(A,C)
```

Pro jedno rameno tedy algoritmus nalezen následující plán:

```
doUnStack(1,D,C)
putDown(1,D)
doUnStack(1,C,B)
putDown(1,C)
doUnStack(1,B,A)
doStack(1,B,D)
pickUp(1,C)
doStack(1,C,B)
pickUp(1,A)
doStack(1,A,C)
```

Plán má deset kroků a jeho výpočet trvá 0,7 vteřiny. Nyní přidáme do vstupního souboru formuli armEmpty(2). Budeme se snažit najít plán pro dvě ramena. Program nalezne následující plán:

```
doUnStack(1,D,C)
```

```
putDown(1,D), doUnStack(2,C,B)
doUnStack(1,B,A), putDown(2,C)
doStack(1,B,D), pickUp(2,C)
doStack(2,C,B), pickUp(1,A)
doStack(1,A,C)
```

Což je šest kroků a výpočet trvá 3,1 vteřiny. Ovšem zde se objevuje problém, že v kroku 3 a 4 druhé rameno místo toho aby čekalo se zdviženým blokem C na první rameno, až přesune B na D tak provede zbytečnou operaci, že položí blok C a v následujícím kroku ho opět zvedne. Toto „aktivní čekání“ je způsobeno tím, že akce v kroku 3 a 4 nejsou konfliktní a mohou být provedeny paralelně. Graphplan nerozlišuje, zda akce může být provedena nebo musí být provedena. Takže když je možné provést nějakou akci, která nepovede k jiným cílům tak jí graphplan přidá do plánu.

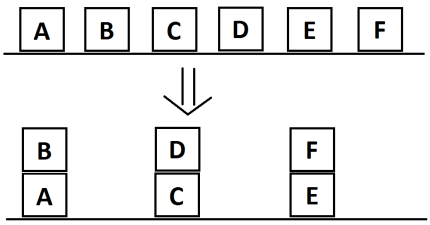
Časová složitost je navýšena logicky proto, že se zvětšují množiny operací a formulí. To co je proveditelné jedním ramenem může být provedeno i ramenem druhým. Formulí holding, je nyní více, protože co drží jedno rameno, může držet i druhé. Tím pádem se zvětšují také množiny mutexů a tím jsou náročnější operace nad těmito seznamy.

Pokud ze zvědavosti zkusíme přidat třetí rameno, výsledek bude následující:

```
doUnStack(2,D,C)
doUnStack(3,C,B)
putDown(2,D), doUnStack(1,B,A), putDown(3,C)
doStack(1,B,D), pickUp(2,C)
doStack(2,C,B), pickUp(1,A)
doStack(1,A,C)
```

Je vidět, že třetí rameno je naprosto zbytečné pro takto malou scénu. Navíc zbytečně vzroste časová složitost na 11,3 vteřiny.

Ukážeme si ale druhý příklad, který bude mít šest bloků na stole a cílem bude poskládat tři dvojice 7.12. V tomto příkladě by mělo být již plně využito třech ramen.



Obrázek 7.12: - Druhá testovaná úloha

S jedním ramenem najde program za 35,1 vteřiny následující plán:

```
pickUp(1,F)
doStack(1,F,E)
pickUp(1,D)
doStack(1,D,C)
pickUp(1,B)
doStack(1,B,A)
```

S dvěmi rameny se časová náročnost již velmi zvedá a výpočet plánu trvá 74,9 vteřin. Ve výsledném plánu je opět zbytečná akce a to když v prvním kroku rameno 1 zvedne blok D a v druhém kroku ho položí na A, odkud ho zvedne ve třetím kroku druhé rameno. Plán je tedy:

```
pickUp(1,D), pickUp(2,F)
doStack(1,D,A), doStack(2,F,E)
pickUp(1,B), doUnStack(2,D,A)
doStack(1,B,A), doStack(2,D,C)
```

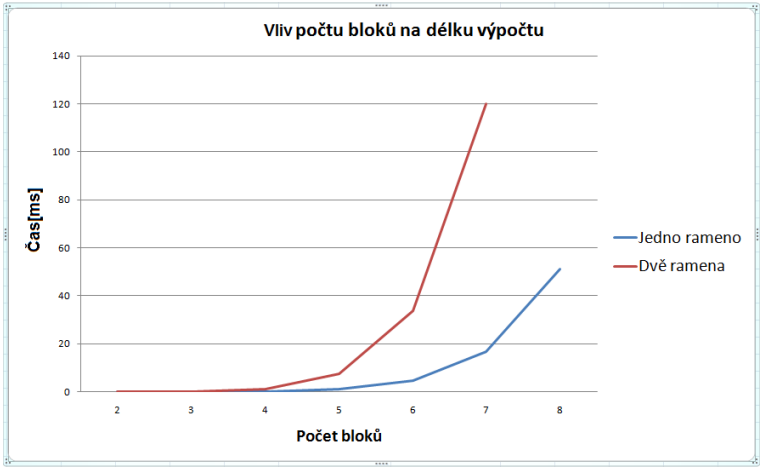
Pokud zvolíme tři ramena, není potřeba expandovat graf do větších hloubek, protože plán má pouze dva kroky a výpočet proto trvá pouze 11 vteřin. Nalezený plán je tedy:

