Metóda zakázaného hľadania aplikovaná na úlohu N dám na šachovnici.

1 Úvod

Úlohou tejto case-study je odskúšať použitie a vlastnosti algoritmu zakázaného hľadania (inak tiež Tabu Search) na probléme N dám na šachovnici. V práci najprv rozoberieme niektoré teoretické aspekty jednoduchých stochastických algoritmov.[1] Potom sa budeme zaoberať konkrétnou implementáciou na danom probléme a nakoniec vyhodnotíme dosiahnuté výsledky.

2 Stochastické optimalizačné algoritmy

Tabu Search patrí do triedy jednoduchých stochastických optimalizačných algoritmov. Tieto algoritmy neobsahujú žiadne evolučné rysy, ich základné vlastnosti sa však používajú v celej škále evolučných algoritmov.

2.1 Slepý algoritmus

Najjednoduchší stochastický optimalizačný algoritmus je tzv. slepý algoritmus. Generuje náhodne riešenia, ak najdené riešenie je lepšie ako to ktoré sme mali doteraz, zapamätá si ho.

```
Pattern BlindAlg( int tmax )
{
   int t=0;
   Pattern best;
   best.Randomize;

   while ( t<tmax )
   {
      t+=1;
      Pattern pattern;
      pattern.Randomize();

      if ( pattern.Fitness()<best.Fitness() )
      {
         best=pattern;
      }
   }

   return best;
}
```

Algoritmus pracuje nad doménou Pattern. Na začiatku sa vygeneruje náhodný vektor. Nasleduje iteračná časť (cykus while). Tam sa vždy vygeneruje náhodný vektor a porovná sa s doteraz najdeným najlepším riešením.

V prípade, že iterujeme do nekonečna sa dá pomerne jednoducho dokázať, že algoritmus generuje korektné globálne mininum.

Slepý algoritmus vo všeobecnosti neobsahuje žiadnu stratégiu generovania riešení na základe už získaných riešení. Každé riešenie je zostrojené úplne nezávisle od predchádzajúcich riešení. Zapamätáva sa riešenie s najnižšou hodnotou Fitness. Posledné je napokon aj výsledným riešením.

[1] Na rozdieľ od bežných zvyklostí nebudeme pracovať nad doménou binárnych vektorov, ale nad doménou permutacií, ktoré budeme nazývať vzorky(alebo patterny)

2.2 Horolezecký algoritmus

Zovšeobecnením slepého algoritmu je tzv. horolezecký (hill climbing) algoritmus. Na rozdieľ od slepého algoritmu sa iteračne hľadá lokálne najlepšie riešenie v určitom okolí, pričom sa toto riešenie použie ako stred pre ďalšiu iteráciu. Aby sme mohli definovať okolie, musíme si zaviesť niekoľko nových pojmov.

Jedným z nich je pojem mutácie. Požadujeme, aby mutácia mala niekoľko dobrých vlastností. Prvá je že ak P je permutácia potom aj $P' = O_{mut}(P)$ je permutácia. Ďalšia je že ak $P_{mut} \to 0$ tak $P = P'$

$$P \in S_N \implies O_{mut}(P) \in S_N \tag{1}$$

$$\lim_{P_{mut} \to 0} O_{mut}(P) = P \tag{2}$$

Jednoduchý algoritmus spĺňajúci tieto kritéria vyzerá takto:

```
Pattern Mutate( Pattern input )
{
   Pattern output=input;

   for( int i=0; i<m_nSize; i++ )
   {
      double lfRand=rand();
      lfRand/=(double)RAND_MAX;

      if ( lfRand<lfMutationProb )
      {
         int nRand=rand()*m_nSize/(RAND_MAX+1);

         swap(output[nRand],output[i]);
      }
   }

   return output;
}
```

Algoritmus najprv skopíruje vstupný pattern do výstupného. Následovne prechádza postupne cez všetky prvky a v prípade že nastane príhodná udalosť (`lfRand<lfMutationProb`) vymení prvok na pozícii `i` s iným, náhodne vybraným. Algoritmus spĺňa obidve podmienky ktoré sme si stanovili a zároveň aj podmienku stochastičnosti.

