

# Slovenská technická univerzita

Fakulta informatiky a informačných technológií

Umelá Inteligencia

**Zadanie 3C**

**Problém ôsmich dám**

## 1) Úvod

Na klasickej šachovnici je rozmiestnených osem dám. Každá má svoj stĺpec. Je potrebné každú vo svojom stĺpci umiestniť tak, aby sa navzájom žiadne dve dámy neohrozovali. To znamená, že v každom riadku môže byť len jedna dáma a tiež na každej diagonále.

Problém bolo možné riešiť tromi spôsobmi:

- 1) **Prehľadávanie v lúči (Beam Search)**
- 2) **Zakázané prehľadávanie (Tabu Search)**
- 3) **Simulované žíhanie (Simulated Annealing)**

postupne popíšem vlastnosti a implementácie každého z nich.

## 2) Spoločné prvky algoritmov

Predtým ako prejdem ku konkrétnym algoritmom, popíšem ešte ich spoločné prvky.

### a) Reprezentácia údajov

Každú šachovnicu s rozmiestnením dám je potrebné nejako reprezentovať.

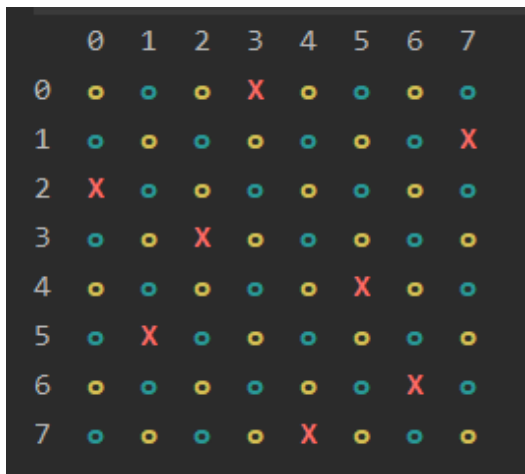
Reprezentácia u mňa pozostáva z poľa bytov, kde každý index zodpovedá stĺpcu na šachovnici a číslo na indexe hovorí o čísle riadku umiestnenia dámy.

Príklad:

Na obrázku je šachovnica, kde sú jednotlivé dámy reprezentované červeným znakom X.

Toto rozmiestnenie by bolo vo vektore reprezentované nasledovne:

[2, 5, 3, 0, 7, 4, 6, 1]



	0	1	2	3	4	5	6	7
0				X				
1								X
2	X							
3			X					
4						X		
5		X						
6							X	
7					X			

Rozmiestnenie dám

Rozmiestnenie je zároveň príkladom správneho riešenia problému – žiadne dámy sa neohrozuju.









### b) Heuristická funkcia (fitness)

Kedže pracujem s informovanými algoritmami, potrebujem nejakú heuristickú funkciu, na základe ktorej sa budú algoritmy rozhodovať.

Výstup heuristickej funkcie nazývam fitness.

Jeho výpočet spočíva v spočítaní všetkých ohrození medzi dámami.

Príklad možného ohodnotenia nasledovníkov jedného stavu je na nasledujúcom obrázku:

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
14	14	14		13	16	13	16
	14	17	15		14	16	16
17		16	18	15		15	
18	14		15	15	14		16
14	14	13	17	12	14	12	18

*Ohodnotenie nasledovníkov stavu*

Čísla znamenajú počty všetkých ohrození medzi dámami, keď sú všetky ostatné dámy na svojom mieste, len tá aktuálna by išla do toho riadku. V zobrazenom stave bez zmeny pozície je počet vzájomných ohrození 17. Dámu z prvého stĺpca ohrozujú dámy zo stĺpcov 2, 3 a 5. Dámu z druhého stĺpca ohrozujú dámy zo stĺpcov 3, 4, 6 a 8. Prvú dámu už nepočítame, lebo by sme príslušné ohrozenie počítali dvakrát.

