FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Heuristické přístupy při řešení NP-těžkých úloh
Student:	Bc. Tomáš Petříček
Vedoucí:	Ing. Michal Šoch, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Seznamte se s problematikou heuristických přístupů při řešení NP-těžkých úloh. Dále se seznamte s problematikou návrhu automatických obchodních strategií burzovních derivátů. Získané znalosti použijte pro:

- návrh aplikace, která pro zadanou automatickou obchodní strategii najde pomoci heuristických aplikací v rozumně krátkém čase nastavení vhodných vstupních parametru, aby strategie byla zisková či oznámí, že strategie není vhodná.
- Naimplementujte a porovnejte alespoň 3 různé heuristické přístupy.
- Aplikace musí generovat detailní logy pro koncové uživatele.
- Aplikaci řádně otestujte.

V diplomové práci se také věnujte diskuzi nad výběrem vhodné technologie pro aplikaci tohoto typu.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 8. září 2022 v Praze.

Diplomová práce

HEURISTICKÉ PŘÍSTUPY PŘI ŘEŠENÍ NP-TĚŽKÝCH ÚLOH

Bc. Tomáš Petříček

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Michal Šoch, Ph.D.
2. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Zvláště bych chtěl poděkovat svému vedoucímu za naslouchání během konzultací, vedení práce a schopnosti mě přimět se soustředit na to podstatné. Rád bych také poděkoval Petru Tešnarovi za poskytnutí obchodní strategie a uvedení do světa algoritmického obchodování.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 2. května 2023 .....

Abstrakt

Cílem práce je nalézt téměř optimální konfiguraci pro algoritmickou obchodní strategii za pomocí heuristik v rozumně krátkém čase. Čtenář je nejprve seznámen se třídami složitosti a heuristickými metodami, včetně těch použitých v praktické části: simulované ochlazování, genetický algoritmus a tabu prohledávání. Následuje popis algoritmického obchodování a fází, které lze automatizovat.

Znalosti z obou těchto oborů jsou dány dohromady v sekci zaměřené na zpětné testování. Cílem zpětného testování je otestovat obchodní strategii na historických tržních datech, aby bylo možné provést úpravy a najít vhodnou konfiguraci předtím, něž je použita pro skutečné testování. Nalezení vhodné konfigurace je NP-těžký problém, nicméně lze použít heuristiky pro jeho nalezení v rozumném čase.

Za účelem dosažení větší flexibility a rychlosti byl vytvořen vlastní framework napsaný v C++20. Historická simulace byla inspirována platformou TradingView a proces obchodování burzou Binance. Byly použity moderní prvky C++, jako jsou koncepty a korutiny. Důraz byl kladen na efektivní využití funkcí jazyka a prostředků počítače.

Kromě heuristiky byl implementován také algoritmus hrubé síly, který byl paralelizován pomocí OpenMP k dosažení vyšší rychlosti. Heuristiky byly implementovány tak, aby byly flexibilní a snadno použitelné, proto byly použity návrhové vzory jako pozorovatel a strategie. Průběh a výsledky shromážděné během optimalizace byly uloženy do souborů ve formátu JSON a CSV, aby mohly být později zpracovány pomocí vědeckých balíčků jazyka Python.

Framework byl unit testován před provedením experimentálního vyhodnocení. Pro experimenty byly použity minutové svíčky 10 různých kryptoměn. Nejprve byl vybrán vyhledávací prostor s rozumnými konfiguracemi a velikostí. Nejlepší konfigurace byly nalezeny pomocí algoritmu hrubé síly. Následně byly provedeny white-box a black-box testy pomocí heuristických přístupů a výsledky porovnány s optimálními. Heuristické přístupy jsou schopny najít téměř optimální konfigurace během několika minut namísto hodin nebo dokonce dnů pomocí i nejoptimalizovanějšího algoritmu hrubé síly.

Klíčová slova zpětné testování, optimalizace, metaheuristiky, algoritmické obchodování

Abstract

The aim of the thesis is to find a near optimal configuration for an algorithmic trading strategy using heuristics in reasonably short time. The reader is firstly introduced to complexity classes and heuristic methods, including those used in the practical part: simulated annealing, genetic algorithm and tabu search. It is followed by the description of algorithmic trading and the stages that can be automated.

Knowledge from both of these fields is combined in the backtesting section. The goal of backtesting is to test a trading strategy against historical data to make adjustments and find appropriate configuration before deciding whether or not to use it in real-time trading. Finding these configurations is an NP-hard problem, and heuristics can be used to find the optimal ones in a reasonable time.

In order to achieve greater flexibility and speed, a custom framework written in C++20 was created. The historical simulation was inspired by the TradingView platform and the trading process by the Binance exchange. Modern C++ features such as concepts and coroutines were used. The emphasis was placed on the effective use of the language features and computer resources.

In addition to heuristics, a brute force algorithm was also implemented. It was parallelized using OpenMP to achieve higher speed. Heuristics were implemented to be flexible and easy to use, thus design patterns such as Observer and Strategy were used. The progress and results collected during the optimization were saved in JSON and CSV files, so that they can later be examined using scientific Python packages.

The framework was unit tested before the experimental evolution was performed. Years of one-minute candlestick data of 10 different cryptocurrencies were used for the evaluation. First, a search space with reasonable configurations and size was selected. The best configurations were found using a brute force algorithm. White-box tests and black-box tests were then performed using heuristic approaches, and the results were compared with the optimal ones. Overall, heuristic approaches are able to find near-optimal configurations in minutes instead of hours or even days using even the most optimized brute-force algorithm.

Keywords backtesting, optimization, metaheuristics, algorithmic trading

Seznam zkratek

CSV	Comma-Separated Values
EMA	Exponential Moving Average
JSON	JavaScript Object Notation
PROM	Pessimistic Return On Margin
SMA	Simple Moving Average

Úvod

Obchodovat lze různá aktiva, jako jsou akcie, kryptoměny nebo indexy, nicméně hlavní cíl zůstává stejný, využít změny tržních cen k maximalizaci zisku a zároveň minimalizovat riziko ztráty. Pro dosažení konzistentních zisků systematickým způsobem bylo vyvinuto algoritmické obchodování, které využívá počítačově naprogramované strategie pro obchodní rozhodnutí na základě jasně definovaných podmínek. Tyto strategie nejsou svázány stejnými omezeními jako lidští obchodníci, necítí únavu a nenechají se svést emocemi.

Mohou být také zpětně testovány na historických tržních datech, aby bylo možné vyhodnotit jejich výkonnost a najít slabá místa. Strategie jsou většinou parametrizované a hledání optimálních parametrů je NP-těžký problém. K nalezení nejlepšího nastavení lze použít algoritmus hrubé síly, který je však v praxi většinou nepoužitelný, protože nenajde vhodné nastavení v dostatečně krátkém čase.

Použití heuristických přístupů se v tomto případě jeví jako přirozené, nicméně existuje pouze pár platforem, které je nabízejí. Možným vysvětlením, proč nejsou běžně dostupné je, že existuje velké množství různých algoritmických strategií, které vyžadují odlišně parametrizované vyhledávací prostory, což implementaci značně ztěžuje. Oproti algoritmu hrubé síly jsou obtížněji vysvětlitelné.

Cílem této práce je implementovat vybranou algoritmickou obchodní strategii a pomocí heuristických přístupů spolehlivě a rychle najít její optimální parametry. V rámci toho lze získat lepší porozumění algoritmickému obchodování a více zkušeností s implementací, používáním a laděním heuristik. Ideálně je možné vytvořit dobrý základ pro budoucí práci.

Kapitola 1

Kombinatorická optimalizace

Cílem řešení *optimalizačního problému* je nalézt nejlepší řešení. Formálně lze problém zapsat jako $P = (S, f, \Omega)$, kde P je optimalizační problém, S je vyhledávací prostor, f je objektivní funkce a Ω je sada optimalizačních omezení. Objektivní funkce je také označována jako cenová nebo fitness funkce. Jejím účelem je ohodnotit konfigurace a její hodnota slouží k jejich porovnání [1].

Problémy lze dělit na minimalizační a maximalizační podle toho, zda je cílem maximalizovat nebo minimalizovat hodnotu objektivní funkce. Vyhledávací prostor se skládá ze sady proměnných označovaných jako konfigurační proměnné. Pro získání platných řešení mohou být použita omezení [2, 1].

Kombinatorické optimalizační problémy jsou definovány pomocí diskrétních konfiguračních proměnných. Jelikož je množina přeskupení těchto proměnných konečná, je konečná i množina řešení. Problémy obvykle spadají do kategorie *NP-těžkých* problémů [1].

Obecně lze kombinatorické problémy řešit pomocí *hrubé síly*, která prohledá všechny možné řešení v prohledávacím prostoru [1]. Největší výhodou tohoto přístupu je, že zaručuje nalezení nejlepšího možného řešení, nicméně je v praxi většinou nepoužitelný, jelikož neskončí v dostatečně krátkém čase [3]. Konfigurace mohou být reprezentovány různými datovými strukturami, mezi nejběžnější patří: pole a stromy. Typ kódování závisí na problému [1].

1.1 Třídy složitosti

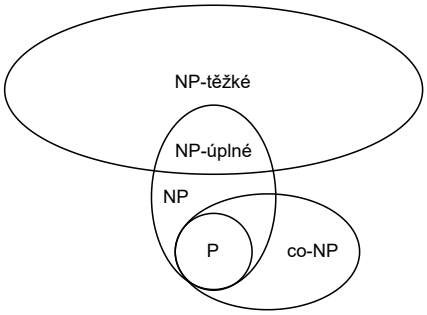
Obtížnost problému lze měřit pomocí *výpočetní složitosti*, která se vztahuje k době, kterou algoritmus potřebuje k získání výsledku v závislosti na velikosti vstupu. Výhodou je, že je nezávislá na hardwaru a umožňuje srovnání algoritmů [1].

Problémy lze podle efektivity jejich řešení rozdělit do tříd. Pokud je doba jejich běhu vázána na polynomiální funkci: $O(n^c)$, kde c je nějaká konstanta a n je velikost vstupu, jsou algoritmy považovány za efektivní. Ne všechny problémy však lze efektivně vyřešit. Pokud problém patří do třídy *NP-těžký* předpokládá se, že neexistuje žádný efektivní algoritmus k jeho řešení [4].

Rozhodovací problémy, jejichž výstupem je booleanovská hodnota, lze zařadit do tří tříd. Do třídy P patří problémy, které lze vyřešit v polynomiálním čase. Pokud lze odpověď *ano* na problém ověřit v polynomiálním čase, patří do třídy *NP*, což znamená nedeterministický polynomiální. Pokud lze odpověď *ne* ověřit v polynomiálním čase, tak patří do třídy *co-NP*, která je komplementární třídě NP.

Každý rozhodovací problém ve třídě P je také v NP a co-NP, jelikož odpověď na problém lze vypočítat v polynomiálním čase, a proto ji také ověřit. Obecně se má za to, že $P \neq NP$. Nicméně to nebylo nikdy dokázáno, a protože jde o jeden z problémů tisíciletí, je za důkaz odměna 1 milion dolarů. Jestli NP a co-NP patří do stejné třídy také není známo [4].

Problém je NP-těžký, pokud by nalezení algoritmu, který jej řeší v polynomiálním čase, znamenalo existenci polynomiálního časového algoritmu pro každý problém v NP. Aby problém patřil mezi NP-těžké problémy, musí být alespoň tak těžký jako každý problém v NP. Pokud je problém NP-těžký a zároveň NP, pak je nazýván NP-úplný. Polynomiální algoritmus pro jeden NP-úplný problém by znamenal polynomiální algoritmus pro všechny NP-úplné problémy [4]. Schéma všech zmíněných tříd složitosti lze vidět na obrázku 1.1.



Obrázek 1.1 Schéma tříd složitosti

1.2 Heuristické metody

Heuristická metoda je postup, který umožňuje nalézt rozumně dobré řešení v relativně krátkém čase. Metody jsou často iterativní, počínaje počátečním řešením, které se během každé iterace snaží vylepšit. Po uplynutí přiměřeně dlouhé doby je proces optimalizace ukončen a nejlepší řešení vráceno [5].

Termín *metaheuristika* byl zaveden v roce 1986. Skládá se ze dvou slov meta znamenající vyšší úroveň abstrakce a heuristiky znamenající umění objevovat pravidla k řešení problému. Jsou to strategie, které využívají vyšší úroveň abstrakce spolu s heuristikou k prohledávání stavového prostoru s cílem nalezení nejlepšího řešení. Jednou z jejich hlavních výhod je, že jsou nezávislé na daném problému. Díky tomu nalezly uplatnění v různých oborech jako je lékařství, aplikovaná matematika, strojírenství nebo ekonomie [5, 1].

Při prohledávání stavového prostoru je klíčové vyvážit fáze průzkumu (exploration) a vytěžování (exploitation). Průzkum slouží jako prostředek pro *diverzifikaci*. Cílem je najít slibné oblasti s vysoce kvalitními řešeními. Jakmile je oblast identifikována, přistoupí se k vytěžování, kdy je hledání v dané oblasti zintenzivněno [1].

Metaheuristiky lze dělit do dvou kategorií na algoritmy založené na jediném řešení nebo na populaci. Metaheuristiky založené na jednom řešení začínají s jedinou kandidátní konfigurací, kterou se snaží postupem času vylepšovat pomocí místního vyhledávání. Nevýhodou těchto technik je, že mohou snadno uvíznout v lokálním optimu. Patří mezi ně tabu prohledávání, simulované ochlazování nebo řízené místní vyhledávání [6].

Na druhé straně metaheuristiky založené na populaci používají během vyhledávání více konfigurací. Díky tomu je zachována diverzita, která ztěžuje uvíznutí v lokálním optimu. Zahrnuje algoritmy, jako jsou genetický algoritmus, optimalizace roje částic a optimalizace mravenčích kolonií [6].

1.2.1 Simulované ochlazování

Simulované ochlazovaní bylo vytvořeno v roce 1982. Heuristika je založena na procesu žíhání pevné látky. Účelem žíhání je donutit materiál se dostat do nízkoenergetického stavu, jako je krystal. Proces spočívá v tom, že je materiál nejprve zahřát, což umožňuje vytvoření mnoha atomových přeskupení, a poté pomalu chlazen. Na začátku existuje mnoho atomových uspořádání, ale postupem času při správném chlazení přetrvají jen ty dobré [7].

Algoritmus začíná s počáteční stavem S_{init} a snaží se jej postupně vylepšit. Aby nedošlo k uvíznutí v lokálním optimu, umožňuje přijmutí horšího stavu. V každé iteraci prohledává okolí aktuálního stavu S_{curr}. Každý krok je ohodnocen změnou energie ΔE. Je-li zlepšující $\Delta E \leq 0$ nahradí S_{curr}, pokud je horší $\Delta E > 0$ může být stále přijat. Pravděpodobnost přijetí horšího stavu je vypočítána pomocí $p_a = e^{\frac{-\Delta E}{T_{curr}}}$, kde $T_{curr} > 0$ je současná teplota. Stav je přijat, pokud platí $r < p_a$, kde r je náhodné číslo z $Unif(0, 1)$. Pravděpodobnost přijetí horšího stavu se snižuje s T_{curr}, nicméně je vždy nenulová [7, 2].

Na konci každé iterace se T_{curr} sníží. Její průběh je řízen pomocí *rozvrhu ochlazování*. Klesání teploty je obecně nejrychlejší na začátku a ke konci se zpomaluje. Zpočátku umožňuje diverzifikaci v podstatě náhodným prohledáváním konfiguračního prostoru. Na závěr vede k intenzifikaci připomínající metodu iterativního zlepšování. Na každé teplotě musí zůstat dostatečně dlouho, aby bylo dosaženo ustáleného stavu zvaném *equilibrium* [7, 2].

Algoritmus je ukončen, pokud platí $T_{curr} \leq T_{min}$. Pro použití heuristiky je třeba implementovat konfiguraci představující nastavení systému, náhodný generátor sousedních konfigurací, objektivní funkci a rozvrh ochlazování [2]. Pseudokód algoritmu lze vidět na 1, kde S_{best} označuje nejlepší stav a S_{cand} kandidátní stav.

Algorithm 1 Simulované ochlazování

```
S_best ← S_curr ← S_init
T_curr ← T_init
while T_curr > T_min do
    for n ← 1 to equilibrium() do                    ▷ Explore neighborhood
        S_cand ← neighbor(S_curr)
        if is_better(S_cand, S_curr) then            ▷ Candidate is better
            S_curr ← S_cand
            if is_better(S_curr, S_best) then
                S_best ← S_curr
            end if
        else                                         ▷ Candidate is worse
            ΔE ← appraise(S_cand, S_curr)
            if rand_prob() < exp(−ΔE / T_curr) then
                S_curr ← S_cand
            end if
        end if
        T_curr ← cool(T_curr)
    end for
end while
return S_best
```

V tabulce 1.1 jsou uvedeny rozvrhy chlazení ze srovnávajícího článku [8], které byly implementovány v praktické části. Všechny generují konečnou sekvenci klesajících teplot. Liší se však časem potřebným pro jejich výpočet a samotný průběhem. Všechny používají T_{init} a hodnotu aktuální iterace $i \geq 0$ k výpočtu T_{curr}. Některé jsou navíc parametrizovány pomocí parametru rozpadu α, jehož rozsah platných hodnot je rovněž uveden.

Tabulka 1.1 Přehled vybraných rozvrhů chlazení

Název	Vzorec
Základní chlazení	$T_{curr} = \frac{T_{init}}{1 + \log(1+i)}$
Exponenciální multiplikativní chlazení	$T_{curr} = T_{init} \cdot \alpha^i, \alpha \in (0,8;\ 0,9)$
Logaritmické multiplikativní chlazení	$T_{curr} = \frac{T_{init}}{1 + \alpha \cdot \log(1+i)}, \alpha > 1$

1.2.2 Genetický algoritmus

Genetický algoritmus je populační metaheuristika, která je inspirována přirozenou evolucí. Napodobuje přežití nejzdatnějších jedinců. Byla navržena v roce 1992. Stavy jsou označovány jako jedinci. Množina jedinců je nazývána populace. Objektivní funkce se označuje jako fitness funkce a slouží k přiřazení zdatnosti jednotlivým jedincům. Algoritmus používá tři biologicky inspirované operátory: *selekci*, *křížení* a *mutaci* [6].

Začíná s náhodně inicializovanou populací P_{init}. Nové generace jsou produkovány iterativně aplikací genetických operátorů. Na začátku každé iterace jsou vybráni pomocí selekčního operátoru rodiče $P_{parents}$. Vybraní jedinci jsou pak spárováni a kříženi. Na výsledné potomky $P_{children}$ je s nějakou pravděpodobností aplikován operátor mutace. Jako poslední dojde k nahrazení současné populace P_{curr}. Proces se opakuje do splnění ukončovacího kritéria [6].

Kódování jedinců je závislé na problému, ale často se provádí pomocí binárního řetězce. Hlavní výhodou je, že operátory křížení a mutace lze snadno a rychle implementovat. Mezi další typy kódování patří permutační kódování, kódování hodnot a stromové kódování [6].

Selekce se používá k určení jedinců, kteří se budou účastnit reprodukce. Ta je spojena s *selekčním tlakem*, který umožňuje algoritmu konvergovat. Určuje pravděpodobnost výběru zdatnějšího jedince. Mezi známé algoritmy selekce patří: *ruletový*, *turnajový* nebo *boltzmannův* výběr [6].

Při použití ruletového výběru jsou nejprve jedinci z dané populace namapováni na kolo rulety. Velikost výřezu je úměrná zdatnosti jedince. Což znamená, že čím je daný jedinec zdatnější, tím větší je pravděpodobnost, že bude vybrán. Při každém náhodném roztočení rulety je vybrán jeden jedinec. Každý jedinec může být vybrán vícekrát [6].

Křížení se používá s cílem vytvořit nové potomky spojením genů dvou nebo více rodičů. Existují dobře známé techniky jako jsou: jednobodové, vícebodové, cyklové a rovnoměrné křížení. Dále může docházet k mutacím, které se vyskytují s určitou nízkou pravděpodobností. Slouží k zavedení náhodných změn. Simulují chyby, ke kterým dochází při kopírování chromozomů [6]. Pseudokód algoritmu lze vidět na 2.

Algorithm 2 Genetický algoritmus

$P_{curr} \leftarrow P_{init}$
while $\neg terminate()$ **do**
  $P_{parents} \leftarrow select(P_{curr})$
  $P_{children} \leftarrow crossover(P_{parents})$
  $P_{children} \leftarrow mutate(P_{children})$
  $P_{curr} \leftarrow replace(P_{parents}, P_{children})$
end while
return P_{curr}

1.2.3 Tabu prohledávání

Tabu prohledávání patří mezi metaheuristiky založené na jednom řešení. K prohledávání stavového prostoru využívá adaptivní paměť, která umožňuje jeho efektivní prohledávání. Prohledávacím prostorem se pohybuje aplikováním změn na aktuální stav S_{curr}. Množina stavů dosažitelných z S_{curr} se nazývá sousedství. Operaci přesunu je používána k dosažení sousedního stavu S_{cand} [9].

Algoritmus začíná hledat z počátečního stavu S_{init}. Během každé iterace je prohledáno sousedství S_{curr}. Pokud je nalezen lepší S_{cand}, je S_{curr} nahrazen. Pokud není nalezen žádný lepší tah, je S_{curr} nahrazena nejlepším nezlepšujícím se sousedem. Algoritmus je ukončen, jakmile je splněno ukončovací kritérium [9].

Dalším mechanismem, který pomáhá algoritmu překročit lokální optima, je *krátkodobá paměť*. Algoritmus si ukládá nedávnou historii tahů, které byly provedeny, a označuje je jako zakázané (tabu). Tahy lze ukládat jako celé konfigurace, tento typ paměti se nazývá *explicitní paměť*. Pokud jsou uloženy pouze vybrané atributy tahu, pak se paměť nazývá *atributivní paměť*. Explicitní paměť obecně spotřebovává méně místa než atributivní paměť. Pokud S_{cand} obsahuje nějaké tabu aktivní atributy, nelze ji přijmout jako S_{curr} [9].

Tabu lhůta (tenure) T se používá k určení počtu iteracích, po které je daný atribut zakázaný. Může být statická nebo dynamická. K překonání tabu klasifikace lze použít *aspirační kritérium*, které umožňuje při splnění určité podmínky přijmout zakázaný S_{cand}. Příkladem je nalezení lepší S_{cand}, než je nejlepší dosud nalezený stav S_{best} [9].

Pro řešení náročnějších problémů lze přidat *dlouhodobou paměť*, která pomáhá lépe vyvážit proces diverzifikace a intenzifikace. Paměť je obvykle založena na frekvenci změn jednotlivých atributů. Pomáhá s lepším výběrem dalšího tahu [9]. Pseudokód algoritmu lze vidět na 3, kde M_{curr} je současný tah, M_{cand} je kandidátní tah a N velikost sousedství.

Algorithm 3 Tabu prohledávání

$S_{best} \leftarrow S_{curr} \leftarrow S_{init}$
while $\neg terminate()$ **do**
  $S_{origin} \leftarrow S_{curr}$
  $M_{curr}, S_{curr} \leftarrow neighbor(S_{origin})$
  for $n \leftarrow 1$ to N **do** ▷ Explore neighborhood
    $M_{cand}, S_{cand} \leftarrow neighbor(S_{origin})$
    if $(\neg is\_tabu(S_{cand}) \wedge is\_better(S_{cand}, S_{curr})) \vee aspire()$ **then**
      $S_{curr} \leftarrow S_{cand}$
      $M_{curr} \leftarrow M_{cand}$
    end if
  end for
  if $is\_better(S_{curr}, S_{best})$ **then**
    $S_{best} \leftarrow S_{curr}$
  end if
  $forget()$
  $remember(M_{curr}, T)$
end while
return S_{best}

Kapitola 2
Algoritmické obchodování

Algoritmické obchodování se používá k automatizaci obchodování pomocí počítačových programů. Hlavním cílem je maximalizovat zisk a zároveň snížit rizika. Vývoj těchto systémů vedl ke zrodu vysokofrekvenčního algoritmického obchodování, které drží pozice po extrémně krátkou dobu v řadech sekund až milisekund [10, 11].

Obchodování se uskutečňuje párováním nákupních a prodejních objednávek. Párovací engine si vede centralizovanou knihu objednávek (order book). Lze ji snadno vizualizovat jako tabulku se dvěma sloupci, prvním s nákupními objednávkami a druhým s prodejními objednávkami. Nákupní objednávky jsou řazeny sestupně, takže objednávka s nejvyšší cenou je první, jelikož cílem prodejce je prodat za co nejvyšší cenu. Prodejní objednávky jsou seřazeny v obráceném pořadí, protože kupující chce koupit za nejnižší možnou cenu. Po úspěšném spárování objednávek je provedena výměna (trade) [10].

Existují různé varianty knihy objednávek, které jsou závislé na dané burze. Mohou podporovat plnění různých typů objednávek, jako jsou limitní, tržní (market) nebo objednávky se stop-lossem. Tržní objednávky jsou provedeny okamžitě, zatímco limitní objednávky jsou splněny za stanovenou cenu nebo lepší. Obchodování se provádí buď přímo přístupem na burzu, nebo prostřednictvím brokera, jako jsou banky. Mezi aktiva, se kterými lze obchodovat, patří akcie, dluhopisy, opce, komodity, měnové páry, kryptoměny a deriváty [10].

Proces obchodování lze rozdělit do čtyř fází: předobchodní analýza, generování obchodních signálů, provedení obchodu a poobchodní analýza. Všechny lze automatizovat. Během předobchodní analýzy je aktivum analyzováno na základě finančních dat a zpráv. Výsledky se používají ke generování nákupních a prodejních signálů, které jsou následně převedeny na objednávky. Proces lze formulovat jako optimalizační problém, který lze řešit pomocí heuristických metod [10].

2.1 Předobchodní analýza

Předobchodní analýzu lze rozdělit do tří kategorií: *fundamentální*, *technická* a *kvantitativní* analýza. Cílem fundamentální analýzy je určit férovou cenu aktiva. Signály jsou obvykle generovány, když se aktuální cena aktiva liší od jeho férové ceny [10].

Technická analýza má za cíl předpovídat budoucí pohyb ceny aktiva na základě její historie a souvisejících informací, jako je objem. Předpokládá, že všechny informace potřebné pro generovaní signálu se odrážejí v její historii. Pracuje s konceptem trendů, které určují dlouhodobější směr pohybu ceny. Trendy lze organizovat do vzorů, jako jsou trendové linie, oblasti podpory a odporu. Používají se také technické ukazatele jako klouzavé průměry, index relativní síly (RSI), stochastické oscilátory nebo pohyblivá průměrná konvergence/divergence (MACD). Obchodní signály jsou generovány při změně trendu [10].

