Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Sterowania i Elektroniki Przemysłowej

Praca dyplomowa magisterska

na kierunku Informatyka Stosowana

w specjalności Inżynieria oprogramowania

Porównanie rozwiązań stosowanych do optymalizacji metod harmonogramowania

inż. Grzegorz Poręba

numer albumu 324205

promotor

dr inż. Krzysztof Hryniów

WARSZAWA 2024

Porównanie rozwiązań stosowanych do optymalizacji metod harmonogramowania

Streszczenie

University Class Scheduling Problem (UCSP) jest to powszechnie spotykany na uczelniach problem rozplanowania zajęć, który polega na przydzieleniu zajęć do sal oraz wykładowców bez powodowania konfliktów czasowych. Po rozpisaniu harmonogramu należy zająć się drugorzędną kwestią, tj. przydzielaniem studentów do danych zajęć. W tej pracy pochylam się nad problemem maksymalizacji zadowolenia studentów poprzez przydzielanie ich do wcześniej utworzonych zajęć, biorąc pod uwagę zdefiniowane przez nich preferencje. We wstępie dokonany zostanie przegląd literatury. Następnie zdefiniowane zostaną miękkie oraz twarde ograniczenia, które dane rozwiązanie musi spełnić. Wprowadzę także teorię opisującą pięć porównywanych rozwiązań: ILP (Integer Linear Programming), SAT (Satisfiability problem), algorytm gentyczny, SA (Simulated Annealing) oraz PSO (Particle swarm optimization). Następnie zostanie omówiona implemetacja rozwiązań wraz z przedstawieniem szczegółowego opisu i kodu dotyczącego ciekawszych fragmentów. Przedstawiona będzie także platforma testowa, na której wykonane zostaną testy dla planu zajęć, który zawiera 26 wydarzeń oraz 50 uczniów. W końcowej części pracy przedstawię wyniki testów.

Słowa kluczowe: University Class Scheduling Problem, Integer Linear Programming, Algorytm genetyczny, Simulated Annealing, maksymalizacja zadowolenia studentów

Comparison of solutions used to optimize scheduling methods

Abstract

University Class Scheduling Problem (UCSP) is a common university scheduling problem that involves assigning classes to classrooms and lecturers without causing time conflicts. Once the schedule has been created, focus is shifted to a secondary issue i.e. assigning students to specific classes. In this paper, I focus on the problem of maximizing student satisfaction by assigning them to previously created classes, taking into account the preferences they have defined. In the introduction, a literature review will be made. This will be followed by a definition of the soft and hard constraints that the solution must meet. I will also introduce the theory describing the five solutions being compared: ILP (Integer Linear Programming), SAT (Satisfiability problem), Genetic algorithm, SA (Simulated Annealing) and PSO (Particle swarm optimization). Then, the implementation of the solutions will be discussed, along with a detailed description and code for the more interesting parts. A test plarform will also be presented on which tests will be performed for a timetable that contains 26 events and 50 students. In the final part of the work I will present the results of the tests.

Keywords: University Class Scheduling Problem, Integer Linear Programming, Genetic Algorithm, Simulated Annealing, maximization of student satisfaction

Spis treści

Rozdział 1

Wstęp

Uczelnie każdego roku borykają się z problemem rozplanowania zajęć na uczelni. Dzieje się tak, gdyż zajęcia przewidziane na dany semestr muszą być odpowiednio przydzielone do wykładowców, a także przypisane do sal, które z kolei powinny spełnić określone kryteria, takie jak rozmiar pomieszczenia i dostępność niezbędnego wyposażenia. Ponadto kursy powinny zostać ułożone w harmonogramie, tak aby nie wystąpiły między nimi konflikty czasowe. W literaturze problem ten jest często określany jako University Class Scheduling Problem (UCSP) - na jego temat powstało już wiele prac rozważających jego specyfikę oraz heurystyki.

Po rozpisaniu harmonogramu należy pochylić się nad drugorzędną kwestią, tj. przydzielaniem studentów do danych zajęć, co może odbywać się poprzez podział uczniów na grupy zajęciowe lub ich losowe rozlokowanie w sposób niełamiący twardych ograniczeń. W tej pracy pochylam się nad problemem maksymalizacji zadowolenia studentów poprzez przydzielanie ich do wcześniej utworzonych zajęć, uwzględniając zdefiniowane przez nich wcześniej preferencje.

Harmonogramowanie jest to trudny i często spotykany problem optymalizacyjny, który polega na przypisaniu określonych wydarzeń do stałej liczby okien czasowych, gwarantując przy tym spełnienie twardych oraz miękkich ograniczeń tak, aby pożądane cele zostały spełnione w największym możliwym stopniu [7]. Problem harmonogramowania jest NP-trudny [26].

Problem harmonogramowania, stanowiący przedmiot tej pracy, pod wieloma względami przypomina dokładnie opisany w literaturze Nurse Scheduling Problem, w którym terminy zmian w szpitalu przydzielane są pielęgniarkom, przy uwzględnieniu ich preferencji oraz danych ograniczeń.

1.1 Rzeczywiste założenia i ograniczenia

Punktem wyjścia do pracy nad opisanym wyżej problemem, jest gotowy harmonogram wydarzeń wraz z przypisanymi do nich prowadzącymi oraz godzinami, w których będą się one odbywać. Wydarzenia dzielimy na wykłady, na które uczęszczają wszyscy zapisani na dany przedmiot, oraz zajęcia, z których każde posiada indywidualny limit osób. Model zezwala na dodawanie w szczególnych przypadkach

blokad godzinowych uczniów e.g. regularne wizyty u lekarza w danym terminie. Zajęcia mogą odbywać się kilka razy w tygodniu.

Studenci otrzymują dostęp do gotowego planu oraz pulę punktów preferencji. Na każdy przedmiot można przeznaczyć maksymalnie 20 punktów, natomiast pojedyncze zajęcia mogą otrzymać maksymalnie 8 punktów. Im więcej punktów jest przypisanych do zajęć, tym bardziej zainteresowany jest nimi student. "Spełnione punkty preferencji" definiujemy jako punkty, które student przydzielił do danych, przypisanych mu wcześniej zajęć.

Przeglądając literaturę, można zauważyć, że modele harmonogramu oraz ich wymagania różnią się, często w zależności od tego, co dany autor uznał w swojej pracy za istotne. W artykule P. Pongcharoena [20] przedstawione zostało porównanie ograniczeń przyjmowanych przez poprzednich autorów, które uwidacznia skalę występujących między nimi różnic. Ograniczenia dzielimy na dwie kategorie — twarde oraz miękkie. Twarde ograniczenia to te, które wygenerowany plan musi spełniać, aby mógł zostać uznany za poprawny. Z kolei miękkie ograniczenia nie są niezbędne, a ich istotą jest to, że wpływają na jakość rozwiązania.

Przyjęte twarde ograniczenia:

1. W każdych zajęciach może uczestniczyć wielu studentów.
2. W każdym wykładzie uczestniczy każdy student przypisany do danego przedmiotu.
3. W danych zajęciach może uczestniczyć tylko student przypisany do przedmiotu zajęć.
4. Student może mieć tylko jedno wydarzenie w tym samym czasie.
5. Student nie może być przypisany do wydarzenia odbywającego się podczas blokady zdefiniowanej przez studenta.
6. Student może mieć maksymalnie jedne zajęcia z danego przedmiotu jednego dnia.

Przyjęte miękkie ograniczenia:

1. Maksymalizacja spełnionych punktów preferencji (maksymalizacja ogólnego założenia).

Ze względu na zmniejszenie skomplikowania problemu niektóre mniej istotne ograniczenia zostały pominięte.

1. Minimalizacja niesprawiedliwości w wyborach (maksimum globalne spełnionych punktów preferencji może prowadzić do występowania wielu studentów z zerową liczbą spełnionych punktów preferencji).
2. Zajęcia nie mają przypisanych prowadzących. Oznacza to, iż nie ma gwarancji, że zajęcia odbywające się kilka razy w tygodniu będą prowadzone przez tego samego prowadzącego.

1.2 Integer linear programming

Linear programming (programowanie liniowe) jest to metoda matematycznej optymalizacji wykorzystywana do rozwiązywania problemów, w których poszukujemy najlepszego rozwiązania w ramach określonych ograniczeń liniowych. Głównym celem programowania liniowego jest maksymalizacja lub minimalizacja liniowej funkcji celu przy zachowaniu liniowych nierówności ograniczających [17]. Integer linear programming (ILP) to szczególny przypadek programowania liniowego, w którym ograniczenia da się wyrazić za pomocą nierówności, w których zmienne to liczby całkowite.[17] Rozwiązanie ogólnego programu liczb całkowitych jest trudniejsze obliczeniowo (NP-trudne) od rozwiązywania programu liniowego. ILP program może być zdefiniowany następująco:

$$\begin{array}{ll} \text{Zmaksymalizuj wartość} & \mathrm{c^T x} \\ \text{pośród wszystkich wektorów } \mathrm{x} \in \mathbb{Z}^{\mathrm{n}} \text{ spełniających} & \mathrm{Ax} \leq \mathrm{b} \end{array}$$

gdzie A jest daną macierzą liczb całkowitych $\mathrm{m} \times \mathrm{n}$, natomiast $\mathrm{c} \in \mathbb{Z}^{\mathrm{n}}, \mathrm{b} \in \mathbb{Z}^{\mathrm{m}}$ są danymi wektorami. Funkcję, którą maksymalizujemy, nazywamy funkcją celu i możemy ją rozwinąć do $\mathrm{c^T x} = \mathrm{c_1 x_1} + ... + \mathrm{c_n x_n}$.

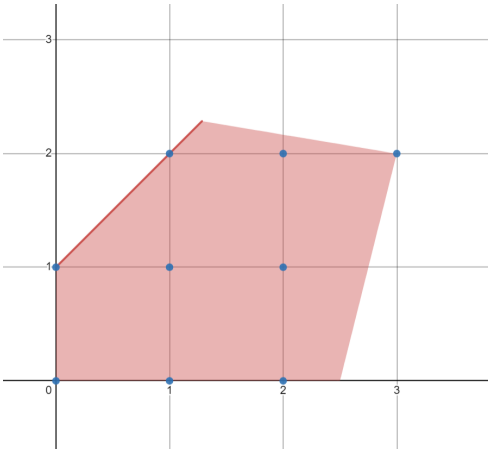
Każdy wektor $\mathrm{x} \in \mathbb{Z}^{\mathrm{n}}$ spełniający wszystkie ograniczenia jest dopuszczalnym rozwiązaniem, natomiast ten, który daje maksymalna możliwą wartość $\mathrm{c^T x}$, nazywany jest optymalną wartością. Całkowitoliczbowy program liniowy może w ogólności mieć zero, jedno lub nieskończenie dużo optymalnych rozwiązań.

Przykładowy problem ILP może wyglądać następująco:

$$\begin{array}{ll} \text{Zmaksymalizuj wartość} & \mathrm{x_1} + \mathrm{x_2} \\ \text{pośród wszystkich wektorów } (\mathrm{x_1}, \mathrm{x_2}) \in \mathbb{Z}^2 & \\ \text{spełniających} & \mathrm{x_1} \geq 0 \\ & \mathrm{x_2} \geq 0 \\ & \mathrm{x_2} - \mathrm{x_1} \leq 1 \\ & \mathrm{x_1} + 6\mathrm{x_2} \leq 15 \\ & 4\mathrm{x_1} - \mathrm{x_2} \leq 10. \end{array}$$

Problem ten możemy zwizualizować na rysunku 1, rysując wielokąt, będący częścią wspólną półpłaszczyzn zdefiniowanych poprzez nierówności. W tym przypadku rozwiązaniem jest punkt (4, 3).

Na rynku dostępnych jest wiele programów rozwiązujących problem ILP, między innymi CBC [8], FICO XPRESS [9], GUROBI [11], IBM ILOG CPLEX [16] oraz SCIP [25]. Wszystkie stosują, opartą na programowaniu liniowym, metodę branch-and-bound [5]. Polega ona na usunięciu ograniczenia definiującego liczby jako całoliczbowe, co redukuje problem do programowania liniowego. Proces ten nazywamy LP relaksacją (linear programming relaxation). Po zastosowaniu relaksacji do rozwiązania problemu stosuje się jedną ze znanych metod e.g. simplex algorithm [17].Sytuacja, w której rozwiązanie składa się z liczb całkowitych, mimo tego, że nie narzucaliśmy ograniczeń, jest bardzo rzadka. Jeśli jednak występuje, udaje się nam znaleźć optymalne rozwiązanie i program się zatrzymuje.



