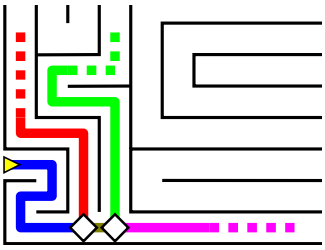
Backtracking

1 Stavový prostor úloh

Mnoho algoritmů z oblasti hraní her a umělé inteligence řeší takzvané rozhodovací problémy. Hledání řešení u takových problémů spočívá v postupném zkoušení posloupnosti rozhodnutí.

Názorným příkladem takového problému je hledání cesty v bludišti. Na každé křižovatce je potřeba se rozhodnout, jakým směrem pokračovat. Posloupnosti některých rozhodnutí pak vedou k cíli a jiné posloupnosti rozhodnutí k cíli nevedou.



Množině všech rozhodovacích posloupností se říká *stavový prostor problému*. Cílem algoritmu, který řeší tento problém je v tomto stavovém prostoru hledat buďto jedno nebo všechna řešení (cesty, které vedou k cíli).

Pokud je řešení problému více, můžeme řešení porovnávat a usilovat o to nejlepší, kterému pak říkáme *optimální řešení* (např. nejkratší cesta bludištěm).

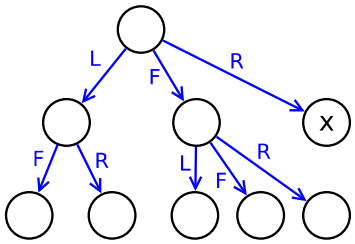
Některé úlohy jsou tak složité, že dopředu neznáme velikost jejich stavového prostoru. Tento stavový prostor může být i nekonečný. Potom nevíme, kolik existuje řešení a často ani nevíme, jak vypadá optimální řešení. U takových úloh můžeme v omezeném čase získat jen omezený počet řešení. Pokud vybíráme nejlepší řešení z omezeného počtu nalezených, říkáme mu *suboptimální řešení*.

Příkladem takového problému může být hledání cesty v nekonečném nebo předem neznámém bludišti. Do stejné kategorie spadají i problémy různých strategických a tahových her, jako jsou třeba šachy. Jejich stavový prostor je obrovský, potenciálně nekonečný.

2 Prohledávání stavového prostoru

Stavový prostor zmíněných úloh si často představujeme pomocí grafových datových struktur, zejména jako strom. Pro implementaci algoritmů, které se stavovým prostorem pracují ovšem není vždy potřeba takovou datovou strukturu používat explicitně. Často je tato struktura jen myšlená, tj. pomáhá nám přemýšlet nad řešením, aplikovat znalosti o ní, ale není nutné ji implementovat a zabírat tak operační paměť. Zvláště u nekonečných stavových prostorů by nám jí jaksi chybělo, že…

Pro úlohu s bludištěm by mohl vypadat myšlený strom jednotlivých řešení nějak takto.



Uzly zde představují křižovatky, hrany pak rozhodnutí, kterým směrem se vydat dále (L – vlevo, R – vpravo, F – dopředu).

Při hledání řešení nyní můžeme využít znalostí algoritmů pro průchody stromem. Chceme-li projít všemi možnými stavy, použijeme *průchod do hloubky*. Takovému řešení se říká hledání hrubou silou. Jeho nevýhodou je extrémní časová náročnost. U problémů s nekonečným stavovým prostorem bude mít strom potenciálně nekonečnou hloubku. Například pokud nebudeme v bludišti detekovat cykly, hrozí, že v nich budeme bloudit donekonečna.

Lepší strategií bývá *průchod do šířky*, kdy se strom prochází do zvolené hloubky. Nenajde-li se řešení ve zvolené hloubce, může se postupně prohledávaná hloubka zvyšovat, až na nějaké řešení narazíme.

Tato strategie má několik výhod. U úlohy typu bludiště částečně řeší cykly (ještě lepší je cyklus skutečně detekovat). Další výhodou je, že díky ní jako první narazíme na řešení s nejmenším počtem rozhodnutí (což ale nemusí být nutně optimální řešení).