Horolezecký algoritmus používa tento druh mutácie na generovanie nového okolia. K doterajšiemu riešeniu vygenerejume operáciou mutácie množinu nových riešení, z ktorých vyberieme to najlepšie ako nový stred.

Algoritmus bude potom vyzerať nasledovne:

```
Pattern HillClimbAlg( int tmax )
{
   int t=0;
   Pattern center;
   center.Randomize;

   while ( t<tmax )
   {
      t+=1;

      PatternSet U=GenerateUSet(center);
      center=argmin( U, Fittness );
   }

   return center;
}
```

Existuje niekoľko modifikácií tohto algoritmu. Napríklad môžeme nahrádzať `center` len vtedy ak najlepší prvok množiny je lepší ako aktuálny stred. Toto nám však môže spôsobiť uviaznutie v lokálnom minime.

Podobne ako v slepom algoritme je možné dokázať, že je schopný v asymptotickom čase nájsť globálne mininum.

Jednoduché numerické aplikácie naznačujú, že aj keď neobsahuje evolučnú stratégiu, je to pomerne efektívny a robustný stochastický algoritmus, schopný nájsť pre jednoduchšie úlohy globálne minimum.

2.3 Algoritmus zakázaného hľadania (Tabu Search)

Koncom 80-tich rokov bol F.Gloverom navrhnutý algoritmus zakázaného hľadania ako rozšírenie horolezeckého algoritmu pre riešenie zložitejších optimalizačných úloh z operačného výskumu. Jednou z nevýhod horolezeckého algoritmu je, že sa po určitom počte krokov vracia k lokálnemu minumu, ktoré sa už vyskytlo v priebehu výpočtu (tzv. problém zacyklenia). Glover navrhol jednoduchú úpravu ktoré rieši tento problém. Zavedieme tzv. krátkodobú pamäť, ktorá si spätne pamätá niekoľko krokov výpočtu. Týmto jednoduchým trikom sa zabezpečí odstránenie krátkych cyklov. Modifikovaný horolezecký algoritmus takto rovnomernejšie prehľadáva celú doménu riešení a rýchlejšie nájde požadované globálne minimum.

Glover navrhol aj ďalšie metódy na zlepšenie (napríklad dlhodobú pamäť), nedokázal však vytvoriť dostatočné teoretické základy svojej metódy. Naznačuje síce nejaké argumenty, sú však poväčšinou vo veľmi všeobecnej rovine. Napriek tomu, že sa doteraz nikomu nepodarilo vysloviť silnejšie tvrdenie, metóda dáva veľmi slušné výsledky.

Rozšírenie pôvodného horolezeckého algoritmu spočíva v tom, že sa pri generovaní okolia vynechávajú vzorky nachádzajúce sa v krátkodobej pamäti. Výnimku tvorí tzv. ašpiračné kritérium. Toto kritérium porušuje reštrikciu zakázaného zoznamu, vtedy, keď existuje vzorka v menšou hodnotou ako je aktuálna.

Algoritmus zakázaného hľadania si zapamätáva históriu výpočtu v zakázanom zozname T (tabu list), slúžiacim ako krátkodobá pamäť. Na začiatku výpočtu je zoznam prázdny. Postupne sa napĺňa až na svoju maximálnu dĺžku. Do zoznamu sa vždy pridá novo nájdené riešenie a v prípade, že dĺžka zoznamu prekročila dopredu stanovený limit, poslednú položku zoznam odstránime.

Správne určenie veľkosti zakázaného zoznamu je kľúčový faktor pri hľadaní riešenia. Príliš malá hodnota neodstráni lokálne cykly a naopak príliš veľká hodnota môže spôsobiť preskočenie globálneho minima.