Rysunek 1. Wielokąt przedstawiający obszar dopuszczalnych rozwiązań. Niebieskie punkty symbolizują dopuszczalne rozwiązania całkowitoliczbowe.

W przeciwnym, występującym o wiele częściej przypadku, wybieramy zmienną, która powinna być liczbą całkowitą, ale w wyniku LP relaksacji stała się ułamkiem, a następnie dodajemy takie ograniczenia, aby wykluczyć to rozwiązanie. Przyjmijmy na przykład zmienną x, której wartość po LP relaksacji to 5.7. Aby wykluczyć to rozwiązanie, możemy dodać ograniczenia $\mathrm{x} \leq 5.0$ oraz $\mathrm{x} \geq 6.0$. Zmienna x nazywana jest zmienną rozgałęziającą (branching variable) - to od niej oryginalny problem rozgałęzia się na dwa podproblemy, które zawierają po jednym z wcześniej wymienionych ograniczeń. Następnie rozwiązujemy oba podproblemy tą samą metodą, co może produkować kolejne zmienne rozgałęziające. Jeżeli oba podproblemy zwrócą optymalne rozwiązania, to wybieramy lepsze z nich i uznajemy je za optymalnego rozwiązanie oryginalnego problemu.

Zastosowanie metody rozgałęziania tworzy tzw. drzewo poszukiwań (search tree) - wygenerowane podproblemy nazywamy węzłami drzewa, a liśćmi określamy wszystkie węzły, które jeszcze się nie rozgałęziły. Jeżeli program dojdzie do momentu, w którym wszystkie liście zostaną usunięte, wtedy oryginalny problem uznajemy za rozwiązany [18]. W przypadku wielu zmiennych rozgałęzianie może produkować drzewo dużych rozmiarów. Oprogramowania umożliwiają ustawienie czasu wykonywania programu. Po jego upływie zwrócone zostanie najlepsze na ten moment rozwiązanie, które spełnia ograniczenia, ale niekoniecznie jest optymalnie. Typowa sytuacja pokazuje jednak, że najczęściej optymalne rozwiązanie jest znalezione na długo, zanim zostanie udowodniona jego optymalność poprzez sprawdzenie pozostałych liści [5].

1.2.1 Constraint programming

Constraint programming jest to bardziej ogólny niż ILP paradygmat, w którym definiujemy ogólne ograniczenia, a następnie poszukujemy spełniającego je rozwiązania. Rozpoczynamy od zdefiniowania Constraint Satisfaction Problem (CSP), czyli pary $\mathrm{CSP} = (\mathfrak{C}, \mathfrak{D})$, gdzie $\mathfrak{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_\mathrm{n}$ reprezentuje domenę ze skończoną ilością zmiennych $\mathrm{x_j} \in \mathcal{D}_\mathrm{j}, \mathrm{j} = 1, \ldots, \mathrm{n}$, oraz $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_\mathrm{m}\}$ reprezentuje skończoną ilość ograniczeń $\mathcal{C}_\mathrm{i} : \mathfrak{D} \to \{0, 1\}, \mathrm{i} = 1, \ldots, \mathrm{m}$, tzn., że dla całej domeny

zmiennych dane ograniczenie zwraca 1 lub 0 w zależności od tego, czy zostało ono spełnione, czy też nie. Zadaniem CSP jest odpowiedzenie na pytanie, czy zbiór

$$\mathrm{X_{CSP}} = \{\mathrm{x} \mid \mathrm{x} \in \mathfrak{D}, \mathfrak{C}(\mathrm{x})\}, \text{ gdzie } \mathfrak{C}(\mathrm{x}) :\Leftrightarrow \forall \mathrm{i} = 1, \ldots, \mathrm{m} : \mathcal{C}_\mathrm{i}(\mathrm{x}) = 1$$

jest niepusty poprzez e.g. znalezienie rozwiązania $\mathrm{x} \in \mathfrak{D}$ spełniającego $\mathfrak{C}(\mathrm{x})$ albo udowodnienie, że nie ma takiego rozwiązania [1].

Constraint program jest to trójka $\mathrm{CP} = (\mathfrak{C}, \mathfrak{D}, \mathrm{f})$, gdzie celem jest rozwiązanie

$$\mathrm{f}^\star = \min\{\mathrm{f}(\mathrm{x}) \mid \mathrm{x} \in \mathfrak{D}, \mathfrak{C}(\mathrm{x})\}$$

gdzie $\mathfrak{D}$ jest wcześniej zdefiniowaną domeną, $\mathfrak{C}$ zbiorem ograniczeń, natomiast $\mathrm{f} : \mathfrak{D} \to \mathbb{R}$ to funkcja celu [1]. Warto zaznaczyć, że w przypadku Constraint program (CP) nie występują restrykcje dotyczące predykatów ograniczeń, w przeciwieństwie do ILP — wcześniej zdefiniowany ILP stanowi specyficzny przypadek CP.

1.2.2 SAT

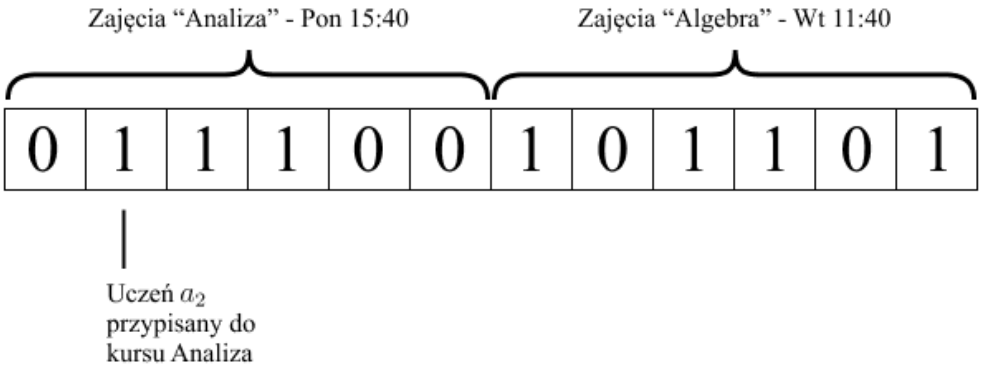
Przyjmijmy formułę logiczną $\mathfrak{C} = \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_\mathrm{m}$ na zbiorze zmiennych $\mathrm{x}_1, \mathrm{x}_2, ..., \mathrm{X}_\mathrm{n}$. Klauzulę definiujemy jako $\mathcal{C}_\mathrm{i} = \ell_1^\mathrm{i} \vee \ldots \vee \ell_{\mathrm{k_i}}^\mathrm{i}$, gdzie literał $\ell \in \mathrm{L} = \{\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n}, \bar{\mathrm{x}}_1, \ldots, \bar{\mathrm{x}}_\mathrm{n}\}$ jest albo zmienną $\mathrm{x_j}$, albo jej zaprzeczeniem $\bar{\mathrm{j}}_1$ dla danego j. Satisfiability problem (SAT) polega na znalezieniu takich wartości dla zmiennych $\mathrm{x_n}$, gdzie $\mathrm{x_j} \in \{0, 1\}$, aby $\mathfrak{C}$ było równe 1 lub udowodnieniu, że nie ma takiej kombinacji wartości zmiennych [1].

SAT jest specyficznym przypadkiem CSP, co oznacza, że możemy rozwiązać problem zdefiniowany poprzez SAT za pomocą solvera do CP. Istnieją specjalne solvery, które specjalizują się w problemach ograniczonych SAT, e.g. Google OR-Tools CP-SAT Solver [10]

CP-SAT solvery lepiej sprawdzają się w rozwiązywaniu problemów, których ograniczenia można zdefiniować jako relacje między zmiennymi logicznymi [27]. W naszym przypadku jest to możliwe, jeżeli przyjmiemy zmienną logiczną $\mathrm{x_{n,m}}$ jako odpowiedz na pytanie, czy uczeń n może uczestniczyć w zajęciach m. W sekcji implementacyjnej porównamy oba te podejścia.

1.3 Algorytmy genetyczne

Algorytm genetyczny, Genetic algorithm (GA) jest to procedura optymalizacyjna wzorowana na naturalnej teorii ewolucji. Polega na poszukiwaniu najlepszych rozwiązań, bazując na biologicznych zasadach selekcji, reprodukcji i mutacji. [28] Algorytm ten inspirowany jest ewolucyjną zasadą przeżycia osobników najlepiej do tego przystosowanych. Potencjalne rozwiązanie danego problemu kodowane jest na wzorowanej na chromosomach strukturze danych, nazywanej genotypem. Pojedyncze elementy genotypu nazywamy genem [29]. Struktura ta najczęściej jest reprezentowana jako ciąg bitów.



Rysunek 2. Przykładowa reprezentacja genotypu dla 2 kursów oraz 6 uczniów. Finalnie zastosowana reprezentacja jest opisana w sekcji implementacyjnej.

Implementację algorytmu genetycznego rozpoczynamy od populacji chromosomów, najczęściej wygenerowanych losowo. Każdy osobnik jest ewaluowany przez funkcje celu i na tej podstawie dokonywana jest selekcja osobników, następnie określane są prawdopodobieństwa reprodukcji poszczególnych osobników. Oznacza to, że osobnik, który stanowi lepsze rozwiązanie problemu, ma większe szanse na reprodukcję od innych, którzy reprezentują gorsze rozwiązanie. [29] Najbardziej popularne metody selekcji to: metoda ruletki, turniejowa oraz rankingowa.

Następnie wybrane osobniki poddawane są operatorom ewolucyjnym: operatorowi krzyżowania oraz operatorowi mutacji. Krzyżowanie polega na wymianie fragmentów genotypu między dwoma osobnikami i odbywa się zgodnie z przyjętym wcześniej prawdopodobieństwem.

Mutacja natomiast polega na zmianie wartości poszczególnych genów na przeciwną również zgodnie z kolejnym ustalonym prawdopodobieństwem.

Nowo utworzone osobniki wchodzą w skład populacji, z której te najsłabsze są z niej usuwane. Następnie sprawdzamy, czy osiągnięto cel tj. czy znaleziono dostatecznie dobre rozwiązanie. Jeśli tak, to algorytm zakańcza swoją pracę, a najlepszy osobnik stanowi rozwiązanie. W przeciwnym wypadku algorytm wraca do kroku selekcji.

Warto pamiętać, że algorytmy genetycznie nie gwarantują wyprodukowania rozwiązania optymalnego. Może się to oczywiście wydarzyć, ale w większości przypadków rozwiązanie znajduje się w sąsiedztwie rozwiązania optymalnego [21].

1.3.1 Funkcja celu

Funkcja celu jest to najistotniejszy fragment algorytmu, ponieważ to od niej zależy ocena chromosomów podczas ewolucji. Jest to miara przystosowania danego osobnika w populacji. Źle dobrana może prowadzić do promowania niepoprawnych osobników.

Wcześniej opisano ograniczenia twarde oraz miękkie. W naszym problemie ograniczeniem miękkim jest maksymalizacja preferencji. Wystarczy więc policzyć spełnione punkty preferencji, czyli punkty,

które studenci przydzielili do otrzymanych terminów i zsumować je, aby dokonać ewaluacji — im wyższa otrzymana wartość, tym lepszy jest osobnik. Sprawa staje się trudniejsza w przypadku twardych ograniczeń, gdyż algorytm podczas krzyżowania nie zwraca uwagi na to, czy harmonogram reprezentujący osobnika jest możliwy do utworzenia. Aby temu zapobiec, można przyjąć jedną z kilku strategii [19] [23]:

- Utworzenie reprezentacji osobnika, która eliminuje możliwość wystąpienia niepoprawnych osobników — bardzo trudne do znalezienia
- eliminacja osobników, które łamią twarde ograniczenia — może powodować utratę cennych informacji (osobnik uznany w danym momencie za niepoprawnego może w kolejnych iteracjach wyprodukować lepszego osobnika od tego, który obecnie spełnia dane ograniczenia).
- Naprawa osobników łamiących zdefiniowane ograniczenia — trudne rozwiązanie; może powodować utratę informacji.
- Dobranie funkcji celu, która przydziela mniejsze wartości osobnikom łamiącym twarde ograniczenia — wymaga dobrego dopasowania tak, aby "kary" za ograniczenia nie były zbyt duże, co spowodowałoby utratę informacji, ani zbyt małe, co groziłoby promowaniu niepoprawnych osobników.