3 Backtracking

Algoritmus backtrackingu můžeme česky nazvat jako *prohledávání s návratem*. Principem algoritmu je využití rekurze pro průchod (myšleným) stavovým stromem rozhodovací úlohy. Jakmile v nějakém stavu řešení zjistíme, že dané řešení je hledaným řešením (pak jej zaznamenáme), nebo v dané cestě nejde nebo nemá smysl pokračovat, vrátíme se k poslednímu bodu rozhodnutí a rozhodneme se v něm jinak, tj. vydáme se jinou cestou.

Algoritmus se v každém bodu rozhodnutí rekurzivně zanoří a pro návrat k poslednímu bodu rozhodnutí proto stačí, aby se z rekurze vynořil. Použití implicitního zásobníku nám umožňuje si pamatovat předchozí rozhodnutí bez nutnosti implementovat dynamickou datovou strukturu.

4 Heuristika

Právě popsaný algoritmus je sice systematické řešení problému, ale v základní verzi jde pořád o řešení hrubou silou s velkou časovou náročností. Řešení jde podstatně urychlit, když dokážeme vyhodnotit, že v některých cestách nemá cenu pokračovat. Tím dojde k redukci prohledávaného stavového prostoru a zásadnímu zrychlení.

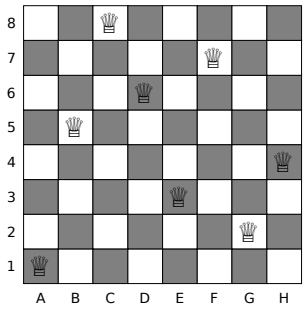
Obecně každé opatření, které vede k podstatnému urychlení takového algoritmu se nazývá *heuristika*. Vytváření heuristik je svého druhu umění a nejde jej automatizovat ani dát jednoznačný návod, jak je vymýšlet. Pokud ale studujete známá řešení podobných úloh, můžete se jimi inspirovat a občas se vám povede vymyslet své vlastní kreativní řešení.

5 Úloha osmi dam

Toto je klasická úloha pro demonstraci algoritmů založených na backtrackingu s použitím heuristik. Výhodou této úlohy je, že stavový prostor je vzhledek k výkonu dnešních počítačů relativně malý a jde jej prozkoumat v reálném čase i hrubou silou. Zároveň jsou známa všechna řešení. Úloha tedy umožňuje tedy zkoušet různé heuristiky, porovnávat řešení a matematicky zkoumat jejich složitosti. Používá se proto jako školní úloha.

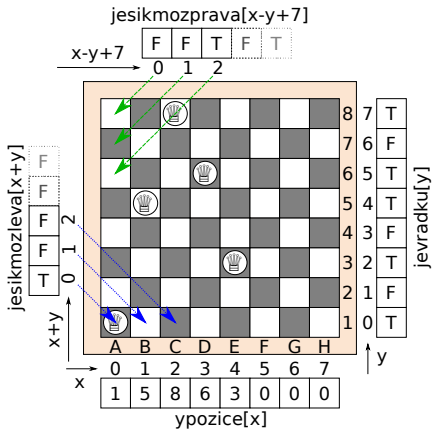
Edsger Dijkstra tuto úlohu použil v roce 1972 pro demonstraci síly strukturovaného programování a algoritmu backtrackingu. Řešení, kterým se budeme zabývat v tomto cvičení navrhl v roce 1976 Niklaus Wirth.

Zadáním úlohy je najít všechna rozmístění osmi dam na šachovnici rozměru 8×8 políček, kdy se žádné dvě dámy neohrožují.



Tuto úlohu budeme řešit backtrackingem, tj. pomocí rekurzivního zkoušení různých variant rozložení. Protože nemá smysl, aby byly dvě dámy v jednom sloupci, budeme zkoušet kombinace dam, kdy v každém sloupci bude vždy právě jedna dáma.

Naivní řešení spočívající v umísťování dam do dvourozměrného pole a procházení pozic v osmi směrech od dámy, abychom zjistili, zda není ohrožena jinou dámou by bylo algoritmicky náročné a pomalé. Heuristika navržená Wirthem spočívá v chytré reprezentaci úlohy pomocí datové struktury, která umožní zjistit ohrožení dámy v jediném kroku výpočtu. Vůbec nebude potřeba procházet šachovnicí, protože ji pro řešení této úlohy vůbec nebudeme potřebovat. Princip navrženého řešení je patrný z obrázku.



