Řešení problému vážené splnitelnosti booleovské formule pokročilou iterativní metodou

Zadání

https://moodle-vyuka.cvut.cz/mod/assign/view.php?id=48355

Algoritmus a implementace

Jazyk a závislosti

Řešení problému je napsáno v jazyce C++ s použitím standardu C++17 a knihovny Boost. Program je napsaný tak, aby jeden proces řešil jednu instanci problému. Při počítání více instancí pomocí skriptu vzniká overhead kvůli nutnosti vytváření nového procesu pro každou instanci. Konfigurační proměnná je bitové pole reprezentované fixním 128 bitovým typem uint128_t, který je součástí knihovny Boost. Pokud by bylo potřeba řešit instance N>128, program lze změnit na používání vícebitového typu (např. uint256_t nebo uint512_t) pomocí typedef configuration_type.

Algoritmus a implementace

Pro řešení problému bylo použito simulované ochlazování. Problém se dělí na 2 části. Nejdříve hledáme splnitelné řešení pro formuli a poté, když nějaké máme, hledáme co nejvíce ohodnocení literálů 1čkou, abychom maximalizovali cenovou funkci. Pro dvě konfigurace A, B se tedy algoritmus řídí následující tabulkou:

	Konfigurace B je splněna	Konfigurace B není splněna
Konfigurace A je splněna	Přijata konfigurace s větším součtem vah literálů ohodnocených 1	Přijato B s pravděpodobností v závislosti na teplotě a relativní odchylce, jinak přijato A
Konfigurace A není splněna	Přijato A s pravděpodobností v závislosti na teplotě a relativní odchylce, jinak přijato B	Přijata konfigurace s větším počtem splněných klauzulí

```cpp
//Simulated annealing
void solve_anneal(ProblemInstance& problem, double current_temperature, double cooling_coeficient, double minimal_temperature, size_t inner_cycle, double weight_coef) noexcept {
    //Current, neighbour and top configurations
    configuration_type current_solution = problem.get_random_configuration();
    configuration_type top_solution = current_solution;
    configuration_type neighbour_solution = problem.get_neighbour_configuration(current_solution, 0);

    //Best (local), neighbour and top weights
    int64_t best_sat_clauses = problem.get_sat_clauses(current_solution);
    int64_t top_sat_clauses = best_sat_clauses;
    int64_t neighbour_sat_clauses = 0;

    //Outer cycle
    while (current_temperature > minimal_temperature) {
        //Inner cycle
        for (size_t i = 0; i < inner_cycle; ++i) {
            neighbour_sat_clauses = problem.get_sat_clauses(neighbour_solution);
            //Remember the best solution ever reached, in case we converge to lower solution in rare cases
            if (neighbour_sat_clauses > top_sat_clauses) {
                top_sat_clauses = neighbour_sat_clauses;
                top_solution = neighbour_solution;
            }
            //When looking for best weights, use modifier to make weights relative rather than absolute
            //If satisfied solution is not yet found, ignore modifier
            double modifier = weight_coef;
            if (neighbour_sat_clauses < 0 && best_sat_clauses < 0) {
                modifier = 1.0;
            }
            //Solution is better or we accept worse
            if (neighbour_sat_clauses > best_sat_clauses || problem.simple_rand() < exp((neighbour_sat_clauses - best_sat_clauses) / modifier / current_temperature)) {
                current_solution = neighbour_solution;
                best_sat_clauses = neighbour_sat_clauses;
            }
            neighbour_solution = problem.get_neighbour_configuration(current_solution, neighbour_sat_clauses);
            if constexpr (GRAPH_OUT) {
                std::cout << best_sat_clauses << '\n';
            }
        }
        current_temperature *= cooling_coeficient;
    }

    if constexpr (!GRAPH_OUT && !RESULT_ONLY) {
        problem.print_solution(top_solution);
    }
    if constexpr (RESULT_ONLY) {
        std::cout << top_sat_clauses << std::endl;
    }
}
```

Algoritmus prošel postupně třemi vylepšeními. Prvním triviálním vylepšením je pamatování si nejlepší hodnoty. Pokud SA v první fázi širokého prohledávání stavového prostoru najde nějaké hodně dobré řešení a poté od něj odejde a už k němu nikdy nedokonverguje, algoritmus si ho bude pořád pamatovat. Tohle vylepšení bylo použito především v první fázi, kdy v algoritmu neexistovaly žádné jiné vylepšení a SA nemělo tak dobré výsledky.

Druhým zásadním vylepšením je použití relativní chyby dvou konfigurací. Před spuštěním simulovaného ochlazování si algoritmus spočítá součet vah (součtová norma) a ten vydělí koeficientem K, který mu uživatel zadá při spuštění. Algoritmus tedy bere jeden parametr navíc oproti běžnému SA. Nejlepší hodnota K se ukázala být v rozmezí od 5 do 10 a bude probrána podrobněji v sekci o nastavování parametrů. R = SUM(weights) / K. Hodnota R se poté použije když hledáme řešení s lepší cenovou funkcí (rozdíl vah se vydělí R a pak ještě teplotou). Tohle vylepšení řeší, že nemusíme nastavovat parametry teploty v závislosti na velikosti vah instance. Algoritmus tedy nezajímá, jestli používáme váhy v rozsahu stovek nebo milionů. Vylepšení lze vypnout nastavením K na 0.

```
250  void solve_start(ProblemInstance& problem, double startingTemp, double coolingCoef, double minTemp, size_t innerCycle, double coef) noexcept {
251      int64_t weightSum = 0;
252      for (auto& w : problem.m_weights) {
253          weightSum += w;
254      }
255      //If coeficient near 0, ignore it
256      if (coef > -0.01 && coef < 0.01) {
257          coef = 1.0;
258      } else {
259          coef = weightSum / coef;
260      }
261      solve_anneal(problem, startingTemp, coolingCoef, minTemp, innerCycle, coef);
262  }
```

Poslední vylepšení zlepšuje hledání sousedů. Pro každý literál si držíme cache, která říká kolikrát byl při vyhodnocování literál nepravdivý. Při hledání souseda flipneme literál, který má největší hodnotu v cache a jeho cache vynulujeme. Cache se nikdy neresetuje (pouze při použití jednotlivých literálů) a tedy dříve nebo později flipneme všechny literály. Do jisté míry by se cache dala představit jako prioritní fronta. Tohle vylepšení je velmi dobré pro fázi hledání splnitelné formule. Vylepšení lze vypnout nastavením constexpr USE_CACHE na false.

```
102      //Returns true if clause is satisfied by configuration, false otherwise
103      //Generates 3 functions (tail recursion) at compile time, requires C++17 for if constexpr
104      template <size_t I = 2>
105      bool is_clause_true(configuration_type& configuration, const clause_type& clause) noexcept {
106          //Negative literal
107          if (std::get<I>(clause) < 0) {
108              //0 & 1 => true, 1 & 1 => false
109              if (!(configuration & (configuration_type(1) << (-std::get<I>(clause) - 1)))) {
110                  return true;
111              }
112              //Not satisfied, add to cache
113              if constexpr (USE_CACHE) {
114                  ++m_sat_cache[(-std::get<I>(clause)) - 1];
115              }
116          //Positive literal
117          } else {
118              //1 & 1 => true, 0 & 1 => false
119              if (configuration & (configuration_type(1) << (std::get<I>(clause) - 1))) {
120                  return true;
121              }
122              //Not satisfied, add to cache
123              if constexpr (USE_CACHE) {
124                  ++m_sat_cache[std::get<I>(clause) - 1];
125              }
126          }
127
128          //End condition
129          if constexpr (I == 0) {
130              return false;
131          } else {
132              return is_clause_true<I - 1>(configuration, clause);
133          }
134      }
```

```
82   //Gets neighbour configuration (flips one bit)
83   configuration_type get_neighbour_configuration(configuration_type& configuration, int64_t sat_clauses) noexcept {
84       if constexpr (USE_CACHE) {
85           if (sat_clauses < 0) {
86               int64_t shift = 0, max = 0;
87               for (size_t i = 0; i < m_sat_cache.size(); ++i) {
88                   if (m_sat_cache[i] > max) {
89                       max = m_sat_cache[i];
90                       shift = i;
91                   }
92               }
93               m_sat_cache[shift] = 0;
94               return configuration ^ configuration_type(1) << shift;
95           }
96       }
97
98       std::uniform_int_distribution<std::mt19937_64::result_type> distr(0, m_variable_count);
99       return configuration ^ configuration_type(1) << distr(m_random_number_generator);
100  }
```

Spuštění programu

Parametry: <P> - počáteční teplota, <O> - ochlazovací koeficient, <F> - koncová teplota, <I> - počet iterací vnitřního cyklu, <K> - relativní koeficient chyby (druhé vylepšení)

Pro Windows (powershell script):
Get-Content <path to problem> | .\satsolver.exe <P> <O> <F> <I> <K> >> <path to output>

Pro Linux (shell script):
.\satsolver.exe <P> <O> <F> <I> <K> < <path to problem> > <path to output>

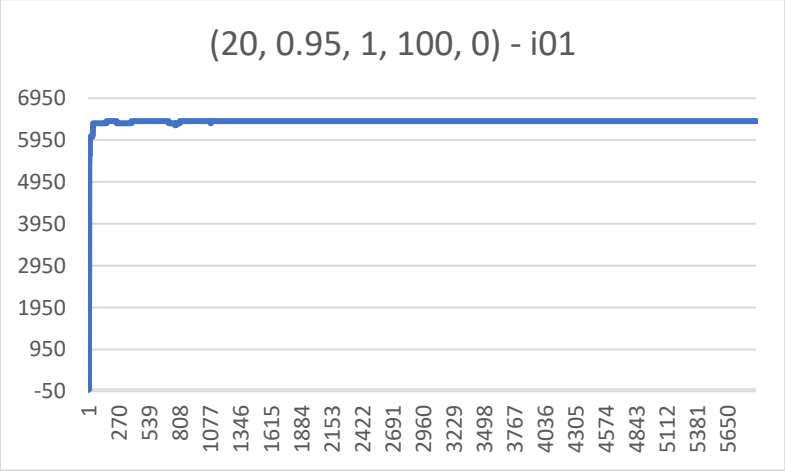
Práce s heuristikou

Nastavení parametrů

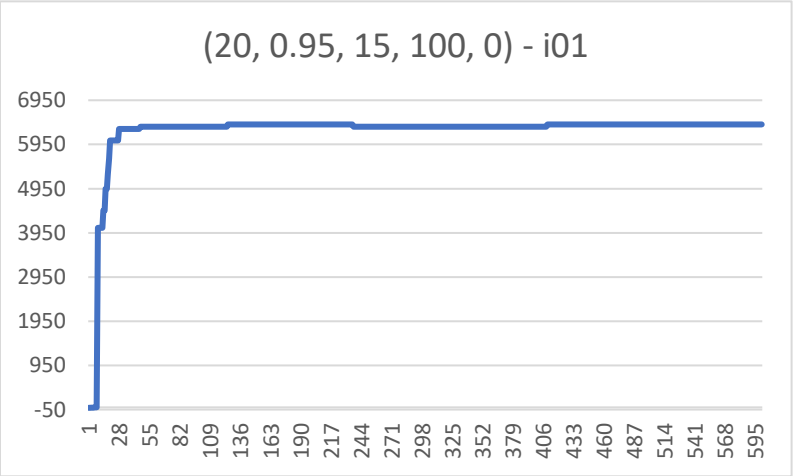
Vybereme jednu náhodnou instanci lehké sady WUF-M-20-78 a získáme parametry, které ji dobře řeší. Tyto parametry pak budeme testovat přes všechny instance WUF-M a vylepšovat je. Poté budeme tyto parametry testovat na složitějších sadách a případně je upravovat a zpětně kontrolovat, že se lehčí sady nerozbijí. Parametry budou uvedeny vždy ve formátu uspořádané 5tice (P, O, F, I, K).

Počáteční nastavení

Pro počáteční nastavení použijeme instanci WUF-M-20-78 wuf20-01, která má optimální řešení 6403 a náhodně zvolené parametry (20, 0.95, 1, 100, 0). Pozn: Pro zatím je heuristika relativní chyby vypnuta (K=0).

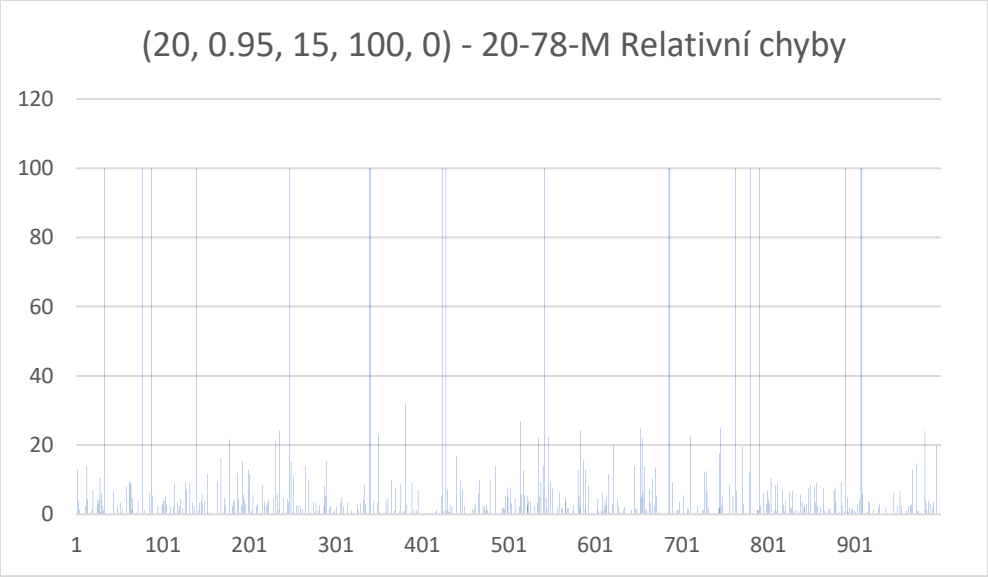


Z grafu lze vidět, že poměrně rychle konvergujeme k optimálnímu řešení a poté zbytečně dlouho hledáme lepší řešení, které už neexistuje. Optimální řešení bylo spolehlivě nalezeno po 1600 cyklech a my jsme počítali 5700 cyklů. Zvětšíme koncovou teplotu, čímž utneme algoritmus rychleji. Nové parametry zvolíme tedy jako (20, 0.95, 15, 100, 0).

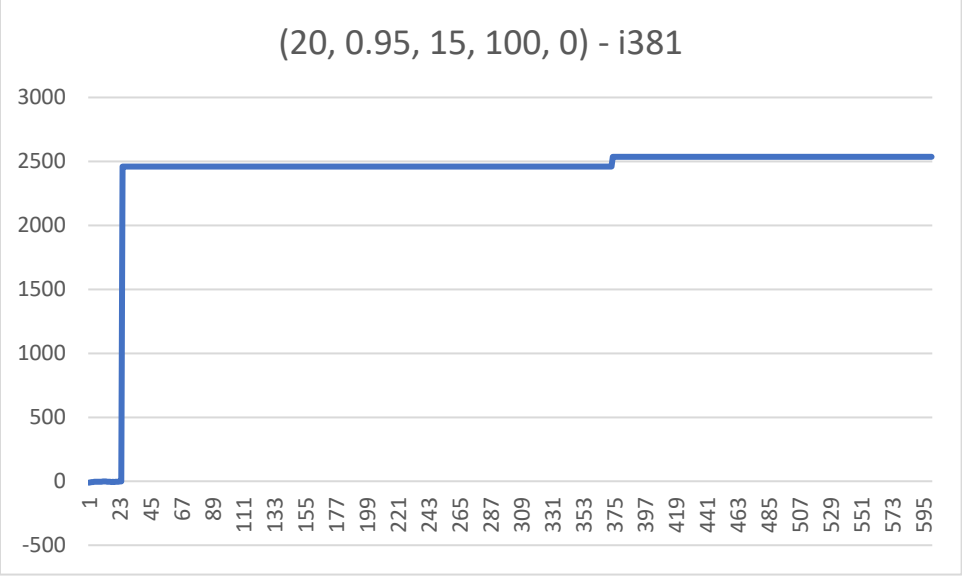


Podařilo se zkrátit počet cyklů na 1/10 při zachování optimálního řešení. Tyto parametry tedy vyzkoušíme na všech instancích WUF-M-20-78 a podíváme se na relativní chyby.

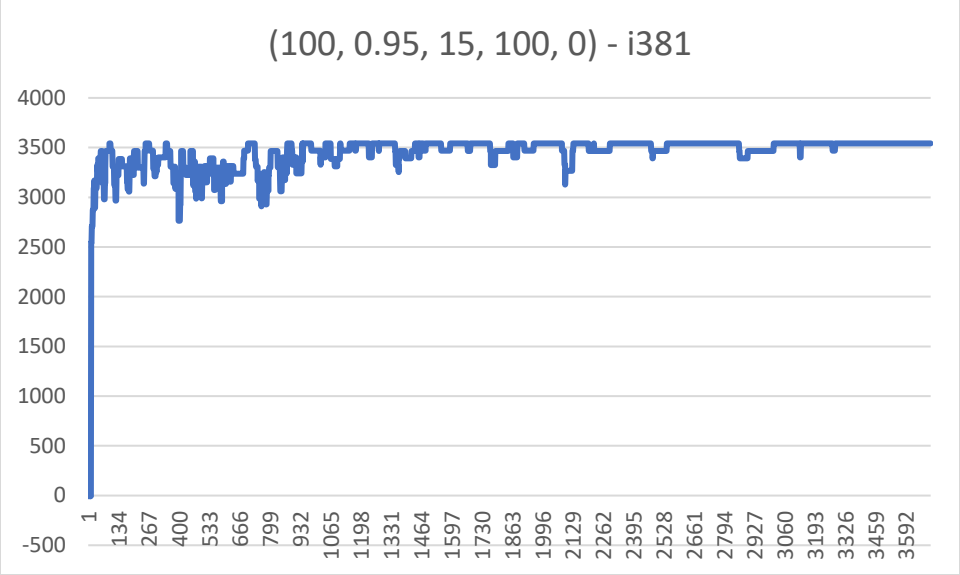
Vylepšování parametrů na lehkých sadách



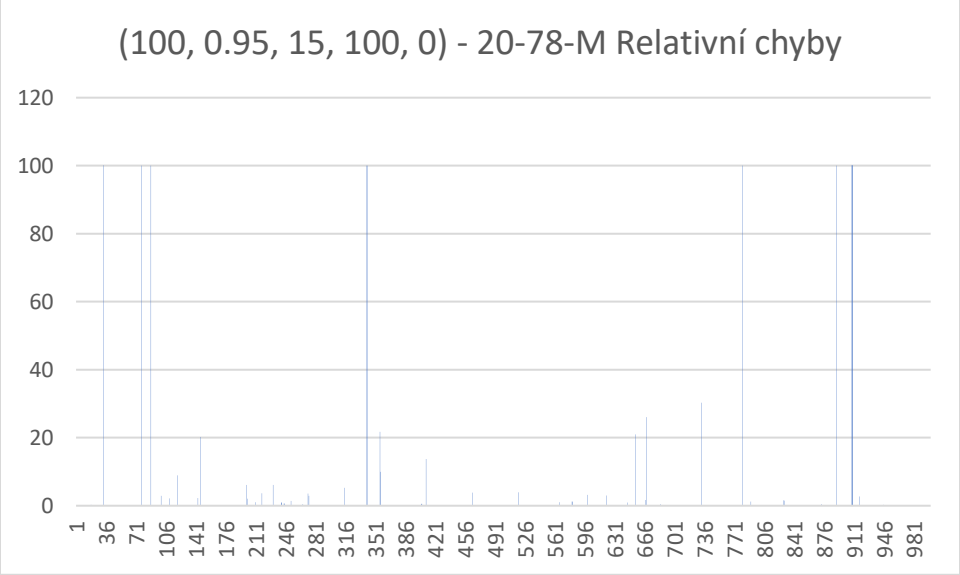
Průměrná chyba parametrů je 3.78% a maximální je 100% (řešení nenalezeno). Z tohoto grafu nepoznáme, co je při vývoji špatně. Zaměříme se tedy opět na jeden konkrétní problém. Vybereme instanci s id 381 a optimálním řešením 3542, která měla nejvyšší chybu, která nebyla 100% (32.2%). A podíváme se detailně, co se stalo.



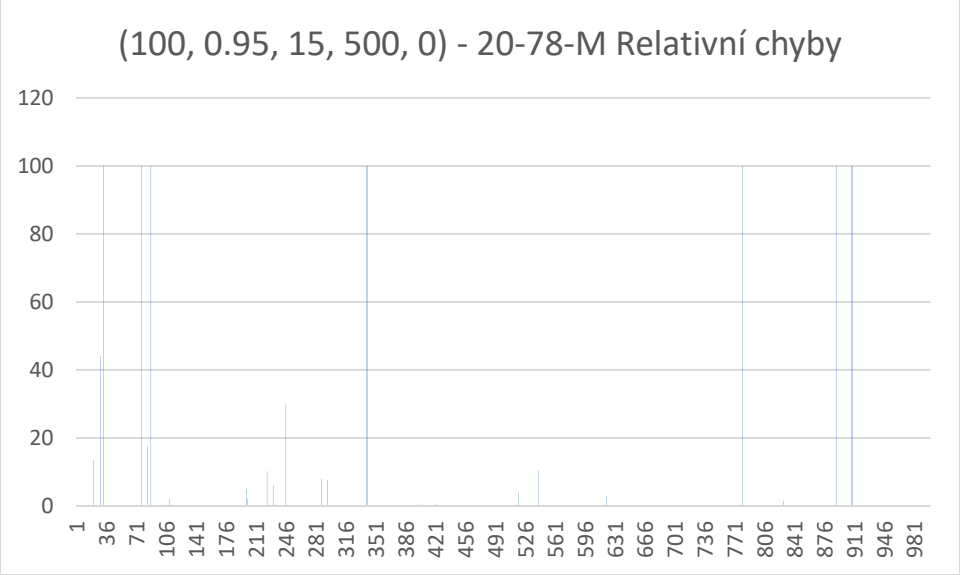
Algoritmus hned na začátku najde nějaké suboptimální řešení a hned k němu konverguje (uvázne v lokálním minimu). Nesnaží se o náhodné prohledávání sousedů se zhoršením. Toho se dá dosáhnout posunutím počáteční teploty na nějaké vyšší číslo. Vybereme tedy parametry (100, 0.95, 15, 100, 0) a podíváme se co se změní.



Graf funkce už vypadá lépe. Na začátku příjmá horší řešení a prohledává stavový prostor I se zhoršením a ke konci konverguje k optimálnímu řešení. Vyzkoušíme tedy tyto parametry opět na všechny instance WUF-M-20-78.

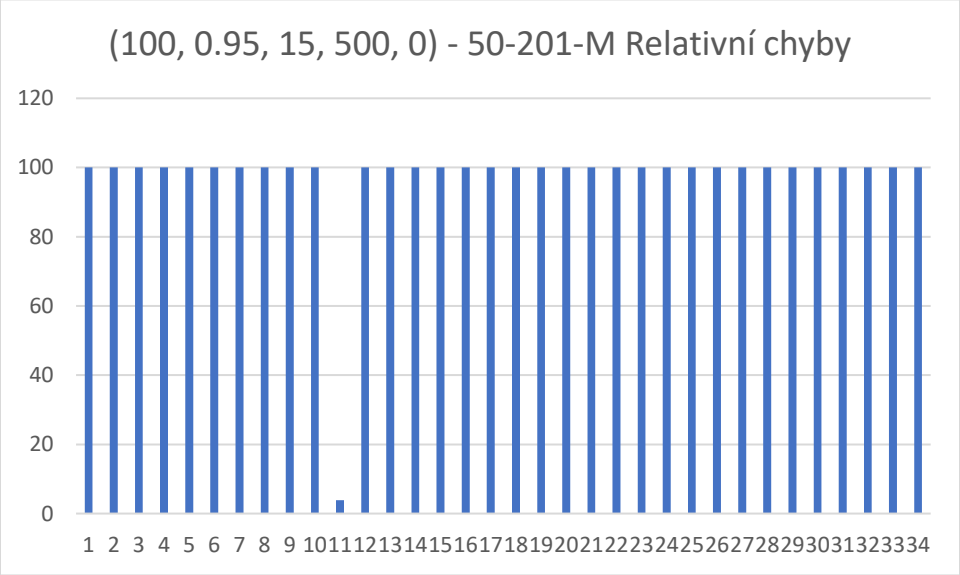


Průměrná chyba spadla na 1.12%, většina řešení dává exaktní výsledek a počet nenalezených řešení se také snížil. Zbývá tedy upravit parametry tak, aby jsme vždy našli alespoň nějaké řešení. Jelikož je algoritmus randomizovaný, v pár případech ze 1000 se může stát, že se něco pokazí a algoritmus nenajde řešení. Pokud zvýšíme počet vnitřních cyklů, algoritmus bude v první fázi déle hledat nějaké řešení a více zapojí třetí heuristiku (cache). Nastavíme tedy parametry na (100, 0.95, 15, 500, 0).

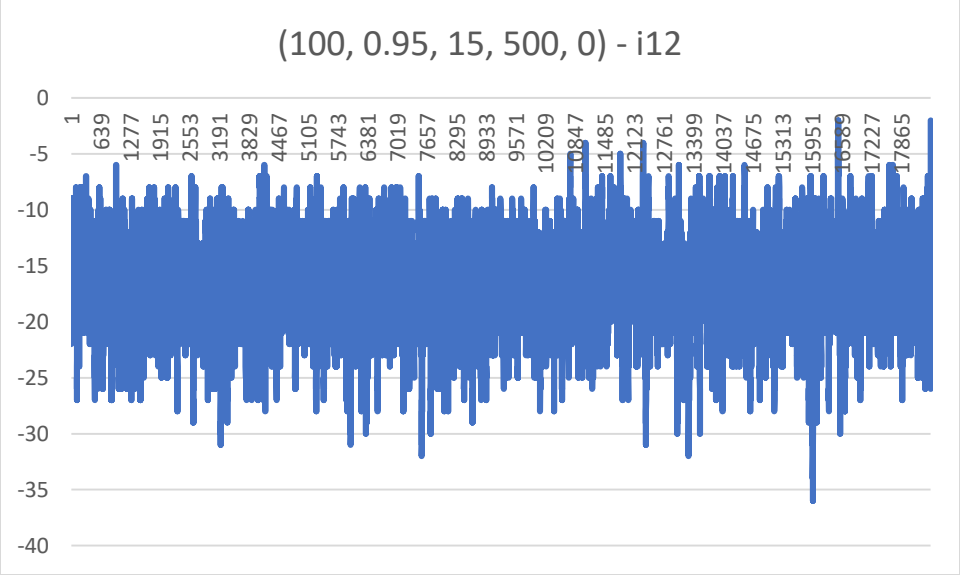


Průměrná chyba klesla na 1.07%, což není velké zlepšení, ovšem výrazně se zmenšil počet instancí s malou chybou. Průměrnou chybu nám tedy už kazí pouze pár specifických instancí, pro které nemůžeme nalézt řešení a je potřeba se na ně podívat po jedné. Jsou to instance 33, 77, 88, 340, 341, 780, 890, 909 a 910. Když se podíváme, co jsou to za instance zjistíme, že se do tohoto souboru zamíchaly instance typu 50-210. Tyto instance mají více proměnných a klauzulí a tedy vyžadují jiné nastavení parametrů nebo dynamicky volené parametry, které jsou momentálně vypnuté. Můžeme tedy udělat závěr, že instance typu WUF-M-20-78 umíme řešit s velmi malou chybou s parametry (100, 0.95, 15, 500, 0) a rovnou se můžeme přesunout na další sadu WUF-M-50-201, kde lépe zjistíme, proč tyto parametry nefungují. Jakmile se nám podaří vyřešit WUF-M-50-201, můžeme zpětně zkontrolovat, jestli vyřešíme WUF-M-20-78 spolu se smíšenými instancemi WUF-M-50-201. Pro instance 50-201 už nemáme tolik exaktních řešení. Zatím budeme řešit pouze ty, pro které ho máme.

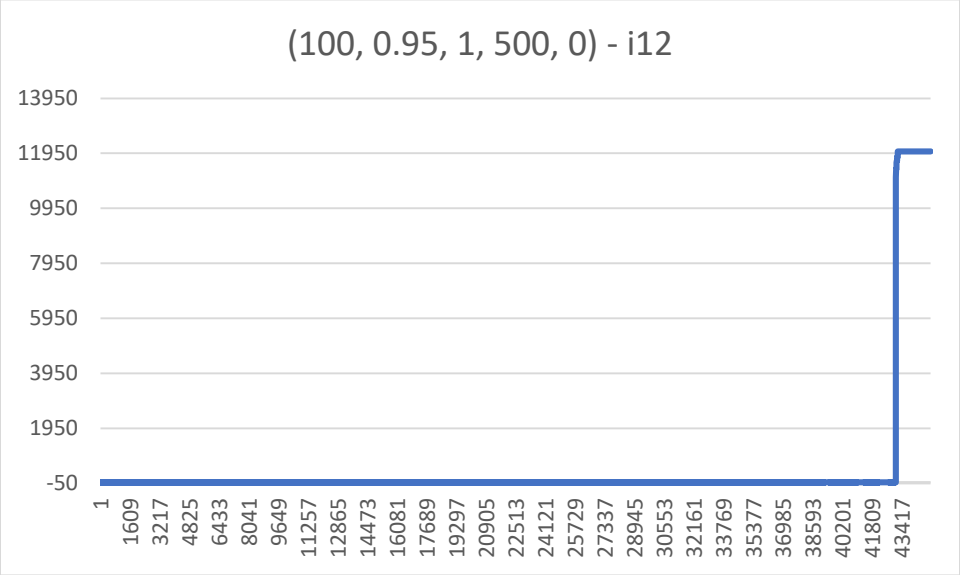
Vylepšování parametrů na těžších sadách



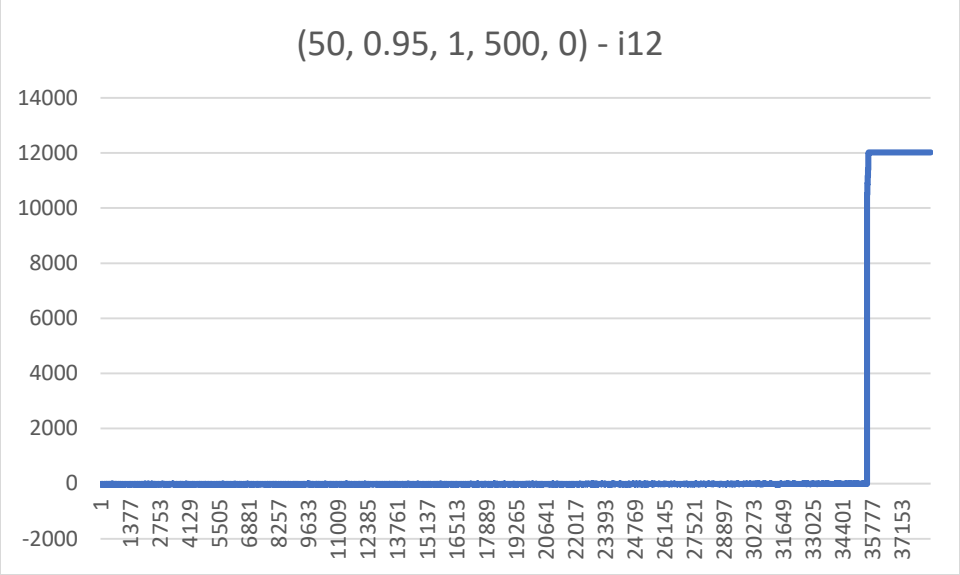
Pro 33 z 34 případů s těmito parametry nemůžeme najít řešení. Je tedy potřeba podívat se přesně co se děje na jedné instanci. Zvolíme první instanci (id 12) s exaktním řešením 12019 a podíváme se z blízka.