1.3.2 Elitaryzm

Elitaryzm polega na automatycznym przenoszeniu do nowej populacji najlepszych osobników. Dzięki temu algorytm ma pewność, że w nowej populacji znajdą się przynajmniej niektóre chromosomy o wysokim przystosowaniu. Krok ten jest realizowany przed krzyżowaniem i zapobiega degradacji populacji. Algorytmy genetyczne, które stosują elitaryzm, średnio radzą sobie lepiej od tych, które nie stosują tej strategii [3]. Istnieje kilka sposobów implementacji elitaryzmu. Jednym z najprostszych jest wybranie określonej liczby najlepszych chromosomów z poprzedniej populacji i ich przeniesienie. Innym sposobem jest wybranie wszystkich tych chromosomów z poprzedniej populacji, które mają przystosowanie wyższe lub równe pewnemu ustalonemu progowi.

1.3.3 Elite Local Search

Elite local search (rozumiane w kontekście algorytmu genetycznego) jest to hybrydowe podejście, które łączy ze sobą globalną eksplorację GA i lokalne udoskonalanie najlepszych osobników. Po reprodukcji wybieramy pewien procent najlepszych osobników, których określamy jako osobniki elitarne, a następnie dla każdego z nich dokonujemy lokalnego poszukiwania w celu znalezienia lepszych osobników. Najlepsze znalezione w ten sposób jednostki zastępują oryginalnych elitarnych osobników.

Lokalne poszukiwanie może się odbyć na kilka sposobów.

Najpopularniejszym i zarazem najprostszym z nich jest algorytm Hill Climbing, którego działanie przedstawię poniżej. Najpierw ustawiamy początkowego osobnika jako obecnie najlepsze rozwiązanie. Następnie tworzymy sąsiedztwo wokół danego osobnika. Może się to odbyć poprzez np. zamianę

miejscami pojedynczych genów w chromosomie lub dokonanie niewielkiej mutacji. Następnie, stosując funkcje celu, ewaluujemy wszystkich utworzonych sąsiadów. Jeżeli którykolwiek z nich jest lepszym rozwiązaniem niż obecnie najlepszy osobnik, to ustawiamy go na miejscu najlepszego osobnika. Ten proces logiczny powtarza się do momentu, w którym wybrany osobnik nie ma lepszego od siebie sąsiedztwa lub po określonej liczbie iteracji [2].

1.4 Simulated Annealing

Simulated Annealing (SA) jest to procedura optymalizacyjna wzorowana na procesie wyżarzania w metalurgii, czyli technice polegającej na ogrzewaniu i kontrolowanym chłodzeniu materiału w celu zmiany jego właściwości fizycznych. W SA, podobnie jak w procesie wyżarzania, rozwiązanie jest stopniowo ochładzane, co sprawia, że sukcesywnie zmniejsza się prawdopodobieństwo przyjęcia przez nie gorszej wartości. W ten sposób unikamy utknięcia w lokalnym optimum, co jest częstym problemem w przypadku innych algorytmów optymalizacyjnych [14].

Podobnie jak w GA reprezentujemy osobniki w formie binarnej i oceniamy ich za pomocą funkcji celu, nazywanej tutaj energią.

Algorytm rozpoczynamy od losowo wygenerowanego osobnika, a następnie w każdej iteracji generujemy rozwiązanie w jego sąsiedztwie. Jeżeli nowe rozwiązanie jest lepsze, to zastępujemy nim obecnego osobnika. Natomiast jeżeli nowy osobnik jest gorszy od obecnego osobnika, to decydujemy się na dokonanie podmiany na podstawie prawdopodobieństwa funkcji temperatury i różnicy między wartościami funkcji celu obu osobników [6]. Funkcja skonstruowana jest w taki sposób, aby przy wyższej temperaturze wystąpiło większe prawdopodobieństwo przyjęcia nowego gorszego rozwiązania. Funkcja prawdopodobieństwa przyjęcia gorszego rozwiązania wygląda następująco:

$$\mathrm{P}(\Delta\mathrm{E},\mathrm{T}) = \mathrm{e}^{-\frac{\mathrm{E(n)}-\mathrm{E(n+1)}}{\mathrm{T}}}$$

gdzie

$\mathrm{E(n)}$ = energia obecnego osobnika
$\mathrm{E(n+1)}$ = energia nowego osobnika z sąsiedztwa
T = obecna temperatura systemu

natomiast warunek akceptacji nowego rozwiązania możemy przedstawić następująco:

$$\mathrm{C_n} = \begin{cases} \mathrm{C_{n+1}}, \text{ jeżeli } \mathrm{P}(\Delta\mathrm{E},\mathrm{T}) > \mathrm{r_1} \\ \mathrm{C_n}, \text{ jeżeli } \mathrm{P}(\Delta\mathrm{E},\mathrm{T}) \leq \mathrm{r_1} \end{cases}$$

gdzie

$\mathrm{C_n}$ = obecny osobnik
$\mathrm{C_{n+1}}$ = osobnik z sąsiedztwa
$\mathrm{r_1}$ = losowa wartość z zakresu od 0 do 1

1.4.1 Schładzanie

W SA niezwykle istotny jest wybór sposobu zmniejszania temperatury. W literaturze można spotkać różne podejścia, między innymi [22] [24]:

- Liniowe schładzanie — najczęściej wykorzystywane, temperatura zmniejsza się o tę samą stałą wartość.
- Wykładnicze schładzanie — powoduje bardzo szybki spadek temperatury, funkcja temperatury przypomina zbieżny ciąg geometryczny

  $$\mathrm{T_k} = \mathrm{T_0} \cdot \alpha^{\mathrm{k}}$$

  gdzie

  $\mathrm{T_k}$ = temperatura podczas iteracji k
  $\mathrm{T_0}$ = początkowa temperatura
  α = parametr prędkości procesu chłodzenia

  Typowe wartości α zawarte są w przedziale między 0.8, a 0.99 [30].
- Logarytmiczne schładzanie — przyjmuje następującą formę:

  $$\mathrm{T_k} = \frac{\alpha \cdot \mathrm{T_0}}{\ln(1 + \mathrm{k})}$$

  Podejście to gwarantuje osiągniecie globalnego minimum przy $\alpha = 1$, ale proces ten może być czasochłonny.
- Hybryda logarytmicznego schładzania z wykładniczym (PCS, Probabilistic Cooling Scheme) [24] — używamy danego podejścia z określonym prawdopodobieństwem, tzn. za każdym razem, gdy decydujemy o zmniejszeniu temperatury, wyliczamy wartości z poniższego równania

$$\mathrm{T_k} = \begin{cases} \mathrm{T_0} \cdot \alpha_{\mathrm{wyk}}^{\mathrm{k}}, \text{ jeżeli } \mathrm{PL} > \mathrm{r_2} \\ \\ \dfrac{\alpha_{\log} \cdot \mathrm{T_0}}{\ln(1 + \mathrm{k})}, \text{ jeżeli } \mathrm{PL} \leq \mathrm{r_2} \end{cases}$$

gdzie

PL = prawdopodobieństwo wykorzystania logarytmicznego schładzania
$\mathrm{r_2}$ = losowa wartość z zakresu od 0 do 1
α_{wyk} = parametr prędkości wykładniczego procesu chłodzenia
$\alpha_{\log}$ = parametr prędkości logarytmicznego procesu chłodzenia

Autor [24] za pomocą testów wykazał, że przyjęcie wartości $\mathrm{PL} = 0.3$ prowadzi do optymalnego połączenia szybkości zmniejszania temperatury wykładniczego podejścia z możliwością szczegółowej analizy logarytmicznego podejścia.

1.4.2 Temperatura początkowa

Po wyborze metody schładzania pozostaje dobrać odpowiednio parametr początkowy temperatury T_0. Jego zbyt wysoka wartość może prowadzić do za dużej ilości losowych skoków na początku procesu, natomiast zbyt mała może spowodować utratę możliwości globalnego przeszukania problemu [4].

Odpowiedni sposób wyboru wartości temperatury różni się w zależności od problemu. Jedną z podstawowych metod jest przyjęcie $\mathrm{T}_0 = \Delta\mathrm{E}_{\max}$, gdzie $\Delta\mathrm{E}_{\max}$ jest maksymalną możliwą różnicą energii między dwoma sąsiednimi rozwiązaniami. Jednak w niektórych modelach ta wartość może okazać się trudna do znalezienia.

Kolejną proponowaną metodą jest przyjęcie $\mathrm{T}_0 = \mathrm{K}\sigma_\infty^2$, gdzie K jest stałą z zakresu 5 do 10, natomiast σ_∞^2 jest wariancją dystrybucji energii, gdy temperatura wynosi ∞. σ_∞^2 jest estymowane poprzez losowe wygenerowanie n rozwiązań C_n i skorzystanie ze wzoru na wariancje $\sigma_\infty^2 = \frac{\sum_{\mathrm{i}=1}^{\mathrm{n}}(\mathrm{C_i} - \overline{\mathrm{C}})^2}{\mathrm{n}}$.

Istnieje wiele innych metod opisanych przez B. Ameur [4], jednakże na potrzeby tej pracy sposób opisany powyżej wydaje się satysfakcjonujący.

1.5 Particle swarm optimization

Particle swarm optimization (PSO) jest to kolejna procedura optymalizacyjna wzorowana na zachowaniu grupa zwierząt występujących w naturze, e.g. stado ptaków. Potencjalne rozwiązanie jest reprezentowane poprzez pojedynczą cząsteczkę, która posiada współrzędne $\mathrm{x_i}$, oraz tempo zmian, inaczej prędkość $\mathrm{v_i}$. Każda cząsteczka zapamiętuje również swoją dotychczas najlepszą pozycję $\mathrm{x_{local\_best,i}}$ [13]. Iteracja rozpoczyna się od ewaluacji danej cząsteczki funkcją celu, a następnie wykonywane jest stochastyczne dostosowanie prędkości w kierunku poprzedniej najlepszej pozycji danej cząsteczki oraz najlepszej pozycji pośród wszystkich cząsteczek według formuły:

$$\mathrm{v_{i,j}(t+1)} = \mathrm{w} \cdot \mathrm{v_{i,j}(t)} + \mathrm{c_1} \cdot \mathrm{r_1} \cdot \left(\mathrm{x_{local\_best,i,j}} - \mathrm{x_{i,j}(t)}\right) + \mathrm{c_2} \cdot \mathrm{r_2} \cdot \left(\mathrm{x_{global\_best,j}} - \mathrm{x_{i,j}(t)}\right)$$
$$\mathrm{x_{i,j}(t+1)} = \mathrm{x_{i,j}(t)} + \mathrm{v_{i,j}(t+1)},$$

, gdzie

$\mathrm{v_{i,j}(t)}$ = wartość prędkości cząsteczki i na pozycji j w iteracji t
$\mathrm{x_{i,j}(t)}$ = wartość współrzędnej cząsteczki i na pozycji j w iteracji t
$\mathrm{x_{local\_best,i,j}}$ = wartość najlepszej poprzedniej współrzędnej osiągnietej przez cząsteczke i na pozycji j w iteracji t
$\mathrm{x_{global\_best,j}}$ = wartość najlepszej współrzędnej pośród wszystkich cząsteczek na pozycji j w iteracji t
w = Parametr współczynnika bezwładności (Inertia coefficient)
$\mathrm{c_1}$ = Parametr współczynnika poznawczego (Cognitive coefficient)
$\mathrm{c_2}$ = Parametr współczynnika społecznego (Social coefficient)
$\mathrm{r_1, r_2}$ = losowe wartość z zakresu od 0 do 1

Następnie nowa pozycja jest obliczana na podstawie formuły:

$$\mathrm{x_{i,j}(t+1) = x_{i,j}(t) + v_{i,j}(t+1).}$$

PSO zostało stworzone do rozwiązywania problemów ciągłych np. poszukiwania optimum funkcji. Jednak nasz problem najlepiej można przedstawić za pomocą formy binarnej opisanej we wcześniejszych akapitach. Do pracy nad takim rodzajem problemów została utworzona modyfikacja algorytmu, nazywana Binary particle swarm optimization (BPSO) [13]. Stosuje ona pojęcie wykorzystania prędkości jako prawdopodobieństwa tego, że bit pozycji j danej współrzędnej cząsteczki i przyjmuje zero lub jeden. Pozycja jest obliczana zatem poprzez formułę:

$$\mathrm{x_{i,j}(t+1)} = \begin{cases} 0 & \text{if } \mathrm{r_3} \geqslant \mathrm{S\left(v_{i,j}(t+1)\right)} \\ 1 & \text{if } \mathrm{r_4} < \mathrm{S\left(v_{i,j}(t+1)\right)} \end{cases}$$

, gdzie S jest sigmoidalną funkcją służącą do przekształcania prędkości do prawdopodobieństwa według wyrażenia:

$$\mathrm{S\left(v_{i,j}(t+1)\right)} = \frac{1}{1 + \mathrm{e}^{-\mathrm{v_{i,j}(t+1)}}}$$

natomiast $\mathrm{r_3, r_4}$ to losowe wartości z zakresu od 0 do 1.