Zhrnutím týchto myšlienok dostávame nasledovný algoritmus zakázaného hľadania:

```
Pattern HillClimbAlg( int tmax )
{
   int t=0;
   Pattern center;
   center.Randomize;

   PatternSet TabuList=Empty;

   while ( t<tmax )
   {
      t+=1;

      PatternSet U = ( GenerateUSet(center) );
      center=argmin(
         {X|( X in U ) & ( ( X not in Tabu ) || ( X.Fitness()<center.Fitness() ) )},
         Fittness );

      TabuList.Add(center);
      if ( TabuList.Size() > nTabuMax )
         TabuList.RemoveLast();
   }

   return center;
}
```

V literatúre sú diskutované ďalšie vylepšenia tejto metódy. Najviac používaná je metóda dlhodobej pamäti. Spočíva vo vylúčení riešení, ktoré sa v predchádzajúcom výpočte najviac opakujú. Hľadanie je tak založené aj na predchádzajúcej histórii výpočtu a nie len na vlastnostiach optimalizovanej funkcie.

3 Implementácia algoritmu Tabu Search

3.1 Reprezentácia vzorky a trieda CPattern

Algoritmus som implementoval v jazyku C++ pod operačným systémom Microsoft Windows. Základným stavebným prvkom je trieda CPattern, ktorá reprezentuje permutáciu dám na šachovnici, pričom i-ty prvok permutácie obsahuje polohu dámy v i-tom stĺpci. Takáto reprezentácia problému nám zabezpečí, že sa nemusíme starať o kolízie v radoch a stĺpcoch.

Pre nás zaujímavé metódy triedy CPattern sú Randomize, Mutate a Fitness. Prvá inicializuje pattern na náhodnú permutáciu. Tu je jej výkonná časť:

```
for( int i=0; i<m_nSize; i++ )
{
   m_Data[i]=i;
}

for( i=m_nSize; i>0; i-- )
{
   int nRand=rand()*i/(RAND_MAX+1);

   int nSwap=m_Data[nRand];
   m_Data[nRand]=m_Data[i-1];
   m_Data[i-1]=nSwap;
}
```

Najprv nastavíme prvky permutácie tak, že na i-tom mieste bude číslo i. Potom postupne prechádzam prvky od konca, pričom vždy zamením jeden prvok z iným náhodne vybraným spomedzi ešte nepoužitých stĺpcov.

Funkcia Fitness nám udáva počet kolízií v danej vzorke. Toto je naše optimalizačné kritérium. Úlohou je nájsť takú vzorku, ktorá má fitness 0. Funkcia funguje podľa vzorca

$$Fitness(P) = \sum_{\substack{i,j=1 \\ (i<j)}}^{N} [\delta(p_i + j - i, p_j) + \delta(p_i - j + i, p_j)] \tag{3}$$

kde $\delta(i,j)$ je Kroneckerovo delta, $\delta(i,j) = 1$ pre $i = j$, $\delta(i,j) = 0$ pre $i \neq j$.

Poslednou je metóda Mutate, ktorá aplikuje operáciu mutácie na danú vzorku. Jej kód je až na implementačné maličkosti zhodný z tým čo bol uvedený v kapitole o horolezeckom algoritme.

3.2 Algoritmy, trieda CAlgorithm a jej potomkovia

Algoritmus je v mojej implementácii reprezentovaný abstraktonou triedou CAlgorithm. Kedže som mal v úmysle porovnať rôzne optimalizačné algoritmy má táto trieda viacej potomkou. Sú to CBlindAlg, CHillClimb a CTabuSearch. Trieda CAlgorithm implementuje základné operácie spoločné pre všetky algoritmy. Hlavná je metóda Iterate, ktoré podľa určeného počtu iterácií volá funkciu OneIter, vykonávajúcu jednu iteráciu. Je ošetrená aj možnosť asymptotického hľadania, ak sa zadá `nMaxTime=-1` bude iterácia ukončená až po nájdení globálneho minima. Tu sa dostávame aj k jednému rozdielu, medzi touto implementáciou a všeobecnou teóriou. V našom probléme poznáme presne fitness hodnotu riešenia a preto si možeme dovoliť po dosiahnutí hodnoty 0 ukončiť iterovanie. To nám tiež umožnuje implementáciu nekonečného iterovania.