Kvantitativní analýza považuje ceny za náhodné a jejím cílem je vytvořit modely, které danou náhodnost dobře popisují. Tím se liší od předchozích dvou typů analýz, které předpokládají deterministické chování ceny. Stejně jako fundamentální analýza generuje obchodní signál, pokud se aktuální cena aktiva liší od jeho férové ceny [10].

2.2 Generování obchodního signálu

Výsledkem předobchodní analýzy jsou pouze doporučení k nákupu a prodeji, která jsou obohacena ve fázi generování obchodních signálu, kde jsou přidány informace, jako je cena a množství. Skládá se ze vstupní a výstupní strategie. Vstupní strategie určuje podmínky, kdy má program pozici otevřít, a výstupní strategie, kdy ji zavřít. Rizika a velikosti objednávek musí být také řízeny [10].

Vstupní strategie mohou být jednoduché, jako je překřížení hodnot dvou technických ukazatelů nebo odchýlení se od férové ceny. I když je cílem pravidla co nejvíce zjednodušit, může být nutné přidat některá pravidla navíc. Například otevírání obchodů může být příliš časté a na krátkou dobu, takže částka zaplacená na poplatcích převýší zisk. Výstupní strategie definuje, kdy uzavřít zisk a kdy odejít se ztrátou. Značně závisí na typu použité analýzy. V případě technické analýzy může být pozice uzavřena, když je dosaženo určitého cíle nebo když se skutečný trend ukáže být odlišný od předpovídaného [10].

Řízení rizik se zabývá otázkou, jaká částka by měla být investována do každé objednávky. Investovaná částka může být pevně stanovena pro každý obchod. Nevýhodou této metody je, že nezohledňuje volatilitu trhu. Může být méně riskantní investovat menší částky během období vysoké volatility a větší částky v obdobích nízké volatility. Další metodou, kterou lze použít, je pevný zlomkový systém (fixed fractional system), který riskuje pevnou část kapitálu na každý obchod [10].

2.3 Provedení obchodu

Dalším krokem je samotné provedení objednávky. Hlavní věc, které se systém snaží dosáhnout, je minimalizace nákladů. V této fázi musí být rozhodnuto o typu objednávky a burze, na kterou má být objednávka zadána. Mezi nejdůležitější aspekty, které je třeba vzít v úvahu při výběru správné burzy, patří nabídka aktiv, mechanismus obchodování, cena provedení, likvidita a stupeň anonymity obchodníka. Burzy s vyšší likviditou mají tendenci provádět objednávky rychleji a účtovat si nižší poplatky [10].

Kapitola 3
Zpětné testování

Základní myšlenkou zpětného testování (backtesting) je zjistit, jak by fungovala obchodní strategie, kdyby byla nasazena v minulosti. Vychází z předpokladu, že pokud strategie fungovala dobře v minulosti, měla by fungovat dobře i v budoucnu. Zpětné testování je nezbytnou součástí rozvoje obchodní strategie, jelikož pomáhá lépe porozumět testované strategii a dává příležitost k jejímu zlepšení [3].

Spuštění strategie na historických datech se nazývá *historická simulace*. Během procesu se shromažďují různé statistiky, které jsou používány k vyhodnocení strategie. Strategie jsou často parametrizované a nalezení jejich optimálních hodnot je NP-těžký problém. Jednoduché metody jako hrubá síla nebo prioritní krokové vyhledávání jsou většinou nepoužitelné, protože nedokončí vyhledávání v dostatečně krátkém čase. Proto se doporučuje používat sofistikovanější vyhledávací metody, jako je simulované ochlazování nebo genetický algoritmus [12, 3].

Statistiky shromážděné během historické simulace se používají k formulaci optimalizačního kritéria. Objektivní funkce založené na zisku jsou intuitivní volbou, ale ne nejlepší, protože mohou ignorovat podstatné nedostatky. Upřednostňuje se použití pokročilejších kritérií, jako je CECPP (Correlation Between Equity Curve and Perfect Profit) a PROM (Pessimistic Return On Margin), protože lépe odrážejí celkovou výkonnost strategie [12].

Dokonce i ta nejlepší nalezená konfigurace nemusí být nejlepší, protože může trpět jevem zvaným overfitting. Důsledkem je, že strategie funguje velmi dobře na historických datech, na kterých byla optimalizována, ale podstatě hůře ve skutečném obchodování [12].

3.1 Historická simulace

Nezbytnou součástí zpětného testování je historická simulace. Slouží k vyhodnocení obchodní strategie na historických datech. K provedení simulace musí být poskytnuta strategie a historická data. Na základě obchodů provedených během simulace jsou vypočítány statistiky [12].

Výsledky jsou často prezentovány jako zpráva, která obvykle zahrnuje souhrn výkonnosti (performance summary) a výpis obchodů. Souhrn výkonnosti se skládá ze statistik, které popisují celkovou výkonnost strategie. Výpis obchodů obsahuje seznam všech uskutečněných obchodů s informacemi, jako je vstupní a výstupní čas, vstupní a výstupní cena a dosažený zisk nebo ztráta. Mohou pomoci lépe porozumět tomu, jak se strategie chová během různých tržních období [12].

Simulace by měla být co nejpřesnější, jinak může vést k chybným závěrům a ztrátám při skutečném obchodování. Trhy používají různé zaokrouhlování svých cen. Pokud simulátor používá nesprávně zaokrouhlené ceny, může to mít za následek neprovedení objednávek nebo nesprávné hodnoty zisku. Chyby při zaokrouhlování mají obecně větší dopad na strategie, které obchodují častěji, protože rozdíl mezi otevírací a uzavírací cenou je menší [12].

Pokud jsou během simulace strategii poskytnuta data z většího časového rámce, jako jsou denní svíčky, tak může během daného období dojít k vytvoření více objednávek a jejich pořadí je nejednoznačné. V reálném obchodování se cena, podle které se strategie rozhoduje, často aktualizuje, takže pořadí objednávek je dané. Jedním ze způsobů, jak se zbavit nejednoznačnosti, je poskytnout strategii cenová data z nižšího časového rámce, jako jsou minutové svíčky. Dalším používaným řešením je uhodnout pořadí objednávek [12].

Započítány by měly být i poplatky (commission) za vyřízení objednávky. Bývají fixní a jejich velikost závisí na zprostředkovateli. Dalším nákladem je cenový skluz (price slippage), ke kterému dochází při plnění objednávky. Je to rozdíl mezi cenou objednávky (order price) a skutečnou cenou (fill price), za kterou bylo aktivum obchodováno. Skluz je nízký, když je trh klidný a je málo objednávek, a vyšší, když je trh rušnější. Záleží také na typu objednávky, velikosti objednávky a likviditě trhu [12].

3.1.1 Výkonnostní statistiky

Výkonnostní statistiky se používají k měření zisku a rizika. Díky tomu, že jsou nezávislé na konkrétní obchodní strategii, tak poskytují způsob, jak je porovnat. Statistiky lze nejen vypočítat na historických datech, ale lze je také použít pro měření výkonnosti běžící strategie v reálném čase. Umožňují tedy kontrolu průběhu obchodování. Zároveň jsou jediným způsobem, jak měřit a řídit riziko. Mohou zahrnovat statistiky jako: celkový počet obchodů, počet vítězných a ztrátových obchodů, největší vítězný obchod, největší ztrátový obchod, celkový čistý zisk (total net profit), hrubý zisk (gross profit) a hrubou ztrátu (gross loss) [12].

realizovaný zisk (realized profit) rp je částka ztracená nebo získaná po uzavření pozice. Lze ji vypočítat pomocí vzorce $rp = tr - ti$, kde tr je celkem získaná částka a ti je celkem investovaná částka. Může být také vyjádřena v procentech $rp\% = \frac{rp}{ti} \cdot 100$. Pokud $rp \leq 0$, tak se označuje jako ztráta, jinak jako zisk.

hrubý zisk (gross profit) $gp \geq 0$ je částka, která se spočítá jako součet realizovaných zisků všech ziskových uzavřených pozic $gp = \sum_{i=0}^{n} max(rp_i, 0)$, kde n je počet uzavřených pozic.

hrubá ztráta (gross loss) $gl \geq 0$ je částka, která se spočítá jako součet realizovaných ztrát všech ztrátových uzavřených pozic $gl = -\sum_{i=0}^{n} min(rp_i, 0)$, kde n je počet uzavřených pozic.

celkový zisk (net profit) np lze vypočítat buď součtem realizovaných zisků všech uzavřených pozic $np = \sum_{i=0}^{n} rp_i$, kde n je počet uzavřených pozic nebo $np = gp - gl$.

ziskový faktor (profit factor) $pf \geq 0$ lze spočítat $pf = \frac{gp}{gl}$. Pokud platí $pf > 1$, tak byla strategie zisková, jelikož celkové zisky převyšují celkové ztráty.

zůstatek (balance) $bn \geq 0$ je částka uložená na účtu.

současná hodnota $cv > 0$ je hodnota otevřené pozice, kdyby byla uzavřena za současnou tržní cenu mp. Lze ji vypočítat jako $cv = ps \cdot mp$, kde ps je velikost pozice.

současný zisk cp je částka, která byla získána nebo ztracena za dobu po otevření pozice. Lze ji spočítat pomocí $cp = cv - ti$ nebo v procentech $cp_p = \frac{cp}{ti} \cdot 100$.

čistá hodnota (equity) $eq \geq 0$ je částka, která by byla na účtu, kdyby byla uzavřena aktivní pozice za aktuální cenu. Lze ji spočítat jako $eq = bn + cv$.

maximální sestupný pohyb (max drawdown) $mdd \leq 0$ je největší pozorovaná ztráta z vrcholu p ke dnu t, do dosažení nového nejvyššího vrcholu. Lze ji vypočítat pomocí $mdd = t - p$ nebo v procentech $mdd_p = \frac{mdd}{p} \cdot 100$ [13].

maximální vzestupný pohyb (max run-up) $mru \geq 0$ je největší pozorovatelný zisk ze dna t k vrcholu p, do dosažení nového nejnižšího dna. Lze jej vypočítat pomocí $mru = p - t$ nebo v procentech $mru_p = \frac{mru}{t} \cdot 100$.

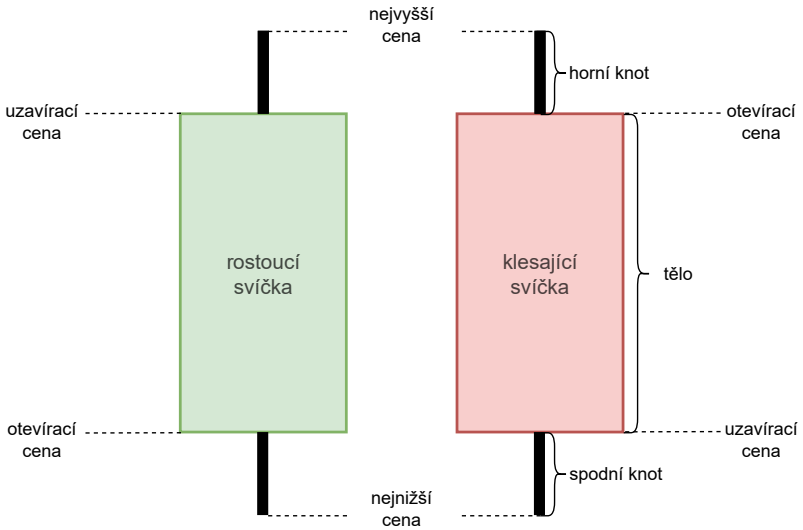
3.2 Vizualizace

Data získaná ze simulace lze vizualizovat pomocí grafu, což také pomáhá při ohodnocování a testování obchodní strategie. Historická data jsou vykreslena pomocí svíčkového grafu, do kterého lze dále vykreslit hodnoty ukazatelů, vstupní a výstupní body. Čistá hodnota v každém okamžiku je často vykreslena na samostatném grafu pod svíčkovým grafem [12].

3.2.1 Svíčkový graf

Svíčkový graf umožňuje vizualizaci a analýzu historických dat. Koncept vyvinul v 18. století japonský obchodník s rýží Munehisa Homma. Na západě byly představeny mnohem později na konci 20. století Stevem Nisonem. Graf může být použit k identifikaci vzorů a trendů [14].

Skládá se ze svíček, které jsou grafickým znázorněním čtyř klíčových cen: otevírací o, nejvyšší h, nejnižší l a uzavírací c ceny za daný časový interval. Platí, že $l \leq h$. Svíčka je tvořena třemi částmi: dvěma knoty a tělem. Knoty jsou znázorněny jako úsečky a tělo jako obdélník. Horní knot začíná od středu horní strany těla a končí v úrovni h. Spodní knot vede od středu spodní strany těla a končí na úrovni l. Horní a spodní strana těla svíčky označují o a c. Pokud platí $o < c$, svíčka je rostoucí, jinak klesající. Zároveň pokud je rostoucí, bývá tělo vybarveno zeleně, v opačném případě červeně (viz 3.1)[14].



Obrázek 3.1 Schéma rostoucí a klesající svíčky

Svíčky jsou většinou dostupné ve formátu OHLC, což je zkratka pro otevírací (open), nejvyšší (high), nejnižší (low) a uzavírací (close) ceny. Nejčastěji jsou ukládány do CSV souborů. Používají se také pro historickou simulaci, pro kterou je třeba tyto čtyři hodnoty zredukovat na jedinou. Obvykle se to provádí zprůměrováním cen a existuje několik běžně používaných metod: $hl2 = \frac{h+l}{2}$, $hlc3 = \frac{h+l+c}{3}$ nebo $ohlc4 = \frac{o+h+l+c}{4}$.

3.3 Historická tržní data

Historická tržní data jsou nezbytnou součástí zpětného testování, jelikož bez nich nelze provést historickou simulaci. Data je nutné pravidelně získávat a udržovat, často jen jejich získání v dostatečné kvalitě je obtížné [12]. Existují specializovaní poskytovatelé dat, jejichž cena se liší v závislosti na jejich kvalitě a množství.

Nejběžněji dostupná data jsou ve formátu OHLCV, což jsou OHLC a objem (volume) neboli množství obchodů uskutečněných v daném časovém intervalu. Dále jsou ukládány samotné obchody ve formě obchodních tiků (trade ticks). Pro dosažení větší přesnosti lze také použít nabídkové tiky (quote ticks), které ukládají údaje o nejlepší nabídce (bid) a poptávce (ask) z knihy objednávek v daném okamžiku [15].

3.4 PROM

PROM je robustní míra používaná k vyhodnocení obchodní strategie. Pesimistická je v tom smyslu, že předpokládá, že při reálném obchodování by byly zisky menší a ztráty větší než hodnoty získané z historické simulace. Lze jej vypočíst pomocí:

$$PROM = \frac{agp - agl}{ibn} \cdot 100,$$

kde agp je přizpůsobený hrubý zisk $((gp > 0) \implies (agp = \frac{gp}{wc} \cdot (wc - \sqrt{wc}))) \wedge ((gp = 0) \implies (agp = 0))$, kde wc je počet ziskových obchodů a gp je hrubý zisk. Výraz agl je přizpůsobená hrubá ztráta $((gl > 0) \implies (agl = \frac{gl}{lc} \cdot (lc + \sqrt{lc}))) \wedge ((gl = 0) \implies (agl = 0))$, kde lc je počet ztrátových obchodů a gl je hrubá ztráta. Poslední člen ibn je počáteční zůstatek [12].

3.5 Obchodní platformy

V dnešní době existuje mnoho platforem nabízejících zpětné testování. Liší se především druhem a množstvím dostupných historických dat, propracovaností, s jakou lze strategii implementovat, a částkou, kterou si účtují za používání jejich produktu [16]. Podrobněji jsou popsány dvě platformy, které byly vybrány z různých důvodů.

TradingView je platforma, kde byla vyvinuta strategie optimalizovaná v praktické části a kde aktuálně probíhá zpětné testování. Nabízí převážně možnosti práce s grafy. Zpětné testování je velmi jednoduché a neposkytuje způsoby, jak optimalizovat parametry strategie. Je to převážně webová aplikace a pokud by nad ní mělo být postaveno vlastní řešení, postrádalo by rychlost k dosažení rozumných výsledků.

MultiCharts je řešení, které nabízí mnoho funkcí používaných pro automatizaci obchodování. Je zdarma po omezenou dobu a je k dispozici jako desktopová aplikace. Nabízí požadovanou funkcionalitu ve formě genetického algoritmu, ale není tak flexibilní. Oproti TradingView je však mnohem více zaměřena na efektivitu.

3.5.1 TradingView

TradingView je oblíbená obchodní platforma, která umožňuje provádět technickou analýzu, vývoj a zpětné testování obchodní strategie. Je přístupná jako webová, mobilní i desktopová aplikace. Nabízí širokou škálu tržních dat, jako jsou akcie, kryptoměny, indexy, energetické a zemědělské komodity, drahé kovy a Forex. Základní funkce lze používat zdarma a pro přístup k pokročilejším jsou k dispozici různé cenové plány [16].

Používá se především pro vizualizaci dat a technickou analýzu, k čemuž nabízí nástroje pro kreslení do grafů a velké množství technických indikátorů. Výsledné nápady lze uložit a sdílet s ostatními uživateli. Grafy lze vizualizovat a organizovat různými způsoby [17].

Strategie lze specifikovat pomocí programovacího jazyka PineScript a testovat pomocí zpětného testování, které je poměrně jednoduché. Zpětné testování nabízí způsob nastavení strategie a vizualizace výsledků, které jsou k dispozici jako souhrn výkonnosti a seznam obchodů. Neposkytuje způsob optimalizace parametrů [17].

3.5.2 MultiCharts

MultiCharts je obchodní platforma, která nabízí nástroje pro vykreslování grafů, vestavěné indikátory a strategie, zpětné testování a optimalizace. Umožňuje napojení na různé dodavatele tržních dat a brokery. Lze ji vyzkoušet zdarma na měsíc, pravidelně platit nebo zakoupit celoživotně [18].

Nástroj pro vytváření grafů umožňuje vykreslovat více dílčích grafů v různých rozlišeních, přidávat technické indikátory, komentáře a značky. Lze vykreslit historická data společně se současnými. Podobu a uspořádání grafů lze do velké míry přizpůsobit. Jsou k dispozici také kreslící nástroje pro zvýraznění trendů, podpůrných (support) nebo odporových (resistance) oblastí [18].

EasyLanguage programovací jazyk lze použít pro implementaci obchodních strategií a technických indikátorů. Jedná se o jednoduchý jazyk, který se používá již 20 let a je podporován více obchodními platformami. Jakmile je automatizovaná obchodní strategie vyvinuta, může být testována. V rámci zpětného testování platforma poskytuje realistickou historickou simulaci, optimalizaci strategie a walk forward testování [18].

Platforma nabízí dva druhy optimalizace hrubou sílu a genetický algoritmus. Algoritmy jsou napsány vícevláknově a výsledek optimalizace lze zobrazit jako zprávu. Podporují použití různých optimalizačních kritérií. Dále je možné využít walk forward testování, jehož hlavním účelem je předcházet overfittingu. Při testování jsou data rozdělena do více segmentů, které jsou dále rozděleny na dvě části, vzorky použité pro optimalizaci a vzorky použité pro validaci [18].

3.6 Bazooka obchodní strategie

Pro optimalizaci v praktické části práce byla vybrána obchodní strategie s názvem *Bazooka*, která byla vytvořena a poskytnuta externím subjektem. Jedná se o jednoduchou dobře definovanou a dříve otestovanou strategii, která pro rozhodování používá ukazatele klouzavého průměru. Využívá také mechanismus řízení rizik. Strategie byla implementována na základě osobních setkání a ukázkové implementace v jazyku *PineScript* pro platformu *TradingView*.

K vyvolání otevíracího nebo uzavíracího signálu se používají klouzavé průměry, které mohou být typu EMA nebo SMA. Obecně mohou být různé, například vstupní ukazatel může být typu EMA s periodou 30 a výstupní ukazatel může být typu SMA s periodou 20. Ukazatelé jsou aktualizovány pomocí průměrné ceny historických svíček v daném intervalu. Pokud by byly například použity 30minutové svíčky, znamenalo by to, že by se ukazatele aktualizovaly každých 30 minut s cenou získanou průměrem právě uzavřené 30minutové svíčky.

Strategie využívá více úrovní nákupu, které se získávají posunutím hodnoty vstupního ukazatele směrem dolů. Hodnota nákupní úrovně lv_i se získá pomocí $lv_i = env \cdot l_i$, kde env je hodnota vstupního ukazatele a l_i je nákupní úroveň. Pro úrovně nákupu platí, že $L = \{l_0, l_1, \ldots, l_n\} \in \mathbb{Q} \cap (0, 1) \wedge n > 0 \wedge [\forall i \in \{0, \ldots, n-1\} : l_i > l_{i+1}]$, kde n je počet nákupních úrovní. Nákupní signál je vyvolán, pokud nastane $mp \leq lv_c$, kde mp je současná tržní cena aktiva a lv_c je hodnota současné nákupní úrovně. Nákup na každé úrovni lze provést pouze jednou.

Investovaná částka ia_i pro nákupní úroveň i je určena $ia_i = bn \cdot s_i$, kde bn je zůstatek na účtu před otevřením pozice a s_i velikost nákupu. Pro velikosti nákupu platí, že $S = \{s_0, s_1, \ldots, s_n\} \in \mathbb{Q} \cap (0, 1] \wedge n > 1 \wedge \sum_{i=0}^{n} s_i = 1$, kde n je počet nákupních úrovní. Pokud například $(bn = 100 \wedge S = \{\frac{2}{10}, \frac{5}{10}, \frac{3}{10}\}) \implies ia = \{20, 50, 30\}$.

Prodejní signál je vyvolán, pokud platí $mv \geq exv$, kde exv je hodnota výstupního ukazatele a byla otevřena pozice. Na rozdíl od vstupní strategie, která zvětšuje velikost pozice po částech, výstupní strategie prodává celou pozici najednou. Strategie obchoduje, dokud není dosaženo minimální čisté ceny nebo je ručně vypnuta.

Použití strategie lze vidět na grafu 3.2. Konfigurace strategie je $n = 3, L = \{\frac{55}{64}, \frac{53}{64}, \frac{52}{64}\}$, $S = \{\frac{1}{8}, \frac{6}{8}, \frac{1}{8}\}$, typ vstupního a výstupního ukazatele je SMA s periodou 15. Ve výřezu jsou vidět dvě ziskové pozice, kde se první skládá ze dvou otevřených objednávek a druhá ze tří. Historická data jsou minutové svíčky, které jsou také vyneseny do grafu. Perioda převzorkování je 45, což znamená, že 45 minutových svíček se převzorkuje na jednu 45minutovou svíčku, která je nejprve zprůměrována metodou OHLC4 a následně použita k aktualizaci ukazatelů. Hodnoty vstupních a výstupních úrovní jsou vyneseny pomocí bodového grafu a odpovídají době aktualizace ukazatelů, což znamená, že jsou body od sebe vzdáleny 45 minut.

Obrázek 3.2 Ukázka použití strategie

3.7 Klouzavé průměry

Klouzavé průměry jsou technické ukazatele, jejichž cílem je určit trend daného aktiva výpočtem průměru jeho cen. Existují tři běžně používané ukazatele: jednoduchý klouzavý průměr (SMA), exponenciální klouzavý průměr (EMA) a vážený klouzavý průměr (WMA). Jsou počítány za určitou periodu $k > 0$, která určuje počet vzorků použitých k výpočtu. Pokud je proveden SMA na cenách představujících 10 dní, nazývá se 10denní SMA. Obvykle jsou vizualizovány společně s grafy cen akcií. Křížení indikátorů lze použít k identifikaci změny trendu [19].

3.7.1 Jednoduchý klouzavý průměr

Jednoduchý klouzavý průměr (SMA) je nejběžněji používaným ukazatelem klouzavého průměru a jak název napovídá, nejsnáze se počítá. Průměrnou hodnotu posledních k cen lze vypočítat pomocí $SMA_k = \frac{1}{k} \sum_{i=n-k+1}^{n} p_i \wedge n \geq k$, kde n je celkový počet cen a p jsou samotné ceny. Pro

výpočet následující hodnoty lze využít $SMA_{k,next} = SMA_{k,prev} + \frac{1}{k}(p_{n+1} - p_{n-k+1})$, kde p_{n+1} je nejnovější a p_{n-k+1} nejstarší cena. Lze jej efektivně implementovat pomocí datové struktury kruhový buffer. Nejčastěji používané SMA jsou období 50, 100 a 200 dnů [19, 20].

3.7.2 Exponenciální klouzavý průměr

Na rozdíl od SMA, kde má každá hodnota při výpočtu průměru stejnou váhu, tak exponenciální klouzavý průměr (EMA) přiřazuje cenám váhy. Hodnoty vah klesají exponenciálně, takže novější hodnoty mají větší váhu, a tím i větší vliv na výslednou průměrnou hodnotu. Díky tomu reaguje rychleji na změny ceny aktiva než SMA. Obvykle se používá k výpočtu průměru za kratší časové období [19, 21].

Počáteční hodnota se vypočítá pomocí SMA se stejnou periodou. Následující hodnoty lze získat použitím $EMA_{k,next} = p_c \cdot wf + EMA_{k,prev} \cdot (1 - wf) \wedge wf = \frac{sf}{k+1} \wedge sf \geq 2$, kde p_c je současná cena, wf je vážený faktor a sf je vyhlazovací faktor. Vážený faktor se používá k určení množství váhy, která je přiřazena nejnovější hodnotě. Například pokud $k = 10 \wedge sf = 2 \implies wf = 0,1818$, což znamená, že 18,18 % celkové váhy je přiřazeno nejnovější hodnotě [22].

Kapitola 4
Volba technologie

Pro dosažení větší flexibility, rychlosti a hlubšího porozumění zpětného testování a optimalizace byl vytvořen vlastní framework. Je napsán ve standardu C++20, což je nejnovější dostupný standard jazyka C++. Jazyk byl zvolen především proto, že nabízí vysokou úroveň abstrakce a prostředky pro efektivnější práci se systémovými zdroji.

Výstupy aplikace jsou zpracovávány skripty v jazyce Python, který je řádově jednodušší než C++ a snáze se používá. Poskytuje přístup k mnoha vynikajícím vědeckým balíkům, které se snadno instalují a jsou dobře zdokumentované. Byl především použit pro automatizaci vytváření přehledových tabulek a grafů.

