Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Oliver Leontiev

Zadanie 3

Problém ôsmich dám

Predmet: Umelá inteligencia
Cvičiaci: Ing. Ivan Kapustík

ZS 2020/2021

Zadanie

Našou úlohou je nájsť riešenie **problému ôsmich dám (n dám)**. Treba nájsť také rozloženie dám na štvorcovej šachovnici, že sa navzájom nebudú ohrozovať. Počet dám je rovnaký ako rozmer šachovnice, teda pre štandardnú 8 x 8 šachovnicu je dám 8. Problém máme riešiť aj pre n x n šachovnice, a teda n dám.

Problém máme riešiť **dvoma z troch** poskytnutých algoritmov:

1. Prehľadávanie v lúči
2. Zakázané prehľadávanie (tabu search)
3. Simulované žíhanie (simulated annealing)

Implementácia

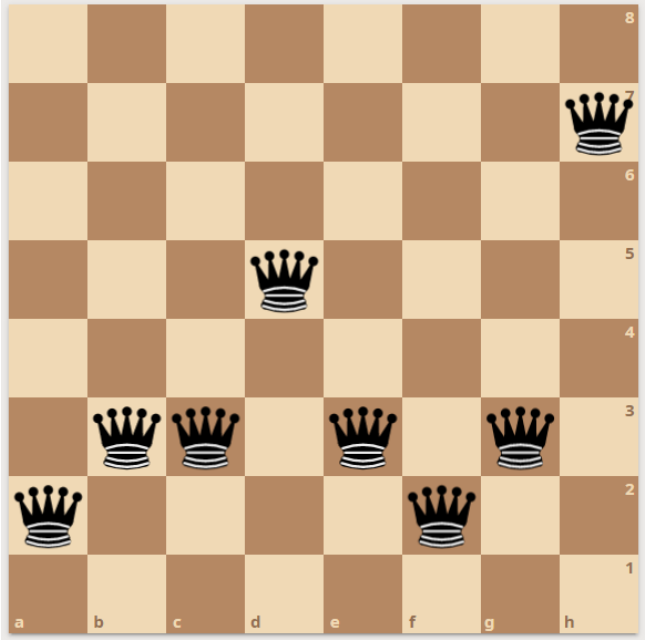
Program je konzolová aplikácia, ktorá bola vytvorená v programovacom jazyku Python 3. Spúšťa sa súbor *Main.py* a to štandardne:

$ python *Main.py*

Reprezentácia

Problém budeme riešiť vylepšovaním stavu, pričom prvý stav (prvé rozloženie dám na šachovnici) bude náhodne vygenerovaný. Ak **nepoužijeme bočné kroky** (presúvanie v riadku), vieme stav reprezentovať **jednorozmerným poľom**, kde index označuje stĺpec a číslo na indexe označuje riadok v tomto stĺpci, kde sa nachádza dáma.

Z toho vyplýva, že **stĺpce číslujeme od nuly**, aby sme využili celé pole. Zároveň, aby sme zachovali logiku pri výpočte fitness funkcie a konzistenciu reprezentácie, budeme aj **riadky číslovať od nuly**. Toto sa týka **internej reprezentácie** - pred výstupom ku každému riadku pripočítame 1.



Takýto stav by bol interne reprezentovaný ako:

[1, 2, 2, 4, 2, 1, 2, 6]

a na výstupe by bol vypísaný ako: 2 3 3 5 3 2 3 7

Základné funkcie

Predtým ako budeme hľadať riešenie potrebujeme dve funkcie: **funkciu na generovanie detí stavu** a **fitness funkciu**, ktorá zistí ako blízko je stav k riešeniu.

Generovanie detí

Dieťa stavu je stav do ktorého sa vieme dostať posunutím jednej dámy na jeden riadok. Tým pádom sa každé dieťa líši od rodiča iba na jednom indexe (ostatné dámy sú na tom istom mieste). Sám rodič sa medzi svoje deti nepočíta.

Pre n x n šachovnicu má každý stav **n * (n – 1)** detí.

Na postupné generovanie detí nám slúži trieda *ChildIter* a jej funkcia *__next__()*. Nad objektom tejto triedy voláme *iter()* a po jednom vytvárame stavy.

