VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

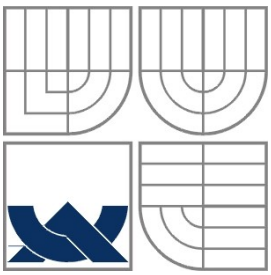
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DEMONSTRACE METOD PLÁNOVÁNÍ

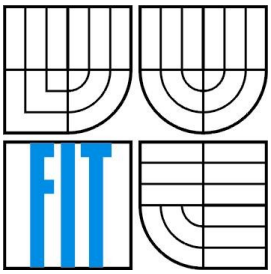
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE ADAM DOSTÁL
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DEMONSTRACE METOD PLÁNOVÁNÍ

DEMONSTRATION OF PLANNING METHODS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

ADAM DOSTÁL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. František Zbořil, Ph.D.

BRNO 2007

Abstrakt

Projekt se zabývá demonstrací moderních plánovacích metod GraphPlan a SATPlan s důrazem na pochopení vnitřních procesů uvnitř jednotlivých algoritmů. Dále diskutuje možnost využití appletů jako doplňku k výkladu algoritmů a využití jednotlivých částí těchto algoritmů i v jiných oblastech umělé inteligence. V druhé části se zabývá applety a jejich využitím při výuce a samostudiu fungování algoritmů.

Klíčová slova

plánování, GraphPlan, SATPlan, java, java applet, metody plánování, plánovací metody

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Keywords

planning methods, GraphPlan, SATPlan, java, java applet

Citace

Jméno Příjmení: Název práce v jazyce práce, bakalářská práce, Brno, FIT VUT v Brně, rok

Demonstrace metod plánování

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením...
Další informace mi poskytli...
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jméno Příjmení
Datum

Poděkování

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc
(externí zadavatel, konzultant, apod.)..

Obsah

1 Úvod

Tato práce se zabývá dvojicí moderních plánovacích algoritmů GraphPlan a SATPlan a možsnotmi na vytvoření učebních pomůcek, které by jejich fungování a jednotlivé výhody vysvětlily studentům vysokých škol.

V první části se zabývá jednotlivými algoritmy, ukazuje jejich princip, možnosti implementace a zvažuje jak vhodně zobrazit jejich průběh a jednotlivé kroky tak, aby to bylo zobrazení přínosem pro studenta a ukázalo mu, jak tyto algorimy fungují.

V druhé části se zabývám možností využívání JAVA appletů ve výuce jako doplněk samostudia a zvažuji jednotlivé problémy a výhody, které může toto využívání přinést.

1.1 Úvod do problematiky plánovacích algoritmů

Automatické plánování je část umělé inteligence, která se zabývá výpočtem plánů a sekvencí operací, které jsou prováděny autonomnímy roboty, neřízenými stroji a inteligentními agenty. Na rozdíl od klasických problémů řízení a klasifikace jsou řešení automatického plánování komplexní, dopředu neznámé a musí být prozkoumány a optimalizovány ve vícerozměrných prostorech.

V neměnných prostředích mohou být plány sestavovány před samotným prováděním, v proměnlivých prostředích musí být často plány průběžně kontrolovány a přizpůsobovány.

Plánem se rozumí přesná sekvence operací nad systémem, které mohou být prováděny paralelně i sériově. Jejich provádění mění jednotlivé stavy systému a umožňují tak z počátečního stavu dosáhnout cílového stavu.

Plánovací algoritmy obvykle zkoumají stavové prostory obsahující různá řešení – některé omezují stavové prostory o řešení, o kterých víme že nevedou k cíli (GraphPlan a jeho modifikace), jiné převádí problémy na saturační problémy (SATPlan a jeho modifikace).

Obvykle se využívá model systém – operace – stav, kdy systém je definován pomocí stavů. Stavy je možné měnit pomocí operací.

Příklad 1:

Systém Bob je definován následujícími stavy:
hladový, žíznivý, není unavený, má peníze, má jídlo, má pití

a může vykonat následující operace: běžet, jíst, pít a spát. Cílem Boba je nebýt hladový ani žíznivý.

Každá operace je definována dvojicí n-tic, které říkají jaké stavy musí systém splňovat před provedením operace a v jaké stavy bude splňovat po provedení operace.
Stav může nabývat hodnot pravda nebo nepravda, složitější implementace algoritmů umožňují používání proměnných – `jdu(odkud, kam),mám(věc).`

Příklad 2:

Robotické rameno má následující úkol:
Krabice jsou položené na pozici [t1], krabice [b] je položena na krabici [a]. Proveď takové operace, ať jsou krabice na pozici [t3] a krabice [b] ať je položena na krabici [a]

Podmínky systému:

on(what, where)	--věc *what* je na pozici *where*
clear(where)	--na pozici *where* nic není
armempty	--značí, že rameno „nic nedrží“
holding(what)	--věc *what* je uchopena ramenem

Počáteční stavy:

```
on(a, t1)
on(b, a)
clear(t2)
clear(t3)
clear(b)
armempty
```

Koncové stavy:

```
on(a, t1)
on(b, a)
```

Stav ostatních podmínek nás nezajímá.

```
Povolené operace jsou
Pickup(X,Y)          --vezmi X z pozice Y
Puton(X,Y)      --polož X na pozici Y

počáteční stavy pro Pickup (X,Y) jsou
on(X,Y)              --X je skutečně tam, odkud jej
chceme vzít
clear(X)             --na X není žádná další krabice
armempty             --rameno je volné

koncové stavy pro Pickup (X,Y) jsou
holding(X)      --rameno drží krabici X
clear(Y)             --pozice Y je nyní volná (=nic na
ní není)

Podobně můžeme určit podmínky Puton(X,Y) jako
ps: holding(X); clear(Y)
ks: on(X,Y); clear(X); armempty

Vidíme, že sekvence požadovaných operací je zdánlivě
velmi jednoduchá (Pickup(b,a); Puton(b,t2); Pickup(a,t1);
Puton(a,t3); Pickup(b,t2); Puton(b,a)).
```

Většina plánovacích algoritmů používá různé modifikace procházení stavového prostoru. Liší se typem stavového prostoru, kde se jedná buď o stavový prostor kde jsou uzly reprezentovány jako plány nebo o stavový prostor, kde jsou uzly reprezentovány jako jednotlivé stavy systému, přičemž obvyklejší je druhý způsob.