4.1 C++

C++ je programovací jazyk na vysoké úrovni vyvinutý v roce 1983 *Bjarnem Stroustrupem* v Bell Labs jako rozšíření programovacího jazyka *C*. Jedná se o multiparadigmatický programovací jazyk, který podporuje datovou abstrakci, objektově orientované programování, procedurální programování a generické programování [23].

V programátorské komunitě je dobře známý a hojně používaný například pro vývoj herních enginů, prohlížečů, aplikací a provádění vysoce-výkonných výpočtů (HPC). Na základě indexu TIOBE za březen 2023 byl ohodnocen jako čtvrtý v popularitě, velmi těsně za programovacím jazykem Java. Byl také vyhodnocen jako programovací jazyk roku 2022 stejnou společností díky jeho rostoucí popularitě [24].

Mezi verzemi C++03 a C++11 byla obrovská mezera, což mohl být důvod, proč mnoho lidí od jazyka raději ustoupilo. Od C++11 se však každé 3 roky objevují nové verze jazyka, které zavádějí nové funkce, zlepšují bezpečnost práce s pamětí a pomáhají získat větší důvěru v jazyk samotný. Posledním vydaným standardem je C++20 a nejnovější standard se chystá být vydán letos.

Je to bohatý jazyk s rozsáhlou standardní knihovnou. Ve srovnání s jinými modernějšími jazyky, jako je Python, je však obtížnější se jej naučit a správně používat. Jazyk je staticky typovaný, což znamená, že typ objektu musí být určen při jeho deklaraci. Na rozdíl od Pythonu, který je interpretován za běhu, je C++ kód kompilován. Ve srovnání s Pythonem trvá psaní kódu obecně více času, jelikož programátor musí být konkrétnější, což může určitou skupinu programátorů odradit. Má ale jednu věc, kterou Python postrádá, a tou je rychlost.

C++ nabízí generické programování, k čemuž jsou používány šablony. Existují pouze během kompilace. Umožňují použití statického polymorfismu, který za běhu stojí méně prostředků než známější dynamický polymorfismus využívající virtuální funkce. V nejnovějších standardech byly zavedeny třídy, které se vyhýbají použití dynamického polymorfismu, jako jsou: `std::optional`

a `std::variant`.

Další věcí, kterou nabízí ke zrychlení kódu, jsou optimalizace, které lze nastavit během kompilace. Pracují s konceptem pozorovatelného chování, které umožňuje produkci jiného byte kódu, který dělá stejnou věc rychleji. Využívá proto datovou a funkční nezávislost.

Paralelní zpracování lze také použít ke zrychlení kódu. Program může být spuštěn na jiném procesu nebo různých vláknech nebo obojí. Standardní knihovna nabízí implementaci vláken a souvisejících prostředků, lze však použít i OpenMP framework. Zpracování na více procesech najednou lze zajistit například použitím MPI frameworku. Ne všechny programy jsou vhodné pro paralizaci, ale je dobré, že existuje možnost ji použít. Například v Pythonu existuje koncept vláken, nicméně v rámci jednoho procesu může současně běžet pouze jedno vlákno díky Global Interpreter Lock (GIL).

Kromě standardní knihovny je k dispozici mnoho open-source, dobře navržených a testovaných knihoven. Existují knihovny Boost, které nabízejí knihovny pro síťování (Asio), unit testování (Test), paralelní zpracování (MPI) a mnoho dalších, jako je POCO, JSON pro moderní C++, Google Test a OpenCV.

4.2 Python

Python je dynamicky typovaný, interpretovaný programovací jazyk, který vytvořil Guido van Rossum v roce 1991. Podporuje objektově orientované i funkční programování. Je vyvíjen a spravován organizací Python Software Foundation. Nejnovější verze jazyka je 3.11 [25].

Ve srovnání s C++ je mnohem snazší se jej naučit. Vyznačuje se jasnou a čitelnou syntaxí. Má rozsáhlou standardní knihovnu a dva správce balíčků Pip a Conda. V březnu 2023 se umístil na 1. místě v oblíbenosti podle indexu TIOBE. Používá se hlavně pro vývoj webových aplikací, skriptování a strojové učení.

V posledních letech je to dominantní jazyk používaný ve strojovém učení díky frameworkům jako jsou: TensorFlow, PyTorch a scikit-learn. Existují také balíčky datové vědy, jako jsou: NumPy, SciPy, Pandas a Matplotlib. Všechny je lze použít v Jupyter Notebook, což je interaktivní webová výpočetní platforma, která usnadňuje zpracování a vizualizaci dat.

Hlavní nevýhodou jazyka je, že ve srovnání s jazyky jako C++ je pomalý. Existují balíčky, jako je NumPy, které jsou napsány částečně v C programovacím jazyku, nebo dokonce vysoce výkonné kompilátory, jako je Numba, které lze použít k urychlení kódu. Obecně je však z hlediska výkonu lepší použít jiný jazyk. Co však Python ztrácí na výkonu, to vynahradí vysokou flexibilitou, rozsáhlou komunitou programátorů a velkým množstvím užitečných balíčků.

Kapitola 5

Návrh

Vzhledem k tomu, že projekt vznikl od nuly, byl zde obrovský prostor pro využití moderních funkcí programovacího jazyka C++, dodržování programovacích principů a použití návrhových vzorů. Byly také použity známé a osvědčené knihovny jako Boost.Test, OpenMP a CppCoro. Z moderních funkcí C++ používaných v projektu nejvíce vynikají korutiny a koncepty.

Pro dosažení vyšší efektivity za cenu nižší flexibility byl preferován statický polymorfismus před dynamickým. Statický polymorfismus je realizován pomocí šablon, a protože existují jenom v době kompilace, nezpůsobují při běhu programu režii navíc. Na druhou stranu dynamický polymorfismus vyžaduje dynamickou alokaci a volání (dispatch).

Požadavky na šablonové argumenty lze specifikovat pomocí konceptů. Současné použití a vzhled byl inspirován videem z konference CppCon 2022 o dependency injection, kde bylo představeno použití konceptů pro specifikaci rozhraní šablonových argumentů [26]. Rozhraní jsou psána dvěma způsoby, pro kontrolu chování volatelného objektu a svazku více metod. Implementaci pro kontrolu třídy pozorovatele a tabu lhůty lze vidět 5.1.

Výpis kódu 5.1 Specifikace rozhraní pomocí konceptů

```
// concept used to check behaviour
template<class Tenure>
concept ITenure = std::invocable<Tenure> &&
    std::same_as<std::size_t, std::invoke_result_t<Tenure>>;

// concept used to check a bundle
template<class Observer, class Optimizer>
concept IObserver = requires(Observer& observer,
    const Optimizer& optimizer) {
    { observer.started(optimizer) } -> std::same_as<void>;
    { observer.better_accepted(optimizer) } -> std::same_as<void>;
    { observer.worse_accepted(optimizer) } -> std::same_as<void>;
    { observer.cooled(optimizer) } -> std::same_as<void>;
    { observer.finished(optimizer) } -> std::same_as<void>;
};
```

Korutiny byly představeny ve standardu C++20. Při volání funkce umožňují její provádění pozastavit a později obnovit [27]. Byly použity k systematickému generování vyhledávacího prostoru. Nejsou zatím široce používány, ale existuje knihovna CppCoro, která poskytuje implementace generátorů `cppcoro::generator` a `cppcoro::recursive_generator` [28].

5.1 Jmenné konvence

Konvence pojmenování jsou převzaty převážně z STL a Boost knihoven. Názvy tříd a funkcí používají `snake_case`. Metody nastavení a čtení atributů třídy nepoužívají `set` a `get` ve svém názvu. Jsou jednoduše odlišeni tak, že gettery neberou žádný parametr a setry berou více než jeden. Názvy šablon a konceptů používají `CamelCase`. Soukromé atributy třídy končí podtržítkem.

5.2 Uspořádání projektu

Projekt je rozdělen do tří hlavních adresářů: include, src a test. Adresář include obsahuje obchodovací framework a vybrané hlavičkové soubory z knihovny CppCoro. Organizace frameworku je inspirovaná knihovnou Boost.Asio. Definice a deklarace tříd jsou uloženy společně v jednom souboru s příponou .hpp. Hierarchie složek se řídí jmennými prostory, například třídu `trading::bazooka::strategy` lze nalézt v souboru /include/trading/bazooka/strategy.hpp. Framework lze zahrnout jako celek pomocí trading.hpp nebo jednotlivě po souborech.

Adresář src se skládá ze souboru main.cpp, Jypiter Notebooků používaných pro vizualizaci, vstupních a výstupních dat. Testy jsou organizovány stejně jako třídy obchodovacího frameworku. Projekt je kompilován pomocí souborů CMakeLists.txt, které obsahují seznam souborů ke kompilaci, nastavení kompilátoru a informace pro hledání externích knihoven. Vytváří dva spustitelné soubory backesting pro použití optimalizátorů a test_ pro provádění testů.

5.3 Návrhové vzory

Účelem návrhových vzorů je snížit složitost kódu a šanci udělat chybu. Existuje celá řada známých a dobře popsaných návrhových vzorů, je však nutné rozpoznat situace, kdy je vhodné je použít. Obecně slouží jako řešení běžně se vyskytujících problémů, nelze je však importovat z knihovny. Je to jen koncept s nějakou základní myšlenkou, kterou lze aplikovat různým způsobem v závislosti na konkrétním problému. V následujících částech jsou popsány pouze ty, které byly použity v samotné práci [29].

5.3.1 Pozorovatel

Návrhový vzor pozorovatel (observer) umožňuje objektům nazývaným pozorovatelé být upozorňováni na události, které nastanou u pozorovaného objektu. Pozorovatelé se přihlásí k odběru oznámení o událostech, které nastávají u pozorovaného objektu, a jsou upozorněni, kdykoli k nim dojde [30].

Pozorovaný objekt musí být nejprve předán pozorovanému objektu, obvykle pomocí vyhrazené metody. Poté, kdykoli dojde k události, je zavolána specifická metoda pozorovatele s událostí spojená. Tímto způsobem není nutné aktivně čekat, jelikož pozorovatel a pozorovaný jsou časově synchronizováni. Musí pouze splňovat dohodnuté rozhraní [30].

Vzor pozorovatel se používá pro upozornění na události, které nastanou během optimalizace a historické simulace. Informace shromážděné během simulace lze použít k výpočtu výkonnostních statistik nebo ke sběru dat pro vizualizaci. Pozorovatelé jsou předáváni jako sada šablonových parametrů při volání instance třídy, což umožňuje předání žádného, jednoho nebo více pozorovatelů, kteří mohou být různého typu, avšak musí být známy v době kompilace.

5.3.2 Strategie

Cílem použití návrhového vzoru strategie (strategy) je zapouzdřit rodinu podobných algoritmů se stejným účelem do samostatných tříd a učinit je vzájemně zaměnitelnými. Třídy těchto algoritmů

se nazývají strategie [31].

Třída používající strategii se většinou nazývá kontext. Neví nic o strategii, která je ji předána. Jediné, co musí strategie splňovat, je dohodnuté rozhraní. Díky tomu je kontext nezávislý na konkrétní strategii. Volba vhodné strategie je na uživateli aplikace. Algoritmy obvykle implementují jednu společnou metodu, která musí přebírat stejnou sadu argumentů. Voláním této metody se spustí daný algoritmus [31].

V práci je vzor strategie použit pro výběr různých druhů rozvrhů chlazení pro algoritmus simulovaného ochlazování nebo ukončovacích kritérií. Strategie chlazení je předána instanci optimalizátoru jako parametr při jeho volání. Musí splňovat koncept `simulated_annealing::ICooler`. Jelikož různé algoritmy chlazení vyžadují různé sady argumentů, je při volání strategie předána přímo instance optimalizátoru. Volba požadovaného parametru jako je aktuální teplota nebo počet proběhlých iterací je na strategii samotné. Je navržen hlavně pro použití pro statický polymorfismus. Nicméně by bylo možné mezi strategiemi přepínat za běhu pomocí `std::function`, který používá *type erasure*.

Kapitola 6
Simulace obchodování

Simulace obchodování byla inspirována kryptoměnovou burzou Binance. Byly přijaty konvence pojmenování, jako je otevírání a zavírání pozic. Otevření longové pozice znamená nákup aktiva, nicméně otevření shortové pozice znamená jeho prodej. Prodej a nákup tedy jdou proti sobě, a proto je lepší používat termíny otevření a uzavření pozice.

Je také běžné zaměnitelně říkat, že obchod nebo pozice jsou otevřené. Na *Binance* je možné si vyžádat historii uskutečněných obchodů. Vygenerované obchody jsou pouze transakce provedené mezi dvěma obchodníky. Proto je lepší používat termín pozice, který představuje seznam souvisejících otevřených a uzavřených obchodů.

6.1 Bazooka

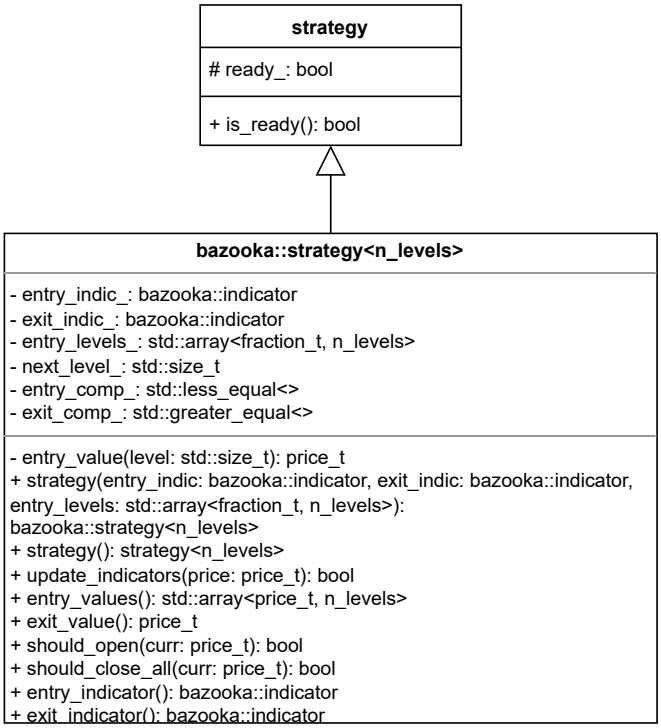
Bazooka obchodní strategie byla rozdělena do dvou tříd `bazooka::strategy` a `bazooka::manager` za účelem oddělení zájmů. Třída `bazooka::strategy` dělá rozhodnutí a třída `bazooka::manager` je realizuje. Je implementována pouze burza podporující pouze longové obchody, ale pokud by v budoucnu existovala např. `futures::market` burza podporující použití páky (leverage), bylo by nutné použít jiný typ manažera, nicméně strategie by zůstala stejná. Jsou dány dohromady pomocí třídy `trader`.

Hodnoty nákupních velikostí a úrovní jsou uloženy jako pole zlomků pevné délky. Zlomky jsou reperezentovány pomocí třídy `fraction`, která je inspirována třídou `boost::rational`. Je však mnohem jednodušší a méně obecná. Předpokládá, na rozdíl od `boost::rational`, že zlomky, mezi kterými se provádějí aritmetické operace, mají stejného jmenovatele. Porušení tohoto předpokladu vede k nedefinovanému chování. Zlomky jsou uloženy pomocí `std::array`, což také znamená, že počet nákupních úrovní musí být znám v době kompilace.

6.1.1 Strategie

Strategie je implementována pomocí třídy `bazooka::strategy`. Má jeden parametr šablony třídy `n_levels`, který uvádí počet úrovní nákupu (viz 6.1. Při inicializaci přebírá vstupní a výstupní ukazatele a úrovně nákupu. V průběhu inicializace jsou také validovány nákupní úrovně, pokud nesplňují požadavky, je vyhozena výjimka `std::invalid_argument`.

Účelem třídy je činit rozhodnutí a k tomu má dvě metody `should_open` a `should_close`. Obě metody používají indikátory, aby zjistily, zda byly splněny podmínky pro vyslání signálu. Vracejí booleovskou hodnotu, která určuje, jestli by měla být akce provedena. V jednu chvíli může nastat pouze jedna událost. Indikátory jsou aktualizovány pomocí metody `update_indicators`. Vstupní a výstupní hodnoty lze získat voláním getrů třídy. Lze je použít pro vykreslování grafů.



Obrázek 6.1 Rozhraní třídy `bazooka::strategy`

6.1.2 Ukazatel

Ukazatel může být typu SMA nebo EMA. Běžným způsobem, jak umožnit záměnu obou typů za běhu programu, by bylo použití runtime polymorfismu prostřednictvím virtuálních funkcí. Nicméně místo toho byla použita třída `std::variant` zavedená standardem C++17, která funguje jako uložiště pro různé datové typy. Pro uložení instance daného typu používá staticky alokované pole bajtů, který má velikost největšího uložitelného typu. Jako třídní šablonové parametry přebírá seznam typů, které mohou být v objektu uloženy.

K hodnotě uložené v instanci objektu lze přistupovat pomocí funkce `std::get`. Funkce přijímá šablonový argument, který je buď indexem typu, nebo samotným typem, který by měl být aktuálně v instanci uložen. Pokud je uložen jiný typ, tak volání funkce způsobí vyhození `std::bad_variant_access` výjimky. Dalším způsobem přístupu k uložené hodnotě je použití funkce `std::visit`. Vyžaduje dva argumenty, první je volatelný objekt a druhý instance třídy `std::variant`. Funkce při volání nejprve získá hodnotu uloženou v instanci a poté ji předá volatelnému objektu. V případě ukazatele může být použita například při jeho aktualizaci (viz 6.1).

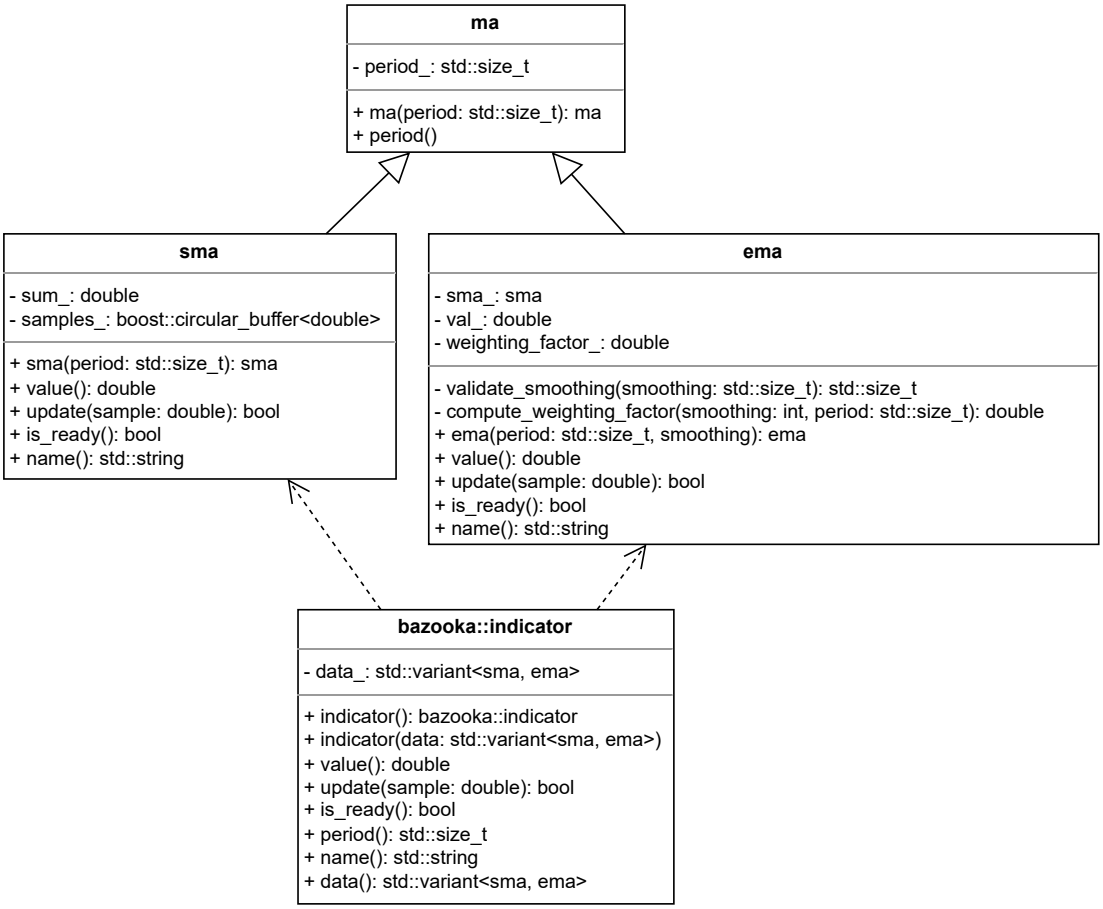
Výpis kódu 6.1 Aktualizace indikátoru pomocí std::visit

```
std::variant<sma, ema> ma{sma{30}};
bool ready = std::visit([&value](auto& indic) {
    return indic.update(value);
}, ma);
```

Instance `std::variant` byla zapouzdřena do třídy `bazooka::indicator`, které usnadňují práci s uloženým ukazatelem. Třída má pouze jeden členský atribut `data_` typu `std::variant<sma, ema>`. Metody používají `std::visit` pro přístup a změnu hodnoty ukazatele. Sdílí stejné rozhraní s třídami `sma` a `ema` (viz 6.2).

Ukazatele jsou aktualizovány voláním metody `update` a jejich hodnotu lze získat voláním metody `value`. Metoda `update` vrací hodnotu boolean indikující, zda je ukazatel připraven. Lze to také zkontrolovat voláním metody `is_ready`.

Počáteční hodnota ukazatele `ema` je získána pomocí ukazatele `sma` se stejnou periodou. Další hodnoty se vypočítají pomocí vzorce, který používá aktuální hodnotu. Třída `sma` používá instanci třídy `boost::circular_buffer` k uložení předchozích hodnot pro výpočet průměru. Jakmile je buffer plný, je ukazatel připraven.



Obrázek 6.2 Rozhraní tříd typu ukazatel

6.1.3 Manažer

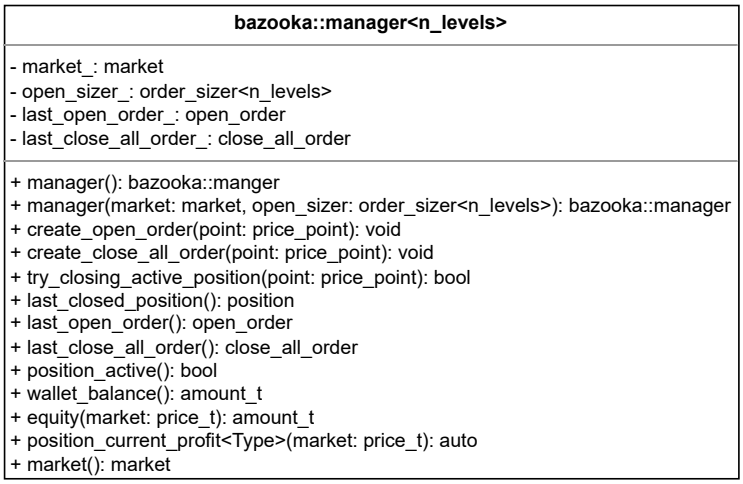
Účelem manažera je realizovat akce vytvořené strategií. Je implementován pomocí třídy `manager`. Komunikuje s burzou, vytváří objednávky a řídí rizika. Dvě nejdůležitější metody pro vytváření objednávek jsou `create_open_order` a `create_close_all_order` (viz 6.3).

Otevírací objednávky jsou reprezentovány strukturou `open_order`, která obsahuje tři atributy: prodané množství, cenu, za kterou bylo aktivum nakoupeno a čas jejího vytvoření. Uzavírací objednávka je implementována strukturou `close_all_order`, která na rozdíl od otevírací objednávky neuvádí prodané množství, jelikož je prodáno vše.

Strategie používá n nákupních úrovní. Vždy začíná obchodovat s počátečním zůstatkem, který je rozdělen na n velikostí nákupu. Částku investovanou na každou objednávku lze vypočítat buď na začátku, nebo průběžně. Výpočet částek za běhu se zdálo být přirozenější, takže byla

implementována třída `order_sizer`. Při konstrukci přijímá pole nákupních velikostí S a převádí jej na pole akumulovaných velikostí a, kde platí, že $s_0 = a_0 \wedge a_n = 1 \wedge [\forall i \in \{0, \ldots, n\} : a_i = \sum_{j=0}^{i} s_j]$. Po zavolání instance je vynásoben zůstatek aktuální akumulovanou velikostí pro získání velikosti objednávky.

Vzhledem k tomu, že má manažer přístup k burze, tak poskytuje metody pro přístup k jejímu aktuálnímu stavu, jako je zůstatek peněženky, čistá hodnotu a aktuální zisk aktivní pozice. Jakmile je objednávka provedena, je uložena jako poslední otevřená/uzavřená objednávka a lze k ni přistupovat prostřednictvím getrů třídy. Je to cenná informace pro vykreslování grafů a výpočet statistik. Může nebo nemusí být použita, takže je lepší ji uložit samostatně a neustále ji přepisovat, než všechny ukládat například do instance `std::vector`, což by způsobilo zbytečnou režii navíc.



Obrázek 6.3 Rozhraní třídy `bazooka::manager`

6.1.4 Obchodník

Funkcionality `manager` a `strategy` jsou propojeny pomocí `trader`. Pro zpřístupnění veřejných metod obou tříd je použita vícenásobná veřejná dědičnost. Volací operátor třídy (viz 6.2) používá metody obou tříd. Strategie činí rozhodnutí, která manažer uskutečňuje. Metoda vrací výčet `action` udávající, která akce byla provedena.

Výpis kódu 6.2 Implementace volacího operátoru třídy `bazooka::trader`

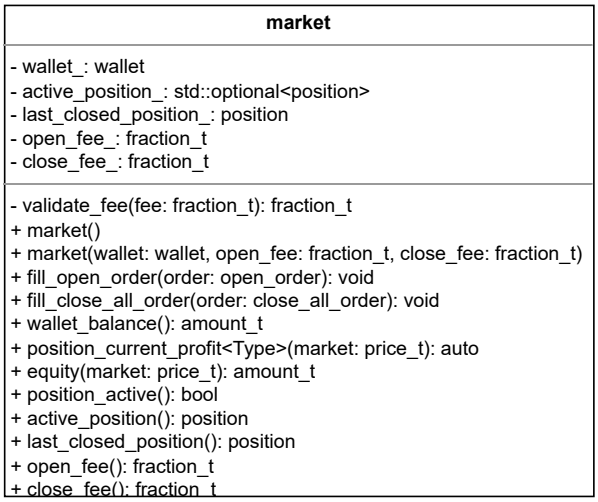
```
action operator()(const price_point& curr)
{
    action done{action::none};
    if (!Strategy::is_ready()) return done;

    if (Strategy::should_open(curr.data)) {
        Manager::create_open_order(curr);
        done = action::opened;
    }
    else if (Strategy::should_close_all(curr.data)) {
        Manager::create_close_all_order(curr);
        done = action::closed_all;
    }
    return done;
}
```

6.2 Trh

Trh je implementován pomocí třídy `market`. Ve srovnání se skutečnou burzou je velmi zjednodušená. Podporuje pouze objednávky typu long a jeden uživatelský účet. Mezi třídní atributy patří aktivní pozice, poslední uzavřená pozice, peněženka a výše poplatků spojených s realizací objednávek (viz 6.4).