Dla prędkości $\mathrm{v_{i,j}(t)} = 0$ prawdopodobieństwo zmiany wartości bitu to 0.5. Funkcja jest podatna na tak zwane nasycenie funkcji sigmoidalnej, które występuje, gdy wartość prędkości jest albo zbyt duża, albo zbyt mała. W takim przypadku szansa zmiany bitu zbliża się do zera. Aby temu zapobiec, stosuje się ograniczenie prędkości oznaczane jako $\mathrm{V_{max}}$, które ogranicza prędkość do przedziału $[-\mathrm{V_{max}}, \mathrm{V_{max}}]$. Przykładowa wartość $\mathrm{V_{max}} = 6$ gwarantuje prawdopodobieństwo, że bit zmieni się na przeciwną wartość, równe co najmniej 2.47%.

Współczynnik bezwładności w kontroluje wpływ poprzedniej prędkości na obecną prędkość, współczynnik poznawczy $\mathrm{c_1}$ kontroluje wpływ poprzedniej najlepszej pozycji, a współczynnik

społeczny c_2 kontroluje wpływ najlepszej pozycji wśród wszystkich osobników. Autorzy prac sugerują, aby wartości $\mathrm{w} \in [0.87, 0.42]$, natomiast $\mathrm{c}_1 = \mathrm{c}_2 > 1$ [15].

Rozdział 2

Implementacja

Algorytmy zostały zaimplementowane w języku kotlin. Kod źródłowy implementacji znajduje się na platformie GitHub [1] Utworzono dwie struktury danych przechowujące dane kolejno wydarzeń oraz studentów. Struktura danych wydarzenia `ClassSlot` zawiera: identyfikator, nazwę wydarzenia, dzień, godzinę startu oraz końca wydarzenia, a także ilość wolnych miejsc. Natomiast struktura danych studenta `Person` zawiera: identyfikator, imię, listę identyfikatorów wydarzeń, które dana osoba zablokowała, mapę preferencji, w której do identyfikatora wydarzenia przypisane są punkty, oraz mapę wymaganych wydarzeń, w której do nazwy wydarzenia przypisana jest ilość wydarzeń, do których student musi zostać przypisany.

```
data class ClassSlot(
    val id: SlotId,
    val name: String,
    val day: DayName,
    val startTime: LocalTime,
    val endTime: LocalTime,
    val seats: Int
)
```

```
data class Person(
    val id: StudentId,
    val name: String,
    val slotsToFulfill: Map<SlotName, Amount>,
    val prefersSlots: Map<SlotId, Points>,
    val blockedSlotsId: Set<Int>
)
```

Rysunek 3. Struktury danych przechowywujące dane wejściowe — zajęcia oraz wydarzenia

Dodano także interfejs `Solver`, który jest implementowany przez każdy z algorytmów. Interfejs zawiera pole `algorithm`, które zwraca wartość enum informującą o nazwie algorytmu oraz funkcję `calculateSchedule`, której argumenty to lista studentów oraz lista wydarzeń. Funkcja zwraca strukturę, która zawiera albo informacje o błędzie, który algorytm mógł napotkać, albo strukturę wynikową, która składa się ze statystyk, parametrów symulacji oraz, co najistotniejsze, listy wydarzeń z przypisanymi studentami.

```
interface Solver {
    val algorithm: Algorithm

```

[1] <Kod źródłowy - backend>

```
    fun calculateSchedule(students: List<Person>, slots: List<ClassSlot>):
    Either<Error, SolverResult>
}

enum class Algorithm {
    CP_SAT,
    ILP,
    GENETIC,
    SA,
    SWARM
}

sealed class Error(val message: String) {
    class NoViableSolution :
        Error("Provided data did not allow to create a solution that does
    not brake any hard constraints")
}
```

Listing 1. Interfejst `Solver`

Pominięto realizacje niektórych ograniczeń, które wydłużyłyby czas implementacji: obsługa zajęć odbywających się co dwa tygodnie oraz obsługa zajęć odbywających się kilka razy w tygodniu w celu przydzielenia tego samego prowadzącego i rozłożenia ich na różne dni.

2.1 Integer Linear Programming

Do realizacji ILP wykorzystano bibliotekę Google OR-TOOLS. Pozwala ona na tworzenie solverów, które znajdują rozwiązanie, korzystając z dostępnych na rynku sposobów. Algorytm rozpoczynamy od utworzenia instancji solvera SCIP, który wzorowany jest na bibliotece o tej samej nazwie. Następnie dla każdej kombinacji osoby i wydarzenia tworzymy zmienną logiczną, która reprezentuje przypisanie studenta do danych zajęć. Już na tym etapie od razu pomijamy tworzenie zmiennych dla kombinacji, które student zablokował.

```
private fun createVariablesDb(
    students: List<Person>,
    slots: List<ClassSlot>,
    model: MPSolver
): SolverVariablesDb<MPVariable> {
    val db = SolverVariablesDb<MPVariable>()
    for (person in students) {
        for (slot in slots) {
            // Do not create variables for blocked slots
            if (!person.blockedSlotsId.contains(slot.id)) {
                db.add(
                    person.id,
                    slot.id,
```

```
                    model.makeBoolVar("student" + person.id + "slot" + slot.id)
                )
            }
        }
    }
    return db
}
```

Listing 2. Tworzenie zmiennych i przechowywanie ich w strukturze `SolverVariablesDb`

Następnie tworzymy ograniczenia, które zapewniają spełnienie twardych ograniczeń za pomocą metody `makeConstraint`. Służy ona do ustawienia, ile zmiennych naraz może wystąpić w jednym ograniczeniu. Ustawiamy w niej ile zmiennych naraz w danym ograniczeniu może się znaleźć

```
private fun ensureSlotCapacity(
    slots: List<ClassSlot>,
    model: MPSolver,
    students: List<Person>,
    db: SolverVariablesDb<MPVariable>
) {
    for (slot in slots) {
        val constraint = model.makeConstraint(0.0, slot.seats.toDouble(), "")
        for (person in students) {
            val literal = db.get(person.id, slot)
            if (literal != null) {
                constraint.setCoefficient(literal, 1.0)
            }
        }
    }
}
```

Listing 3. Funkcja tworząca ograniczenia zapewniające nieprzekroczenie ilości miejsc na danych zajęciach.

Ostatnim krokiem przed uruchomieniem symulacji jest utworzenie celu solvera. Bez tego etapu solver zwróciłby pierwszy poprawny wynik.

```
private fun createHappinessObjective(
    model: MPSolver,
    students: List<Person>,
    slots: List<ClassSlot>,
    db: SolverVariablesDb<MPVariable>
): MPObjective {
    val objective = model.objective()

    for (person in students) {
        for (slot in slots) {
```

```
            val literal = db.get(person.id, slot)
            if (literal != null) {
                objective.setCoefficient(literal, person.prefersSlots.
    getOrDefault(slot.id, 0).toDouble())
            }
        }
    }

    objective.setMaximization()
    return objective
}
```

Listing 4. Funkcja tworząca cel algorytmu poprzez przypisanie współczynników dla zmiennych.

2.2 Satisfiability problem

W tym przypadku również skorzystano z biblioteki Google OR-TOOLS, tym razem jednak utworzono silnik `CpModel` przeznaczony do rozwiązywania problemów SAT. Różnica między nim a silnikiem do ILP polega na tym, że `CpModel` został stworzony głównie do operowania na literałach. Oznacza to, że nie tworzymy przedziałów ograniczonych liczbami rzeczywistymi `Double`, tylko dodajemy kolejne ograniczenia, korzystając z metod takich jak `addAtMostOne` czy `addLessOrEqual`.

```
private fun ensureSlotCapacity(
    slots: List<ClassSlot>,
    students: List<Person>,
    db: SolverVariablesDb<Literal>,
    model: CpModel
) {
    for (slot in slots) {
        val studentsToOne = ArrayList<Literal>()
        for (person in students) {
            val literal = db.get(person.id, slot)
            if (literal != null) {
                studentsToOne.add(literal)
            }
        }
        model.addLessOrEqual(LinearExpr.sum(studentsToOne.toTypedArray()),
    slot.seats.toLong())
    }
}
```

Listing 5. Funkcja tworząca ograniczenia zapewniające nieprzekroczenie ilości miejsc na danych zajęciach dla algorytmu SAT.

Cały algorytm SAT przypomina implementację ILP, dlatego wszystkie ograniczenia były tworzone analogicznie.

2.3 Algorytm genetyczny

Do realizacji algorytmu genetycznego wykorzystano bibliotekę c. Rozwiązanie reprezentowane jest przez listę bitowych chromosomów. Każdy chromosom reprezentuje jedne zajęcia i składa się z genów bitowych, z których każdy odpowiada jednemu studentowi. Gen może mieć wartość `TRUE` lub `FALSE`, co wskazuje na to, czy student jest przypisany do zajęć danego chromosomu.

```
fun createRandomStarterGenotype(
    students: List<Person>,
    slots: List<ClassSlot>
): Genotype<BitGene> {
    return Genotype.of(
        slots.map {
            BitChromosome.of(/* length = */ students.size, /* p = */ 0.5)
        }
    )
}
```

Listing 6. Metoda inicjująca losowy genotyp startowy. Parametr p symbolizuje prawdopodobieństwo przyjecia przez bit wartości TRUE

Biblioteka udostępnia klasę Engine, która przeprowadza całą symulację. Za pomocą buildera tworzymy silnik poprzez podanie:

- startowego genotypu,
- lambdy, która pozwala na ewaluację genotypu,
- rozmiaru populacji,
- selektorów,
- modyfikatorów.

```
val engine = Engine
    .builder(
        { gen -> evaluator.evaluateFromGenotype(gen) },
        starterGenotype
    )
    .populationSize(populationSize)
    .selector(CustomEliteLocalSearchSelector(localSearchParams, eliteSize))
    .alterers(CustomSwapMutator(0.5, mutationProbability))
    .build()

val (result, timeTaken) = measureTimedValue {
    engine.stream()
        .limit(maxGenerations)
        .collect(EvolutionResult.toBestEvolutionResult())
}
```

Listing 7. Fragment funkcji, w której silnik jest inicjowany, a następnie uruchamiana jest symulacja wraz z pomiarem czasu jej wykonania.

Podany modyfikator `CustomSwapMutator` jest autorską implementacją, która lepiej odpowiada na potrzeby tego problemu. Modyfikator losowo przekształca genotyp na jeden z dwóch sposobów: zmieniając losowy bit z 0 do 1 lub dokonując zamiany wartości bitów między dwoma różnymi chromosomami na tej samej pozycji, co sprawia, że dany student zostaje przeniesiony z jednych zajęć na drugie. O prawdopodobieństwie zastosowania danej strategii decyduje parametr podawany przy utworzeniu modyfikatora.

Zastosowany selektor `CustomEliteLocalSearchSelector` jest również własną implementacją, która realizuje strategie elitaryzmu wraz z elite local search. Dla elitarnych osobników przeprowadzone zostaje wtedy wyszukiwanie oparte na hill climbing. Sąsiedztwo osobnika jest generowane także przy użyciu wcześniej opisanego modyfikatora.

```
1
2
3
4     private fun ISeq<Phenotype<BitGene, C>>.mutate(generation: Long): ISeq<
    Phenotype<BitGene, C>> {
5         return mutator.alter(this, generation).population
6     }
7
8     private fun localSearch(initial: Phenotype<BitGene, C>): Phenotype<
    BitGene, C> {
9         val generation = initial.generation()
10        var bestSolution = initial
11
12        for (i in 0..<localSearchParams.searchIterations) {
13
14            val neighbourBest = ISeq.of(0..<localSearchParams.
    neighboursCount)
15                .map { bestSolution }
16                .mutate(generation)
17                .map { ph -> ph.eval { (localSearchParams.evaluate(it)) } }
18                .maxBy { it.fitness() }
19
20            if (bestSolution.fitness() < neighbourBest.fitness()) {
21                bestSolution = neighbourBest
22            }
23        }
24
25        return bestSolution
26    }
```

Listing 8. Funkcja w klasie `CustomEliteLocalSearchSelector`, która realizuje elite local search. Jest ona wywoływana dla każdego elitarnego osobnika.

