VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

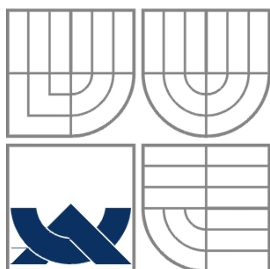
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GRAFICKÉ ANIMACE METOD ŘEŠENÍ ÚLOH

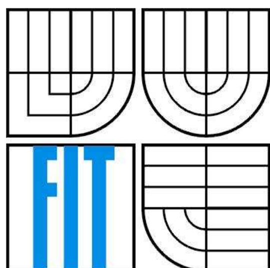
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JIŘÍ MACEK
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

GRAFICKÉ ANIMACE METOD ŘEŠENÍ ÚLOH

GRAPHIC ANIMATION OF PROBLEM SOLVING METODS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JIŘÍ MACEK
AUTHOR

VEDOUCÍ PRÁCE doc. Ing. FRANTIŠEK V. ZBOŘIL, CSc.
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Macek Jiří**
Obor: Informační technologie
Téma: **Grafické animace metod řešení úloh**
Kategorie: Umělá inteligence

Pokyny:

1. Prostudujte metody řešení úloh.
2. Navrhněte demonstrační program pro grafické animace činnosti jednotlivých metod včetně zobrazování příslušných seznamů, se kterými pracují.
3. Navržený program implementujte.
4. Proveďte experimenty.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Podle zadání vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Zpráva o stavu analytické a přípravné části řešení bakalářské práce.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zbořil František, doc. Ing., CSc.**, UITS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
L.S.
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení:	**Jiří Macek**
Id studenta:	84258
Bytem:	9.května 35, 789 63 Ruda nad Moravou
Narozen:	18. 09. 1984, Šumperk

(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....

(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP:	Grafické animace metod řešení úloh
Vedoucí/školitel VŠKP:	Zbořil František V., doc. Ing., CSc.
Ústav:	Ústav inteligentních systémů
Datum obhajoby VŠKP:	....

VŠKP odevzdal autor nabyvateli v:

tištěné formě	počet exemplářů: 1
elektronické formě	počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   ☒ ihned po uzavření této smlouvy
   ☐ 1 rok po uzavření této smlouvy
   ☐ 3 roky po uzavření této smlouvy
   ☐ 5 let po uzavření této smlouvy
   ☐ 10 let po uzavření této smlouvy
   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: 15.4.2007

....
Nabyvatel Autor

Abstrakt

Pro automatizované řešení problémů výpočetní technikou se používají různé implementace umělé inteligence. Tato práce se zabývá některými typickými metodami, popisuje jejich vlastnosti, porovnává je a uvádí možný způsob algoritmizace a implementace. Cílem je vytvoření aplikace, která názorným způsobem demonstruje na vybraných úlohách metody jejich řešení.

Klíčová slova

Umělá inteligence, metody řešení úloh, neinformované metody, metoda prohledávání do šířky, metoda stejných cen, metoda prohledávání do hloubky, metoda omezeného prohledávání do hloubky, metoda postupného zanořování do hloubky, metoda zpětného navracení, metoda obousměrného prohledávání, informované metody, metoda založená na výběru nejlépe ohodnoceného stavu, metoda lačného prohledávání, A*, metody lokálního prohledávání, metoda stoupání do kopce, metoda simulovaného žíhání, metody s omezujícími podmínkami, metoda zpětného navracení pro CSP, metoda dopředné kontroly, metoda minimálního konfliktu, jednoduché hry, prohledávání AND/OR grafu, složité hry, Alfa-Beta řezy.

Abstract

There are many kinds of implementation artificial intelligence for automatic solving problems by computer technology. The main topics of this bachelor’s thesis are some typical methods, describing of their features, comparing them among and shows some useful techniques of algoritmization and implementation too. Main purpose of this thesis is creating application, which clearly demonstrates at chosen problems methods of their solving.

Keywords

Artificial intelligence, problem solving methods, Uninformed Search, Breadth First Search, Depth First Search, Uniform Cost Search, Depth Limited Search, Iterative deeping DFS, Backtracking, Bidirectional BFS, Informed Search, Best First Search, Greedy search, A* search, Local search, Hill climbing, Simulated annealing, Constraint Satisfaction Problem, Backtracking for CSP, Forward checking, Min-conflict, AND/OR, Alfa-Beta cutoff.

Citace

Jiří Macek: Grafické animace metod řešení úloh, bakalářská práce, Brno, FIT VUT v Brně, 2007

Grafické animace metod řešení úloh

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením doc. Ing. Františka Vítězslava Zbořila, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jiří Macek
15.5.2007

Poděkování

Na prvním místě bych rád poděkoval vedoucímu své práce, doc. Ing. Františkovi V. Zbořilovi CSc., jenž mi umožnil, aby tato práce vznikla a který je také spoluautorem učebního textu [1], který mi sloužil jako výchozí informační materiál.

Dále bych rád poděkoval všem, kteří se mnou měli během mé tvorby trpělivost a tím mi pomohli k dokončení tohoto projektu.

© Jiří Macek, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Úvod 3

Cíl 3

Stručný obsah 3

Prekvizity 3

1 Vymezení pojmů 4

1.1 Úloha 4

1.1.1 Úloha dvou džbánů (*Two water jugs problem*) 4

1.1.2 Hlavolam „8“ (*8 puzzle*) 4

1.1.3 Cesta z A do B 5

1.1.4 Úloha osmi dam 5

1.2 Hra 5

1.2.1 hra Zápalky 5

1.3 Stavový prostor 5

1.3.1 AND/OR graf 5

1.4 Metoda řešení úlohy 6

1.5 Průchod stavovým prostorem 6

1.6 Shrnutí 6

2 Popis metod řešení úloh 7

2.1 Neinformované metody 7

2.1.1 Metoda prohledávání do šířky (*Breadth First Search*) 7

2.1.2 Metoda stejných cen (*Uniform Cost Search*) 9

2.1.3 Metoda prohledávání do hloubky (*Depth First Search*) 10

2.1.4 Metoda omezeného prohledávání do hloubky (*Depth Limited Search*) 12

2.1.5 Metoda postupného zanořování do hloubky (*Iterative Deeping Search*) 13

2.1.6 Metoda zpětného navracení (*Backtracking*) 14

2.1.7 Metoda obousměrného prohledávání (*Bidirectional BFS search*) 16

2.2 Informované metody 18

2.2.1 Metoda založená na výběru nejlépe ohodnoceného stavu (*BestFS - Best First Search*) 18

2.2.2 Metoda lačného prohledávání (*Greedy Search*) 18

2.2.3 Metoda A* (*A**) 18

2.3 Metody lokálního prohledávání 19

2.3.1 Metoda stoupání do kopce (Hill Climbing) 19

2.3.2 Metoda simulovaného žíhání (*Simulated annealing*) 20

2.4 Metody řešení úloh s omezujícími podmínkami (CSP) 21

2.4.1 Metoda zpětného navracení pro CSP (*Backtracking for CSP*) 21
2.4.2 Metoda dopředné kontroly (*Forward checking*) 22
2.4.3 Metoda minimálního konfliktu (Min-conflict) 24
2.5 Jednoduché hry 25
2.5.1 Prohledávání AND/OR grafu 25
2.6 Složité hry 26
2.6.1 Alfa-Beta řezy (*Alpha-Beta Cutoff*) 26
2.7 Shrnutí 28
3 Grafické animace metod řešení úloh 29
3.1 Implementační prostředí 29
3.2 Teoretický návrh aplikace 29
3.3 Struktura zdrojových kódů 29
3.3.1 main.cpp 29
3.3.2 main.h 30
3.3.3 functions.h 30
3.3.4 manage.h 30
3.3.5 xprint.h, xmetods.h 30
3.4 Systém názvů proměnných a funkcí 31
3.5 Implementace uživatelského rozhraní 31
3.5.1 Inicializace programu 31
3.5.2 Uživatelský vstup 31
3.5.3 Zpracování vstupu 32
3.5.4 Zobrazení menu 32
3.5.5 Fonty 32
3.6 Implementace jednotlivých úloh 33
3.6.1 Úloha dvou džbánů 34
3.6.2 Hlavolam „8“ 35
3.6.3 Cesta z A do B 36
3.6.4 Úloha 8 dam 37
3.6.5 Zápalky 38
3.6.6 Alfa Beta prořezávání 39
3.7 Závěr 41
3.8 Autorství zdrojových kódů 41

Úvod

Již od pradávna se člověk potýkal s problémy a jejich řešením. S nástupem informačních technologií se mu však otevřela cesta, díky které je možné řešit problémy, o jejichž řešení se mu dříve ani nesnilo, popřípadě tyto problémy bylo schopno vyřešit pouze několik lidí na světě. Ovšem myšlení člověka a stroje, respektive počítače, je značně odlišné. Pokud chceme s pomocí počítače vyřešit nějaký problém, musíme tento problém přesně logicky specifikovat a zadat možné postupy, které by mohly vést k jeho řešení. Výhoda počítače spočívá právě v tom, že může rychle a přesně aplikovat tyto postupy, tedy kvalitně prohledat stavový prostor daného problému a nalézt řešení. A právě pro zadávání problému a prohledávání jeho stavového prostoru byly navrženy metody řešení úloh.

Cíl

Cílem tohoto projektu je vytvoření textu a aplikace, která by názorným způsobem pomohla zájemcům o tuto problematiku pochopit metody řešení úloh.

Stručný obsah

Tato bakalářské práce se zabývám grafickým znázorněním metod řešení úloh. Práce sestává ze tří částí. V první části jsou vymezeny pojmy, které se v textu používají. V části druhé jsou uvedeny výsledky ročníkového projektu, tj. teoretický popis jednotlivých úloh, metod, zařazení a popis algoritmů těchto metod. Ve třetí části je pak popsáno, jak jsou tyto úlohy a metody jejich řešení implementovány do grafické aplikace, která je součástí této práce.

Prekvizity

Na následujících stranách jsou používány pojmy z programování, C/C++, umělé inteligence a algoritmizace. Pro pochopení textu by měl čtenář mít alespoň základní znalosti z těchto oblastí. Pokud se v textu vyskytne ne úplně obvyklý výraz, bude patřičně vysvětlen. Většina termínů je v češtině. Ne všechny anglické termíny však mají jednotný český překlad. Pokud se nějaký takový termín v textu objeví, bude pro přesnost uveden i jeho anglický ekvivalent. Typicky se jedná o názvy úloh a metod jejich řešení.

Pokud čtenáři vznikne nějaká nejasnost, nechť se pokusí vysvětlení najít v textu [1], na jehož základech stojí uvedené algoritmy a terminologie.

1 Vymezení pojmů

Tato kapitola obsahuje vymezení základních pojmů, které se vyskytují v textu. Dále obsahuje popis jednotlivých úloh, které jsou pro demonstraci metod použity.