Peněženka je implementována třídou `wallet` a slouží ke správě zůstatku zákaznického účtu. Zůstatek lze snížit voláním metody `withdraw` nebo navýšit použitím `deposit`. Pokud je částka k výběru větší než aktuální zůstatek, tak je vyhozena výjimka `insufficient_funds`. Aktivní pozice je uložena v instanci `std::optional`, jelikož pozice nemusí být otevřena.

Pomocí metody `fill_open_order` může být aktivní pozice otevřena nebo navýšena v závislosti na tom, zda již existuje. Po otevření lze pozici uzavřít jako celek pomocí `fill_close_all_order` metody, nikoli po částech. Třída také poskytuje getry pro získání přístupu k aktuální čisté hodnotě, kontrolu, zda je otevřená pozice, nebo zjištění aktuálního zisku aktivní pozice.



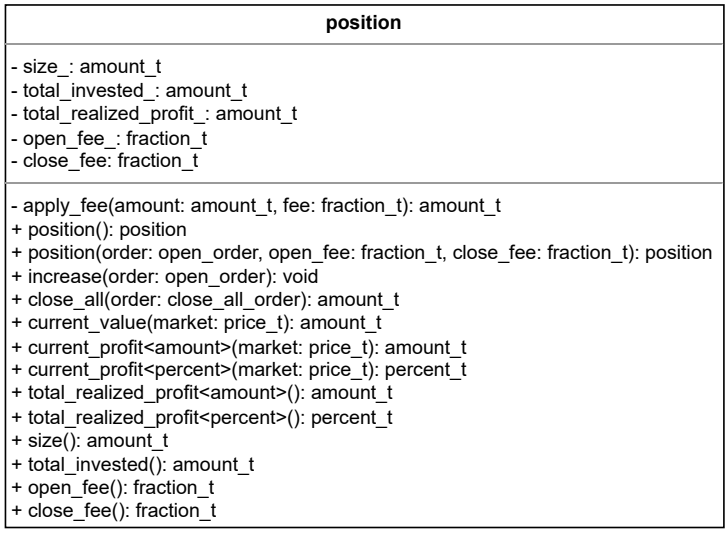
Obrázek 6.4 Rozhraní třídy `market`

6.3 Pozice

Pozice je implementována třídou `position`. Lze ji otevřít vytvořením instance třídy, zvýšit voláním metody `increase` nebo uzavřít pomocí metody `close`. Celkové množství nakoupeného aktiva lze zjistit voláním metody `size`. Dále lze získat aktuální hodnotu a aktuální zisk. Vzhledem k tomu, že je v průběhu ukládán celkový realizovaný zisk a celková investovaná částka, tak je lze zjistit i po uzavření pozice. Při změně velikosti pozice jsou uplatňovány poplatky. Výše otevíracích a uzavíracích poplatků jsou poskytnuty při inicializaci.

6.4 Simulátor

Obchodování lze simulovat pomocí třídy `simulator`, která uchovává a připravuje historická data pro simulaci a umožňuje její provedení. Hlavním účelem třídy je použít ji jako objektivní funkci během optimalizace. Obchodování lze pozorovat pomocí pozorovatelů. Tímto způsobem lze například použít třídu `bazooka::statistics::collector` ke sběru statistik nebo ke sběru časových řad třídu `chart_series::collector`, které mouhou být použity pro vykreslování grafů.



Obrázek 6.5 Rozhraní třídy `position`

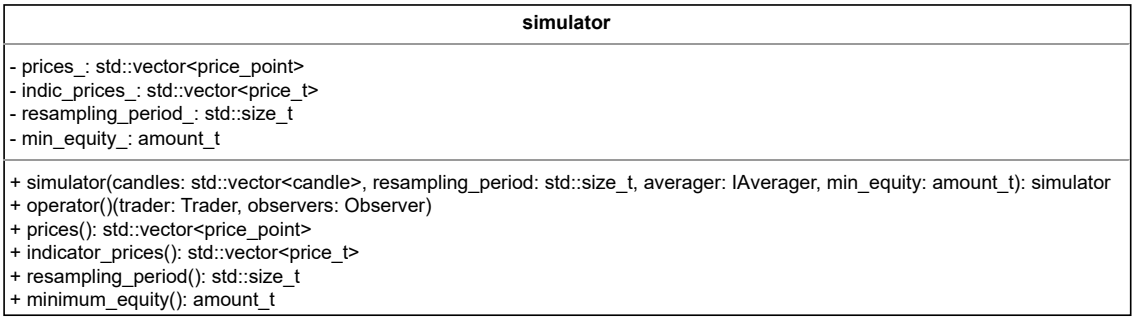
Při inicializaci jsou předány simulátoru historické ceny v podobě svíček. Pro rozhodování se používají uzavírací ceny. Ceny používané pro aktualizaci indikátorů jsou získány tak, že jsou svíčky nejprve převzorkovány pomocí `resampler` na požadované časové období a zprůměrovány pomocí zvolené průměrovací metody. Pokud je například doba převzorkování 30 a průměrovací metoda *ohlc*4, tak jsou minutové svíčky nejprve převzorkovány na 30minutové svíčky a poté zprůměrovány pomocí `candle::ohlc4`.

Proces převzorkování probíhá tak, že za dané období se otevírací cena první svíčky stane otevírací cenou převzorkované svíčky a uzavírací cena poslední svíčky se stane její uzavírací cenou. Vysoká a nízká cena jsou nejvyšší a nejnižší ceny v daném období. Jelikož je převzorkování poměrně časově náročné, tak se provádí pouze jednou v konstruktoru, takže se neopakuje při každém volání simulátoru.

Simulátor při volání přebírá instanci obchodníka a seznam pozorovatelů 6.3. Iteruje přes rozhodovací ceny až do konce nebo dosažení minimální čisté hodnoty. Pozorovatelé jsou během optimalizace informováni o událostech, jako je otevření pozice nebo aktualizace ukazatelů. Ceny použité pro aktualizaci ukazatelů jsou procházeny pomocí iterátoru.

Výpis kódu 6.3 Implementace simulace obchodování

```
template<class Trader, class... Observer>
void operator()(Trader&& trader, Observer& ... observers)
{
    auto indic_prices_it = indic_prices_.begin();
    (observers.started(trader, prices_.front()), ...);
    for (std::size_t i{0}; i<prices_.size() &&
        trader.equity(prices_[i].data)>min_equity_; i++) {
        (observers.decided(trader, trader(prices_[i]), prices_[i]), ...);

        if (trader.position_active())
            (observers.position_active(trader, prices_[i]), ...);

        if (i && (i+1)%resampling_period_==0)
            if (trader.update_indicators((*indic_prices_it++)))
                (observers.indicators_updated(trader, prices_[i]), ...);
    }
    (observers.finished(trader, prices_.back()), ...); }
```



Obrázek 6.6 Rozhraní třídy `simulator`

6.5 Statistiky

Nezbytnou součástí optimalizace jsou statistiky, jelikož jsou použity k formulaci optimalizačního kritéria a omezujících podmínek. Statistiky, které jsou společné pro všechny obchodní strategie, jsou reprezentovány třídou `statistics` a jsou rozšířeny pro Bazooka obchodní strategii `bazooka::statistics`.

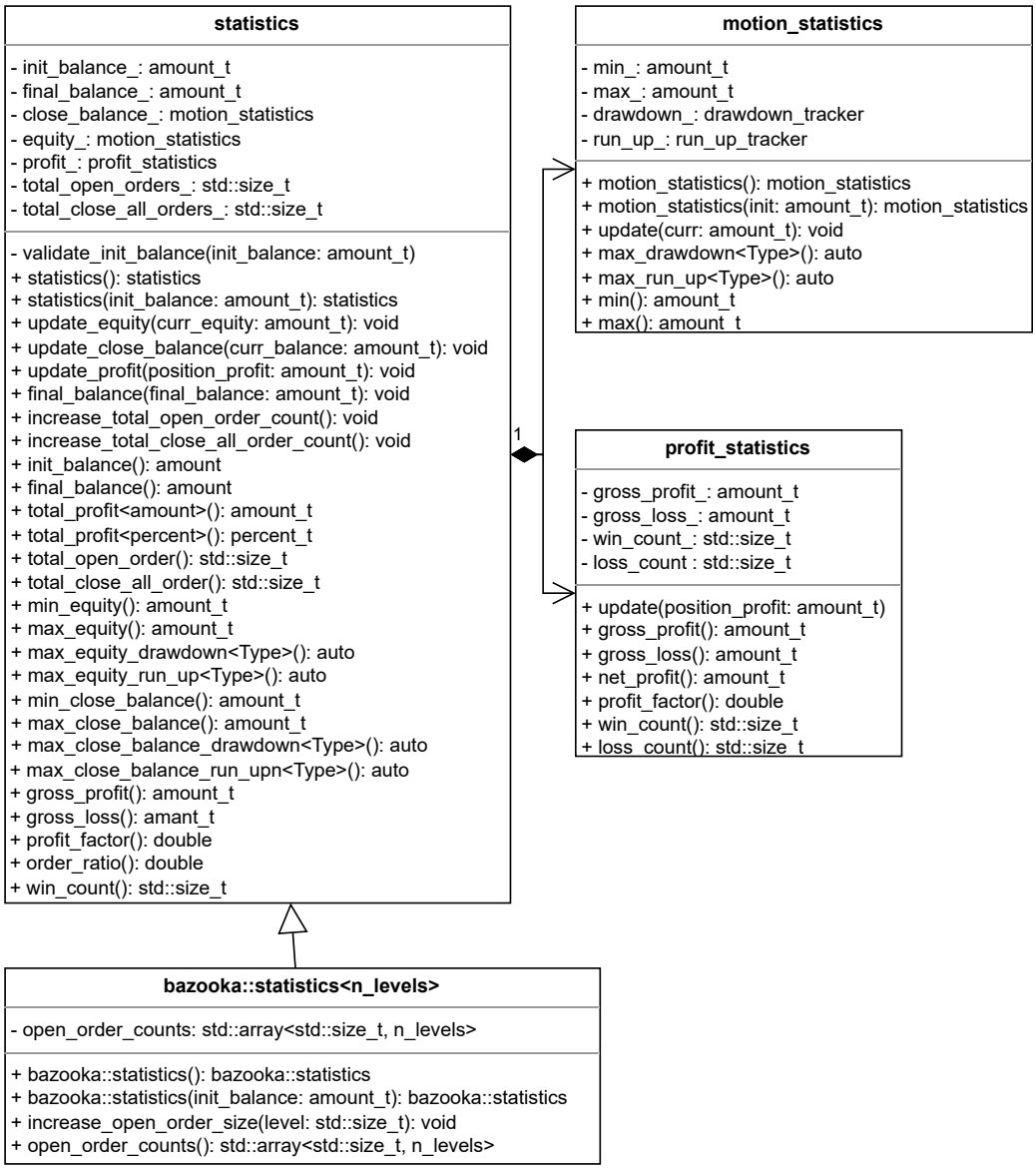
Instance třídy je inicializována předáním počátečního zůstatku. Statistiky jsou aktualizovány pomocí metod pro aktualizaci `update_equity()` nebo setrů `final_balance` (viz 6.7). K uloženým statistikám lze přistovat pomocí getrů. Některé hodnoty, jako je celkový zisk, lze získat buď jako částku nebo v procentech. Pro upřesnění jednotky a se používá značka `percent` nebo `amount` 6.4 při volání metody.

Výpis kódu 6.4 Metody pro získání celkového zisku v různých jednotkách

```
template<class T>
requires std::same_as<T, amount>
amount_t total_profit() const
{
    return final_balance_-init_balance_;
}

template<class T>
requires std::same_as<T, percent>
percent_t total_profit() const
{
    return ((final_balance_-init_balance_)/init_balance_)*100;
}
```

Všechny statistiky související se ziskem jsou sledovány pomocí instance `profit_statistics`, které lze použít k získání hrubého zisku, hrubé ztráty nebo faktoru zisku. Dále jsou sledovány statistiky související s pohybem pomocí `motion_statistics`, které poskytují minimální a maximální hodnotu, maximální vzestupný a sestupný pohyb. Vzestupný pohyb je sledován pomocí `run_up_tracker` a sestupný pomocí `drawdown_tracker`. Tímto způsobem je sledována čistá hodnota a zůstatek po uzavření pozice.

Do třídy `bazooka::statistics` byl přidán čítač otevíracích objednávek pro jednotlivé nákupní úrovně. Nese informace o tom, kolikrát bylo dosaženo jednotlivých úrovní nákupu, a může tak pomoci lépe porozumět tomu, jak se strategie chová.



Obrázek 6.7 Rozhraní třídy typu statistika

Kapitola 7

Generátory stavového prostoru

Pro optimalizaci byly vybrány čtyři konfigurační proměnné: typ a perioda ukazatele, úrovně a velikosti nákupů. Hodnoty proměnných lze generovat buď systematicky nebo náhodně. Důležitým aspektem, který bylo nutné dodržet, bylo, že obě verze generátorů musí generovat hodnoty ze stejného vyhledávacího prostoru, jinak by byly výsledky jednotlivých optimalizačních metod nesrovnatelné. Generátory sdílejí společné rozhraní. Jakmile jsou vytvořeny, lze je použít voláním volacího operátoru třídy.

Náhodné generátory mají volací operátor přetížen. Specializace přebírá argument představující počáteční hodnotu, která je používaná při generování následující hodnoty. Generátory vracejí přímo datové typy představující konfigurační proměnné. V ideálním případě by měly generovat stejné hodnoty se stejnou pravděpodobností. Na druhou stranu systematické generátory vracejí stejné hodnoty právě jednou v podobě korutiny, která je buď implementována třídou `cppcoro::generator` nebo `cppcoro::recursive_generator`.

7.1 Typ ukazatele

Typ ukazatele je reprezentován enumerační třídou `indicator_tag` se dvěma možnými hodnotami `indicator_tag::ema` a `indicator_tag::sma`. Jediný způsob, jak parametrizovat tuto proměnnou, je použít buď jednu z hodnot nebo obě. Hodnoty jsou stejného typu, takže je lze uložit do pole. Jelikož je jeho velikost malá a maximální velikost je známa v době kompilace, tak lze použít `etl::vector` k uložení hodnot pro generování. Hodnoty lze náhodně rovnoměrně generovat pomocí instance třídy `std::uniform_int_distribution`.

7.2 Perioda ukazatele

SMA i EMA mají parametr perioda $p > 1$. Oba generátory generují hodnoty v intervalu $[f, t]$ s krokem s, kde $((f > t \implies ((f - t) \bmod s) = 0 \wedge s < 0) \wedge (t < f \implies ((t - f) \bmod s) = 0 \wedge s > 0)) \wedge s \neq 0 \wedge t \neq f$. Počet možných výstupních hodnot je roven $n = \frac{|t-f|}{s} + 1$. Pro systematické generování period lze použít `systematic::int_range_generator` a náhodné `random::int_range_generator`.

7.2.1 Náhodné generování

Při generování bez počáteční periody je nejprve pomocí `std::uniform_int_distribution` vygenerována hodnota $v \in [\frac{p_{min}}{|s|}, = \frac{p_{max}}{|s|}]$, kde $p_{min} = min(f, t) \wedge p_{max} = max(f, t)$, která je následně

přeškálována na výstupní periodu $p_o = v \cdot s \wedge p_o \in [p_{min}, p_{max}]$ (viz 7.1).

Výpis kódu 7.1 Metoda pro náhodné generování periody indikátoru

```
int operator()()
{
    using param_t = std::uniform_int_distribution<int>::param_type;
    int positive_step = std::abs(step_);
    distrib_.param(param_t{min_/positive_step, max_/positive_step});
    int val = distrib_(gen_)*positive_step;
    assert(val>=min_ && val<=max_);
    return val;
}
```

Při generování periody z počátku o používá generátor navíc parametr rozpětí změny $k \in [1, \frac{n}{2}]$. Například, když $f = 1$, $t = 5$, $s = 1$, $o = 3$ a $k = 1$, pak jsou všechny možné výstupní hodnoty periody $p_o \in [o - (k \cdot s), o + (k \cdot s)] = [2, 4]$. Na okrajích intervalu, kdy mohou být hodnoty generovány ze dvou podintervalů. Například, když $f = 1$, $t = 5$, $s = 1$ a $k = 1$, pak jsou všechny možné výstupní periody $p_o \in [1, 2] \cup [5, 5]$. Jelikož jsou periody generovány pomocí datového typu integer, který může nabývat i záporných hodnot, lze nejprve vygenerovat hodnotu $w \in [o - (k \cdot s), o + (k \cdot s)]$ a výslednou periodu získat $(w < p_{min} \implies (p_o = w + (n * |s|))) \wedge (w > p_{max} \implies (p_o = w - (n * |s|))) \wedge (w \in [p_{min}, p_{max}] \implies p_o = w)$ (viz 7.2).

Výpis kódu 7.2 Metoda pro náhodné generování periody z počátku

```
int operator()(int origin)
{
    using param_t = std::uniform_int_distribution<int>::param_type;
    int positive_step = std::abs(step_);
    int curr_min = origin-change_span_*positive_step;
    int curr_max = origin+change_span_*positive_step;
    distrib_.param(param_t{curr_min/positive_step,curr_max/positive_step});
    int val = distrib_(gen_)*positive_step;
    assert(val>=curr_min && val<=curr_max);

    if (val<min_) val += n_vals_*positive_step;
    if (val>max_) val -= n_vals_*positive_step;
    assert(val>=min_ && val<=max_);
    return val;
}
```

7.3 Úrovně nákupu

Pro úrovně nákupu platí, že $L = \{l_0, l_1, \ldots, l_n\} \in \mathbb{Q} \cap (0, 1) \wedge n > 0 \wedge [\forall i \in \{0, \ldots, n-1\} : l_i > l_{i+1}]$. Příkladem takové sekvence racionálních čísel může být $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\}$. Generátory jsou parametrizovány pomocí parametru $u \geq n$, který označuje počet jedinečných zlomků, které se mohou v sekvenci objevit. Zlomky, které tvoří výslednou sekvenci, jsou brány z množiny $\{\frac{u}{d}, \frac{u-1}{d}, \ldots, \frac{1}{d}\} \in (0, 1) \wedge d = u + 1$. Pro $n = 3$, $u = 4$ jsou všechny možné platné sekvence: $\{\frac{4}{5}, \frac{3}{5}, \frac{2}{5}\}, \{\frac{4}{5}, \frac{3}{5}, \frac{1}{5}\}, \{\frac{4}{5}, \frac{2}{5}, \frac{1}{5}\}, \{\frac{3}{5}, \frac{2}{5}, \frac{1}{5}\}$.

Později ve vývoji během experimentální fáze byla přidána spodní hranice $lb \in [0, 1)$, takže pro nákupní úrovně navíc platí, že $[\forall i \in \{0, \ldots, n-1\} : l_i > lb]$. Nákupní úrovně byly vygenerovány výše popsaným způsobem, a poté přeškálovány pomocí upraveného vzorce pro lineární škálování $l_s = l \cdot (1 - lb) + lb$, kde l_s je přeškálovaná hodnota a l je vstupní hodnota. Pro $n = u = 2$ existuje pouze 1 možná platná sekvence $\{\frac{2}{3}, \frac{1}{3}\}$ a při použití $lb = \frac{1}{2}$ byla by přeškálována na $\{\frac{5}{6}, \frac{4}{6}\}$.

7.3.1 Systematické generování

Sekvence jsou generovány rekurzivně (viz 7.3), protože se počet smyček `for` liší v závislosti na velikosti sekvence n. Smyčky jsou na sobě závislé, následující (hlubší) smyčka musí znát předchozí hodnotu čitatele. Nejvnitřnější smyčka vrací výslednou sekvenci.

Výpis kódu 7.3 Metody pro systematické generování úrovní nákupu

```
cppcoro::recursive_generator<value_type> operator()()
{
    co_yield generate<0>(this->unscaled_denom());
}

template<std::size_t depth = n_levels>
requires (depth==n_levels)
cppcoro::recursive_generator<value_type> generate(std::size_t)
{
    co_yield this->levels_;
}

template<std::size_t depth = 0>
requires (depth<n_levels)
cppcoro::recursive_generator<value_type> generate(std::size_t prev_num)
{
    for (std::size_t num{--prev_num}; num>n_levels-depth-1; num--) {
        this->levels_[depth] = this->rescale(num);
        co_yield generate<depth+1>(num);
    }
}
```

7.3.2 Náhodné generování

Instance třídy `random::levels_generator` má mezi svými atributy pole se všemi možnými hodnotami výstupní sekvence. Při generování náhodné sekvence bez počátku nejprve tyto hodnoty zamíchá, seřadí prvních n prvků v sestupném pořadí a nakonec je zkopíruje do výstupního pole (viz 7.4).

Výpis kódu 7.4 Metoda pro náhodné generování úrovní nákupu bez počátku

```
const value_type& operator()()
{
    std::shuffle(options_.begin(), options_.end(), gen_);
    std::sort(options_.begin(),options_.begin()+n_levels,
        std::greater<>());
    std::copy(options_.begin(),options_.begin()+n_levels,
        levels_.begin());
    return levels_;
}
```

Náhodné generování na základě počáteční sekvence je složitější. Používá parametr $k \in [1, n]$, který určuje počet nákupních úrovní, které se mají změnit. Je předán instanci generátoru při její konstrukci. Při generování je nejprve náhodně vybráno $n - k$ prvků, které budou zachovány z původní sekvence, a jsou uloženy do instance `etl::flat_set`, což je kontejner pevné velikosti, který ukládá jedinečné prvky v určeném pořadí. Dále je zamícháno pole se všemi možnými hodnotami. Ve smyčce `while` jsou hodnoty vkládány do setu až do jeho naplnění. Nakonec jsou hodnoty překopírovány do výstupního pole (viz 7.5).

Výpis kódu 7.5 Metoda pro náhodné generování úrovní nákupu z počátku

```
const value_type& operator()(const value_type& origin){
    std::size_t keep_count{n_levels-change_count_};
    std::shuffle(indices_.begin(), indices_.end(), gen_);
    std::sort(indices_.begin(), indices_.begin()+keep_count);
    etl::flat_set<fraction_t, n_levels, std::greater<>> unique;

    for (std::size_t i{0}; i<keep_count; i++)
        unique.emplace(origin[indices_[i]]);

    std::shuffle(options_.begin(), options_.end(), gen_);
    auto options_it = options_.begin();
    while (!unique.full()) unique.insert(*options_it++);
    std::copy(unique.begin(), unique.end(), levels_.begin());
    return levels_;
}
```

7.4 Velikosti nákupu

Pro velikosti nákupu platí, že $S = \{s_0, s_1, \ldots, s_n\} \in \mathbb{Q} \cap (0, 1] \wedge n > 1 \wedge \sum_{i=0}^{n} s_i = 1$. Příkladem takové sekvence může být $\{\frac{1}{4}, \frac{2}{4}, \frac{1}{4}\}$. Generátory jsou parametrizovány pomocí parametru $u \geq 1$, který označuje počet jedinečných zlomků, které se mohou v sekvenci objevit. Zlomky, které tvoří výslednou sekvenci, jsou brány z množiny $\{\frac{u}{d}, \frac{u-1}{d}, \ldots, \frac{1}{d}\} \in [0, 1] \wedge d = n+u-1$. Pro $n = 3$, $u = 3$ jsou všechny možné platné sekvence: $\{\frac{1}{5}, \frac{1}{5}, \frac{3}{5}\}, \{\frac{1}{5}, \frac{2}{5}, \frac{2}{5}\}, \{\frac{1}{5}, \frac{3}{5}, \frac{1}{5}\}, \{\frac{2}{5}, \frac{1}{5}, \frac{2}{5}\}, \{\frac{2}{5}, \frac{2}{5}, \frac{1}{5}\}, \{\frac{3}{5}, \frac{1}{5}, \frac{1}{5}\}$.

7.4.1 Systematické generování

Velikosti nákupů jsou systematicky generovány pomocí třídy `systematic::sizes_generator` podobně jako úrovně nákupů (viz 7.6). Opět je použita rekurze, každá smyčka `for` předává součet zbývajících čitatelů následující smyčce. Maximální současný čitatel má buď hodnotu maximálního čitatele nebo součet zbývajících čitatelů snížený o jedna.

Výpis kódu 7.6 Metody pro systematické generování velikostí nákupu

```
cppcoro::recursive_generator<value_type> operator()()
{
    co_yield generate<0>(this->denom_);
}

template<std::size_t depth = n_sizes>
requires (depth+1==n_sizes)
cppcoro::recursive_generator<value_type> generate(std::size_t remaining)
{
    this->sizes_[depth] = fraction_t{remaining, this->denom_};
    co_yield this->sizes_;
}

template<std::size_t depth = 0>
requires (depth+1<n_sizes)
cppcoro::recursive_generator<value_type> generate(std::size_t remaining)
{
    std::size_t curr_max = (remaining>this->max_num_) ?
            this->max_num_ : remaining-1;
    for (std::size_t num{1}; num<=curr_max; num++) {
```

```
        this->sizes_[depth] = fraction_t{num, this->denom_};
        co_yield generate<depth+1>(remaining-num);
    }
}
```

7.4.2 Náhodné generování

Generování náhodných velikostí nákupu je implementováno třídou `random::sizes_generator`. Bez počátku se provádí nejprve promícháním indexů výstupní sekvence (viz 7.7). Následně je zavolána metoda `fill_rest`, která doplní chybějící velikosti nákupu, v tomto případě všechny. Ve smyčce se nejprve vygeneruje velikost nákupu z intervalu od minimální po aktuální maximální možnou velikost, a poté jsou přiřazeny na příslušný index. Následuje snížení celkové zbývající velikosti nákupu a aktualizace aktuální maximální velikosti nákupu. Cyklus končí, když jsou přiřazeny všechny velikosti nákupu kromě poslední, které je přiřazen zbytek.

