



POLITECHNIKA ŚLĄSKA
WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI
KIERUNEK INFORMATYKA

Praca dyplomowa magisterska

Algorytm wyszukiwania z tabu do rozwiązywania problemu układania
planu zajęć

Autor: Michał Szluzy

Kierujący pracą: prof. dr hab. inż. Zbigniew Czech

Gliwice, czerwiec 2017

Streszczenie

The abstract will go here....

W tym miejscu można umieścić abstrakt pracy. W przeciwnym wypadku należy usunąć/zakomentować niniejszy fragment kodu.

Spis treści

1	Wprowadzenie	1
2	Algorytm wyszukiwania z tabu	3
2.1	Algorytmy heurystyczne i metaheurystyczne	3
2.2	Wyszukiwanie z tabu	4
2.2.1	Zasada działania algorytmu wyszukiwania z tabu	5
2.2.2	Sąsiedztwo rozwiązania	5
2.2.3	Kryteria aspiracji	7
2.2.4	Dywersyfikacji i intensyfikacja	8
3	Problem układania planu zajęć	9
3.1	Sformułowanie problemu	9
3.2	Rozmiar problemu	11
4	Format danych wejściowych	13
4.1	Budowa archiwum	13
4.2	Instancje	14
4.3	Okna czasowe	15
4.4	Zasoby	15
4.5	Zdarzenia	17
4.6	Ograniczenia	17

4.6.1	Przypisany czas	19
4.6.2	Unikanie konfliktów	19
4.6.3	Podział zdarzeń	19
4.6.4	Limit bezczynności	19
5	Implementacja	21
5.1	Implementacja algorytmu	21
5.1.1	Wczytywanie danych	21
5.1.2	Generowanie rozwiązania początkowego	22
5.1.3	Generowanie dostępnych ruchów algorytmu	23
5.1.4	Generowanie i ocenianie sąsiedztwa dla wybranego rozwiązania .	23
5.1.5	Funkcja oceny rozwiązania	24
5.2	Specyfikacja wewnętrzna	25
5.2.1	Użyte technologie i narzędzia	25
5.2.2	Opis klas i ważniejszych funkcji	26
5.3	Specyfikacja zewnętrzna	30
6	Testowanie	33
6.1	Środowisko testowe	33
6.2	Zestaw danych testowych	33
6.3	Wyniki działania aplikacji	34
6.4	Znaczenie parametru długości Tabu.	36
7	Podsumowanie	41
	Bibliografia	43

Spis rysunków

2.1	Pseudokod algorytmu Tabu search	6
4.1	Szablon archiwum XHSTT.	13
4.2	Przykład archiwum w języku XML.	14
4.3	Szablon pojedynczej instancji.	14
4.4	Składnia kategorii: <i>Times</i>	15
4.5	Przykład deklaracji czwartego okna czasowego w poniedziałek.	16
4.6	Składnia kategorii: <i>Resources</i>	16
4.7	Przykład deklaracji zasobu jako sali komputerowej.	16
4.8	Składnia kategorii: <i>Events</i>	17
4.9	Przykład deklaracji zajęć języka angielskiego.	18
4.10	Składnia kategorii: <i>Constraints</i>	18
4.11	Przykład ograniczenia: <i>Assign time constraints</i>	20
5.1	Przykład wczytywania instancji za pomocą biblioteki <i>Xml.Linq</i>	22
5.2	Funkcja <i>ResolveSimpleProblem</i> zawierająca szkielet algorytmu Tabu search.	28
5.3	Funkcja <i>SelectBestInstanceFromNeighborhood</i> generująca sąsiedztwo i wybierająca najlepszego sąsiada.	29
5.4	Interfejs użytkownika aplikacji.	31

6.1	Najlepszy otrzymany plan zajęć dla najlepszego rozwiązania.	35
6.2	Najgorszy otrzymany plan zajęć dla najlepszego rozwiązania (2 okienka).	35
6.3	Przebieg pracy algorytmu.	36
6.4	Przebieg pracy algorytmu dla parametru długości Tabu - 0.	37
6.5	Przebieg pracy algorytmu dla parametru długości Tabu - 100.	38
6.6	Przebieg pracy algorytmu dla parametru długości Tabu - 300.	38
6.7	Przebieg pracy algorytmu dla parametru długości Tabu - 500.	39
6.8	Przebieg pracy algorytmu dla parametru długości Tabu - 800.	39

Rozdział 1

Wprowadzenie

Ala ma kota i kot ma ale

Rozdział 2

Algorytm wyszukiwania z tabu

2.1 Algorytmy heurystyczne i metaheurystyczne

Optymalizacja rozwiązań wykorzystywana jest w prawie każdym aspekcie życia. Podwyższanie jakości usług, obniżanie kosztów wyrobów, czy minimalizacja zużycia surowców to bardzo popularne zagadnienia optymalizacyjne. Medycyna, logistyka, ekonomia to przykłady dziedzin, w których optymalizacja znajduje swoje zastosowanie. Optymalizacja to minimalizacja bądź maksymalizacja pewnej funkcji, zwanej często funkcją oceny, która określa jakość rozwiązania danego problemu. Znalezienie minimum lub maksimum wymaga więc wyznaczenia funkcji oceny dla każdego możliwego rozwiązania problemu i wyborze rozwiązania o najlepszej wartości funkcji oceny. Niestety często rozmiar zadania, dla którego szukamy optymalnego rozwiązania, jest tak duży, że sprawdzenie wszystkich rozwiązań, bądź zastosowanie algorytmu znajdującego najlepsze rozwiązanie nie jest możliwe ze względu na zbyt długi czas wykonania. Liczba operacji, które należy wykonać często rośnie wykładniczo wraz ze wzrostem rozmiaru problemu. Stwarza to możliwość zastosowania heurystyk.

„Terminem heurystyka (z języka greckiego *heurisko* - znajduję) określa się sposób postępowania oparty na zdobytym doświadczeniu, wykorzystaniu istniejących faktów i reguł, w celu znalezienia odpowiedzi na postawione pytanie” [1]. Algorytmy heurystyczne to takie algorytmy, które na podstawie przebiegu obliczeń i otrzymywanych wynikach, starają się znaleźć jak najlepsze rozwiązanie problemu, jednak zwykle nie jest to rozwiązanie optymalne (tj. najlepsze wśród wszystkich możliwych rozwiązań). W zamian za możliwość otrzymania nieco gorszego rozwiązania uzyskamy krótszy czas

działania algorytmu. Algorytmy heurystyczne wykorzystywane są w przypadku, gdy dokładne algorytmy są z przyczyn technicznych zbyt kosztowne, lub gdy są nieznane (np. dla problemu przewidywania pogody). Często też używa się heurystyk by „nakierować” gotowy algorytm na rozwiązanie optymalne, co w rezultacie skróci czas jego wykonania.

Algorytmy heurystyczne możemy podzielić ze względu na sposób, w jaki generowane są nowe rozwiązania:

- Algorytmy probabilistyczne - wykorzystują czynnik losowości; często kolejne rozwiązanie wybierane jest losowo z określonej puli rozwiązań. Może to doprowadzić do różnych wyników końcowych otrzymanych w kolejnych wykonaniach algorytmu.
- Algorytmy deterministyczne - nie zawierają czynnika losowego. Otrzymywane rozwiązanie jest zawsze takie same, przy każdym wykonaniu algorytmu na takich samych danych wejściowych.

W niektórych algorytmach wykorzystane są dwie heurystyki, nadrzędna i podrzędna. Pierwsza z nich steruje i uzupełnia działanie drugiej heurystyki. Takie podejście nazywane jest przez niektórych badaczy metaheurystykami [1]. Inna definicja metaheurystyki to „procesy iteracje działające zgodnie z klasyczną metodą heurystyczną, wspomagane inteligentnie przez różne koncepcje eksplorowania i eksploataowania przestrzeni rozwiązań z użyciem technik uczących. Wspomaganie to ustrukturalnia informacje w celu sprawnego znalezienia rozwiązań bliskich optymalnemu” [3]. Po raz pierwszy termin metaheurystyki został użyty przez Freda Glovera w 1986 roku jako określenie algorytmów, które nie rozwiązują bezpośrednio żadnego problemu, lecz określają w jaki sposób budować algorytmy podrzędne w celu uzyskania rozwiązania [2].

2.2 Wyszukiwanie z tabu

Przykładem algorytmu metaheurystycznego jest algorytm wyszukiwania z tabu. Algorytm ten został zaproponowany w 1977 r. kiedy to Fred Glover przedstawił pracę na temat wykorzystania pamięci krótkotrwałej i długotrwałej w przeszukiwaniu lokalnym. Pamięć krótkotrwała służyła do zapamiętywania ostatnich „ruchów” algorytmu i była modyfikowana przez kolejne jego iteracje (pamiętane były wybrane wartości wykorzystywane przez algorytm w ostatnich iteracjach). Natomiast pamięć długotrwała

miała na celu pamiętanie najbardziej atrakcyjnych rozwiązań przestrzeni poszukiwań. To właśnie w oparciu o tą zasadę, Glover zaproponował w 1986 r. algorytm *Tabu Search*. Glover jest uznawany za autora algorytmu mimo tego, że w tym samym roku Michael Hansen opublikował pracę opisującą bardzo podobną heurystykę. Na przestrzeni lat algorytm został ulepszony i aktualnie dostępnych jest wiele jego różnych wersji, np. *Probabilistic Tabu Search* lub *Reactive Tabu Search*.

2.2.1 Zasada działania algorytmu wyszukiwania z tabu

Wyszukiwanie z tabu to metaheurystyka służąca do rozwiązywania problemów optymalizacji. Algorytm oparty na tej metaheurystyce dokonuje iteracyjnego przeszukiwania przestrzeni rozwiązań, z użyciem tzw. sąsiedztwa oraz na zapamiętuje ostatnio wykonane ruchy w celu uniknięcia ich powtarzania. Wywodzi się on bezpośrednio z metody przeszukiwania lokalnego, jednak jest od niej zdecydowanie skuteczniejszy dzięki możliwości „wychodzenia” z minimów lokalnych. Podstawą tej możliwości jest zaakceptowanie gorszego aktualnego rozwiązania w celu uzyskania rozwiązania lepszego. Możliwe jest to dzięki uaktualnianiu danych tabu, czyli listy ruchów, które algorytm już wykonał, co zabezpiecza algorytm przed powrotem w obszary przestrzeni rozwiązań już przeszukane. Obecność ruchów na liście tabu jest tymczasowa, co w konsekwencji blokuje dany ruch przez określoną liczbę iteracji. Możliwe jest złamanie tej zasady, ale tylko wtedy, gdy ruch spełnia tzw. kryterium aspiracji. Warunkiem zakończenia działania algorytmu jest najczęściej wykonanie określonej liczby iteracji lub osiągnięcie satysfakcjonującego rozwiązania. Możliwe jest również monitorowanie aktualnego wyniku i, jeżeli nie ulega on poprawie przez określoną liczbę iteracji, zatrzymanie wykonania algorytmu. Pseudokod działania algorytmu przedstawiony został na rysunku 2.1.