Prohledávání může probíhat od počátku k cíli nebo od cíle k počátku. Při prvním způsobu musíme umět rozhodnout které operace jsou přípustné, jak bude vypadat stav systémů po případném provedení akcí a navíc musíme umět rozhodnout, jestli je daný stav cílový. Prohledávání od počátku je korektní a úplná, může být implementována mnoha algoritmy prohledávání stavového prostoru (např. Breath-first search, Uniform cost search, A*). Problémem může být vysoký větvící faktor a z toho plynoucí vysoká paměťová a výkonová náročnost. Paměťová náročnost se dá snížit omezením stavového prostoru vynecháním irelevantních akcí.

Pro prohledávání od cíle k počátku musíme umět vyhledat podcíl a najít akce pro daný cíl. Toto prohledávání je opět korektní a úplné a může být implementováno deterministicky, pokud implementujeme i vyhledávač cyklů. Stále ale může docházet k velkému větvení i když už k menšímu než při prohledávání od počátku k cíli.

1.2 Zástupci plánovacích algoritmů

Typyckým zástupcem plánovacích algoritmů je algoritmus STRIPS (Stanford Research Institute Problem Solver), vyvinutý v 70tých letech pro plánování činnosti robota Shakey, který dodnes slouží k základům plánovacích algoritmů. Mezi další používané algoritmy patří GraphPlan a SATPlan, kterými se zabývá tato práce a například v Hubbleově teleskopu byli použité algoritmy SPSS a Spike.

1.2.1 Srovnání GraphPlanu a SATPlanu

GraphPlan je oproti SATPlanu snáze pochopitelný. Graf nám umožní vyloučit všechny možnosti, do kterých bychom mohli vstopit při běžném slepém procházení stavového prostoru, nakonec ale i on je vlastně procházením stavového prostoru, který byl pouze velmi zajímavou metodou zmenšen. Podobná metoda by se mohla uplatnit i v jiných metodách prohledávání stavového prostoru nebo při řešení problému obchodního cestujícího.

SATPlan vyžaduje znalost i jiné oblasti než jen problematiky plánování a obecněji umělé inteligence – je potřeba znát i základy výrokové logiky a pak teprve můžeme pochopit proč vlastně může SATPlan fungovat. Ten je navíc poměrně abstraktní, protože ve svém běhu nepoužívá model operace – stav, ale obě entity bere jako výrok ve formuli. Samotný překlad problému do CNF je poměrně zdlouhavý a pracný, je potřeba sepsat větší množství překladových pravidel.

Pro SATPlan hovoří řádově vyšší rychlost provádění, proti je naopak složitost implementace.

1.2.2 GraphPlan a SATPlan: dosavadní výuka na FIT

V přednáškových slajdech, které mi poskytl vedoucí práce, je podrobněji probírán pouze GraphPlan – velký důraz je kladen na ukázku rozvíjejícího se grafu a vysvětlení vzájemných vyloučení, nicméně algoritmus získávání plánu z grafu je popsán jen abstraktním algoritmem. Získávání plánu je v podstatě forward-checking prováděný od cíle k počátku, takže to co v algoritmu předchází, ve skutečnosti následuje. Zjištění této

skutečnosti mi trvalo poměrně dlouhou dobu, myslím, že několik obrázků by problematiku osvětlilo poměrně snadno.

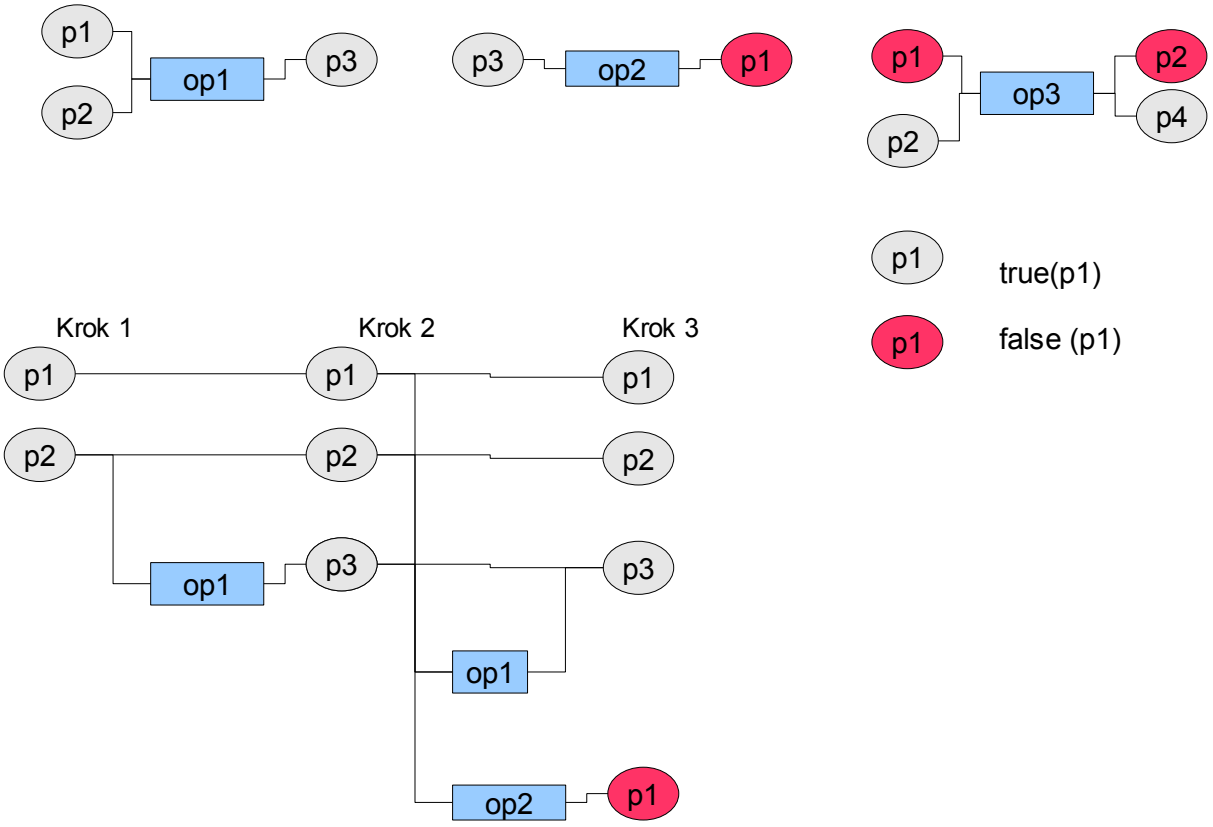
SATPlan není ve slajdech probírán vůbec.