Trieda CAlgorithm obsahuje naviac metódu Batch, ktorá slúži na štatistické testovanie toho ktorého algoritmu. Spušta niekoľko krát za sebou iteráciu a sumarizuje dosiahnuté výsledky v jednotlivých pokusoch.

Potomkovia triedy CAlgorithm spravidla len implementujú virtuálnu metódu OneIter().

3.2.1 Blind Search (CBlindAlg)

Trieda CBlindAlg implementuje len metódu OneIter a to veľmi triviálnym spôsobom.

```
int CBlindAlg::OneIter()
{
```

```
    CPattern pattern;
    pattern.Randomize();

    if ( pattern.Fitness() < m_Result.Fitness() )
       m_Result=pattern;

    return m_Result.Fitness();
}
```

Náhodne sa vygeneruje novy pattern a ten sa porovná so starým v prípade, že má menšiu fitness nahradí ho.

3.2.2 Horolezecký algoritmus (CHillClimb)

Trieda CHillClimb doplňa okrem implementácie OneIter ešte jednu pomocnú funkciu GenerateUSet. Táto slúži na vytvorenie okolia aktuálneho riešenia. Jej kód je opäť elementárny. Jednoduchu vygeneruje určený počet nových vzoriek mutáciou aktuálneho riešenia.

Metóda OneIter pracuje podľa štandardnej špecifikácie pre Hill Climb Algoritmus.

3.2.3 Algoritmus Tabu Search (CTabuSearch)

Trieda CTabuSearch je potomkom triedy CHillClimb. Využiva modifikovaný algoritmus generovania okolia a taktiež volá OneIter.

Metóda GenerateUSet volá funkciu CheckTabu na overenie, či sa vzorka nenachádza v zakázanom zozname. V prípade, že sa v ňom nachádza a nespĺňa ani ašpiračné kritérium, vygeneruje sa ďalšia až kým sa nenájde vyhovujúca.

Metóda OneIter jedoplnená o prácu so zakázaným zoznamom. Po vygenerovaní nového riešenia sa toto pridá na začiatok zoznamu a podľa potreby sa zo zoznamu odstráni posledný prvok.

4 Dosiahnuté výsledky

Rozsiahlejšiemu testovaniu som podrobil horolezecký algritmus a algoritmus zakázaného hľadania. Sledoval som správanie algoritmu pri zmene parametrov "veľkosť okolia" a "veľkosť tabu zoznamu". Toto všetko aplikované na rôzne veľkosti šachovnice (konkrétne pre veľkosti 10 až 15).

Pre každú kombináciu parametrov som spustil algoritmus 50 krát aby sa zmenšili vplyvy stochastičnosti na výsledok. Veľkosť okolia sa pohybovala od 4 do 10 a veľkosť tabu zoznamu od 1 do 10. Pravedepodobnosť mutácie bola počas celého výpočtu 10

Numerické výsledky sa nachádzajú v prílohe. Vo všeobecnosti možno povedať, že Tabu Search nie je v našom algoritme vždy prínos. V najlepšom prípade bol Tabu Search 2x rýchlejší ako jednoduchý horolezecký algoritmus. Oveľa väščí vlpyv na rýchlosť výpočtu mala veľkosť okolia ako veľkosť tabu zoznamu. Výsledné dáta tiež ukázali, že pri použití tejto metódy nami implementovaným spôsobom, algoritmus Tabu Search neprináša oproti klasickéhu horolezeckému algoritmu veľké zlepšenie. Ako možný dôvod sa ukazuje, že nami požitá operácia mutácie je príliš silná nato, aby bol vplyv odstránenia lokálnych cyklov relevantný.

Tieto výsledky však nemusia podávať úplne pravdivý obraz o skutočnosti. Sú zhruba dva dôvody prečo. Po prvé jedná sa o pravdepodobnostné algoritmy a teda určitú roľu hrá náhoda. Aby sme mohli element náhody dostať do rozumných medzí bolo by potrebných rádovo viac pokusov. No a po druhé testovacie intervaly parametrov nemuseli nutne zodpovedať ideálnym hodnotám pre ten ktorý algoritmus.