Výpis kódu 7.7 Metody pro náhodné generování velikostí nákupu bez počátku

```
void fill_rest(std::size_t rest_num_sum, std::size_t curr_max_num,
    std::size_t first_index = 0)
{
    using param_t=std::uniform_int_distribution<std::size_t>::param_type;
    std::size_t num;

    for (std::size_t i{first_index}; i<n_sizes-1; i++) {
        distrib_.param(param_t{min_num_, curr_max_num});
        num = distrib_(gen_);
        sizes_[indices_[i]] = fraction_t{num, this->denom_};
        rest_num_sum -= num;
        curr_max_num = (curr_max_num==num) ? min_num_:curr_max_num-num+1;
    }
    sizes_[indices_.back()] = fraction_t{rest_num_sum, denom_};
}

const value_type& operator()()
{
    std::shuffle(indices_.begin(), indices_.end(), gen_);
    fill_rest(denom_, max_num_);
    return sizes_;
}
```

Generování nákupních velikostí z počátku začíná stejně zamícháním indexů sekvence (viz 7.8). Používá však také parametr $k \in [0, n]$, který určuje počet úrovní nákupu, které se mají změnit. Nejprve se tedy výstupní sekvenci přiřadí $n - k$ nákupních velikostí z počátku. Celková velikost nákupu zbývajících úrovní se odpovídajícím způsobem sníží. Zbytek kódu je podobný jako u předchozí ukázky.

Výpis kódu 7.8 Metody pro náhodné generování velikostí nákupu z počátku

```
const value_type& operator()(const value_type& origin)
{
    std::size_t i, keep_count{n_sizes-change_count_},rest_num_sum{denom_};
    std::shuffle(indices_.begin(), indices_.end(), gen_);

    for (i = 0; i<keep_count; i++) {
        auto idx = indices_[i];
        auto size = origin[idx];
        sizes_[idx] = size;
```

```
            rest_num_sum -= size.numerator();
        }
        std::size_t curr_max_num{rest_num_sum-change_count_+1};
        fill_rest(rest_num_sum, curr_max_num, i);
        return sizes_;
    }
```

Kapitola 8
Optimalizátory

Optimalizátory byly implementovány co nejobecněji, aby je bylo možné použít i pro řešení jiných optimalizačních problémů. Zobecnění bylo dosaženo pomocí konceptů, které kontrolují rozhraní předávaných argumentů. Samotné optimalizátory mají pouze několik atributů, které by měly být společné pro všechny možné problémy. Další části algoritmu jsou dodány při samotném volání instance třídy.

Pro sledování průběhu heuristických optimalizací se používají pozorovatelé, které jsou upozorňováni na události, ke kterým při optimalizaci dochází. Průběh optimalizace mohou hlásit nebo ukládat. Příkladem události, která může nastat během optimalizace simulovaného ochlazování, je snížení teploty nebo přijetí lepšího stavu. Pozorovatelé jsou předáni jako poslední parametr při volání optimalizátoru a jsou uloženi jako sada parametrů šablony. Tímto způsobem lze přijmout víc než jeden pozorovatel nebo žádný.

Všechny optimalizátory používají objektivní funkci, která nejprve přijme konfiguraci, spustí historickou simulaci, během níž se shromažďují výkonnostní statistiky, vypočítá se hodnota optimalizačního kritéria a je vrácen stav (viz 8.1). Omezení kontroluje samotný optimalizátor a nejlepší stavy se shromažďují pomocí třídy s výsledky, která je porovnává na základě uložené hodnoty optimalizačního kritéria.

Výpis kódu 8.1 Implementace objektivní funkce

```
auto objective = [&](const config_t& curr) {
    bazooka::statistics<n_levels>::collector collector{};
    simulator(create_trader(curr), collector);
    auto stats = collector.get();
    return state_t{{curr, optim_criterion(stats)}, stats};
};
```

8.1 Hrubá síla

Algoritmus je implementován třídou `brute_force::parallel::optimizer`. Prochází všechny stavy prohledávacího prostoru a zajišťuje nalezení optimálního řešení. Je vhodný pro relativně malé vyhledávací prostory a může pomoci lépe porozumět vlastnostem objektivní funkce.

Vyhledávací prostor je generován pomocí systematických generátorů, které jsou volány ve vnořených smyčkách `for` (viz 8.2). Nejvnitřnější cyklus vrací konfiguraci jako korutinu typu `cppcoro::generator`. Vytvoření vyhledávacího prostoru tímto způsobem je jednodušší než navrhování vlastního iterátoru, jelikož na sobě mohou být jednotlivého hodnoty závislé.

Výpis kódu 8.2 Generování systematického stavového prostoru

```
auto search_space = [&]() -> cppcoro::generator<config_t> {
    for (std::size_t period: periods_gen())
        for (const auto& tag: tags)
            for (const auto& levels: levels_gen())
                for (const auto& sizes: sizes_gen())
                    co_yield config_t{tag, period, levels, sizes};};
```

Vzhledem k tomu, že jsou jednotlivé stavy na sobě nezávislé, může být vyhledávání paralyzováno. Paralelizace byla implementována pomocí direktiv z knihovny *OpenMP*. Jelikož je vyhledávací prostor předáván jako korutina, lze jej paralelizovat pouze pomocí taskového paralelismu použitím direktivy `task`.

Implementaci paralelizace lze vidět ve výpise 8.3. Nejprve je použita direktiva `parallel` k vytvoření paralelního regionu, který vytváří, provádí a ukončuje jednotlivá vlákna. Následně je pomocí direktivy `single` proveden cyklus `for` právě jedním vláknem. Uvnitř cyklu jsou vytvářeny jednotlivé úlohy pomocí direktivy `task` a prováděny paralelně jednotlivými vlákny. Každé vlákno nejprve zavolá objektivní funkci k získání stavu. Pokud stav splňuje omezující podmínky, je v kritické sekci aktualizován výsledek. Je nutné použít synchronizaci, protože může být výsledek modifikován více vlákny najednou.

Výpis kódu 8.3 Optimalizace pomocí algoritmu hrubé síly

```
void operator()(IResult<state_t> auto& result,
    IConstraints<state_t> auto&& constraints,
    IObjectiveFunction<state_t> auto&& objective,
    ISearchSpace<config_t> auto&& search_space) const
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (const config_t& config: search_space()) {
                #pragma omp task
                {
                    auto state = objective(config);
                    if (constraints(state)) {
                        #pragma omp critical
                        result.update(state);
                    }}}}}}
```

8.2 Simulované ochlazování

Algoritmus je implementován třídou `simulated_annealing::optimizer`. Třída má pouze několik členských atributů: současnou, počáteční a minimální teplotu, počet iterací, současný a nejlepší stav. Jiné části algoritmu jako chlazení, ohodnocení horšího stavu a nalezení sousedního stavu jsou specifické pro řešený problém, proto jsou předány jako argumenty při volání instance objektu.

Bylo implementováno několik rozvrhů chlazení vycházejících ze srovnávajícího článku [8]. Všechny jsou umístěny v souboru `simulated_annealing/cooler.hpp`. Většina je parametrizována parametrem `decay_`, který se používá k řízení rychlosti snižování teploty. Správnost jeho hodnoty závisí na algoritmu chlazení a ověřuje se při inicializaci konkrétního objektu. Chladiče používají návrhový vzor *Strategy*, takže když jsou zavolány, vezmou instanci třídy optimalizátoru, přistoupí k hodnotám atributů potřebným k výpočtu další teploty, vypočítají ji a aktualizují. Příklad implementace chladiče lze vidět na 8.4.

Výpis kódu 8.4 Ukázka implementace volacího operátoru třídy `exp_mul_cooler`

```
void operator()(SimulatedAnnealing& optimizer)
{
    optimizer.current_temperature(optimizer.start_temperature()*
        std::pow(this->decay_, optimizer.it()));
}
```

Algoritmus pro nalezení náhodného souseda je implementován třídou `bazooka::neighbor`. Ke generování sousední konfigurace využívá náhodné generátory, které na základě aktuální hodnoty generují následující hodnotu. Náhodně je vždy změněn pouze jeden konfigurační parametr.

Pro určení počtu prohledávaných sousedů byly implementovány dva algoritmy. Jednodušší z nich `simulated_annealing::fixed_equilibrium` vždy vrací stejný počet sousedů. Během experimentální fáze byl přidán `simulated_annealing::temperature_base_equilibrium`, který vypočítává aktuální počet iterací pomocí $N = m \cdot T_{curr}$, kde $m \geq 0$ je multiplikátor a T_{curr} je současná teplota.

Implementaci algoritmu lze vidět ve výpise 8.5. Během optimalizace dochází k několika událostem, které jsou hlášeny pozorovatelům. Pro sběr dat o průběhu optimalizace byla implementována třída `simulated_annealing::progress_collector`. Za účelem logování byla vytvořena třída `simulated_annealing::progress_reporter`.

Výpis kódu 8.5 Implementace simulovaného ochlazování

```
void operator()(const config_t& init_config,
    IResult<state_t> auto& result,
    IConstraints<state_t> auto&& constraints,
    IObjectiveFunction<state_t> auto&& objective,
    ICooler<optimizer> auto&& cool,
    INeighbor<config_t> auto&& neighbor,
    IAppraiser<state_t> auto&& appraise,
    IEquilibrium<optimizer> auto&& equilibrium,
    IObserver<optimizer> auto& ... observers)
{
    curr_state_ = best_state_ = objective(init_config);
    (observers.started(*this), ...);

    for (it_ = 0; curr_temp_>min_temp_; it_++) {
        for (std::size_t e{0}; e<equilibrium(*this); e++) {
            auto candidate = objective(neighbor(curr_state_.config));

            if (result.compare(candidate, curr_state_)) {
                curr_state_ = candidate;
                (observers.better_accepted(*this), ...);

                if (constraints(curr_state_))
                    if (result.compare(curr_state_, best_state_)) {
                        best_state_ = curr_state_;
                        result.update(best_state_);
                    }
            }
            else {
                double diff = appraise(curr_state_, candidate);
                double threshold = std::exp(-diff/curr_temp_);

                if (rand_prob_()<threshold) {
                    curr_state_ = candidate;
                    (observers.worse_accepted(*this), ...);
                }
```

```
            }
        }
        cool(*this);
        (observers.cooled(*this), ...);
    }
    (observers.finished(*this), ...);
}
```

8.3 Genetický algoritmus

Genetický algoritmus je implementován třídou `genetic_algorithm::optimizer`. Má pouze dva třídní atributy počet iterací a populaci. Další části algoritmu jsou poskytovány při jeho volání (viz 8.7). Populace je uložena pomocí třídy `std::vector`, která spravuje dynamicky alokované pole prvků. Počet dynamických realokací byl minimalizován pomocí metod `clear` a `reserve`, jinak by se optimalizace zbytečně zpomalovala. Data o průběhu optimalizace lze shromažďovat pomocí `progress_collector` a zaznamenávat pomocí `progress_reporter`.

Pro mutaci lze použít instanci třídy `bazooka::neighbor`. Algoritmus pro křížení je implementován třídou `bazooka::crossover`. Při křížení se používají všechny konfigurační proměnné, a proto je pro každou z nich implementován operátor křížení. O tom, zda potomek obdrží periodu ukazatele nebo typ ukazatele od své matky nebo otce, se rozhoduje hodem mincí. Pro křížení úrovní a velikostí nákupů byly vytvořeny vlastní třídy.

Křížení úrovní nákupu je implementováno třídou `bazooka::levels_crossover` (viz 8.6). Funguje tak, že genetickou informaci od otce a matky vloží do společného pole typu `etl::vector`, které na rozdíl od `std::vector` používá statickou alokaci místo dynamické. Geny jsou náhodně zamíchány a následně vkládány ve `while` cyklu do instance `elt::flat_set` do jejího naplnění. Hodnoty jsou nakonec překopírovány do výstupního pole. Algoritmus má několik vlastností, jednou z nich je, že pokud jsou rodiče stejní, bude i potomek stejný. Pokud mají rodiče společný gen, je větší šance, že bude vybrán.

Výpis kódu 8.6 Implementace křížení nákupních úrovní

```
value_type operator()(const value_type& mother, const value_type& father)
{
    etl::vector<fraction_t,n_levels*2> genes{mother.begin(),mother.end()};
    genes.insert(genes.end(), father.begin(), father.end());

    std::shuffle(genes.begin(), genes.end(), gen_);
    auto genes_it = genes.begin();

    etl::set<fraction_t, n_levels, std::greater<>> child_genes;
    while (!child_genes.full()) child_genes.insert(*genes_it++);

    std::array<fraction_t, n_levels> child;
    std::copy(child_genes.begin(), child_genes.end(), child.begin());
    return child;
}
```

Křížení velikostí nákupů je implementováno třídou `bazooka::sizes_crossover`. Nejmenší hodnoty rodičů jsou použity jako základ pro potomka. Dále jsou nákupní velikosti potomka náhodně zvyšovány v cyklu `while` tak, aby velikost nákupu nikdy nebyla větší než nejvyšší hodnota rodiče. Smyčka končí, jakmile je součet všech velikostí roven jedné. Výsledné hodnoty potomků se tedy budou vždy pohybovat v rozmezí hodnot rodičů. Pokud rodiče nejsou stejní, potomek je jiný, ale stále je podobný svým rodičům.

Ruletový výběr byl implementován třídou `genetic_algorithm::roulette_selection`, která interně používá instanci třídy `std::discrete_distribution` pro výběr rodičů. Zdatnost každého

jedince představuje váhu. Čím vyšší je váha, tím pravděpodobněji bude daný jedinec vybrán jako rodič.

Pro náhodné párování rodičů byl implementován algoritmus `random_matchmaker`. Rodiče jsou nejprve náhodně zamícháni a následně párováni. Celkový počet rodičů nemusí být dělitelný počtem vybíraných n-tic, takže někteří rodiče mohou být použiti vícekrát. N-tice jsou vraceny jako korutina typu `cppcoro::generator`.

Byly implementovány dva přístupy nahrazování populace. Třída `en_block_replacement` zachovává pouze děti k vytvoření nové generace. Třída `elitism_replacement` k dětem ještě přidává $er \cdot n$ nejzdatnějších rodičů, kde $er \in [0, 1]$ je poměr elitních rodičů a n jejich celkový počet.

Výpis kódu 8.7 Implementace genetického algoritmu

```
template<IResult<state_t> Result>
void operator()(const std::vector<config_t>& init_genes,
        Result& result,
        IConstraints<state_t> auto&& constraints,
        IObjectiveFunction<state_t> auto&& fitness,
        IPopulationSizer auto&& size,
        ISelection<state_t> auto&& select,
        IMatchmaker<state_t> auto&& match,
        ICrossover<config_t> auto&& crossover,
        IMutation<config_t> auto&& mutate,
        IReplacement<state_t, typename Result::comparator_type>
            auto&& replace,
        ITerminationCriteria<optimizer> auto&& terminate,
        IObserver<optimizer> auto& ... observers)
{
    population_.clear(), population_.reserve(init_genes.size());
    for (const auto& genes: init_genes)
        population_.template emplace_back(fitness(genes));

    std::vector<state_t> parents, children;
    (observers.started(*this), ...);
    for (; population_.size() && !terminate(*this); it_++) {
        parents.clear();
        select(size(population_.size()), population_, parents);

        // mate
        children.clear();
        for (const auto& mates: match(parents))
            for (auto&& genes: crossover(mates))
                children.emplace_back(fitness(mutate(std::move(genes))));

        population_.clear();
        replace(parents, children, result.comparator(), population_);

        // update results
        for (const auto& individual: population_)
            if (constraints(individual))
                result.update(individual);

        (observers.population_updated(*this), ...);
    };
    (observers.finished(*this), ...); }
```

8.4 Tabu prohledávání

Tabu prohledávání je implementováno třídou `tabu_search::optimizer`, která má čtyři třídní atributy: počet iterací, seznam tabu, aktuální a nejlepší stav. Pro ukládání zakázaných tahů byla implementována třída `bazooka::configuration_memory`. Ukládá pouze konfigurační atributy, které byly změněny, nikoli celé konfigurace. Perioda a typ ukazatele jsou zapamatovány tak, jak jsou. Hodnoty úrovně a velikosti nákupu, které nebyly změněny, jsou vynulovány před uložením do paměti. K ukládání úrovní a velikostí nákupu se používá `std::map`.

Jelikož existují čtyři konfigurační proměnné, z nichž se vždy mění pouze jedna, tak se tah ukládá do instance třídy `std::variant`, která je zapouzdřena pomocí třídy `bazooka::movement`. Vzhledem k tomu, že úrovně a velikosti nákupu jsou reprezentovány stejným datovým typem, polem zlomků, je nutné implementovat způsob, jak je rozlišit. Třída proto obsahuje výčtovou třídu zvanou `indices`, ke které lze přistupovat zvenčí. Lze ji použít ve výrazech `switch` 8.8. Tah lze uložit a zpřístupnit pomocí setrů a getrů.

Výpis kódu 8.8 Použití třídy `bazooka::movement`

```
bool contains(const move_t& move) const
{
    switch (move.indices()) {
    case move_t::indices::tag:
        return indic_mem_.contains(move.tag);
    case move_t::indices::period:
        return period_mem_.contains(move.period());
    case move_t::indices::levels:
        return levels_mem_.contains(move.levels());
    default:
        return sizes_mem_.contains(move.open_sizes());
    }
}
```

Hodnoty se vkládají do paměti pomocí metody `remember` a zůstanou v ni po daný počet iterací určený tabu lhůtou, která je předána paměti při konstrukci. Počet iterací se sníží o jednu pomocí metody `forget` (viz 8.9). Sousední konfigurace lze vygenerovat pomocí `bazooka::neighbor`, která vrací sousední konfiguraci včetně tahu. Velikost sousedství udává počet sousedních konfigurací, které jsou prohledány během jedné iterace.

Výpis kódu 8.9 Implementace genetického algoritmu

```
void operator()(const config_t& init,
    IResult<state_t> auto& result,
    IConstraints<state_t> auto&& constraints,
    IObjectiveFunction<state_t> auto&& objective,
    INeighbor<config_t, move_t> auto&& neighbor,
    INeighborhoodSizer<optimizer> auto&& neighborhood,
    ITerminationCriteria<optimizer> auto&& terminate,
    IAspirationCriteria<state_t, optimizer> auto&& aspire,
    IObserver<optimizer> auto& ... observers)
{
    best_state_ = curr_state_ = objective(init);
    state_t candidate, origin;

    (observers.started(*this), ...);
    for (it_ = 0; !terminate(*this); it_++) {
        // explore neighborhood
        origin = curr_state_;
        std::tie(curr_state_.config, curr_move_)=neighbor(origin.config);
        curr_state_ = objective(curr_state_.config);
```

```
        for (std::size_t i{0}; i<neighborhood(*this)-1; i++) {
            std::tie(candidate.config,candidate_move_) =
                neighbor(origin.config);
            candidate = objective(candidate.config);

            if ((!tabu_list_.contains(candidate_move_)
                && result.compare(candidate, curr_state_))
                || aspire(candidate, *this)) {
                curr_state_ = candidate;
                curr_move_ = candidate_move_;
            }
        }

        // try update best state
        if (result.compare(curr_state_, best_state_)) {
            best_state_ = curr_state_;
            if (constraints(best_state_))
                result.update(best_state_);
        }

        tabu_list_.forget();
        tabu_list_.remember(curr_move_);
        (observers.iteration_passed(*this), ...);
    }
    (observers.finished(*this), ...); }
```

8.5 Shromažďování výsledků

Třída `enumeration_result` slouží ke shromažďování n nejlepších výsledků. Má dva šablonové parametry: typ výsledku a funkci pro jejich porovnávání. Interně používá datovou strukturu *halda*, jelikož má složitost $\mathcal{O}(logn)$ pro vložení nového prvku a odstranění prvního. K ukládání výsledků používá kontejner `std::vector`.

Hodnoty jsou aktualizovány voláním metody `update` (viz 8.10). K vytvoření haldy používá funkci `std::push_heap`. Jakmile je plná, použije po vložení nového prvku funkci `std::pop_back` k odstranění nejhoršího prvku. K získání výsledku lze zavolat metodu `get`, která používá funkci `std::sort_heap` k seřazení prvků v požadovaném pořadí.

Výpis kódu 8.10 Implementace metody `update` třídy `enumerative_result`

```
void update(const Type& candidate)
{
    if (best_.size()<best_.capacity()-1) {
        best_.emplace_back(candidate);
        std::push_heap(best_.begin(), best_.end(), this->comp_);
    }
    else if (best_.size()==best_.capacity()-1) {
        best_.emplace_back(candidate);
        pop_heap(best_.begin(), best_.end(), this->comp_);
    }
    else {
        best_.back() = candidate;
        pop_heap(best_.begin(), best_.end(), this->comp_);
    }
}
```

Kapitola 9

Zpracování vstupů a výstupů

Byly vytvořeny vlastní třídy pro čtení a zápis do souborů ve *CSV* (Comma-Separated Values). Jsou užitečné při čtení historických svíčkových dat a při ukládání časových řad pro vykreslování grafů. Pro uložení nastavení a výsledků experimentu byl zvolen formát *JSON* (*JavaScript Object Notation*), který je standardizovaný a čitelný. Nebylo nutné vytvářet vlastní implementaci, protože existuje dobře známá, používaná a udržovaná knihovna `nlohmann/json` [32].

9.1 Formát CSV

Pro čtení a zápis do souborů CSV byly vytvořeny třídy `csv::reader` a `csv::writer`. Hodnoty jsou uloženy v tabulce, kde sloupce jsou hodnoty stejného typu odděleny oddělovačem, obvykle čárkou. První řádek může obsahovat názvy sloupců, ale nemusí.

Obě třídy mají jeden třídní šablonový udávající počet sloupců. Při inicializaci přebírají jako parametr cestu k souboru uloženému v instanci `std::filesystem::path` a oddělovač. Soubor je otevřen při konstrukci třídy, pokud jej nelze otevřít, je vyhozena výjimka `std::runtime_error`. Soubor je uzavřen při destrukci objektu.

Třída `csv::reader` nabízí dvě metody pro čtení řádků: `read_header` a `read_row`. Názvy sloupců jsou čteny do instance třídy `std::array`. Čtení přímo do proměnných je dosaženo použitím sady šablonových parametrů (viz 9.1). Třída vrací booleovský příznak s informací o tom, zda byl řádek přečten nebo ne. Hodnoty pak lze snadno číst v cyklu `while` (viz 9.2). Třída `io::parser` je používána k parsování jednotlivých hodnot.

Výpis kódu 9.1 Implementace čtení řádky třídy `io::csv::reader`

```
template<class ...Types>
bool read_row(Types& ...inputs)
{
    static_assert(sizeof...(Types)==n_cols);
    if (!std::getline(this->file_, line_)) return false;

    line_.erase(std::remove(line_.begin(), line_.end(), '\r'), line_.end());
    std::stringstream ss(line_);
    ss.exceptions(std::ios::failbit);

    auto read_line = [&](auto& in) { read_value(ss, in); };
    (read_line(inputs), ...);
    return true;
}
```

Třída `csv::reader` je používána velice podobně. Má metodu pro zápis sloupců `write_header` a pro zápis hodnot `write_row`. K převodu hodnot na textové řetězce používá třídu `stringifier`.

Výpis kódu 9.2 Ukázka čtení svíček pomocí `io::csv::reader`

```
io::csv::reader<5> reader{path, sep};
std::time_t opened;
price_t open, high, low, close;
std::vector<candle> candles;

while (reader.read_row(opened, open, high, low, close))
    candles.emplace_back(candle{opened, open, high, low, close});
```

9.2 Formát JSON

Samotný formát se skládá ze dvou struktur: kolekce párů klíč-hodnota a pole. Je založen na programovacím jazyce JavaScript, ale protože je postaven ze základních datových struktur, je implementován v mnoha programovacích jazycích [33].

Pro C++ je k dispozici několik JSON knihoven, ale zdá se, že `nlohmann/json` mezi nimi vyniká nejvíc. Na příslušné GitHub stránce je uvedeno, že jde o knihovnu JSON pro moderní C++, má 34 tisíc hvězdiček, je důkladně testována a pečlivě zdokumentována [32].

Podporuje vytvoření specifikace pro serializaci a deserializaci vlastních typů. Pro daný typ je vytvořena specializace struktury `adl_serializer`, u které mohou být implementovány metody `from_json` pro deserializaci a `to_json` pro serializaci. Implementaci pro `bazooka::indicator` lze vidět ve výpise 9.3.

Výpis kódu 9.3 Implementace deserializace pro vlastní datový typ

```
template<>
struct adl_serializer<trading::bazooka::indicator> {
    static void to_json(json& j, const bazooka::indicator& indic)
    {
        j = {{"period", indic.period()},
             {"name",   indic.name()}};
    }
};
```

Kapitola 10
Vizualizace

V současnosti Python nabízí jednu z nejrozšířenějších a nejrozvinutějších sad nástrojů pro vizualizaci a datovou vědu jako takovou, a proto byl v této části práce upřednostněn před C++. Grafy jsou potřebné k vizualizaci průběhu optimalizačního procesu, aby bylo možné činit rozhodnutí ve fázi ladění. Jsou zároveň užitečné při vizuálním testování strategie nebo prohlížení průběhu historické simulace.

Data určená pro tvorbu grafů se ukládají do CSV souborů, jelikož je lze snadno načíst pomocí knihovny `Pandas` do instance třídy `DataFrame`. Třída představuje tabulku s řádky a sloupci a nabízí širokou škálu metod pro manipulaci s uloženými daty. Operace mohou zahrnovat konverzi dat, výběr, čištění a spojení více instancí tříd `DataFrame`.

Jakmile jsou data připravena, jsou vykreslena pomocí knihovny `Matplotlib`. Nabízí způsoby, jak upřesnit typ grafu, legendu, nadpis, štítky, barvy a další. Instance třídy `DataFrame` lze často použít přímo pro vykreslování. Grafy lze uložit jako obrázky ve formátu PNG. Knihovna je dobře zdokumentována a na fóru Stack Overflow existuje spousta užitečných diskuzí.

K vývoji a spuštění kódu byl použit Jupyter Notebook. Prostředí je dostupné pomocí webového prohlížeče. Kód lze organizovat do samostatných buněk, které lze anotovat pomocí jazyka Markdown. Jakmile je program spuštěn, lze přistupovat ke všem hodnotám proměnných, dokud běží jádro platformy. Hodnoty lze zobrazit do prostoru pod buňkami. Vizuální vzhled notebooku je zachován i po zastavení běhu jádra, což zahrnuje i vykreslené grafy.