Generovacie funkcie v triede *ChildIter:*

```
# Reset this column and update status
def reset_column(self, next, pos):
    self.status += 1
    next[pos] = self.parent[pos]
    self.prev = next

# Generates next child state based on previously generated child state and status
def __next__(self):
    pos = self.status // 2
    next = self.prev.copy()
    if pos > self.max:
        raise StopIteration
    if self.status % 2 == 0:
        if next[pos] == 0:
            self.reset_column(next, pos)
            return self.__next__()
        else:
            next[pos] -= 1
    else:
        if next[pos] == self.max:
            self.reset_column(next, pos)
            return self.__next__()
        else:
            next[pos] += 1
    self.prev = next
    return next
```

Fitness funkcia

Fitness funkcia určuje ako blízko je nejaký stav k optimálnemu riešeniu, teda ako vhodný je daný stav. My sa snažíme minimalizovať počet na seba útočiacich dám (hľadáme **globálne minimum**), a preto sa fitness funkcia volá *count_attacks(stav)* a čím menšiu hodnotu pre stav vráti, tým je stav vhodnejší. Funkcia *count_attacks()* sa nachádza v súbore *Main.py.*

```
# Fitness function that counts the amount of attacks on the board
def count_attacks(state):
    attacks = 0
    for x1, y1 in enumerate(state[:-1]):
        for x2 in range(x1 + 1, len(state)):
            y2 = state[x2]
            if (y1 == y2) or (abs(x1 - x2) == abs(y1 - y2)):
                attacks += 1
    return attacks
```

Táto funkcia prechádza šachovnicu zľava doprava a počíta dámy na tom istom riadku a tej istej diagonále pozerajúc sa doprava. V tom istom stĺpci nemôžeme mať niečo iné ako práve jednu dámu, a teda taký útok neprichádza do úvahy.

Riešenie má fitness hodnotu 0.

Zakázané prehľadávanie (tabu search)

Prvý algoritmus na nájdenie riešenia, ktorý implementujeme je **zakázané prehľadávanie**.

Funkcia:

```
def tabu_search(size):
    MAX_TABU = 20
    tabu_states = queue.Queue()
    parent = random_state(size)
    parent_fitness = count_attacks(parent)
    while parent_fitness != 0:
        found = 0  # did we find a better state in the children
        children = ChildIter(parent, size)
        safe_state = None  # last state that isn't tabu, used to get out of local minimum
        for child in iter(children):
            if child not in tabu_states.queue:
                child_fitness = count_attacks(child)
                if safe_state is None or rn.random() > 0.5:
                    safe_state = child
                if child_fitness < parent_fitness:
                    parent = child
                    parent_fitness = child_fitness
                    found = 1
                    if parent_fitness == 0:
                        break
        if found == 0:
            if len(tabu_states.queue) >= MAX_TABU:
                tabu_states.get()
            tabu_states.put(parent.copy())
            parent = safe_state
            parent_fitness = count_attacks(parent)
    return parent
```

Postup:

1. Inicializujeme tabu list ako FIFO queue (používame modul *queue.py*) a náhodne vygenerujeme prvý stav a spočítame jeho fitness hodnotu.
2. Kým fitness rodiča nie je 0:
    a. pre každé dieťa rodiča, ktoré nie je v tabu liste:
        i. ak ešte nemáme bezpečný stav alebo s 50% pravdepodobnosťou označ dieťa ako posledný bezpečný stav.
        ii. ak má dieťa menšiu fitness hodnotu ako rodič, stane sa novým rodičom
        iii. ak má nový rodič fitness hodnotu 0, vráť sa na 2. a ukonči
    b. ak sme nenašli dieťa s menšou fitness hodnotou ako pôvodný rodič, tak pôvodný rodič je lokálne minimum:
        i. pridáme rodiča do tabu listu
        ii. novým rodičom sa stane posledný bezpečný stav

Kroky 2ai a 2bii slúžia na to, aby pri sérii lokálnych miním, ktoré zaplnia tabu list (prvé sa uvoľní a z posledného sa do neho môžeme vrátiť) program necyklil, kvôli tomu že ide stále do toho istého horšieho stavu. V podstate sa snažíme o to, aby z lokálneho minima sme vyšli do náhodného dieťaťa, ktoré nie je zakázané. Toto simulujeme tou 50% pravdepodobnosťou na označenie dieťaťa ako bezpečný stav.

Vlastnosti a lokálne minimá

Tento algoritmus sa postupne posúva k riešeniu **prechodom do najlepšieho susedného stavu**. Narazí na veľa **lokálnych miním**, z ktorých prejde do iného (horšieho) susedného stavu a zapamätá si konkrétne lokálne minimum ako **zakázané**. Má iba **krátkodobú pamäť** a preto po čase na staré lokálne minimá zabudne.

Ak by sme z konkrétneho lokálneho minima vždy prešli do toho istého stavu, tak sa môže stať, že narazíme na sekvenciu stavov, ktoré zaplnia náš tabu list a potom prídeme do stavu, ktorý sme z neho už vyhodili, v takom prípade sme spravili nekonečný cyklus. Pri prvotnom testovaní sa to stávalo často.

Tento problém sme vyriešili tým, že sme pridali náhodnosť do výberu stavu pri presune z lokálneho minima. Stále ale platí, že takéto cykly sa môžu niekoľkokrát zopakovať, kým sa nám podarí z nich vyjsť. Preto je optimálna dĺžka tabu listu veľmi dôležitá.

Maximálna veľkosť tabu listu

Pri zakázanom prehľadávaní závisí efektivita od maximálnej veľkosti listu zakázaných stavov. Opakovanými pokusmi sme odpozorovali, že optimálne veľkosti sa líšia pri rôznych rozmeroch šachovnice, ale iba minimálne a nie je potrebné to brať do úvahy:

Rozmery šachovnice	Veľkosť tabu listu a čas pre nájdenie riešenia 20-krát s tou istou sekvenciou náhodných stavov				
8 x 8	8 -> 0.426 s	10 -> 0.205 s	15 -> 0.261 s	20 -> 0.189 s	25 -> 0.264 s
10 x 10	20 -> 1.232 s	25 -> 1.367 s	30 -> 0.915 s	35 -> 1.416 s	40 -> 1.795 s
13 x 13	20 -> 5.45 s	40 -> 3.916 s	50 -> 3.829s	60 -> 3.859 s	70 -> 4.036 s
15 x 15	20 -> 5.586 s	70 -> 4.640 s	80 -> 4.478	90 -> 4.382	100-> 4.434

Ako vidíme v tabuľke, napr. pre veľkosť 15 je optimálna veľkosť listu niekde v okolí 90, zatiaľ čo pre veľkosť 8 je to okolo 20. Tieto hodnoty nie sú konzistentné a záleží aj na konkrétnych štartovacích stavoch (niektoré vygenerujú viac lokálnych miním a sú rýchlejšie pri väčšom liste, iné naopak). Napríklad pre 8 x 8 šachovnicu je čas vykonania 500 opakovaní:

pre veľkosť tabu listu 20: 5.927 s

pre veľkosť tabu listu 90: 6.275

Ale pre 15 x 15 šachovnicu je čas vykonania 50 opakovaní:

pre veľkosť tabu listu 20: 19.955 s

pre veľkosť tabu listu 90: 17.154

Takýmto skúšaním sme nakoniec vybrali **hodnotu 20 za optimálnu,** keďže má najlepšie výsledky pre 8 dám a pre väčšie počty zaručuje dostatočnú rýchlosť.

Prehľadávanie v lúči

Druhý algoritmus na nájdenie riešenia, ktorý implementujeme je **prehľadávanie v lúči**.

Funkcia:

```
def ray_search(size):
    k = 50
    limit = size  # Maximum runtime in seconds - used to determine if the algorithm is stuck
    ray = fill_ray(k)
    start = timer()
    while 1:
        all = {}
        for parent in ray:
            for child in iter(ChildIter(parent, size)):
                child_help = tuple(child.copy())  # Makes child hashable for the dict
                if child_help not in all.keys():
                    child_fitness = count_attacks(child)
                    if child_fitness == 0:  # Found the solution
                        return child
                    all[child_help] = child_fitness
        all = sorted(all.items(), key=lambda item: item[1])
        end = timer()
        if end - start > limit:  # Stuck in local minimum
            return None
        ray = fill_ray(k, all)
```

Postup:

1. Nastavíme časový limit riešenia (v sekundách) na počet dám.
2. Pole *ray[]* naplníme *k* náhodnými a rozdielnymi stavmi.
3. Cyklus:
    a. Vytvoríme všetky deti všetkých stavov z *ray[]* (bez duplikátov)
    b. Pre každé dieťa vypočítame fitness funkciu. Ak je nulová, tak sme **našli riešenie a končíme**.
    c. Zoradíme všetky deti vzostupne podľa fitness.
    d. Ak vypršal čas tak končíme, inak naplníme ray[] *k* najlepšími deťmi.
    e. Ideme na začiatok cyklu (krok 3.)

Vlastnosti a lokálne minimá

Tento algoritmus prehľadáva *k* susedstiev naraz, a potom z nich zoberie *k* najlepších susedov a pokračuje odznova (tvorí si lúč najlepších stavov). Takýto prístup znižuje počet cyklov, kým nájdeme riešenie. Zvyšuje sa počet detí, ktoré generujeme v jednom cykle - k * n * (n-1).

Problém je že na rozdiel od zakázaného prehľadávania, nemáme mechanizmus ako pokračovať, keď zapadneme do lokálneho minima. Lokálne minimá sa nám budú v lúči postupne hromadiť, až ho môžu zaplniť - v takom prípade **nikdy nenájdeme riešenie.**

Ak aj niekoľko stavov v lúči budú lokálne minimá, stále je šanca, že z ostatných sa nám podarí vygenerovať riešenie. Vždy ale existuje šanca, že naplníme lúč lokálnymi minimami a už sa cyklus nikdy neskončí. Preto určíme **časový limit**, ktorý algoritmus ukončí po jeho prekročení.

Hľadanie optimálneho *k*

Veľkosť lúča (*k*) je hodnota, ktorá nie len ovplyvňuje rýchlosť nájdenia riešenia ale aj **pravdepodobnosť zapadnutia v lokálnom minime**. Ako prioritu sme si stanovili minimalizovanie pravdepodobnosti zapadnutia v lokálnom minime. V tabuľke je zobrazené ako sme hľadali optimálne *k*:

počet zapadnutí v lokálnom minime a čas vykonania pre 50 spustení (bez zapadnutí)	k = 15	k = 20	k = 25	k = 30	k = 40	k = 50
8 x 8	0 0.852 s	0 1.07 s	0 1.06 s	0 1.240 s	0 1.702 s	0 2.133 s
10 x 10	6 2.894 s	6 3.226 s	0 4.167 s	1 5.251 s	0 6.771 s	0 7.507 s
13 x 13	6 9.795 s	8 10.928 s	3 17.913 s	1 19.644 s	1 27.624 s	1 27.046 s
15 x 15	6 20.679	3 24.803 s	2 34.427 s	5 40.543 s	0 53.217 s	0 60.34
20 x 20	*	*	*	2 158.049 s	2 213 s	0 264 s
25 x 25	*	*	*	2 481 s	*	2 15 min

* pre väčšie veľkosti šachovnice (20 x 20, 25 x 25) by výpočet všetkých polí trval veľmi dlho. Nás zaujímalo, či *k* bude dostatočné aj pre tieto prípady.

Zistili sme, že s **rastúcim *k*** sa **pravdepodobnosť zapadnutia v lokálnom minime znižuje** a pre každú veľkosť šachovnice sa nám nepodarilo nájsť hodnotu od ktorej je táto pravdepodobnosť konzistentne nulová.

Taktiež sme zistili, že **zvyšovaním *k*** sa **znižuje rýchlosť nájdenia riešenia**. Čiže sa jedná o kompromis medzi rýchlosťou a pravdepodobnosťou úspechu.

Vzťah medzi konkrétnym *k* a pravdepodobnosťou zapadnutia v lokálnom minime sa tiež líši pre rôzne veľkosti šachovnice a pre väčšie rozmery je pravdepodobnosť zapadnutia väčšia. Pokusmi sme zistili, že táto pravdepodobnosť nikdy nie je úplne nulová - **nie vždy sa nám podarí nájsť riešenie.**

Nakoniec sme sa rozhodli pre **k = 35**, ktoré je vhodným kompromisom medzi pravdepodobnosťou zapadnutia v lokálnom minime a časom vykonania najmä pre nižšie počty dám.

Testovanie

Funkčnosť

Obidve implementácie nájdu správne riešenie, ale **prehľadávanie v lúči** má šancu zapadnúť v lokálnom minime, pričom **nemá mechanizmus z neho vyjsť** (nenájdeme riešenie, ak sa to stane).

Z tabuľky vyššie (Hľadanie optimálneho *k*) a nášho testovania vyplýva, že pravdepodobnosť zapadnutia pre **prehľadávanie v lúči** je pod 10% (Okolo 2-5%).

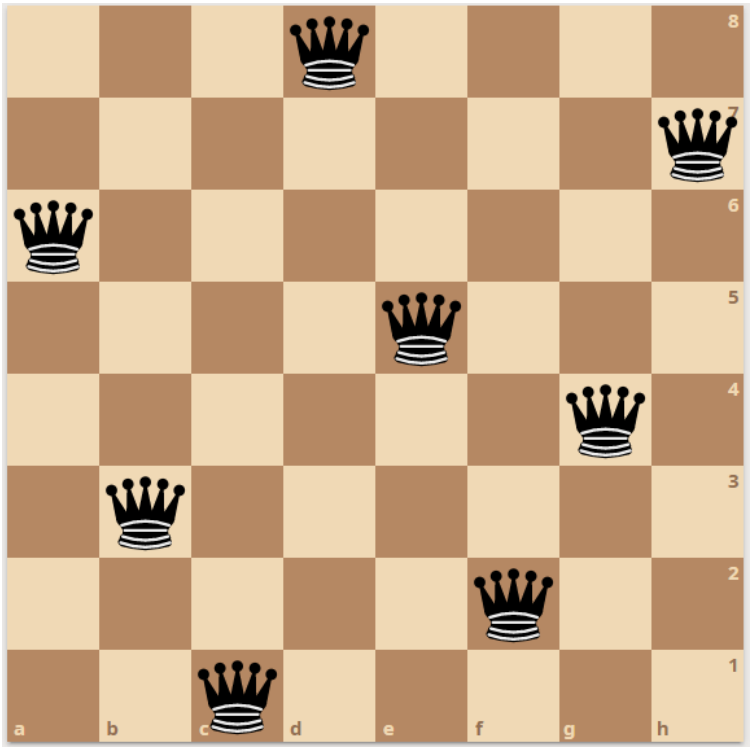
Pár príkladov pre rôzne počty dám:

8 dám:

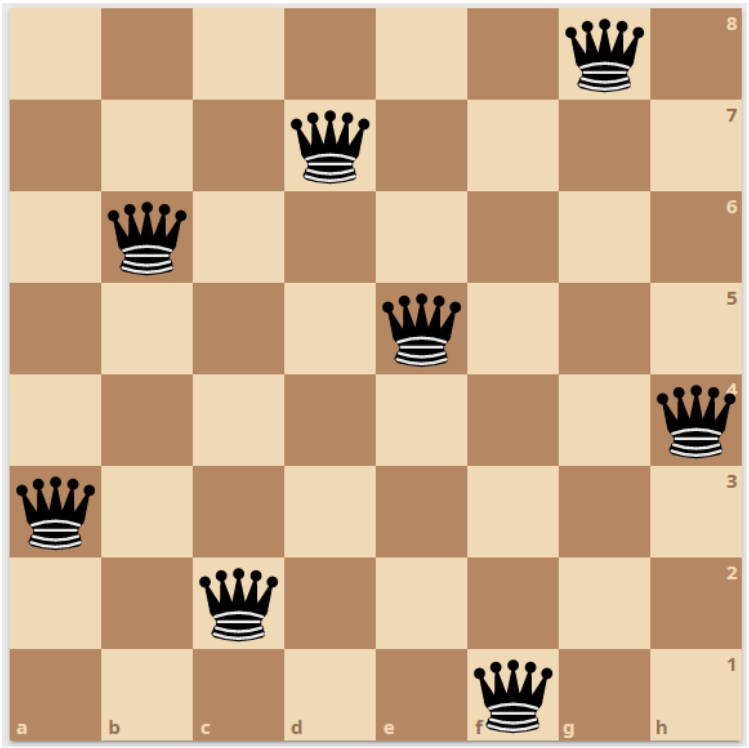
```
RAY SEARCH
6 3 1 8 5 2 4 7
0.03355009999999999

TABU  SEARCH
3 6 2 7 5 1 8 4
0.008974300000000004
```

"Ray search" riešenie:



“Tabu search” riešenie:



10 dám:

```
Number of queens: 10
RAY SEARCH
9 6 4 7 1 8 2 5 3 10
0.09711859999999994

TABU  SEARCH
5 8 6 1 3 7 9 4 2 10
0.013568599999999709
```

```
RAY SEARCH
8 6 2 7 10 1 3 5 9 4
0.09189690000000006

TABU  SEARCH
5 3 10 4 7 9 2 6 8 1
0.16922670000000029
```

15 dám:

```
Number of queens: 15
RAY SEARCH
15 10 8 1 3 13 6 14 11 5 12 2 4 7 9
0.7176583000000001

TABU  SEARCH
14 3 8 12 4 1 10 5 13 2 9 6 15 7 11
0.37426579999999987
```

```
Number of queens: 15
RAY SEARCH
6 14 1 4 12 5 11 2 15 13 8 3 9 7 10
0.5634380000000001

TABU  SEARCH
3 9 14 8 15 4 7 1 12 5 2 6 10 13 11
0.9917183
```

20 dám:

```
Number of queens: 20
RAY SEARCH
11 6 3 15 19 14 4 9 16 5 20 18 1 8 13 7 10 17 2 12
3.2223755

TABU  SEARCH
8 19 9 6 17 5 11 18 20 4 16 12 3 1 10 7 14 2 15 13
2.2041190999999998
```

25 dám:

```
Number of queens: 25
RAY SEARCH
17 24 14 16 9 15 22 1 7 2 11 18 23 5 10 12 6 19 21 4 20 25 8 3 13
11.623191599999998

TABU  SEARCH
6 9 15 24 18 7 22 1 8 16 14 21 2 10 23 20 5 3 13 17 19 12 4 11 25
6.7004587
```

30 dám:

```
Number of queens: 30
RAY SEARCH
24 8 16 25 19 3 9 26 17 11 20 2 4 29 15 23 1 22 18 7 14 10 5 27 30 12 6 13 21 28
25.381757800000003

TABU  SEARCH
24 6 18 16 21 13 17 14 29 5 3 30 10 25 1 12 19 27 8 11 20 15 7 22 26 9 2 28 23 4
9.654227300000002
```

Porovnanie efektívnosti

Napriek tomu, že sme vyššie uviedli pár prípadov, kedy bolo prehľadávanie v lúči rýchlejšie, tak z nášho testovania vyplýva, že **zakázané prehľadávanie je naprieč všetkými veľkosťami väčšinou rýchlejšie** (pri väčších veľkostiach rozdiel narastá). Pričom prehľadávanie v lúči môže nenájsť riešenie z dôvodu zapadnutia v lokálnom minime. Prípady, kedy bolo prehľadávanie v lúči rýchlejšie sú skôr výnimky spôsobené zlým prvotným stavom v zakázanom prehľadávaní.

Zhodnotenie

Problém osem dám sme rozšírili na problém n dám a riešili ho **zakázaným prehľadávaním** a **prehľadávaním v lúči**.

Zakázané prehľadávanie sa ukázalo ako efektívnejší algoritmus, ktorý vždy nájde riešenie, ale môže sa na nejaký čas zacykliť medzi lokálnymi minimami. Vďaka náhodnému prechodu z lokálneho minima do horšieho stavu sa nezasekne v nekonečnom cykle. Zároveň má lepšiu pamäťovú efektivitu, lebo nám umožňuje udržiavať iba krátky list zakázaných stavov a jeden stav, ktorý vyhodnocujeme.

Prehľadávanie v lúči má väčšie pamäťové zaťaženie, keďže musíme vygenerovať všetky deti *k* rodičovských stavov, aby sme ich mohli zoradiť. Zároveň sme pokusmi zistili, že je pomalšie vo väčšine prípadov a nezaručuje úspešné nájdenie riešenia (pravdepodobnosť zacyklenia je malá).