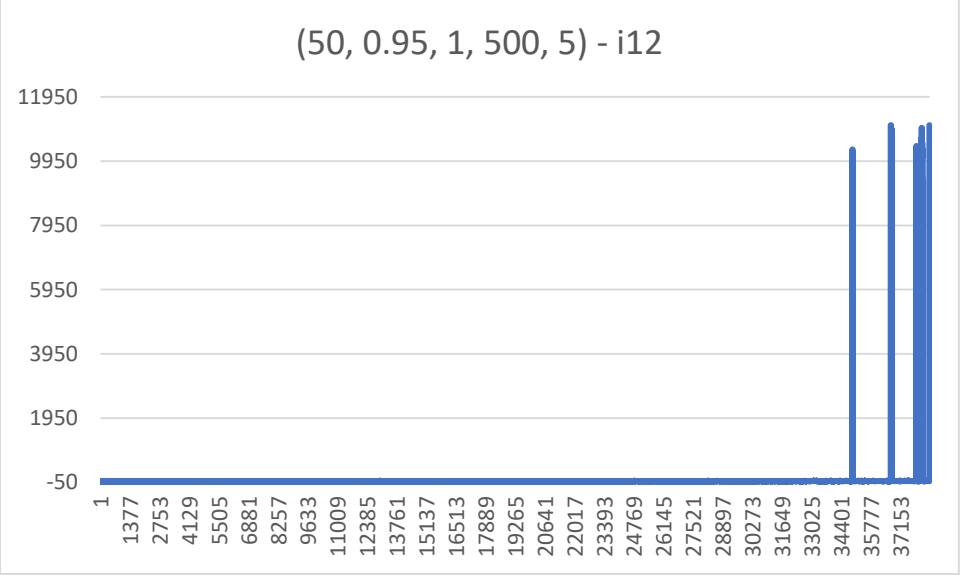
SA bloudí stavovým prostorem a není schopné najít řešení, chybí jakákoliv konvergence či zlepšení. Jelikož chceme konvergenci, posuneme koncovou teplotu z 15 na 1. Chceme dosáhnout konvergence k nějakému výsledku a ta se děje na konci, což je u 1. Nové parametry tedy budou (100, 0.95, 1, 500, 0).



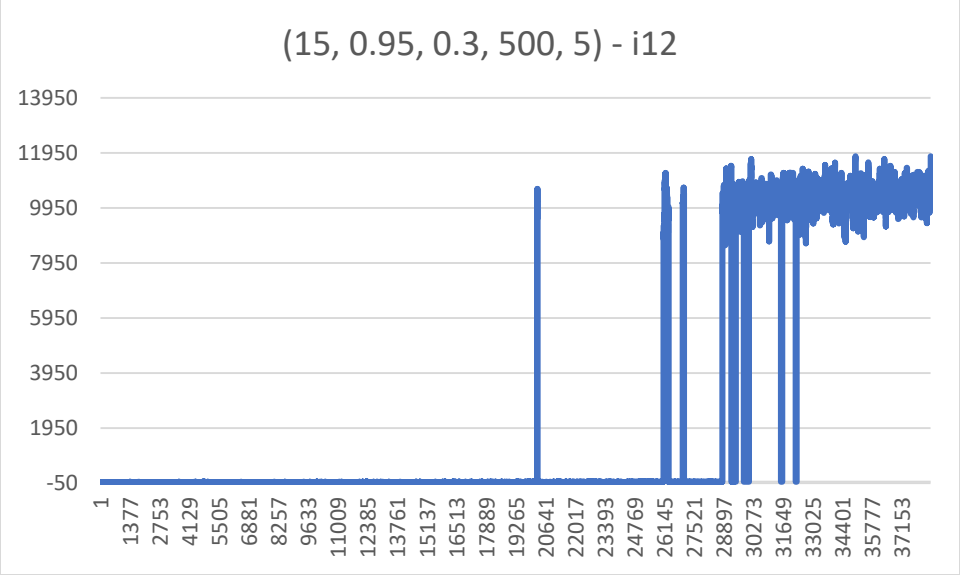
SA dosáhlo exaktního řešení, ale až těsně na konci a 95% výpočtu strávilo stejným blouděním. Řešením je očividně snížení počáteční teploty, čímž utneme přebytečné bloudění. SA potom velmi rychle konverguje a nesnaží se hledat zhoršení. Tímto problémem se budeme zabývat, jakmile vyřešíme zbytečné bloudění. Nové parametry budou (50, 0.95, 1, 500, 0).



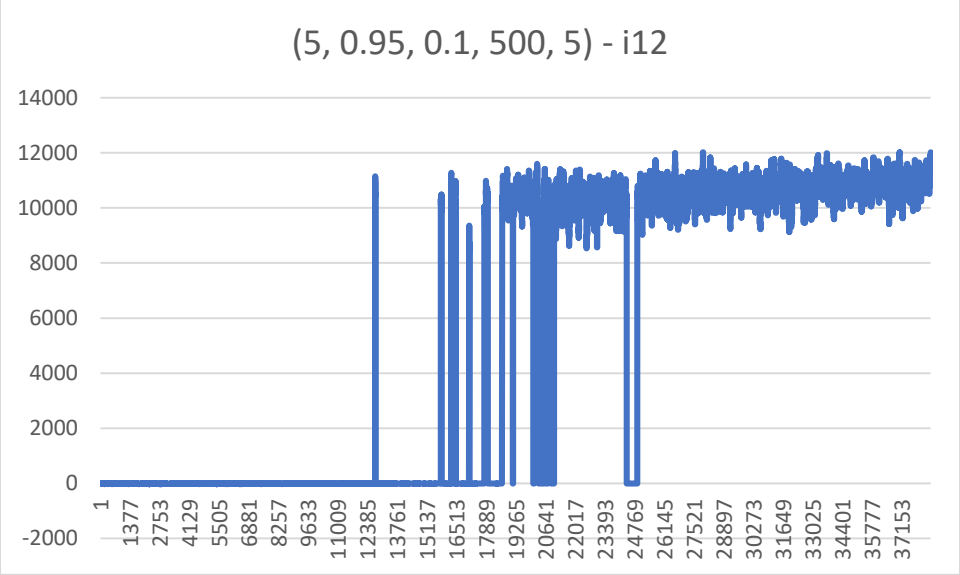
Povedlo se zmenšit bloudění o velmi málo – cca 10%. Kdybychom ovšem šli s teplotou pod 50, rozbije se nám pravděpodobně lehčí sada WUF-M-20-78. Řešením bude zapojení heuristiky dynamické teploty, která nám umožní nastavovat teploty jakožto dynamický rozsah místo nějakých absolutních hodnot. Heuristika byla doteď vypnutá, protože nebyla potřeba. Začneme tedy s náhodným parametrem 5, tedy součet vah / 5. Parametry budou (50, 0.95, 1, 500, 5).



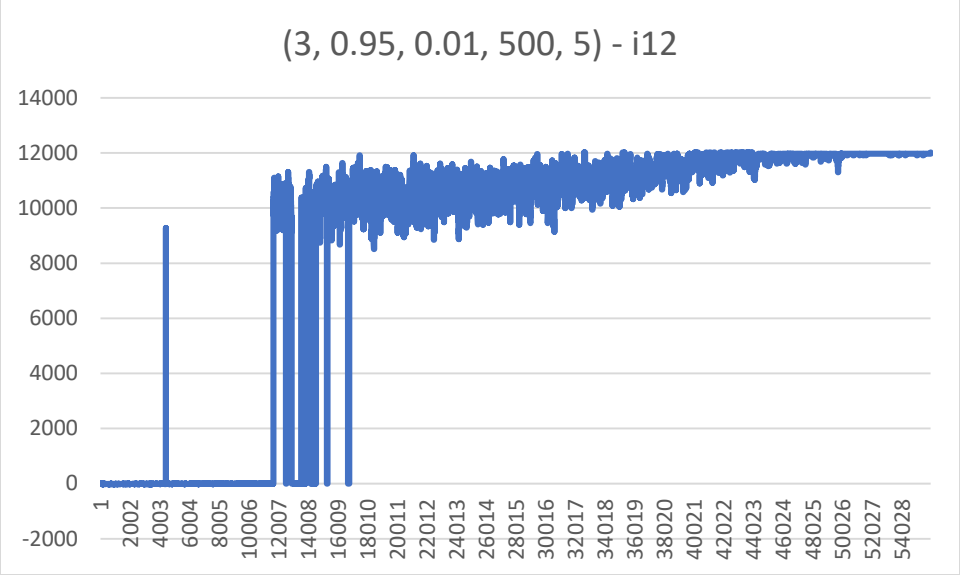
Na první pohled vypadá, že se parametry rozbily, protože nedostáváme optimální řešení, nekonvergujeme nikam a bloudíme prostorem. Je potřeba rozepsat, co heuristika udělala. Heuristika vezme R = součet vah / 5. Poté, když testujeme, jestli přijmout horší řešení, absolutní chybu (rozdíl vah) vydělíme R což ji udělá relativní k R, a tedy i ke koeficientu K který je nastavený na 5. Když bude K malé, dynamické teploty jsou posunuty více k 0, protože absolutní chybu dělíme větším číslem. Stačí tedy posunout koeficienty více k 0. Počáteční a koncovou teplotu tedy posuneme (vezmeme cca 1/3) a podíváme se, co se stane. Parametry (15, 0.95, 0.3, 500, 5).