```
pickUp(1,B), pickUp(2,D), pickUp(3,F)
doStack(1,B,A), doStack(2,D,C), doStack(3,F,E)
```

Na předchozích dvou příkladech jde vidět, že pokud je úloha snadno paralelizovatelná potom výpočet plánu pro více ramen může trvat kratší dobu. Pokud ovšem úloha zrovna nevyžaduje paralelismus akcí, poté s větším počtem ramen pouze narůstá časová složitost. Problém nárůstu časové složitosti bude vysvětlen v další kapitole.

7.8 Problém nárůstu časové složitosti

Jak je vidět v předchozí kapitole, tak časová náročnost s rostoucím počtem bloků velmi výrazně stoupá. Pro ilustraci vytvoříme graf závislosti počtu bloků na délce výpočtu plánu. Úlohy budou mít pro jednoduchost stejný charakter. Několik bloků bude vždy na sobě v jednom sloupci a úkolem bude tyto bloky přerovnat do druhého sloupce v opačném pořadí.



Obrázek 7.13: - Graf závislosti množství kostek na délku výpočtu

Z grafu 7.13 je jasné, že pokud bychom program chtěli použít pro větší množství bloků, výsledku bychom se v rozumném čase nedočkali. Rapidní nárůst časové složitosti je způsoben především typem úlohy.

Graphplan pracuje tak, že v každém kroku zkouší všechny operace, jejichž předpoklady jsou splněny a nejsou v relaci mutex. Ovšem pokud máme definované operace, jako v tomto případě, predikátovou logikou, vzniká nám velké množství možných dosazení do logických formulí.

Víme, že s rostoucím krokem, se množství operací zvyšuje. Pokud budeme mít například úlohu se čtyřmi bloky, tak počet možných operací v posledním kroku bude 66 pro jedno rameno. Pokud použijeme dvě ramena, je počet operací 105. Mezi těmito operacemi, vzniká velké množství mutexů. Problémem této úlohy je v tom, že všechny akce v daném kroku, které jsou prováděny stejným ramenem, jsou vzájemně exkluzivní. Je to logické, protože v jednom kroku nelze ramenem provést více jak jednu akci. Pokud tedy máme 66 operací, z toho 27 jsou akce ramene (zbytek jsou perzistentní akce) a tvoříme dvojice mutexů, tedy kombinace bez opakování, tak nám vznikne 351 konfliktů, pouze mezi těmito akcemi. Přičemž výpočet plánu pro čtyři bloky jedním ramenem trvá 0,18 vteřiny.

Mějme příklad s osmi bloky, kde výpočet trvá 51,3 vteřiny. V tomto případě je v poslední vrstvě grafu 250 možných operací, bez perzistentních akcí 115. Dvojic mutexů mezi těmito operacemi je 6555.

Jak je vidět, počty mutexů jsou velmi vysoké. Operace nad takovými rozsáhlými seznamy, jsou pochopitelně časově velice náročné. Tento fakt je ovšem způsoben typem úlohy. Optimalizace pro prostředí block world by mohlo být předmětem další práce.

Kapitola 8

Program blackbox

Blackbox je plánovací systém, který pracuje tak, že převádí problém, specifikovaný ve STRIPS na boolovský problém pokrytí (boolean satisfiability problem) a ten pak řeší různými technikami (algoritmy). Hlavním algoritmem je graphplan (Blum a Furst 1995). Program ovšem ve volbě algoritmu nabízí velkou volnost a je možné výběr kombinovat. Například je možné použít walksat(Selman, Kautz a Cohen 1994) po dobu 60 vteřin a pokud algoritmus selže, lze nastavit, aby se poté spustil satz (Li a Anbulagan 1997) po dobu 1000 sekund. Tento fakt umožňuje blackboxu pracovat efektivně na velmi široké množině problémů. Název blackbox odkazuje na fakt, že generátor plánu neví nic o výpočetním zařízení SAT a naopak. Program je možné stáhnout na oficiálních stránkách blackboxu [2] a to jednoduše jako již spustitelný binární soubor (jak pro windows tak i pro linux) nebo ve formě zdrojových kódů.

8.1 Jazyk PDDL

Jazyk PDDL (planning domain definition language) byl vyvinut ve snaze standardizovat popis domény a plánovacího problému (to umožnilo první mezinárodní soutěž plánovacích algoritmů 1998/2000). PDDL obsahuje STRIPS, ADL a další jazyky, avšak většina plánovačů nepodporuje celé PDDL. Většinou je podporována nejpoužívanější podmnožina a to STRIPS. PDDL se stále rozvíjí a vznikají nové specifikace (současně verze 3.1 z roku 2008). Definice úlohy v jazyce PDDL se skládá ze dvou částí:

- Definice domény
- Definice problému

Požadavky
Protože je PDDL velmi obecný jazyk a většina plánovačů podporuje pouze určitou podmnožinu, mohou být v doméně deklarovány požadavky. Nejčastějšími požadavky jsou:

- : *strips* - nejzákladnější podmnožina PDDL skládající se pouze ze STRIPS.
- : *equality* - tento požadavek znamená, že doména využívá symbol = jako rovnost (nikoliv přiřazení).
- : *typing* - doména používá typy.
- : *adl* - doména využívá některých nebo všech funkcí adl.

Definice domény
Definice domény obsahuje predikáty a operátory (akce). Může také obsahovat typy, konstanty, statická fakta atd. ale tyto vlastnosti nebývají podporovány většinou plánovačů. Příklad - jednoduchá definice domény [2].

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  (:predicates (PREDICATE_1_NAME [?A1 ?A2 ... ?AN])
               (PREDICATE_2_NAME [?A1 ?A2 ... ?AN])
       ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
   )

  (:action ACTION_2_NAME
    ...)

  ...)
```