2.2.2 Sąsiedztwo rozwiązania

Najważniejszym czynnikiem, od którego zależy sukces algorytmu jest poprawnie zdefiniowane sąsiedztwo, które będzie przeszukiwane w danej iteracji. Do sąsiedztwa powinny należeć rozwiązania różniące się w sposób nieznaczny od rozwiązania bieżącego. Jednak sąsiedztwo powinno umożliwiać algorytmowi przejście w każdy obszar przestrzeni rozwiązań. Sposób w jaki definiowane jest sąsiedztwo zależy od danego problemu i typu jego rozwiązań. Rozwiązania problemu mogą być reprezentowane np.

```
1  s := s0;
2  sBest := s;
3  tabuList := new list[];
4  while (not stoppingCondition())
5      candidateList := generateCandidates();
6      bestCandidate := null;
7      for (sCandidate in candidateList)
8          if ((not tabuList.contains(sCandidate)) and
9              (fitness(sCandidate) > fitness(bestCandidate)))
10             bestCandidate := sCandidate;
11         end;
12     end;
13     s := bestCandidate;
14     if (fitness(bestCandidate) > fitness(sBest))
15         sBest := bestCandidate;
16     end
17     tabuList.push(bestCandidate);
18     if (tabuList.size > maxTabuSize)
19         tabuList.removeFirst();
20     end;
21 end;
22 return sBest;
```

Rysunek 2.1: Pseudokod algorytmu Tabu search

przez wektory binarne, wektory liczb rzeczywistych, czy dowolne permutacje elementów zadanych zbiorów. Jeżeli na przykład rozwiązaniem będzie permutacja pewnego zbioru n elementowego, to sąsiedztwem możemy określić jedno z trzech typów przejść (ruchów) między permutacjami:

- $wstaw(x, y)$ - wstawienie elementu y na pozycję x (permutacje z powtórzeniami),
- $zamień(x, y)$ - zamiana elementów na pozycjach x i y ,
- $odwróć(x, y)$ - odwrócenie kolejności występowania elementów, począwszy od elementu o indeksie x , aż do elementu na pozycji y .

Sąsiedztwem danej permutacji będzie więc każda inna permutacja uzyskana za pomocą, wybranego na początku, sposobu modyfikacji bieżącej permutacji. Dzięki tak zdefiniowanemu sąsiedztwu możliwe jest łatwe zidentyfikowanie ruchu za pomocą pary indeksów (x, y) . Para ta zostanie zapisana na liście tabu, a ruch ten będzie zablokowany przez następne iteracje.

Może się jednak okazać, że generowane sąsiedztwa są zbyt duże, by każdorazowo przeszukiwać je w całości. Stosowane jest wtedy zawężanie sąsiedztwa. Jednym ze sposobów zawężania jest losowy dobór sąsiedztwa. Wprowadza to element probabilistyczny do algorytmu, co zmniejsza prawdopodobieństwo powstania niepożądanych cykli. Jednak przy takim podejściu możemy pominąć obszary przestrzeni rozwiązań, w których znajduje się rozwiązanie optymalne.

2.2.3 Kryteria aspiracji

Może się zdarzyć, że zablokowanie pewnych ruchów doprowadzi do stagnacji procesu przeszukiwania lub całkowicie zablokuje kolejny ruch (np. w sytuacji, gdy wszystkie możliwe ruchy są na liście tabu). Jest to możliwe, ponieważ algorytm przechowuje tylko atrybuty rozwiązań, a nie całe rozwiązania. Kryterium aspiracji umożliwia zapobieganiu takiej sytuacji. Spełnienie kryterium aspiracji pozwala na złamanie zakazu tabu, czyli wykonanie ruchu, który znajduje się na liście ostatnio wykonanych. Najpopularniejszym i najprostszym kryterium aspiracji jest uzyskanie najlepszego, nieznanego jak dotąd, wyniku. Musi być ono lepsze od aktualnie najlepszego w celu uniknięcia zapętlenia. Jednak większość kryteriów aspiracji jest bardziej skomplikowana i opiera się na wyspecjalizowanym przewidywaniu możliwości powstania cyklu po wykonaniu określonego ruchu.

2.2.4 Dywersyfikacji i intensyfikacja

Ważnym aspektem algorytmu wyszukiwania z tabu jest pamięć długoterminowa. Służy ona do przechowywania danych o wykonanych już iteracjach i do budowania statystyk. Dzięki tym statystykom można modyfikować strategię poszukiwania. Głównym celem takich modyfikacji może być spełnienie jednego z dwóch kryteriów:

- Intensyfikacja - jeżeli według statystyki, w danym obszarze znajduje się dużo dobrych rozwiązań to algorytm zagęści obszar poszukiwań. Dzięki temu istnieje szansa na znalezienie jeszcze lepszego rozwiązania w danym obszarze.
- Dywersyfikacja - jest to powiększenie obszaru poszukiwań. Najczęściej stosowanym sposobem jest nakładanie kary na ruchy, które powtarzają się w perspektywie dłuższego czasu. Efektem tego jest „przeniesienie” algorytmu w inne rejony poszukiwań, co zmniejsza szanse na pominięcie najlepszych rozwiązań.

Wykorzystanie intensyfikacji i dywersyfikacji w znacznej mierze poprawia efektywność algorytmu, dlatego kryteria te są wykorzystywane w większości nowych wersji algorytmu *Tabu Search*.

Rozdział 3

Problem układania planu zajęć

3.1 Sformułowanie problemu

Problem układania planu zajęć można sformułować następująco. Dana jest lista określonych wydarzeń, dostępnych okien czasowych i zasobów. Należy przyporządkować wydarzeniom okna czasowe oraz zasoby w taki sposób, by spełnić przyjęte założenia. W formułowanym problemie:

- wydarzeniami są poszczególne lekcje odbywające się w ciągu jednego tygodnia,
- dostępnymi oknami czasowymi są godziny w których mogą odbywać się zajęcia (np. od poniedziałku do piątku w godzinach między 8:00 a 18:00),
- zasoby to dostępne sale lekcyjne, nauczyciele prowadzący zajęcia, grupy uczniów itp.

Układanie planu zajęć wymaga spełnienia określonych ograniczeń. Mianowicie nie można wypełnić dostępnych okien czasowych według z góry przyjętej kolejności, ponieważ jest prawdopodobne, że pojawią się konflikty i plan zajęć nie będzie możliwy do zrealizowania. Aby tego uniknąć trzeba spełnić tzw. ograniczenia twarde, czyli takie, które zawsze muszą zostać spełnione, by plan mógł być możliwy do zrealizowania. Istnieją również ograniczenia miękkie, które określają jakość ułożonego planu. Ograniczenia te nie muszą zostać spełnione, ale algorytm układania planu zajęć powinien brać je pod uwagę.

Przykłady ograniczeń:

- twarde:
 - zasób nie może być wykorzystany w dwóch miejscach w tym samym czasie (np. dany nauczyciel nie może prowadzić jednocześnie zajęć w dwóch różnych salach),
 - nie występują zajęcia, które nie zostały przypisane do okien czasowych w ułożonym planie.
- miękkie:
 - brak dłuższych przerw między zajęciami dla uczniów,
 - odpowiednie przerwy między zajęciami (np. 15 - lub 20 - minutowe),
 - liczba dni roboczych, w których nauczyciele prowadzą zajęcia, powinna być minimalizowana,
 - brak bloków zajęć danego typu (np. dana grupa uczniów nie powinna mieć kolejno czterech lekcji matematyki).

Często jednak dla wyjściowej siatki zajęć spełnienie wymagań twardych jest niemożliwe. Wynika to najczęściej z za małej liczby zasobów (za mało nauczycieli mogących prowadzić jeden przedmiot lub za mało sal). W algorytmach szuka się wtedy planu z najmniejszą liczbą niespełnionych wymagań, które są usuwane ręcznie przez szukanie określonych kompromisów (np. dołożenie okna czasowego, zatrudnienie dodatkowego nauczyciela, zaplanowanie zajęć tego samego typu dla dwóch mniejszych grup w jednej sali).

Ograniczenia miękkie to w istocie życzenia nauczycieli i uczniów co do tego jak plan powinien wyglądać. Niespełnienie tych ograniczeń wpływa na końcową ocenę planu zajęć, niespełnione ograniczenia mogą mieć określone wagi w zależności od ich istotności. Niektóre plany zajęć układane są ze szczególnym uwzględnieniem uczniów (mała liczba okienek, równomierne rozłożenie zajęć, brak bloków zajęć tego samego typu, odpowiednia przerwa między zajęciami), a niektóre z uwzględnieniem potrzeb prowadzących (zajęcia skumulowane w ciągu dwóch dni tygodnia by umożliwić pracę w innej szkole). Założenia te z reguły precyzuje osoba uruchamiająca wykonanie planu przez odpowiednie skonfigurowanie parametrów algorytmu i zdefiniowanie funkcji oceny wygenerowanego planu.

Funkcja oceny planu zajęć z reguły przewiduje nakładanie punktów karnych za niespełnienie określonych ograniczeń. Każde ograniczenie ma ustaloną liczbę punktów karnych, przy czym ograniczenia twarde powinny mieć dużo większą wagę od ograniczeń miękkich. Im więcej punktów karnych ma plan tym jest gorszy. To właśnie na podstawie funkcji oceny planu większość algorytmów starta się ułożyć jak najlepszy plan zajęć.

3.2 Rozmiar problemu

By dobrze zrozumieć potrzebę używania algorytmów heurystycznych przy wyszukiwaniu najlepszego planu zajęć warto przybliżyć pojęcie skali problemu. W tym celu przedstawimy przykład planu zajęć dla amerykańskiej szkoły średniego rozmiaru. W przykładzie tym, każde ze zdarzeń ma przyporządkowane wcześniej zasoby. Przykład ten zawiera:

- okna czasowe - 40,
- nauczyciele - 27,
- sale - 30,
- grupy uczniów - 24,
- zdarzenia - 832,
- ograniczenia - 4.