Ewaluacja odbywała się za pomocą pomocniczej klasy `GenotypeCacheEvaluator`. Funkcja `evaluateWithViolations` oblicza fitness danego osobnika oraz, w zależności od parametru, zbiera informacje o tym, jakie ograniczenia zostały złamane.

```
1
2  fun evaluateWithViolations(
3      assignedSlots: List<ClassWithPeopleAssigned>,
4      addViolationDetails: Boolean = false
5  ): Pair<Int, List<String>> {
6
7      val violations = ArrayList<String>()
8      val peopleToSlots: List<Pair<Person, List<ClassSlot>>> =
       groupPeopleToSlots(assignedSlots)
9
10     var fitness = 0
11
12     fitness -= ensureSlotCapacity(assignedSlots, addViolationDetails,
       violations)
13     fitness -= ensureNoBlockedSlots(peopleToSlots, addViolationDetails,
       violations)
14     fitness -= ensureNoOverlappingSlots(peopleToSlots, addViolationDetails,
        violations)
15     fitness -= ensureStudentBeAssignedToCorrectAmountOfClasses(
       peopleToSlots, addViolationDetails, violations)
16
17     fitness += calculateHappiness(peopleToSlots)
18
19     return fitness to violations.sorted()
20 }
21
22 fun evaluateFromGenotype(
23     givenGenotype: Genotype<BitGene>
24 ): Int {
25     val e = {
26         evaluateWithViolations(
27             transformGenotype(givenGenotype),
28             false
29         ).first
30     }
31
32     return if (cacheFitness)
33         cache.getOrPut(givenGenotype) { e() }
34     else
35         e()
36 }
```

Listing 9. Funkcja odpowiedzialna za: transformację genotypu do listy przypisanych zajęć, ewaluację oraz zapamiętanie fitness osobników (w zależności od parametru cacheFitness).

```
1
2     private fun ensureSlotCapacity(
3         assignedSlots: List<ClassWithPeopleAssigned>,
4         addViolationDetails: Boolean,
5         violations: ArrayList<String>
6     ): Int {
7         var fitnessPenalty = 0
8         for (assignedSlot in assignedSlots) {
9             if (assignedSlot.classSlot.seats < assignedSlot.people.size) {
10                fitnessPenalty += violationWeight * (assignedSlot.people.
    size - assignedSlot.classSlot.seats)
11
12                if (addViolationDetails)
13                    violations.add("Too many students in class ${
    assignedSlot.classSlot}")
14            }
15        }
16        return fitnessPenalty
17    }
```

Listing 10. Funkcja obliczająca punkty odejmowane za przekroczenie ilości osob na zajęciach. Liczba osób ponad limit mnożona jest razy parametr wagi.

Solver przyjmuje następujące parametry:

- `populationSize` - liczba określająca rozmiar populacji,
- `eliteSize` - liczba określająca ile osobników elitarnych powinno zostać zachowanych oraz poddanych elite local search przy każdej generacji,
- `maxGenerations` - liczba generacji,
- `mutationProbability` - prawdopodobieństwo wystąpienia mutacji,
- `crossoverProbability` - prawdopodobieństwo wystąpienia krzyżowania,
- `violationWeight` - waga kary za każde złamane twarde ograniczenie,
- `localSearchNeighboursCount` - liczba określająca, ile osobników sąsiedztwa powinno zostać wygenerowanych w każdej iteracji podczas elite local search,
- `localSearchIterations` - liczba iteracji elite local search.
- `cacheFitness` - wartość logiczna określająca, czy wartośc fitnesu powinna być zapamiętywana.

2.4 Simulated Annealing

W tym algorytmie do przedstawienia osobników wykorzystano tę samą klasę z biblioteki `Jenetics` [12], co w przypadku algorytmu genetycznego. Klasą genotypu można łatwo operować, a skorzystanie

z niej pozwala na wykorzystanie implementacji opisanych wcześniej elementów: ewaluatora oraz mutatora.

Algorytm rozpoczyna się od wygenerowania losowego startowego osobnika oraz obliczenia początkowej temperatury.

```
1
2  private fun calculateInitialTemperature(
3      starter: Genotype<BitGene>,
4      evaluationFunction: (g: Genotype<BitGene>) -> Int
5  ): Double {
6      val allRandomFitness = (0..<n)
7          .map { starter.newInstance() }
8          .map { evaluationFunction(it) }
9
10     val mean = allRandomFitness.average()
11
12     val variance = allRandomFitness
13         .sumOf { (it - mean).pow(2) } / n
14
15     return k * variance
16 }
```

Listing 11. Funkcja obliczająca początkową temperature.

Główna część algorytmu składa się z dwóch zagnieżdżonych pętli. Pętla wewnętrzna zajmuje się mutowaniem obecnego rozwiązania. Jeżeli zmutowane rozwiązanie jest lepsze od obecnie najlepszego, to jest ono zapamiętywane. Następnie jeśli zmutowane rozwiązanie jest lepsze niż obecne, to jest ono przyjmowane na jego miejsce. W przeciwnym przypadku następuje losowanie, które decyduje o tym, czy zmutowane rozwiązanie przyjmowane jest jako obecne. Prawdopodobieństwo tego zdarzenia jest zależne od fitnessu obu osobników oraz temperatury. Pętla ta jest wykonywana $\mathrm{trialMultiplier} * \mathrm{slotsSize} * \mathrm{studentsSize}$ razy, gdzie `trialMultiplier` jest parametrem. Za każdym razem, gdy przyjmowane jest inne obecne rozwiązanie, inkrementowana jest zmienna `changes`, natomiast zmienna `trials` inkrementowana jest w każdej iteracji.

```
1  while (trials < trialMultiplier * variablesCount) {
2      iterations += 1
3      trials += 1
4
5      val neighbour = mutate(evaluator, current, iterations)
6      if (neighbour.fitness() > best.fitness())
7          best = neighbour
8
9      if (neighbour.fitness() > current.fitness()) {
10         changes += 1
11         current = neighbour
12     } else if (neighbour.fitness() < current.fitness()
```

```
        || neighbour.genotype() != current.genotype()
    ) {
        val acceptanceProbability =
            exp(-(current.fitness() - neighbour.fitness()) / temperature)

        if (acceptanceProbability > Random.nextDouble()) {
            changes += 1
            current = neighbour
        }
    }
}
```

Listing 12. Zawartość zagnieżdzonej pętli algorytmu SA.

Po zakończeniu tej pętli obliczany jest procent zmian, tzn. w ilu procentach iteracji przyjmowany był inny osobnik: $\mathrm{changesPercentage} = \mathrm{changes}/\mathrm{trials}$ Następnie na podstawie tego parametru obliczana jest temperatura. Opisana w częsci teoretycznej strategia PCS nie przyniosła oczekiwanych rezultatów. Dla danej dużej wartości temperatury wykładnicza część szybko osiągała zero, podczas gdy część logarytmiczna zwracała wysokie temperatury przez dłuższy czas po osiągnięciu zera przez część wykładniczą. Powodowało to gwałtowne skoki temperatury między 0 a wysoką wartością. W tej sytuacji zdecydowano się więc na zastosowanie przerobionego wykładniczego schładzania, w którym parametr prędkości procesu schładzania zależał od procentowej ilości zmian w iteracjach.

```
temperature *= if (
    changesPercentage >= fastReductionMinimalChangesPercentage
)
    fastCoolingFactor
else
    slowCoolingFactor
```

Listing 13. Zawartość zagnieżdzonej pętli algorytmu SA.

Na końcu zmienna `changesPercentage` jest porównywana z parametrem `freezeMinimalChangesPercentage` - jeżeli jest od niego mniejsza, to inkrementowana jest zmienna `freezeCount`. W przeciwnym przypadku `freezeCount` jest ustawiana na zero. Jeżeli `freezeCount` przekroczy parametr `freezeLimit` główna pętla się zakończy.

```

while (freezeCount < freezeLimit) {
   ...
    if (changesPercentage < freezeMinimalChangesPercentage)
        freezeCount += 1
    else
        freezeCount = 0

}
```

Listing 14. Warunek stopu głównej pętli oraz logika odpowiedzialna za jego modyfikację.

Solver przyjmuje następujące parametry:

- `freezeMinimalChangesPercentage` — procent zmian w wewnętrznej pętli, poniżej którego dana iteracja określona jest jako zamrożona. Bierze on udział w określeniu, czy zostały spełnione warunki stopu,
- `freezeLimit` — określa, ile zamrożonych iteracji musi wystąpić, aby główna pętla zakończyła działanie,
- `trialMultiplier` — wskazuje, ilokrotnie zwiększamy ilość iteracji pętli wewnętrznej
- `fastReductionMinimalChangesPercentage` - ile procent zmian musi wystąpić w wewnętrznej pętli, aby zastosowane zostało schładzanie z szybkim współczynnikiem schładzania zamiast z wolnym współczynnikiem schładzania,
- `fastCoolingFactor` — szybki współczynnik schładzania,
- `slowCoolingFactor` — wolny współczynnik schładzania,
- `bitFlipInsteadOfSwapMutationProbability` — wskazuje ile wynosi prawdopodobieństwo, że dojdzie do zamienienia jednego bitu w mutacji, a nie do zamiany dwóch bitów wartościami.

2.5 Particle swarm optimization

Z tych samych względów, co przy wyżej opisanych algorytmach, także w tym przypadku do przedstawienia osobników wykorzystano tę samą klasę z biblioteki `Jenetics` [12]. Algorytm rozpoczynany jest generacją losowych osobników, ewaluacją ich i opakowaniem w klasy reprezentujące cząsteczki. Prędkość przedstawiona jest w formie tablicy z wartościami typu `double` i jest inicjowana zerami.

```
data class BestParticle(
    val value: Genotype<BitGene>,
    val fitness: Int,
)

data class Particle(
    val value: Genotype<BitGene>,
    val fitness: Int,
    val velocity: Array<Double>,
    val bestPosition: BestParticle,
)
```

Listing 15. Klasy pomocnicze. Cząsteczka zawiera wartość w formie genotypu, a także fitness, tablice prędkości oraz najlepszą pozycje danej cząstki.

Następnie wybierany jest obecnie najlepszy osobnik i rozpoczyna się główna pętla, która wykonywana jest tyle razy, ile wskazuje parametr `iterations`. W każdej iteracji cząstki poruszają się - w przypadku, gdy wśród nich wystąpi globalnie lepszy osobnik, zostaje on zapisany.

```
var bestGenotype = particles.maxBy { it.fitness }.toBestParticle()

for (i in 0..iterations) {

    particles = particles.map { move(it, bestGenotype, evaluator) }
    val potentialBest = particles.maxBy { it.fitness }.toBestParticle()
    if (potentialBest.fitness > bestGenotype.fitness) {
        bestGenotype = potentialBest
    }
}
```

Listing 16. Pętla wykonująca iteracje algorytmu PSO.

Poruszanie realizowane jest przez funkcję `move`. Pierwszy krok to obliczenie nowej tablicy prędkości. Następnie generowana jest nowa cząsteczka: każda wartość tablicy przekształcana jest za pomocą funkcji sigmoidalnej i porównywana z losową wartością w celu wygenerowania nowej cząsteczki. Po obliczeniu fitnessu, w zależności od parametru logicznego, dokonywana jest mutacja. Na końcu procesu zostaje wybrana najlepsza lokalna cząsteczka, następnie zwracana jest gotowa struktura nowej cząsteczki.

```

private fun move(current: Particle, bestGlobal: BestParticle, evaluator:
    GenotypeCacheEvaluator): Particle {
    val newVelocity = calculateVelocity(current, bestGlobal)
    var newGenotype = newVelocity.map {
        BitGene.of(Random.nextDouble() < sigmoidFunction(it))
    }.chunked(current.value.chromosome().length())
        .map { current.value.chromosome().newInstance(ISeq.of(it)) }
        .let { Genotype.of(it) }

    var fitness = evaluator.evaluateFromGenotype(newGenotype)

    if (mutate) {
        val mutatedGenotype = mutator
            .alter(Seq.of(Phenotype.of(newGenotype, 0L)), 0)
            .population[0].genotype()
        val mutatedFitness = evaluator.evaluateFromGenotype(mutatedGenotype
    )
        if (mutatedFitness > fitness) {
            fitness = mutatedFitness
            newGenotype = mutatedGenotype
        }
    }
```

```
22
23      val newLocalBest = if (fitness > current.bestPosition.fitness)
24          BestParticle(newGenotype, fitness)
25      else
26          current.bestPosition
27
28      return Particle(
29          newGenotype,
30          fitness,
31          newVelocity,
32          newLocalBest
33      )
34  }
35
36  private fun sigmoidFunction(velocity: Double): Double {
37      return 1.0 / (1.0 + exp(-velocity))
38  }
```

Listing 17. Funkcja wykonująca ruch cząsteczki.

Nową prędkość uzyskuje się poprzez wywołanie funkcji `calculateBitVelocity` dla każdego elementu poprzedniej tablicy prędkości oraz bitu obecnej, globalnej oraz lokalnie najlepszej cząsteczki, który odpowiada danemu elementowi.

```
1
2   private fun calculateBitVelocity(
3       previousVelocity: Double,
4       currentBit: BitGene,
5       bestLocalBit: BitGene,
6       bestGlobalBit: BitGene
7   ): Double {
8       return (w * previousVelocity +
9               c1 * Random.nextDouble() *
10                  (bestLocalBit.ordinal - currentBit.ordinal) +
11              c2 * Random.nextDouble() *
12                  (bestGlobalBit.ordinal - currentBit.ordinal))
13          .coerceIn(-maxVelocity, maxVelocity)
14  }
```

Listing 18. Funkcja `calculateBitVelocity`. Wywołanie `coerceIn` powoduje ograniczenie wartości w przedziale [-maxVelocity, maxVelocity]

Solver przyjmuje następujące parametry:

- `c1` - współczynnik poznawczy,
- `c2` - współczynnik społeczny,
- `w` — współczynnik bezwładności,
- `mutate` — wartość logiczna decydująca, czy cząsteczki będą mutowane

- `maxVelocity` — minimalna oraz maksymalna wartość prędkości
- `population` — rozmiar populacji
- `iterations` — liczba iteracji symulacji

2.6 Platforma testowa

Wszystkie algorytmy znajdują się na serwerze REST API. Przetwarzanie zapytań odbywa się za pomocą biblioteki Spring Boot, która została wybrana ze względu na jej wcześniejszą znajomość i jej obecną popularność na rynku. Zapytania są obsługiwane za pomocą adnotacji `@PostMapping` w klasie kontrolera. Serwer posiada jeden węzeł `/generate`, który obsługuje generowanie planu według wszystkich dostępnych algorytmów, przechowywanych w zapisanej na sztywno liście. Platforma nie umożliwia ani podawania parametrów w zapytaniu, ani wskazania, które konkretnie algorytmy mają zostać wywołane, dlatego w trakcie testów odpowiednie elementy były komentowane lub zmieniane bezpośrednio w kodzie serwera.

```
1
2  @RestController
3  class Controller {
4
5      val solvers = listOf(
6          SwarmOptimizationSolver(),
7          SimulatedAnnealing(),
8          GeneticSolver(),
9          CpSatSolver(),
10         ILPSolver(),
11     )
12
13     @CrossOrigin
14     @PostMapping("/generate")
15     fun generateSchedule(@RequestBody request: GenerationRequest):
       ResponseEntity<GenerationResults> {
16
17         return solvers
18             .map {
19                 it.algorithm to
20                     it.calculateSchedule(request.students, request.slots)
21             }.map { (algorithm, result) ->
22                 result.fold({ error ->
23                     mapToError(algorithm, error)
24                 }, { solverResult ->
25                     mapToCorrectResult(algorithm, solverResult)
26                 })
27             }.let { GenerationResults(it) }
28             .let { ResponseEntity.ok(it) }
29     }
30 }
```

Listing 19. Klasa `Controller` odpowiedzialna za przetwarzanie zapytań generujących plan

Interfejs graficzny do generowania planu został zrealizowany w formie aplikacji webowej. Została ona zbudowana przy użyciu biblioteki React w połączeniu z językiem TypeScript. Prosta aplikacja pozwala na: wczytanie planu z pliku w formie json, wyświetlenie go, wywołanie zapytania generacji planu oraz wyświetlenie wyników.

W lewym dolnym rogu każdego wydarzenia znajduje się liczba oznaczająca wskaźnik pożądania danego terminu (liczba zero jest pomijana). Jest ona obliczana poprzez: znalezienie wszystkich studentów, którzy przypisali jakiekolwiek punkty preferencji na dane zajęcia, zsumowanie przydzielonych punktów i podzielenie otrzymanej sumy przez ilość miejsc oraz 8, czyli maksymalną ilość punktów, którą można przeznaczyć na dane wydarzenie.

Rysunek 4. Przykładowy plan zajęć oraz szczegóły studentów. Wykłady oznaczone są liniami w paski.

Podczas wczytywania pliku następuje walidacja danych pod kątem spełnienia odpowiedniego formatu, który jest zdefiniowany poprzez typ `Dataset`. Wykłady różnią się od innych typów zajęć (e.g. laboratoria, ćwiczenia) brakiem znaku "–" w nazwie.

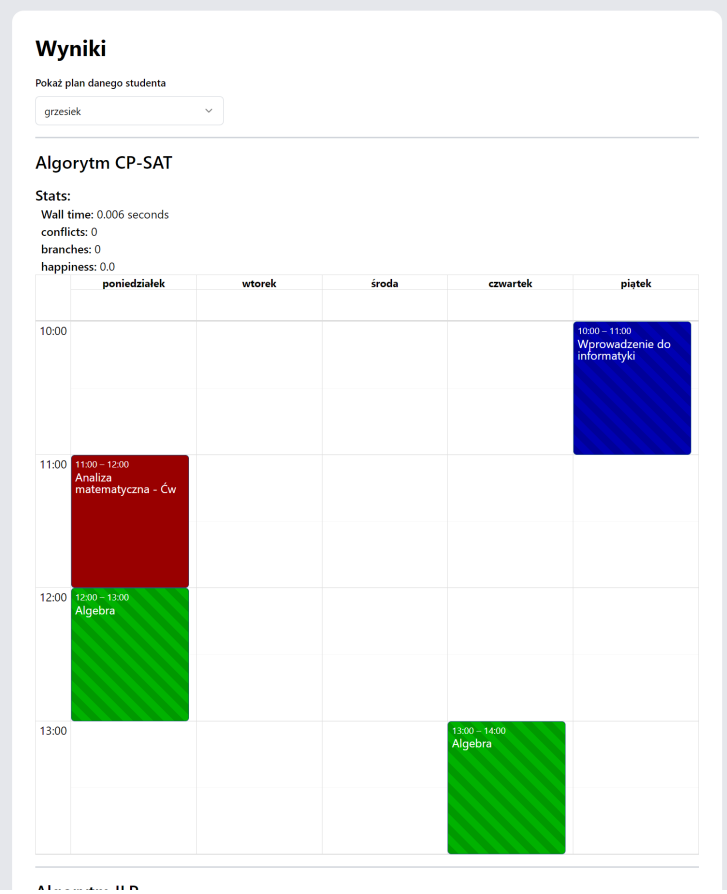
```
1
2 export type Slot = {
3     name: string;
```

```
    startTime: string;
    id: number;
    endTime: string;
    day: DayName,
    seats: number
}

export type Student = {
    blockedSlotsId: number[];
    name: string;
    id: number;
    prefersSlots: Record<string, number>
    slotsToFulfill: Record<string, number>
}

export type Dataset = { slots: Slot[]; students: Student[] };
```

Listing 20. Typy określające format danych wejściowych symulacji.

Sekcja wyników pozwala na dogłębną analizę wygenerowanego planu. Rozwijane pole umożliwia wybór studenta, którego plan chcemy zobaczyć.

Naciśnięcie na wydarzenie w sekcji danych wejściowych oraz wyników spowoduje wyświetlenie szczegółów danego wydarzenia.

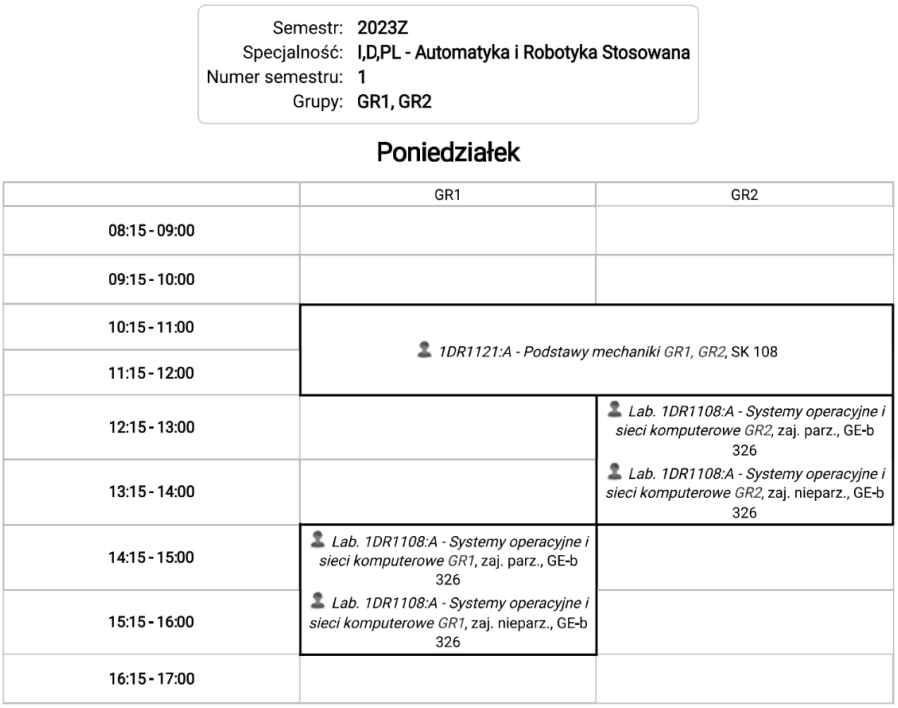


Rysunek 5. Rezultat generacji przykładowego planu. Obecnie wyświetlony jest plan studenta o imieniu "Grzesiek".

Rozdział 3

Testy

Do przeprowadzenia testów wykorzystano plan pierwszego semestru Automatyki i Robotyki Stosowanej 2023.Plan został ułożony dla dwóch grup po dwadzieścia pięć osób. Członkowie grup są przydzieleni do różnych laboratoriów oraz ćwiczeń, natomiast na wykłady uczęszczają wszyscy studenci. Przy tworzeniu danych wejściowych symulacji pominięto pojęcie grup, gdyż głównym zadaniem systemu jest oddanie wyboru danych zajęć w ręce studentów. Ze względu na ograniczenie nakładu pracy zdecydowano, że pominięta zostanie implementacja zajęć odbywających się co dwa tygodnie. Zamiast tego w planie pojawiły się dwa zajęcia, które mają 13 wolnych miejsc, odbywające się w tym samym czasie.



Semestr: **2023Z**
Specjalność: **I,D,PL - Automatyka i Robotyka Stosowana**
Numer semestru: **1**
Grupy: **GR1, GR2**

Poniedziałek

	GR1	GR2
08:15 - 09:00		
09:15 - 10:00		
10:15 - 11:00	*1DR1121:A - Podstawy mechaniki GR1, GR2*, SK 108	
11:15 - 12:00		
12:15 - 13:00		*Lab. 1DR1108:A - Systemy operacyjne i sieci komputerowe GR2*, zaj. parz., GE-b 326
13:15 - 14:00		*Lab. 1DR1108:A - Systemy operacyjne i sieci komputerowe GR2*, zaj. nieparz., GE-b 326
14:15 - 15:00	*Lab. 1DR1108:A - Systemy operacyjne i sieci komputerowe GR1*, zaj. parz., GE-b 326	
15:15 - 16:00	*Lab. 1DR1108:A - Systemy operacyjne i sieci komputerowe GR1*, zaj. nieparz., GE-b 326	
16:15 - 17:00		

Rysunek 6. Pierwsza strona planu przedstawiająca plan zajęć w poniedziałek.

Plan został ręcznie przepisany w odpowiednim formacie. Następnie prostym programem wygenerowano 50 studentów. Każdemu studentowi przypisano te same zajęcia i wykłady do zrealizowania. Ponadto każdemu ze studentów losowo przypisane zostały preferencje dotyczące wydarzeń, a prawdopodobieństwo wybrania danych zajęć określała bliskość do godziny 11. Wartość preferencji również była losowo generowana z przedziału $[5, 8]$. Nie generowano preferencji do wykładów.

```
"slotsToFulfill": {
    "Podstawy Mechaniki": 1,
    "Podstawy programowania": 1,
    "Algebra liniowa z geometria": 1,
    "Podstawy teorii mnogosci i matematyki dyskretnej": 1,
    "Analiza matematyczna": 2,
    "Fizyka": 1,
    "Systemy operacyjne i sieci komputerowe": 1,
    "Algebra liniowa z geometria - Ćw": 1,
    "Analiza matematyczna - Ćw": 1,
    "Fizyka - Lab": 1,
    "Podstawy Mechaniki - Ćw": 1,
    "Podstawy programowania - Lab": 1,
    "Podstawy programowania - Proj": 2,
    "Systemy operacyjne i sieci komputerowe - Lab": 1
  }
```

Listing 21. Przedmioty do zrealizowania przez każdego studenta.