Takže celkovo:

$$3 + 4 + 2 + 3 + 2 + 2 + 1 + 0 = 17$$

Podstatou všetkých algoritmov je postupne nachádzať stavy, ktoré majú nižšiu hodnotu fitness (menší počet ohrozených dám).

Za riešenie sa považuje stav v ktorom je hodnota fitness 0.

### c) Lokálne vyhľadávanie

Všetky použité algoritmy sú založené na lokálnom vyhľadávaní. Z náhodného počiatočného stavu sa objavujú susedné stavy, pri čom ako nasledovník sa vyberá jeden z nich.

Jedným z najjednoduchších algoritmov lokálneho vyhľadávania je **lokálne vylepšovanie (Hill Climbing)**.

Tento algoritmus spočíva vo vyberaní lepšieho nasledovníka.

Problém pri tomto jednoduchom algoritme je, že sa veľmi ľahko zasekne v lokálnom extréme – stačí ak raz nenájde lepšieho nasledovníka a končí.

Algoritmy, ktoré použijem na riešenie ja, sa snažia problém s lokálnymi extrémami riešiť rôznymi spôsobmi.

#### **d) Vlastnosti algoritmov v závislosti od parametrov a testovanie**

Vo všetkých algoritmoch sa vyskytujú parametre, ktoré značne vplyvajú na ich výkonnosť. Tento vplyv zaznamenávam pomocou testov.

Vždy otestujem percentuálnu úspešnosť algoritmu, teda pomer úspešných behov algoritmu ku všetkým pokusom a rýchlosť.

Za úspešný beh považujem taký, v ktorom sa podarilo nájsť globálne optimum, teda žiadne dámy sa neohrozujú.

Keď sa algoritmy zaseknú v lokálnom extréme, nejde to ľahko detegovať a program by mohol bežať do nekonečna, preto používam limitovanie počtu iterácií, ktorý môže program vykonať.

Skúšaním som sa snažil nájsť také limity, ktoré by nemali prerušiť beh programu, ktorý je ešte schopný nájsť riešenie.

Za neúspešný beh algoritmu teda považujem taký, ktorý prekročil limit iterácií bez nájdania riešenia (lokálne optimum za riešenie nepovažujem).

Pri simulovanom žíhaní môžu nastať aj iné prípady ukončenia algoritmu, popíšem ich neskôr. Rýchlosť zaznamenávam len pri behoch, ktoré riešenie našli. Neúspešne nezarátavam, keďže ich rýchlosť závisí hlavne od počtu povolených iterácií.

Testy sú uložené v triede Tester. Na každý algoritmus sú vytvorené 2, ich parametrami sú rozmery šachovnice, počet testovaní pre každú hodnotu parametra (čím viac, tým lepšia aproximácia) a limit iterácií.

Samotné počiatočné hodnoty konkrétnych algoritmov a ich spôsob zvyšovania sú už zapísané v kóde.

### **3) Prehľadávanie v lúči**

#### **Opis algoritmu**

Táto stratégia nezačína s jedným počiatočným stavom, ale s K stavmi. Na začiatku sa náhodne vygeneruje K stavov. V ďalšom kroku sa vygenerujú všetci potomkovia K stavov, zoradia sa a vyberie sa K najlepších, ktorí poputujú aj do ďalšieho kroku. Pri výbere sa zabraňuje pridávaniu duplicitných nasledovníkov.

Ďalej sa skontroluje, či prvý z vybraných nasledovníkov nie je riešením (stačí skontrolovať prvého, keďže sú zoradený podľa fitness). Ak je jeho fitness rovný nule, je riešením problému.

Ak sa riešenie nenašlo vykonáva sa hľadanie s vybranými najlepšími nasledovníkmi.

Výhodou vyhľadávania v lúči je, že sa potomkovia vyhodnocujú spoločne. Miesta, ktoré produkujú zlé stavy môžu teda takto postupne zaniknúť.