1.1 Úloha

Úloha je v podstatě definovaný stav určitého prostředí, na nějž se vážou operátory, které tento stav mohou měnit. Všechny možné stavy dané úlohy vytváří stavový prostor. Speciálními případy těchto stavů, jsou stav počáteční a stav cílový. Počáteční stav je většinou přesným popisem prostředí, zatímco cílový stav může být například popsán podmínkou, která se váže pouze na určitý aspekt stavu. Mezi stavy lze přecházet aplikováním operátorů. Pokud aplikuji na stav operátor, posunu se tím ve stavovém prostoru. Řešení úlohy lze tedy vyjádřit posloupností operátorů. Tato soustava určuje cestu stavovým prostorem od stavu počátečního, ke stavu cílovému.

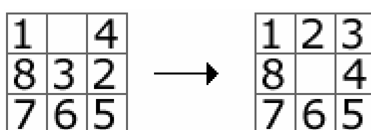
Níže zmíněné úlohy slouží spíše k demonstraci metod řešení, než že by byly skutečnou příčinou vzniku těchto metod. Většina z nich je relativně jednoduchá a v reálném čase řešitelná člověkem. Ovšem metody, které jsou pomocí těchto úloh demonstrovány, se dají použít i pro řešení složitějších problémů.

1.1.1 Úloha dvou džbánů (*Two water jugs problem*)

Mějme dva džbány o různém obsahu. Pro potřeby tohoto textu uvažujme pro jednoduchost obsah první nádoby 4 litry a obsah druhé nádoby 3 litry. Dále máme k dispozici zdroj vody. Cílem úlohy je postupným plněním, vyléváním a přeléváním vody ve džbánech odměřit přesně 2 litry vody.

1.1.2 Hlavolam „8“ (*8 puzzle*)

Mějme matici 3*3. V matici je umístěno náhodně 8 číslic od jedné do osmi, přičemž jedno políčko matice je prázdné. Cílem úlohy je prohazováním prázdného políčka se sousedními pozicemi dosáhnout předem určeného stavu.



Obrázek 1.1 - 8 puzzle

1.1.3 Cesta z A do B

Je dán seznam míst a možností přechodů mezi těmito místy. Cílem úlohy je najít cestu z určeného místa A do místa B. Modifikací této úlohy může být, že cesta musí být nejkratší, popřípadě se hledá místo, které je nejblíže místu B.

1.1.4 Úloha osmi dam

Cílem úlohy je rozestavit na šachovnici 8 šachových dam tak, aby se navzájem neohrožovaly. Modifikace této úlohy spočívá ve změně velikosti šachovnice.

1.2 Hra

Pod pojmem hra zde budeme uvažovat deskovou hru pro dva hráče, která má pevně daná pravidla a konečnou množinu operátorů pro změnu stavu hrací desky.

Hry si dále rozdělíme na jednoduché a složité. Jednoduchá hra je taková, u které v reálném čase můžeme prohledat celý její stavový prostor. Složitá hra je naopak tak rozsáhlá, že to možné není.

1.2.1 hra Zápalky

Jedná se o jednoduchou hru. V základní verzi je počet zápalek na stole 7. Hráči postupně odebírají 1-3 zápalky. Ten kdo odebere poslední zápalku, vyhrál. Modifikace této hry spočívá ve změně počátečního počtu zápalek a ve změně možného počtu zápalek k odebrání.

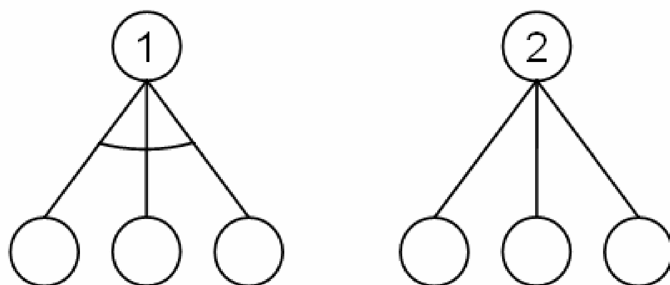
1.3 Stavový prostor

Pro znázornění stavového prostoru se nejlépe hodí reprezentace orientovaným stromem. Uzly tohoto stromu představují jednotlivé stavy úlohy a přechody mezi nimi aplikování operátorů. Kořenový uzel je stavem počátečním. Listové uzly reprezentují konečné stavy (stavy ze kterých se nedá posunout dále stavovým prostorem). Cesta od kořenového uzlu k uzlu cílovému představuje řešení úlohy. Další použité termíny: hloubka stromu, předchůdce (rodič), následník (potomek), bezprostřední předchůdce, bezprostřední následník.

1.3.1 AND/OR graf

AND/OR graf je speciálním případem stromu. Skládá se z uzlů AND a OR pro které platí:

- uzel AND je řešitelný, pokud jsou řešitelní všichni jeho následníci
- uzel OR je řešitelný, pokud je řešitelný alespoň jeden jeho následník.



Obrázek 1.2 – Grafické znázornění uzlů AND (1) a OR (2)

1.4 Metoda řešení úlohy

Pokud je úloha řešitelná, nalézá se její řešení v jejím stavovém prostoru. Prohledáváním tohoto stavového prostoru lze toto řešení nalézt. Metody řešení úloh se od sebe liší systémem, kterým tento stavový prostor prohledávají. Metoda řešení úlohy je tedy systém, respektive algoritmus, pro prohledávání stavového prostoru dané úlohy.

1.5 Průchod stavovým prostorem

Když metody řešení úloh procházejí stavovým prostorem, v podstatě ho generují, respektive jeho část. Obecně lze průchod stavovým prostorem popsat takto:

1. expanduj aktuální uzel
2. všechny jeho vygenerované uzly ulož jako jeho následníky
3. vyber následující uzel a označ ho jako aktuální
4. vrať se na bod 1

Jednotlivé metody řešení úloh se liší v bodě 2 a 3. Tedy kolik a jaké uzly se budou generovat a jaký uzel se vybere jako aktuální.

1.6 Shrnutí

Tato kapitola měla za cíl seznámit čtenáře s nejvýznamnější pojmy, které jsou nutné pro pochopení textu.

2 Popis metod řešení úloh

V této kapitole jsou představeny všechny vybrané metody, které jsou implementovány v aplikaci grafických animací těchto úloh.

2.1 Neinformované metody

Neinformované metody jsou založeny na slepém prohledávání stavového prostoru. Neexistuje u nich žádná funkce, která by odhadovala ještě neprozkoumaný stavový prostor a určovala nejlepší směr prohledávání k cílovému stavu. Jejich výhodou oproti informovaným metodám však je, že jelikož tuto funkci nemají, může jejich použití znamenat, že budou časově méně náročnější než metody, které musí ohodnocovací funkci počítat.

Pro matematickou reprezentaci paměťové a časové náročnosti jsou použity následující zkratky:

O počet operátorů
b faktor větvení (branching factor) – průměrný počet bezprostředních následníků
d hloubka nejlepšího řešení pro danou metodu
k koeficient podílu ceny nejlepšího řešení a nejnižšího přírůstku ceny
m maximální prohledávaná hloubka
l maximální povolená hloubka prohledávání

2.1.1 Metoda prohledávání do šířky (*Breadth First Search*)

Úplnost – ANO
Optimálnost – ANO
Časová náročnost: O(b^(d+1))
Paměťová náročnost: O(b^(d+1))
Implementované úlohy: Dva džbány, 8-puzzle

Algoritmus:

1. Inicializuj frontu Open a seznam Closed a do fronty Open vlož počáteční uzel.
2. Pokud je fronta Open prázdná, ukonči prohledávání jako neúspěšné.
3. Vyjmi z fronty Open uzel a vlož ho do seznamu Closed
4. Pokud je právě vyjmutý uzel uzlem cílovým, vypiš cestu k řešení a ukonči prohledávání jako úspěšné.
5. Expanduj vyjmutý uzel, a všechny jeho bezprostřední následníky, kteří nejsou ani ve frontě Open, ani v seznamu Closed, vlož do fronty Open a vrať se na bod 2.

Pseudokód:

```
bool BFS (first_nod) {

1. InitQ(Open);
   InitL(Closed);
   Insert(Open,first_nod)

2. while (NotEmpty(Open))){
3.    Remove(Open,exp_nod);
      Add(Closed,exp_nod);
4.    if(Goal(exp_nod)) {
          PrintPath(exp_nod);
          return true;
      }
5.    Expand(exp_nod,Open,Closed);
   }
   return false;
}
```

`InitQ` – inicializuje frontu

`InitL` – inicializuje seznam

`Insert/Remove` – vloží/vyjme uzel do/z fronty

`Add` – přidá do seznamu uzel

`NotEmtpy` – vrací true, pokud není fronta prázdná

`Goal` – vrací true, pokud je uzel cílový

`PrintPath` – vytiskne cestu od cílového k počátečnímu uzlu (resp. naopak)

`Expand` – expanduje uzel, a jeho bezprostřední následníky, kteří nejsou ani v Open, ani v Closed vloží do fronty Open

Popis:

Metoda BFS je sice úplná a optimální, nicméně jelikož nepoužívá žádnou heuristiku a prohledává kompletní stavový prostor mezi počátečním a cílovým uzlem, její paměťová a časová náročnost ji dělá pro složitější příklady nepoužitelnou. Nicméně je to základní metoda, na které se dá dobře demonstrovat fungování prohledávání stavového prostoru a problémy s tím spojené.

Výše uvedený algoritmus je podroben několika úpravám, oproti čistému základu metody BFS. Obsahuje především seznam Closed, díky němuž nedochází k opakovanému generování stejných uzlů a tím ještě větší paměťové a časové náročnosti.

2.1.2 Metoda stejných cen (*Uniform Cost Search*)

Úplnost – ANO

Optimálnost – ANO

Časová náročnost: O(b^k)

Paměťová náročnost: O(b^k)

Implementované úlohy: Cesta z A do B

Algoritmus:

1. Sestroj seznamy Open a Closed. Do seznamu Open umísti počáteční uzel..
2. Pokud je seznam Open prázdný, ukonči prohledávání jako neúspěšné.
3. Vyjmi ze seznamu Open uzel s nejlepším ohodnocením a vlož ho do Closed.
4. Pokud je právě vyjmutý uzel uzlem cílovým, vypiš cestu k řešení a ukonči prohledávání jako úspěšné.
5. Expanduj vyjmutý uzel, a všechny jeho bezprostřední následníky,kteří nejsou v seznamu Closed, vlož do fronty Open. Pokud se nějaký uzel vyskytuje ve seznamu Open vícekrát, ponech v něm pouze ten s nejlepším ohodnocením a vrať se k bodu 2.

Pseudokód:

```
bool UCS (first_state) {
1. InitL(Open);
   InitL(Closed);
   Add(Open,first_nod);

2. while(NotEmpty(Open)) {

3.   RemoveBest(Open,exp_nod);
     Add(Closed,exp_nod);

4.   if(Goal(exp_nod)) {
       PrintPath(exp_nod);
       return true;
     }
5.   ExpandBest(exp_nod,Open,Closed);
   }
   return false;
}
```

`InitL` – inicializuje seznam

`Add` – přidá uzel do seznamu

`NotEmtpy` – vrací true, pokud není fronta prázdná

`RemoveBest` – vyjme ze seznamu uzel s nejlepším ohodnocením

`Goal` – vrací true, pokud je uzel cílový

`PrintPath` – vytiskne cestu od cílového k počátečnímu uzlu (resp. naopak)

`ExpandBest` – expanduje uzel, a jeho bezprostřední následníky, kteří nejsou v seznamu Closed vloží do fronty Open; pokud se nějaké uzly vyskytují ve frontě Open vícekrát, ponechá v ní jen ten s nejlepším ohodnocením.