Gotowe dane wejściowe zostały przedstawione na Rysunku 7.

3.1 Wyniki

Dla każdego z algorytmów uruchomiono symulację. Algorytm genetyczny, SA oraz PSO posiadają dużą ilość parametrów, które można modyfikować. W tabeli nr 1 przedstawiono najlepsze wyniki, jakie udało się uzyskać dla każdego z algorytmów. Przedstawione rozwiązania nie łamią żadnych ograniczeń.

Algorytm	ILP	SAT	Genetyczny	SA	PSO
Czas	0.06 sek	0.14 sek	3 min 7 sek	53 min 32 sek	34 min 58 sek
Zadowolenie	1851 (max)	1851 (max)	1194	1413	1337

Tabela 1. Wyniki najlepszych symulacji dla wszystkich algorytmów.

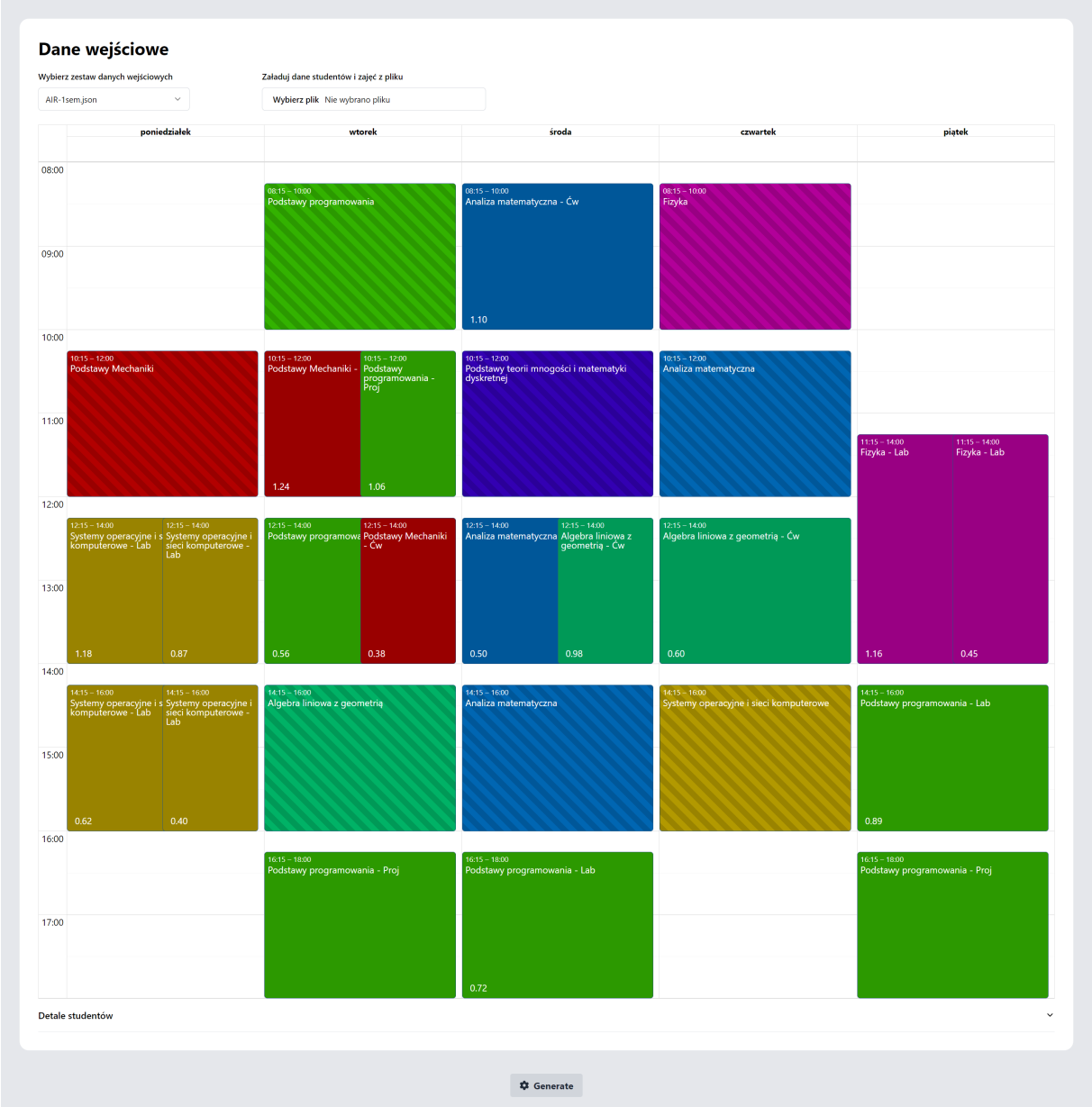
Przeprowadzono kilka symulacji dla algorytmu: genetycznego, SA oraz PSO. W każdej symulacji modyfikowano parametry. Wartości tych parametrów, które dały najlepsze rezultaty przedstawiono w tabelach nr 2, 3, 4.

Parametr	Najlepsza	Notatki
`populationSize`	400	Większa populacja znacznie spowalniała działanie bez poprawy symulacji.
`eliteSize`	6	Wartość w okolicach 1-2% populacji pozwalała na większą eksplorację algorytmu przy zachowywaniu najlepszych rozwiązań.
`maxGenerations`	1000	Większa liczba iteracji nie powodowała poprawy wyniku.
`mutationProbability`	90%	Przewaga mutacji prowadziła do lepszych rezultatów.
`crossoverProbability`	10%	Rozmnażanie na ogół wytwarzało niepoprawne rozwiązania.
`violationWeight`	80	Najlepsza wartość to 10-krotność maksymalnej liczby punktów, które student może przydzielić na zajęcia. Zapewnia to właściwy balans między dyskryminacją niepoprawnych rozwiązań a umożliwieniem eksploracji.
`localSearchNeighboursCount`	20	Większa liczba znacząco wydłużała czas symulacji bez poprawy wyniku.
`localSearchIterations`	40	Większa liczba znacząco wydłużała czas symulacji bez poprawy wyniku.
`cacheFitness`	true	Zapamiętywanie fitnessu poprawiało czas symulacji, prawdopodobnie ze względu na częste generowanie podobnych osobników podczas mutacji.

Tabela 2. Tabela przedstawiająca wartości parametrów wykorzystanych podczas generowania najlepszego rozwiązania za pomocą algorytmu genetycznego.

Parametr	Najlepsza	Notatki
`freezeMinimalChangesPercentage`	10%	Wartość na poziomie 20% procent przyniosło pożądane efekty, natomiast przy 10% przeszukiwanie zajęło więcej czasu i doprowadziło do lepszego wyniku. przypadku mniejszej wartości parametru symulacja trwała bardzo długo i nie przynosiła poprawy rozwiązania.
`freezeLimit`	300	Większe wartości nie poprawiały znacznie wyniku, podczas gdy wartość parametru miedzy 100-300 przynosiła zadowalające rezultaty.
`trialMultiplier`	10	Większa wartość bardzo wydłuża działanie programu. Testowano wartości od 0.1 do 10.
`fastReductionMinimalChangesPercentage`	70%	Wartość na tym poziomie pozwalała na szybsze schładzanie przy wysokich temperaturach, ale umożliwiała jedynie wolniejsze przy niskich.
`fastCoolingFactor`	0.8	Początek przedziału rekomendowanego w wykładniczym schładzaniu.
`slowCoolingFactor`	0.99	Koniec przedziału rekomendowanego w wykładniczym schładzaniu.
`bitFlipInsteadOfSwapMutationProbability`	50%	Obie metody mutacji działały skutecznie, więc używano ich naprzemiennie.

Tabela 3. Tabela przedstawiająca wartości parametrów wykorzystanych podczas generowania najlepszego rozwiązania za pomocą SA.



Rysunek 7. Cały plan zajęć w sekcji danych wejściowych. Najbardziej popularnymi wśród studentów zajęciami były ćwiczenia podstawy mechaniki odbywające się we wtorek o 10:15, na które preferencje przypisało 38 uczniów.

Parametr	Najlepsza	Notatki
`c1`	3.4	Testowano wartości między 1.0 a 5.4, ta wartość przyniosła najlepszy wynik. Wartości poniżej 1.0 prowadziły do słabych rezultatów.
`c2`	3.4	Zaleca się, aby ta wartość była równa `c1` w celu różnego traktowania lokalnej i globalnej najlepszej cząsteczki
`w`	1.2	Autorzy rekomendują wartości z przedziału $[0.87, 0.42]$, ale w testowanych przypadkach ich zastosowanie skutkowało małym wpływem poprzedniej prędkości. Prowadziło to do za dużej zmiany prędkości oraz otrzymania znacznie innych od wcześniejszych cząsteczek. Skutkowało to otrzymaniem słabych rozwiązań.
`mutate`	true	Dodatkowe mutowanie po przemieszczeniu poprawiło wyniki.
`maxVelocity`	6.0	Wartość ta gwarantuje, że zawsze będzie co najmniej 0.25% szans na zmianę bitu na przeciwny. Mniejsze wartości powodowały duże fluktuacje, natomiast większe powodowały zastój rozwiązań.
`population`	30	Większe wartości pogarszały wynik.
`iterations`	50000	Większa wartość od 10000 nieznacznie poprawia wynik. Jednakże zastosowanie większej wartości doprowadziło do utworzenia najlepszego przy zwiększonym nakładzie czasu.

Tabela 4. Tabela przedstawiająca wartości parametrów wykorzystanych podczas generowania najlepszego rozwiązania za pomocą PSO.

Rozdział 4

Podsumowanie

Celem niniejszej pracy było porównanie rozwiązań stosowanych do optymalizacji harmonogramowania studentów przy uwzględnieniu ich preferencji. Problem zakładał przyjęcie gotowego planu oraz listy uczniów wraz z ich wymaganiami i preferencjami. Celem problemu było przypisanie uczniów do zajęć w taki sposób, aby zadowolenie uczniów było globalnie jak największe, a także spełnione były ograniczenia twarde (e.g. nieprzydzielenie studenta do zajęć odbywających się w tym samym czasie).

Ograniczenia stosowane w innych pracach na ten temat różnią się, często w zależności od tego, co dany autor uznał w swojej pracy za istotne. W tej pracy skupiono się na zestawie najważniejszych ograniczeń:

- student nie może zostać przydzielony do zajęć odbywających się w tym samym czasie,
- student nie może zostać przydzielony do zajęć, które wcześniej zablokował,
- na zajęcia nie może uczęszczać więcej osób, niż wynosi określony limit,
- studenci mogą zostać przypisani do tylu zajęć, na ile są zapisani.

Utworzono oprogramowanie pozwalające na przypisanie uczniów na 5 sposobów, przy wykorzystaniu: algorytmu Integer linear programming (ILP), algorytmu Satisfiability problem (SAT), algorytmu genetycznego, algorytmu Simulated Annealing (SA) oraz algorytmu Particle swarm optimization (PSO). Dwa pierwsze wymienione algorytmy można sklasyfikować jako algorytmy dokładne, które przeszukują całą przestrzeń, natomiast pozostałe można określić jako heurystyczne.

W algorytmie genetycznym udało się zrealizować strategię elitaryzmu oraz elite local search poprzez utworzenie własnej implementacji selektora. Algorytm przyjmował pewien procent osobników, a następnie poddawał lokalnemu przeszukaniu każdego nich w celu znalezienia lepszego rozwiązania w sąsiedztwie. W czasie przeprowadzania testów dodano logowanie, które pokazało, że strategia skutecznie i z zadowalającą częstotliwością znajdywała nowe dobre rozwiązania.

Dobrym krokiem było zdecydowanie się na użycie tego samego modelu dla wszystkich algorytmów heurystycznych. Dzięki temu mogły one współdzielić kod ewaluatora oraz własną implementację klasy przeprowadzającej mutacje, która odpowiadała na potrzeby problemu.

W PSO szczególnie interesującym zagadnieniem było dostosowanie algorytmu do operowania na binarnej reprezentacji. Zamiast klasycznego dodania wartości z tablicy prędkości do danego osobnika,

wykorzystano tę tablicę do obliczenia, ile wynosi prawdopodobieństwo tego, że w nowym osobniku gen ma przyjąć wartości 1 lub 0. Zaskakująca okazała się najlepsza wartość współczynnika bezwładności, ponieważ w literaturze często zalecane było przyjęcie wartości mniejszej niż 1, natomiast w trakcie testów najlepsze rezultaty przyniosła wartość 1.2

Przeprowadzono generacje dla planu zawierającego 26 wydarzeń (wykłady, zajęcia oraz laboratoria) oraz 50 uczniów. Każdy z uczniów miał zostać zapisany na 16 wydarzeń oraz posiadał 7 preferencji przypisanych do różnych terminów.

Algorytmy dokładne świetnie poradziły sobie z generowaniem — każdy z nich wygenerował plan w czasie nieprzekraczającym 0.2 sekund, a otrzymane przez nie wyniki gwarantowały uzyskanie planu z maksymalnym zadowoleniem, które wynosiło 1851. Natomiast algorytmy heurystyczne potrzebowały dużo czasu, aby osiągnąć wynik, który nie łamał żadnych twardych ograniczeń. Najlepszy z nich wynik, równy 1413, otrzymał algorytm SA, ale wymagało to wydłużenia czasu symulacji o 50 minut. Można przypuszczać, iż w przypadku bardziej skomplikowanych planów to algorytmy dokładne wypadłyby gorzej, ponieważ ich złożoność rośnie wykładniczo, w przeciwieństwie do algorytmów heurystycznych, które relatywnie szybko znajdują aproksymację rozwiązania.

Gotowy system spełnia swoją rolę w podstawowym zakresie. Interfejs graficzny pozwala na generowanie planu oraz analizę danych wejściowych i wyników, a także jest dopracowany graficznie.

4.1 Dalszy rozwój

Dalszy rozwój mógłby polegać na rozszerzeniu platformy o opcję tworzenia planu lub na wprowadzeniu możliwości dodawania preferencji z perspektywy danego studenta. Docelowo platforma mogłaby zostać zintegrowana z systemami uczelnianymi — umożliwiałaby wtedy studentom logowanie za pomocą uczelnianych kont, a po generacji wyniki przypisania mogłyby być eksportowane bezpośrednio do USOSa.

Algorytmy ILP oraz SAT mają ograniczenie, które powoduje, że jeżeli plan jest błędnie skonstruowany (tj. nie można wygenerować planu, który nie łamie twardych ograniczeń), to nie jest zwracany żaden wynik. Możliwe byłoby także dodanie opcji, która generowałaby poprawny plan, odrzucając minimalną ilość studentów. Pozwoliłoby to na przeprowadzenie analizy, dlaczego plan nie może zostać wygenerowany przez e.g. ucznia, który zablokował za dużo terminów przedmiotu. Należałoby wtedy zwrócić uwagę, aby blokowanie terminów przechodziło proces weryfikacji i było dostępne tylko w wyjątkowych przypadkach.

Platforma nie umożliwia aktualnie modyfikacji parametrów z aplikacji webowej. Dodane tej możliwości umożliwiłoby równoległe uruchamianie symulacji z różnymi parametrami.

Obecnie symulacje uruchamiane są sekwencyjnie — potencjalnym ulepszeniem mogłoby być więc doprowadzenie do przeprowadzania operacji równolegle.

Mimo tego, że platforma pozostawia pole do wprowadzenia ulepszeń, otrzymane rezultaty pracy są dla mnie satysfakcjonujące. Na jej potrzeby powstała aplikacja serwera, która zawiera 1730

znaczących linii kodu (bez komentarzy, pustych linii oraz konfiguracji) oraz aplikacja webowa, która zawiera 1243 znaczących linii kodu.

Bibliografia

[1] Achterberg, T., *Constraint Integer Programming*, 2007. DOI: 10.14279/DEPOSITONCE-1634. adr.: https://depositonce.tu-berlin.de/handle/11303/1931.

[2] Asadzadeh, L., „A local search genetic algorithm for the job shop scheduling problem with intelligent agents", *Computers & Industrial Engineering*, t. 85, s. 376–383, lip. 2015, ISSN: 0360-8352. DOI: 10.1016/j.cie.2015.04.006. adr.: http://dx.doi.org/10.1016/j.cie.2015.04.006.

[3] Baluja, S. i Caruana, R., „Removing the Genetics from the Standard Genetic Algorithm", w *Machine Learning Proceedings 1995*. Elsevier, 1995, s. 38–46. DOI: 10.1016/b978-1-55860-377-6.50014-1. adr.: http://dx.doi.org/10.1016/b978-1-55860-377-6.50014-1.

[4] Ben-Ameur, W., „Computing the Initial Temperature of Simulated Annealing", *Computational Optimization and Applications*, t. 29, nr. 3, s. 369–385, grud. 2004, ISSN: 0926-6003. DOI: 10.1023/b:coap.0000044187.23143.bd. adr.: http://dx.doi.org/10.1023/b:coap.0000044187.23143.bd.

[5] Berthold, T., Hendela, G. i Koch, T., „The Three Phases of MIP Solving", *Optimization Methods and Software*, 2016, https://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/6160>, dostęp uzyskano 2023-11-12.

[6] Chahyadi, F., SN, A. i Kurniawan, H., „Hospital Nurse Scheduling Optimization Using Simulated Annealing and Probabilistic Cooling Scheme", *IJCCS (Indonesian Journal of Computing and Cybernetics Systems)*, t. 12, nr. 1, s. 21, sty. 2018, ISSN: 1978-1520. DOI: 10.22146/ijccs.23056. adr.: http://dx.doi.org/10.22146/ijccs.23056.

[7] Chiarandini, M., Birattari, M., Socha, K. i Rossi-Doria, O., „An effective hybrid algorithm for university course timetabling", *Journal of Scheduling*, t. 9, nr. 5, s. 403–432, paź. 2006, ISSN: 1099-1425. DOI: 10.1007/s10951-006-8495-8. adr.: http://dx.doi.org/10.1007/s10951-006-8495-8.

[8] *COIN-OR Branch-and-Cut solver*, https://github.com/coin-or/Cbc, dostęp uzyskano: 2023-12-22, 2023.

[9] *FICO Xpress Optimization | Data Optimization Software | FICO*, https://www.fico.com/en/products/fico-xpress-optimization, dostęp uzyskano: 2023-12-22, 2023.

[10] Google, *Employee Scheduling via CP-SAT Solver*, https://developers.google.com/optimization/cp/cp_solver, dostęp uzyskano: 2023-10-28, 2023.

[11] *Gurobi Optimizer - Gurobi Optimization*, https://www.gurobi.com/solutions/gurobi-optimizer/, dostęp uzyskano: 2023-12-22, 2023.

[12] *Jenetics: Java Genetic Algorithm Library*, https://jenetics.io/, dostęp uzyskano: 2024-01-15, 2024.

[13] Kennedy, J. i Eberhart, R., „A discrete binary version of the particle swarm algorithm", w *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, ser. ICSMC-97, IEEE. DOI: 10.1109/icsmc.1997.637339. adr.: http://dx.doi.org/10.1109/icsmc.1997.637339.

[14] Laarhoven, P. J. M. van i Aarts, E. H. L., *Simulated Annealing: Theory and Applications*. Springer Netherlands, 1987, ISBN: 9789401577441. DOI: 10.1007/978-94-015-7744-1. adr.: http://dx.doi.org/10.1007/978-94-015-7744-1.

[15] Lee, S., Soak, S., Oh, S., Pedrycz, W. i Jeon, M., „Modified binary particle swarm optimization", *Progress in Natural Science*, t. 18, nr. 9, s. 1161–1166, wrz. 2008, ISSN: 1002-0071. DOI: 10.1016/j.pnsc.2008.03.018. adr.: http://dx.doi.org/10.1016/j.pnsc.2008.03.018.

[16] *Mathematical program solvers - IBM CPLEX*, https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer, dostęp uzyskano: 2023-12-22, 2023.

[17] Matoušek, J. i Gärtner, B., *Understanding and Using Linear Programming*. Springer Berlin Heidelberg, 2007, ISBN: 9783540306979. DOI: 10.1007/978-3-540-30717-4. adr.: http://dx.doi.org/10.1007/978-3-540-30717-4.

[18] *Mixed-Integer Programming (MIP) – A Primer on the Basics - Gurobi Optimization*, https://www.gurobi.com/resources/mixed-integer-programming-mip-a-primer-on-the-basics/, dostęp uzyskano: 2023-11-12, 2023.

[19] Muafira, *Solving Nurse Scheduling/Rostering Problems in Python | by Muafira | Medium*, https://medium.com/@muafirathasnikt/solving-nurse-scheduling-rostering-problems-in-python-d44acc3ed74f, dostęp uzyskano: 2023-12-22, 2023.

[20] Pongcharoen, P., Promtet, W., Yenradee, P. i Hicks, C., „Stochastic Optimisation Timetabling Tool for university course scheduling", *International Journal of Production Economics*, t. 112, nr. 2, s. 903–918, kw. 2008, ISSN: 0925-5273. DOI: 10.1016/j.ijpe.2007.07.009. adr.: http://dx.doi.org/10.1016/j.ijpe.2007.07.009.

[21] Radosław, W., „Algorytmy genetyczne i ich zastosowania", *Postępy Techniki Przetwórstwa Spożywczego*, nr. 1, s. 107–110, 2008.

[22] Rosocha, L., Vernerova, S. i Verner, R., „MEDICAL STAFF SCHEDULING USING SIMULATED ANNEALING", *Quality Innovation Prosperity*, t. 19, nr. 1, lip. 2015, ISSN: 1335-1745. DOI: 10.12776/qip.v19i1.405. adr.: http://dx.doi.org/10.12776/qip.v19i1.405.

[23] S, K., Mahanty, B. i Acharyya, S., „Comparative Performance of Simulated Annealing and Genetic Algorithm in Solving Nurse Scheduling Problem", *Lecture Notes in Engineering and Computer Science*, t. 2168, mar. 2008.

[24] Samuel, R. K. i Venkumar, P., „Optimized Temperature Reduction Schedule for Simulated Annealing Algorithm", *Materials Today: Proceedings*, t. 2, nr. 4–5, s. 2576–2580, 2015, ISSN: 2214-7853. DOI: 10.1016/j.matpr.2015.07.209. adr.: http://dx.doi.org/10.1016/j.matpr.2015.07.209.

[25] *SCIP*, https://www.scipopt.org/, dostęp uzyskano: 2023-12-22, 2023.

[26] Shiau, D.-F., „A hybrid particle swarm optimization for a university course scheduling problem with flexible preferences", *Expert Systems with Applications*, t. 38, nr. 1, s. 235–248, sty. 2011, ISSN: 0957-4174. DOI: 10.1016/j.eswa.2010.06.051. adr.: http://dx.doi.org/10.1016/j.eswa.2010.06.051.

[27] *Using and Understanding ortools' CP-SAT: A Primer and Cheat Sheet*, https://github.com/d-krupke/cpsat-primer, dostęp uzyskano: 2023-12-22, 2023.

[28] Wang, Y.-Z., „Using genetic algorithm methods to solve course scheduling problems", *Expert Systems with Applications*, t. 25, nr. 1, s. 39–50, lip. 2003, ISSN: 0957-4174. DOI: 10.1016/s0957-4174(03)00004-6. adr.: http://dx.doi.org/10.1016/S0957-4174(03)00004-6.

[29] Whitley, D., „A genetic algorithm tutorial", *Statistics and Computing*, t. 4, nr. 2, czer. 1994, ISSN: 1573-1375. DOI: 10.1007/bf00175354. adr.: http://dx.doi.org/10.1007/BF00175354.

[30] Yang, X.-S., *Engineering Optimization: An Introduction with Metaheuristic Applications, Chapter 12: Simulated Annealing*, czer. 2010. DOI: 10.1002/9780470640425.ch12. adr.: http://dx.doi.org/10.1002/9780470640425.ch12.

Wykaz skrótów i symboli

BPSO Binary particle swarm optimization 19

CP Constraint program 13

CSP Constraint Satisfaction Problem 12, 13

GA Genetic algorithm 13, 15, 16

ILP Integer linear programming 11–13, 22, 24, 40, 45, 46

PSO Particle swarm optimization 18, 19, 32, 40, 44, 45, 57, 59

SA Simulated Annealing 16, 17, 30, 40, 42, 45, 46, 57, 59

SAT Satisfiability problem 13, 24, 40, 45, 46, 59

UCSP University Class Scheduling Problem 9

Spis rysunków

Spis tabel

Listings