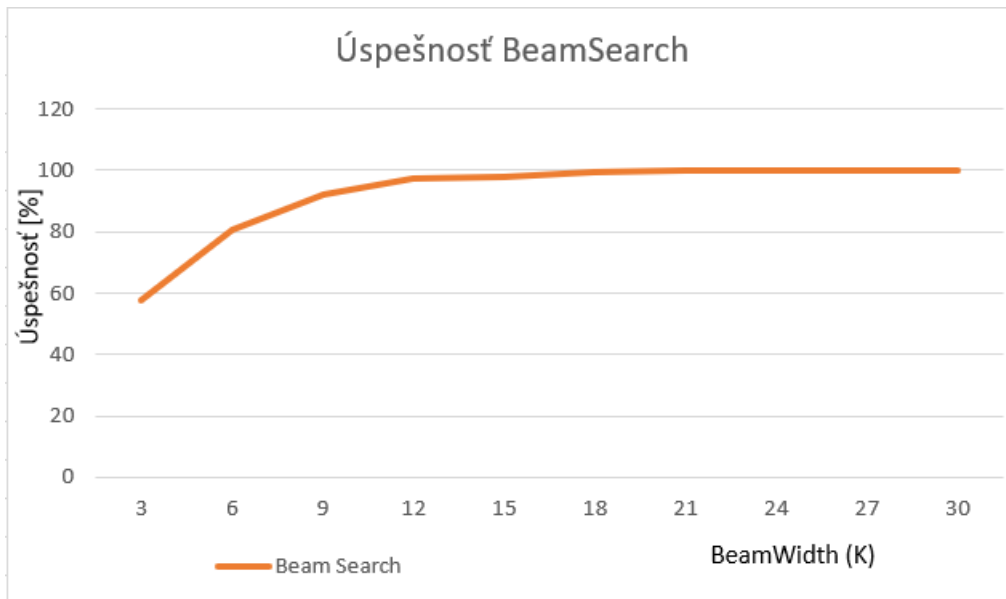
Týmto spôsobom sa stratégia snaží zabrániť uviaznutiu v lokálnom extréme, stále je však pomerne vysoká šanca, že uviaznutie nastane.

Algoritmus sa dá chápať ako viacvláknový hill climbing, s tým, že vlákna majú spoločnú pamäť.

Keby bola šírka lúča 1, algoritmus by aj ako hill climbing fungoval.

## Vlastnosti algoritmu v závislosti od hodnoty K (beamWidth)

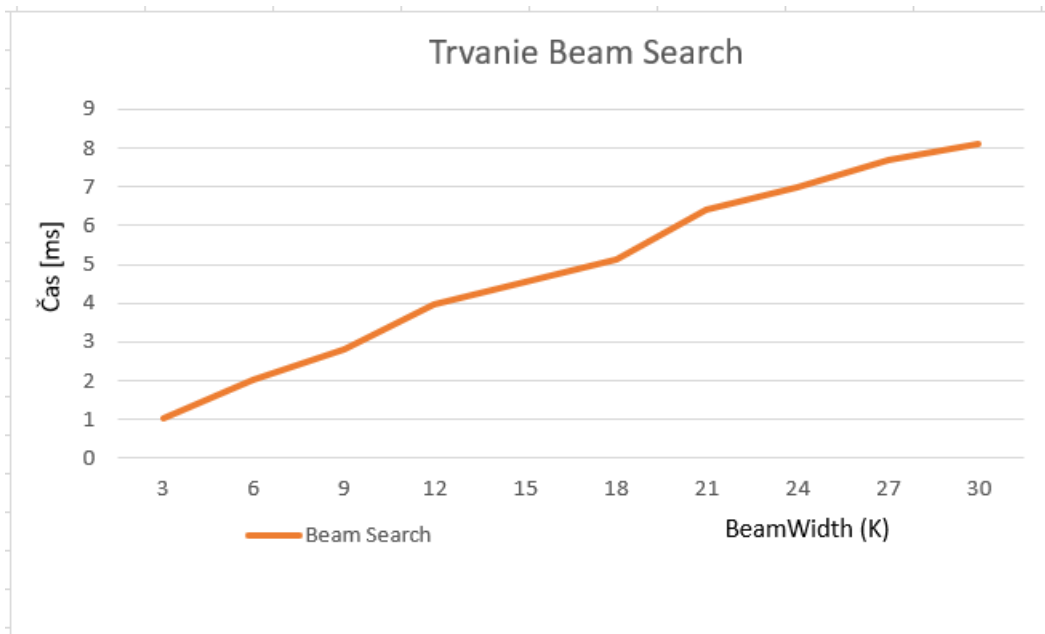
Veľmi dôležitým parametrom pri prehľadávaní v lúči je šírka lúča (hodnota K). Závisí od nej úspešnosť a taktiež aj čas vykonávania algoritmu.



Prehľadávanie v lúči: úspešnosť v závislosti od šírky lúča

Test: `Tester.BSSuccessRate()`

Parametre: `rozmary=8`, `limit iterácií = 500`, `počet testovaní = 1000`



Prehľadávanie v lúči: čas v závislosti od šírky lúča

Test: `Tester.BSTime()`

Parametre: `rozmary=8`, `limit iterácií = 500`, `počet testovaní = 1000`

Z grafov je vplyv hodnoty  $K$  na výkon algoritmu zrejmý. S narastajúcou šírkou lúča rastie úspešnosť, ale taktiež narastá aj čas, za ktorý sa algoritmus dostane k riešeniu.

Dôvodom tohto správania je, že pri menšej šírke lúča sa jedna iterácia vykoná rýchlejšie, keďže sa nemusí spracovávať veľké množstvo stavov, na druhej strane však menej stavov dokáže rýchlejšie uviaznuť v lokálnom extréme, čo vyústi v neúspech.

## 4) Zakázané vyhľadávanie

### Opis algoritmu

Pri zakázanom vyhľadávaní sa postupuje podobne ako pri lokálnom vylepšovaní. Z počiatočného stavu sa snaží dostať vždy do lepšieho stavu. Konkrétne v mojej implementácii sa vytvoria všetci potomkovia a vyberie sa najlepší. Rozdiel oproti lokálnemu vylepšovaniu je, že ak sa nenájde lepšie ohodnotený nasledovník nezostane algoritmus zaseknutý v lokálnom extréme, ale vyberie si horšie, resp. rovnako ohodnoteného potomka a aktuálny stav sa zapíše do listu zakázaných stavov. Následne sa prejde do nového stavu. V prípade, že je zoznam zakázaných stavov dlhší ako je limit, najstarší stav v ňom sa zahadzuje.

Takto sa postupuje až kým sa nenájde stav s hodnotou fitness 0.

Zakázané vyhľadávanie sa už z lokálnych extrémov dokáže vymaniť celkom efektívne, veľmi to však záleží od povolenej veľkosti zakázaného zoznamu.

### Detaily implementácie

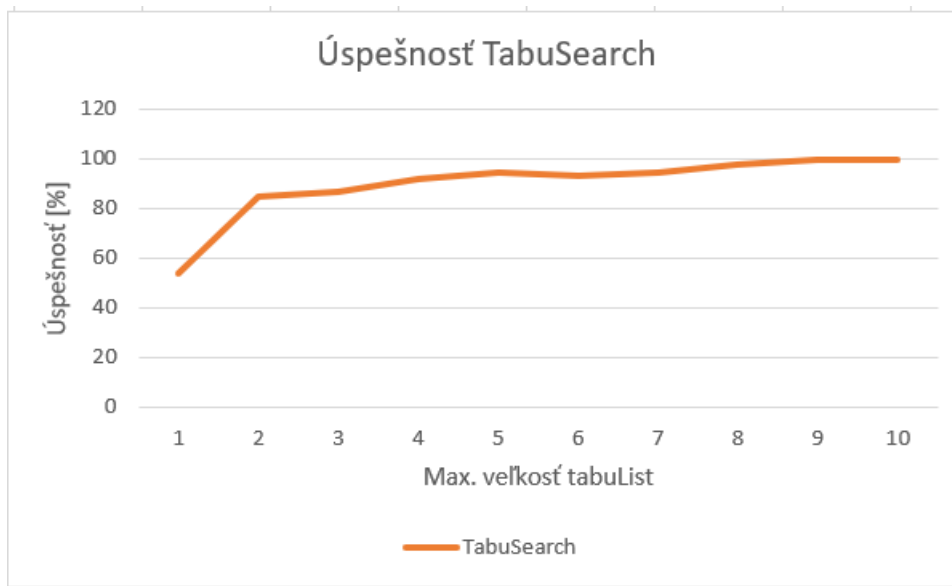
Najlepšieho nasledovníka zo všetkých vyberám opäť pomocou usporiadania podľa všetkých potomkov a následného výberu prvého.

Zoznam zakázaných stavov reprezentujem pomocou fronty, nový zakázaný stav pridám vždy na koniec a v prípade že treba nejaký stav odstrániť, odstránim ho zo začiatku.

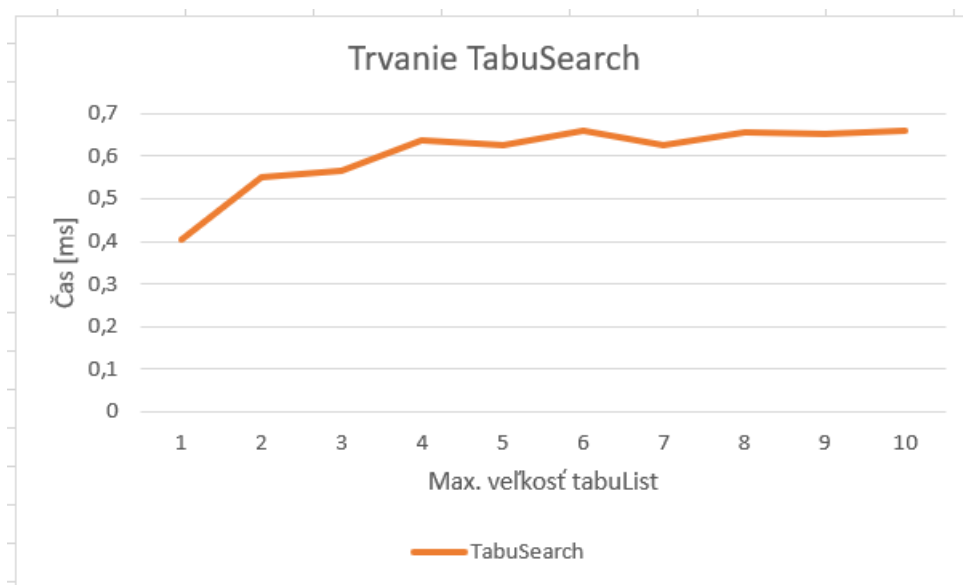
Do fronty nepridávam samotné stavy, ale ich hash kódy.

### Vlastnosti algoritmu v závislosti od dĺžky zoznamu zakázaných stavov

Maximálna dĺžka zoznamu zakázaných stavov je parameter, ktorý ma opäť vplyv aj na rýchlosť aj na úspešnosť riešenia.



Zakázané vyhľadávanie: úspešnosť v závislosti od max. veľkosti zoznamu zakázaných stavov  
Test: `Tester.TSSuccessRate()`  
Parametre: `rozmary=8`, limit iterácií = 3000, počet testovaní = 1000



Zakázané vyhľadávanie: čas v závislosti od max. veľkosti zoznamu zakázaných stavov  
Test: `Tester.TSTime()`  
Parametre: `rozmary=8`, limit iterácií = 3000, počet testovaní = 1000

Pri zakázanom vyhľadávaní sa opäť pri zvyšovaní parametra vymieňa rýchlosť za istotu nájdenia riešenia.

Pri veľmi malých maximálnych veľkostiach zoznamu je úspešnosť nízka, keďže pri malej pamäti sa algoritmus dokáže stále ľahko zacykliť (uviaznuť). Pri väčších veľkostiach zoznamu sa uviaznutie deje oveľa menej avšak trvanie sa mierne zvyšuje.

Zvolené hodnoty maximálnych veľkosti zoznamu sú však stále veľmi malé – na šachovnicu 8x8 boli dostatočné, vyššie hodnoty by správanie už moc neovplyvnili.

Pri šachovniciach väčších rozmerov by tak nízke limity nestačili a algoritmus by k riešeniu prišiel len zriedka.

## 5) Simulované žíhanie

### Opis algoritmu

Simulované žíhanie má opäť základy v algoritme lokálneho vylepšovania. Rozdiel oproti lokálnemu vylepšovaniu je v tom, že si nevyberá vždy lepšieho, ale hociktorého potomka. Pokiaľ má tento potomok lepšie ohodnotenie fitness ako aktuálny stav, prejde sa doň s pravdepodobnosťou 100%. Ak má potomok horšie alebo rovnaké ohodnotenie, prejde sa doň len s pravdepodobnosťou menšou ako 100%.

Pokiaľ sa prechod nepodarí, algoritmus skúša vyberať ďalšieho potomka. Program skončí ak sa nepodarí prejsť do žiadneho zo susedov.

Pravdepodobnosť závisí od aktuálnej teploty (parameter funkcie). Teplota sa postupne znižuje, čím klesá šanca prechodu do horšieho stavu. Cieľom algoritmu je teda spočiatku pracovať viac náhodne, prehľadať rôzne miesta stavového priestoru a potom sa postupne sústrediť na miesta s lepšími výsledkami, až príde k nejakému maximu.

Žíhanie sa dokáže vysporiadať s lokálnymi extrémami tým, že má nenulovú pravdepodobnosť výberu horšieho nasledovníka. Tento spôsob je pomerne efektívny a algoritmus by mal vždy nájsť nejaké optimum, pokiaľ je vhodne nastavený rozvrh zmien pravdepodobnosti a algoritmus beží dostatočne dlho, optimum by mohlo byť globálne.

### Detaily implementácie

Susedov aktuálneho stavu generujem všetkých naraz a následne sa z nich snažím vybrať jedného, ktorý bude pokračovať. Výber uskutočňujem náhodne – takto je menšia šanca, že algoritmus začne cykliť (keby sa postupuje vždy od prvého potomka, môže sa zacykliť ľahšie).

Náhodné čísla, ktoré generujem pri výbere nasledovníka nie sú unikátne, môže sa teda stať, že sa ten istý sused vyskúša viac krát, čo môže byť výhodou, keďže stav nemusí byť vybraný na prvý krát, ale až po viacerých pokusoch.

Aby sa vhodný nasledovník nehľadal donekonečna, je nastavený limit pokusov.

Limit počítam nasledujúcou rovnicou ( $dim$  – rozmery,  $c$  – konštanta (v implementácii som zvolil hodnotu 100), výraz  $dim * (dim - 1)$  zodpovedá počtu nasledovníkov stavu):

$$limit = dim * (dim - 1) * c$$

Ak program nestihne vybrať nasledovníka kým dosiahne limit, program končí (v prípade 8 dám vráti null – nenašiel riešenie, v inom prípade by mohol vrátiť aktuálny stav).

Program takisto skončí ak sa teplota rovná nule, alebo ako v iných prípadoch sa dosiahne maximálny počet povolených iterácií.

Dôležitá časť algoritmu je tiež samotné počítanie pravdepodobnosti v závislosti od aktuálnej teploty. To počítam nasledovne ( $T$  je aktuálna teplota,  $diff$  je rozdiel fitness aktuálneho stavu a nového vybraného):

$$p = e^{\frac{diff}{T}}; diff \in \mathbb{Z}^-, T \in \mathbb{N}, T \geq 0$$

Takto vypočítaná pravdepodobnosť nadobúda vždy hodnoty od 0 do 1. Špeciálny prípad je keď má premenná  $diff$  hodnotu 0, teda nájdený stav je rovnako dobrý ako aktuálny.

Pravdepodobnosť by v tomto prípade vždy nabrala hodnotu 1, nezávisle od teploty.

Rovnako dobre stavy si však nechcem vyberať so 100% pravdepodobnosťou, ak teda nastane situácia

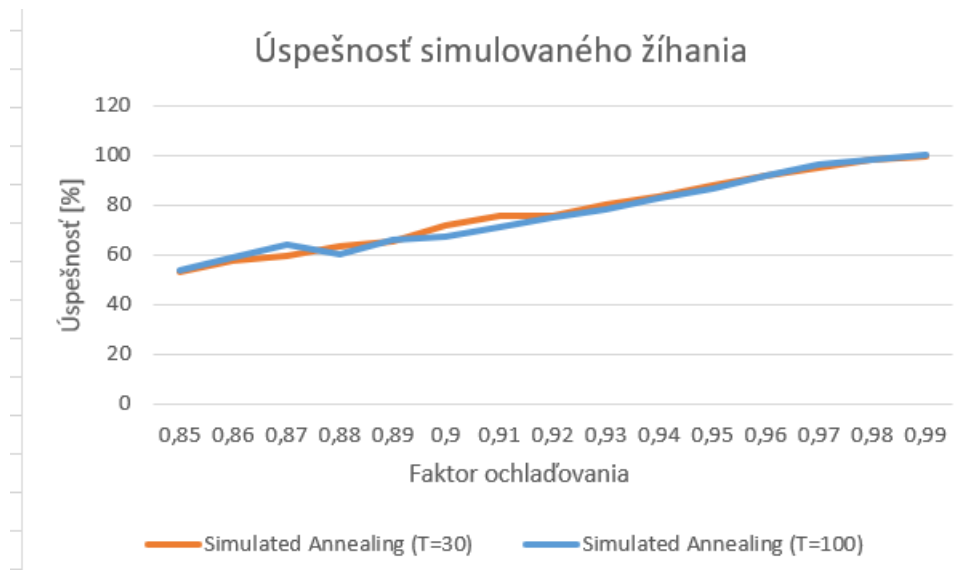


že sa  $\text{diff} = 0$ , nastavím ho na  $-0.1$ , takto je šanca prechodu do rovnako ohodnoteného stavu stále závislá od teploty a je vyššia ako pri stavoch, ktoré sú horšie.

Postupné klesanie teploty sa deje lineárne – v každej iterácii sa aktuálna teplota prenášobí koeficientom, ktorý je parametrom funkcie.

## Vlastnosti algoritmu v závislosti od počiatočnej teploty a koeficientu ochladzovania

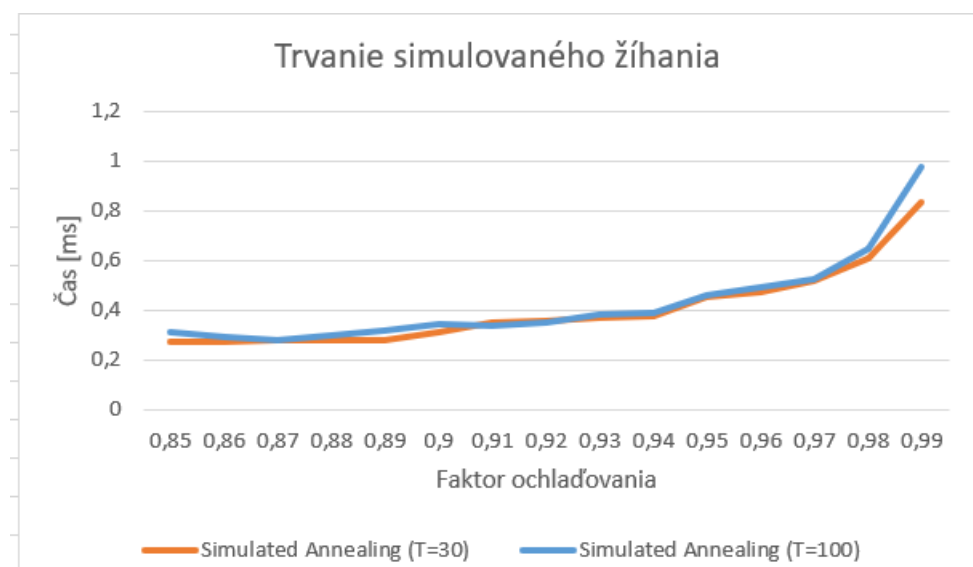
Na rozdiel od predchádzajúcich algoritmov sú pri simulovanom žíhaní až 2 voliteľné parametre. Jedným je počiatočná teplota a druhým koeficient, ktorým sa teplota po každej iterácii prenášobí. Kombináciou týchto parametrov sa vytvorí rozvrh zmeny pravdepodobnosti výberu horšieho nasledovníka.



Simulované žíhanie: úspešnosť v závislosti od faktoru ochladzovania a počiatočnej teploty

Test: `Tester.SASuccessRate`

Parametre: `rozmary=8`, `limit iterácií = 50000`, `počet testovaní = 1000`



Simulované žíhanie: trvanie v závislosti od faktoru ochladzovania a počiatočnej teploty

Test: `Tester.SATime`

Parametre: `rozmary=8`, `limit iterácií = 50000`, `počet testovaní = 1000`

Pri zmene faktoru sa znova ukázala výmena úspešnosti za čas. Príliš nízke faktory ochladovania spôsobia, že teplota klesne príliš rýchlo a globálne optimum sa nepodarí nájsť.

Naopak faktor hraničiaci s hodnotou 1 riešenia nachádza s oveľa väčšou istotou, to však za cenu dlhšieho trvania.

Počiatkové teploty som vyskúšal len 2 a vplyv tejto zmeny sa preukázal len minimálne.

Voľba počiatkovej teploty a jej vplyv na výsledky pri simulovanom žíhaní nie je triviálna záležitosť, preto som ju volil len intuitívne.

Jednou z metód na hľadanie optimálnej počiatkovej teploty je napríklad Ben-Ameurová metóda[1].

## 6) Zhrnutie

Algoritmy lokálneho vyhľadávania sa pri probléme ôsmich dám ukázali ako dosť účinné. Všetky stratégie dokázali pri vhodne zvolených parametroch dosiahnuť takmer 100% úspešnosť.

Pri prehľadávaní v lúči bol čas nájdenia riešenia rádovo vyšší ako pri simulovanom žíhaní alebo zakázanom vyhľadávaní.

Všetky implementácie podporujú aj rôzne rozmery šachovníc.

Pri vyšších rozmeroch nároky na výkon rastú a prehľadávanie v lúči prestalo byť účinné. Zakázané vyhľadávanie riešenia dokázalo nachádzať avšak maximálna dĺžka zoznamu zakázaných stavov musela byť nastavená omnoho vyššia ako pri šachovnici 8x8, čím rastie pamäťová náročnosť programu.

Simulované žíhanie sa ukázalo ako najúčinnnejšie a dokáže správne umiestniť dámy aj na veľmi veľkých šachovniciach (100+) za pomerne krátky čas. Testy s väčšími rozmermi som nezahrnul do grafov, dajú sa však pustiť priamo v programe cez konzolové užívateľské rozhranie.

## Zdroje

1. Názov: Computing the initial temperature of simulated annealing

Autor: Ben-Ameur

Dátum: 2004

Link: <https://www.mendeley.com/catalogue/8a3521ca-3e4d-362e-86d6-0d2aad69f398/>