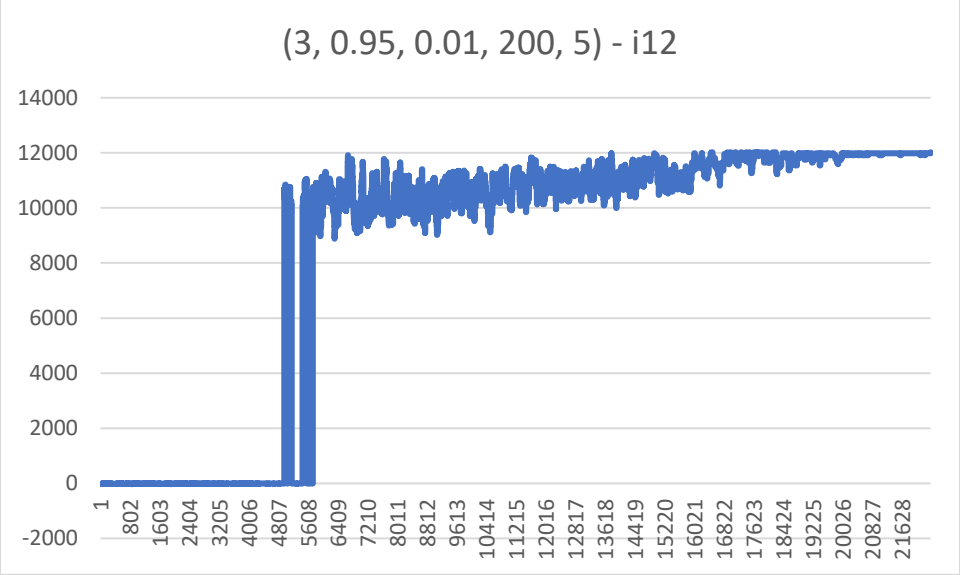
Můžeme vidět značné zlepšení. Sice nekonvergujeme k nějakému řešení, ale SA najde splnitelné řešení a začne prohledávat jeho okolí. Zkusíme posunout ještě na parametry (5, 0.95, 0.1, 500, 5).



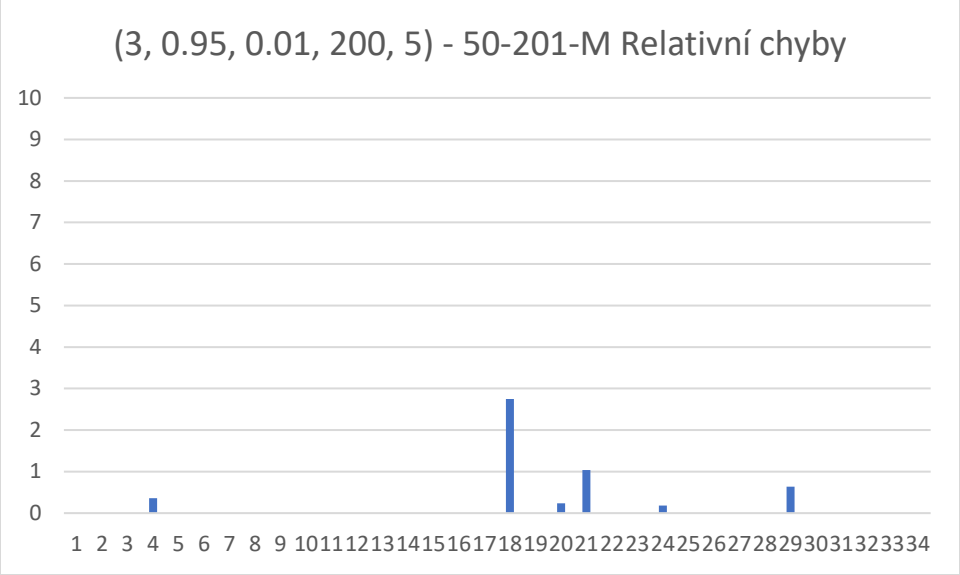
Opět zlepšení, můžeme posunout ještě na (3, 0.95, 0.01, 500, 5).



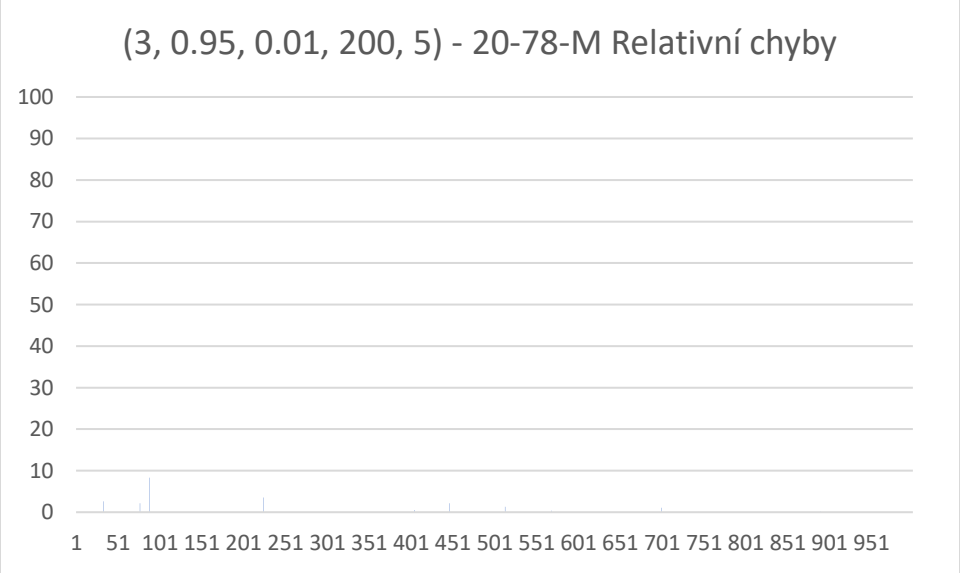
SA nyní konverguje k exaktnímu řešení a pracuje, jak bychom chtěli. Používá ovšem 54000 cyklů. Hodnota vnitřního cyklu je možná příliš vysoká, zkusíme ji značně snížit. Parametry (3, 0.95, 0.01, 200, 5).



Počet cyklů jsme snížili na méně jak polovinu, exaktní řešení je zachováno a SA k němu konverguje. Jediné, co se oproti vnitřnímu cyklu 500 změnilo je, že SA nezkouší dlouho hledat se stejnou teplotou. Tyto parametry se jeví dostatečné pro řešení instance s id 12. Zkusíme tyto parametry přes všechny instance WUF-M-50-201.



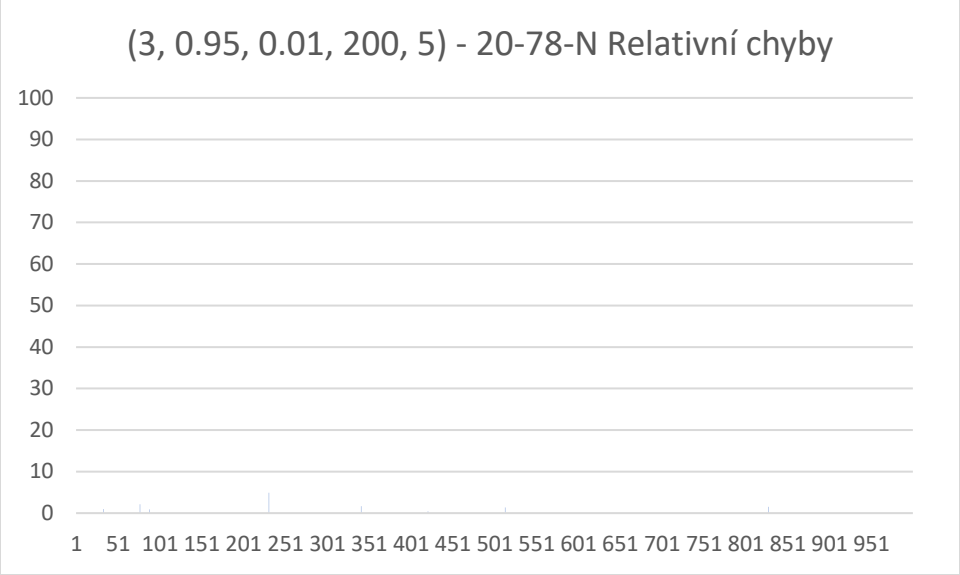
Průměrná chyba je 0.15%, maximální chyba je 2.75%, pouze 6 instancí z 34 nemá exaktní řešení. Můžeme udělat závěr, že pro WUF-M-50-201 jsou parametry (3, 0.95, 0.01, 200, 5) dostačující. I když to vypadá, že oproti WUF-M-20-78 se parametry velmi změnili, není tomu tak, pouze jsme zapojili heuristiku a zvýšili celkový počet cyklů programu, protože je to těžší problém. Parametry (3, 0.95, 0.01, 200, 5) by měli stejně dobře řešit i WUF-M-20-78. To teď musíme ověřit.



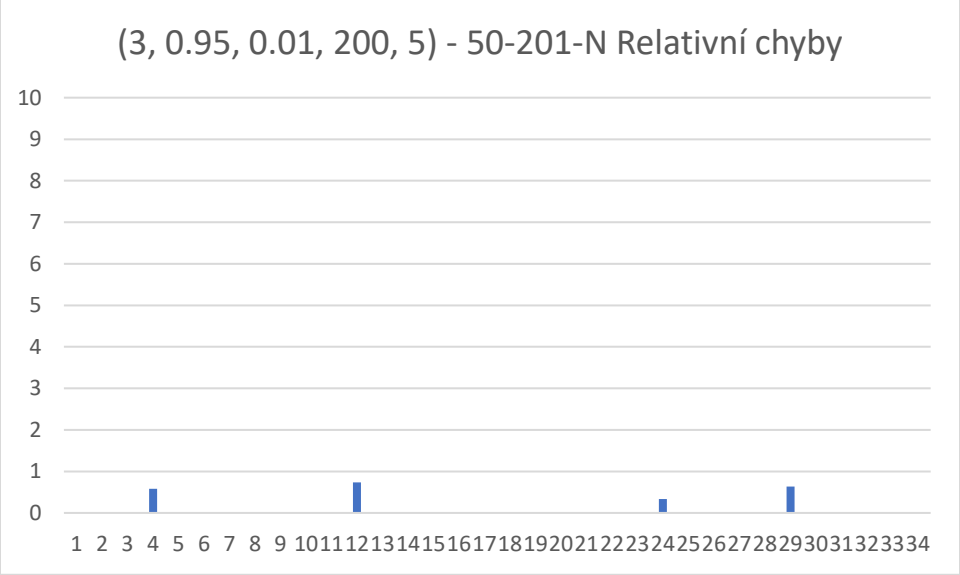
Průměrná chyba je 0.02%, maximální 8.28% (instance která ovšem patří do 50-201 a je zde zamíchaná). Parametry (3, 0.95, 0.01, 200, 5) tedy velmi dobře řeší obě dvě sady problému. Můžeme tedy jít dál a začít testovat na různě modifikovaných sadách.

Vylepšování parametrů na těžších modifikovaných sadách

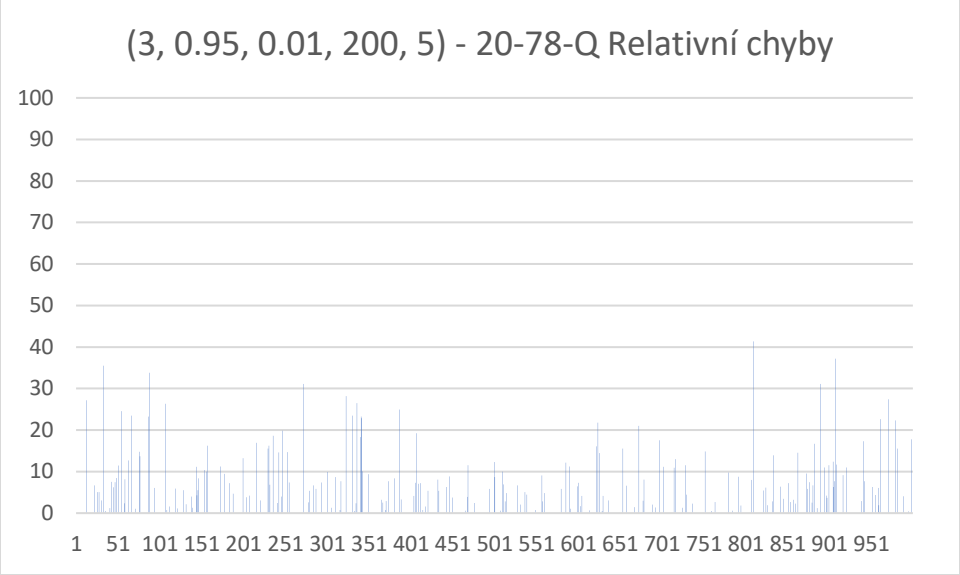
Začneme se sadou WUF-N, která je velmi podobná sadě WUF-M, akorát má různě velké váhy. Tohle by naše parametry nemělo rozbít, protože jsme již zapojili heuristiku dynamických teplot. Vyzkoušíme tedy relativní chybu parametrů (3, 0.95, 0.01, 200, 5) na WUF-N-20-78.



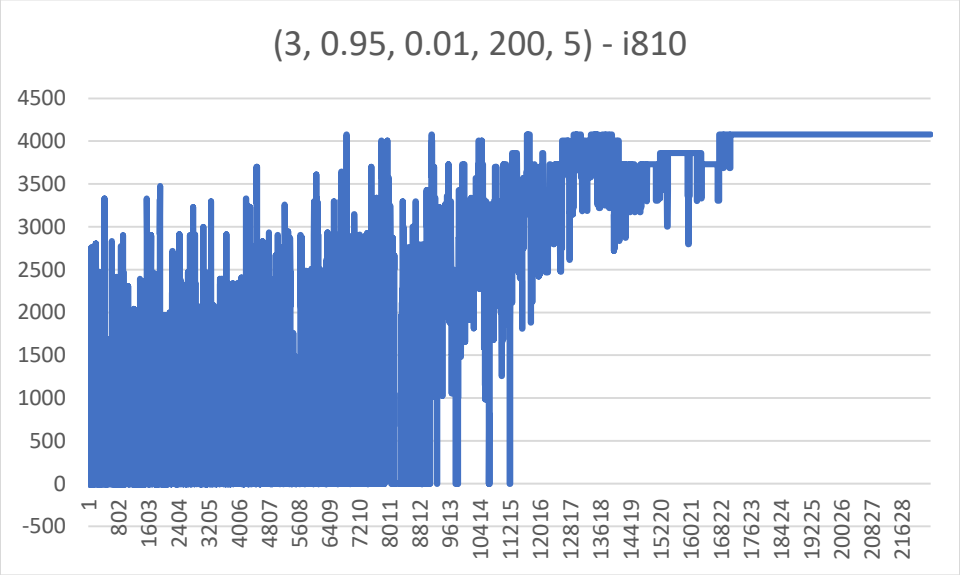
Průměrná chyba 0.014%, maximální 4.93%. Parametry úspěšně řeší tuto sadu, jak se dalo předpokládat. Vyzkoušíme je tedy na WUF-50-201-N, která by měla být opět podobná sadě M díky dynamické heuristice.



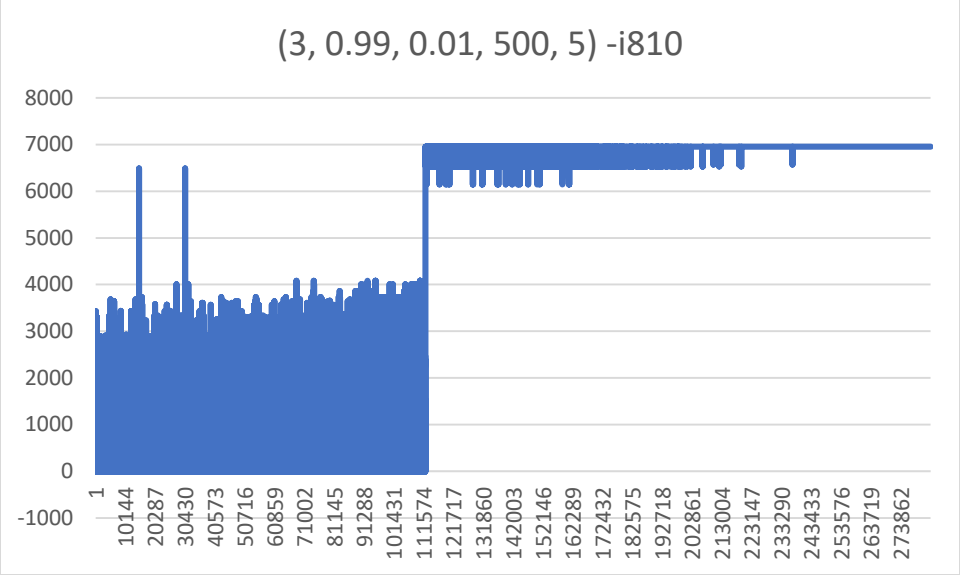
Průměrná chyba 0.067%, maximální 0.74%. Můžeme tedy říct, že parametry (3, 0.95, 0.01, 200, 5) obstojně řeší sady M i N pro velikosti 20-78 a 50-201. Přesuneme se na sady Q a R.



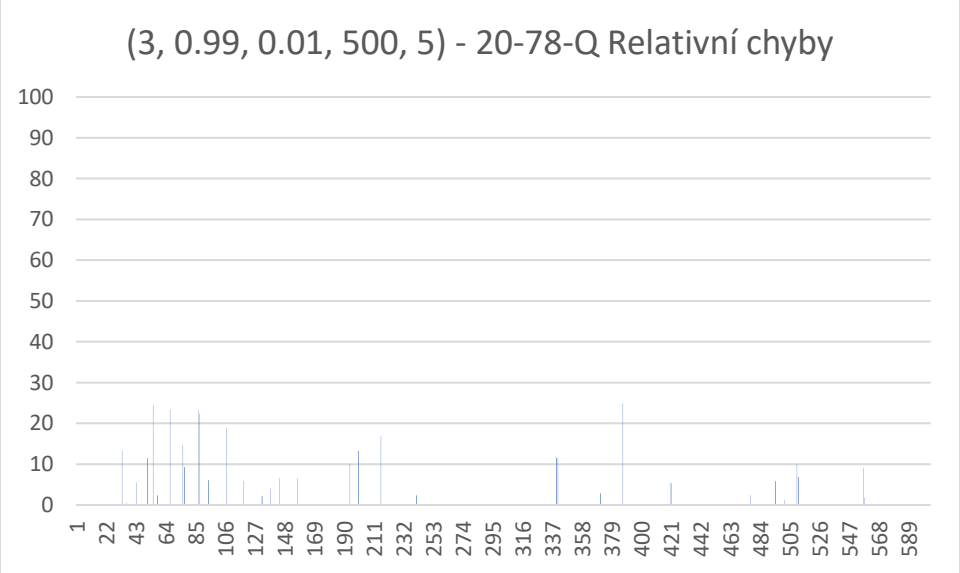
Průměrná chyba 1.86%, maximální chyba 41.38%. Průměrná chyba je ještě dobrá, ale v sadě se objevuje poměrně dost instancí, které mají velkou chybu. Podíváme se tedy zblízka na úlohu 810, která má optimální řešení 6958 a chybu 41.38%.



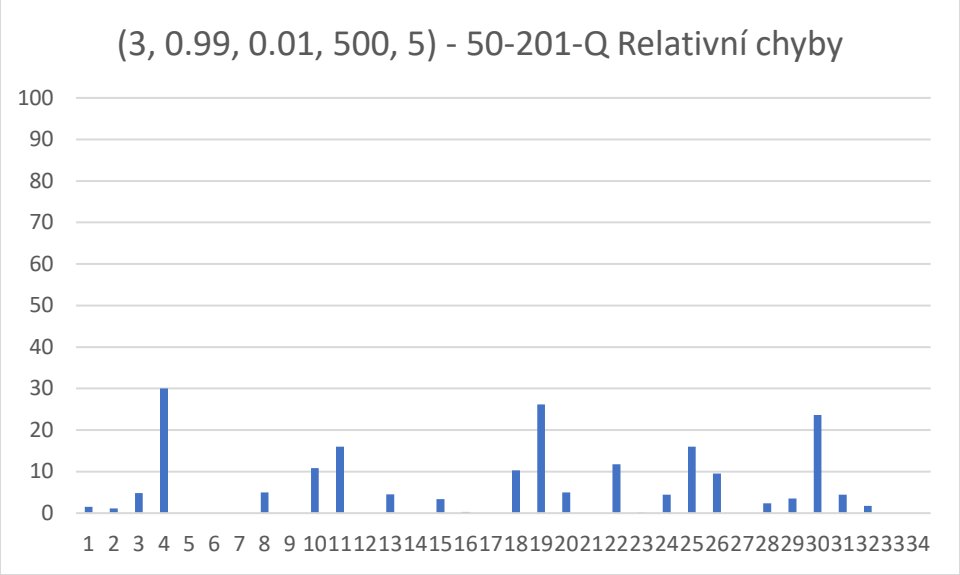
Tento graf ukazuje případ, kdy jsme z algoritmu dostali 41% chybu. Při vícenásobném spuštění bylo exaktní řešení nalezeno cca v 60% případech, v 10% případů bylo nalezeno řešení velmi blízké optimálnímu a ve zbylých 30% bylo nalezeno tohle s 40% chybou. Pokud heuristika nachází řešení, ale jen někdy a z grafu vidíme, že rozsah teplot je nastaven dobře, tedy že na začátku prohledáváme a na konci konvergujeme, chybí nám dostatek cyklů. Přidáme tedy parametrům více počítání a nastavíme je na (3, 0.99, 0.01, 500, 5).



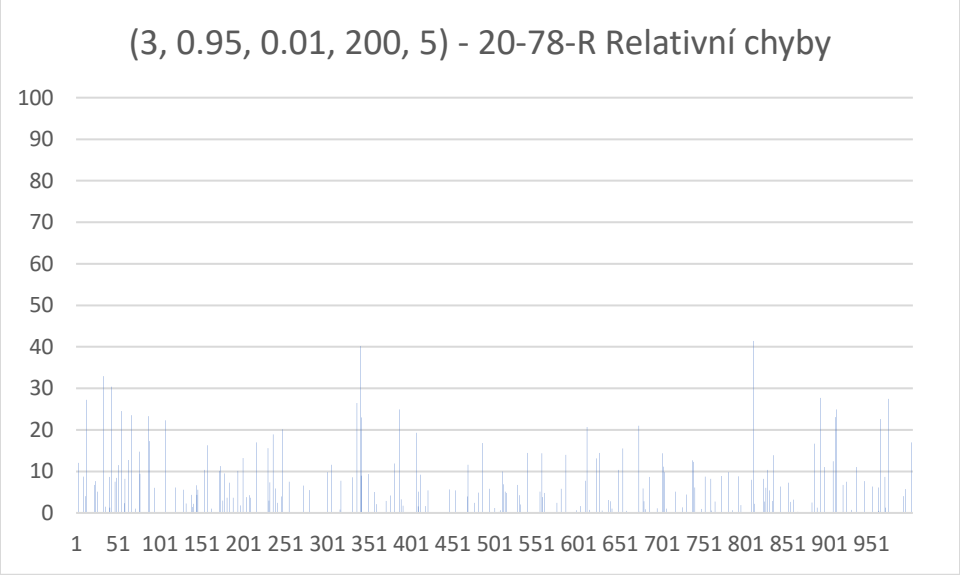
Při vícenásobném spouštění jsme dostali v 80% exaktní řešení (6958) a v 20% řešení velmi blízké exaktnímu (6486). Řešení s 40% chybou jsme už nedostali vůbec. Z tohoto a předchozího grafu lze usoudit, že tato úloha má lokální minimum s 40% chybou, do kterého se lze velmi jednoduše dostat a poté abychom dostali řešení blízké exaktnímu musíme jít přes velmi špatné řešení. Vyzkoušíme parametry (3, 0.99, 0.01, 500, 5) na prvních 600 instancí.



Průměrná chyba 0.63%, maximální chyba 24.9%. Podařilo se nám pomocí výpočetní síly zlepšit výsledek, ale algoritmus už běží poměrně dlouho. Chyba 0.63% je stále dobrá, ale je potřeba dát si pozor na instance kde chyba může být větší. V případě řešení jedné takové instance po několikanásobném spuštění získáme exaktní řešení. Případně by šlo pro tyto specifické instance zvýšit výpočetní výkon nebo upravit parametry na míru. Dále se podíváme na verzi 50-201.

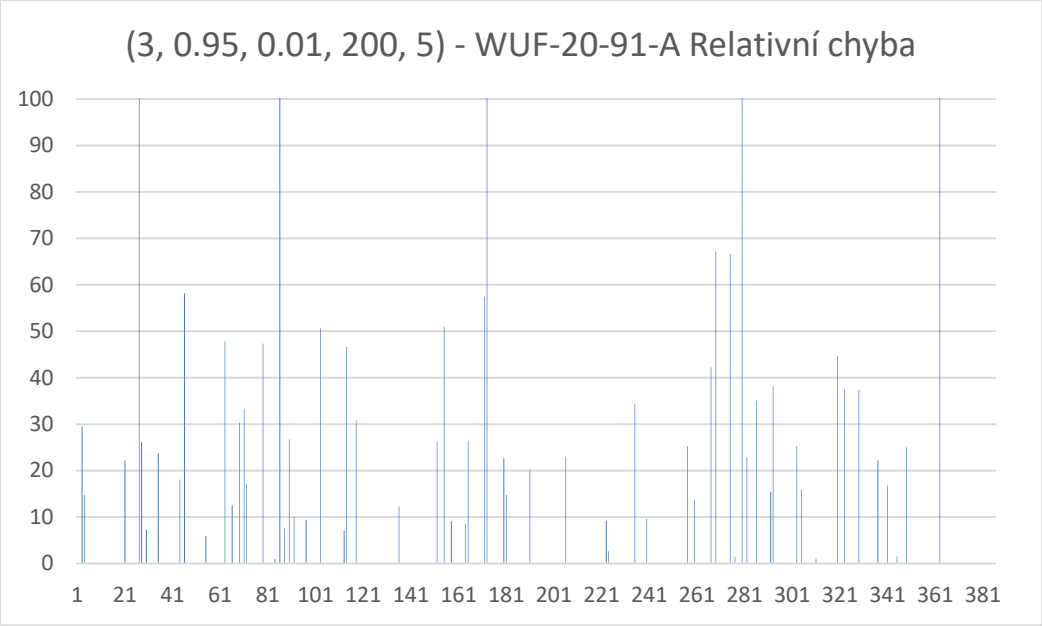


Průměrná chyba 5.78%, maximální chyba 30.07%. Pro tyto instance má algoritmus poměrně vysokou průměrnou chybu. Přejdeme na sady typu R a použijeme opět parametry které jsme odvodili z instancí N a M (3, 0.95, 0.01, 200, 5).

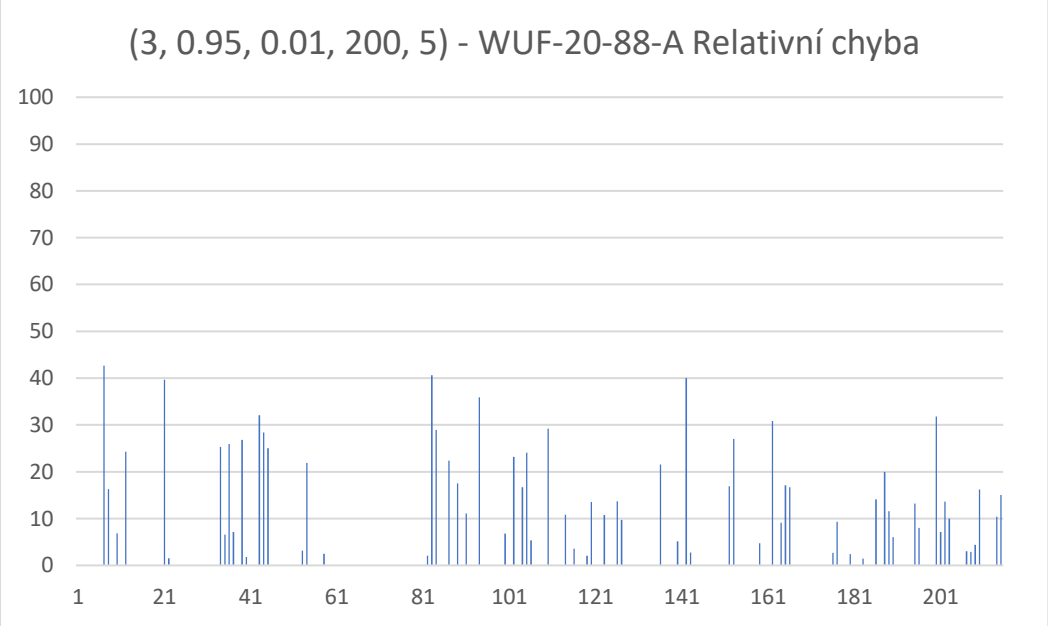


Průměrná chyba 1.65%, maximální 41.39%. SA se chová velmi podobně, jako v sadě Q. Sada R a sada Q se liší pouze velikostí vah. Jelikož používáme dynamickou teplotu tak bychom neměli dostat odlišné hodnoty. Lze tedy předpokládat, že při zvýšení výpočetní síly dostaneme opět zlepšení. Podíváme se tedy spíše na sady A, které jsou zlomyslné a mají často málo řešení díky poměru literálů a klauzulí. Začneme opět s našimi parametry (3, 0.95, 0.01, 200, 5) a sadou WUF-20-91-A a WUF-20-88-A.

Vylepšování parametrů na těžkých sadách

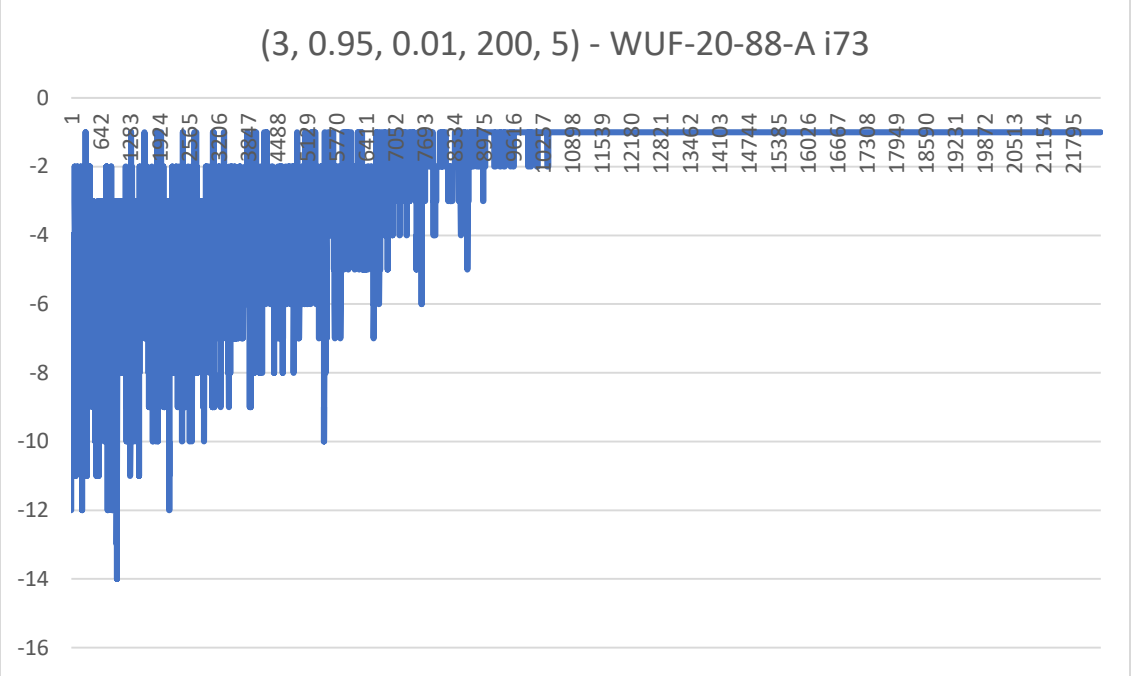


Průměrná chyba 5.11%, maximální chyba 100% (řešení nenalezeno).

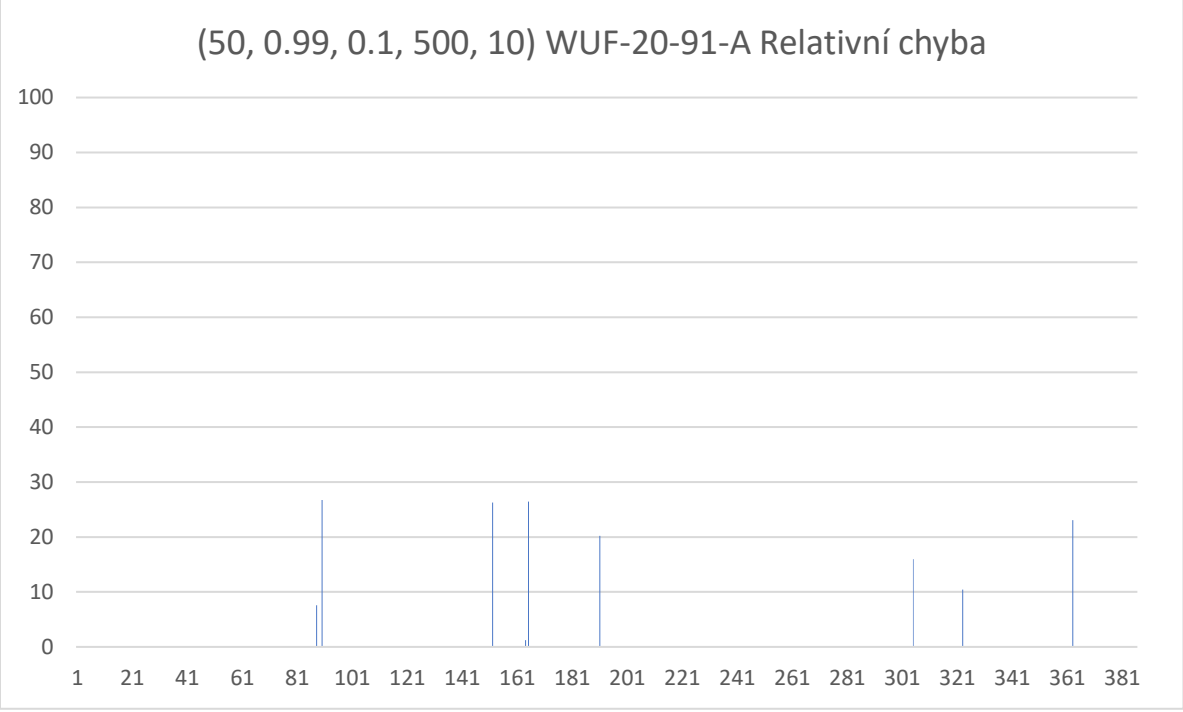


Průměrná chyba 4.93%, maximální chyba 42.67%.

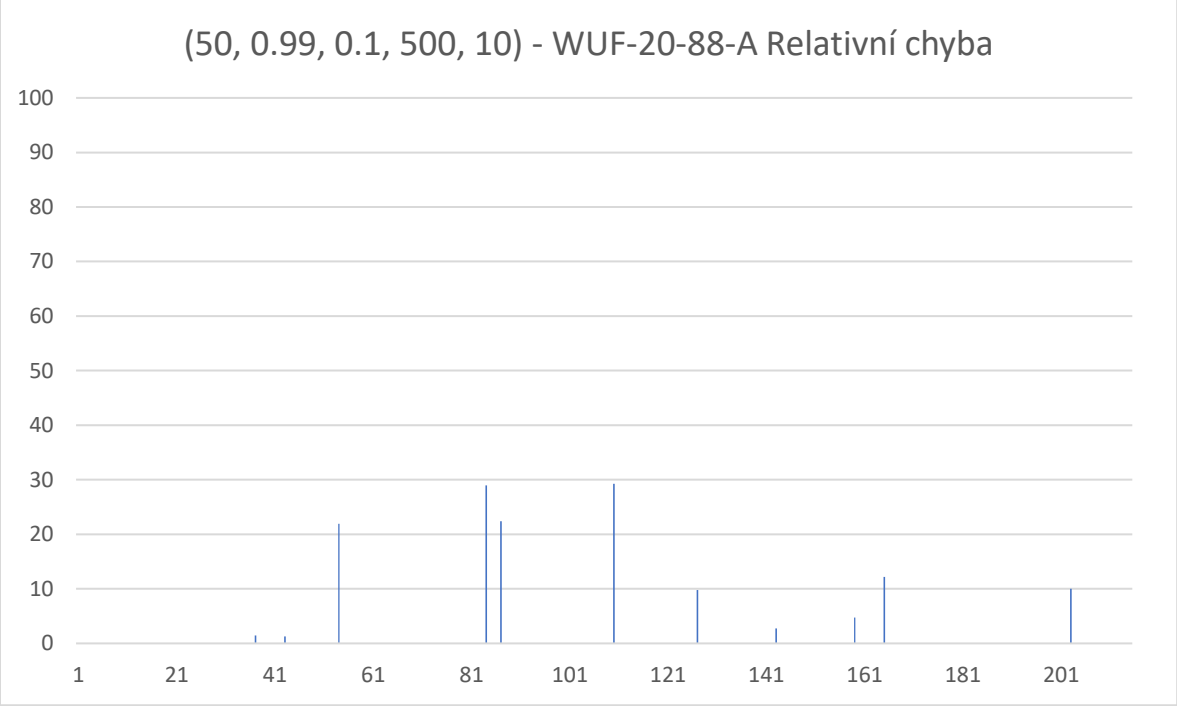
Zákeřné instance se řeší těžko, heuristika dělá větší chyby a musíme se podívat na jednotlivé případy z blízka. Vezmeme instanci s id 73 ze setu 20-91, které SA nenašlo řešení (optimální je 129).



Rozsah teplot vypadá dobře, chybí akorát nalezení splnitelného ohodnocení. Při vícenásobném spuštění zjistíme, že SA nenajde ohodnocení pouze 1 z 10 a ve zbylých 9 případech dokonce nalezne optimální řešení. Sada A obsahuje hodně malé váhy v rozsahu stovek. Nastavení parametru K pro heuristiku dynamických vah nemusí sedět s jinými nastaveními, protože ve všech ostatních sadách šly váhy od tisíců do desetitisíců, kdežto tady jdou níže. Zkusíme tedy zvýšit koeficient K na 10 a postupně upravíme i teploty, aby fungoval posun. Na konec lehce zvýšíme výpočetní sílu, protože ji instance vyžadují. Tohle postupné nastavování v tomto textu přeskočíme a dostaneme se na parametry (50, 0.99, 0.1, 500, 10). Se zvýšením K z 5 na 10 jsme posunuli rozsah teplot od 50 do 0.1. Výpočetní sílu by měl zajistit jak koeficient ochlazování, tak zvýšený počet cyklů.



Průměrná chyba 0.41%, maximální chyba 26.72%. Zvýšení výpočetní síly a zvýšení koeficientu funguje dobře. Vyzkoušíme ještě na WUF-20-88-A.



Relativní chyba 0.67%, maximální chyba 29.23%. Parametry fungují dobře i na těchto instancích. Závěr tedy je, že parametry (50, 0.99, 0.1, 500, 10) fungují dobře pro sadu A.

Finální parametry – závěr

V následující tabulce můžeme vidět nejlepší volby parametrů pro jednotlivé sady. Zároveň platí, že parametry pro sadu A lze použít i pro sadu R, Q, M a N, parametry pro sadu Q a R lze použít pro sady M a N. Parametry (50, 0.99, 0.1, 500, 10) jsou tedy univerzální, ale jejich použití na sady lehkých problémů má za příčinu zbytečný výpočetní overkill, který není potřeba. Pokud ovšem chceme univerzální řešení, je to nutné zlo.

Sada	Nejlepší parametry	Průměrná chyba (ta horší)
N	(3, 0.95, 0.01, 200, 5)	0.15%
M	(3, 0.95, 0.01, 200, 5)	0.067%
Q	(3, 0.99, 0.01, 500, 5)	0.63% N20, 5.78% N50
R	(3, 0.99, 0.01, 500, 5)	1.65%
A	(50, 0.99, 0.1, 500, 10)	0.67%

Závěr

Implementace simulovaného ochlazování je schopna řešit jakékoliv instance 3-SAT (váženého) problému s počtem literálů 20 s velmi malou průměrnou chybou. Pro instance s počtem literálů 50 je schopna řešit s malou chybou instance sad N, M a R. U instancí sady Q pro N50 je potřeba zvýšit délku výpočtu a nastavovat parametry více specificky dané instanci. Při zvolení univerzálních parametrů (50, 0.99, 0.1, 500, 10) je schopen program řešit s velmi malou chybou velkou škálu instancí s počtem literálů 50 či méně. Pro více literálů nemáme k dispozici exaktní řešení a šlo by tedy pouze řešit splnitelnost (najít ohodnocení, pro kterou je formule pravdivá). Implementace je poměrně robustní vůči rozdílům ve velikostech vah a také zvládá těžké instance (sada A), které mají horší poměr počet literálů : počet klauzulí. Tento rozsah je možný především díky dvěma heuristikám – hledání lepších sousedů pomocí cache a počítání relativní chyby vah. Program je psaný v C++17 a pro dané parametry se 1 instance problému počítá v řádu sekund.