Definice akcí
Všechny části definice akcí jsou, kromě jména, dle specifikace, nepovinné. Nicméně pro akce, které nemají předpoklady, mohou nějaké plánovače vyžadovat prázdný předpoklad (:precondition()).

Předpoklady formulí
V doméně STRIPS mohou být předpoklady akcí:

- Atomické formule.
- Konjunkce atomických formulí.

Pokud doména používá :equality, mohou mít atomické formule tvar (= arg1 arg2). Hodně plánovačů, které podporují equality, také podporují negovanou rovnost, tedy: (not (=arg1 arg2)) a to i v případě, že doména neumožňuje negace v jiných částech definice.
V doméně ADL mohou být předpoklady:

- Negace, konjunkce nebo disjunkce atomických formulí (not, and, or).
- Kavantifikované formule (pro všechny (?v1 ?v2 ...) podmínka).

Efekty operací
V jazyce PDDL nejsou efekty děleny do add a delete listů. Místo toho je negativní efekt reprezentován negací.
Ve STRIPS mohou být efekty:

- Přidané atomické formule.
- Přidané negace atomických formulí.

- Konjunkce (negací) atomických formulí.

V doméně ADL mohou být efekty:

- Podmíněné - tedy efekt je přidán, pokud je splněna nějaká podmínka.
- Kvantifikované formule.

Definice problému
Definice problému obsahuje všechny objekty, které jsou přítomny v dané instanci a popis počátečního a koncového stavu.
Příklad - jednoduchá definice problému [2].

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA)
)
```

Definice počátečního stavu (část :init) je seznam atomických formulí, které jsou na začátku platné. Všechny ostatní formule jsou dle definice false. Popis cíle je formule, ve stejném tvaru jako jsou předpoklady akcí.

8.2 Typy úloh

Pro testování a porovnání programu blackbox využijeme kromě úlohy blockworld také další dvě domény.

Bulldozer
Doména s vozidlem, kde musí osoba nasednout do vozidla, dojet s ním na určité místo, vystoupit a dojít na nějaké jiné místo. Jsou definována místa a mezi nimi je možný přesun po cestě nebo mostě. Je možné použít následující akce:

- Drive :(?thing ?from ?to)
- Cross :(?thing ?from ?to)
- Board :(?person ?place ?vehicle)
- Disembark :(?person ?place ?vehicle)

Tire world
V této úloze je cílem naleznout správný postup při výměně pneumatiky. Tato doména obsahuje 13 operací, které je možné provádět buď sériově, nebo paralelně. Operace jsou:

```
open-container - otevřít kontajner
close-container - zavřít kontajner
fetch - vytáhnout (z˜kontajneru)
put-away - uklidit (do kontajneru)
```

```
loosen - povolit (šrouby)
tighten - utáhnout (šrouby)
jack-up - zvednout heverem
jack-down - spustit heverem
remove-nuts - odebrat šrouby
put-on-nuts - nasadit šrouby
remove-wheel - odstranit kolo
put-on-wheel - nasadit kolo
infalate - nafouknout (napumpovat)
```

Zadání úlohy v takovéto doméně může být například vyhledání postupu jak sundat, vyměnit nebo pouze nafouknout pneumatiku.

8.3 Testy a výsledky

Začneme u domény bulldozer.
Mějme:

- Množinu míst a,b,c,d,e,f,g.
- Osobu Jack.
- Bulldozer.

V počátečním stavu bude Jack v místě a. Bulldozer v místě e. Cílem je, aby Jack dojel s bulldozerem do místa g a sám skončil v a. (kompletní zadání v PDDL lze najít v příloze A). Mezi některými místy jsou definovány silnice a mosty. Blackbox využívající plánovač satz, nalezne za 0.08 sekundy následující plán:

```
1. (drive jack a e)
2. (board jack e bulldozer)
3. (drive bulldozer e b)
4. (cross bulldozer b d)
5. (drive bulldozer d g)
6. (disembark jack g bulldozer)
7. (drive jack g d)
8. (cross jack d b)
9. (drive jack b a)
```

Je zřejmé, že nalezený plán je správný. Bohužel tento typ úlohy nenabízí žádnou možnost paralelismu.

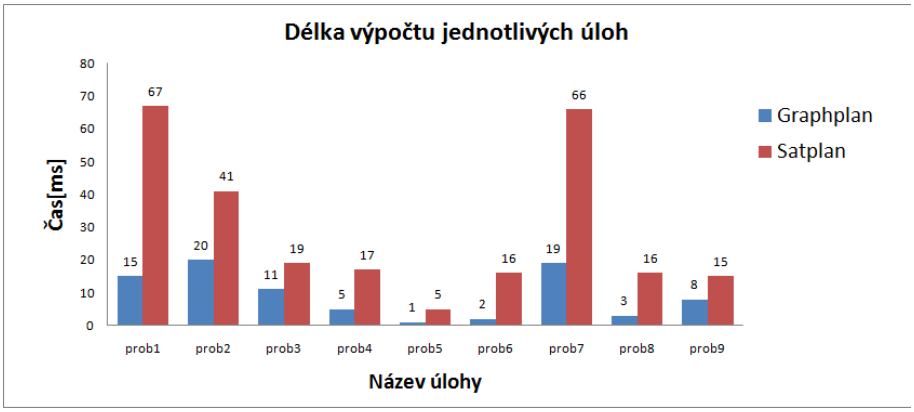
Druhý test provedeme v doméně tire-world. Zde je již možné některé akce provádět paralelně. Uvedeme tedy hned takový příklad. Mějme kufr auto (trunk), ve kterém se nachází rezerva (wheel2), klíč (wrench), hever (jack). Cílem bude vyměnit kolo (wheel1). Celé zadání v PDDL lze nalézt v příloze B. Blackbox využívající plánovač satz nalezne za 0.03 sekundy následující plán:

```
1. (open-container trunk)
2. (fetch wrench trunk)
2. (fetch jack trunk)
2. (fetch wheel2 trunk)
3. (loosen nuts the-hub)
4. (jack-up the-hub)
5. (remove-nuts nuts the-hub)
6. (remove-wheel wheel1 the-hub)
7. (put-on-wheel wheel2 the-hub)
7. (put-away wrench trunk)
8. (jack-down the hub)
9. (put-away wheel1 trunk)
9. (put-away jack trunk)
10. (close-container trunk)
```

Vidíme, že v kroku 2, 7 a 9 je prováděno několik operací paralelně. V druhém kroku je najednou vytažen z kufru klíč, hever a rezerva. V sedmém kroku je zároveň nasazena rezerva a uklizen klíč (ve specifikaci úlohy není dané, že rezerva musí být přidělána šrouby). V devátém kroku je zároveň uklizeno píchlé kolo a hever.

Z toho vyplívá, že satplan, je stejně jako graphplan, schopen nalézt nekonfliktní operace, které je možné ve výsledném plánu provádět paralelně.

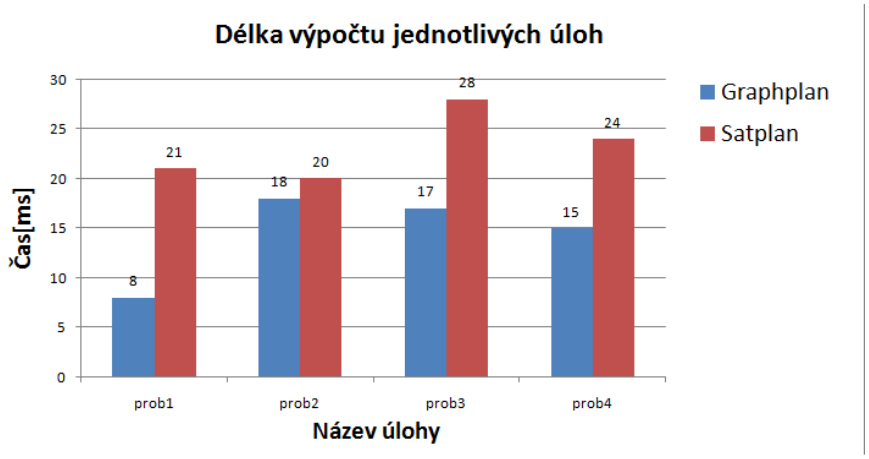
Nakonec zbývá úloha block world. Nicméně jsme již na předchozích příkladech otestovali funkčnost satplanu. Proto provedeme několik porovnání časové složitosti plánovačů graphplan a satplan. Využijeme proto všech příkladů z oficiálních stránek blackboxu [2], v doménách tire-world, bulldozer a block world. Do grafů vyznačíme délku výpočtu jednotlivých příkladů pro oba algoritmy.



Obrázek 8.1: - Graf, udávající délky výpočtů úloh z domény bulldozer, pro graphplan a satplan

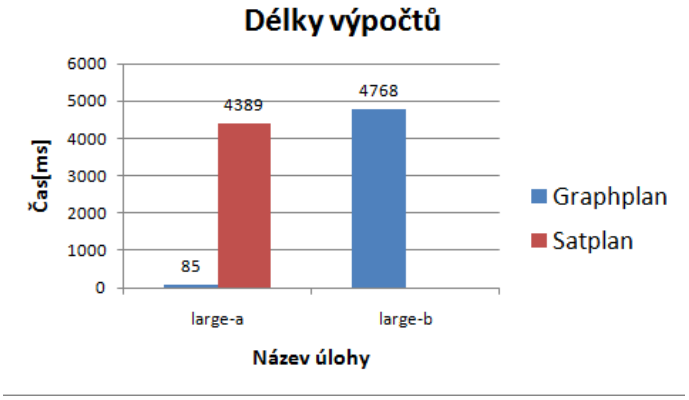
Jak je z grafů [8.1,8.2] zřejmé, tak v doménách typu bulldozer a tire-world podávají oba algoritmy téměř stejné výsledky.

Pokud se ovšem podíváme na poslední graf [8.3] je vidět, že ve složitější doméně typu block world, satplan zaostává za graphplanem. Zatímco graphplan plán pro první úlohu o devíti blocích najde za 85 milisekund, satplan až za 4768 milisekund. U druhého úkolu



Obrázek 8.2: - Graf, udávající délky výpočtů úloh z domény tire-world, pro graphplan a satplan

(o jedenácti blocích) už i graphplanu trvá výpočet 4389 a satplan dokonce nedojde výsledku v rozumném čase (při spouštění na školním serveru merlin, po několika minutách dojde k hlášení - CPU time limit exceeded). Třetí úlohu s patnácti bloky, již nespočítá ani graphplan s proto v grafu ani není.



Obrázek 8.3: - Graf, udávající délky výpočtů úloh z domény block-world, pro graphplan a satplan

Z výsledků je možné vyvodit závěr, že jak satplan tak i graphplan jsou rychlé a efektivní algoritmy pro jednodušší domény. Pokud ovšem zvolíme složitější příklady, stavový prostor (i když redukovaný graphplanem) a množství faktů, které je nutné prohledat je natolik veliké, že algoritmy selhávají. Tímto faktem se také vysvětluje časová náročnost implementovaného graphplanu z kapitoly 7. Pokud je časově náročné řešit block world pro jedno rameno, je logické, že s více rameny se bude časová složitost zvyšovat.

Kapitola 9

Trendy ve vývoji plánovacích algoritmů

Při studiu plánovacích algoritmů jsem zjistil, že často místo toho, aby vznikaly nové algoritmy, tak se objevují optimalizace a rozšíření stávajících (již osvědčených) plánovačů jako jsou právě graphplan nebo A*.

Cílem těchto modifikací je, upravit algoritmy takovým způsobem, aby měli nějaké (v dnešní době už nezbytné) vlastnosti jako je paralelismus, schopnost provádět výpočty v reálném čase nebo distribuce jednoho algoritmu mezi více agentů, kteří poté spolupracují.

Z modifikací A* jsem zvolil k implementaci real time adaptive A*. Tento algoritmu však není jediným vylepšením. Existuje mnoho dalších jako je fringe-saving A*, který hledá minimální cestu rychleji za pomocí opakovaného vyhledávání a využívá předešlé hodnoty, k urychlení aktuálního výpočtu. Další modifikací je eager and lazy target-moving adaptive A*, který počítá cestu k pohybujícímu se cíly. Je založený na pozorování okolí přes senzory a opakovaném vyhledávání algoritmem A*. Nebo například learning real time A*, který využívá učení. A* a jeho modifikace by sami o osobě stačily na celou práci.