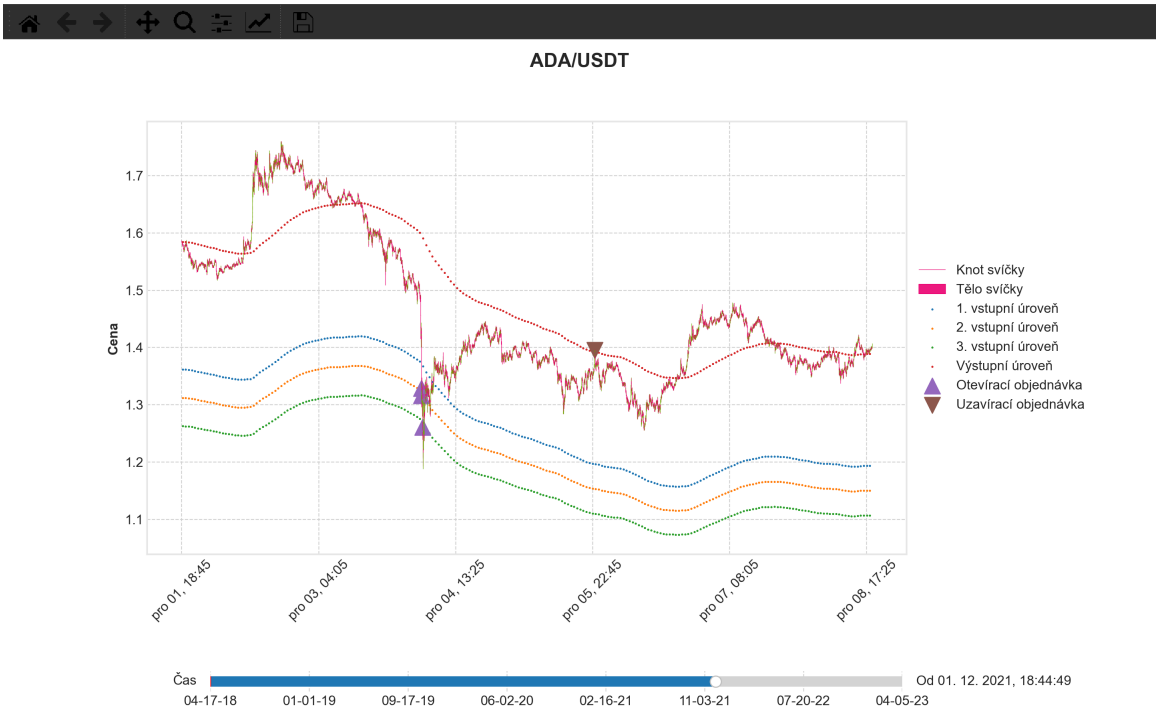
10.1 Svíčkový graf

Během historické simulace lze shromažďovat časové řady pomocí `chart_series::collector` třídy, která sleduje čistou hodnotu, zůstatek po uzavření pozice, otevírací a uzavírací ceny a hodnoty vstupního a výstupního indikátoru. Po uložení dat do CSV souboru je lze zpracovat pomocí výše popsaných nástrojů a zobrazit pomocí třídy `CandlestickWidget` spolu s historickými svíčkami.

Vlastní widget byl vyvinut, protože se graf obvykle nevejde na šíři obrazovky, proto bylo nutné přidat posuvník. Graf je vykreslen pomocí knihovny `mplfinance`, což je větev (branch) `Matplotlib`, která se specializuje na vykreslování finančních dat a musí být instalována samostatně. Nabízí způsob, jak snadno vykreslit svíčková data a také dobře integruje použití třídy `Pandas.DataFrame` pro předání dat.

Widget se skládá ze dvou částí: samotného grafu a posuvníku pro určení počátečního časového bodu (viz 10.1). Na graf jsou nejprve vynesena svíčková data, na které jsou bodově umístěny zbývající časové řady. Vedle něj je umístěna legenda. Posuvník je popsán pomocí časových značek, aby se dal snadněji používat.

Počet viditelných časových bodů a název grafu lze specifikovat při inicializaci třídy. Jelikož je vykreslováno mnoho časových bodů najednou, může aktualizace počátečního časového bodu chvíli trvat. Nástroj může být užitečný pro získání lepší představy o tom, jak strategie funguje.



Obrázek 10.1 Widget pro vykreslování svíček

Kapitola 11

Testování

K zajištění správnosti kódu byly použity dva nástroje: tvrzení (assertions) a unit testování. Pokud něco vždy platí, lze jej ověřit pomocí tvrzení. V C++ jsou k dispozici makra z hlavičkového souboru `assert.h`. Lze je například použít ke kontrole předpokladu, že vygenerované číslo je v požadovaném rozsahu. Pokud neplatí, způsobí ukončení programu a nelze je zachytit jako výjimky. Jejich cílem je zajistit, aby programátor neudělal chybu. Mohou být použity ve fázi ladění a odstraněny, když je program připravený do produkce. V C++ je lze snadno odstranit během kompilace pomocí příznaku `-DNDEBUG`.

Hlavním účelem výjimek je sdělit uživateli, proč program nemůže provést požadovanou úlohu. Mohou být způsobeny například poskytnutím nesprávně naformátovaného CSV souboru s historickými svíčkovými daty, bez nichž program nemůže pokračovat. Kontrola výjimečných stavů způsobuje režii navíc, takže se používají hlavně ve fázi konstrukce objektu, která se provádí pouze jednou. Výjimky jsou vnořeny pomocí `std::throw_with_nested`, aby mohl uživatel získat co nejvíce informací k vyřešení problému.

V C++ existuje několik testovacích frameworků, ale Boost.Test a GoogleTest jsou dva, které vynikají nejvíce. Oba používají makra k vytváření testovacích sad, testovacích případů a asercí, což minimalizuje množství práce potřebné k vytvoření testu. Jsou dobře zdokumentované, široce používané a také integrované do IDE, jako je Visual Studio a CLion. Pro tuto práci byl vybrán Boost.Test, protože je známější.

Testy jsou organizovány do testovacích modulů, které se skládají z testovacích sad obsahujících testovací případy. Existuje několik způsobů, jak danou hierarchii vytvářet a používat. Způsob vybraný pro projekt začíná vytvořením souboru `test.cpp`, který spojuje všechny moduly dohromady. V souboru je jako první definováno makro `BOOST_TEST_MAIN` a pod ním jsou zahrnuty všechny moduly. Na začátku každého modulu je nejprve zahrnut hlavičkový soubor frameworku. Testovací sady a případy jsou vytvářeny pomocí maker. Minimální příklad použití lze vidět na 11.1.

Výpis kódu 11.1 Ukázka testovacího modulu

```
#include <boost/test/unit_test.hpp>
#include <trading/ema.hpp>

BOOST_AUTO_TEST_SUITE(ema_test)
    BOOST_AUTO_TEST_CASE(default_constructor_test)
    {
        BOOST_REQUIRE_EQUAL(trading::ema{}.period(), std::size_t{1});
    }
    BOOST_AUTO_TEST_CASE(constructor_exception_test)
    {
```

```
        BOOST_REQUIRE_THROW(ema{0}, std::invalid_argument);
    }
    BOOST_AUTO_TEST_CASE(usage_test)
    {
        ma_usage_test<5,3,ema>({1, 2, 3, 7, 9}, {2.0, 4.5, 6.75}, 0.01);
    }
BOOST_AUTO_TEST_SUITE_END()
```

K samotnému testování jsou používány tvrzení, které jsou vyhodnoceny jako pravdivé nebo nepravdivé. Framework nabízí různá tvrzení, která jsou spojena s jednou ze tří úrovní testování. Nejméně závažná je úroveň `WARN`, která vydá varovnou zprávu, pokud výrok není pravdivý. Pokud je použita úroveň `CHECK`, vygeneruje chybu, ale umožní pokračovat v testování. Nejzávažnější je úroveň `REQUIRE`, která vygeneruje chybovou hlášku a přeruší provedení daného testovacího případu.

Nabízí velmi dobře formátované a přehledné konzolové logy, které slouží k hlášení průběhu testování. K dispozici jsou různé úrovně logování. Ve výchozím nastavení je hlášeno vše: úspěch, varování, chyby, nezachycené výjimky a další. Úroveň lze nastavit pomocí proměnné prostředí nebo příznaků příkazového řádku.

V CLion IDE lze testovací sady a případy spouštět jednotlivě. Mohou být také spuštěny v režimu ladění. Poskytuje různé nástroje, jak lépe prozkoumat výsledky testování.

11.1 Strategie

Strategie byla testována především proto, aby se zjistilo, zda byly správně implementovány podmínky pro vyvolání signálu. Pro testování byl použit ukazatel typu SMA s periodou jedna, jelikož je jeho hodnota rovna poslednímu předanému vzorku, což usnadňuje vytváření testů. Každá metoda pro generování signálu vrací booleovskou hodnotu označující, zda má být vytvořena objednávka, čehož bylo využito při testování (viz 11.2). Byly také provedeny testy pro kontrolu inicializace, aktualizace indikátorů a metod pro získávání hodnot indikátorů.

Výpis kódu 11.2 Testování podmínek pro signalizaci vytvoření otevírací objednávky

```
BOOST_AUTO_TEST_CASE(should_open_test)
{
    constexpr std::size_t n_levels{2};
    using indicator_t = trading::bazooka::indicator;
    using strategy_t = trading::bazooka::strategy<n_levels>;
    trading::sma entry{1}, exit{1};
    entry.update(10), exit.update(20);
    std::array<trading::fraction_t, n_levels> levels{{{2, 4}, {1, 4}}};
    strategy_t strategy(indicator_t{entry}, indicator_t{exit}, levels);

    // try open position above
    BOOST_REQUIRE_EQUAL(strategy.should_open(entry.value()*
        trading::fraction_cast<double>(levels[0])*1.001), false);
    BOOST_REQUIRE_EQUAL(strategy.next_entry_level(), 0);

    // open position
    BOOST_REQUIRE_EQUAL(strategy.should_open(entry.value()*
        trading::fraction_cast<double>(levels[0])), true);
    BOOST_REQUIRE_EQUAL(strategy.next_entry_level(), 1);
    ...
}
```

11.2 Ukazatelé

Hodnoty generované implementací ukazatelů EMA a SMA byly porovnány s hodnotami generovanými knihovnou pro technickou analýzu TA-lib. Pro testování byly použity různé hodnoty period a hodnot vzorků. Otestováno bylo také, že při zavolání konstruktoru s neplatnými parametry, je vyhozena výjimka. Byla zkontrolována správná indikace připravenosti ukazatelů jak při zavolání metody `update` tak `is_ready`.

11.3 Manažer

Manažer přijímá signály generované strategií a převádí je na objednávky, které jsou následně předány trhu. I když je to složitější třída, tak lze její stav simulovat zvenčí. Proto byly při testování pro srovnání stavu trhu použity očekávané instance tříd `market`, `wallet`, `order_sizer`, `open_order` a `close_all_order`. Nejprve byla zkontrolována správná inicializace třídy a poté byl proveden test použití.

Během testování byly volány metody pro vytváření tržních objednávek k otevření, zvýšení a uzavření pozice. V každé fázi byly nejprve vypočítány očekávané objednávky, které byly použity k aktualizaci očekávaného trhu. Objednávky bylo možné porovnat se skutečnými objednávkami pomocí metod pro přístup k posledním objednávkám, jako je `last_open_order`. Jakmile byly tržní stavy identické, mohly být k jejich porovnání použity metody přístupu k tržnímu stavu jako `wallet_balance`.

11.4 Obchodník

Třída `trader` spojuje funkcionality strategie a manažera. Implementuje jedinou metodu, která rozhoduje, kdy obchodovat. Jelikož třída přebírá manažera a strategii jako šablonové argumenty, mohly by být zaměněny za falešné (mock) objekty. Objekty jsou velmi jednoduché, implementují dané rozhraní a jejich atributy jsou typu boolean. Atributy se používají k nastavení výsledků volání metod. Implementaci falešné strategie lze vidět na výpisu 11.3. Tímto způsobem mohly být snadno otestovány všechny možné scénáře.

Výpis kódu 11.3 Implementace falešná strategie

```
struct mock_strategy {
    bool ready{false}, open{false}, close_all{false};

    bool is_ready() const { return ready; }

    bool should_open(trading::price_t) const { return open; }

    bool should_close_all(trading::price_t) const { return close_all; }
};
```

11.5 Trh

Třída `market` byla nejprve testována na správnou inicializaci a výjimky, které mohou být vyvolány. Konstruktor vyvolá výjimku typu `std::invalid_argument`, když je mu předána neplatná hodnota poplatku. Aby bylo možné otestovat její použití, byly k porovnání hodnot udržovány instance tříd `position` a `wallet` představující očekávaný stav třídy. Test prošel všemi stavy aktivní pozice vytvoření, navýšení a uzavření, během kterých byly skutečné hodnoty poskytované metodami třídy porovnány s očekávanými.

11.6 Pozice

Nejprve byla testována inicializace pro kontrolu, že jsou všechny atributy správně inicializovány. Hlavním provedeným testem byl test použití, během kterého byla pozice otevřena, navýšena a uzavřena. Během testu byla počítána očekávaná celková investovaná částka a očekávaná velikost pozice pro kontrolu výsledků volání metod a stavu atributů pozice.

11.7 Simulátor

Cílem prvního testu bylo ujistit se, že proces převzorkování prováděný v těle konstruktoru je správně implementován. Za tímto účelem byly vypočteny očekávané ceny a ceny pro ukazatele a porovnány s hodnotami získanými pomocí getrů instanciované třídy. Poté byla testována samotná simulace přístupná pomocí volacího operátoru třídy, který přebírá dva typy parametrů: obchodníka a sadu pozorovatelů. Ke kontrole, zda jsou pozorovatelé voláni správně, byl použit pozorovatel počítající počet událostí. Jako obchodník byla použita mockovací třída, která umožňovala nastavení výsledků volání metod před použitím simulátoru. Pomocí těchto dvou objektů bylo provedeno několik testovacích případů.

11.8 Statistiky

Výkonnostní statistiky byly testovány, aby bylo zajištěno, že jsou hodnoty správně inicializovány a aktualizovány. Bylo využito toho, že jsou rozděleny do 4 nezávislých skupin: statistik související s pohybem, statistik související se ziskem, čítače otevřených a uzavřených objednávek a zůstatku. Na všech byly proto provedeny samostatné testovací případy.

11.9 Generátory stavového prostoru

Generátory byly testovány na výjimky, které generují pro neplatný vstup. Hodnoty generované systematickými generátory byly zkontrolovány na platnost a jedinečnost a porovnány se sadou očekávaných hodnot. Současně byly použity různé kombinace hodnot parametrů. Jakmile byla získána určitá úroveň důvěry, byly systematické generátory použity k testování náhodných generátorů. Hlavním cílem byla kontrola dosažitelnosti, pokud náhodné generátory vygenerovaly všechny možné hodnoty v určitém počtu iterací, byly považovány za validní 11.4.

Výpis kódu 11.4 Testování dosažitelnosti stavů

```
template<class Value, class SystematicGenerator, class RandomGenerator>
void test_reachability(Value origin, SystematicGenerator&& sys_gen,
    RandomGenerator&& rand_gen, std::size_t n_it) {
    using map_t = std::map<Value, std::size_t>;
    std::size_t it{0};
    map_t options;

    for (const auto& value: sys_gen())
        options.insert(typename map_t::value_type{value, 0});

    while (it++!=n_it) {
        origin = rand_gen(origin);
        BOOST_REQUIRE(options.contains(origin));
        options[origin] += 1;
    }
    for (const auto& [opt, count]: options)
        BOOST_REQUIRE(count); }
```

11.10 Optimalizátory

Optimalizátory nabízejí vysokou úroveň abstrakce díky použití konceptů a šablon, takže pro jejich testování mohl být použit integer jako typ konfigurace, což značně zjednodušilo testy. Pro generování stavového prostoru mohly být využity generátory `systematic::int_range_generator` a `random::int_range_generator`.

Jako objektivní funkce byla zvolena funkce převádějící integer na double. Cílem všech testů bylo nalézt její maximální hodnotu. Omezení mohla být definována uvedením minimální hodnoty optimalizačního kritéria. Byly testovány na výjimky, které by mohly být vyvolány během inicializace objektu. Aby bylo možné zkontrolovat, zda jsou pozorovatelé informováni o událostech správně, byli vytvořeni pozorovatelé pro počítání událostí. Ukázku testování simulovaného ochlazování pomocí čítacího pozorovatele lze vidět na 11.5.

Výpis kódu 11.5 Testování optimalizátoru pomocí pozorovatele

```
BOOST_AUTO_TEST_CASE(count_events_test)
{
    using config_t = int;
    using optimizer_t =trading::simulated_annealing::optimizer<config_t>;
    using state_t = optimizer_t::state_type;
    double start_temp{100}, min_temp{1};
    auto cooler = [](auto& optimizer) {
        optimizer.current_temperature(optimizer.current_temperature()-1);
    };
    auto appraiser = [](const auto& current, const auto& candidate) {
        return current.value-candidate.value;;
    };
    auto objective = [](const auto& config) {
        return static_cast<double>(config);
    };
    auto constraints = [](const state_t& candidate) { return true; };
    auto neighbor = trading::random::int_range(1, 100, 1);
    auto equilibrium=trading::simulated_annealing::fixed_equilibrium{10};
    auto optimizer = optimizer_t{start_temp, min_temp};
    auto counter = event_counter{};
    int init_value{neighbor()};
    state_t init{init_value, objective(init_value)};
    trading::constructive_result result{init,
        [&](const auto& lhs, const auto& rhs) {
        return lhs.value>rhs.value;
    }};
    optimizer(init.config, result, constraints, cooler,
        objective, neighbor, appraiser, equilibrium, counter);
    BOOST_REQUIRE_EQUAL(counter.started_count, 1);
    BOOST_REQUIRE_EQUAL(counter.finished_count, 1);
    BOOST_REQUIRE_EQUAL(counter.cooled_count, optimizer.it());
    BOOST_REQUIRE(counter.better_accepted_count>0);
    BOOST_REQUIRE(counter.worse_accepted_count>0);
}
```

11.10.1 Hrubá síla

Optimalizátor hrubé síly byl nejsnáze testovatelný. Vyhledávací prostor byl generován systematicky, takže optimalizátor vždy najde nejlepší stav. Testováno bylo také použití omezení a procházení prázdného vyhledávacího prostoru.

11.10.2 Simulované ochlazování

Třída `simulated_annealing::optimizer` byla nejprve testována na správnou inicializaci a vyhození vyjímek, které jsou vyvolány při předání neplatných hodnot. Ke kontrole, zda byli pozorovatelé správně informováni o událostech, byl použit čítač událostí. Jednotlivé čítače událostí byly kontrolovány během optimalizace uvnitř metod čítače (viz 11.6) a nakonec po jejím skončení. Hledání závisí na náhodném prohledávání sousedních stavů, takže test použití byl nastaven tak, aby po dostatečném množství iterací bylo nalezení nejlepšího stavu téměř jisté.

Výpis kódu 11.6 Kontrola hodnot čítače uvnitř metody `finished`

```
template<class Optimizer>
void finished(const Optimizer&)
{
    BOOST_REQUIRE_EQUAL(started_count, 1);
    BOOST_REQUIRE_EQUAL(finished_count, 0);
    finished_count++;
}
```

11.10.3 Genetický algoritmus

Třída `genetic_algorithm::optimizer` byla testována velmi podobně. Operátor křížení byl implementován jako průměr konfigurací dvou rodičů. Mutace byla provedena stejně, pouze druhý rodič byl generován náhodně. Pro kontrolu třídy `bazooka::crossover` byly navrženy dva testovací případy. Cílem prvního bylo se ujistit, že pokud jsou rodiče stejní, tak je totožné i jejich dítě. Při druhém byli opakovaně vygenerováni náhodní rodiče a bylo zkontrolováno, že produkují validní potomky.

11.10.4 Tabu prohledávání

Třída `int_range_memory` byla použita jako paměť pro testování `tabu_search::optimizer`. Jinak testování probíhalo stejně jako u ostatních optimalizátorů. Paměťové třídy byly také testovány, aby bylo zajištěno, že si správně pamatují a zapomínají hodnoty, které jim byly předány.

11.11 Čtení a zápis do CSV

Pro testování čtení a zápisu do CSV souborů byly připraveny čtyři soubory: prázdný, pouze s hlavičkou, pouze s daty a kompletní soubor. Při použití třídy `io::csv::reader` může být vyvolána výjimka `std::runtime_error`, buď při volání konstruktoru, pokud soubor neexistuje, nebo při samotném čtení, pokud je soubor nesprávně naformátován nebo je třída nesprávně používána. Rozlišuje mezi výjimkou, která nastane při oddělování dat oddělovačem, a výjimkou, která nastane při převodu dat na požadovaný datový typ. Hlavním rozdílem mezi nimi je obsah chybové zprávy. Byly provedeny důkladné testy k simulaci požadovaných a výjimečných situací. Třída pro čtení byla použita k testování třídy pro zápis, aby se ověřilo, že produkuje validní výstup.

Kapitola 12

Experimentální vyhodnocení

Cílem experimentálního vyhodnocení bylo najít vhodná nastavení pro optimalizátory a mít možnost je vzájemně porovnat. Pomocí algoritmu hrubé síly byly nalezeny optimální stavy pro vyhledávací prostor použitý pro vyhodnocení heuristik. Při vyhodnocování heuristiky byly nejprve nalezeny vhodné parametry na sadě historických dat white-box a následně testovány na sadě black-box.

Výsledky experimentů byly pečlivě a systematicky uchovávány, aby bylo co nejjednodušší rozhodování, co upravit pro další experiment. Jsou umístěny v uspořádané hierarchii složek. Všechny mají společný adresář výstupních dat, který obsahuje čtyři složky s názvy odpovídajícími jednotlivým optimalizátorům, do nichž patří složky s výsledky jednotlivých experimentů. Například výsledky experimentu s názvem white-box-1 genetického algoritmu by bylo možné najít ve složce data/out/genetic-algorithm/white-box-1.

Výsledky všech experimentů se skládají z minimálně tří souborů: log.txt, best-states.json a settings.json. Soubor log.txt obsahuje zprávy, které byly během procesu optimalizace zobrazeny do konzole. Aby bylo možné výsledky zobrazit na standardní výstup a zároveň je zapsat do souboru, byl použit `boost::iostreams::stream` spolu s `boost::iostreams::tee_device`. Všechny optimalizátory mají společný začátek logu s informacemi o načtených historických datech 12.1.

Výpis kódu 12.1 Informace o načtených historických svíčkách z výstupního logu

```
candles read:
from: 2017-Aug-17 04:00:00
to: 2022-Oct-28 12:27:00
difference: 1898d 08:27:00.000000
count: 2725146
duration: 3.664928s
```

Nastavení experimentu jsou ukládána do souboru ve formátu JSON. Díky tomu je experiment opakovatelný, i když výsledky nemusí být stejné. Všechny optimalizátory mají společné sekce nastavení. Obsahují informace o historických datech, převzorkování a vyhledávacím prostoru A.1. Nejlepší stavy jsou uloženy jako seznam konfigurací a statistik v souboru best-states.json. K dispozici jsou různé statistiky, které mohou pomoci lépe porozumět chování optimalizované strategie A.2.

Python skripty byly použity k vytvoření souhrnných tabulek a grafů z dat generovaných optimalizací. Tímto způsobem bylo následné zpracování automatizováno, což snížilo možnost udělat chybu a výrazně zkrátilo čas potřebný k vytvoření přehledových tabulek.

Experimenty byly prováděny na notebooku MacBook Pro s 2,3 GHz Intel 4jádrovým procesorem. Procesor podporuje hyper-trading, který umožňuje současný běh 2 vláken na každém jádru. Což ovlivňuje framework OpenMP, který díky tomu může používat 8 vláken.

12.1 Historická data

Data pro experimenty byla získána ze stránky CryptoArchive (www.cryptoarchive.com.au). Svíčky pocházejí z Binance kryptoměnové burzy a jsou aktualizovány denně. Mezi hlavní důvody, proč byl web upřednostněn před ostatními dostupnými poskytovateli, patřilo to, že data byla zdarma, snadno dostupná a v požadované granularitě.

Bylo vybráno 10 různých kryptoměn včetně některých známějších jako Bitcoin (BTC), Ethereum (ETH) a Litecoin (LTC) a také novějších jako Solana (SOL). Pokrývají **2,7–5,7 let** historie, což stačí k zachycení různých tržních trendů. Například v případě Bitcoinu pokrývá nárůst ceny na konci roku 2017 a 2022, jakož i její pokles na začátku roku 2019 a 2023. Hodnoty kryptoměn jsou uvedeny v USDT, což je stablecoin, jehož hodnota se blíží americkému dolaru. Přehled informací o vybraných historických datech je uveden v tabulce 12.1.

Tabulka 12.1 Přehled historických dat

Měna	Počet svíček	Od (den)	Do (den)	Doba (roky)
ADA	2 616 803	17. 04. 2018	12. 04. 2023	5,0
BTC	2 963 250	17. 08. 2017	11. 04. 2023	5,7
DASH	2 122 039	28. 03. 2019	12. 04. 2023	4,0
DOGE	1 979 751	05. 07. 2019	12. 04. 2023	3,8
ETH	2 963 365	17. 08. 2017	12. 04. 2023	5,7
LTC	2 794 419	13. 12. 2017	12. 04. 2023	5,3
SOL	1 400 292	11. 08. 2020	11. 04. 2023	2,7
XLM	2 552 037	31. 05. 2018	11. 04. 2023	4,9
XRP	2 591 038	04. 05. 2018	11. 04. 2023	4,9
ZRX	2 160 979	28. 02. 2019	11. 04. 2023	4,1

12.2 Hrubá síla

Hlavním účelem algoritmu hrubé síly bylo vyzkoušet použití různých optimalizačních kritérií a omezení. Je to také dobrý způsob měření vlivu optimalizace. Zároveň to může být jednoduché a dostatečně efektivní řešení pro optimalizaci na malém vyhledávacím prostoru pro jednodušší strategie. Nebylo potřeba provádět white-box ani black-box vyhodnocení, protože optimalizátor nemá žádné nastavitelné parametry.

Log byl doplněn o informace o celkovém počtu stavů prohledávaného stavového prostoru a čase zahájení a ukončení optimalizace (viz 12.2). Časy byly získány a naformátovány pomocí funkce z knihovny Boost.DateTime. Celkový počet stavů byl vypočten procházením vyhledávacího prostoru, což u malých vyhledávacích prostorů nezabere ve srovnání s dobou optimalizace významné množství času.