2 Popis algoritmů GraphPlan a SATPlan

2.1 GraphPlan

GraphPlan byl vyvinut v roce 1995 Avrimem Blumem a Merrickem Furstem. Jako vstup mu slouží uspořádaná čtveřice ve stejném tvaru jako pro STRIPS. Graphplanu dal jméno plánovací graf, který je jakýmsi pomocnou strukturou, která redukuje množství stavů, které je třeba prozkoumat v průběhu výpočtu.

Jedná se o stavový prostor, který je posléze prohledáván, kde jednotlivé uzly grafu jsou stavy a operace systému a jednotlivé hrany ukazují dosažitelnost jednotlivých stavů. Uzly grafu jsou střídavě operace a podmínky, které se pravidelně střídají (tzn. hrany od podmínek vedou jen k operacím a naopak, viz obrázek).



Ilustrace 1: Rozvíjení grafu u GraphPlanu

Jak je na obrázku vidět, graf se může velice rychle rozrůstat. Navíc je pro každý stav systému definována perzistentní operace (`žádná operace`), tedy se s daným stavem v konkrétním kroku nemusí stát nic.

GraphPlan ve svém běhu využívá tzv. vzájemných vyloučení – jsou to dvojice operací, které nemohou být prováděny paralelně. Vzájemná vyloučení musí být určena pro každý krok v algoritmu zvlášť.

Známe čtyři druhy vzájemných vyloučení:

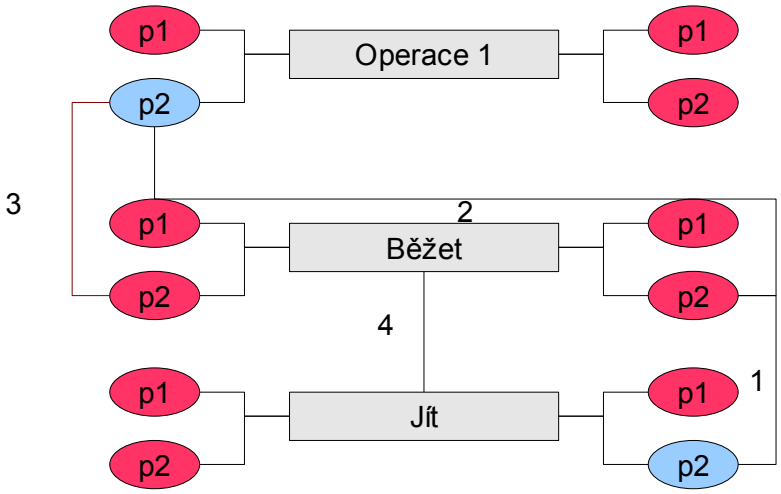
1. Nekonzistentní důsledky

   -důsledek jedné akce je negací důsledku akce jiné.
2. Interference

   -důsledek jedné akce je negací předpokladu pro akci druhou
3. Nekonzistentní předpoklady

   -předpoklad jedné akce je negací předpokladu akce jiné
4.

   -Literál je negací literálu druhého nebo jsou vzájemně vyloučené akce, které mají tyto literály jako efekt.



1: Nekonzistentní následky

2: Interference

3: Nekonzistentní předpoklady

4: Nemohu zároveň jít a běžet

Ilustrace 2: Ukázka vzájemných vyloučení

Poslední částí graphplanu je vyhledávání plánu v příslušném grafu – nejobvyklejší je metoda zpětného forward checking. V něm se využívá parciálních plánů, vlastně jednotlivých kroků potenciálních řešení, kdy v prvním kroku vyhledávání plánu jsou jako cíl určen cílové stavy systému. Pro každý stav jsou v grafu nalezeny všechny operace, kterými lze jednotlivých stavů dosáhnout (graf nám vyloučí ty operace, které sice k cílovým stavům vedou, ale nemohou být použity, protože mohou být dosaženy až v dalších krocích). S pomocí tabulky vzájemných vyloučení zjistíme, kolika různými způsoby může systém získat cílové stavy z předchozícho kroku. Každý způsob musí být označen jako jeden krok potenciálního plánu.

Pokud pro danný cíl nenajdeme žádné řešení, můžeme odstranit z potenciálních plánů všechny kroky, které k tomuto cíli vedou a i všechny kroky před ním, které přijdou o všechny následovníky.

Nyní tedy můžeme v kostce shrnout algoritmus graphplanu:

```
GraphPlan(I, G, O) -- I: počáteční stavy, G: cílové stavy, O:
povolené operace
  expandGraph(); -- expanduj graf
  loop:
      if isGraphSolvable(): -- obsahuje poslední krok grafu
      všechny cílové stavy
            solution = extractSolution(); -- zkusíme vyřešit
            if (solution):
                  return solution; -- pokud řešení existuje,
                  vrátíme řešení
      else if isGraphExpandable():
            expandGraph(); -- pokud jsme řešení nenašli, zkusíme
            znovu expandovat graf
      else
            return fail -- pokud nemáme ani řešení, ani
            nemůžeme expandovat graf, vracíme neúspěch
```

Vzhledem ke složitosti algoritmu extrakce plánu jej uvedu zvlášť:

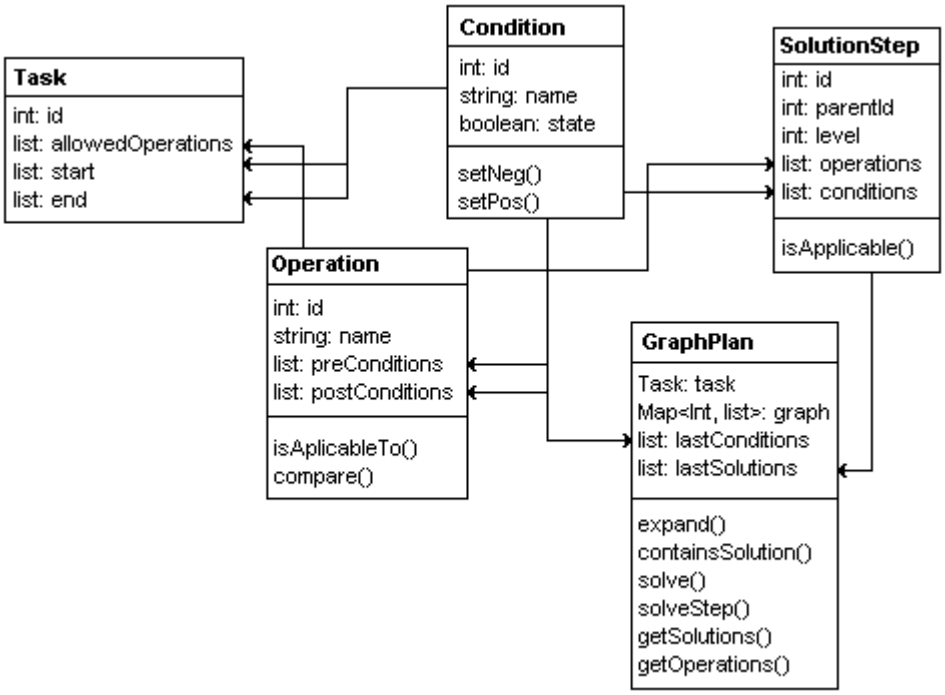
```
solveGraph(Graph G, level) -- level značí počet kroků, který
graf obsahuje
  nastav jako první cíl cíl celého úkolu
  pro všechna level (1)
    pro všechny kroky v příslušném level (2)
        pro všechny cíle v příslušném kroku (3)
           najdi všechny operace, které mají jako následky
           některé nebo všechny stavy v cíli
           pokud nemůžeme vyřešit všechny stavy v cíli, zahoď
           krok
           pro všechny nalezené operace
              sestav kroky pro následující level
           pro všechny kroky pro následující level
              všechny předpoklady kroku nastav jako cíl kroku
              pro následující level.
              Pokud je cíl kroku stejný jako počáteční stav,
              vrať plán
           end (3)
        pokud nejsou žádné kroky, vráť chybu
        end(2)
     end(1)

     vrať chybu -- jsme na počátku grafu, ale žádný plán jsme
        nenalezli
```

Pro implementaci jsem zvolil jazyk JAVA a programovací prostředí NetBeans, protože jsem již s těmito nástroji a programovacím jazykem seznámen, navíc je snadné vytvořené programy umístit na webovou stránku jako JavaApplet.

2.2 Implementace GraphPlanu

2.2.1 Diagram tříd

(diagram neobsahuje třídu použitou pro zobrazování, jedná se pouze o implementaci algoritmu)



Ilustrace 3: Diagram tříd

2.2.2 Popis použitých tříd

Jak je vidět, jedná se o poměrně složitý algoritmus – snad jej lépe vysvětlí následující shrnutí.

Condition.java – stav systému

Operation.java – operace nad systémem

- ArrayList<Condition> preConditions – předpoklady pro operaci
- ArrayList<Condition> postConditions – následky operace

Task.java – úkol v systému

- ArrayList<Operation> operations – povolené operace pro tento úkol
- ArrayList<Condition> start, end – počáteční a cílové stavy systému

SolutionStep.java – jeden krok potenciálního řešení

- int parentId – identifikátor následovníka v konečném plánu – toto pojmenování se může zdát poněkud matoucí, ale zvyšuje přehlednost při implementaci algoritmu, jelikož postupujeme grafem od konce k počátku

- ArrayList<Operation> operations – seznam operací, které jsou potencionálně prováděny během jednoho kroku řešení
- ArrayList<Condition> conditions – seznam stavů systému před prováděním operací – pokud si uvědomíme, jak algoritmus grafem prochází a jak hledá řešení, zjistíme, že je tento seznam úkolem, pro který hledáme řešení v krocích předcházejících této instanci třídy SolutionStep (tento seznam bude cílem předcházejícího kroku)

GraphPlan.java – třída starající se o správný výpočet plánu metodou graphplan

- ArrayList<Condition> lastConditions – seznam stavů systém, používá se při expanzi pro definici všech dosažitelných kroků systému
- Task task – popis systému, pro který hledám plán
- HashMap<Integer, ArrayList<Operation>> graph – graf
- private void go() - spouští algoritmus hledání; spouští funkce expand() v každém kroku pro rozvíjení grafu a containsSolution() pokud obsahuje proměnná lastConditions cílový stav úkolu pro nalezení plánu.
- private void expand() - rozvíjí graf, prochází jednotlivé povolené operace úkolu a zjišťuje, která z nich může být použita. Rozšiřuje obsah lastConditions.
- private Map<Condition, ArrayList<Operation>> getOperations(ArrayList<Condition> target, int lev) – pro každou podmínku z target vrátí seznam operací, kterými ji lze řešit na úrovni lev.
- private ArrayList<SolutionStep> getSolutions(int lev, int parent) – nejdříve si pomocí identifikátoru parent zjistí ze SolutionStep, který v algoritmu předcházel volání této funkce, seznam stavů, který tomuto SolutionStep předcházel. Pak funkcí getOperatoins zjistíme, jak bychom mohli těchto stavů dosáhnout (seznam bude proměnnou target, s kterou se bude funce getOperations volat).

  Jakmile známe operace, kterými dosáhneme kýžených stavů, můžeme začít plnit seznam, který bude návratovou hodnotou. Pro každý SolutionStep zjistíme, které operace řešící jednotlivé podmínky nejsou vzájemně vyloučené v tomto SolutionStep. Pokud je možné použít více než jednu funkci, musíme tento SolutionStep klonovat a zařadit do seznamu, pokud nenalezneme žádnou operaci, kterou bychom mohli dosáhnout konkrétní stav, musíme jej ze seznamu vyjmout.

- private ArrayList<SolutionStep> solve() - postupně projde kroky od konce k počátku a na konci vrátí seznam dosažených SolutionSteps. V každém kroku cyklu volá

funkci getSolutions pro každý SolutionStep (respektive jeho id) a výsledek použije v cyklu dalším. Vrátí kroky, které teroeticky být použity na začátku běhu plánu (ale ještě nevíme, jestli tomu tak skutečně mohlo být).

- private boolean containsSolution() - zavolá funkci solve(). Nad seznamem, který funkce vrátí, zjistí, jestli některý prvek seznamu obsahuje takový SolutionStep, jehož seznam conditions jsou ekvivalentní seznamu start proměnné task této instance třídy GraphPlan. Pokud ano, byl nalezen první krok řešení a další kroky lze získat výpisem kroků navázaných přes parentId.

2.2.3 Prezentace fungování algoritmu

Doposud je v přednáškách předmětu, kde se GraphPlan probíral velmi dobře prezentován jak graf, tak vzájemná vyloučení, ale samotné získávání plánu z grafu je popsáno jen obecným algoritmem a není rozvedeno ani nejsou zváženy možné modifikace GraphPlanu.

V prezentaci fungování algoritmu musíme brát v úvahu potenciálně velký rozsah vykreslovaného grafu a potenciálně velké množství řešení, které jsou v dalších krocích zavrhnuty.

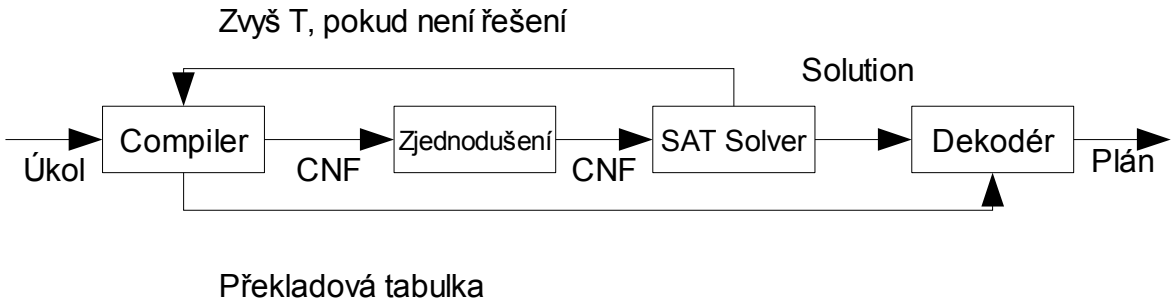
Rozhodl jsem se vykreslovat pouze několik prvních kroků grafu – postupně by došlo ke ztrátě přehlednosti a vznikla by jen rozsáhlá pavučina překrývajících se čar, navíc kombinování „všeho na všechno“ za účelem rozvoje grafu předpokládám stačí naznačit v maximálně pěti krocích.

Vykreslování algoritmu pro vyhledání plánu obsahuje tři zásadní kroky: výběr použitelných operací podle cílů z předchozích kroků, sestavení parciálních plánů a nalezení jednotlivých kroků řešení. Zvolil jsem takové zobrazení, kdy zůstává zachován zobrazený graf.

Jako vhodné úlohy k zobrazování jsem zvolil takové, které nezpůsobují velké větvení grafu do šířky, ale i tak rozvíjejí pravidelně graf a u složitějších příkladů projde v průběhu extrakce plánu i několik slepých potenciálních řešení.

2.3 SATPlan

SATPlan je plánovací algoritmus na zcela jiném principu než GraphPlan – GraphPlan je v podstatě jen prohledávání stavového prostoru, SATPlan úkol převede do tvaru, který je řešitelný výrokovou logikou a ten potom řeší (SAT = propositional satisfability = řešitelný výrokovou logikou). Jednotlivé části algoritmu spolu spolupracují asi takto:



Ilustrace 4: Komponenty SATPlanu

Výroková logika se skládá z množiny proměnných (v našem případě to jsou operace a stavy systému v jednotlivých krocích) a základních operátorů – AND (Δ, konjunkce, logický součin), OR (v, disjunkce, logický součet) a NOT (~, negace). Jednotlivé výrazy se skládají z literálů (což jsou proměnné nebo jejich negace) a základních operátorů. Obvykle se používají ještě operátory implikace (->) a ekvivalence (<->) . Operace AND, OR a ekvivalence jsou komutativní, distributivní a asociativní, ale ekvivalence komutativní není.

Každou formuli výrokové logiky je možné přepsat do normálových forem – konjunktní (CNF) a disjunktní (DNF) normálové formy. Konjukntní normálová forma je konjunkce disjunkcí, disjunktní normálová forma je disjunkce konjunkcí.

```
Příklad: Konjunktní a disjunktí normálová forma:
DNF: (A Δ B) v (B Δ C Δ D) v (~A Δ ~C)
CNF: (A v B) Δ (B v C v D) Δ (~A v ~C)
```

Převod formulí do jednotlivých normálových forem se děje podle přesných pravidel, přičemž musíme dát pozor, aby nedošlo k chybě – můžeme tak znehodnotit celou práci a chybu je velmi těžké dohledat.

Při převodu se nejdříve zbavíme ekvivalencí a implikací podle zákona ekvivalence a implikace, potom podle De Morganových zákonů přeneseme negace od klauzulí k atomům a dokončíme úpravu odstraněním závorek.

```
Příklad: převod formule do CNF
1: (~A -> B) v (~A Δ B) <-> C
2: ((~A->B) v (~A Δ C)->(C)) Δ ((C) -> (~A -> B) v (~A Δ C))
3: ~((~A->B) v (~A Δ C) v (C)) Δ ((~C) v (~A -> B) v (~A Δ C))
4: ~((A v B) v (~A Δ C) v (C)) Δ ((~C) v (A v B) v (~A Δ C))
```

5: (~(A v B) Δ ~(~A Δ C) v (C)) Δ ((~C) v (A v B) v (~A Δ C))

6: ((~A Δ ~B) Δ (A v ~C) v (C)) Δ ((~C) v (A v B) v (~A Δ C))

7: (~A) Δ (~B) Δ (A v ~C v C) Δ (~C v A v B v ~A) Δ C

Každá formule je splnitelná, pokud existuje takové přiřazení jednotlivých literálů tak, že potom celá formule bude ohodnocena jako true.

SAT problém patří do třídy NP-úplných problémů – to jsou takové problémy, které jsou řešitelné v nedeterministickém polynomálním čase. NP problém je takový, který je řešitelný v polynomálním čase. Každý NP problém je řešitelný i jako NP-úplný problém, ale ne naopak. Dá se říci, že NP-úplné problémy jsou ty nejobtížnější problémy ze všech NP problémů. Nalezení jednoho algoritmu pro řešení NP-úplného problému v polynomálním čase by znamenalo, že každý NP-úplný problém je řešitelný v třídě polynomálním čase (Clay Mathematics Institute označil nalezení tohoto algoritmu jako otázku tisíciletí a nabízí za její vyřešení 1 milion amerických dolarů). Mezi další typické NP-úplné problémy patří také problém obchodního cestujícího nebo problém sčítání podmnožin.

Abychom mohli použít pro plánování algoritmus SATPlan, je třeba převést jak počáteční a cílové stavy tak všechny operace ve všech krocích, navíc je třeba definovat i dvojice operace-stav, kde operace nemění hodnotu stavu pro všechny kroky. Pro větší počet kroků tak vzniká velké množství klauzulí.

2.3.1 Algoritmus SATPlanu

`function SatPlan(task, Tmax)` -- funkce vrací plán nebo neúspěch; vstupem je úkol a maximální délka plánu

`for (T=0, T<=Tmax, T++) {` -- iterace

`cnf = Translate To SAT(task, T)` -- překlad úkolu a délka plánu, navíc uloží informace překladu z původní formy na cnf, aby bylo možné posléze extrahovat řešení

`solution = SAT-Solver(cnf)` – libovolný algoritmus pro nalezení řešení zadané formule, např DPLL nebo WalkSAT

`}`

`if (solution) return ExtractSolution`

`return false`

2.3.2 Překlad úkolu

Překlad úkolu je stěžejní část algoritmu. V následujícím textu budu operovat s následujícím zadáním:

Příklad 2: Zadání ve tvaru STRIPS

S: dinner, cleanhands, tidy, tired, hungry

I: ~dinner, cleanhands, ~tired, hungry

G: ~dinner, ~hungry, cleanhands

O:

- cook (PreConditions: ~dinner, ~tired, cleanhands; PostConditions: dinner, ~cleanhands)
- eat (Pre: dinner, cleanhands; Post: ~dinner, ~hungry)
- washHands (Pre: ~cleanhands; Post: cleanhands)

Nyní začneme převod do formy použitelné pro SAT.

Převod I:

Převedeme jednoduše do CNF a přidáme index, kdy tento stav platí

~dinner(0) ∆ cleanhands(0) ∆ ~tired(0) ∆ hungry(0)

Převod G:

Probíhá stejně jako převod I, jen musíme místo nuly dávat T, tedy maximální délku plánu.

~dinner(T) ∆ cleanhands(T) ∆ ~hungry(T)

Převod O:

Převod operací do CNF je již složitější. Musíme definovat kompletní převodní tabulku:

cook(0) → ~dinner(0) ∆ ~tired(0) ∆ cleanhands(0) ∆ dinner(1) ∆ ~cleanhand(1)

cook(1) → ~dinner(1) ∆ ~tired(1) ∆ cleanhands(1) ∆ dinner(2) ∆ ~cleanhand(2)

…

až po:

```
cook(T-1) → ~dinner(T-1)  Δ ~tired(T-1) Δ cleanhands(T-1)
Δ dinner(T) Δ ~cleanhand(T)
```

Podobně musíme definovat pro všechny operace.

Dále je potřeba přeložit tyto definice do CNF – uvedu nejjednodušší příklad pro operaci washHands

```
washHands(0)→ ~cleanHands(0) Δ cleanHands(1) =
~wasHands(0) V (~cleanhands(0) Δ cleanHands(1)) =
(~wasHands(0) V ~cleanHands(0)) Δ (~wasHands(0) V
cleanHands(1)).
```

Tento převod opět musíme provést pro všechny operace.

Dále musíme definovat, která operace nemění stav, např:

```
~dinner(0) Δ washHands(0) → ~dinner(1)
```

Toto musí být opět definováno pro všechny dvojice operace – stav, které na sebe nemají vliv.

Další formuli, kterou musíme sestavit, je taková, která zajistí, že v každém kroku dojde k alespoň jedné operaci:

```
washHands(0) v cook(0) v eat(0)

washHands(1) v cook(1) v eat(1)
```

Kombinace výše uvedených axiomů zajistí, že všechny stavy systému, které nemohou být ovlivněny prováděnými operacemi ovlivněny nejsou v průběhu algoritmu. Praktickým důsledkem celé formule je, že dvě operace nemohou být provedeny v jednom kroku.