Graphplan má také své rozšíření, ovšem ty se nezaměřují na časovou složitost, ale na distribuci a paralelismus. Na internetu je možné nalézt mnoho prací a článků, které řeší paralelní a distribuované úlohy, například problém z domény block-world s více rameny, přičemž každé rameno má svůj vlastní cíl.

Nicméně vznikají také nové plánovací algoritmy, ale nikoliv pro staré úlohy. Jak vznikají nové technologie, je zapotřebí nových algoritmů. Jedná se například o 3D navigaci ve virtuální scéně a s tím spojené automatické řízení dopravních prostředků.

Technická úroveň nových robotů a zařízení se zvyšuje a bude potřeba stále nových a složitějších algoritmů. Přesto se však domnívám, že minimálně základní principy budou vycházet ze starších a osvědčených algoritmů.

Kapitola 10

Závěr

V teoretické části byly popsány principy algoritmů graphplan, satplan a real-time adaptive A*. V kapitole věnované graphplanu byl vysvětlen princip plánovacích grafů, práce s operacemi a stavy, které jsou v relaci mutex a na pseudokódu byla ukázána činnost algoritmu. Kapitola o real-time adaptive A* popisuje hlavní myšlenku tohoto algoritmu a vlastnosti, kterými se liší od klasického A*. V poslední teoretické kapitole je vysvětlen satplan a základy plánování s výrokovou logikou.

Praktická část práce je rozdělena do několika kapitol, které obsahují popis implementací jednotlivých algoritmů. Každá kapitola obsahuje popis uživatelského prostředí, použitých tříd a chodu programu. Zároveň jsou zde uvedeny výsledky testů z netriviálních domén.

Algoritmus real-time adaptive A* byl testován v doméně tile-world. Jedná se o úlohu, kdy se agent pohybuje po mřížce a jeho cílem je dosáhnout, náhodně se objevujících, cílů. Prostředí je tedy dynamické. Testy byly prováděny v délce 1 a 2 sekundy, přičemž tento čas byl vymezen pouze pro samotný výpočet cesty. Výsledky tedy podávají zprávu o tom, kolik dokáže algoritmus naplánovat cest v daném limitu. Testy byly prováděny s různým nastavením rozhledu (lookahead). Hlavním cílem bylo porovnat real-time variantu s klasickým A*. Z výsledků je zřejmé, že s rostoucím rozhledem se zvyšuje časová náročnost, ale také klesá cena cesty. Real-time varianta je tedy schopna nalézt neoptimální cestu velice rychle, avšak jejím hlavním problémem je správná volba rozhledu, jak je nastíněno v sekci 6.5.

Graphplan byl testován v doméně block-world a to s možností práce s více rameny. Hlavním cílem tedy bylo, otestovat schopnost algoritmu hledat paralelně proveditelné akce. Nejdříve se implementovaný program testoval na úloze Sussmanovi anomálie, aby se potvrdilo, že graphplan netrpí stejným problémem jako lineární plánovače. Testy proběhli úspěšně jak pro jedno, tak i dvě ramena. V dalších úlohách se kladl důraz na plné využití paralelismu, tedy najití plánu s co nejmenším počtem kroků. Při těchto testech se ukázal problém, způsobený větším počtem ramen, který by se dal nazvat jako aktivní čekání. Jedná se o situaci, kdy by rameno mělo nějaký čas držet blok a čekat, ale místo toho, blok položí a opět zvedne. I přes tuto vlastnost se prokázalo, že graphplan je schopen nalézt nekonfliktní akce pro více ramen, které lze vykonat paralelně v jednom kroku plánu. Nakonec byla popsána časová složitost algoritmu a zároveň byl vysvětlen velký nárůst časové složitosti s přibývajícím počtem bloků.

Pro testy algoritmu satplan, byl využit existující program blackbox. Ten dokáže řešit libovolnou ulohu zapsanou v jazyce PDDL a to buď pomocí satplanu nebo graphplanu. Nabídla se tedy možnost, tyto dva algoritmy otestovat na stejných úlohách a porovnat dosažené výsledky. Testy probíhaly v doménách block-world, tire-world a bulldozer. Zatímco v posledních dvou zmiňovaných příkladech podávali oba algoritmy podobné výsledky (sat-

plan byl pomalejší v řádech pár desítek milisekund), v prostředí block-world se projevilo, že s větším stavovým prostorem, který je nutné prohledat, se vysoce snižuje výkonost satplanu oproti graphplanu. Také se ale ukázalo, že satplan dokáže vyhledávat paralelně proveditelné akce.

Testy prokázaly, že jsou tyto algoritmy schopny tvořit plány, na které jsou kladeny vyšší nároky jako je real-time výpočet (RTAA*) nebo paralelismus (graphplan, satplan). Nicméně jejich využití v opravdových robotických systémech se nezdá být příliš reálné, ať už z důvodu délky výpočtu nebo paměťové náročnosti. Studiem a úpravou principů těchto algoritmů by však mohly být vytvořené nové a lepší plánovací techniky.

Vývoj algoritmů stále pokračuje. S rostoucím výkonem počítačů a s rozvojem nových technologií se budou čím dál více objevovat nové směry a odvětví plánování. Příkladem může být například 3D navigace autonomně řízeného automobilu.

Literatura

[1] Mutexes in Graphplan and their use to find a plan.
http://www-users.itlabs.umn.edu/classes/Spring-2009/csci5511/mutex.html.

[2] Blackbox. http://www.cs.rochester.edu/ kautz/satplan/blackbox/, 2003.

[3] Blum, A.: Graphplan Home Page. http://www.cs.cmu.edu/ãvrim/graphplan.html,
Last modified: June 2001.

[4] DASGUPTA, P. P.: Graphplan and Satplan, Lecture Video.
http://www.eetindia.co.in/VIDEO_DETAILS_700000075.HTM, 2004.

[5] Koenig, S.; Likhachev, M.: Real-Time Adaptive A.
http://idm-lab.org/bib/abstracts/papers/aamas06.pdf, 2006.

[6] Russell, S. J.; Norvig, P.: *Artificial Intelligence - A Modern Approach*. Alan Apt,
1995, iSBN 0-13-103805-2.

[7] Schmidt, C. F.: Planning.
http://www.rci.rutgers.edu/cfs/472_html/Planning/PlanningToc.html.

[8] wikipedia: Sussman anomaly. http://en.wikipedia.org/wiki/Sussman_Anomaly, 2009.

[9] Zbořil, F.; ml., F. Z.: Opora kurzu Základy umělé inteligence.
https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IZU-IT/texts/oporaIZU-ESF-4.pdf,
2006.

Dodatek A

Zadání úlohy z domény bulldozer v jazyce PDDL

A.1 Definice domény

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Simple Vehicle domain where a person has to get in to a vehicle,
;;; drive it somewhere, get out, and return to some other location.
;;; Recursion is a problem in this domain because the roads and
;;; bridges go both directions.

(define (domain bulldozer)
  (:requirements :strips :equality)
  (:predicates (road ?from ?to)
       (at ?thing ?place)
       (mobile ?thing)
       (bridge ?from ?to)
       (person ?p)
       (vehicle ?v)
       (driving ?p ?v))

  (:action Drive
     :parameters (?thing ?from ?to)
     :precondition (and (road ?from ?to)
 (at ?thing ?from)
 (mobile ?thing)
 (not (= ?from ?to)))
     :effect (and (at ?thing ?to) (not (at ?thing ?from))))
  (:action Cross
     :parameters (?thing ?from ?to)
     :precondition (and (bridge ?from ?to)
 (at ?thing ?from)
 (mobile ?thing)
 (not (= ?from ?to)))
     :effect (and (at ?thing ?to) (not (at ?thing ?from))))
```

```
 (:action Board
    :parameters (?person ?place ?vehicle)
    :precondition (and (at ?person ?place)
(person ?person)
(vehicle ?vehicle)
(at ?vehicle ?place)
(not (= ?person ?vehicle)))
    :effect (and (driving ?person ?vehicle)
  (mobile ?vehicle)
  (not (at ?person ?place))
  (not (mobile ?person))))
 (:action Disembark
    :parameters (?person ?place ?vehicle)
    :precondition (and (person ?person)
(vehicle ?vehicle)
(driving ?person ?vehicle)
(at ?vehicle ?place)
(not (= ?person ?vehicle)))
    :effect (and (at ?person ?place)
  (mobile ?person)
  (not (driving ?person ?vehicle))
  (not (mobile ?vehicle))))
 )
```