Dostępnych jest 40 okien czasowych, które rozłożone na pięć dni zajęć w tygodniu dają osiem godzin zajęć dziennie. Jednocześnie odbywać się mogą 24 zajęcia. Liczba ta wynika z niepodzielności zasobów i jest minimalną liczbą spośród liczby nauczycieli, sal i grup uczniów. Maksymalna liczba zdarzeń, które mogą się odbywać w danym tygodniu wynosi więc:

$$40 \cdot 24 = 960,$$

$$960 > 832.$$

Maksymalna liczba zdarzeń jest większa od liczby zdarzeń zawartych w siatce zajęć podanego przykładu, więc realizacja planu zajęć zgodnego z powyższą specyfikacją zasobów jest możliwa.

Na jedną grupę uczniów przypada około 35 zdarzeń ($832/24$), które powinny być rozłożone na 40 okien czasowych. Uwzględniając możliwość wystąpienia okienek w trakcie zajęć, mamy więc 40-wyrazowy ciąg zdarzeń. Liczba możliwości, w które można te zajęcia rozłożyć, określa liczba permutacji $40!$, która wynosi:

$$40! = 815915283247897734345611269596115894272000000000$$

Jest to 48 cyfrowa liczba możliwości rozłożenia zajęć w oknach czasowych dla jednej grupy. Liczbę tę trzeba pomnożyć przez liczbę grup.

Liczba możliwości może być jeszcze większa, jeżeli uwzględnimy to, że każde zajęcia mogą mieć przypisanego dowolnego nauczyciela z grupy osób uprawnionych do prowadzenia przedmiotu. Również sale zajęć mogą być różne dla danego wydarzenia. By zmniejszyć tę liczbę możliwości, odpowiednio modyfikuje się dane wejściowe. Każde zajęcia (zdarzenia) w danych wejściowych, mogą mieć przypisane na stałe zasoby takie jak sala, grupa i nauczyciel. Szukaną niewiadomą jest wtedy tylko termin odbywania się zajęć. Dzięki temu przestrzeń możliwych rozwiązań dla algorytmu układania planu zajęć zmniejsza się. Dodatkowo przypadek, w którym każde zajęcia mają wcześniej przypisanego prowadzącego, ma odzwierciedlenie w potrzebach współczesnych szkół, gdzie wyspecjalizowana kadra zatrudniana jest często w niepełnym wymiarze godzin, przez to może być przydzielona tylko do wybranej ilości zajęć.

Podjęcie, w którym zasoby, takie jak sala i nauczyciel, są przypisane z góry do zdarzeń, umożliwia szybsze działanie algorytmu. Jednak trzeba pamiętać, że ma to również wpływ na wynik końcowy. Może się zdarzyć przypadek, w który nie istnieje możliwość ułożenia optymalnego planu dla danych wejściowych, a wystarczyło by zamienić nauczyciela przypisanego do jednych zajęć, z innym nauczycielem tego samego przedmiotu, by odblokować nowe możliwości. Jednak z uwagi na czas działania algorytmu, jak i na dostępne dane testowe, zdecydowano się wybrać wariant, w którym do algorytmu przekazywane są zdarzenia z przypisanymi nauczycielami, salami oraz grupami, a algorytm wyszukuje tylko odpowiednie okna czasowe dla podanych zdarzeń.

Rozdział 4

Format danych wejściowych

Problem rozwiązywania planu zajęć jest zagadnieniem popularnym w świecie naukowym. Dostępnych jest wiele rozwiązań i ciągle powstają nowe. Jednak, by móc porównać ze sobą dwa różne algorytmy, konieczne jest operowanie na tych samych danych testowych. Z tego powodu naukowcy uzgodnili wspólny format danych wejściowych o nazwie XHSTT i stworzyli bazę danych wraz z przykładowymi rozwiązaniami [6]. Dane te zapisane są z wykorzystaniem języka XML. Format XHSTT ma duży stopień skomplikowania, dlatego postanowiono poświęcić cały rozdział na jego opisanie [4].

4.1 Budowa archiwum

Archiwum jest kolekcją instancji zawierających dane dla problemu układania planu zajęć, razem z potencjalnymi grupami rozwiązań. Każda grupa rozwiązań zawiera rozwiązania dla jednej instancji z archiwum. Ogólny szablon dokumentu przedstawiono na rysunku 4.1.

W tej notacji, słowa znajdujące się w tej samej linii oznaczają, że pierwsze słowo

```
1 HighSchoolTimetableArchive +Id
2   +MetaData
3   +Instances
4     *Instance
5   +SolutionGroups
6     *SolutionGroup
```

Rysunek 4.1: Szablon archiwum XHSTT.

```
1 <HighSchoolTimetableArchive Id="MyArchive">
2   <Instances>
3     <Instance>
4       ...
5     </Instance>
6     <Instance>
7       ...
8     </Instance>
9   </Instances>
10 </HighSchoolTimetableArchive>
```

Rysunek 4.2: Przykład archiwum w języku XML.

```
1 Instance Id
2   Metadata
3   Times
4   Resources
5   Events
6   Constraints
```

Rysunek 4.3: Szablon pojedynczej instancji.

jest nazwą kategorii, a kolejne słowa to jej atrybuty. Słowa umieszczone poniżej w zagłębieniach to nazwy podkategorii. Znak + przed nazwą oznacza, że kategoria lub atrybut jest czymś opcjonalnym. Znak * oznacza, że dana kategoria może wystąpić zero lub więcej razy. Przykład w formacie XML zaprezentowano na rysunku 4.2.

Opcjonalna kategoria *Metadata* zawiera podstawowe informacje na temat archiwum, takie jak: nazwa, autor, data powstania, opis oraz uwagi.

4.2 Instancje

Instancja jest to pojedynczy zestaw danych dla danego problemu układania planu zajęć, najczęściej dla pojedynczej szkoły i konkretnego roku (lub semestru). Składnie instancji zaprezentowano na rysunku 4.3. Wiele kategorii posiada atrybut *Id*, który służy do odnoszenia się do danej kategorii w dalszej części archiwum.

1	Times
2	+TimeGroups
3	*Time
4	...
5	TimeGroups
6	*Week
7	*Day
8	*TimeGroup
9	...
10	Time Id
11	Name
12	+Week
13	+Day
14	+TimeGroups

Rysunek 4.4: Składnia kategorii: *Times*.

4.3 Okna czasowe

Format XHSTT wspiera tylko prosty model czasu, gdzie czas podzielony jest na równe interwały zwane oknami czasowymi. Kategoria *Times* służy do definiowania tych okien. Podkategoria *TimeGroups* definiuje różne grupy czasowe takie jak dni, tygodnie oraz okresy w ciągu dnia. Podkategoria *Time* zawiera definicje okna czasowego, gdzie każde okno oprócz nazwy może mieć przypisany konkretny tydzień, dzień i inne grupy czasowe. Składnia kategorii *Times* wraz z podkategoriami znajduje się na rysunku 4.4. Przykład w języku XML znajduje się na rysunku 4.5. Przedstawia on czwarte okno czasowe w poniedziałek.

4.4 Zasoby

Zasoby są to wszystkie elementy przypisane do zdarzeń. Do zasobów najczęściej zaliczamy nauczycieli, sale i grupy uczniów, ale format XHSTT dopuszcza dowolne definiowanie zasobów. W skład kategorii *Resources* wchodzi również grupy i typy zasobów. Typ zasobów to np: nauczyciel, sala, grupa. Grupy zasobów gromadzą zasoby jednego typu np: nauczyciele języka polskiego. Składnia kategorii *Resources* zaprezentowana jest na rysunku 4.6, na rysunku 4.7 zaprezentowano przykładową deklarację sali komputerowej.

```

1      <Day Id="Monday">
2          <Name>Monday</Name>
3      </Day>
4      <TimeGroup Id="BeforeLunch">
5          <Name>BeforeLunch</Name>
6      </TimeGroup>
7      <Time Id="Mon4">
8          <Name>Mon4</Name>
9          <Day Reference="Monday"/>
10         <TimeGroups>
11             <TimeGroup Reference="BeforeLunch"/>
12         </TimeGroups>
13     </Time>

```

Rysunek 4.5: Przykład deklaracji czwartego okna czasowego w poniedziałek.

```

1      Resources
2          +ResourceTypes
3          +ResourceGroups
4          *Resource
5
6      ResourceType Id
7          Name
8
9      ResourceGroup Id
10         Name
11         ResourceType
12
13     Resource Id
14         Name
15         ResourceType
16         +ResourceGroups

```

Rysunek 4.6: Składnia kategorii: *Resources*.

```

1      <Resource Id="r03">
2          <Name>r03</Name>
3          <ResourceType Reference="Room"/>
4          <ResourceGroups>
5              <ResourceGroup Reference="LargeRoom"/>
6              <ResourceGroup Reference="ComputerLab"/>
7          </ResourceGroups>
8      </Resource>

```

Rysunek 4.7: Przykład deklaracji zasobu jako sali komputerowej.

1	Events
2	+EventGroups
3	*Event
4	
5	EventGroup Id
6	Name
7	
8	Event Id +Color
9	Name
10	Duration
11	+Course
12	+Time
13	+Resources
14	+ResourceGroups
15	+EventGroups

Rysunek 4.8: Składnia kategorii: *Events*.

4.5 Zdarzenia

Zdarzenia określone jest jako spotkanie pomiędzy zasobami, czyli w uproszczeniu są to zajęcia odbywające się w konkretnej sali, z konkretnym nauczycielem i grupą studentów. Przed zdarzeniami zdefiniowane mogą być ich grupy, takie jak np. zajęcia z kursu języka obcego. Same zdarzenia zawierają atrybuty określające ich czas trwania oraz termin odbycia się zajęć. Termin odbywania zajęć może być przypisany wcześniej lub może być pozostawiony pusty, w celu przypisania go przez algorytm układający plan zajęć. Dodatkowo zdarzenia posiadają parametr określający kolor zajęć wyświetlany na ułożonym planie. Składnia kategorii *Events* zaprezentowana jest na rysunku 4.8, na rysunku 4.9 zaprezentowano przykładową deklarację zajęć z języka angielskiego.

4.6 Ograniczenia

Format XHSTT pozwala definiować wiele ograniczeń planu zajęć. Wszystkie one opisane są szczegółowo na stronie internetowej [5]. W pracy zostały opisane tylko te, które są uwzględnione w implementacji. Ogólna składnia ograniczenia przedstawiona jest na rysunku 4.10. Każde z ograniczeń ma zdefiniowaną swoją wagę oraz sposób wyliczania funkcji kosztu (liniowa, kwadratowa). Każde ograniczenie jest rozdzielone na ograniczenie twarde lub miękkie poprzez parametr *Required*.

```

1      <Event Id="Event_Ang_01">
2          <Name>Ang_01</Name>
3          <Duration>2</Duration>
4          <Course Reference="AngCourse"/>
5          <Resources>
6              <Resource Reference="OS2">
7                  <Role>Clas</Role>
8              </Resource>
9              <Resource Reference="HIL">
10                 <Role>Teacher</Role>
11             </Resource>
12             <Resource Reference="Room_22">
13                 <Role>Room</Role>
14             </Resource>
15         </Resources>
16         <EventGroups>
17             <EventGroup Reference="gr_EventsGeneral"/>
18             <EventGroup Reference="gr_EventsDuration2"/>
19         </EventGroups>
20     </Event>

```

Rysunek 4.9: Przykład deklaracji zajęć języka angielskiego.

```

1      AnyConstraint Id
2          Name
3          Required
4          Weight
5          CostFunction
6          AppliesTo

```

Rysunek 4.10: Składnia kategorii: *Constraints*.

4.6.1 Przypisany czas

Ograniczenie *Assign time constraints* określa, że każde zdarzenie musi mieć przypisany czas oraz definiuje koszt złamania tego ograniczenia. Może być przypisane do wszystkich wydarzeń lub tylko do wybranych grup. Przykład przedstawiono na rysunku 4.11.

4.6.2 Unikanie konfliktów

Ograniczenie *Avoid clashes constraints* określa, czy dane zasoby mogą być podzielone. To znaczy, czy nie są przypisane do dwóch lub więcej zdarzeń odbywających się w tym samym czasie. Jest to ograniczenie stosowane powszechnie, jednak mogą zdarzyć się przypadki, że układający dopuszcza odbywanie się dwóch zajęć w jednej sali jednocześnie (np. zajęcia wychowania fizycznego).

4.6.3 Podział zdarzeń

Ograniczenie *Split events constraints* określa, czy zdarzenia o długości dłuższej niż jeden mogą zostać podzielone i rozłożone na kilka dni. Ograniczenie to pozwala nam definiować minimalne i maksymalne długości bloków jednego zdarzenia.

4.6.4 Limit bezczynności

Ograniczenie *Limit idle times constraints* określa, czy mogą występować i jak długie są dopuszczalne przerwy między zajęciami. Określają, czy pomiędzy dwoma zajęciami jednego dnia, może wystąpić jedno lub więcej nieprzypisanych okien czasowych.

```
1 <AssignTimeConstraint Id="AssignTimes_2">
2   <Name>AssignTimes</Name>
3   <Required>true</Required>
4   <Weight>1</Weight>
5   <CostFunction>Linear</CostFunction>
6   <AppliesTo>
7     <EventGroups>
8       <EventGroup Reference="gr_EventsGeneral"/>
9     </EventGroups>
10  </AppliesTo>
11 </AssignTimeConstraint>
```

Rysunek 4.11: Przykład ograniczenia: *Assign time constraints*.

Rozdział 5

Implementacja

Problem układania planu zajęć można próbować rozwiązać za pomocą algorytmu *Tabu search*. Jednak podczas implementacji napotkać można dużą ilość przeszkód, związaną ze złożonością problemu układania planu zajęć. Rozmiar generowanego sąsiedztwa czy poziom skomplikowania funkcji oceny to tylko niektóre z nich. Poniżej omówiona została próba implementacji algorytmu oraz specyfikacja wewnętrzna i zewnętrzna powstałej aplikacji.

5.1 Implementacja algorytmu

Implementacja algorytmu napisana jest w języku C#. Język ten został wybrany, z uwagi na dużą liczbę darmowych bibliotek oraz dostępu do bardzo dobrej dokumentacji technicznej. Nie bez znaczenia pozostał też fakt dużej popularności języka, co przekłada się na większą możliwość uzyskania pomocy na forach programistycznych. Szczegółowy opis języka znajduje się w podrozdziale o specyfikacji wewnętrznej.

5.1.1 Wczytywanie danych

Pierwszym problemem, który stanął przed autorem tej pracy, było wczytywanie danych z formatu XHSTT. Format ten został szczegółowo omówiony w poprzednim rozdziale. Dane zostały zapisane w języku XML. Skorzystano więc z biblioteki systemowej *Xml.Linq*, która pozwala traktować dokument XML jako bazę danych i wysyłać do niego stosowne zapytania. Przykład wczytywania instancji za pomocą tej biblioteki

```
1 XElement inst = xdoc.Root.Elements("Instances")
2   .Elements("Instance")
3   .Where(x => (string)x.Attribute("Id") == instance.IdText)
4   .First();
```

Rysunek 5.1: Przykład wczytywania instancji za pomocą biblioteki *Xml.Linq*.

przedstawiony został na rysunku 5.1. Biblioteka *Xml.Linq* stanowiła spore ułatwienie, jednak w dalszym ciągu problemem pozostał złożony charakter dokumentu, ponieważ każde zagnieżdżenie musiało być wczytane za pomocą odpowiedniej linii kodu.

Wczytywane dane zapisywane są w bazie danych, dlatego każde kolejne uruchomienie aplikacji umożliwia pracę na już wczytanych danych.

Jak zostało powiedziane w poprzednich rozdziałach, założono, że wczytane dane zawierają zdarzenia z przypisanymi wcześniej zasobami takimi jak nauczyciel, sala i grupa. Dalsza implementacja algorytmu opiera się na tym założeniu.

5.1.2 Generowanie rozwiązania początkowego

Algorytm *Tabu search* poszukiwanie najlepszego rozwiązania musi rozpocząć od jakiegoś miejsca. Miejsce to nazywane jest rozwiązaniem początkowym. Warto przypomnieć, że algorytm *Tabu search* jest algorytmem heurystycznym. W zależności od tego, w jakim miejscu rozpocznie on poszukiwanie, może uzyskać różne rezultaty po określonej liczbie iteracji. Stworzenie początkowego planu zajęć, gdzie wszystkie zajęcia rozpoczynająby się w poniedziałek, w pierwszym dostępnym oknie czasowym, mogłoby spowodować znaczne wydłużenie czasu, w którym algorytm znajdzie optymalne rozwiązanie. Wydłużony czas, spowodowany był by bardzo złą oceną takiego planu i dłuższą drogą, jaką algorytm musi pokonać, by dojść do rozwiązania optymalnego. Zdecydowano się więc na wprowadzenie elementu probabilistycznego, by w prosty sposób zróżnicować początkowy plan zajęć. Każdemu wydarzeniu przypisywane jest losowe okno czasowe. Dobrym pomysłem było by zastosowanie prostego algorytmu, który utworzył by początkowy plan zajęć z jeszcze lepszą oceną, jednak, z uwagi na ograniczony czas implementacji, zdecydowano się pozostać przy rozwiązaniu losowym.

5.1.3 Generowanie dostępnych ruchów algorytmu

By swobodniej generować sąsiedztwo niezbędne w dalszym działaniu algorytmu, zdecydowano się na wygenerowanie wszystkich dostępnych ruchów dla jednego kroku algorytmu. Jako dostępny ruch rozumiana jest para: wydarzenie i okno czasowe. Oznacza to, że wybranemu wydarzeniu przyporządkowane jest określone okno czasowe. Generowane są więc wszystkie pary, których liczba równa jest liczbie wydarzeń pomnożonej razy liczbę okien czasowych. Tak zinterpretowany ruch, może być w łatwy sposób wykorzystany przy generacji sąsiedztwa. Co więcej, tak wygenerowane pary nie są zależne od aktualnego rozwiązania i, dla każdego kroku algorytmu, są zawsze takie same (o ile nie znajdują się na liście Tabu). Dzięki temu, można łatwo identyfikować ruch jako parę indeksów i przechowywać go na liście Tabu. Liczba wszystkich dostępnych ruchów określa rozmiar sąsiedztwa, które algorytm powinien przeszukać w jednym kroku. Ponieważ liczba ta jest duża, konieczne było ograniczenie przeszukiwanego sąsiedztwa.

5.1.4 Generowanie i ocenianie sąsiedztwa dla wybranego rozwiązania

W ramach jednej iteracji algorytm powinien wygenerować sąsiedztwo dla wybranego rozwiązania, ocenić każdego sąsiada i wybrać najlepszego z nich. Ponieważ, funkcja oceny rozwiązania jest bardzo kosztowna, konieczne było ograniczenie rozmiaru przeszukiwanego sąsiedztwa. Parametr określający jego rozmiar jest jednym z parametrów wejściowych algorytmu. Po raz kolejny zdecydowano się wprowadzić element probabilistyczny. Z puli dostępnych ruchów wybierane są losowe jednostki. Każdy z wylosowanych ruchów wykonywany jest na aktualnym rozwiązaniu, tworząc nowe rozwiązanie, wchodzące w skład sąsiedztwa rozwiązania aktualnego. Tak wygenerowane sąsiedztwo jest oceniane i wybierane jest rozwiązanie najlepsze spośród dostępnych. Ruch, który doprowadził nas do rozwiązania najlepszego trafia na listę Tabu. Następnie wybrane rozwiązanie porównywane jest z rozwiązaniem globalnie najlepszym, jakie do tej pory udało się znaleźć. Jeżeli nowo wybrane rozwiązanie jest lepsze, staje się wtedy rozwiązaniem najlepszym globalnie. Ostatnim krokiem w ramach jednej iteracji algorytmu jest aktualizowanie listy Tabu. Dla każdego elementu na liście aktualizowany jest czas trwania w Tabu i, jeżeli jest on równy zeru, ruch trafia z powrotem do listy dostępnych ruchów algorytmu.

5.1.5 Funkcja oceny rozwiązania

Najtrudniejszym elementem implementacji była funkcja oceny danego rozwiązania. Z uwagi na konieczność sprawdzania wielu warunków i iteracji po wielu elementach, jest to najbardziej czasochłonna część wykonania algorytmu. Wszystkie ograniczenia, które ma spełniać otrzymany plan zajęć, muszą być zaimplementowane w tej funkcji. Funkcja ta bezpośrednio decyduje o jakości otrzymanego planu i daje duże możliwości, by rozwinąć ją w przyszłości.

Zdecydowano się wybrać do implementacji następujące ograniczenia:

- przypisanie wszystkim zdarzeniom określonego czasu,
- unikanie konfliktów między zasobami,
- unikanie dzielenia zajęć trwających dłużej niż jedno okno czasowe,
- minimalizacja pustych okien czasowych między zdarzeniami dla grup uczniów.

Każde z tych ograniczeń posiada określoną wartość kary, która zostanie dodana do końcowej oceny rozwiązania, za każdym razem, gdy ograniczenie nie zostanie spełnione.

Każde zdarzenie musi mieć przypisane okno czasowe. Podczas oceny tego ograniczenia sprawdzane są wszystkie dostępne wydarzenia. Za każdym razem, gdy któreś z wydarzeń nie będzie miało przypisanego czasu, nałożona zostanie odpowiednia kara. Ograniczenie to miało zastosowanie tylko w początkowej fazie implementacji algorytmu, ponieważ aktualnie, każde zdarzenie, już podczas generacji rozwiązania początkowego ma przypisane okno czasowe. Nie ma również możliwości, by algorytm usunął przypisane okno czasowe. Jedynymi zmianami może być przypisanie nowego okna czasowego.

Unikanie konfliktów między zasobami. Ograniczenie to sprawdzane jest w następujący sposób: dla każdego okna czasowego wyszukiwane są wszystkie zdarzenia, które w nim występują. Konieczne jest również przeszukanie dwóch okien czasowych w tył, by sprawdzić czy nie ma tam zajęć o długości dwóch lub trzech jednostek czasu. Po znalezieniu wszystkich zdarzeń, przeszukiwane są wszystkie zasoby, do nich przypisane. Każdy zasób sprawdzany jest pod kątem wystąpienia po raz kolejny w tym samym oknie czasowym. Za każde dodatkowe wystąpienie zasobu nakładana jest stosowna kara. Sprawdzanie tego ograniczenia ma największą złożoność obliczeniową ponieważ wymaga wielokrotnego sprawdzania zasobów.

Unikanie dzielenia zajęć trwających dłużej niż jedno okno czasowe. To ograniczenie może być złamane na przykład w przypadku, gdy zajęcia o czasie trwania dwóch okien czasowych przypisane są na ostatnie okno czasowe danego dnia. Wtedy druga godzina zajęć przypisana jest w pierwszym oknie czasowym dnia następnego. Nie jest to zjawisko pożądane. By wykryć złamanie tego ograniczenia konieczne jest sprawdzenie wszystkich zajęć o czasie trwania dłuższym niż jeden. Dla każdego z tych zajęć sprawdzane są kolejne okna czasowe występujące po nim (w zależności od długości zajęć jest to jedno lub dwa okna czasowe). Jeżeli okna czasowe przypadające na jedno zajęcie należą do różnych dni, to nakładana jest ustalona kara.

Wszystkie dotychczasowe ograniczenia można było określić jako ograniczenia twarde. Kara za ich złamanie była bardzo wysoka. Ostatnie ograniczenie jest ograniczeniem miękkim. W kara za jego złamanie to około 10% wartości kary poprzednich ograniczeń. Minimalizacja pustych okien czasowych między zdarzeniami dla grup uczniów nie jest obowiązkowa dla poprawnego planu zajęć. Jest jednak zjawiskiem pożądanym przez uczniów. By sprawdzić te ograniczenie konieczne jest przeszukanie planu dla każdej klasy z osobna. Jeżeli dla danej klasy, w danym dniu, wystąpiły już zajęcia, następnie wystąpiło puste okno czasowe, a po nim kolejne zajęcia to nałożona zostanie kara. Kara jest zależna od liczby pustych okien czasowych między zajęciami.

5.2 Specyfikacja wewnętrzna

Poprzedni podrozdział miał na celu przybliżyć czytelnikowi pomysł i sposób w jaki zaimplementowany został algorytm *Tabu search*. Ten podrozdział zawiera szczegóły techniczne implementacji, opis użytych technologii, używanych klas oraz zawiera najważniejsze fragmenty kodu.

5.2.1 Użyte technologie i narzędzia

- Język programowania C# - jest to obiektowy język programowania, którego początki sięgają roku 1998. Został zaprojektowany dla firmy Microsoft przez Andersa Hejlsberga. Program napisany w tym języku jest kompilowany do języka *Common Intermediate Language* (CIL), który jest wykonywany w środowisku *.NET Framework*. Jest to język prosty, o dużych możliwościach i wielu cechach wspólnych z językami programowania C++ oraz Java [7].

- Technologia Linq - Language-Integrated Query (LINQ) to technologia zapytań w języku zintegrowanym, pozwalająca odczytywać dane, pochodzące z różnych źródeł, w postaci obiektów. Udostępnia szereg funkcji filtrujących i wyszukiujących, które, odpowiednio użyte, w znacznej mierze przyspieszają działanie programu. W implementacji algorytmu, którego dotyczy ta praca, zapytania Linq ułatwiają komunikację z bazą danych oraz czytanie dokumentów XML.
- Windows Forms - interfejs programowania graficznych aplikacji należący do środowiska *.NET Framework*. Jest to technologia tworzenia aplikacji z graficznym interfejsem użytkownika, umożliwiającą obsługę zdarzeń napływających od użytkownika. Obecnie wypierana przez technologię *WPF*, mimo to, dzięki swej prostocie i przejrzystości, jest dalej używana przez programistów.
- Entity Framework - jest to wolne oprogramowanie typu Object Relational Mapping (ORM), pozwalające odwzorować relacyjną bazę danych za pomocą architektury obiektowej. Dzięki temu narzędziu programista może w łatwy sposób zaprojektować bazę danych nie mając wiedzy na temat języka SQL. Stosując podejście *code first* można zaprojektować tylko klasy bazowe, a resztę powierzyć narzędziu Entity Framework, które utworzy odpowiednią strukturę bazy danych.
- Microsoft Visual Studio 2015 - zintegrowane środowisko programistyczne firmy Microsoft, umożliwia pisanie i kompilowanie aplikacji w językach takich jak C#, C++, C czy Visual Basic, na różne platformy. Dzięki dużej liczbie narzędzi umożliwia łatwe testowanie i debugowanie kodu, oraz pozwala wydajnie tworzyć nowe aplikacje. Jest to naturalny wybór przy języku programowania C#.
- SQL Server 2014 Managment Studio - zintegrowane środowisko do zarządzania bazami danych. Zawiera narzędzia do konfiguracji i monitorowania baz danych, umożliwia budowę zapytań i podgląd wszystkich tabel w bazie danych. Autor pracy wykorzystał to środowisko do testowania poprawności wykonanych operacji i analizy uzyskanych wyników.

5.2.2 Opis klas i ważniejszych funkcji

Solucja aplikacji podzielona została na foldery. Każdy z folderów zawiera klasy odpowiedzialne za inną funkcjonalność aplikacji. Folder *Code* zawiera klasy odpowiedzialne za cały algorytm wyszukiwania. W folderze *Data* znajdują się klasy służące

do parsowania plików wejściowych XML na obiekty, i zapisywania nowo powstałych obiektów do bazy danych. Folder *Model* zawiera klasy odwzorowujące obiekty znajdujące się w bazie danych, znajdują się więc tu klasy takie jak *Instance*, *Event*, *Resource*. Ostatni folder *ViewModel* przechowuje klasy odpowiedzialne za przygotowanie danych do zaprezentowania ich w interfejsie graficznym. Poza folderami znajduje się osobna klasa *Form1*, która obsługuje interfejs graficzny użytkownika.

Poniżej przedstawiono ważniejsze klasy programu razem z opisem funkcji w nich zawartych.

- *Code/SolutionManager* - klasa ta zawiera szkielet algorytmu Tabu search. Jest to główna klasa algorytmu. Znajdują się w niej funkcja odpowiedzialna za rozwiązywanie problemu układania planu zajęć dla instancji o podanym Id. Klasa ta udostępnia również funkcje zapisującą raport z pracy algorytmu do pliku txt oraz funkcję wypisującą rozwiązanie na konsolę. Główna funkcja klasy zaprezentowana została na rysunku 5.2.
- *Code/TabuSearch* - jest to klasa zawierająca statyczne funkcje służące do rozwiązywania problemu za pomocą algorytmu Tabu search. Znajdują się tu funkcje generujące dostępne ruchy, losowe rozwiązanie startowe, aktualizujące listę Tabu, generujące sąsiedztwo i wybierające z nich najlepszego osobnika. W tej klasie znajdują się najważniejsze fragmenty kodu implementujące algorytm przeszukiwania Tabu. Funkcja generująca sąsiedztwo i wybierająca z niego najlepszą instancję została zaprezentowana na rysunku 5.3.
- *Code/EvaluationFunction* - w skład tej klasy wchodzi funkcje oceniające bieżące rozwiązanie algorytmu. Znajduje się tu implementacja wszystkich ograniczeń jakie mają być uwzględnione przy rozwiązaniu. Klasę tę można w łatwy sposób rozbudowywać dodając kolejne funkcje oceniające, które odpowiadają kolejnym ograniczeniom.
- *Code/TabuItem* - klasa reprezentująca pojedynczy ruch w algorytmie przeszukiwania Tabu. Jako ruch rozumiana jest para dwóch indeksów, pierwszy z nich odpowiada id zdarzenia, a drugi to id okna czasowego, które, w danym ruchu, jest do tego zdarzenia przypisane. Elementy tej klasy przechowywane są na liście Tabu, dlatego klasa zawiera dodatkową właściwość, jaką jest czas trwania na liście Tabu dla danego elementu.

```
public void ResolveSimpleProblem(int Id)
{
    int resultIteration = 0;
    LoadEntityFromDB(Id);

    List<TabuItem> tabuList = new List<TabuItem>();
    List<TabuItem> availableList = new List<TabuItem>();
    actualInstance = (Instance)instanceToResolve.Clone();

    TabuSearch.GenerateStartSolutionForTimes(actualInstance);
    availableList = TabuSearch.GenerateAvaibleMoveList
        (actualInstance);
    bestRating = EvaluationFunction.EvaluateInstance
        (actualInstance);
    for (int i = 0; i < iteration; i++)
    {
        var tempInstance = TabuSearch
            .SelectBestInstanceFromNeighborhood
            (actualInstance, tabuList, availableList);
        if (tempInstance != null)
            actualInstance = tempInstance;

        int rating = EvaluationFunction.EvaluateInstance
            (actualInstance);
        raport.Add(rating);
        if (bestRating > rating)
        {
            bestInstance = actualInstance;
            bestRating = rating;
        }

        if (bestRating <= 0)
        {
            resultIteration = i;
            break;
        }

        TabuSearch.UpdateTabuList(tabuList, availableList);
    }

    EvaluationFunction.EvaluateInstanceWithRaport(bestInstance);
    SaveRaportToFile();
}
```

Rysunek 5.2: Funkcja *ResolveSimpleProblem* zawierająca szkielet algorytmu Tabu search.

```
public static Instance SelectBestInstanceFromNeighborhood
(Instance instance, List<TabuItem> availableList)
{
    Random r = new Random();
    List<Instance> neighborhood = new List<Instance>();
    List<TabuItem> usedItems = new List<TabuItem>();
    //Neighborhood generation
    for (int i = 0; i < NeighborhoodSize; i++)
    {
        if (availableList.Count > 0)
        {
            TabuItem item = availableList.ElementAt
                (r.Next(availableList.Count));
            availableList.Remove(item);
            usedItems.Add(item);
            Instance copy = (Instance)instance.Clone();
            copy.Events[item.IndexX].Time
                = copy.Times[item.IndexY];

            neighborhood.Add(copy);
        }
        else
        {
            System.Debug.WriteLine("Non moves.");
        }
    }

    var inst = SelectBestInstanceFromGeneratedNeighborhood
        (neighborhood);

    var itemToTabu = usedItems[neighborhood.IndexOf(inst)];
    availableList.AddRange(usedItems);
    availableList.Remove(itemToTabu);
    itemToTabu.TabuDuration = TabuDuration;
    tabuList.Add(itemToTabu);

    return inst;
}
```

Rysunek 5.3: Funkcja *SelectBestInstanceFromNeighborhood* generująca sąsiedztwo i wybierająca najlepszego sąsiada.

- *Data/XMLLoader* - w tej klasie zawarte są funkcje prasujące dane wejściowe w formacie XHSTT, zapisane w języku XML, na klasy modelu. Znajduje się tu również funkcja zapisująca odczytaną instancję do bazy danych.
- *Form1* - klasa obsługująca interfejs użytkownika. Zawiera funkcję reagującą na każdą wykonaną przez użytkownika czynność oraz funkcje wypisujące dane do odpowiednich obiektów interfejsu.

5.3 Specyfikacja zewnętrzna

Po uruchomieniu aplikacji użytkownik zobaczy okno przedstawione na rysunku 5.4. Po pierwszym uruchomieniu użytkownik powinien wczytać instancje z pliku wejściowego. Może to wykonać za pomocą przycisku "Load instance from file,,", który znajduje się w prawym górnym rogu aplikacji. Otworzy się okno dialogowe, gdzie należy wskazać plik XML zawierający dane wejściowe dla problemu rozwiązywania planu zajęć, zapisane w formacie XHSTT, omówiony w poprzednim rozdziale. Po wczytaniu wielu instancji (lub po ponownym uruchomieniu aplikacji), użytkownik może wybrać instancje do rozwiązania. Uczyni to za pomocą rozwijanej listy w lewym górnym rogu.

Przycisk "Resolve,,", służy do uruchomienia algorytmu. Przebieg algorytmu jest stale monitorowany, a postępy wyświetlane są na konsoli na dole aplikacji. Po zakończeniu obliczeń, ułożony plan zajęć będzie można oglądać w tabeli. Rozwijana lista nad tabelą służy do wyboru klasy, dla której ma zostać zaprezentowany plan zajęć.

Użytkownik może zmieniać parametry wejściowe algorytmu. Algorytm posiada trzy parametry wejściowe, których opis znajduje się poniżej:

- Iterations - maksymalna liczba iteracji jaką algorytm wykona w czasie rozwiązywania problemu układania planu zajęć. Jeżeli algorytm osiągnie wynik zero wcześniej, to zakończy swoje działanie.
- Tabu duration - parametr oznaczający długość trwania pojedynczego ruchu na liście Tabu. Długość rozumiana jest jako liczba iteracji algorytmu.
- Neighborhood - rozmiar sąsiedztwa jakie zostanie losowo wybrane i przeszukane podczas jednej iteracji algorytmu. Rozmiar sąsiedztwa nie powinien być większy od liczby wszystkich dostępnych ruchów algorytmu (pełnego sąsiedztwa).

FI-WP-06

Resolve

Load instance from file

Iterations 1000

Tabu duration 500

Neighborhood 300

SB

	Godzina	Poniedziałek	Wtorek	Środa	Czwartek	Piątek
▶	7:00	C034_1	C031_1		C027_1	
	8:00	C034_1	C011_1	C058_1	C027_2	C031_3
	9:00	C058_2	C002_2	C058_1	C034_3	C034_2
	10:00	C058_2	C002_2	C043_3	C034_3	C034_2
	11:00	C043_1	C011_3	C043_3	C034_3	C011_2
	12:00	C002_1	C011_3	C043_2	C027_3	C011_2
	13:00	C002_1		C043_2	C031_2	C058_3

Iteration: 940, Actual best result: 39
Iteration: 950, Actual best result: 39
Iteration: 960, Actual best result: 39
Iteration: 970, Actual best result: 39
Iteration: 980, Actual best result: 39
Iteration: 990, Actual best result: 39
&&& Result iteration: 0

^
▼

Rysunek 5.4: Interfejs użytkownika aplikacji.

Rozdział 6

Testowanie

6.1 Środowisko testowe

Aplikacja została zbudowana i testowana na komputerze stacjonarnym z systemem operacyjnym Windows 10. Specyfikacja jednostki testowej prezentuje się następująco:

- procesor
- karta graficzna
- pamięć RAM
- dysk twardy

6.2 Zestaw danych testowych

Jako zestaw danych wejściowych, dla których testowany był algorytm układania planu zajęć, wybrany został zestaw fińskiej szkoły ponadpodstawowej. Opart on został na danych z roku 2006 szkoły West-Pori High School, gdzie wiek uczniów należy do przedziału 16-19 lat. Zestaw został wybrany z uwagi na rozmiar danych oraz duże podobieństwo siatki zajęć do polskiego gimnazjum.

Wyżej wspomniany zestaw danych zawiera jedną instancję problemu układania planu zajęć. W instancji tej występuje:

- 35 okien czasowych rozłożonych na 5 dni w tygodniu (maksymalnie 7 zajęć dziennie),
- 18 dostępnych nauczycieli,
- 13 sal, w których mogą być prowadzone zajęcia,
- 10 grup uczniów,
- 172 wydarzenia o łącznej długości 297 okien czasowych (wiele zajęć dwu i trzy godzinnych).

Pokrycie okien czasowych w tym planie zajęć wynosi około 85%. Liczba ta jest stosunkiem łącznej długości wydarzeń do całkowitej liczby dostępnych okien czasowych dla wszystkich grup ($35 \cdot 10$). Patrząc na pokrycie można określić stopień trudności wykonania zadania układania planu zajęć. Dla pokrycia wynoszącego 100% ułożenie poprawnego planu zajęć jest bardzo trudne, ponieważ w oczekiwanym planie zajęć nie może być ani jednego wolnego okna czasowego. Pokrycie wynoszące 85 % to średnio zaawansowany stopień trudności. Dla porównania, pokrycie planu zajęć dla polskiego liceum to średnio 75 %.

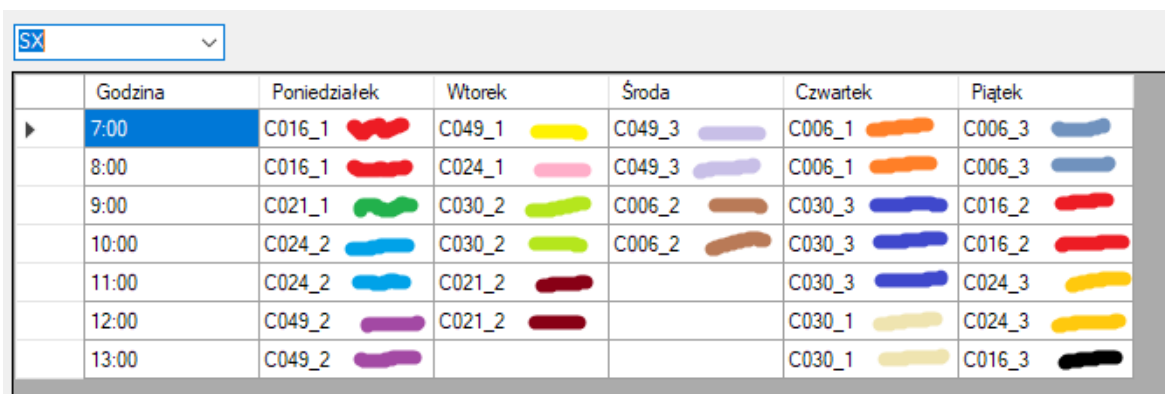
Testowany zestaw danych wejściowych zawiera cztery ograniczenia:

- przypisany czas,
- unikanie konfliktów,
- brak podziału zdarzeń,
- ograniczenie bezczynności uczniów.

Ograniczenia te zostały szczegółowo omówione w rozdziale 4.

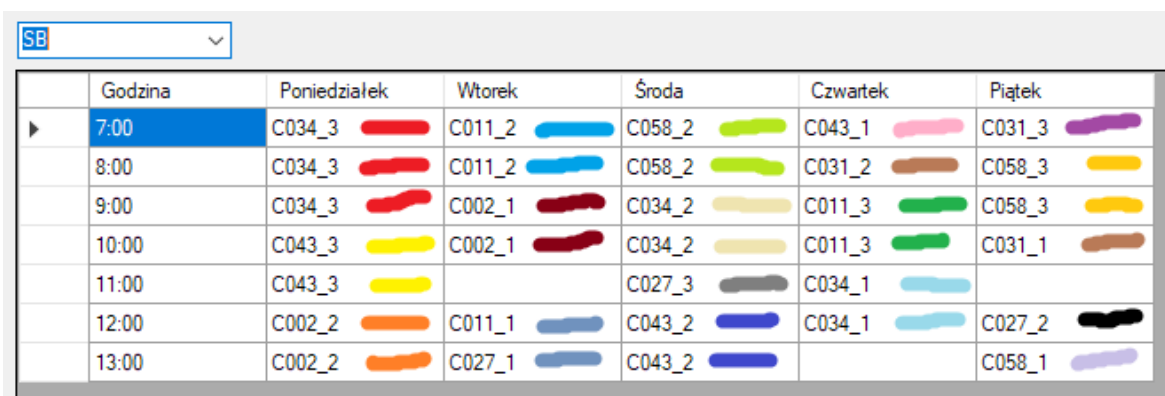
6.3 Wyniki działania aplikacji

Aplikacja została uruchamiana i testowana wielokrotnie dla różnych parametrów algorytmu. Średni czas pracy algorytmu wynosił około 8 minut. Najlepszy wynik jaki udało się uzyskać to plan zajęć o ocenie 12. Oznacza to, że w otrzymanym planie zajęć nie wystąpił żaden konflikt zasobów oraz, że wystąpiło tylko 6 okienek (wolnych okien



	Godzina	Poniedziałek	Wtorek	Środa	Czwartek	Piątek
▶	7:00	C016_1	C049_1	C049_3	C006_1	C006_3
	8:00	C016_1	C024_1	C049_3	C006_1	C006_3
	9:00	C021_1	C030_2	C006_2	C030_3	C016_2
	10:00	C024_2	C030_2	C006_2	C030_3	C016_2
	11:00	C024_2	C021_2		C030_3	C024_3
	12:00	C049_2	C021_2		C030_1	C024_3
	13:00	C049_2			C030_1	C016_3

Rysunek 6.1: Najlepszy otrzymany plan zajęć dla najlepszego rozwiązania.



	Godzina	Poniedziałek	Wtorek	Środa	Czwartek	Piątek
▶	7:00	C034_3	C011_2	C058_2	C043_1	C031_3
	8:00	C034_3	C011_2	C058_2	C031_2	C058_3
	9:00	C034_3	C002_1	C034_2	C011_3	C058_3
	10:00	C043_3	C002_1	C034_2	C011_3	C031_1
	11:00	C043_3		C027_3	C034_1	
	12:00	C002_2	C011_1	C043_2	C034_1	C027_2
	13:00	C002_2	C027_1	C043_2		C058_1

Rysunek 6.2: Najgorszy otrzymany plan zajęć dla najlepszego rozwiązania (2 okienka).

czasowych pomiędzy zasobami), co daje wynik około jednego okienka na dwie klasy uczniów. Jest to wynik bardzo dobry. Na rysunkach 6.1 i 6.2 przedstawiono najlepszy i najgorszy plan dla różnych klas w tym rozwiązaniu. Kolory zostały naniesione w celu lepszego pokazania zajęć o długości większej niż jedno okno czasowe.

Na wykresie przedstawionym na rysunku 6.3 przedstawiono drogę algorytmu jaką musiał pokonać w czasie swojej pracy. Na osi pionowej przedstawiona jest ogólna ocena aktualnego rozwiązania, na osi poziomej przedstawiono liczbę iteracji.

Najlepszy wynik otrzymany został dla następujących parametrów wejściowych: liczba iteracji - 1000, długość tabu - 500, rozmiar sąsiedztwa - 300. Liczba iteracji algorytmu została ustalona biorąc pod uwagę czas pracy programu oraz fakt, że liczba ta zazwyczaj wystarczała, by usunąć wszystkie konflikty z planu zajęć.

Rozmiar sąsiedztwa również w dużej mierze wpływa na czas pracy algorytmu. Z drugiej strony doświadczenia pokazały, że im większe sąsiedztwo tym mniej iteracji po-



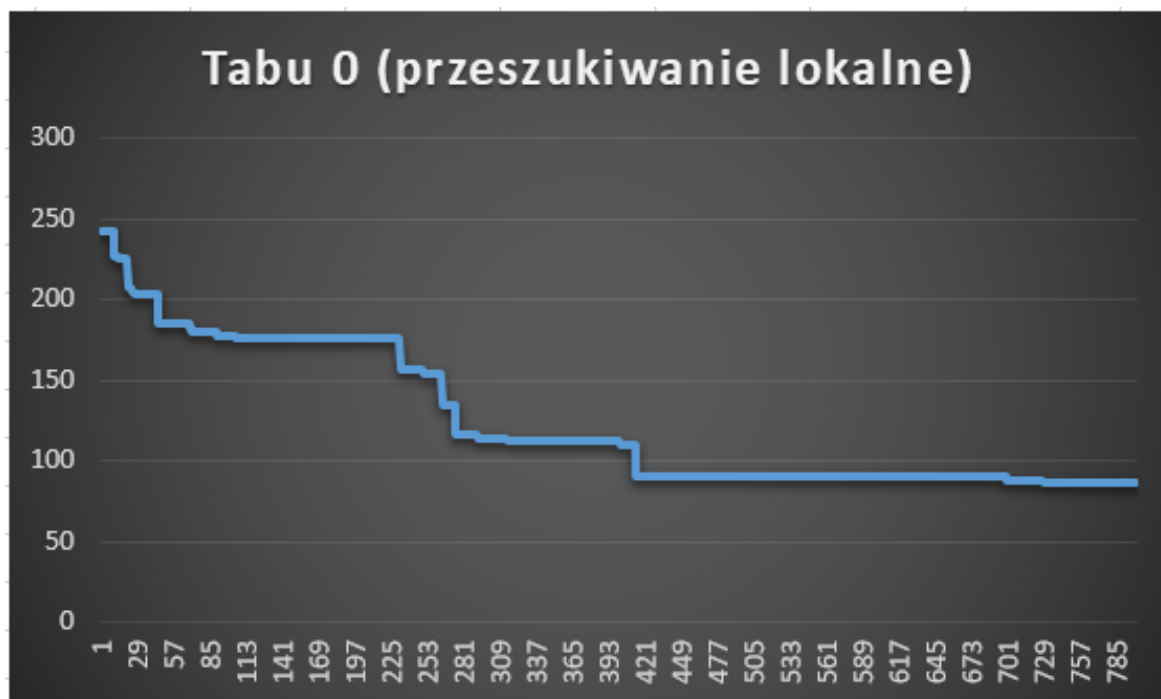
Rysunek 6.3: Przebieg pracy algorytmu.

trzebnych do uzyskania lepszych wyników. Rozmiar 300 to około 5% całego sąsiedztwa możliwego do przeszukania. Ustawienie algorytmu na przeszukiwanie 100 % sąsiedztwa, spowodowało by wydłużenie pracy algorytmu dwudziestokrotnie. Ustalony rozmiar jest próbą pogodzenia oczekiwań jak najkrótszego czasu pracy algorytmu i jak najlepszych wyników.

Długość tabu zmienia całkowicie sposób pracy algorytmu. Ustalenie wartości 500 wynikało z obserwacji wyników jakie dostarczał algorytm oraz wykresów, obrazujących jego pracę. Szczegółowe omówienie wpływu parametru długości Tabu przedstawiono w kolejnym podrozdziale.

6.4 Znaczenie parametru długości Tabu.

Algorytm wyszukiwania z Tabu opiera się na algorytmie przeszukiwania lokalnego. Najważniejszą różnicą między tymi algorytmami jest lista Tabu. Dlatego parametr określający długość trwania elementów na liście Tabu ma bardzo duże znaczenie. Zmiana tego parametru powoduje zupełnie inną pracę algorytmu. Potwierdzeniem tego niech będą poniższe wykresy.

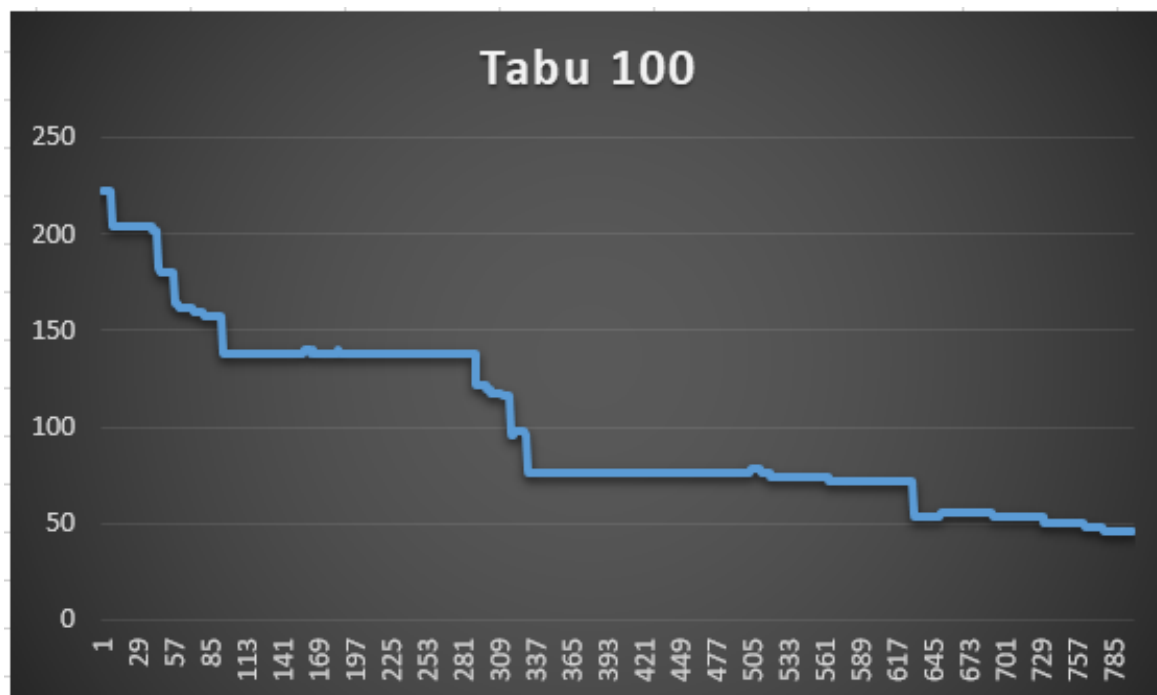


Rysunek 6.4: Przebieg pracy algorytmu dla parametru długości Tabu - 0.

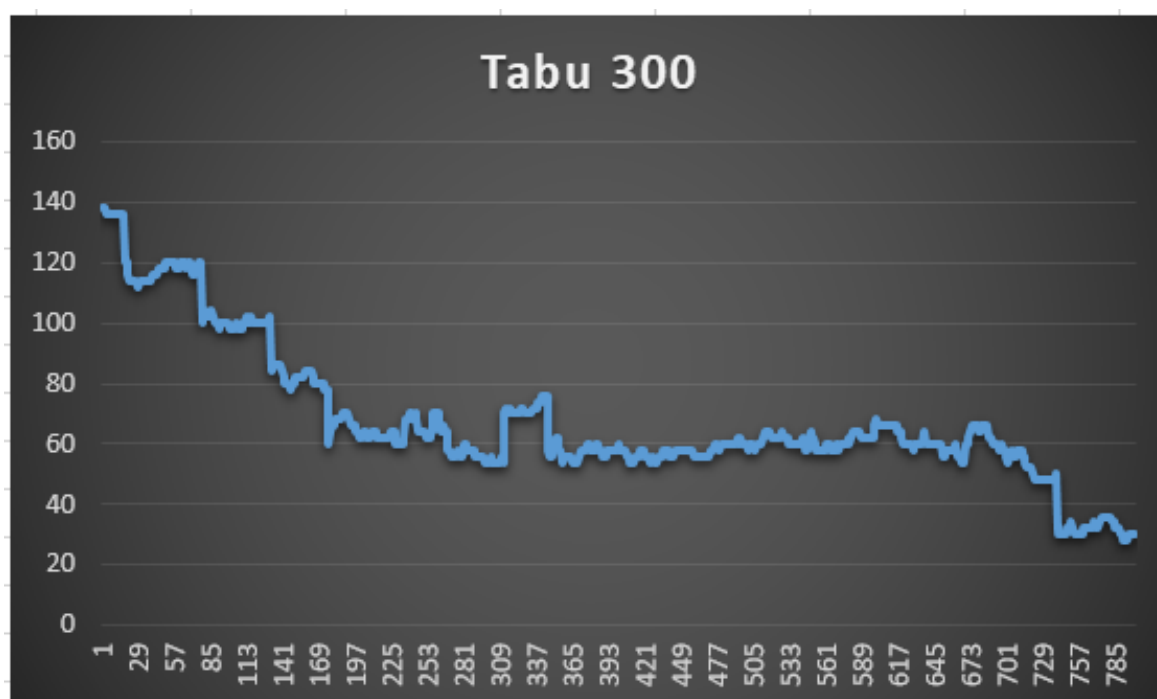
Na wykresach zaprezentowanych na rysunkach 6.4 - 6.8 przedstawiono przebieg pracy algorytmu dla parametrów: 1000 iteracji i rozmiarze sąsiedztwa równym 300. Zmieniał się tylko parametr długości Tabu. Jak można zauważyć na rysunku 6.3, algorytm w początkowej fazie pracy znacznie poprawia swój wyniki (z wyniku około 6000 do wyniku 200). Z tego powodu na poniższych wykresach przedstawiono tylko ostatnie 800 iteracji, by obciąć skok i móc zaobserwować zmiany pracy algorytmu w mniejszej skali.

Pierwszy z wykresów (rysunek 6.4) prezentuje pracę algorytmu dla parametru długości Tabu równym 0. Oznacza to, że algorytm w ogóle nie korzysta z listy Tabu. Jest więc to przeszukiwanie lokalne. Jak widać, algorytm przeszukuje tylko wyniki, które mają tę samą ocenę lub wyższą. Na wykresie nie można zauważyć żadnego skoku wartości w górę. Jest to spowodowane wpadaniem algorytmu w lokalne minima, a poprawa rezultatu najczęściej jest skutkiem innego losowego doboru sąsiadów. Wynik jaki udało się osiągnąć to 84.

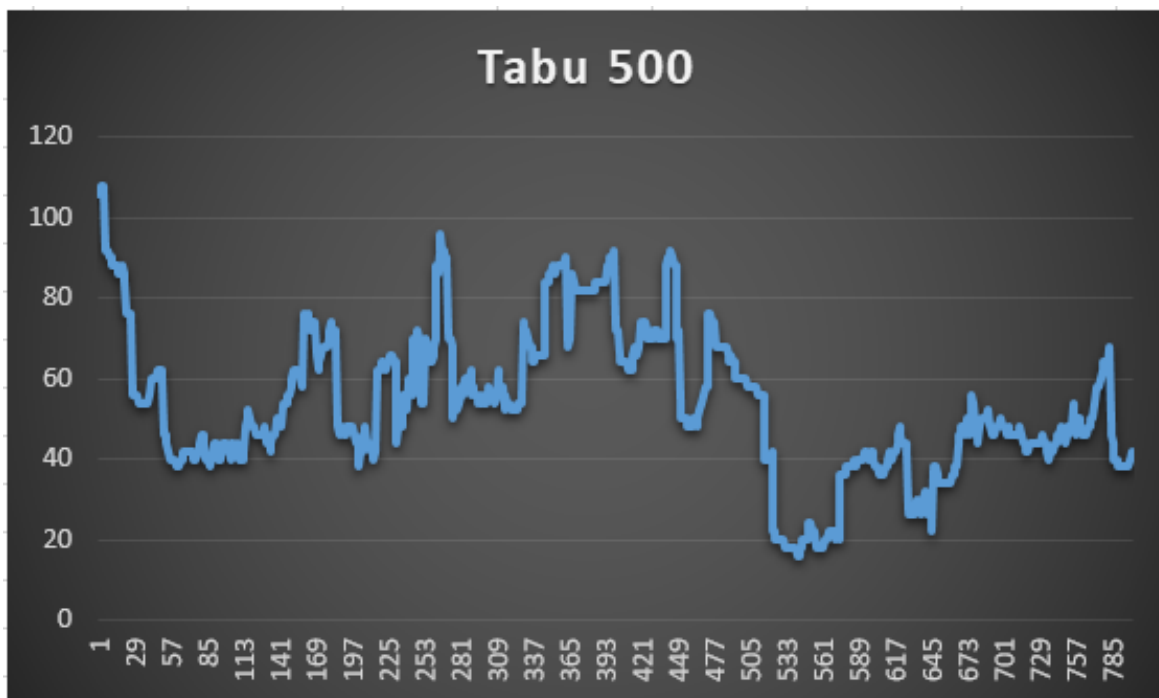
Drugi wykres (rysunek 6.5) przedstawia przebieg algorytmu dla parametru długości Tabu równego 100. Otrzymany wykres jest bardzo zbliżony do poprzedniego, jednak można zaobserwować małe próby pogorszenia wyników w celu dotarcia do rozwiązania



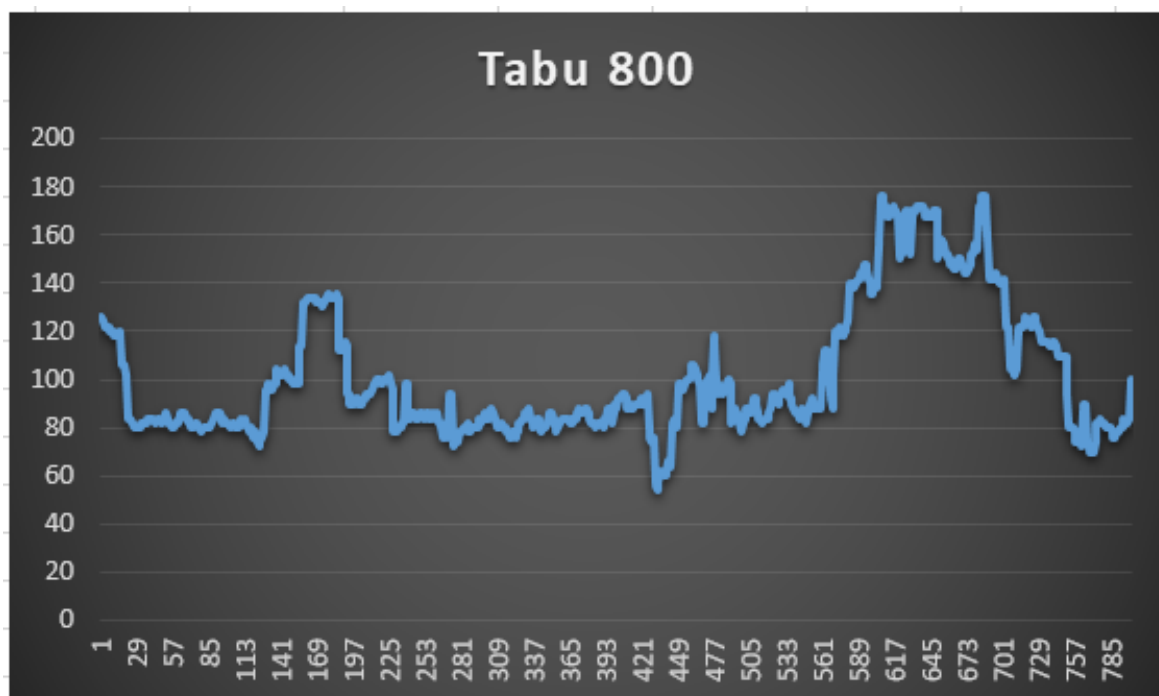
Rysunek 6.5: Przebieg pracy algorytmu dla parametru długości Tabu - 100.



Rysunek 6.6: Przebieg pracy algorytmu dla parametru długości Tabu - 300.



Rysunek 6.7: Przebieg pracy algorytmu dla parametru długości Tabu - 500.



Rysunek 6.8: Przebieg pracy algorytmu dla parametru długości Tabu - 800.

jeszcze lepszego. Najlepszy wynik to 48. Jak widać, 100 ruchów zabronionych z puli 6000 dostępnych (dla tego przypadku testowego), to zdecydowanie za mało, by efektywnie korzystać z zalet wyszukiwania z Tabu.

Na trzecim wykresie (rysunek 6.6) sytuacja wygląda już zdecydowanie lepiej. Długość Tabu równa 300 wymusza na algorytmie przeszukiwanie obszarów, do których dotrzeć można tylko przez pogorszenie bieżącego wyniku. Większe skoki algorytmu do miejsc, w których udało się wyeliminować pewien konflikt zasobów (ocena 20). Mniejsze zmiany obrazują próby minimalizacji okienek (ocena 2 dla jednego okna czasowego). Najlepszy uzyskany wynik to 30.

Czwarty wykres (rysunek 6.7) prezentuje sytuację, w której udało się osiągnąć najlepsze rezultaty (w tym przypadku 18). Najlepszy wynik udało się osiągnąć już w 740 iteracji (540 + 200 iteracji nie przedstawionych na wykresie). Dla parametru długości Tabu równym 500 przeszukiwanie odbywa się już bardzo skokowo. Algorytm nie może wykonać 500 ruchów, które w poprzednich iteracjach najbardziej poprawiły wyniki algorytmu. Dzięki temu jest w stanie dotrzeć do obszarów, które dla przeszukiwania lokalnego są nieosiągalne.

Ostatni wykres (rysunek 6.8) przedstawia przebieg algorytmu dla parametru długości Tabu równego 800. Przy 1000 iteracjach algorytmu tylko pierwsze 200 ruchów będzie mogło zostać powtórzone. Niestety, tak duża liczba zabronionych ruchów spowodowała, że algorytm zdecydowanie pogorszył swoje wyniki (najlepszy rezultat to 52). Choć program próbował przejść w inne obszary poszukiwań, to nie otrzymano oczekiwanych rezultatów. Najprawdopodobniej, kluczowe ruchy znajdowały się na liście Tabu.

Rozdział 7

Podsumowanie

Bibliografia

- [1] A. Debudaj-Grabysz, J. Widuch, and S. Deorowicz. *Algorytmy i struktury danych. Wybór zaawansowanych metod*. Wydawnictwo Politechniki śląskiej, Gliwice, 2012.
- [2] Fred Glover. *Future Paths for Integer Programming and Links to Artificial Intelligence*. Oxford: Elsevier, 1986.
- [3] I.H. Osman and J.P. Kelly. *Meta-heuristics: An Overview*. Kluwer Academic Publishers, 1996.
- [4] Strona internetowa: <http://www.it.usyd.edu.au/~jeff/cgi-bin/hseval.cgi?op=spec>, czerwiec 2017.
- [5] Strona internetowa: <http://www.it.usyd.edu.au/~jeff/cgi-bin/hseval.cgi?op=spec&part=constraints>, czerwiec 2017.
- [6] Strona internetowa: <https://www.utwente.nl/ctit/hstt>, czerwiec 2017.
- [7] Strona internetowa: <https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework>, czerwiec 2017.