Problém u SATPlanu nastává, chceme-li používat i proměnné, protože musíme vypsat všechny kombinace pro všechny proměnné a čas, kdy k operaci může dojít:

```
walk(who, from, to)
who: Anna, Bob, Cecilia
from, to: home, pub, school, work
T: 0-10
```

Musíme vytvořit kombinace:

```
walk – Bob, home(0), pub(1)
walk – Bob, home(1), pub(2)
walk – Bob, pub(0), work(1)
…
```

Celkově tím pro T kroků, O operací, B objektů (v našem případě Anna, Bob a Cecilia) a A jako celkovou [[arity]] akcí získáme T x O x B^A (V našem případě 870 instancí pro jednu operaci). Tomu se dá zabránit rozpisem operací na více operací, kde každá operace může mít jen jednu proměnnou – v našem případě by se jednalo o operace `walkWho, walkFrom, walkTo`.

2.3.3 Sestavení formule

Pokud jsme v pořádku vykonali vše předchozí, můžeme formuli sestavit – vezmeme počáteční stav I, k němu přidáme všechny klauzule, které zajišťují provedení alespoň jedné formule v každém kroku, všechny klauzule, které jsou přepisem operace a koncový stav G.

```
I Δ (krok 1 Δ krok 2 Δ krok 3 Δ … Δ krok n) Δ (přepis
operace 1 (0) Δ přepis operace 1 (1) Δ přepis operace 1(2) Δ …
Δ přepis operace 1 (n) Δ přepis operace 2 (0))  Δ G.
```

Pomocí obecně známých pravidel pak tuto formuli převedeme do CNF, zjednodušíme a vyřešíme jako běžnou formuli. Když nabyde operace s určeným krokem hodnoty true víme, ve kterém kroku k operaci dojde. Tak sestavíme celý plán.

2.3.4 Zjednodušení formule

Ještě před samotným výpočtem můžeme formule zjednodušit

- Samostatná klauzule - taková klauzule, která obsahuje jen jeden literál – ten musí mít tím pádem hodnotu true, jinak nebude mít hodnotu true celá formule.

```
Př: (~A) Δ (B v C) Δ (~A) Δ (B V ~C)
klauzule ~A musí nabývat hodnoty true, jinak nemůže být
formule vyřešena
```

- Úplný literál – takový literál, který může být označen hodnotou true, tak, že zároveň nemůže dojít k nastavení žádné z klauzulí na hodnotu false:

```
Př: (B v C v D) Δ (B v ~C v ~D) Δ (C v D)
-literál B můžeme nastavit na hodnotu true – v žádném případě
tím nevyloučíme, že kterákoliv z klauzulí musí nabývat hodnotu
true jako celek.
Př: (B v C v D) Δ (B v ~C v ~D) Δ (~B v C)
```

-nyní literál B nemůžeme nastavit na hodnotu true, protože tím můžeme (v případě, že C je false) znemožnit nastavení poslední klauzule na hodnotu true.

2.3.5 Algoritmy řešení CNF

Jen v krátkosti shrnu algoritmy pro řešení CNF – liší se základním přístupem k řešení problémů na dvě skupiny:

- Systematické – prochází jednu možnost za druhou, většinou variací backtrackingu
- Náhodné – provádí náhodný průchod všemi možnostmi. Většinou mění hotnotu takového literálu, který přiřadí nejvíce klauzulím hodnotu true.

2.3.6 Problém: jak vhodně a názorně zobrazit všechny kroky SATPlanu

Vzhledem k tomu, že algoritmus SATPlanu je principielně odlišný od algoritmů ostatních, je nutné zamyslet se nad způsobem, jak zobrazit průběh algoritmu – všechny tradiční přístupy zobrazování průběhu algoritmů jsou zde nevhodné.

Zobrazování stromem operací můžeme vyloučit, v průběhu algoritmu nedochází k rozvoji žádného stromu.

Selhává i možnost zobrazovat jednotlivé kroky algoritmu pro jejich velkou komplexnost a složitost.

Chvíli se jako vhodná zdála možnost zobrazovat tabulku pravdivostních hodnot jak je zvykem například při výuce logiky na střední škole. Bohužel se tato metoda ukázala jako nevhodná, protože i u nejjednodušších příkladů docházelo k velké rozměrnosti této tabulky a student v ní velice rychle ztrácel přehled. Musíme si uvědomit, že pro O operací, S stavů a T kroků je možno docílit až $(O \times (T - 1) + S \times (T - 2))^2$ řádků (u operací musíme odečíst jedničku, protože pro T_{max} již operace neprovádíme a u stavů odečítáme dvojku, protože T_0 i T_{max} máme dáno v zadání) a vzhledem k tomu, že je didakticky vhodné zobrazovat pravdivostní hodnoty pro jednotlivé kombinace literálů všechny klauzule formule, musíme počítat pro K klauzulí až K*(T-1) sloupců tabulky. Přesný počet klauzulí lze určit jen obtížně, ale můžeme si udělat obrázek o množství klauzulí pomocí následujícího příkladu:

Př: Operace O. Předpoklady pro O jsou stavy A a B, následky jsou stavy ~A a C. Zapsáno výrokovou logikou:

O(0) -> A(0) Δ B(0) Δ C(1) Δ ~A(1)
~O(0) v (A(0) Δ B(0) Δ C(1) Δ ~A(1))
(~O(0) v A(0)) Δ (~O(0) v B(0)) Δ (~O(0) v C(1)) Δ (~O(0) v ~A(1))

To znamená, že získáme z této poměrně jednoduché operace jsme získali 4 klauzule zapsané v CNF formě. Tento výsledek musíme ještě vynásobit číslem T-1.

Pro 4 podobné operace, T=5 a 6 stavů systému získáme:
$(4 \times (5 - 1) + 6 \times (5-2))^2$ = až 1156 řádků tabulky. Část řádků tabulky sice budeme mít možnost vypustit díky zjednodušování ale i tak se málokdy stane, že je vypuštěno více než 20% řádků.

Proto jsem došel k názoru, že nelze zpracovat krokovatelnou názornou učební pomůcku pro vysvětlení algoritmu SATPlan – algoritmus je natolik abstraktní, že k jeho vysvětlení je vhodné přesně popsat jednotlivé kroky sestavování formule a přepis operací do CNF slovy a krátkými příklady. Toto svoje tvrzení jsem si ověřil u skupiny šesti studentů informatických oborů (tři z VUT FIT a tři z MU FI), kterým jsem se snažil nejdříve vysvětlit SATPlan jen pomocí dvojice příkladů a slajdů, které se podobaly přednáškovým slajdům používaným na naší fakultě a potom jsem jim jednotlivé kroky ukázal a okomentoval každý svůj krok. Zatímco první metoda většinou postačovala, u tohoto algoritmu byla vždy neúspěšná.

Zvolil jsem takové příklady, kde je dobře vidět, jak algoritmus funguje. Kladl jsem přitom důraz na nevelký počet kroků (tři a méně) a nepříliš složité operace (jejich předpoklady a následky neobsahují více než tři prvky). Jeden ukázkový příklad obsahuje operaci, kterou nevyužije.

2.3.7 Hlavní výhody SATPlanu

Díky tomu, že SATPlan pracuje v rámci základních logických operací AND a OR, je možné jej implementovat již na úrovni hardware a tím může dosáhou bezkonkurenčních výsledků (ještě lepších než dosahuje při běžné kvalitní implementaci).

3 Využití appletů ve výuce

Z vlastních i cizích zkušeností jsem zjistil, že studentům při samostudiu velmi prospívá možnost procházet jednotlivé algoritmy vlastním tempem, sledovat změny různých seznamů a kontrolovat výpočty na větším množství příkladů – běžné slajdy nebo přednášky toto obvykle neposkytují, tam je probírán většinou jen jeden nebo dva příklady. Případná cvičení zase neposkytují možnost sledovat děj vlastním tempem, navíc mezi cvičením a zkouškou může uběhnout delší doba a student může některé detaily algoritmu zapomenout.

3.1 Výhody appletů

Hlavní výhodou appletů je jejich jednoduchá a masová distribuovatelnost a kompatibilita. Navíc v případě, že chceme ukazovat konkrétní úlohy, u kterých není možnost změny zadání, můžeme implementaci omezit na správné zobrazování jednotlivých kroků.

3.2 Problémy implementace appletů

Největším problémem v implementaci výukových appletů je stanovení vhodných omezovacích podmínek – většinou jsou individuální pro každý probíraný problém. V mých appletech jsem řešil hlavně problém, zda umožnit studentům používat proměnné v definici úkolu a zda umožnit vkládání problémů v predikátové logice. Nakonec jsem obojí zavrhl, hlavní úlohou výukových appletů by měla být názornost a důraz na pochopení principů algoritmů.

Hlavním problémem implementace appletů je vhodné zobrazování jednotlivých kroků, možnost pohybovat se v jednotlivých krocích vpřed i zpět a vhodně graficky znázornit všechny vztahy, které mají na chod algoritmu vliv.

Otázkou ovšem zůstává, jak řešit rozdíly jednotlivých algoritmech a vhodnost výběru demonstračních úloh – některé algoritmy (namátkou alfabeta, minimax, plánování, and/or grafy) umožňují poměrně jednoduchou úpravu jednotlivých úkolů. U jiných algoritmů (např. algoritmy prohledávání stavového prostoru) se vychází spíše z konkrétních úloh – kanibalové a misionáři, úloha dvou džbánů apod. Zde se nabízí prostor spíše pro srovnávání grafů, stromů a různých uspořádaných seznamů a zásobníků různých algoritmů vždy pro jednu konkrétní úlohu.

Ovšem, jak ukázal problém SATPlanu, k vysvětlení některých algoritmů jsou JAVA applety nevhodné.

4 Závěr

V této práci jsem se snažil na konkrétním příkladu dvou plánovacích metod objasnit, v čem může být přínos využívání výukových programů a utilit studenty při přípravě na zkoušky a při implementaci jednotlivých algoritmů v rámci rozsáhlejších projektů. Ukazuje, jak možnost krokovat algoritmus a v každém kroku zobrazovat jednotlivé proměnné algoritmu prospívá při snaze pochopit jak který algoritmus funguje.

V implementaci a výběru ukázkových příkladů jsem se opíral o vlastní zkušenosti při vysvětlování jednotlivých algoritmů svým spolužákům a o běžné problémy, na které jsme naráželi. Ovšem u algoritmu SATPlan jsem nenalezl vhodnou metodu jak zobrazovat postup algoritmu – jedná se o algoritmus velmi abstraktní a obávám se, že není možné zobrazovat jeho průběh. K pochopení, jak algoritmus funguje je třeba mít již zkušenosti s plánovacími algoritmy i s výrokovou logikou. Naučit se jak algoritmus funguje (tedy postup, jak jej používat) je poměrně jednoduché, ale pochopit proč je velmi náročné.

V další části jsem se zamyslel nad možností využívat applety jako prostředky k samostudiu obecně. Fungování většiny algoritmů lze názorně osvětlit pomocí rozvíjených stromů, grafů, zobrazováním hodnot jednotlivých seznamů, zásobníků a proměnných.

Myslím, že se jedná o lepší možnost jak se při samostatné přípravě naučit či pochopit fungování jednotlivých algoritmů díky možnosti pozorvoat rozdíly při procházení či srovnávání jednotlivých algoritmů (či variant algoritmů) a jednotlivých příkladů. Navíc je možné přidávat vlastní algoritmy a tak si ověřit správnost vlastních předpokladů a postupů.

U takových appletů či utilit není důležitá rychlost či datová nenáročnost, ale názornost.

V přípravě učebních pomůcek je důležité mít odezvu od studentů, zda jim konkrétní použité zobrazení pomohlo. V případě, že se zvolené zobrazování neukáže jako vhodné, je třeba použít zobrazování jiné nebo se smířit se skutečností, že konkrétní algoritmus nelze tímto způsobem vysvětlit.

Zajímavým projektem do budoucna by mohl být pokus o řešení dnes neřešitelných problémů pomocí modifikace těchto dvou algoritmů – především SATPlan je omezující jen ve formě přepisů problémů do výrokové logiky, jinak se jedná o extrémně rychlý způsob řešení problémů.