A.2 Definice problému

```
(define (problem Big-bull1)
    (:domain bulldozer)
  (:objects a b c d e f g jack bulldozer)
  (:goal (and (at bulldozer g) (at jack a)))
  (:init (at jack a) (at bulldozer e)
 (vehicle bulldozer)
 (mobile jack)
 (person jack)
 (road a b) (road b a)
 (road a e) (road e a)
 (road e b) (road b e)
 (road a c) (road c a)
 (road c b) (road b c)
 (bridge b d) (bridge d b)
 (bridge c f) (bridge f c)
 (road d f) (road f d)
 (road f g) (road g f)
 (road d g) (road g d)))
```

Dodatek B

Zadání úlohy z domény tire-world v jazyce PDDL

B.1 Definice domény

```
; (c) 1993,1994 Copyright (c) University of Washington
;  Written by Tony Barrett.

;  All rights reserved. Use of this software is permitted for non-commercial
;  research purposes, and it may be copied only for that use.  All copies must
;  include this copyright message.  This software is made available AS IS, and
;  neither the authors nor the University of Washington make any warranty about
;  the software or its performance.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; flat-tire domain (from Stuart Russell)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; First the Strips version

(define (domain flat-tire-strips)
  (:requirements :strips :equality)

  (:constants wrench jack pump)
  (:predicates (annoyed)
       (container ?c)
       (locked ?c)
       (open ?c)
       (in ?x ?c)
       (have ?x)
       (nut ?n)
       (hub ?h)
       (loose ?x ?h)
       (tight ?x ?h)
```

```
     (on-ground ?h)
     (unfastened ?h)
     (on ?x ?h)
     (wheel ?w)
     (free ?h)
     (inflated ?w)
     (intact ?w))

(:action cuss
    :effect (not (annoyed)))

(:action open-container
    :parameters (?c)
    :precondition (and (container ?c) (not (locked ?c)) (not (open ?c)))
    :effect (open ?c))

(:action close-container
    :parameters (?c)
    :precondition (and (container ?c) (open ?c))
    :effect (not (open ?c)))

(:action fetch
    :parameters (?x ?c)
    :precondition (and (container ?c) (in ?x ?c) (open ?c) (not (= ?x ?c)))
    :effect (and (have ?x)
  (not (in ?x ?c))))

(:action put-away
    :parameters (?x ?c)
    :precondition (and (container ?c) (have ?x) (open ?c) (not (= ?x ?c)))
    :effect (and (in ?x ?c)
  (not (have ?x))))

(:action loosen
    :parameters (?x ?h)
    :precondition (and (nut ?x) (hub ?h) (have wrench)
(tight ?x ?h) (on-ground ?h) (not (= ?x ?h)))
    :effect (and (loose ?x ?h)
  (not (tight ?x ?h))))

(:action tighten
    :parameters (?x ?h)
    :precondition (and (nut ?x) (hub ?h) (have wrench) (loose ?x ?h)
(on-ground ?h) (not (= ?x ?h)))
    :effect (and (tight ?x ?h)
  (not (loose ?x ?h))))

(:action jack-up
```

```
    :parameters (?h)
    :precondition (and (hub ?h) (on-ground ?h) (have jack))
    :effect (and (not (on-ground ?h))
  (not (have jack))))

;; jacking down wheel x on hub y (dependency would be better)
(:action jack-down
    :parameters (?h)
    :precondition (and (hub ?h) (not (on-ground ?h)))
    :effect (and (on-ground ?h)
  (have jack)))

(:action remove-nuts
    :parameters (?x ?h)
    :precondition (and (nut ?x) (hub ?h) (not (= ?x ?h))
(not (on-ground ?h)) (not (unfastened ?h))
(have wrench) (loose ?x ?h))
    :effect (and (have ?x) (unfastened ?h)
  (not (on ?x ?h)) (not (loose ?x ?h))))

(:action put-on-nuts
    :parameters (?x ?h)
    :precondition (and (nut ?x) (hub ?h) (not (= ?x ?h))
(have wrench) (unfastened ?h)
(not (on-ground ?h)) (have ?x))
    :effect
    (and (loose ?x ?h) (not (unfastened ?h)) (not (have ?x))))

(:action remove-wheel
    :parameters (?w ?h)
    :precondition (and  (wheel ?w) (hub ?h) (not (= ?w ?h))
 (not (on-ground ?h)) (on ?w ?h) (unfastened ?h))
    :effect (and (have ?w) (free ?h) (not (on ?w ?h))))

(:action put-on-wheel
    :parameters (?w ?h)
    :precondition (and (hub ?h) (wheel ?w) (not (= ?w ?h)) (have ?w)
(free ?h) (unfastened ?h) (not (on-ground ?h)))
    :effect
    (and (on ?w ?h) (not (have ?w)) (not (free ?h))))

(:action inflate
    :parameters (?w)
    :precondition (and (wheel ?w) (have pump) (not (inflated ?w))
(intact ?w))
    :effect (inflated ?w)))
```

B.2 Definice problému

```
(DEFINE (PROBLEM FIX-STRIPS2)
    (:DOMAIN FLAT-TIRE-STRIPS)
  (:objects wheel1 wheel2 the-hub nuts trunk)
  (:init (WHEEL WHEEL1) (WHEEL WHEEL2) (HUB the-HUB) (NUT NUTS) (CONTAINER TRUNK)
 (INTACT WHEEL2) (IN JACK TRUNK) (IN PUMP TRUNK) (IN WHEEL2 TRUNK)
 (IN WRENCH TRUNK) (ON WHEEL1 THE-HUB) (ON-GROUND THE-HUB) (TIGHT NUTS THE-HUB)
 (NOT (LOCKED TRUNK)) (NOT (OPEN TRUNK)) (NOT (UNFASTENED THE-HUB))
 (NOT (INFLATED WHEEL2)) (NOT (INFLATED WHEEL1)) (NOT (INTACT WHEEL1)))
  (:GOAL (AND (NOT (OPEN TRUNK)) (IN JACK TRUNK) (IN PUMP TRUNK) (IN WHEEL1 TRUNK)
      (IN WRENCH TRUNK) (ON WHEEL2 THE-HUB)))
  (:length (:serial 14) (:parallel 10)))
```

Dodatek C

Obsah CD

Struktura složek a podsložek:

- Implementace graphplanu
  - Příklady - obsahuje předpřipravené úlohy zapsané v souboru txt.
  - Spustitelný soubor - obsahuje binární soubor přeloženého programu.
  - VS project - uložený projekt z visual studia.
- Implementace RTAA*
  - Spustitelný soubor - obsahuje binární soubor přeloženého programu.
  - VS project - uložený projekt ve visual studiu.
  - Vytvořené gridy - obsahuje předpřipravené scény s překážkami.
- Implementace RTAA* - první verze (první verze použitá k visualizaci extremních případů v kapitole 6.5)
  - Spustitelný soubor - obsahuje binární soubor přeloženého programu.
  - Uložené gridy - obsahuje předpřipravené scény s překážkami.
  - VS project - uložený projekt ve visual studiu.
  - Dokumentace.pdf
  - Prezentace.pdf
- Program blackbox
  - Příklady - příklady různých domén, stažené z oficiálních stránek programu.
  - Sputitelný soubor (linux) - binární soubor, spustitelný v systému linux.
- Zdrojové kódy páce v Latexu