Výpis kódu 12.2 Rozšíření logu pro algoritmus hrubé síly

```
periods count: 35
tag count: 1
levels count: 455
sizes count: 66
total count: 1051050
began: 2023-Apr-16 05:16:06
ended: 2023-Apr-16 06:20:26
duration: 01:04:20.000111
```

12.2.1 Optimalizace

Nejprve byly provedeny experimenty zaměřené na měření účinků optimalizací. Byly použity dva typy optimalizací: optimalizace kompilátoru a paralelismus. Hlavní použitá optimalizace kompilátoru byla `O3`, která nastavuje nejvyšší úroveň optimalizací. Druhou je `march=native`, která říká kompilátoru, aby provedl optimalizaci pro konkrétní CPU na daném počítači.

Dopad těchto optimalizací byl znatelný, protože sekvenční implementace se zapnutými optimalizacemi byla **11krát** rychlejší než ta bez nich. Použitím paralelismu s 8 vlákny byl algoritmus ještě **3,6krát** rychlejší. Porovnání výsledků experimentu s vyhledávacím prostorem skládajícím se z 3600 stavů a simulací běžící na 2,5 milionu minutových svíček lze vidět v tabulce 12.2.

Tabulka 12.2 Vliv použití optimalizace

Optimalizace	Doba trvání (s)	Zrychlení
žádné	986	1
optimalizace kompilátoru	88	11,2
optimalizace kompilátoru a paralelismus	24	41,1

Průměrná doba výpočtu objektivní funkce je přibližně **6 ms**, což znamená, že prohledání 1 milionu různých stavů by trvalo přibližně 1,5 h. Výpočet byl proveden na mnohem větším vyhledávacím prostoru pro získání přesnějších výsledků.

12.2.2 Experimenty

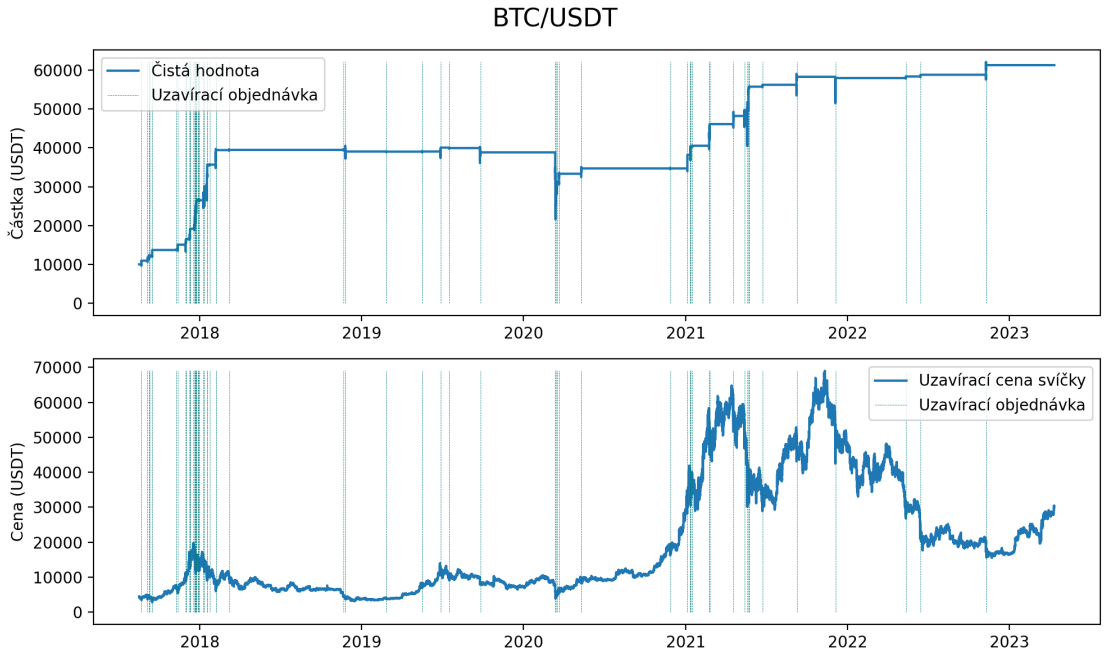
Na začátku bylo provedeno několik experimentů k nalezení vhodného vyhledávacího prostoru. Hodnoty úrovně nákupu se ukázaly být nejdůležitějším konfiguračním parametrem, který musel mít širokou škálu možných hodnot. Ke generátorům úrovně nákupu byl přidán parametr spodní hranice, čímž se snížil počet jedinečných hodnot potřebných k nalezení přijatelných výsledků a zmenšil se i vyhledávací prostor.

PROM byl zvolen jako optimální kritérium, jelikož poskytuje rozumné výsledky s vyváženým poměrem rizika a zisku. Byla také vyzkoušena jiná jednodušší optimalizační kritéria, jako maximalizace ziskového faktoru, což vedlo k nalezení konfigurací, které byly velmi bezpečné, avšak přinášely nízké zisky.

Výsledky byly primárně porovnávány pomocí statistik, nicméně byla použita i vizualizace. Na grafu 12.1 lze vidět, že strategie obchoduje, když dochází k rychlým poklesům ceny. Při vhodném nastavení přináší zisky, ale může se stát, že strategie otevře pozici příliš brzy, což vede k nižším ziskům nebo dokonce ztrátám. Stalo by se to například na začátku roku 2020, kdy začala pandemie COVID, následky jsou vidět na grafu s čistou hodnotou. Lze si všimnout, že většina obchodů byla provedena na začátku kolem roku 2018 a počet obchodů s postupem času klesá, což může naznačovat, že strategie byla optimalizována na příliš velkém množství dat.

Po dokončení počátečních testů byly provedeny dvě sady testů na všech historických datech. Hodnoty parametrů vyhledávacího prostoru byly zvoleny na základě získaných výsledků. I při použití optimalizace byl čas stále limitujícím faktorem. Rozumná velikost vyhledávacího prostoru se pohybuje 0,5–1 milion možných konfigurací, což trvá přibližně 0,75–1,5 h, než se dokončí jeden běh optimalizace.

Stavový prostor pro 1. sadu experimentů měl přibližně 0,4 milionu konfigurací a pro 2. sadu přibližně 1 milion. Hodnoty parametrů generátorů stavového prostoru a perioda převzorkování jsou uvedeny v tabulce 12.3. Změny provedené po 1. sadě experimentů spočívaly v tom, že počet jedinečných hodnot úrovní nákupu byl zvýšen z 15 na 20 a v seznamu možných typů ukazatelů byl ponechán pouze SMA. Perioda převzorkování zkrácena ze 45 minut na 15, počet jedinečných hodnot úrovní nákupu byl vrácen zpět na 15, byl zvýšen počet jedinečných hodnot velikostí nákupu z 6 na 11 a spodní hranice úrovní nákupu byla snížena z -20% na -40%.



Obrázek 12.1 Průběh historické simulace

Tabulka 12.3 Přehled hodnot parametrů vyhledávacího prostoru pro sady experimentů

Sada	Perioda převzorkování	Úrovně nákupu počet jedinečných hodnot	Úrovně nákupu spodní hranice (%)
1	45	15	$-25,0$
2	15	15	$-40,0$

Velikosti nákupu počet jedinečných hodnot	Perioda od	Perioda do	Perioda krok	Ukazatelé
6	3	60	3	$[\ SMA \quad EMA\ ]$
11	3	105	3	$[\ SMA\ ]$

Hodnoty úrovně nákupu a velikosti nákupu byly převedeny ze zlomků do čitelnějšího formátu. Obě jsou uvedeny v procentech. Úrovně jsou záporné a udávají, o kolik je úroveň snížena od hodnoty vstupního ukazatele. Velikosti označují část zůstatku na účtu, která je použita k nákupu na dané úrovni.

Z přehledu nejlepších konfigurací z 1. sady experimentů (viz 12.4) lze vidět, že 8 z 10 distribucí úrovně nákupu klade důraz na jednu úroveň nákupu tím, že na ni vloží 75 % zůstatku a minimum možné na zbytek. Na základě toho by se měl zvýšit počet jedinečných velikostí, aby byla dostupná větší rozmanitost velikostí nákupu. Medián první úrovně nákupu je -12.5% a poslední úrovně -19.6%. Poslední úroveň je velmi blízko spodní hranici úrovně nákupu, což naznačuje, že může být výhodné ji snížit ještě více. SMA je přítomen v 7 z 10 nejlepších konfigurací, takže ukazatel EMA lze v dalších experimentech vynechat. V případě měny XLM dosáhlo maximální hodnoty období, což naznačuje, že by měla být maximální hodnota zvýšena.

Tabulka 12.4 Přehled nejlepších konfigurací z 1. sady experimentů

Měna	Ukazatel	Perioda	Nákupní úrovně (%)	Nákupní velikosti (%)
ADA	EMA	51	$[\ -14,1 \quad -17,2 \quad -20,3\ ]$	$[\ 12,5 \quad 12,5 \quad 75,0\ ]$
BTC	SMA	15	$[\ -9,4 \quad -10,9 \quad -12,5\ ]$	$[\ 12,5 \quad 75,0 \quad 12,5\ ]$
DASH	SMA	15	$[\ -14,1 \quad -17,2 \quad -18,8\ ]$	$[\ 12,5 \quad 75,0 \quad 12,5\ ]$
DOGE	SMA	15	$[\ -12,5 \quad -20,3 \quad -21,9\ ]$	$[\ 25,0 \quad 12,5 \quad 62,5\ ]$
ETH	SMA	18	$[\ -14,1 \quad -15,6 \quad -17,2\ ]$	$[\ 75,0 \quad 12,5 \quad 12,5\ ]$
LTC	EMA	12	$[\ -10,9 \quad -12,5 \quad -14,1\ ]$	$[\ 12,5 \quad 12,5 \quad 75,0\ ]$
SOL	EMA	9	$[\ -7,8 \quad -12,5 \quad -17,2\ ]$	$[\ 12,5 \quad 62,5 \quad 25,0\ ]$
XLM	SMA	60	$[\ -12,5 \quad -17,2 \quad -23,4\ ]$	$[\ 12,5 \quad 12,5 \quad 75,0\ ]$
XRP	SMA	9	$[\ -10,9 \quad -12,5 \quad -21,9\ ]$	$[\ 12,5 \quad 12,5 \quad 75,0\ ]$
ZRX	SMA	15	$[\ -14,1 \quad -15,6 \quad -21,9\ ]$	$[\ 12,5 \quad 62,5 \quad 25,0\ ]$

Z přehledu statistik z 1. sady experimentu (viz 12.5) lze vidět, že nejnižší ziskový faktor je 3,5, což znamená, že hrubý zisk byl alespoň 3,5krát vyšší než hrubá ztráta. Nejvyššího ziskového faktoru 947,8 dosáhla měna XLM a zároveň nejnižšího sestupného pohybu zůstatku po uzavření pozice $-0,3$ %.

Tabulka 12.5 Přehled vybraných statistik z 1. sady experimentů

Měna	PROM	Zisk (%)	Ziskový faktor	Největší sestupný pohyb zůstatku po uzavření pozice (%)	Největší sestupný pohyb čisté hodnoty (%)
ADA	754,8	913,4	65,9	$-6,4$	$-26,4$
BTC	391,0	512,8	5,3	$-26,7$	$-46,7$
DASH	138,1	185,3	13,4	$-9,8$	$-40,0$
DOGE	1 309,3	1 541,7	35,5	$-4,7$	$-22,2$
ETH	664,5	940,5	5,1	$-8,5$	$-47,4$
LTC	739,8	885,6	14,9	$-6,2$	$-38,5$
SOL	202,5	262,8	3,5	$-16,7$	$-33,2$
XLM	581,3	674,5	947,8	$-0,3$	$-42,5$
XRP	225,2	292,4	5,9	$-18,3$	$-24,9$
ZRX	268,3	326,7	165,5	$-0,3$	$-36,7$

Na základě poznatků z 1. sady experimentů byla sestavena 2. sada experimentů. Z přehledu nejlepších konfigurací z 2. sady experimentů (viz 12.6) lze vidět, že zvýšení počtu jedinečných velikostí nákupu způsobilo zvýšení částky zůstatku vloženého na jednu úroveň, ze 75 % na 84,6 %. Minimální část nákupu 7,7 %, kterou bylo možné vložit na obchod, byla nejčastěji se vyskytující hodnotou. Perioda převzorkování byla 3krát zkrácena ze 45 minut na 15 minut, což například způsobilo ztrojnásobení hodnoty období u BTC. Úrovně nákupu se v podstatě nezměnily a ještě větší část zůstatku byla soustředěna na 2. nákupní úroveň. Převzorkování také způsobilo, že ADA

dosáhla nejvyšší hodnoty periody, což bránilo nalezení lepšího řešení. Všechny periody ukazatele se zvýšily 2–3krát kromě XLM, kde se snížila na polovinu z 60 na 27. Medián poslední nákupní úrovně se snížil na $-18,75$ %, což je stále velmi blízko -19 %, avšak zvýšení spodní hranice umožnilo DASH snížit hodnotu poslední úrovně na -32 %.

Tabulka 12.6 Přehled nejlepších konfigurací z 2. sady experimentů

Měna	Ukazatel	Perioda	Nákupní úrovně (%)	Nákupní velikosti (%)
ADA	SMA	105	[−12,5 −15,0 −17,5]	[7,7 7,7 84,6]
BTC	SMA	45	[−7,5 −10,0 −12,5]	[7,7 84,6 7,7]
DASH	SMA	51	[−15,0 −17,5 −32,5]	[7,7 61,5 30,8]
DOGE	SMA	27	[−15,0 −17,5 −20,0]	[7,7 84,6 7,7]
ETH	SMA	45	[−7,5 −12,5 −15,0]	[7,7 84,6 7,7]
LTC	SMA	36	[−10,0 −12,5 −15,0]	[7,7 84,6 7,7]
SOL	SMA	24	[−7,5 −10,0 −25,0]	[7,7 76,9 15,4]
XLM	SMA	27	[−10,0 −12,5 −15,0]	[7,7 84,6 7,7]
XRP	SMA	30	[−10,0 −17,5 −20,0]	[7,7 7,7 84,6]
ZRX	SMA	54	[−10,0 −15,0 −22,5]	[7,7 76,9 15,4]

Z přehledu statistik (viz 12.7) lze vypozorovat, že hodnoty zisku jsou řádově stejné u všech měn kromě ADA, LTC a XLM, kde se snížily o -47 %, -18.6 % a -27.7 %. Největší celkové zlepšení lze zaznamenat u měny DASH, jejíž ziskový faktor vzrostl 4,5krát a sestupné pohyby se zmenšily, na výsledný zisk to však mělo minimální dopad. Výsledky 1. sady experimentů byly tedy celkově lepší. Hlavním důvodem zhoršení bylo nejspíš snížení periody převzorkování.

Tabulka 12.7 Přehled vybraných statistik z 2. sady experimentů

Měna	PROM	Zisk (%)	Ziskový faktor	Největší sestupný pohyb zůstatku po uzavření pozice (%)	Největší sestupný pohyb čisté hodnoty (%)
ADA	395,3	483,9	14,2	−3,0	−26,4
BTC	439,0	540,7	5,2	−24,8	−49,3
DASH	146,0	189,3	60,3	−3,0	−33,6
DOGE	1 097,5	1 438,3	11,2	−7,5	−43,3
ETH	746,3	905,9	3,7	−22,6	−35,5
LTC	528,5	720,8	3,5	−15,1	−38,5
SOL	177,4	253,7	2,3	−31,2	−41,9
XLM	399,8	487,6	13,4	−4,7	−42,5
XRP	233,2	283,5	7,6	−17,8	−33,4
ZRX	250,2	326,9	4,0	−19,8	−37,7

12.3 Genetický algoritmus

Do souboru `settings.json` byly přidány informace o nastavení heuristiky, jako je počáteční velikost populace, způsob nahrazení populace, selekce a ukončovací kritérium. Průběh optimalizace byl uložen do souboru `progress.csv`, aby mohl být později použit pro tvorbu grafů. Logy byly doplněny o informace o průběhu optimalizace (viz 12.3) zahrnující velikost populace, průměrnou a nejlepší zdatnost.

Výpis kódu 12.3 Rozšíření logu pro genetický algoritmus

```
it: 0, population size: 161, mean fitness: 248.451, best fitness: 598.148
it: 1, population size: 204, mean fitness: 289.648, best fitness: 622.364
it: 2, population size: 257, mean fitness: 323.594, best fitness: 651.961
```

12.3.1 Experimenty

Experimentální vyhodnocení bylo provedeno na stejném vyhledávacím prostoru jako 1. sada experimentů algoritmu hrubé síly. Vyhledávací prostor se skládá z přibližně 0,4 milionu různých konfigurací. Byl vybrán především proto, že pro něj byly známy nejlepší stavy a mohly být tedy porovnány s výsledky vyhledávání genetického algoritmu.

Historická data byla rozdělena na data použitá pro white-box a black-box testování. Menší část sady zahrnující měny ADA, BTC a DASH byla použita pro testování white-box a zbytek pro black-box. Cílem vytvoření white-box sady bylo vybrat měny, které se při optimalizaci chovají dostatečně odlišně na to, aby bylo možné najít nastavení, které bude vyhovovat nejen konkrétní sadě, ale ideálně i jakékoli jiné měně pro daný vyhledávací prostor.

Bylo provedeno několik počátečních testů na jednom členu sady white-box. Jakmile se pro něj našlo vhodné nastavení, bylo vyzkoušeno na zbytku. Pro výběr rodičů byl použit ruletový výběr a pro nahrazení populace byl použit elitismus. Ukázalo se, že je lepší volbou než en-block nahrazování. Dále bylo provedeno několik sad testů s cílem spolehlivě najít nejlepší konfigurace pro všechny měny v co nejkratším čase. Počet rodičů a potomků byl nastaven na 2. Už od začátku se zdálo, že k nalezení nejlepších stavů bude stačit 10 iterací.

Ukázalo se, že tři nejdůležitější parametry jsou ty, které kontrolují velikost populace. Prvním parametrem je velikost počáteční populace. Pokud je příliš velká, zpomaluje optimalizaci a pokud je příliš malá, populace postrádá diverzitu. Dalším parametrem je elitní poměr, který udává počet elitních rodičů, kteří přejdou spolu s dětmi do další generace. Posledním parametrem je růstový faktor, který udává počet jedinců, kteří jsou vybráni jako rodiče. Nejlepší nalezenou kombinaci těchto parametrů lze vidět v tabulce 12.8.

Tabulka 12.8 Nejlepší nalezené nastavení pro sadu white-box pro genetický algoritmus

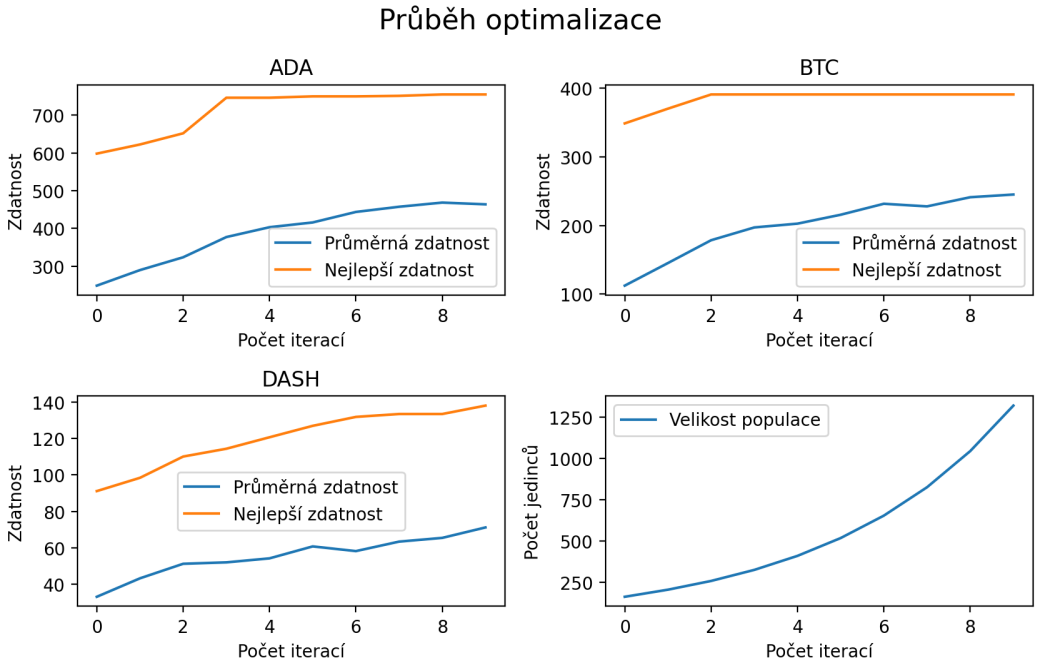
Velikost počáteční populace	Počet rodičů	Počet dětí	Elitní poměr (%)	Růstový faktor (%)	Počet iterací
128	2	2	15	110	10

Průběh optimalizace white-box testování lze vidět na 12.2. Nejlepší konfigurace pro ADA a BTC byly nalezeny v 1. polovině iterací a pro DASH během poslední, což může naznačovat, že počet iterací mohl být trochu zvýšen. Průměrná zdatnost má u všech rostoucí tendenci. Průměrné zrychlení ve srovnání s algoritmem hrubé síly je **22,5** a optimalizace trvá **1,3–2 min**, takže je hledání výrazně méně časově náročné. Výsledky lze nalézt v tabulce 12.9.

Tabulka 12.9 Výsledky white-box testování genetického algoritmu

Měna	Rozdíl hodnot optimalizačního kritéria (%)	Doba trvání (s)	Zrychlení
ADA	0,0	95,1	25,1
BTC	0,0	121,1	20,4
DASH	0,0	75,9	22,1

Nakonec bylo provedeno black-box testování. Výsledky jsou vidět v tabulce 12.10. Optimální stav byl nalezen pro 4 ze 7 měn. V případě SOL a XRP byly nalezené stavy o méně než -1 % horší než nejlepší a pro ETH o $-4,4$ %. Celkově byly nalezeny velmi dobré konfigurace v přiměřeně krátkém čase.



Obrázek 12.2 Průběh optimalizace white-box testování genetického algoritmu

Tabulka 12.10 Výsledky black-box testování genetického algoritmu

Měna	Rozdíl hodnot optimalizačního kritéria (%)	Doba trvání (s)	Zrychlení
DOGE	0,0	72,5	22,1
ETH	−4, 4	104,6	22,9
LTC	0,0	98,4	21,5
SOL	−0, 8	50,1	24,3
XLM	0,0	90,0	22,0
XRP	−0, 3	89,9	22,8
ZRX	0,0	79,3	21,6

12.4 Simulované ochlazování

Soubor `settings.json` byl rozšířen o informace o optimalizátoru jako je počáteční a minimalní teplota, použitý rozvrh ochlazování a equilibrium. Průběh optimalizace byl uložen do souboru `progress.csv`. Obsahuje informace o hodnotě aktuální teploty, aktuálního stavu a nejlepšího stavu. Ukázku rozšíření logu lze vidět 12.4.

Výpis kódu 12.4 Rozšíření logu pro simulované ochlazování

```
it: 0, temperature: 94, curr value: 230.451, best value: 592.883
it: 1, temperature: 55.5179, curr value: 312.835, best value: 592.883
it: 2, temperature: 44.7915, curr value: 206.282, best value: 592.883
```

12.4.1 Experimenty

Sady měn pro white-box a black-box testování zůstaly stejné jako u předchozích experimentů. Nejprve byly provedeny experimenty ke zjištění vhodného rozvrhu ochlazování, z nichž vynikla implementace třídy `basic_cooler`. Nepřebírá žádný parametr rozkladu a je závislá pouze na počáteční teplotě a aktuální iteraci.

Pro výpočet počtu pokusů o změnu aktuálního stavu byla použita třída `fixed_equilibrium`. Po několika počátečních experimentech se sada rozdělila do dvou skupin, pro které dobře fungovala dvě různá nastavení. Pro 1. skupinu skládající se z měn ADA a BTC byly hodnoty parametrů: počáteční teplota 128, minimální teplota 26 a počet pokusů 64 a pro 2. skupinu, která zahrnovala pouze DASH, bylo nastavení: počáteční teplota 54, minimální teplota 8 a počet pokusů 16. Při použití nastavení první skupiny pro druhou DASH nedosáhl intenzifikační fáze a při opačném použití nastala intenzifikační fáze u první skupiny příliš brzy.

Jelikož byly rozsahy vhodných hodnot parametrů příliš daleko od sebe, byla navržena nová dynamičtější implementace equilibria `temperature_based_equilibrium` založená na aktuální teplotě. Implementace umožnila snížení počtu pokusů s teplotou a tím spojení obou nastavení parametrů.

Nejlepší nalezené nastavení parametrů pro sadu white-box je uvedeno v tabulce 12.11. Hodnota počáteční teploty je velmi blízká hodnotě vhodné pro 1. skupinu a hodnota minimální teploty zase hodnotě teploty vhodné pro 2. skupinu. Počet pokusů je o 20 % nižší než aktuální teplota, začíná tedy na 75 pokusech a končí na 10. Rozsah změn generátoru náhodných období byl nastaven na 10 z výchozí hodnoty 1, jinak byl počet velikostí a úrovní, které lze změnit, ponechán na výchozí minimum.