Samotnou šachovnici budeme reprezentovat osmiprvkovým polem `ypozice`, obsahujícím pozice každé dámy v jejím sloupci. Dále využijeme tři pomocná pole logických hodnot pro vyznačení, že dáma na pozici x, y ohrožuje

všechny pozice ve vodorovném směru (pole `jevradku`) a v šikmých směrech (pole `jesikmozleva` a `jesikmozprava`). Pole `jevradku` má délku 8 prvků, pole pro šikmé směry mají délku 15 prvků. V obrázku je rovněž vyznačeno, jak je potřeba jednotlivá pole indexovat pomocí souřadnic x, y. Pro snazší práci tato pole zabalíme do struktury, kterou budou využívat jednotlivé podprogramy.

```
typedef struct {
  char ypozice[8];
  bool jevradku[8];
  bool jesikmozleva[15];
  bool jesikmozprava[15];
} TSach;

typedef struct {
  /* lin. seznam */
} TReseni;
```

Pro záznam jednotlivých nalezených řešení použijeme lineární seznam.[1]

Na začátku programu je potřeba datovou strukturu správně inicializovat. Zavedeme si pravidlo, že v poli ypozice bude hodnota 0 znamenat, že v daném sloupci ještě žádná dáma není položena. Pozice dam pak budou zaznamenány pomocí hodnot 1–8. Na začátku bude v logických polích ve všech prvcích hodnota `false`, protože zatím není ohrožena žádná pozice ze žádného směru.

```
TSach plocha = {
.ypozice = {0},
.jevradku = {false},
.jesikmozleva = {false},
.jesikmozprava = {false},
};
```

Položení dámy na souřadnici x, y bude znamenat změnu hodnot v jednotlivých polích. Tato pole je potřeba je indexovat výrazy složenými z hodnot x, y tak, jak je to vyznačeno v obrázku. Do pole `.ypozice` se na index x zapisuje hodnota y+1 a do logických polích na odpovídající pozice hodnota `true`. Při odebrání dámy z pozice se provedou opačné akce.

1 Je to lepší, než nalezená řešení rovnou vypisovat. Při pozdějších vylepšováních algoritmu je možné například generovat z jednoho nalezeného řešení další řešení pomocí otáčení a symetrického převracení. Do algoritmu samotného pak jde do testování zapojit i prohledávání již nalezených řešení a ještě více zmenšit prohledávaný stavový prostor.

Klíčová je pak funkce pro test, zda je pozice na souřadnicích x, y ohrožena. Tento test se v algoritmu backtrackingu použije k tomu, abychom na ohrožené pozice dámu *vůbec nepokládali*. To je podstata heuristiky tohoto algoritmu. Pro provedení tohoto testu stačí pomocí logické spojky OR otestovat hodnoty všech tří logických polí.

```
plocha.jevradku[y] ||
plocha.jesikmozleva[x+y] ||
plocha.jesikmozprava[x-y+7]
```

Samotný algoritmus prohledávání s návratem s touto heuristikou je již poměrně přímočarý. Zde je jeho náčrt pomocí pseudokódu.

```
algorithm zkusSloupec(plocha, x) {
  for (y: 0 → 7) {
    if (!jeOhrozena(plocha, x, y)) {
      polozDamu(plocha, x, y)

      if (x == 7)
        zapamatuj(plocha)
      else
        zkusSloupec(plocha, x + 1)

      odeberDamu(plocha, x, y)
} } }
```

Všimněte si, že koncová podmínka rekurze je na posledním sloupci (x == 7), když se povedlo položit dámu. To, co se bude pamatovat v tomto případě bude celé pole ypozice, protože to teď obsahuje rozmístění osmi dam v osmi sloupcích.

6 Úloha – všechna řešení osmi dam

Vytvoř program, který najde a vypíše všechna řešení problému osmi dam.

1. Vytvoř podprogramy `polozDamu`, `odeberDamu` a `jeOhrozena`, které budou pracovat výše popsaným způsobem.
2. Pro ukládání nalezených řešení použij lineární seznam (modifikuj řešení z minulých cvičení).
3. Vytvoř podprogram `zkusSloupec` realizující backtracking. (Kvůli efektivitě předávej strukturu odkazem).
4. Nalezená řešení vhodně vypiš (ideálně jako šachovnici).