Popis:

Hlavní rozdíl metody UCS oproti metodě BFS je ten, že se při výběru uzlu k expanzi orientuje podle cen přechodů, respektive podle cen cest. Tím pádem je tato metoda použitelná právě v úlohách, kde mají přechody ohodnocení a cesta kterou v těchto úlohách nalezne je optimální.

Výše uvedený algoritmus opět používá seznamu Closed, aby se zabránilo znovuprohledávání stavů, popřípadě zacyklení celé metody.

2.1.3 Metoda prohledávání do hloubky (*Depth First Search*)

Úplnost – NE

Optimálnost – NE

Časová náročnost: O(b^m)

Paměťová náročnost: O(bm)

Implementované úlohy: Dva džbány

Algotritmus:

1. Inicializuj zásobník Open a vlož do něj počáteční uzel.
2. Pokud je zásobník Open prázdný, ukonči prohledávání jako neúspěšné.
3. Vyjmi ze zásobníku Open uzel.
4. Pokud je právě vyjmutý uzel uzlem cílovým, vypiš cestu k řešení a ukonči prohledávání jako úspěšné.
5. Expanduj vyjmutý uzel, a všechny jeho bezprostřední následníky, kteří nejsou v zásobníku Open a ani nejsou předky tohoto uzlu, vlož do zásobníku Open a vrať se na bod 2.

Pseudokód:

```
bool DFS (first_nod) {
1. InitS(Open);
   Push(Open,first_nod)

2. while (NotEmpty(Open))){

3.    Pop(Open,exp_nod);

4.    if(Goal(exp_nod)) {
        PrintPath(exp_nod);
        return true;
      }
5.    Expand(exp_nod,Open);
   }
   return false;
}
```

`InitS` – inicializuje zásobník

`Push/Pop` – vloží/vyjme uzel do/ze zásobníku

`NotEmtpy` – vrací true, pokud není fronta prázdná

`Goal` – vrací true, pokud je uzel cílový

`PrintPath` – vytiskne cestu od cílového k počátečnímu uzlu (resp. naopak)

`Expand` – expanduje uzel, a jeho bezprostřední následníky, kteří nejsou v Open a ani nejsou předky tohoto uzlu, vloží do zásobníku Open

Popis:

Hlavní výhodou metody DFS je lineární paměťová náročnost. Pokud se však při generování nových uzlů neprovede porovnání s předky expandovaného uzlu, může metoda jednoduše selhat, díky tomu že se zacyklí.

2.1.4 Metoda omezeného prohledávání do hloubky (*Depth Limited Search*)

Úplnost – NE

Optimálnost – NE

Časová náročnost: O(b^l)

Paměťová náročnost: O(bl)

Implementované úlohy: Dva džbány

Algotritmus:

1. Inicializuj zásobník Open a vlož do něj počáteční uzel s přiřazenou hloubkou 0.
2. Pokud je zásobník Open prázdný, ukonči prohledávání jako neúspěšné.
3. Vyjmi ze zásobníku Open uzel.
4. Pokud je právě vyjmutý uzel uzlem cílovým, vypiš cestu k řešení a ukonči prohledávání jako úspěšné.
5. Pokud je hloubka vyjmutého uzlu menší, než maximální povolená hloubka, expanduj vyjmutý uzel a všechny jeho bezprostřední následníky, kteří nejsou v zásobníku Open a ani nejsou předky tohoto uzlu, vlož do zásobníku Open a přiřaď jim hloubku o jedna větší.
6. Vrať se na bod 2.

Pseudokód:

```
bool DLS (first_nod, max_depth) {
1. InitS(Open);
   first_nod->depth = 0;
   Push(Open,first_nod)

2. while (NotEmpty(Open))){

3.   Pop(Open,exp_nod);

4.   if(Goal(exp_nod)) {
       PrintPath(exp_nod);
       return true;
     }
```

```
5.   if (exp_nod->depth < max_depth) {
        Expand(exp_nod,Open);
     }
6. }
   return false;
}
```

`InitS` – inicializuje zásobník

`Push/Pop` – vloží/vyjme uzel do/ze zásobníku

`NotEmtpy` – vrací true, pokud není fronta prázdná

`Goal` – vrací true, pokud je uzel cílový

`PrintPath` – vytiskne cestu od cílového k počátečnímu uzlu (resp. naopak)

`Expand` – expanduje uzel a jeho bezprostřední následníky, kteří nejsou v Open, vloží do Open s přiřazenou hloubkou o jedna větší, než měl daný uzel.

Popis:

Metoda DLS je použitelná v případě, že můžeme odhadnout maximální hloubku řešení. Tím, že prohledávání stavového prostoru omezíme touto hloubkou, zabráníme zacyklení metody.

2.1.5 Metoda postupného zanořování do hloubky (*Iterative Deeping Search*)

Úplnost – NE

Optimálnost – NE

Časová náročnost: O(b^d)

Paměťová náročnost: O(bd)

Implementované úlohy: Dva džbány

Algotritmus:

1. Nastav hloubku na 1.
2. Zavolej funkci DLS pro počáteční stav a aktuální hloubku. Pokud skončí úspěchem, ukonči prohledávání jako úspěšné.
3. Pokud aktuální hloubka dosáhla maximální povolené hloubky, ukonči prohledávání jako neúspěšné.
4. Inkrementuj hloubku a vrať se na bod 2.

Pseudokód:

```
bool IDS (first_nod) {
1. depth = 1;

2. while (!(DLS(first_nod,depth))){
3.   if (depth == MAX_DEPTH) return false;
4.   depth++;
   }
   return true;
}
```

`DLS` – funkce implementující metodu Depth Limited Search

`MAX_DEPTH` – konstanta/proměnná udávající maximální hloubku prohledávání

Popis:

Metoda IDS se používá v případech, kdy bychom chtěli použít metodu DLS, ale nedokážeme předem odhadnout hloubku, ve které se nachází řešení.

2.1.6 Metoda zpětného navracení (*Backtracking*)

Úplnost – NE

Optimálnost – NE

Časová náročnost: O(b^m)

Paměťová náročnost: m

Implementované úlohy: Dva džbány

Algotritmus:

1. Inicializuj zásobník Open a vlož do něj počáteční uzel.
2. Pokud je zásobník Open prázdný, ukonči prohledávání jako neúspěšné.
3. Pokud lze na uzel na vršku zásobníku aplikovat první/další operátor, aplikuj ho. Jinak vyjmi uzel ze zásobníku a vrať se na bod 2.
4. Pokud je nově vzniklý uzel uzlem cílovým, vypiš cestu k řešení a ukonči prohledávání jako úspěšné. Jinak ulož uzel do zásobníku Open a vrať se na bod 2.

Pseudokód:

```
bool Backtracking (first_nod) {
1. InitS(Open);
   Push(Open,first_nod)

2. while (NotEmpty(Open))){

3.    Top(Open,exp_nod);
      new_nod = ApplyNextOp(exp_nod);
      if (new_nod == NULL) {
        Pop(Open);
      }
      else {
4.      if(Goal(new_nod)) {
          PrintPath(new_nod);
          return true;
        }
        Push(Open,new_nod);
      }
   } //while

   return false;
}
```

`InitS` – inicializuje zásobník

`Push/Pop/Top` – vloží/vyjme/ukáže uzel do/ze zásobníku

`NotEmtpy` – vrací true, pokud není fronta prázdná

`ApplyNextOp` – aplikuje první/další operátor, vrací ukazatel na vygenerovaný stav, pokud nelze žádný operátor aplikovat, vrací NULL

`Goal` – vrací true, pokud je uzel cílový

`PrintPath` – vytiskne cestu od cílového k počátečnímu uzlu (resp. naopak)

`Expand` – expanduje uzel, a jeho bezprostřední následníky, kteří nejsou v Open a ani nejsou předky tohoto uzlu, vloží do zásobníku Open

Popis:

Výhoda metody Backtracking spočívá v paměťové náročnosti. Jelikož se v paměti uchovává pouze cesta k aktuálnímu uzlu, je náročnost rovna největší prohledávané hloubce.

2.1.7 Metoda obousměrného prohledávání (*Bidirectional BFS search*)

Úplnost – ANO

Optimálnost – ANO

Časová náročnost: O(b^d)

Paměťová náročnost: O(2b^(d/2))

Implementované úlohy: 8-puzzle

Algotritmus:

1. Inicializuj dvě fronty Open a dva seznamy Closed. Do jedné fronty vlož počáteční uzel a do druhé uzel cílový.
2. Pokud je alespoň jedna fronta Open prázdná, ukonči prohledávání jako neúspěšné.
3. Vyjmi z první fronty Open uzel a vlož ho do příslušného seznamu Closed.
4. Expanduj vyjmutý uzel, a všechny jeho bezprostřední následníky, kteří nejsou v příslušné frontě Open, ani v příslušném seznamu Closed, vlož do fronty Open.
5. Porovnej obě fronty Open. Pokud obsahují stejný stav, zrekonstruuj cestu, a ukonči prohledávání jako úspěšné.
6. Vyjmi z druhé fronty Open uzel a vlož ho do příslušného seznamu Closed.
7. Expanduj vyjmutý uzel pomocí inverzních operátorů, a všechny jeho bezprostřední následníky, kteří nejsou v příslušné frontě Open, ani v příslušném seznamu Closed, vlož do fronty Open.
8. Porovnej obě fronty Open. Pokud obsahují stejný stav, zrekonstruuj cestu, a ukonči prohledávání jako úspěšné. Pokud neobsahují shodný stav, pokračuj na bod 2.

Pseudokód:

```
bool BS (first_nod,goal_nod) {

1. InitQ(OpenF);
   InitQ(OpenG);
   InitL(ClosedF);
   InitL(ClosedG);
   Insert(OpenF,first_nod)
   Insert(OpenG,goal_nod)
```

```
2. while (NotEmpty(OpenF) && NotEmpty(OpenG)){
3.   Remove(OpenF,exp_nod);
     Add(ClosedF,exp_nod);
4.   Expand(exp_nod,OpenF,ClosedF);
5.   if(SameNods(OpenF,OpenG)) {
       PrintPath(OpenF,OpenG);
       return true;
     }
6.   Remove(OpenG,exp_nod);
     Add(ClosedG,exp_nod);
7.   ExpandInv(exp_nod,OpenG,ClosedG);
8.   if(SameNods(OpenF,OpenG)) {
       PrintPath(OpenF,OpenG);
       return true;
     }
   }
   return false;
}
```

`InitQ/InitL` – inicializuje frontu/seznam

`Insert/Remove` – vloží/vyjme uzel do/z fronty

`Add` – přidá do seznamu uzel

`NotEmtpy` – vrací true, pokud není fronta prázdná

`SameNods` – prohledá fronty, a pokud v nich najde stejné uzly, vrátí true

`PrintPath` – vyhledá ve frontách stejné stavy a zrekonstruuje cestu od počátečního k cílovému uzlu, kterou následně vytiskne.

`Expand` – expanduje uzel, a jeho bezprostřední následníky, kteří nejsou ani v Open, ani v Closed vloží do fronty Open

`ExpandInv` – expanduje uzel inverzními operátory, a jeho bezprostřední následníky, kteří nejsou ani v Open, ani v Closed vloží do fronty Open

Popis:

Metoda BS je použitelná pro úlohy, jejichž operátory lze invertovat. U takových úloh pak jde prohledávat stavový prostor metodou BFS zároveň od počátečního a cílového uzlu. Časová náročnost zůstává stejná jako u metody BFS, jelikož je nutno v každém kroku porovnávat koncové stavy. Paměťová náročnost se však sníží, jelikož se z každé strany prohledá stavový prostor o poloviční hloubce.

2.2 Informované metody

Informované metody používají na rozdíl od metod neinformovaných funkci, díky níž mohou omezit velikost prohledávaného stavového prostoru. Ta je většinou založena na odhadu ceny cesty ze současného stavu do stavu cílového. Tato funkce se nazývá heuristická (zkr. heuristika). Od přesnosti odhadu se odvíjí i časová a paměťová náročnost celé metody. Všechny níže zmíněné informované metody používají stejný algoritmus jako metoda UCS , rozdíl je pouze v heuristické funkci

2.2.1 Metoda založená na výběru nejlépe ohodnoceného stavu (*BestFS - Best First Search*)

Úplnost – ANO

Optimálnost – ANO

Implementované úlohy: cesta z A do B

Popis ohodnocovací funkce:

U metody BestFS spočívá ohodnocovací funkce v kombinaci ohodnocení ceny dosavadní cesty a odhadu ceny cesty do cílového uzlu.

2.2.2 Metoda lačného prohledávání (*Greedy Search*)

Úplnost – NE

Optimálnost – NE

Implementované úlohy: cesta z A do B

Popis ohodnocovací funkce:

Metoda Greedy Search bere při výběru nejvýhodnějšího uzlu v potaz pouze heuristickou funkci, která určuje odhad ceny cesty do cílového stavu.

2.2.3 Metoda A* (*A**)

Úplnost – ANO

Optimálnost – ANO

Implementované úlohy: 8 puzzle

Popis ohodnocovací funkce:

Používá v podstatě stejný princip jako metoda BestFS, s tím rozdílem, že heuristická funkce musí být spodním odhadem skutečné ceny cesty.

2.3 Metody lokálního prohledávání

Pokud máme úlohu, jejíž řešení není nalezení optimální cesty, ale optimálního stavu, můžeme použít metody lokálního prohledávání. Tyto metody neprohledávají celý stavový prostor, nicméně mají ve spoustě případu uspokojivé výsledky, naprosto zanedbatelné paměťové nároky a dají se pomocí nich vyřešit úlohy, které jsou jinými metodami neřešitelné. Níže zmíněné metody lokálního prohledávání nejsou ani optimální, ani úplné.

Implementovaná úloha pro tyto metody je úloha cesta z A do B.

2.3.1 Metoda stoupání do kopce (Hill Climbing)

Algoritmus:

1. Ulož počáteční uzel jako uzel Current.
2. Expanduj uzel Current a z jeho následníků vyber toho, který má nejlepší ohodnocení. Tento uzel ulož jako uzel Next.
3. Pokud má uzel Next horší ohodnocení než uzel Current, ukonči prohledávání a vrať uzel Current.
4. Ulož uzel Next jako uzel Current.

Pseudokód:

```
nod * HillClimb(first_nod) {
1. current = first_nod;
   while () {
2.   next = ExpandBest(current);
3.   if (h(current) > h(next)) return current;
4.   current = next;
   }
}
```

`ExpandBest` – funkce expanduje uzel a vrací ukazatel na následníka s nejlepším ohodnocením.
`h` – návratovou hodnotou funkce je ohodnocení uzlu.

Popis:

Metoda Hill Climbing je vysoce účinná pro úlohy, jejichž cesta stavovým prostorem od počátečního uzlu k uzlu optimálnímu neobsahuje lokální extrémy. V případě, že se takový lokální extrém vyskytne, funkce v tomto bodě uvízne a skončí.

Lokální extrém – je to uzel, respektive uzly, které mají v určitém úseku optimální cesty horší ohodnocení než uzly předchozí.

2.3.2 Metoda simulovaného žíhání (*Simulated annealing*)

Algoritmus:

1. Vytvoř tabulku obsahující popis klesání teploty v závislosti na kroku metody a ulož ji jako T.
2. Ulož počáteční uzel jako Current. Krok metody k nastav na 0.
3. Načti aktuální teplotu z tabulky T. Pokud se teplota rovná nule, ukonči prohledávání a vrať uzel Current.
4. Expanduj uzel Current, náhodně vyber jednoho z následníků a ulož ho jako uzel Next.
5. Vypočítej rozdíl ohodnocení uzlů Next a Current ΔE.
6. Pokud je ohodnocení uzlu Next lepší, tj. rozdíl je větší než nula, ulož uzel Next jako Current jinak ulož uzel Next jako Current s pravděpodobností $e^{(\Delta E/T)}$.
7. Zvyš hodnotu kroku o jedna a vrať se na bod 3.

Pseudokód:

```
nod * SimAneal (first_nod) {

1. T = LoadTable();

2. current = first_nod;
     k = 0;

3. while (T[k] != 0) {
4.   next = ExpandRand(current);
5.   dE = h(nextn) - h(current);
6.   if (Rand() < exp(dE/T[k]) current = next;
7.   k++;
   }
   return current;
}
```

`LoadTable` – funkce vytvoří tabulku klesání teploty

`ExpandRand` – funkce expanduje uzel a náhodně vybere ještě neprozkoumaného následníka

`h` – návratovou hodnotou funkce je ohodnocení uzlu.

`Rand` – generuje náhodné číslo v intervalu <0-1).

Popis:

Metoda simulovaného žíhání se snaží překonat selhávání metody Hill Climbing, které je způsobeno lokálními extrémy. Využívá k tomu modelu ochlazování kovů. Od velikosti teploty se odvíjí pravděpodobnost s jakou bude přijat hůře ohodnocený stav. S přibývajícím krokem výpočtu se teplota snižuje, a když se blíží 0, jedná se v podstatě o algoritmus Hill Climbing, neboť jsou akceptovány pouze uzly které mají lepší ohodnocení než uzel Current. Hodnotu teploty je třeba nastavit experimentálně.

2.4 Metody řešení úloh s omezujícími podmínkami (CSP)

Při řešení úlohy s omezujícími podmínkami je při každém kroku metody nutné kontrolovat, zda-li se nedostala úloha do rozporu s předem stanovenou podmínkou, respektive podmínkami.

V podstatě se v těchto úlohách jedná o to, že skupině proměnných jsou postupně přiřazovány hodnoty z neprázdné množiny možností pro danou proměnnou. Systematicky se zkoušejí různé kombinace přiřazených hodnot tak, aby nebyly porušeny žádné ze stanovených podmínek. Řešením úlohy je stav ve kterém mají všechny proměnné přiřazenou hodnotu při současném splnění všech podmínek.

Všechny metody řešení úloh s omezujícími podmínkami jsou v aplikaci implementovány pro úlohu osmi dam a její modifikace.

2.4.1 Metoda zpětného navracení pro CSP (*Backtracking for CSP*)

Algotritmus:

Init:

1. Vytvoř pro všechny proměnné množiny hodnot, kterých mohou nabývat
2. Zavolej funkci *BacktrackingCSP Main* a vrať její návratovou hodnotu.

Main

1. Přiřaď aktuální proměnné první/další hodnotu z množiny jejích hodnot, která splňuje podmínky. Pokud to není možné, ukonči prohledávání jako neúspěšné a vrať false.
2. Zavolej funkci *BacktrackingCSP Main* pro následující volnou proměnnou. Pokud funkce vrátí true, ukonči prohledávání jako úspěšné, jinak se vrať na bod 1.

Pseudokód:

```
bool BacktrackingCSPInit(VarList) {
1,Init(VarList);
  for (i = 0;i <= VAR_SUM; i++) {
    CreateValList(VarList[i]);
  }
2,return BacktrackingCSPMain(VarList,0);
}
bool BacktrackingCSPMain(VarList,i) {
1. while (NextValue(VarList[i])) {
2.   if (BacktrackingCSPMain(VarList,i+1)) return true;
   }
}
```

Popis:

Metoda Backtracking pro CSP je v podstatě postavena na kombinaci neinformované metody backtracking a upřesněném principu obecného popisu řešení CSP úloh. Z této kombinace vzniká relativně jednoduchá a paměťově nenáročná metoda.

2.4.2 Metoda dopředné kontroly (*Forward checking*)

Algoritmus:

Init:

3. Vytvoř pro všechny proměnné množiny hodnot, kterých mohou nabývat
4. Zavolej funkci Forward Checking Main a vrať její návratovou hodnotu.

Main:

1. Vyber z množiny hodnot pro aktivní proměnnou první hodnotu a této proměnné ji přiřaď.
2. Zazálohuj seznam proměnných a jejich množin.
3. Pokud je stav stavem cílovým, ukonči prohledávání a vrať true.
4. Odstraň z množin hodnot všech volných proměnných hodnoty, které porušují podmínky.
5. Pokud je nějaká z množin hodnot prázdná, obnov ze zálohy seznam proměnných a jejich množin a vrať se na bod 1.
6. Zavolej funkci Main pro následující proměnnou. Pokud je návratová hodnota funkce true, ukonči prohledávání a vrať tuto hodnotu.
7. Obnov ze zálohy seznam proměnných a jejich množin.
8. Pokud je množina hodnot aktuální proměnné prázdná, ukonči prohledávání jako neúspěšné. Jinak se vrať na bod 1.

Pseudokód:

```
bool ForwardCheckInit(VarList) {

1,Init(VarList);
  for (i = 0;i <= VAR_SUM; i++) {
    CreateValList(VarList[i]);
  }
2,return ForwardCheckMain(VarList,0);
}

bool ForwardChceck(VarList,i) {
  do {
1.  VarList[i]->val = Remove(VarList[i]->ValList);
2.    BackupVarList = CopyList(VarList);
3.    if (Goal(VarList)) {
        return true;
      }
4.    RemoveConflict(VarList,i);
5.    if(SomeEmpty(VarList)) {
        VarList = CopyList(BackupVarList);
      }
6.    else if(ForwardCheck(VarList,i+1)) return true;
7.    VarList = CopyList(BackupVarList);

8.  } while(NotEmpty(VarList[i]->ValList))
  return false;
}
```

Popis:

Metoda Forward Checking, jejíž algoritmus je na první pohled relativně složitý, využívá v podstatě jednoduchého principu. Systematicky přiřazuje proměnným hodnoty, a když následující proměnné nemůže hodnotu přiřadit, vrací se o krok zpět.

2.4.3 Metoda minimálního konfliktu (Min-conflict)

Algoritmus:

1. Označ první proměnnou stavu, jako proměnnou aktuální.
2. Ohodnoť množství konfliktů s podmínkami, pro každou hodnotu z definičního oboru aktuální proměnné.
3. Pokud existuje hodnota, která má stejné, nebo lepší ohodnocení než aktuální hodnota proměnné, resp. než ostatní hodnoty v definičním oboru hodnot, nechť je její novou hodnotou. Pokud jich je více se stejným nejnižším počtem konfliktů, přiřaď jí tu z nich, která je první v seznamu.
4. Pokud je nový stav stavem cílovým, ukonči prohledávání jako úspěšné, a vrať cílový stav.
5. Posuň se na další proměnnou v seznamu. Pokud jsi na konci seznamu, přesuň se opět na proměnnou první.
6. Vrať se na bod 2.

Pseudokód:

```
bool MinConflict(VarList) {
1. i = 0;
   while() {
2.
3.   VarList->val = FindBest(VarList[i]);
4.   if (Goal(VarList)) return true;
5.   i++;
     if (i >= VAR_SUM) i = 0;
6. }
}
```

Popis:

Metoda Min-conflict je metoda lokálního prohledávání. Jako výchozí stav akceptuje jakýkoli stav, který je úplně definovaný (všechny proměnné mají přiřazenou hodnotu), ale nesplňuje podmínky. Metoda postupně přiřazuje hodnoty proměnným tak, aby porušování podmínek eliminovala.

2.5 Jednoduché hry

2.5.1 Prohledávání AND/OR grafu

Implementované hry: Zápalky

Algoritmus:

1. Vytvoř frontu Open a ulož do ní počáteční (kořenový) uzel.
2. Pokud není fronta prázdná, vyjmi z ní uzel určený k expanzi.
3. Pokud má právě vybraný uzel jako bezprostředního následníka jeden či více listů, prozkoumej je a ohodnoť je.
4. Pokud mají všichni přímí následníci aktuálního uzlu ohodnocení, ohodnoť aktuální uzel a přenes jeho ohodnocení jeho předchůdcům.
5. Pokud je kořenový uzel ohodnocen a jeho ohodnocení je přípustné, ukonči prohledávání jako úspěšné.
6. Expanduj aktuální uzel – vygenerování nelistových přímých následníků.

Pseudokód:

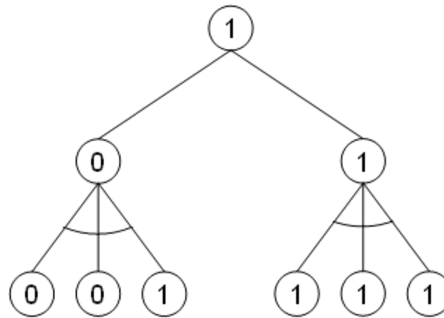
```
bool AndOrBFS (first_nod) {
   1. InitQ(Open);
      Insert(Open,first_nod);

   2. while (NotEmpty(Open))){
         Remove(Open,exp_nod);
   3.    if(LeafParent(exp_nod)) {
            ValLeafs(exp_nod);
   4.       ReValuate(exp_nod);
         }
   5.    if (first_nod->val >= MIN_VAL) return true;
   6.    Expand(exp_nod,Open);
      }
      return false;
}
```

`ReValuate(exp_nod)` – Rekurzivní funkce která postupuje směrem ke kořenovému uzlu a dle pravidel AND/OR grafu přenáší ohodnocení uzlu jeho předchůdcům.

Popis

Již bylo zmíněno, že jednoduchá hra je taková hra, u které lze v reálném čase prohledat celý její AND/OR graf. Prohledávání tohoto grafu může probíhat buď do šířky, nebo do hloubky. Cílem tohoto prohledávání je nalezení koncových stavů a postupné přenesení jejich ohodnocení AND/OR grafem až k počátečnímu uzlu.



Obrázek 2.1 - část AND/OR grafu s přeneseným ohodnocením

2.6 Složité hry

Jelikož u složitých her nelze v reálném čase prohledat celý jejich stavový prostor, řeší se to prohledáváním do určité hloubky.

Algoritmus je v podstatě stejný jako u prohledávání AND/OR grafu u jednoduchých her. Rozdíl je v tom, že všechny uzly v maximální hloubce jsou označeny jako listové a je pro ně zavolána heuristická funkce, která určí jejich hodnotu a podle pravidel AND/OR grafu je tato hodnota přenesena na jejich předchůdce. Pokud je uzel uzlem AND, získá od svých bezprostředních následníků hodnotu nejnižší, pokud je uzel uzlem OR, získá hodnotu nejvyšší. Tento algoritmus se nazývá MiniMax. Má však nevýhodu v tom, že zbytečně prohledává a ohodnocuje uzly, na kterých nezáleží. Vylepšenou verzí tohoto algoritmu, který tento problém odstraňuje, je algoritmus AlfaBeta řezů.

2.6.1 Alfa-Beta řezy (*Alpha-Beta Cutoff*)

Algoritmus:

1. Hodnoty proměnné Alfa a proměnné beta jsou předány jako parametr funkce.
2. Pokud se aktuální uzel nachází v maximální povolené hloubce nebo je uzlem listovým, zavolej pro něj heuristickou funkci, ukonči prohledávání a vrať jeho hodnotu.
3. Pokud já aktuální uzel typu AND, jdi na bod 4 Pokud je typu OR tak pokračuj.
   a. Pokud má aktuální uzel jednoho, nebo více neohodnocených následníků, zavolej pro prvního z nich metodu Alfa-Beta se současnými hodnotami proměnných Alfa, Beta a ulož její návratovou hodnotu. Jinak jdi na bod 3d.

b. Pokud je návratová hodnota větší nebo rovna hodnotě Beta, ukonči prohledávání a vrať hodnotu proměnné Beta, jinak pokračuj.
c. Pokud je návratová hodnota větší než hodnota proměnné Alfa, nechť je její novou hodnotou. Pro kořenový uzel ulož tento uzel s nejlepším ohodnocením. Vrať se na bod 3a.
d. Ukonči prohledávání a vrať hodnotu proměnné Alfa.

4. Aktuální proměnná je typu AND.
   a. Pokud má aktuální uzel jednoho, nebo více neohodnocených následníků, zavolej pro prvního z nich metodu Alfa-Beta se současnými hodnotami proměnných Alfa, Beta a ulož její návratovou hodnotu. Jinak jdi na bod 3c.
   b. Pokud je návratová hodnota menší než hodnota proměnné Beta, nechť je její novou hodnotou. Vrať se na bod 3a.
   c. Ukonči prohledávání a vrať hodnotu proměnné Beta.

Pseudokód:

```
1. int AlphaBeta (Alpha,Beta,nod,max_depth,&best) {
2.   if (IsLeaf(nod) || nod->depth == max_depth) {
       return heur(nod);
     }
3.   if (isOR) {
   a.  while (SomeUnknwChild(nod)) {
        val = AlphaBeta(Alpha,Beta,FstUnknwChld(nod),depth,best);
   b.    if (val > Beta) return Beta;
   c.    if (val > Alpha) {
           Alpha = val;
           if(isRoot(nod)) best = LstKnwnChld(nod);
         }
       }
   d.  return Alpha;
     }
4.   else {
   a.  while (SomeUnknwChild(nod)) {
         val = AlphaBeta(Alpha,Beta,FstUnknwChld(nod),max_depth);
   b.    if (val < Beta) Beta = val;
       }
   c.  return Beta;
   }
```

2.7 Shrnutí

V této kapitole byly představeny všechny vybrané metody řešení úloh, které jsou implementovány v demonstrační aplikaci grafických animací.

3 Grafické animace metod řešení úloh

Cílovou skupinou lidí, pro které je aplikace určena, jsou především studenti předmětu Základy umělé inteligence, který se vyučuje ve druhém ročníku bakalářského programu na Fakultě informačních technologií VUT v Brně. Měla by jim usnadnit pochopení těchto metod.

3.1 Implementační prostředí

Aplikace je napsána v jazyce C s využitím některých funkcí jazyka C++. Dále byla použita volně dostupná grafická knihovna glut, která zajišťuje vykreslení okna a předává vstupy z klávesnice a myši programu. Tato knihovna sice neobsahuje téměř žádné uživatelské rozhraní, ale zato poskytuje velikou volnost, při jeho tvorbě.

Pro psaní, kompilaci a ladění programu, bylo použito vývojové prostředí Dev-C++ v. 4.9.9.2.

3.2 Teoretický návrh aplikace

Základní otázkou je logické rozvržení metod. Buď se dají setřídit podle jejich vlastností (informované, neinformované, lokální prohledávání atd.) nebo podle typu úloh které řeší. Jako finální přístup byl zvolen způsob řazení dle typů úloh, poskytuje totiž menší zanořování v hlavním menu a hlavně umožňuje porovnávání efektivity různých metod pro stejnou úlohu. Jako výsledek návrhu vznikl seznam stavů programu jenž naleznete v [příloha 1].

3.3 Struktura zdrojových kódů

Struktura souborů zdrojových kódů obsahuje jeden soubor s příponou .cpp a zbytek souborů s příponou .h. Jelikož je program napsán v podstatě v jazyku C a obsahuje velký počet globálních proměnných, je pro něj tento model velice praktický.

3.3.1 main.cpp

Tento soubor obsahuje především funkce pro řízení vstupů a výstupů programu. Jedná se zde hlavně o správu událostí. V závislosti na tom co uživatel zadá pomocí klávesnice či myši se volají další funkce programu.

Další důležitou část souboru main.cpp, je vkládání knihoven, které obsahují další funkce programu.

3.3.2 main.h

V knihovně main.h jsou definovány globální konstanty používané pro správu uživatelského rozhraní. Dále jsou tu definovány důležité struktury, například pro načítání souborů bmp z disku, ukládání pixelů, či načtení fontů.

Knihovna dále obsahuje definice externích proměnných, což je důležité pro správné slinkování programu.

Poslední věcí, která se zde nachází je výčet všech hlaviček funkcí, které se nachází v ostatních knihovnách, spolu s jejich hrubým popisem.

3.3.3 functions.h

Knihovna obsahující funkce pro vykreslování geometrických objektů jako je kružnice, přímka, obdélník či bod.

3.3.4 manage.h

Nejobsáhlejší knihovna programu. Stará se o správu uživatelského rozhraní. Obsahuje funkce pro vyhodnocování vstupů z myši a klávesnice a v závislosti na nich upravuje stav konečného automatu programu.

3.3.5 xprint.h, xmetods.h

Pro každou úlohu, byly vytvořeny tyto dva soubory. Název je pro všechny v podstatě stejný, liší se pouze počátečním písmenem. Obecně knihovna xmetods.h obsahuje funkce implementující metody řešení a knihovna xprint jejich zobrazení.

Seznam jednotlivých knihoven a jejich příslušnost k úlohám:

ametods.h, aprint.h	cesta z A do B
cmetods.h, cprint.h	Alfa Beta řezy (c - cutoff)
jmetods.h, jprint.h	Úloha dvou džbánů (j - jugs)
mmetods.h, mprint.h	Zápalky (m - matches)
pmetods.h, pprint.h	Hlavolam „8“ (p - puzzle)
qmetods.h, sprint.h	Úloha osmi dam (q - queens)

3.4 Systém názvů proměnných a funkcí

Celý zdrojový text je psán v anglickém jazyce, stejně tak názvy proměnných, struktur a typů. Tyto názvy jsou většinou výmluvné, takže z nich jde vyčíst k čemu proměnná, funkce či struktura slouží. Navíc každá inicializace obsahuje komentář v programátorské češtině (bez hacku a carek).

Ve zdrojovém kódu se vyskytuje mnoho funkcí, které implementují podobné věci, ale pro různé úlohy či metody. Jejich název je proto odlišen podobnou metodou jako v případě knihoven – liší se v počátečním písmeně, popřípadě obsahují slovo vázající se k příslušné úloze. Například struktura pro uchovávání informací o stavu dané úlohy má tyto modifikace S_Jug, S_Puzz. První je pro úlohu džbánů, druhá pro úlohu hlavolam „8“.

3.5 Implementace uživatelského rozhraní

3.5.1 Inicializace programu

Inicializace programu sestává prakticky ze dvou částí. První z nich je registrace funkcí, které jsou volány v případě vzniku událostí (změna velikosti okna, kliknutí myši, stisknutí klávesy). Tato registrace probíhá v hlavní funkci programu `main(...)`. O druhou část inicializace se stará funkce `StartProgram()`, která je volána při první změně velikosti okna, která proběhne při jeho vytvoření. Funkce `StartProgram()` obsahuje nahrání fontů do paměti, zobrazení hlavního menu, inicializaci funkce random a hlavně nastavení stavové proměnné programu `state`.

3.5.2 Uživatelský vstup

Nejčastěji využívaným vstupem je vstup z myši. Funkce grafické knihovny glut sledují události které myš generuje a zpracovává je. Program je nastaven tak, že reaguje pouze na kliknutí levým nebo pravým tlačítkem. Pokud nastane tato událost, glut volá funkci `onMouseClick(...)`, která zpracuje souřadnice myši a které tlačítko bylo stisknuto. Z této funkce se zavolá s parametry souřadnic a stisknutého tlačítka funkce `ManagerMenu(...)`.

Dalším možným vstupem je vstup z klávesnice. Ten je aktivován pouze v některých stavech programu (většinou při nastavování počátečních stavů úloh). V tomto případě glut volá buď funkci `onKeyboard(...)` v případě stisknutí kláves, které jdou definovat ASCII hodnotou, nebo funkci `onKeyboardSpecial(...)` v případě speciálních kláves (kurzorové šipky). Z těchto je pak volána již dříve zmíněná funkce `ManagerMenu(...)` s parametry, které simulují kliknutí myši na tlačítko, které klávesa na klávesnici počítače nahrazuje. Typicky se jedná o kurzorové klávesy šipek, které nahrazují kliknutí na ikonku šipky, jenž je zobrazena v okně programu.

3.5.3 Zpracování vstupu

Údaje ze vstupů následně zpracovává funkce `ManagerMenu(...)`. Napřed dle globální proměnné `state` určí stav v jakém se program nachází a následně dle souřadnic určí, které tlačítko bylo stisknuto a na základě toho přiřazuje hodnoty řídícím proměnným, inicializuje struktury, volá příslušné funkce. V případě, že se kliknutím na dané tlačítko změní stav konečného automatu, změní i hodnotu proměnné `state`.

Proměnná `state` je typu integer. Pro přehlednost jsou ji však přiřazovány předem definované konstanty, jejichž názvy jsou voleny tak, aby bylo jasné, v jakém stavu chceme, aby se program nacházel. Jako příklad je možné uvést stavy, kterými program prochází, pokud chce uživatel zobrazit úlohu dvou džbánů řešenou metodou DLS.: `MAIN -> JUGS -> JDLS_INIT -> JDLS` popřípadě posléze `PLAY_JDLS`.

3.5.4 Zobrazení menu

Zobrazení menu je zajišťováno funkcí `LoadMenu(...)`, která má jediný parametr a tím je proměnná `MenuNo`. V závislosti na ní pak zobrazí dané nabídky. Proměnná `MenuNo` je stejně jako proměnná `state` typu integer a jou jí také přiřazovány předem definované konstanty. Pro zobrazení menu pro výběr metody řešení úlohy dvou džbánů je to například konstanta `M_JUGS`.

Odtud je také volána funkce `LoadButton(...)`, která dle předaných parametrů vykreslí tlačítko. Parametry jsou souřadnice x, y a textový řetězec, který určuje popisek tlačítka.

3.5.5 Fonty

Fonty jsou řešeny rastrově. Při inicializaci programu se volá funkce LoadFonts, která načte soubory fontů do paměti. Jména souborů jsou následující:

123_8.bmp	číslice 0-9
ABC_8.bmp	velká písmena české abecedy
abcd_8.bmp	malá písmena české abecedy
zn_8.bmp	znaky (:.!?,-/<>|*)

Soubory jsou v paměti načteny jako pole struktur `S_RGBA`. O vypisování textových řetězců se stará funkce `WriteText(...)`. Jako parametry má souřadnice počátku vypisovaného textu a konstantní textový řetězec. Ten postupně prochází a převádí ASCII hodnoty znaků na hodnoty interní, podle převodní tabulky. Ta třídí vstupní znaky do tří kategorií: malé písmeno, velké písmeno, znak. U čísel se zachovává jejich původní ASCII hodnota.

Pro vysvětlení principu převodní tabulky si vezměme jako příklad řetězec Ac. První znak má ASCII hodnotu 65. Na 65. pozici se v tabulce nachází číslo 1, pro znak c je to pak ASCII hodnota 99

a příslušné číslo v převodní tabulce 104. První znak má tedy interní hodnotu 1. Jelikož je menší než 100, je znak identifikován jako velké písmeno a volá se funkce `LoadChar(...)`, která načte z příslušné struktury obsahující velké znaky abecedy první znak. Interní hodnota druhého znaku je větší než sto, ale menší než 150, proto je identifikován jako malé písmeno a opět se volá funkce `LoadChar(...)`, která načte z příslušné struktury obsahující malé znaky abecedy čtvrtý znak (104-100 = 4). Speciální znaky mají interní hodnotu mezi 150 a 200. Jak již bylo zmíněno, číslům se žádná interní hodnota nepřiřazuje.

Tento systém byl zvolený kvůli zobrazování českých znaků, protože ty na rozdíl od anglických nejdou za sebou a jsou roztroušeny napříč ASCII tabulkou. Stejně tak i speciální znaky.

Funkce `LoadChar(...)` vyhledává znaky podle úplných mezer v ose y mezi nimi. Proto je nutné při vytváření bmp souboru, aby tam tyto mezery opravdu byly. V klasickém proporcionálním textu dochází totiž k vytváření záporných mezer, tedy zasouvání jednotlivých písmen pod sebe. Taková dvojice písmen by pak byla identifikována jako písmeno jedno.

Změna fontu je jednoduchá. Stačí v jakémkoli grafickém editoru napsat českou abecedu a uložit jako 24 bitový bmp obrázek. Nutné podmínky jsou absolutní mezery, bílé pozadí, oříznutý začátek obrázku podle prvního znaku a minimálně jeden sloupec bílých pixelů na konci.

3.6 Implementace jednotlivých úloh

Implementace každé z úloh se skládá ze dvou částí. Z implementování úlohy spolu s metodou jejího řešení a ze zobrazení postupu řešení. Jelikož má být aplikace názorná, je vyžadována možnost zobrazení postupu řešení krokovat. To v podstatě vylučuje reálné použití rekurzivních metod. Ty musely být upraveny tak, aby se chovaly jako rekurzivní, ovšem při nerekurzivní implementaci, což místy vyžadovalo docela krkolomné konstrukce.

Dalším problémem způsobeným nutností krokování je uchovávání stavu úlohy, metody a jejich struktur. Pokud by tyto byly lokální, při předání řízení programu správci událostí, by se všechny ztratily. Proto je pro každou úlohu vytvořen globální ukazatel na hlavní kontrolní strukturu, která obsahuje stěžejní informace o dané úloze. Jako příklad je níže uvedena řídící struktura pro úlohu 8 puzzle.

```
typedef struct PuzzMain {
  S_Puzz * topPuzz;        ukazatel na kořenový uzel
  PQueue * Qopen;          ukazatel na frontu Open
  PQueue * Qclose;         ukazatel na frontu Close
  S_Puzz * goal;           ukazatel na cílový stav
  int states;              počet stavů generovaného stromu
  int depth;               hloubka stromu
} PuzzMain;
```

U metod kde je nutné vracet cestu k řešení, byl tento problém vyřešen tak, že každý uzel stromu obsahuje ukazatel na svého přímého předchůdce. Pokud je nalezen cílový stav, stačí pouze sledovat tyto ukazatele.

3.6.1 Úloha dvou džbánů

Jelikož cílem aplikace je především demonstrace principu metod řešení úloh, obsahuje pouze základní verzi této úlohy. Tedy dva džbány, jeden o obsahu dva litry a druhý o obsahu tři litry. Pokud by byla velikost zadávána uživatelem, stavový prostor k prozkoumávání by se mohl neúměrně zvětšit a nastal by problém se zobrazením stromu, který by se na obrazovku nevlezl.

Stav úlohy je reprezentován dvěma způsoby. První je určen pro metody BFS, DFS a Backtracking. Druhý pak pro metody DLS a IDS, které generují více rozsáhlejší stromy. Obrázek uvádí příklad reprezentace stavu, kdy 1. džbán obsahuje 4 litry vody a 2. džbán 3 litry.

Obrázek 3.1 - Ukázka reprezentace stavu úlohy dvou džbánů

Praktický průchod programem je následující:

1. Uživatel si vybere v menu úlohu dvou džbánů. Zobrazí se mu nabídka pro výběr metody řešení
2. Uživatel vybere metodu řešení. Inicializuje se řídící struktura pro tuto metodu a zobrazí se navigační menu pro danou metodu.
3. Uživatel vybere spuštění metody
    a. krokově manuálně – opětovným kliknutím na tlačítko další, popřípadě stisknutím mezerníku.
    b. krokově automaticky – klikne na tlačítko spustit. Metoda je v tomto případě opakovaně volána za daný časový interval dokud nedosáhne cílového stavu, nebo uživatel nestiskne tlačítko stop.
4. Při zavolání metody, respektive funkce `DrawJugsTree(...)` se provádějí následující kroky:
    a. Rozpoznání metody podle předaného parametru.
    b. Zavolání funkce, která generuje jeden krok dané metody
    c. Vykreslení stromu zobrazujícího současný stav
    d. Pokud metoda dosáhla cílového stavu, výpis řešení
5. Uživatel stiskne tlačítko zpět – uvolnění struktur z paměti

Stěžejní funkce pro úlohu dvou džbánů mají název `JGenerateXXX(...)` Generuje samotný průběh metody. Existuje v modifikacích pro BFS, DFS, DLS a Backtracking. Tyto funkce dále používají procedury pro operaci se zásobníkem a frontou, pro testování aplikovatelnosti operátoru, aplikování operátoru, vytváření stromu.

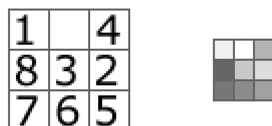
Pro vykreslování stavu slouží funkce `DrawJugState(...)`. Jako parametr je nutno zadat souřadnici pro vykreslení, které jsou posléze uloženy do struktury uchovávající informace o stavu. Vykreslování stromu provádí funkce `ThruTreeJugs()`, která rekurzivně prochází stromovou strukturou, a podle hloubky stavů vypočítává ypsilonové souřadnice pro jejich vykreslení. Souřadnice osy x se určují podle již vykreslených stavů. Celý strom je pro jednoduchost zarovnáván na levou stranu.

Další dvě obdobné funkce se jmenují `ThruTreeLines()` a `ThruTreeJugsRedraw()`, první vykresluje spojovací čáry druhá překresluje stavy, které byly spojovacími čárami přeškrtnuty. Tato situace nastává u metod DLS a IDS, jelikož kvůli velké šířce stromu, je nutno zobrazit uzly stromu o stejné hloubce do dvou řad pod sebou.

V případě nalezení cílového stavu se postup řešení vypíše na spodní straně obrazovky. Zároveň se také zelenou barvou vysvítí toto řešení ve vykresleném stromu.

3.6.2 Hlavolam „8"

Největší změnou oproti úlohám dvou džbánu je to, že si uživatel může zvolit výchozí stav úlohy. To přináší mnohé problémy. Především ten, že může být zadán počáteční stav, který k vyřešení potřebuje prohledat velký kus stavového prostoru. Zobrazování bylo tedy vyřešeno proměnnou velikostí reprezentace stavu. Existují dvě velikosti. Pokud nelze vykreslit ani strom skládající se z menších stavů, vypisují se pouze informace o hloubce stromu a počtu stavů.



Obrázek 3.2 - Reprezentace dvou totožných stavů úlohy Hlavolam "8"

Průchod programem je v podstatě stejný jako u úlohy dvou džbánů. Rozdíl spočívá pouze v inicializaci výchozího stavu.

Hlavní funkce implementující metody řešení se jmenují `PGenerateXXX(...)`. Existují v modifikacích pro BFS, BS a A*. Opět používají další funkce nutné pro generování stavů. Speciálně pro metodu A* je ve funkcích `h1` a `h2` implementováno heuristické ohodnocení stavů.

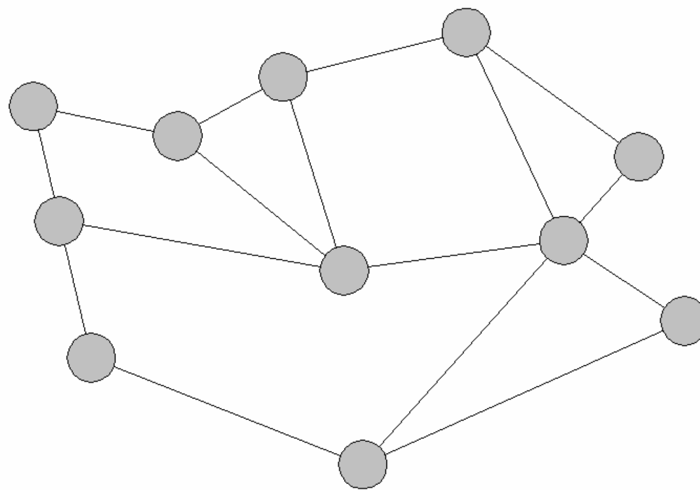
Pro vykreslování stromů a stavů platí téměř totéž jako pro úlohu dvou džbánů. Jedna z odlišností je vykreslování stromů při metodě BS. Pro znázornění fungování této metody je vytvořeno vykreslení dvou stromů nad sebou, respektive proti sobě. Jelikož se spodní strom

vykresluje odspoda nahoru, musela pro něj být vytvořena speciální vykreslovací funkce, která počítá jeho souřadnice.

Vykreslení postupu k dosažení cílového stavu opět přináší problém. Co když je počet kroků větší jak 10, což je maximální počet stavů, které lze na obrazovku vedle sebe vykreslit. Tento problém je vyřešen tak, že se v případě nutnosti vynechává každý druhý, respektive třetí stav. Z takovéhoto zápisu je postup stále patrný a je jej možné vypsat na obrazovku.

3.6.3 Cesta z A do B

Předlohous pro graf, který je použit pro úlohu nalezení cesty z místa A do místa B, je mapka ze zdroje [1]. Ta původně obsahovala i názvy míst, nicméně pro demonstraci v rámci této aplikace to není nutné, neboť průběh hledání cesty se zobrazuje přímo v grafu.



Obrázek 3.3 - graf míst a možných přechodů mezi nimi

Na této úloze jsou demonstrovány tři druhy metod. Neinformované, informované a metody lokálního prohledávání. Mají pak tyto konkrétní zástupce:

Neinformované metody: UCS

Informované metody: Best First Search, Greedy Search

Metody lokálního prohledávání: Hill-climbing, Simulated annealing

Průchod programem je následující:

1. Uživatel si vybere úlohu hledání cesty z A do B
2. Dále si vybere jakou metodou chce aby byla tato úloha řešena.
3. Na zobrazené mapce vybere myší bod A a bod B
4. Spustí metodu v režimu krokování nebo automatického krokování
5. Stiskne tlačítko zpět. Nyní si může vybrat jinou metodu, buď se stejnými uzly A a B, nebo si může vybrat jiné.

U informovaných metod a u metod lokálního prohledávání je zapotřebí použití heuristické funkce, která odhaduje cenu cesty do cíle. V této úloze ji supluje funkce, která podle souřadnic jednotlivých uzlů vypočítá jejich vzdálenost.

Každý uzel je reprezentován strukturou, ve které jsou uloženy informace o jeho poloze, zda již byl prohledán či ne a samozřejmě ukazatele na uzly do kterých z něj vede cesta.

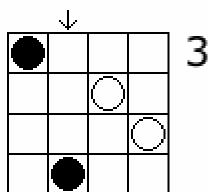
Systém prohledávání se provádí jednoduchým způsobem. Jako následníci uzlu jsou označeny všechny jeho sousední uzly. Z těch se pak vybírá ten, který splňuje nejlepší kritéria pro danou metodu.

Vybírání uzlů myší implementuje funkce `PickNod(...)`. Jejími parametry jsou souřadnice x a y na které uživatel klikl myší. Funkce posléze prohledá celý graf a najde uzel, který těmto souřadnicím odpovídá. Pokud takový uzel nalezne, uloží si na něj ukazatel do hlavní kontrolní struktury pro tuto úlohu.

3.6.4 Úloha 8 dam

Tato úloha se liší od ostatních v tom, že není třeba vykreslovat žádné stromy či seznamy. Jde v ní čistě jenom o nalezení konečného stavu, nikoliv o nalezení cesty. To přináší mnohé ulehčení při její implementaci. Na druhou stranu obtížnost spočívá v tom, že všechny metody řešení, které jsou na této úloze demonstrovány, mají rekurzivní algoritmus. Jak již bylo zmíněno dříve, nelze v demonstrační aplikaci tohoto typu rekurzivní funkci použít, neboť by uživatel viděl pouze počáteční stav a o chvíli později stav cílový. Postup řešení by mu zůstal skryt.

Zobrazení úlohy je implementováno nejednodušší možnou cestou. Zobrazením mřížky o určitém počtu řádků a sloupců, ve které jsou umístěny dámy, reprezentované plnými nebo prázdnými kolečky v závislosti na tom, jestli byla dáma vyšetřována ohledně splňování výchozích podmínek úlohy.



Obrázek 3.4 - Reprezentace stavu úlohy 8 dam řešené metodou ForwardChecking

Aktuální vyšetřovanou dámu označuje malá šipka která se posouvá po horní straně šachovnice. Napravo od prvního řádku je pak zobrazen počet dosavadních kroků metody.

Šachovnice je interně reprezentována dvojrozměrným polem pravdivostních hodnot. Přítomnost dámy na dané souřadnici značí hodnota true, nepřítomnost hodnota false.

Pro každou z metod je vytvořena funkce, která se stará o generování této metody. Jejich názvy jsou `QBacktrackingCSP()`, `QForwardCheck()` a `QMinConflict()`. Z nich se dále volají funkce pro zjišťování konfliktů `Conflicts()`, či pro odstraňování konfliktních stavů `RemConflicts()` (pro metodu ForwardChecking).

Největší problém představovala implementace metody Forwrad Checking. Základní algoritmus počítá s dopředným odstraňováním konfliktních stavů. Pokud se po odstranění těchto konfliktů vyskytne volná proměnná, které nelze přiřadit žádnou hodnotu (dámu nelze nikam umístit), obnoví se seznam proměnných ze zálohy a proměnná, která způsobila tento stav je z něj vymazána. Následně se pokračuje přiřazením proměnné další. Při rekurzivním algoritmu je programátorské řešení relativně snadné. Stačí vždy vytvořit pouze jednu zálohu. Ovšem při nerekurzivním by se muselo vytvářet až tolik záloh, kolik má šachovnice sloupců, což má své nevýhody. Druhou možností, je kompletní naplnění všech sloupců napravo od dámy včetně, která konflikt způsobila, její odstranění, a poté odstranění konfliktních stavů pro dámy nalevo. Toto řešení je programátorsky jednodušší a pro potřeby této demonstrační aplikace naprosto dostačující.

Princip odstraňování konfliktů je takový, že funkce převezme jako parametr souřadnice aktuální dámy. Poté prozkoumá všechny souřadnice, které jsou od ní dolů horizontálně, vpravo vertikálně a na částech diagonál po pravé straně. Pokud narazí na dámu (hodnota true) odstraní ji (přepíše hodnotou false).

3.6.5 Zápalky

Hlavní věc která je na této úloze demonstrována je prohledávání AND/OR grafu, za účelem vyhledávání listových uzlů, a přenosu těchto ohodnocení k počátečnímu stavu. Uživatel si může zvolit zda-li chce prohledávat graf funkcí BFS, DFS, nebo DFS s obráceným pořadím aplikování operátorů. Běžné pořadí aplikování operátorů je odebrání jedné, dvou, tří zápalek.

Také si může zvolit výchozí počet zápalek. Nutno podotknout, že po překročení určité hranice již nelze strom zobrazit a tak se vypíše pouze statistika o stromu (počet uzlů, hloubka stromu) a tah který vede k úspěšnému řešení. Pro některé počty ovšem tato úloha pro začínajícího hráče vždy končí prohrou. (4,8,12,16…)

Struktura uzlu obsahuje informace o pozici uzlu, počtu zápalek, pravdivostní hodnotu prohledání, pravdivostní hodnotu ohodnocení uzlu, ukazatele na přímého předchůdce a ukazatele na následníky. Odkaz na následníky je řešen pevným polem tří ukazatelů. Pokud má stav méně potomků, přiřadí se nevyužitým ukazatelům hodnota NULL.

Grafické znázornění uzlů je následující

Neohodnocený uzel – bílá

Ohodnocený příznivě – zelená

Ohodnocený nepříznivě - červená

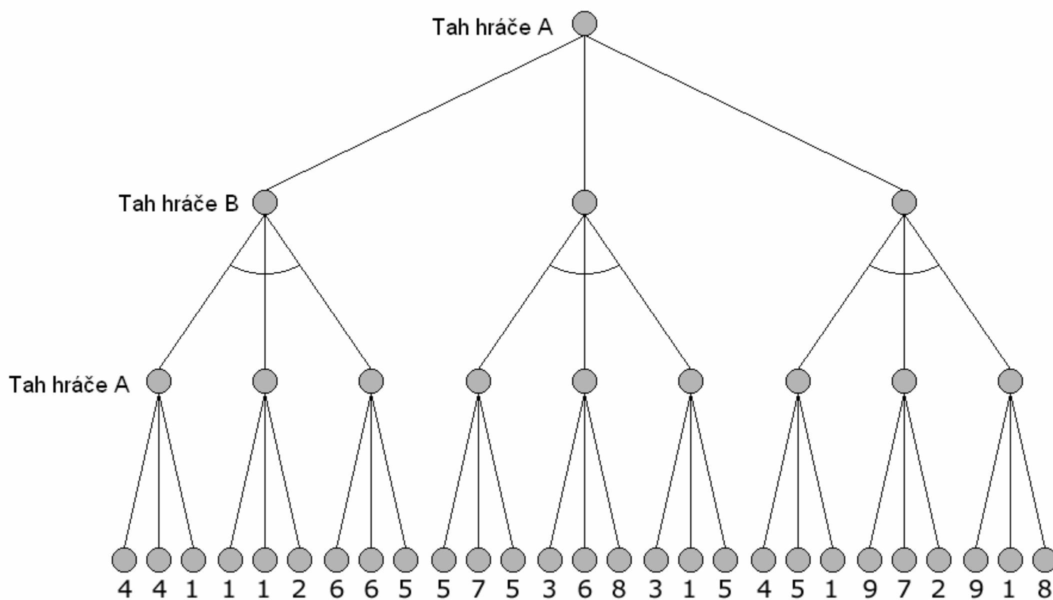
O prozkoumávání stavového prostoru se starají dvě funkce. `MGenerateBFS(...)`, `MGenerateDFS(...)`. Funkce DFS obsahuje navíc pravdivostní parametr inv. Pokud je jeho hodnota true, prohledává se stavový prostor s operátory v opačném pořadí.

Přenos ohodnocení uzlu směrem k jeho předchůdcům má na starosti funkce `Valuate(...)`. Kroková implementace této jinak rekurzivní funkce, je udělána pomocí zvláštního stavu, do kterého se program dostane po zavolání této funkce. Pokud je v dalším kroku opět zavolána například funkce MGenerateBFS(...), nespustí se generování dalších stavů, ale volá se opět funkce `Valuate(...)`. Přenášení ohodnocení se řídí následujícími pravidly:

- Pokud je přenášeno ohodnocení příznivé, přenáší se jeho hodnota tak dlouho, dokud se nenarazí na uzel AND. V tomto případě se zkontrolují jeho ostatní následníci, a pokud i oni mají ohodnocení příznivé, přenáší se dále.
- Pokud se přenáší hodnocení nepříznivé, přenáší se tak dlouho, dokud se nenarazí na uzel OR. V tomto případě se zkontrolují všichni jeho následníci, a pokud i oni mají hodnocení nepříznivé, přenáší se toto ohodnocení dále.

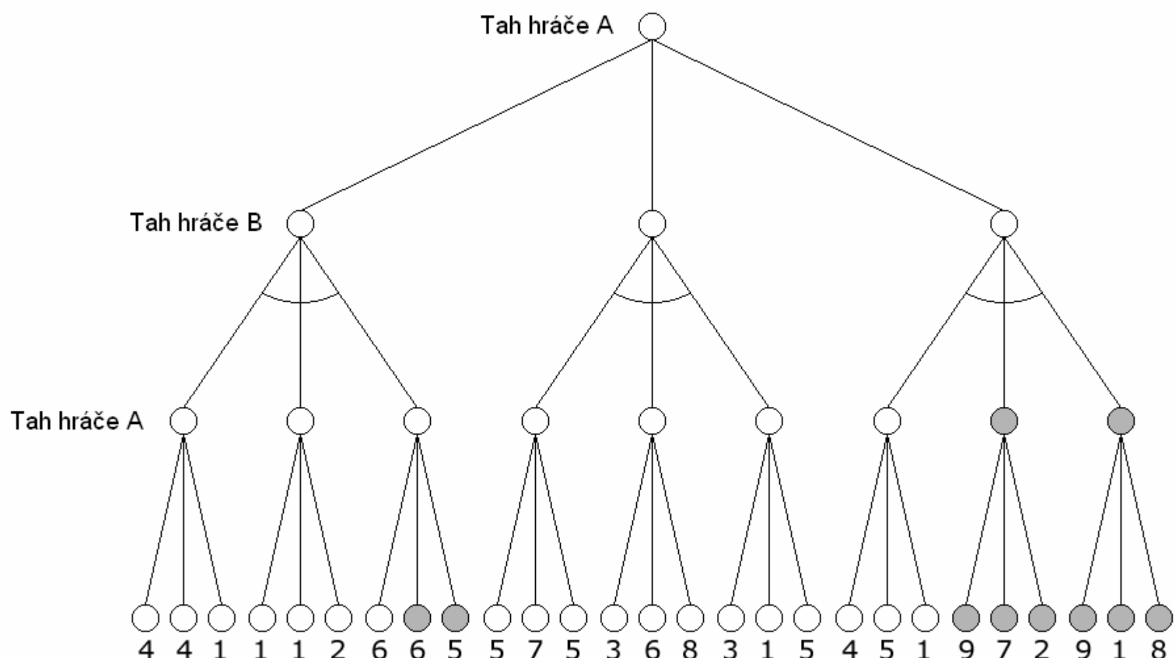
3.6.6 Alfa Beta prořezávání

Demonstrace AlfaBeta řezů není implementována pro žádnou konkrétní hru, ale pro hypotetickou část stavového prostoru. Jedná se o AND/OR graf, který reprezentuje hru dvou hráčů. Na tahu je hráč A. Cílem je najít tah, který vede k nejlepší možnosti výběru dalšího tahu po tahu hráče B. Hloubka prohledávání je tedy nastavena na 3. Vše vysvětluje následující obrázek:



Obrázek 3.5 - hypotetický stavový prostor pro AlfaBeta řezy

V hloubce 3 se nacházejí ohodnocené stavy. Úkolem Alfa Beta řezu je najít nejlépe dosažitelný stav a při tom neprohledávat uzly, na kterých nezáleží. Pokud by si například hráč A vybral první tah zleva, hráč B má na výběr ze tří možností. Prozkoumá tedy následníky těchto tří možností a zjistí jejich ohodnocení. Prohledá svůj první tah a zjistí, že ten nabízí možnosti pro hráče A 4,4,1. Prozkoumá druhý tah, zde jsou možnosti pro hráče A 1,1,2. Pokud začne zkoumat třetí, již při první možnosti hráče A, jenž je reprezentována uzlem s ohodnocením 6, je mu jasné, že třetí tah nezvolí, jelikož obsahuje lepší možnost pro hráče A, než jeho tahy 1 a 2. V tomto místě se tedy provede řez, a další dva tahy, ze kterých by si mohl hráč A vybrat, se nezkoumají. Na následujícím obrázku jsou šedou barvou znázorněny neprohledané uzly.



Obrázek 3.6 - vynechané uzly při AlfaBeta řezech

Implementace této metody spočívá ve vytvoření grafu, který obsahuje AND/OR uzly přesně tak jak jsou zobrazeny na předchozích dvou obrázcích. Ohodnocení listových uzlů lze přenechat generátoru náhodných čísel. Po inicializaci stromu a vygenerováni hodnot jeho listových uzlů přichází na řadu samotné prohledávání. Algoritmus reálné funkce se opět liší od původního rekurzivního. Odlišnost však spočívá více méně v přeházení podmínek a použití pomocných proměnných a_min a b_max, které simulují vynořování rekurze.

Vykreslování samotného stromu probíhá dle souřadnic, které byly všem uzlům přiřazeny již při generování tohoto stromu.

3.7 Závěr

V této kapitole se čtenář mohl seznámit s možnostmi implementace metod řešení úloh, pro potřeby jejich grafického znázornění. Chtěl bych ještě na závěr zdůraznit, že postupy uvedené v třetí části kapitoly se nehodí pro obecné použití, neboť jsou všechny algoritmy upraveny tak, aby se daly krokovat. Což je v mnoha případech dělá pomalejšími.

Doufám, že výsledkem této práce je hodnotná aplikace, která pomůže zájemcům o metody řešení úloh pochopit tuto problematiku.

3.8 Autorství zdrojových kódů

Při psaní zdrojových kódů jsem se svolením Ing. Přemysla Krška, Ph.D. vycházel z projektů, jenž se váží k předmětu Základy počítačové grafiky.

Literatura

[1] Zbořil, F.: Základy umělé inteligence, studijní opora, FIT VUT v Brně, 2006.

[2] Veselý, A.: Úvod do umělé inteligence, Praha, Reprografické studio PEF ČZU v Praze, 2005

[3] Kilgard, M. J.: The OpenGL Utility Toolkit - Programming Interface, Silicon Graphics, Inc., http://www.opengl.org/documentation/specs/glut/glut-3.spec.pdf, 1996

Seznam příloh

Příloha 1. Seznam stavů programu

Příloha 2. CD s aplikací Grafické animace metod řešení úloh, manuálem a zdrojovými texty.

Příloha 1

Hlavní menu

- *Džbány*
  - ↘ **BFS**
    - ↘ Navigace
  - ↘ **DFS**
    - ↘ Navigace
  - ↘ **DLS**
    - ↘ Nastavení hloubky
      - ↘ Navigace
  - ↘ **IDS**
    - ↘ Navigace
  - ↘ **Backtracking**
    - ↘ Navigace

- *8 Puzzle*
  - ↘ **BFS**
    - ↘ Nastavení počátečního stavu
      - ↘ Navigace
  - ↘ **BS**
    - ↘ Nastavení počátečního stavu
      - ↘ Navigace
  - ↘ **A* search**
    - ↘ H1
      - ↘ *Nastavení počátečního stavu*
        - ↘ Navigace
    - ↘ H2
      - ↘ *Nastavení počátečního stavu*
        - ↘ Navigace

- *Cesta z A do B*
  - ↘ **UCS**
    - ↘ Výběr bodů A a B
      - ↘ *Navigace*
  - ↘ **Best First Search:**
    - ↘ Výběr bodů A a B
      - ↘ *Navigace*
  - ↘ **Greedy search**
    - ↘ Výběr bodů A a B
      - ↘ *Navigace*
  - ↘ **Hill-climbing,**
    - ↘ Výběr bodů A a B
      - ↘ *Navigace*
  - ↘ **Simulated annealing**
    - ↘ Výběr bodů A a B
      - ↘ *Navigace*

- *Úloha čtyř dam*
  - ↘ *Výběr počtu dam*
    - ↘ **Backtracking for CSP**
      - ↘ Navigace
    - ↘ **Forward checking**
      - ↘ Navigace
    - ↘ **Min-conflict**
      - ↘ Navigace

- *Zápalky*
  - ↘ **BFS**
    - ↘ Navigace
  - ↘ **DFS**
    - ↘ Navigace
  - ↘ **DFS -1**
    - ↘ Navigace

- *AlfaBeta graf*
  - ↘ **Navigace 1**

\- **Navigace**
- Spustit <-> Stop -> [Restart]
- Další
- Zpět

\- **Navigace1**
- Spustit <-> Stop -> Restart
- Další
- Generovat
- Zpět