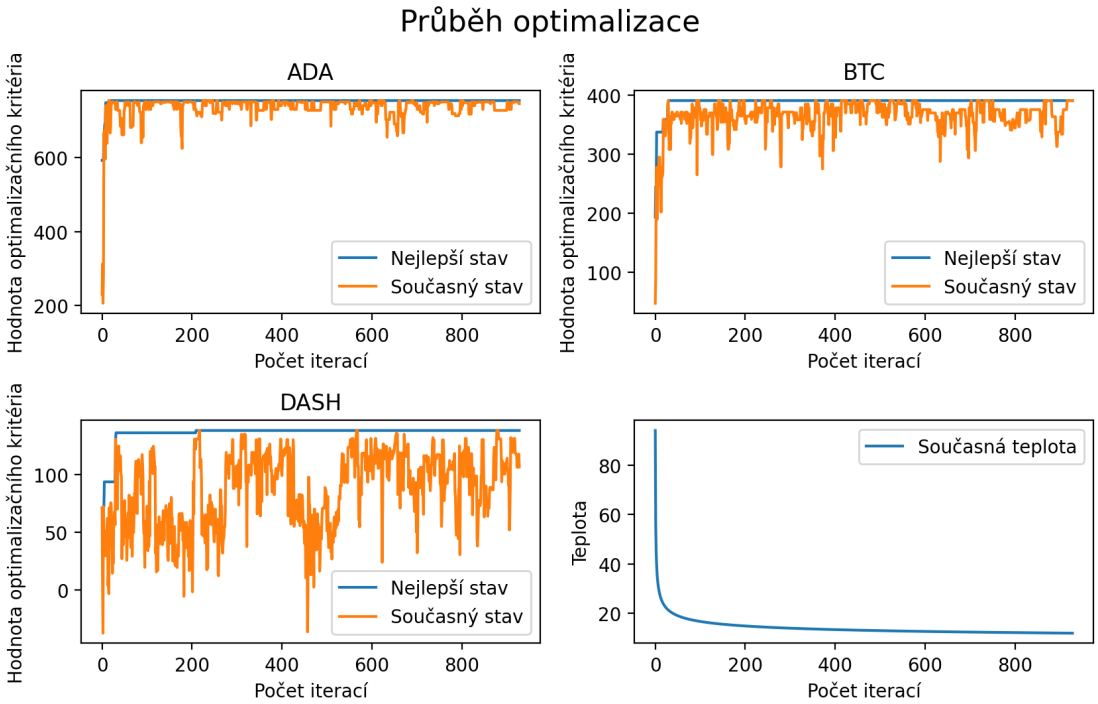
Tabulka 12.11 Nejlepší nalezené nastavení pro sadu white-box pro simulované ochlazování

Počáteční teplota	Minimální teplota	Koeficient equilibria založeném na teplotě (%)	Rozsah změny periody
94	12	−20	10

Počet změn nákupních úrovní	Počet změn nákupních velikostí
2	1

Průběh optimalizace white-box testování je vidět na grafu 12.3. V případě ADA dochází ke krátkému období diverzifikace, které rychle přechází do fáze intenzifikace. U BTC je přechod pomalejší a u DASH lze vypozorovat, že dochází k výraznější diverzifikační fázi. Individuálně lze tedy nalézt lepší hodnoty parametrů, ale i přesto se podařilo konvergovat ke globálním optimům, což lze vidět v tabulce 12.12.

Počet iterací je řádově vyšší než u genetického algoritmu, ale trvají mnohem kratší dobu. Průměrné zrychlení je **11,4** oproti hrubé síle a čas optimalizace se pohybuje mezi **2,5–3,9 min**. Je tedy asi **2krát** pomalejší než genetický algoritmus. Nicméně ve výsledcích testování black-box byl genetický algoritmus překonán, protože byly nalezeny optimální konfigurace pro všechny měny v sadě (viz 12.13).



Obrázek 12.3 Průběh optimalizace white-box testování simulovaného ochlazování

Tabulka 12.12 Výsledky white-box testování simulovaného ochlazování

Měna	Rozdíl hodnot optimalizačního kritéria (%)	Doba trvání (s)	Zrychlení
ADA	0,0	191,5	12,4
BTC	0,0	233,3	10,6
DASH	0,0	151,2	11,1

Tabulka 12.13 Výsledky black-box testování simulovaného ochlazování

Měna	Rozdíl hodnot optimalizačního kritéria (%)	Doba trvání (s)	Zrychlení
DOGE	0,0	159,7	10,0
ETH	0,0	192,5	12,5
LTC	0,0	177,2	12,0
SOL	0,0	86,9	14,0
XLM	0,0	163,8	12,1
XRP	0,0	161,2	12,7
ZRX	0,0	133,8	12,8

12.5 Tabu prohledávání

Do souboru `settings.json` byly přidány informace o nastavení optimalizátoru počet iterací tabu lhůty, velikost sousedství a kritérium ukončení. Průběh optimalizace byl uložen do souboru `progress.csv` a obsahoval informace o hodnotě optimalizačního kritéria aktuálního a nejlepšího stavu a velikosti krátkodobé paměti. Log byl také náležitě doplněn (viz 12.5).

Výpis kódu 12.5 Rozšíření logu pro tabu prohledávání

```
it: 10, best value: 27.2462, curr value: 27.2462, tabu list size: 11
it: 11, best value: 37.0617, curr value: 37.0617, tabu list size: 12
it: 12, best value: 52.0412, curr value: 52.0412, tabu list size: 13
```

12.5.1 Experimenty

Při hledání vhodného nastavení byly nejdůležitější dva parametry. První je tabu lhůta udávající počet iterací, během kterých je atribut označen jako zakázaný. Druhým je velikost sousedství, která určuje počet sousedů, kteří jsou prohledáni během jedné iterace. Všechny parametry byly pevně dané, včetně počtu iterací ukončovacího kritéria.

Pokud je hodnota tabu lhůty příliš nízká, dostane se do lokálního optima, z kterého se ji nepodaří uniknout a zůstane ve fázi vytěžování. Na druhou stranu, pokud je příliš vysoká, heuristika se nemůže vrátit do slibné oblasti a převažuje fáze průzkumu. Když je velikost sousedství příliš malá, pohybuje se heuristika rychleji skrze stavový prostor a okolí aktuálního stavu zůstává neprozkoumané. Naopak pokud je příliš velké, prozkoumává podobné konfigurace, pohybuje se pomalu a zůstává ve stejné oblasti stavového prostoru. Jinými slovy, tabu lhůta pomáhá držet směr a velikost sousedství udává velikost kroku.

Nejlepší parametry nalezené pro sadu white-box jsou uvedeny v tabulce 12.14. Nastavení náhodných generátorů stavového prostoru bylo ponecháno stejné jako u předchozí heuristiky. Z grafu 12.4 je vidět, že v prvních 20 iteracích došlo k rychlé konvergenci. Dále si lze všimnout, že když se heuristika začne přesouvat z fáze vytěžování do fáze průzkumu, tak se hodnota optimalizačního kritéria aktuálního stavu začne vzdalovat tomu nejlepšímu. Například v případě BTC se začne vzdalovat od nejlepšího stavu kolem 20. iterace a vrací se až v 50. iteraci, kdy najde lepší konfiguraci.

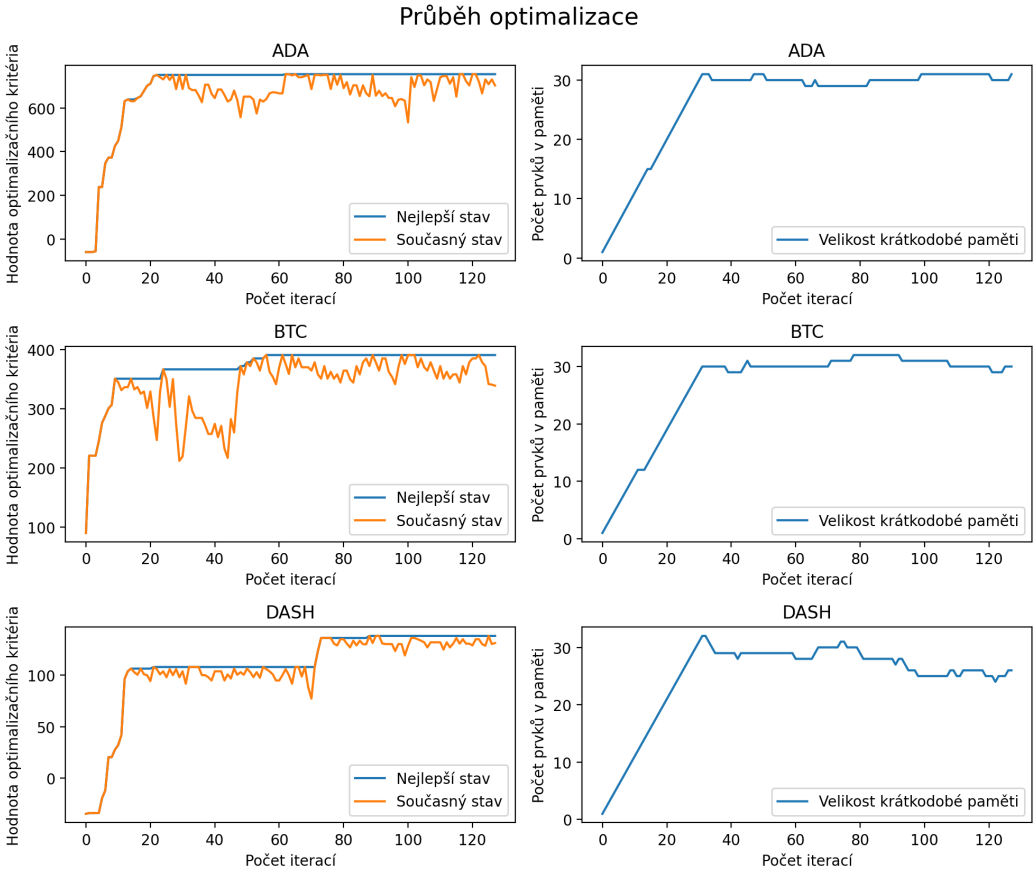
Tabulka 12.14 Nejlepší nalezené nastavení pro sadu white-box pro tabu prohledávání

Tabu lhůta	Velikost sousedství	Počet iterací	Rozsah změny periody
32	128	128	10

Počet změn nákupních úrovní	Počet změn nákupních velikostí
2	1

Velikost paměti téměř vždy dosáhne hodnoty blízké své maximální kapacitě, která je rovna tabu lhůtě. Pokud je stejný atribut zapamatován vícekrát po sobě, velikost paměti se sníží. Tah aktuálního stavu může být tabu, jelikož se musí aktuální stav každou iteraci změnit.

Průměrné zrychlení oproti hrubé síle pro sadu white-box je **9,5**, což je o trochu pomalejší než simulované ochlazování. Doba výpočtu se pohybuje v rozmezí **3,1-4,4 min**. Opět bylo dosaženo globálního optima pro všechny měny. Při použití na vzorky ze sady black-box bylo dosaženo 3 ze 7 globálních optim. Rozdíl mezi globálním optimem byl nižší než -2 % ve 3 případech. Nejvýznamnější rozdíl byl $-9,8$ %, což je 2krát horší výsledek než nejhorší výsledek při použití genetického algoritmu.



Obrázek 12.4 Průběh optimalizace white-box testování tabu prohledávání

Tabulka 12.15 Výsledky white-box testování simulovaného ochlazování

Měna	Rozdíl hodnot optimalizačního kritéria (%)	Doba trvání (s)	Zrychlení
ADA	0,0	241,1	9,9
BTC	0,0	260,8	9,5
DASH	0,0	184,3	9,1

Tabulka 12.16 Výsledky black-box testování tabu prohledávání

Měna	Rozdíl hodnot optimalizačního kritéria (%)	Doba trvání (s)	Zrychlení
DOGE	0,0	158,8	10,1
ETH	0,0	272,4	8,8
LTC	0,0	249,4	8,5
SOL	-0,8	91,2	13,4
XLM	-9,8	207,1	9,6
XRP	-1,7	227,5	9,0
ZRX	-1,4	227,9	7,5

Závěr

Nejprve byly představeny třídy složitosti a teorie kombinatorické optimalizace. Hlavním účelem bylo zavést koncepty a přístupy, které byly zásadní pro pochopení procesu optimalizace. Dále byly popsány tři vybrané metaheuristiky použité v praktické části: simulované ochlazování, genetický algoritmus a tabu prohledávání.

V další části bylo představeno algoritmické obchodování. Nejprve byl popsán samotný proces obchodování, který zahrnuje vytváření a plnění objednávek. Poté byly představeny různé automatizovatelné fáze obchodování včetně tří hlavních typů analýz, které se používají ke generování obchodních signálů.

Následně bylo popsáno zpětné testování, jehož účelem je vyhodnocení obchodní strategie na historických tržních datech. Byly uvedeny vzorce výkonových statistik používané v praktické části. Představeny byly také prostředky pro vizualizaci historických dat včetně jejich zdrojů. Dále byla popsána obchodní strategie Bazooka, jejíž parametry byly optimalizovány, a platforma TradingView, která byla použita pro její vývoj.

Aplikace byla napsána v C++20. Jazyk byl použit, jelikož nabízí prostředky k efektivnějšímu využívání počítačových zdrojů a použití optimalizací. Kód je organizován do frameworku, který se skládá ze dvou hlavních částí, z nichž první se zaměřuje na historickou simulaci a druhá na optimalizaci. Výstupy optimalizace se ukládají do souborů ve formátu CSV a JSON, které lze následně zpracovat pomocí vědeckých balíčků jazyka Python.

Návrh a konvence pojmenování frameworku byly inspirovány knihovnami Boost. Kód byl napsán objektově především proto, že umožňuje zapouzdření, snadnější unit testování a použití návrhových vzorů. Pro zjednodušení kódu a snížení závislostí byly použity návrhové vzory strategie a pozorovatel. Z důvodů vyšší efektivity byl preferován statický polymorfismus před dynamickým. Minimalizována byla také nutnost použití dynamické alokace.

Zpětné testování bylo inspirováno platformou TradingView a proces vytváření a plnění objednávek burzou Binance. Byla vytvořena sada tříd, které byly použity pro simulaci obchodování. Důvodem proč jich bylo více, bylo rozdělení odpovědností. Existuje třída, která má za úkol pouze rozhodovat, další, která vytváří objednávky a poslední je plní. Celá historická simulace se provádí voláním jediné metody.

Konfigurace se skládá ze čtyř atributů: typu ukazatele, periody ukazatele, velikostí nákupu a úrovní nákupu. Byly navrženy a implementovány systematické generátory stavového prostoru používané algoritmem hrubé síly a náhodné generátory používané heuristikami. K systematickému generování konfigurací byly použity korutiny zavedené standardem C++20. Náhodné generátory používaly rovnoměrné rozdělení.

Při navrhování optimalizátorů bylo cílem, aby byly co nejjednodušší a nejflexibilnější. Rozhraní parametrů předávaných optimalizátorům je kontrolováno pomocí konceptů zavedených ve standardu C++20. Průběh optimalizace lze pozorovat a zaznamenávat pomocí tříd pozorovatelů, které jsou předány při volání optimalizátoru. Algoritmus hrubé síly byl paralelizován

pomocí knihovny OpenMP.

Framework byl nejprve otestován než došlo k finálnímu vyhodnocení. Pro testování jednotlivých tříd byly vytvořeny unit testy. Byly použito mockování i fuzzing. Největší pozornost byla věnována generátorům vyhledávacího prostoru. Bylo zajištěno, že systematické generátory generují jedinečné a validní výstupy. Poté byly přidány k testování náhodných generátorů, pro které byl přidán test dosažitelnosti, aby bylo zajištěno, že každý stav ve vyhledávacím prostoru je dosažitelný.

Nakonec bylo provedeno experimentální vyhodnocení. Jako zdroj historických dat byly zvoleny minutové svíčky 10 kryptoměn s 1,4–2,9 miliony záznamy, což odpovídá 2,7–5,7 letům historie. Jako optimalizační kritérium byla použita statistika PROM, což je pesimističtější zisk, díky kterému bylo možné získat použitelnější výsledky.

První experimenty byly provedeny pomocí algoritmu hrubé síly. Při aplikaci optimalizací kompilátoru a použití více vláken byl algoritmus 44,1krát zrychlen oproti jeho neoptimalizované verzi. Na základě provedených sad experimentů byl vymezen vyhledávací prostor pro experimentální vyhodnocení heuristických přístupů. Skládal se z 0,4 milionu konfigurací. Algoritmu hrubé síly, se všemi zapnutými optimalizacemi, trvalo nalezení optimálního stavu v průměru 0,5 hodiny pro jednu měnu.

Při vyhodnocování heuristických optimalizátorů byla data rozdělena do dvou sad white-box a black-box. Parametry optimalizátoru byly nejprve vyladěny na sadě white-box a poté otestovány na sadě black-box. Nejlepších celkových výsledků dosáhlo simulované ochlazování, které našlo všechna globální optima v průměru za 2,6 minuty a průměrného zrychlení oproti hrubé síle 11,4 pro sadu black-box. Genetický algoritmus dosáhl nejvyššího zrychlení 22,5krát o proti hrubé síle, ale u některých vzorků nenašel globální optimum.

Celkově se genetický algoritmus zdál nejsnáze laditelný a nejrobustnější. Nalezení vhodných parametrů pro simulované ochlazování zabralo nejvíce času a pro dosažení rozumných výsledků bylo nutné vytvořit vlastní equilibrium založené na hodnotě aktuální teploty. Nejméně konzistentní, z hlediska dosažených výsledků, se jevilo tabu prohledávání.

Jako každý optimalizační problém je aplikace náchylná na overfitting. Výsledky optimalizace jsou nejlepší nalezenou konfigurací pro daná historická data z poskytnutého vyhledávacího prostoru, to však neznamená, že by na nově poskytnutých datech dosáhly stejných výsledků. Z hlediska strojového učení je nyní strategie optimalizována na trénovací sadě, ale výsledky nejsou porovnány s validační či testovací sadou. Jedním ze způsobů, jak se tomu vyhnout, by bylo použít walk-forward optimalizaci používanou platformou MultiCharts.

Historickou simulaci lze zpřesnit zohledněním skluzu nebo použitím přesnějších historických dat, jako jsou tiky, namísto minutových svíček. Dále lze zavést podporu pro více typů objednávek, jako jsou limitní objednávky. Pozice může být rozšířena, aby bylo možné využívat páku a stop loss. Mezi výstup historické simulace lze zařadit i seznam provedených objednávek.

Dalším způsobem, jak zlepšit aplikaci, by bylo použití databází k ukládání a manipulaci s daty. InfluxDB by se hodila, pokud by se zvýšil objem historických dat, a tím i potřeba je lépe organizovat a vytvářet složitější dotazy. MongoDB lze použít k lepší organizaci a manipulaci s výstupními daty, jako jsou nejlepší stavy a nastavení, které jsou aktuálně uloženy ve formátu JSON.

Aplikaci lze také obohatit o uživatelské rozhraní, které by ji učinilo uživatelsky přívětivější. Mohlo by poskytnout snazší a přehlednější způsob organizace, vyhodnocení a prohlížení výsledků experimentů. Může být webové nebo desktopové v závislosti na tom, co by se ukázalo jako výhodnější.

Příloha A

Přílohy

Výpis kódu A.1 Společné části nastavení

```
{
    "candles": {
        "count": 2160979,
        "currency pair": {
            "base": "ZRX",
            "quote": "USDT"
        },
        "from": 1551326400,
        "to": 1681230420
    },
    "optimization criterion": "prom",
    "resampling": {
        "averaging method": "ohlc4",
        "period[min]": 15
    },
    "search space": {
        "indicator": {
            "period": {
                "from": 3,
                "step": 3,
                "to": 105
            },
            "types": [
                "sma"
            ]
        },
        "levels": {
            "count": 3,
            "lower bound": "12/20",
            "unique count": 15
        },
        "open order sizes": {
            "unique count": 11
        }
    }
}
```

Výpis kódu A.2 Informace uložené o nejlepším stavu


```
{
    "configuration": {
        "indicator": {
            "period": 36,
            "type": "ema"
        },
        "levels": ["5/6","4/6","2/6"],
        "open sizes": ["5/7","1/7","1/7"]
    },
    "statistics": {
        "close balance": {
            "max": 78961.828125,
            "max drawdown": {
                "amount": -459.23828125,
                "percent": -1.4765747785568237
            },
            "max run up": {
                "amount": 68961.828125,
                "percent": 689.6182861328125
            },
            "min": 10000.0
        },
        "equity": {
            "max": 78961.828125,
            "max drawdown": {
                "amount": -11520.890625,
                "percent": -34.7911491394043
            },
            "max run up": {
                "amount": 69040.4375,
                "percent": 695.8746337890625
            },
            "min": 9921.390625
        },
        "gross loss": -459.23828125,
        "gross profit": 69421.0625,
        "net profit": 68961.828125,
        "open order counts": [24,2,0],
        "order ratio": 1.0833333333333333,
        "profit factor": 151.16566467285156,
        "total close orders": 24,
        "total open orders": 26
    }
},
```

Bibliografie

1. PERES, Fernando; CASTELLI, Mauro. Combinatorial Optimization Problems and Metaheuristics: Review, Challenges, Design, and Development. *Applied Sciences*. 2021, roč. 11, č. 14. ISSN 2076-3417. Dostupné z DOI: 10.3390/app11146449.
2. KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by Simulated Annealing. *Science*. 1983, roč. 220, č. 4598, s. 671–680. ISSN 0036-8075, ISSN 1095-9203. Dostupné z DOI: 10.1126/science.220.4598.671.
3. NI, Jiarui; ZHANG, Chengqi. An Efficient Implementation of the Backtesting of Trading Strategies. In: 2005, sv. 3758, s. 126–131. ISBN 978-3-540-29769-7. Dostupné z DOI: 10.1007/11576235_17.
4. ERICKSON, Jeff. *Algorithms*. 1st paperback edition. Erscheinungsort nicht ermittelbar: Selbstverlag, 0013. ISBN 9781792644832.
5. KAVEH, A. *Applications of Metaheuristic Optimization Algorithms in Civil Engineering*. Cham: Springer International Publishing, 2017. ISBN 9783319480114. Dostupné z DOI: 10.1007/978-3-319-48012-1.
6. KATOCH, Sourabh; CHAUHAN, Sumit Singh; KUMAR, Vijay. A review on genetic algorithm: past, present, and future. *Multim. Tools Appl.* 2021, roč. 80, č. 5, s. 8091–8126. Dostupné z DOI: 10.1007/s11042-020-10139-6.
7. RUTENBAR, R.A. Simulated annealing algorithms: an overview. *IEEE Circuits and Devices Magazine*. 1989, roč. 5, č. 1, s. 19–26. Dostupné z DOI: 10.1109/101.17235.
8. *A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence)* [online]. [B.r.]. Dostupné také z: http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/.
9. LAGUNA, Manuel. Tabu Search. In: *Handbook of Heuristics*. Ed. MARTÍ, Rafael; PARDALOS, Panos M.; RESENDE, Mauricio G. C. Cham: Springer International Publishing, 2018, s. 741–758. ISBN 978-3-319-07124-4. Dostupné z DOI: 10.1007/978-3-319-07124-4_24.
10. NUTI, Giuseppe; MIRGHAEMI, Mahnoosh; TRELEAVEN, Philip; YINGSAEREE, Chaiyakorn. Algorithmic Trading. *Computer*. 2011, roč. 44, č. 11, s. 61–69. Dostupné z DOI: 10.1109/MC.2011.31.
11. TRELEAVEN, Philip; GALAS, Michal; LALCHAND, Vidhi. Algorithmic trading review. *Communications of the ACM*. 2013, roč. 56, č. 11, s. 76–85. ISSN 0001-0782, ISSN 1557-7317. Dostupné z DOI: 10.1145/2500117.
12. PARDO, Robert; PARDO, Robert. *The evaluation and optimization of trading strategies*. 2nd ed. Hoboken, N.J: John Wiley, 2008. Wiley trading. ISBN 9780470128015.

13. *Maximum Drawdown (MDD) Defined, With Formula for Calculation* [online]. [B.r.]. Dostupné také z: https://www.investopedia.com/terms/m/maximum-drawdown-mdd.asp.
14. *Japanese candlestick* [online]. 2023. Dostupné také z: https://corporatefinanceinstitute.com/resources/capital-markets/japanese-candlestick/.
15. *Stock API - polygon* [online]. [B.r.]. Dostupné také z: https://polygon.io/stocks#stocks-product-cards.
16. BOWMAN, Richard. *5 best stock backtesting platforms of 2023* [online]. 2023. Dostupné také z: https://finmasters.com/best-stock-backtesting-platforms/#gref.
17. *TradingView trading platform capabilities and features* [online]. [B.r.]. Dostupné také z: https://www.tradingview.com/features/.
18. *MutliCharts Trading Platform features* [online]. [B.r.]. Dostupné také z: https://www.multicharts.com/features/.
19. WEST, Zck. *Moving averages: Smoothing out the noise for better predictions* [online]. 2022. Dostupné také z: https://www.alpharithms.com/moving-averages-083315/.
20. *Moving average* [online]. Wikimedia Foundation, 2023. Dostupné také z: https://en.wikipedia.org/wiki/Moving_average.
21. *Exponential moving average (EMA)* [online]. 2023. Dostupné také z: https://corporatefinanceinstitute.com/resources/capital-markets/exponential-moving-average-ema/.
22. *Moving averages - simple and exponential* [online]. [B.r.]. Dostupné také z: https://school.stockcharts.com/doku.php?id=technical_indicators%5C%3Amoving_averages.
23. *C++* [online]. [B.r.]. Dostupné také z: https://www.cs.mcgill.ca/~rwest/wikispeedia/wpcd/wp/c/C%5C%252B%5C%252B.htm#:~:text=Bjarne%5C%20Stroustrup%5C%20developed%5C%20C%5C%2B%5C%2B%5C%20(originally,to%5C%20the%5C%20C%5C%20programming%5C%20language..
24. *TIOBE Index* [online]. [B.r.]. Dostupné také z: https://www.tiobe.com/tiobe-index/.
25. *General Python FAQ* [online]. [B.r.]. Dostupné také z: https://docs.python.org/3/faq/general.html.
26. ZOFFOLI, Francesco. *How to Use C++ Dependency Injection to Write Maintainable Software - Francesco Zoffoli CppCon 2022* [online]. 2022. Dostupné také z: https://www.youtube.com/watch?v=l6Y9PqyK1Mc.
27. *Coroutines* [online]. [B.r.]. Dostupné také z: https://en.cppreference.com/w/cpp/language/coroutines.
28. BAKER, Lewiss. *CppCoro - A coroutine library for C++* [online]. [B.r.]. Dostupné také z: https://github.com/lewissbaker/cppcoro.
29. *What's a design pattern?* [Online]. [B.r.]. Dostupné také z: https://refactoring.guru/design-patterns/what-is-pattern.
30. *Observer* [online]. [B.r.]. Dostupné také z: https://refactoring.guru/design-patterns/observer.
31. *Strategy* [online]. [B.r.]. Dostupné také z: https://refactoring.guru/design-patterns/strategy.
32. LOHMANN, Niels. *JSON for Modern C++* [online]. 2022. Dostupné také z: https://github.com/nlohmann.
33. *Introducing JSON* [online]. [B.r.]. Dostupné také z: https://www.json.org/json-en.html.

Obsah přiloženého média

- `readme.txt` — stručný popis obsahu média
- `exe` — adresář se spustitelnou formou implementace
- `src`
  - `impl` — zdrojové kódy implementace
    - `include` — hlavičkové soubory
      - `cppcoro` — vybrané části knihovny CppCoro
      - `trading` — obchodní framework
    - `src` — zdrojové soubory
      - `data` — data pro experimenty
        - `in` — vstupy
        - `out` — výstupy
        - `report` — výstupy pro zprávu
    - `test` — unit testy
      - `data` — testovací data
        - `in` — vstupy
        - `out` — výstupy
      - `trading` — testy obchodního frameworku
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF