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

W niniejszej pracy został opisany proces przystosowania algorytmu wyszukiwania z tabu do rozwiązywania problemu układania planu zajęć. Aspektem badawczym pracy było zbadanie wpływu parametrów zaimplementowanego algorytmu na szybkość jego działania i otrzymane wyniki. Omówione zostały podstawy teoretyczne badanego problemu oraz trudności towarzyszące implementacji algorytmu jego rozwiązania. Testowanie algorytmu zostało przeprowadzone na danych rzeczywistych, a wyniki zaprezentowane na wykresach.

Spis treści

Spis rysunków

Rozdział 1

Wstęp

Problem układania planu zajęć jest dobrze znany w środowiskach szkolnych i akademickich. Układanie planu to zajęcie żmudne oraz wymagające dużej spostrzegawczości i wyobraźni u osoby, która plan układa. Problem ten jest aktualny każdego roku, lub w przypadkach koniecznych np. przy zmianie liczby prowadzących, bądź zmienionej siatce zajęć. Układanie planu zajęć metodami tradycyjnymi pochłania dużo czasu i często kończy się niepowodzeniem. Jeżeli już uda się ułożyć plan, to często nie jest on najlepszy i nie spełnia oczekiwań uczniów bądź nauczycieli. Dlatego też badacze w wielu środowiskach na świecie rozważają problem układania planu zajęć i poszukują efektywnych algorytmów jego rozwiązania. W obecnych czasach mało która szkoła lub uczelnia nie wspomaga się przy tej czynności komputerem. Możliwość szybkiego utworzenia i ocenienia wielu planów lub uzyskanie wiedzy, czy dany plan jest możliwy do zrealizowania, to tylko niektóre zalety zastosowania do tego celu komputera. Najważniejszą zaletą jest możliwość skorzystania z algorytmów, które w krótkim czasie, np. 10 minut, ułożą plan zajęć zgodnie z podanymi założeniami.

Celem niniejszej pracy jest zbadanie możliwości zastosowania algorytmu wyszukiwania z tabu do problemu układania planu zajęć. Algorytm ten, rozwiązujący trudne problemy z wielu dziedzin życia, może być również zastosowany do rozwiązywania problemu układania planu zajęć. Praca oparta jest na materiałach źródłowych, w których do optymalizacji układania planu zajęć wykorzystano różne algorytmy heurystyczne. Korzystając z doświadczeń poprzedników, udało się uniknąć typowych

błędów pojawiających się przy implementacji algorytmów rozwiązywania rozważanego problemu.

Aspektem badawczym pracy jest zbadanie wpływu parametrów zaimplementowanego algorytmu na szybkość jego działania i otrzymywane wyniki. Korzystając z dużej bazy danych testowych pochodzących z różnych szkół i uczelni na świecie dokonamy testowania algorytmu dla rzeczywistych problemów układania planu zajęć.

Praca, poza wstępem i zakończeniem, składa się z pięciu rozdziałów oraz jednego dodatku. Pierwszy rozdział poświęcony jest omówieniu algorytmu wyszukiwania z tabu. Przedstawiono w nim cechy algorytmów heurystycznych i metaheurystycznych oraz genezę algorytmu wyszukiwania z tabu. W rozdziale znaleźć można również omówienie podstawowej idei działania algorytmu oraz różnych jego wariantów. Zdefiniowano także podstawowe pojęcia związane z algorytmem, takie jak sąsiedztwo, kryterium aspiracji, dywersyfikacja, intensyfikacja, etc. Drugi rozdział formułuje problem układania planu zajęć. Zawiera formalizację problemu i omówienie ograniczeń związanych z planami zajęć. Przedstawiona jest również złożoność problemu układania planu zajęć w celu uzasadnienia potrzeby zastosowania algorytmów heurystycznych. Kolejny rozdział w całości poświęcony jest formatowi danych wejściowych, jakie są niezbędne do zaimplementowania algorytmu w celu poprawnego działania. Używany format danych jest wykorzystywany w problemie układania planu zajęć przez wielu badaczy na całym świecie. Opisanie formatu pozwala zrozumieć problemy, z jakimi trzeba się zmierzyć podczas implementowania algorytmu. Implementacja algorytmu oraz jego specyfikacja wewnętrzna i zewnętrzna zostały opisane w następnym rozdziale. Przedstawiono w nim funkcje wykorzystane w jej realizacji. Omówiono również wykorzystane podczas pracy narzędzia i środowiska. W ramach specyfikacji zewnętrznej przedstawiono opis interfejsu graficznego użytkownika oraz sposób korzystania z aplikacji. Ostatni rozdział opisuje proces testowania algorytmu. Omówiona została w nim specyfikacja środowiska testowego oraz zestaw danych wybrany do testowania. Wyniki działania aplikacji zostały zaprezentowane w postaci dwóch wygenerowanych planów zajęć wraz z oceną uzyskanych rozwiązań. Na końcu rozdziału zaprezentowano wpływ parametru długości listy tabu na wyniki działania algorytmu. Do pracy dołączono dodatek, w którym przedstawiono wybrane fragmenty kodu źródłowego.

Rozdział 2

Wprowadzenie do algorytmów heurystycznych

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 otrzymy-

wanych 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.

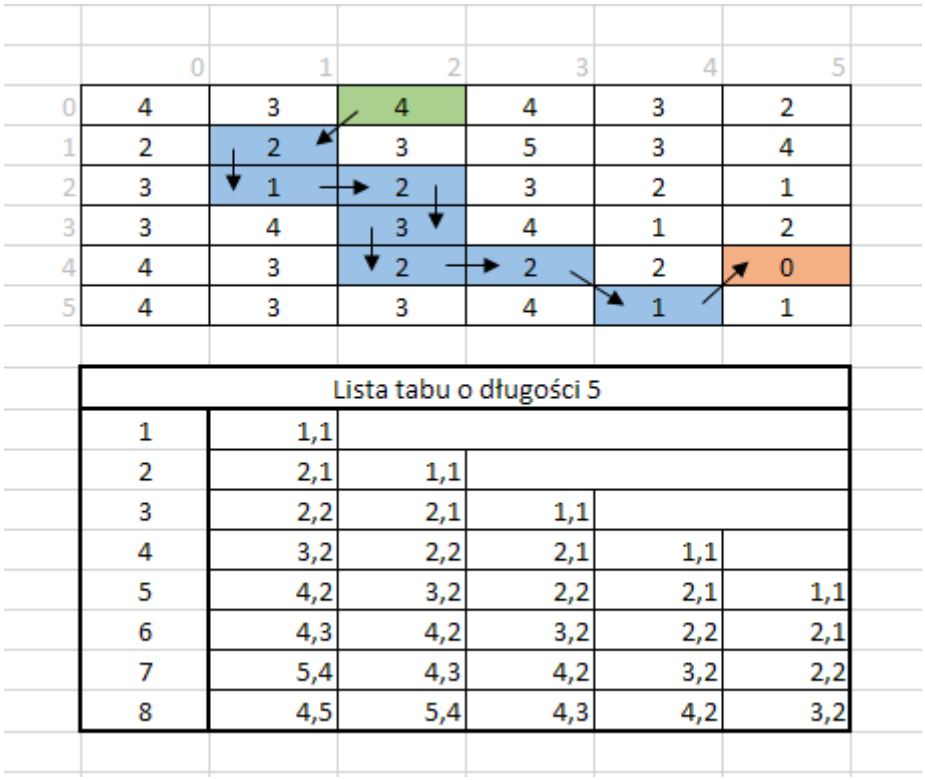
W niektórych algorytmach wykorzystane są dwie heurystyki, nadrzędna i podrzędna. Pierwsza z nich steruje i uzupełnia działanie drugiej heurystyki. Takie podejście nazywane jest przez niektórych badaczy metaheurystykami [1]. Inna definicja metaheurystyki to „procesy iteracje działające zgodnie z klasyczną metodą heurystyczną, wspomagane inteligentnie przez różne koncepcje eksplorowania i eksploatowania 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 Algorytm wyszukiwania z tabu

Przykładem algorytmu metaheurystycznego jest algorytm wyszukiwania z tabu (ang. *Tabu Search*). 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

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 (rysunek 2.1). Możliwe jest złamanie tej zasady, ale tylko wtedy, gdy ruch spełnia tzw. kryterium aspiracji. Warunkiem zakończenia działania algorytmu jest najczęściej wykonanie określonej liczba iteracji lub osiągnięcie satysfakcjonującego rozwiązania. Możliwe jest również monitorowanie aktualnego wyniku i, jeżeli nie ulega on poprawie przez określoną liczbę iteracji, zatrzymanie wykonania algorytmu.



	0	1	2	3	4	5
0	4	3	4	4	3	2
1	2	2	3	5	3	4
2	3	1	2	3	2	1
3	3	4	3	4	1	2
4	4	3	2	2	2	0
5	4	3	3	4	1	1

Lista tabu o długości 5					
1	1,1				
2	2,1	1,1			
3	2,2	2,1	1,1		
4	3,2	2,2	2,1	1,1	
5	4,2	3,2	2,2	2,1	1,1
6	4,3	4,2	3,2	2,2	2,1
7	5,4	4,3	4,2	3,2	2,2
8	4,5	5,4	4,3	4,2	3,2

Rysunek 2.1: Ilustracja procesu wyszukiwania z tabu

Pseudokod działania algorytmu przedstawiony został na rysunku 2.2.

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. 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:

```
s := s0;
sNajlepsze := s;
listaTabu := new list[];
while (not warunkiZatrzymania())
    listaKandydatów := generujKandydatów();
    najlepszyKandydat := null;
    for (kandydat in listaKandydatów)
        if ((not listaTabu.Zawiera(kandydat)) and
            (Ocena(kandydat) > Ocena(najlepszyKandydat)))
            najlepszyKandydat := kandydat;
        end
    end;
    s:= najlepszyKandydat;
    if (Ocena(najlepszyKandydat) > Ocena(sNajlepsze))
        sNajlepsze ;= najlepszyKandydat;
    end;
    listaTabu.Dodaj(najlepszyKandydat);
    if (listaTabu.Rozmiar > maxRozmiarTabu)
        listaTabu.UsuńPierwszy();
    end;
end;
return sNajlepsze;
```

Rysunek 2.2: Pseudokod algorytmu wyszukiwania z tabu

- 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 zapętleń. 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 obszaru poszukiwań

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 znaczniej mierze poprawia efektywność algorytmu, dlatego kryteria te są wykorzystywane w większości nowych wersji algorytmu *Tabu Search*.

Rozdział 3

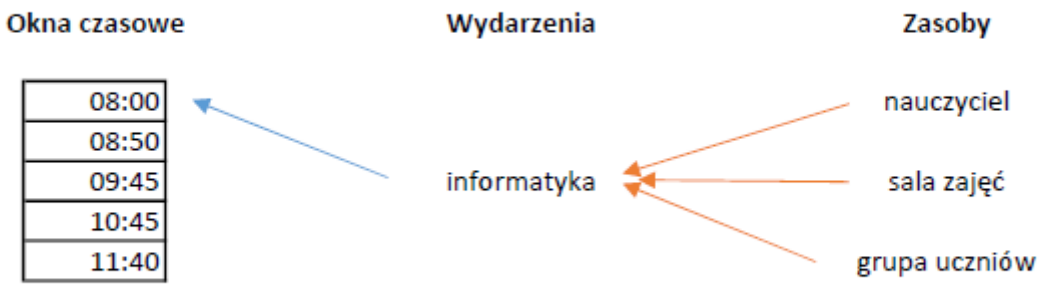
Problem układania planu zajęć

3.1 Sformułowanie problemu

Problem układania planu zajęć można sformułować następująco. Dana jest lista określonych wydarzeń, dostepnych okien czasowych i zasobów. Należy przyporządkować wydarzeniom okna czasowe oraz zasoby w taki sposób, by spełnić przyjęte założenia (rysunek 3.1). 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.



Rysunek 3.1: Ilustracja problemu układania planu zajęć

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, niespelnione 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 prace w innej szkole). Założenia te z reguły precyjzuje osoba uruchomiająca 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.

Podejście, 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 układania planu zajęć jest popularny w świecie naukowym. Dostępnych jest wiele rozwiązań tego problemu i ciągle powstają nowe. Jednak, by móc porównać ze sobą algorytmy rozwiązywania problemu, konieczne jest operowanie na tych samych danych testowych. Z tego powodu badacze uzgodnili jednolity format danych wejściowych o nazwie XHSTT i stworzyli bazę danych wraz z przykładowymi rozwiązaniami [6]. Dane te zapisane są za pomocą języka XML. Format XHSTT jest skomplikowany, dlatego w niniejszej pracy poświęcono 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 rozwiązaniami. Każda grupa rozwiązań

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

Rysunek 4.1: Szablon dokumentu w archiwum XHSTT.

```
<HighSchoolTimetableArchive Id="MyArchive">
    <Instances>
        <Instance>
            ...
        </Instance>
        <Instance>
            ...
        </Instance>
    </Instances>
</HighSchoolTimetableArchive>
```

Rysunek 4.2: Przykład archiwum w języku XML.

zawiera rozwiązania dla jednej instancji z archiwum. Ogólny szablon dokumentu przedstawiono na rysunku 4.1. W tym szablonie, słowa znajdujące się w tym samym wierszu oznaczają, że pierwsze słowo 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 opcjonalny. Znak * oznacza, że dana kategoria może wystąpić zero lub więcej razy. Przykład dokumentu w formacie XML zaprezentowano na rysunku 4.2.

Opcjonalna kategoria *Metadata* zawiera podstawowe informacje na temat archiwum, takie jak nazwa archiwum, 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). Postać instancji zaprezentowano na rysunku 4.3. Wiele kategorii ma atrybut *Id*, który służy do odwoływania się do danej kategorii w dalszej części archiwum.

4.3 Okna czasowe

W formacie XHSTT przyjęto tylko prosty model czasu, w którym czas podzielony jest na równe interwały zwane oknami czasowymi. Kategoria *Times* (rysunek 4.4) służy do definiowania okien. Podkategoria *TimeGroups* definiuje różne grupy cza-

```
Instance Id
    MetaData
    Times
    Resources
    Events
    Constraints
```

Rysunek 4.3: Szablon pojedynczej instancji.

```
Times
    +TimeGroups
    *Time

TimeGroups
    *Week
    *Day
    *TimeGroup

Time Id
    Name
    +Week
    +Day
    +TimeGroups
```

Rysunek 4.4: Składnia kategorii *Times*.

sowe, takie jak dni, tygodnie oraz okresy w ciągu dnia. Podkategoria *Time* zawiera definicje okna czasowego, w której każde okno oprócz nazwy może mieć przypisany konkretny tydzień, dzień i inne grupy czasowe. 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 dodatkowo format XHSTT umożliwia dowolne definiowanie zasobów. W skład kategorii *Resources* (rysunek 4.6) wchodzą również grupy i typy zasobów. Typ zasobów to np. nauczyciel, sala, grupa. Grupy zasobów gromadzą zasoby jednego typu, np. nauczyciele języka polskiego.Na

```
<Day Id="Monday">
    <Name>Monday</Name>
</Day>
<TimeGroup Id="BeforeLunch">
    <Name>BeforeLunch</Name>
</TimeGroup>
<Time Id="Mon4">
    <Name>Mon4</Name>
    <Day Reference="Monday"/>
    <TimeGroups>
        <TimeGroup Reference="BeforeLunch"/>
    </TimeGroups>
</Time>
```

Rysunek 4.5: Przykład deklaracji czwartego okna czasowego w poniedziałek.

rysunku 4.7 zaprezentowano przykładową deklarację sali komputerowej.

4.5 Zdarzenia

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

4.6 Ograniczenia

Format XHSTT umożliwia definiowanie wielu ograniczeń dotyczących planu zajęć. Wszystkie ograniczenia opisane są szczegółowo na stronie internetowej [5]. W niniejszej pracy zostały opisane tylko te, które są uwzględnione w wykonanej imple-

```
Resources
    +ResourceTypes
    +ResourceGroups
    *Resource

ResourceType Id
    Name

ResourceGroup Id
    Name
    ResourceType

Resource Id
    Name
    ResourceType
    +ResourceGroups
```

Rysunek 4.6: Składnia kategorii *Resources*.

```
<Resource Id="r03">
    <Name>r03</Name>
    <ResourceType Reference="Room"/>
    <ResourceGroups>
        <ResourceGroup Reference="LargeRoom"/>
        <ResourceGroup Reference="ComputerLab"/>
    </ResourceGroups>
</Resource>
```

Rysunek 4.7: Przykład deklaracji zasobu w postaci sali komputerowej.

```
Events
    +EventGroups
    *Event

EventGroup Id
    Name

Event Id +Color
    Name
    Duration
    +Course
    +Time
    +Resources
    +ResourceGroups
    +EventGroups
```

Rysunek 4.8: Składnia kategorii *Events*.

```
<Event Id="Event_Ang_01">
    <Name>Ang_01</Name>
    <Duration>2</Duration>
    <Course Reference="AngCourse"/>
    <Resources>
        <Resource Reference="OS2">
            <Role>Clas</Role>
        </Resource>
        <Resource Reference="HIL">
            <Role>Teacher</Role>
        </Resource>
        <Resource Reference="Room_22">
            <Role>Room</Role>
        </Resource>
    </Resources>
    <EventGroups>
        <EventGroup Reference="gr_EventsGeneral"/>
        <EventGroup Reference="gr_EventsDuration2"/>
    </EventGroups>
</Event>
```

Rysunek 4.9: Przykład deklaracji zajęć z języka angielskiego.

```
AnyConstraint Id
    Name
    Required
    Weight
    CostFunction
    AppliesTo
```

Rysunek 4.10: Składnia kategorii *Constraints*.

mentacji. Ogólna składnia ograniczenia przedstawiona jest na rysunku 4.10. Każde z ograniczeń ma zdefiniowaną wagę oraz sposób wyliczania funkcji kosztu (liniowa, kwadratowa itp.). Każde ograniczenie może być twarde lub miękkie w zależności od parametru *Required*.

- Przypisany czas - Ograniczenie *Assign time constraints* mówi, że każde zdarzenie musi mieć przypisany czas, oraz definiuje koszt złamania tego ograniczenia. Dane ograniczenie może być przypisane do wszystkich wydarzeń, lub tylko do wybranych grup zdarzeń. Przykład przedstawiono na rysunku 4.11.
- Unikanie konfliktów - Ograniczenie *Avoid clashes constraints* określa, czy dane zasoby mogą być dzielone. To znaczy, czy nie są one przypisane do dwóch lub więcej zdarzeń odbywających się w tym samym czasie. Jest to ograniczenie często stosowane, jednak mogą zdarzyć się przypadki, w których układający dopuszcza odbywanie się dwóch zajęć w jednej sali jednocześnie (np. zajęcia wychowania fizycznego).
- 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 definiować minimalne i maksymalne długości bloków danego zdarzenia.
- Limit bezczynności - Ograniczenie *Limit idle times constraints* określa, czy mogą występować, i jak długie mogą być przerwy między zajęciami. Ograniczenie określa, czy między dwoma zajęciami jednego dnia może wystąpić jedno lub więcej nieprzypisanych okien czasowych.

```
<AssignTimeConstraint Id="AssignTimes_2">
    <Name>AssignTimes</Name>
    <Required>true</Required>
    <Weight>1</Weight>
    <CostFunction>Linear</CostFunction>
    <AppliesTo>
        <EventGroups>
            <EventGroup Reference="gr_EventsGeneral"/>
        </EventGroups>
    </AppliesTo>
</AssignTimeConstraint>
```

Rysunek 4.11: Przykład ograniczenia *Assign time constraints*.

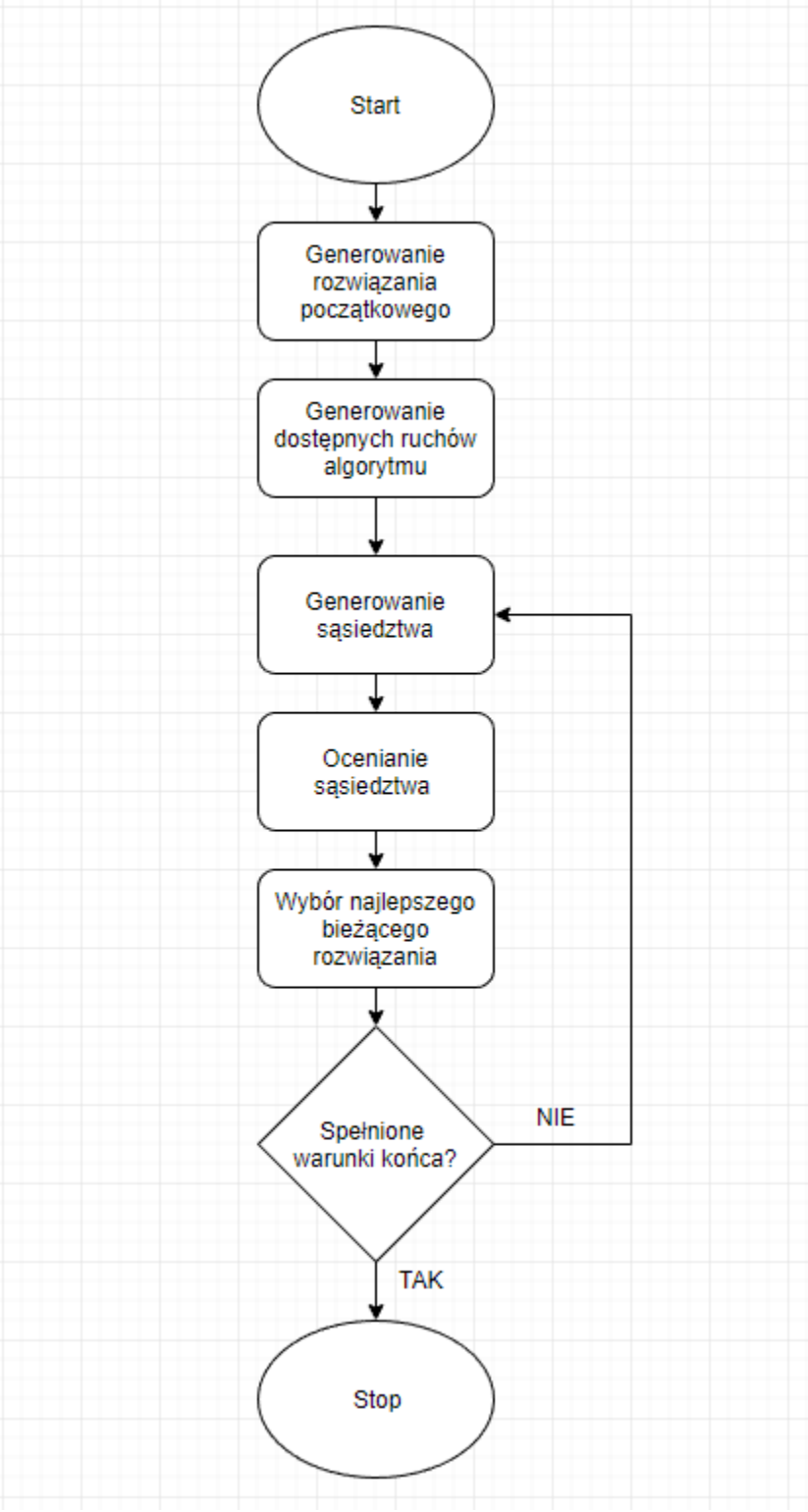
Rozdział 5

Algorytm wyszukiwania z tabu dla problemu układania planu zajęć

Problem układania planu zajęć można próbować rozwiązać za pomocą algorytmu wyszukiwania z tabu. Jednak przystosowanie algorytmu do tego celu jest trudne, z uwagi na dużą złożoność problemów układania planu zajęć. Rozmiar generowanego sąsiedztwa lub stopień złożoności funkcji oceny, to tylko niektóre z nich. Poniżej omówiona została idea przystosowania algorytmu wyszukiwania z tabu do rozwiązywania problemu układania planu zajęć. Na rysunku 5.1 przedstawiono schemat przystosowanego algorytmu.

5.1 Generowanie rozwiązania początkowego

Algorytm wyszukiwania z tabu w celu znalezienia najlepszego rozwiązania musi rozpocząć się od pewnego rozwiązania początkowego. Warto przypomnieć, że algorytm wyszukiwania z tabu jest algorytmem heurystycznym. W zależności od tego,z jakim rozwiązaniem początkowym rozpocznie on swoje działanie, może uzyskać różne rezultaty po określonej liczbie iteracji. Stworzenie początkowego planu zajęć, w którym wszystkie zajęcia rozpoczynałyby się np. w poniedziałek, w pierwszym dostępnym oknie czasowym, mogłoby spowodować znaczne wydłużenie czasu,pow którym algorytm znajdzie dostatecznie dobre rozwiązanie. Wydłużony czas wynikał by z niskiej oceny takiego planu i dłuższej „drogi", jaką algorytm musi pokonać,



Rysunek 5.1: Schemat działania algorytmu wyszukiwania z tabu do rozwiązywania problemu układania planu zajęć.

by dojść do dobrego rozwiązania. Zdecydowano się więc na wprowadzenie elementu probabilistycznego, by zróżnicować początkowy plan zajęć. Każdemu wydarzeniu przypisywane jest losowe okno czasowe. Dobrym pomysłem byłoby zastosowanie algorytmu, który utworzył by początkowy plan zajęć z jeszcze lepszą oceną. Jednak, z uwagi na ograniczony czas implementacji, zdecydowano się pozostać przy rozwiązaniu losowym.

5.2 Generowanie dostępnych ruchów algorytmu

By ułatwić generowanie sąsiedztwa niezbędnego w dalszym działaniu algorytmu, zdecydowano się na generowanie wszystkich dostępnych ruchów dla danego kroku algorytmu. Przez ruch rozumiana jest para: wydarzenie i okno czasowe. Oznacza to, że wybranemu wydarzeniu przyporządkowane zostaje określone okno czasowe. Generowane są więc wszystkie pary, których liczba równa jest liczbie wydarzeń pomnożonej przez liczbę okien czasowych. Tak zdefiniowany ruch, może być w łatwy sposób wykorzystany podczas generacji sąsiedztwa bieżącego rozwiązania (tj. planu zajęć). 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 reprezentować ruch jako parę indeksów i przechowywać go na liście tabu. Liczba wszystkich dostępnych ruchów określa rozmiar sąsiedztwa, które algorytm powinien przeszukać w danym kroku. Ponieważ liczba ta jest duża, konieczne było ograniczenie przeszukiwanego sąsiedztwa.

5.3 Generowanie i ocenianie sąsiedztwa dla wybranego rozwiązania

W ramach jednej iteracji, algorytm powinien wygenerować sąsiedztwo dla bieżącego rozwiązania, ocenić każdego sąsiada i wybrać najlepszego z nich. Ponieważ obliczenie funkcji oceny rozwiązania jest bardzo kosztowne, konieczne było ograniczenie rozmiaru przeszukiwanego sąsiedztwa. Rozmiar ten jest jednym z parametrów algorytmu. Po raz kolejny zdecydowano się wprowadzić element probabilistyczny. Z puli dostępnych ruchów wybierane są ruchy w sposób losowy. Każdy z wylosowa-

a)

Godzina	Zajęcia
08:00	A
08:50	B
09:45	C
10:45	D
11:40	

b)

Godzina	Zajęcia
08:00	A
08:50	B
09:45	
10:45	D
11:40	C

c)

Godzina	Zajęcia
08:00	A|B
08:50	
09:45	C
10:45	D
11:40	

Rysunek 5.2: Przykład generacji sąsiedztwa: a) bieżące rozwiązanie, b) i c) przykładowe sąsiedztwo rozwiązania bieżącego.

nych ruchów wykonywany jest dla aktualnego rozwiązania, tworząc nowe rozwiązanie, wchodzące w skład sąsiedztwa rozwiązania aktualnego. Przykład generacji sąsiedztwa pokazano na rysunku 5.2. Tak wygenerowane sąsiedztwo jest oceniane i wybierane jest rozwiązanie najlepsze spośród dostępnych rozwiązań. 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 nowe rozwiązanie jest lepsze, to staje się rozwiązaniem najlepszym globalnie. Ostatnim krokiem w ramach jednej iteracji algorytmu jest aktualizowanie listy Tabu. Dla każdego elementu aktualizowany jest czas jego trwania na liście tabu. Jeżeli jest on równy zeru, to ruch trafia z powrotem do zbioru dostępnych ruchów algorytmu.

5.4 Funkcja oceny rozwiązania

Najtrudniejszym elementem implementacji była funkcja oceny rozwiązania (tj. planu zajęć). Z uwagi na konieczność sprawdzania wielu warunków w stosunku do wielu elementów, jest to najbardziej czasochłonna część wykonania algorytmu. Wszystkie ograniczenia, które ma spełniać otrzymany plan zajęć muszą być badane w tej funkcji. Funkcja ta decyduje o jakości otrzymanego planu i daje duże możliwości, by w przyszłości ją udoskonalić.

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 ma przyporządkowaną określoną wartość kary, która jest dodawana do końcowej oceny rozwiązania za każdym razem, gdy ograniczenie nie jest spełnione.

Każde zdarzenie musi mieć przypisane okno czasowe. Podczas oceny tego ograniczenia, sprawdzane są wszystkie wydarzenia. Za każdym razem, gdy któreś z wydarzeń nie będzie miało przypisanego okna czasowego, nałożona zostanie odpowiednia kara. Ograniczenie to miało zastosowanie tylko w początkowej fazie implementacji algorytmu, kiedy istniała konieczność sprawdzania poprawności wykonania algorytmu. W dalszej fazie implementacji zdecydowano, że każde zdarzenie już podczas generacji rozwiązania początkowego ma przypisane okno czasowe, a nie ma możliwości, by algorytm usunął przypisane okno czasowe. Jedynymi zmianami może być przypisanie nowego okna czasowego. Z tego powodu ograniczenie to zostało pominięte.

Ograniczenie związane z unikaniem konfliktów między zasobami 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 ono wielokrotnego sprawdzania zasobów.

Ograniczenie dotyczące unikania dzielenia zajęć trwających dłużej niż jedno okno czasowe może być złamane np. 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. Oczywiście nie jest to pożądane. By wykryć złamanie tego ograniczenia konieczne jest

sprawdzenie wszystkich zajęć o czasie trwania dłuższym niż jedno okno czasowe. 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 wyżej wymienione ograniczenia można było określić jako ograniczenia twarde. Kara za ich złamanie jest bardzo wysoka. Ostatnie ograniczenie jest ograniczeniem miękkim. Kara za jego złamanie to około 10% wartości kary poprzednich ograniczeń. Minimalizacja pustych okien czasowych między zdarzeniami dla grup uczniów nie jest obowiązkowa aby plan był poprawny. Brak tych okien jest jednak pożądany przez uczniów. By sprawdzić to ograniczenie konieczne jest przeszukanie planu dla każdej klasy z osobna. Jeżeli dla danej klasy, w danym dniu, wystąpiły już zajęcia, a następnie wystąpiło puste okno czasowe, a po nim kolejne zajęcia, to nałożona zostanie kara. Kara jest zależna od liczby pustych okien czasowych między zajęciami.

Rozdział 6

Program układania planu zajęć

Implementacja algorytmu została napisana w języku C#. Język ten został wybrany z uwagi na dużą liczbę bezpłatnych bibliotek oraz możliwość dostępu do bardzo dobrej dokumentacji technicznej. Nie bez znaczenia był także fakt dużej popularności języka, co skutkuje większymi możliwościami uzyskania pomocy na forach programistycznych.

6.1 Użyte narzędzia

Podczas implementowania algorytmu wyszukiwania z tabu użyto następujących narzędzi:

- 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].

- Biblioteka Linq - Language-Integrated Query (LINQ) pozwala odczytywać dane pochodzące z różnych źródeł w postaci obiektów. Biblioteka udostępnia funkcje filtrujące i wyszukujące, które odpowiednio użyte w znacznej mierze

przyspieszają działanie programu. W implementacji algorytmu, którego dotyczy praca, zapytania Linq ułatwiają komunikację z bazą danych oraz czytanie dokumentów XML.

- Windows Forms - interfejs programowania graficznych aplikacji należący do środowiska *.NET Framework*. Służy do tworzenia aplikacji z graficznym interfejsem użytkownika, umożliwiająca obsługę zdarzeń napływających od użytkownika. Obecnie interfejs wypierany jest przez interfejs Windows Presentation Foundation (*WPF*). Jednak mimo tego, dzięki swej prostocie i przejrzystości, jest dalej używany przez programistów.

- Entity Framework - jest to bezpłatne 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, umożliwiające 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 debagowanie kodu, oraz pozwala szybko tworzyć nowe aplikacje. Jest to naturalny wybór dla języka programowania C#.

- SQL Server 2014 Managment Studio - zintegrowane środowisko do zarządzania bazami danych. Zawiera narzędzia do konfigurowania i monitorowania baz danych. Umożliwia budowę zapytań i oglądanie tabel w bazie danych. Autor pracy wykorzystał to środowisko do testowania poprawności wykonanych operacji i analizy uzyskanych wyników.

6.2 Wprowadzanie danych

Pierwszym problemem, który stanął przed autorem tej pracy, było wprowadź danych, korzystając z formatu XHSTT. Format ten został omówiony w rozdzia-

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

Rysunek 6.1: Przykład wprowadzania instancji problemu układania planu zajęć za pomocą biblioteki *Xml.Linq*.

le czwartym. Ponieważ dane zawarte w tym formacie zapisane są w języku XML, skorzystano z biblioteki systemowej *Xml.Linq*, która pozwala traktować dokument XML jako bazę danych i wysyłać do niej stosowne zapytania. Przykład wprowadzania instancji problemu układania planu zajęć za pomocą tej biblioteki przedstawiony został na rysunku 6.1. Biblioteka *Xml.Linq* stanowiła spore ułatwienie, jednak w dalszym ciągu trudności sprawiał złożony charakter dokumentu, ponieważ każde zagnieżdżenie musiało być wczytane za pomocą odpowiedniego wiersza kodu.

Wprowadzone dane zapisywane są w bazie danych, dlatego każde kolejne wykonanie aplikacji umożliwia pracę na już wczytanych danych. Jak stwierdzono w poprzednich rozdziałach, założono, że wprowadzone dane zawierają zdarzenia z przypisanymi wcześniej zasobami, takimi jak nauczyciel, sala i grupa.

6.3 Specyfikacja wewnętrzna programu

Poprzedni podrozdział miał na celu przedstawienie pomysłu i sposobu w jaki zaimplementowany został algorytm wyszukiwania z tabu. Ten podrozdział zawiera szczegóły techniczne implementacji, opis użytych narzędzi, używanych klas oraz zawiera najistotniejsze fragmenty kodu.

6.3.1 Opis klas i ważniejszych funkcji

Aplikacja zapisana jest w kilku folderach. Każdy z folderów zawiera klasy realizujące odrębne funkcje aplikacji. Folder *Code* zawiera klasy realizujące algorytm wyszukiwania. W folderze *Data* znajdują się klasy służące do translacji 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 realizujące przygotowanie danych to zaprezentowania ich w interfejsie graficznym. Poza folderami znajduje się osobna klasa *Form1*, która obsługuje interfejs graficzny użytkownika.

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

- *Code/SolutionManager* - klasa zawiera szkielet algorytmu wyszukiwania z tabu. Jest to główna klasa algorytmu. Znajdują się w niej funkcja rozwiązująca problem układania planu zajęć dla instancji o podanym identyfikatorze. Klasa ta zawiera również funkcje zapisującą raport działania algorytmu do pliku txt oraz funkcję wyświetlającą rozwiązanie na konsolę. Główna funkcja klasy zaprezentowana została na rysunku 6.2.

- *Code/TabuSearch* - klasa zawiera statyczne funkcje służące do rozwiązywania problemu za pomocą algorytmu wyszukiwania z tabu. Znajdują się tu funkcję 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 wyszukiwania z tabu. Funkcja generująca sąsiedztwo i wybierająca z niego najlepszą instancje została zaprezentowana na rysunku 6.3.

- *Code/EvaluationFunction* - w skład tej klasy wchodzą funkcje oceniające bieżące rozwiązanie uzyskane przez algorytm. Znajduje się tu implementacja wszystkich ograniczeń, jakie mają być uwzględnione w rozwiązaniu. Klasę tę można w łatwy sposób rozszerzyć dodając kolejne funkcje oceniające, które odpowiadają kolejnym ograniczeniom.

- *Code/TabuItem* - klasa reprezentująca pojedynczy ruch w algorytmie wyszukiwania z tabu. Jako ruch rozumiana jest para dwóch indeksów, pierwszy z nich odpowiada identyfikatorowi zdarzenia, a drugi to identyfikator okna czasowego, które w danym ruchu jest do tego zdarzenia przypisane. Elementy 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); //wczytanie danych z bazy.

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

    //Generowanie rozwiązania początkowego
    TabuSearch.GenerateStartSolutionForTimes(actualInstance);
    //Generowanie dostępnych ruchów
    availableList = TabuSearch.GenerateAvaiableMoveList(actualInstance);
    //Ocena rozwiązania startowego
    bestRating = EvaluationFunction.EvaluateInstance(actualInstance);
    for (int i = 0; i < iteration; i++)
    {
        //Generacja sąsiedztwa i wybór najlepszego sąsiada.
        var tempInstance = TabuSearch.SelectBestInstanceFromNeighborhood
                                (actualInstance, tabuList, availableList);
        if (tempInstance != null)
            actualInstance = tempInstance;

        //Tworzenie raportu i porównanie najlepszych rozwiązań.
        int rating = EvaluationFunction.EvaluateInstance(actualInstance);
        raport.Add(rating);
        if (bestRating > rating)
        {
            bestInstance = actualInstance;
            bestRating = rating;
        }

        //Warunek końca.
        if (bestRating <= 0)
        {
            resultIteration = i;
            break;
        }

        TabuSearch.UpdateTabuList(tabuList, availableList);
    }

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

Rysunek 6.2: Funkcja *ResolveSimpleProblem* zawierająca szkielet algorytmu Tabu search.

```
public static Instance SelectBestInstanceFromNeighborhood
(Instance instance, List<TabuItem> tabuList, List<TabuItem> availableList)
{
    Random r = new Random();
    List<Instance> neighborhood = new List<Instance>();
    List<TabuItem> usedItems = new List<TabuItem>();
    //Generacja sąsiedztwa
    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.Diagnostics.Debug.WriteLine("Non moves available.");
        }
    }

    //Wybranie najelpszego sąsiada
    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 6.3: Funkcja *SelectBestInstanceFromNeighborhood* generująca sąsiedztwo i wybierająca najlepszego sąsiada.

- *Data/XMLLoader* - w tej klasie zawarte są funkcje przekształcające dane wejściowe w formacie XHSTT, zapisane w języku XML, na klasy modelu. Znajduje się tu również funkcja zapisująca odczytaną instancje do bazy danych.

- *Form1* - klasa obsługująca interfejs użytkownika. Zawiera funkcje reagujące na każdą wykonaną przez użytkownika czynność oraz funkcje wypisujące dane do odpowiednich obiektów interfejsu.

6.4 Specyfikacja zewnętrzna programu

Po zainicjowaniu działania aplikacji wyświetlane jest okno główne przedstawione na rysunku 6.4. Przy pierwszym wykonaniu aplikacji użytkownik powinien wczytać instancje z pliku wejściowego. Może to wykonać za pomocą przycisku „Load instance from file", który znajduje się w prawym górnym rogu okna. Otworzy się okno dialogowe, gdzie należy wskazać plik XML zawierający dane wejściowe dla problemu rozwiązywania planu zajęć, zapisane w formacie XHSTT. Po wczytaniu wielu instancji (lub po ponownym zainicjowaniu działania aplikacji), użytkownik może wybrać instancje do rozwiązania. Służy do tego rozwijana lista w lewym górnym rogu okna.

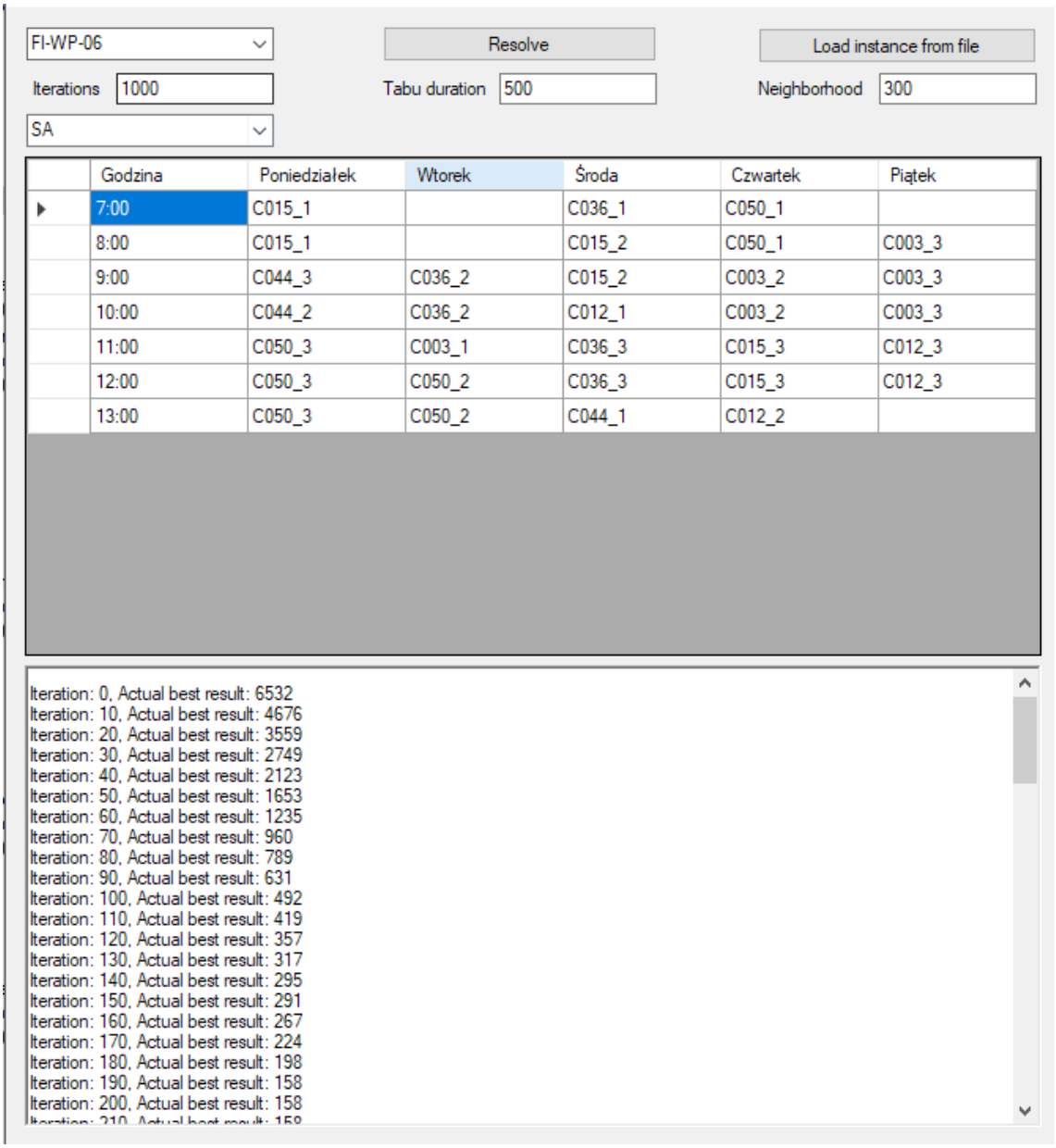
Przycisk „Resolve" służy do wykonania algorytmu. Działanie algorytmu jest monitorowane, a postępy wyświetlane są na konsoli na dole okna aplikacji. Po zakończeniu obliczeń, ułożony plan zajęć można oglądać w tabeli. Rozwijana lista nad tabelą służy do wyboru klasy, dla której ma zostać zaprezentowany plan zajęć.

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

- Iterations - maksymalna liczba iteracji jaką algorytm wykona w czasie rozwiązywania problemu układania planu zajęć. Jeżeli algorytm wcześniej znajdzie rozwiązanie, na które nie jest nałożona żadna kara, to zakończy swoje działanie.

- Tabu duration - parametr oznaczający czas pozostawania pojedynczego ruchu na liście tabu. Długość mierzona jest liczbą 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, które może wykonać algorytm (pełne sąsiedztwo).



Rysunek 6.4: Okno główne aplikacji.

Rozdział 7

Testowanie aplikacji

7.1 Środowisko testowe

Aplikacja została zrealizowana i testowana na komputerze stacjonarnym z systemem operacyjnym Windows 10. Wyposażenie komputera było następujące:

- procesor Intel Core i5-4670K 3.40 GHz,
- karta graficzna Nvidia GeForce GTX 770,
- pamięć operacyjna RAM 8 GB DDR3,
- dysk twardy SSD Crucial CT120M.

7.2 Zestaw danych testowych

Jako zestaw danych testowych, z pomocą których testowany był algorytm układania planu zajęć, wybrany został zestaw fińskiej szkoły ponadpodstawowej. Oparty on został na danych z roku 2006 szkoły West-Pori High School, w której wiek uczniów należał do przedziału 16-19 lat. Zestaw został wybrany z uwagi na rozmiar danych oraz duże podobieństwo siatki zajęć do polskiego gimnazjum.

Wyżej wymieniony 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 łącznego czasu trwania wydarzeń do całkowitej liczby dostępnych okien czasowych dla wszystkich grup (35x10). Analizując pokrycie można określić stopień trudności wykonania danego 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 wynosi średnio 75%.

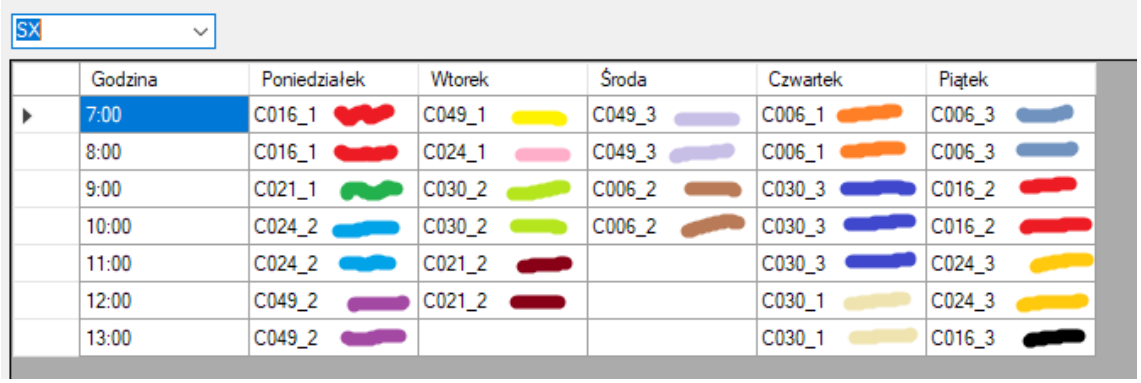
Testowany zestaw danych wejściowych zawiera cztery ograniczenia:

- przypisany czas dla wszystkich zdarzeń,
- unikanie konfliktów zasobów,
- brak podziału zdarzeń,
- ograniczenie bezczynności uczniów (okienek w planie).

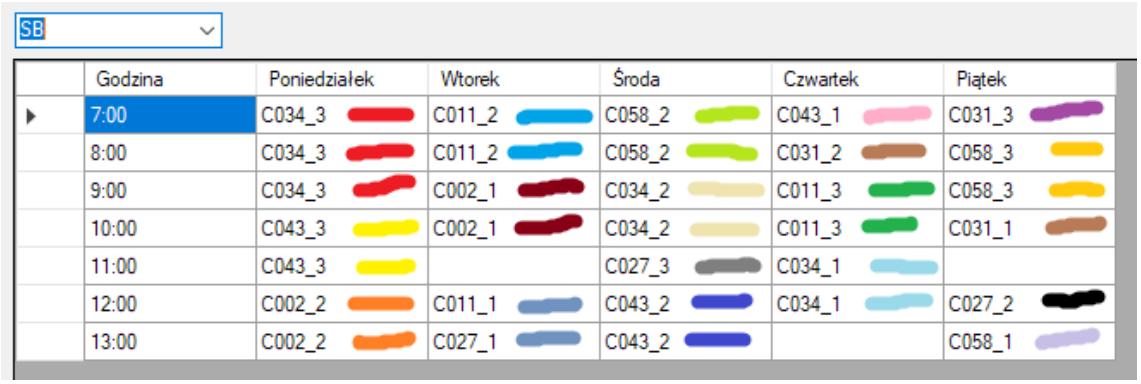
Ograniczenia te zostały szczegółowo omówione w rozdziale 4.

7.3 Wyniki działania aplikacji

Aplikacja została wykonana wielokrotnie dla różnych parametrów. Średni czas pracy aplikacji wynosił ok. 8 minut. Najlepszy wynik, jaki udało się uzyskać to



Rysunek 7.1: Najlepszy otrzymany plan zajęć spośród planów dla dziesięciu klas.

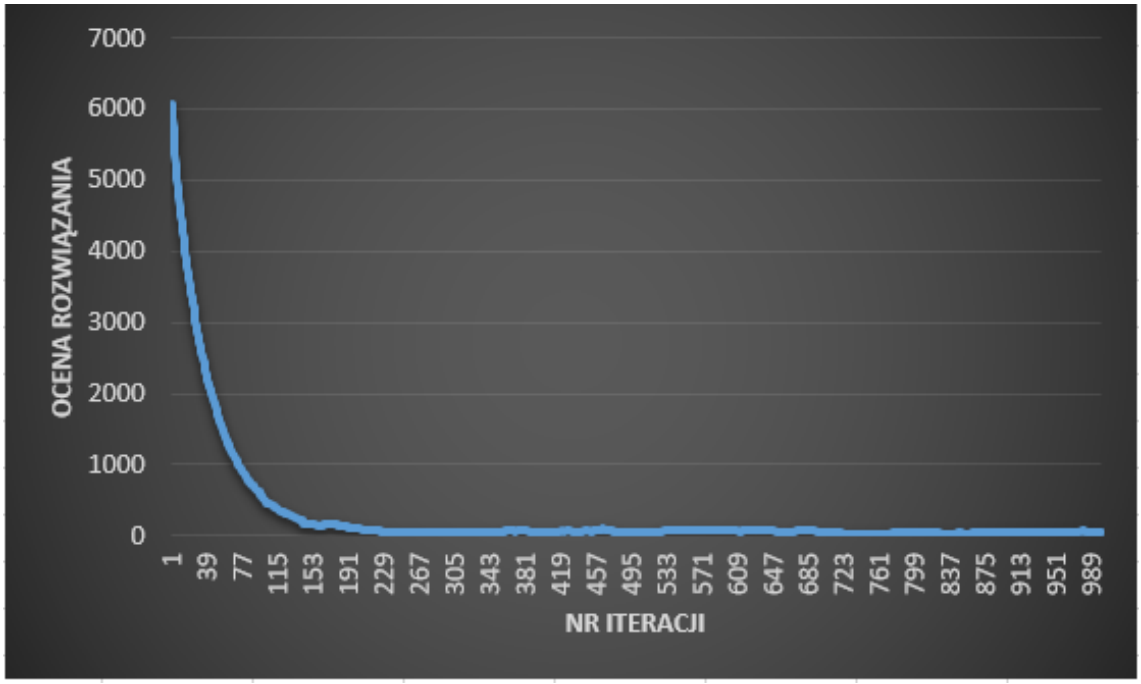


Rysunek 7.2: Najgorszy otrzymany plan zajęć spośród planów dla dziesięciu klas (dwa okienka).

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 czasowych między zajęciami), co daje wynik około jednego okienka na dwie klasy uczniów. Jest to wynik bardzo dobry. Na rysunkach 7.1 i 7.2 przedstawiono najlepszy i najgorszy plan dla różnych klas uczniowskich w tym rozwiązaniu. Kolory ilustrują zajęcia o czasie trwania większym niż jedno okno czasowe.

Wykres na rysunku 7.3 przedstawia drogę algorytmu jaką musiał pokonać w czasie swojej pracy. Na osi pionowej przedstawiona jest ocena aktualnego rozwiązania a na osi poziomej liczba iteracji.

Najlepszy wynik otrzymany został dla następujących parametrów algorytmu: liczba iteracji - 1000, czas trwania tabu - 500, rozmiar sąsiedztwa - 300. Liczba iteracji algorytmu została ustalona, biorąc pod uwagę czas pracy programu oraz to,

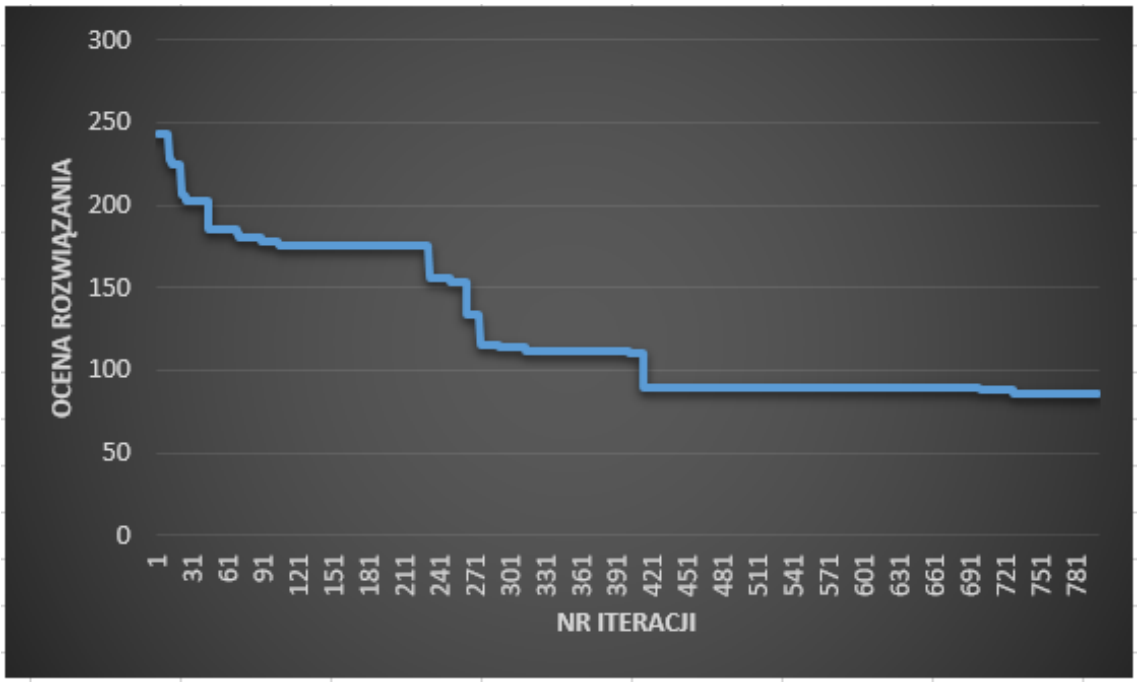


Rysunek 7.3: Przebieg pracy algorytmu.

że liczba ta zazwyczaj wystarczała, by usunąć wszystkie konflikty z planu zajęć.

Rozmiar sąsiedztwa również w dużej mierze wpływa na czas pracy algorytmu. Doświadczenia pokazały, że im większe sąsiedztwo tym mniej iteracji potrzebnych jest do uzyskania lepszych wyników. Rozmiar 300 to około 5% całego sąsiedztwa możliwego do przeszukania. Przyjęcie rozmiaru sąsiedztwa na 100%, spowodowało by wydłużenie pracy algorytmu dwudziestokrotnie. Ustalony rozmiar jest próbą pogodzenia oczekiwań, jak najkrótszego czasu pracy algorytmu i jak najlpeszych wyników.

Czas trwania tabu zmienia całkowicie sposób pracy algorytmu. Ustalenie wartości 500 wynikało z obserwacji wyników, jakie uzyskał algorytm oraz wykresów obrazujących jego pracę. Szczegółowe omówienie wpływu parametru czasu trwania tabu przedstawiono w kolejnym podrozdziale.



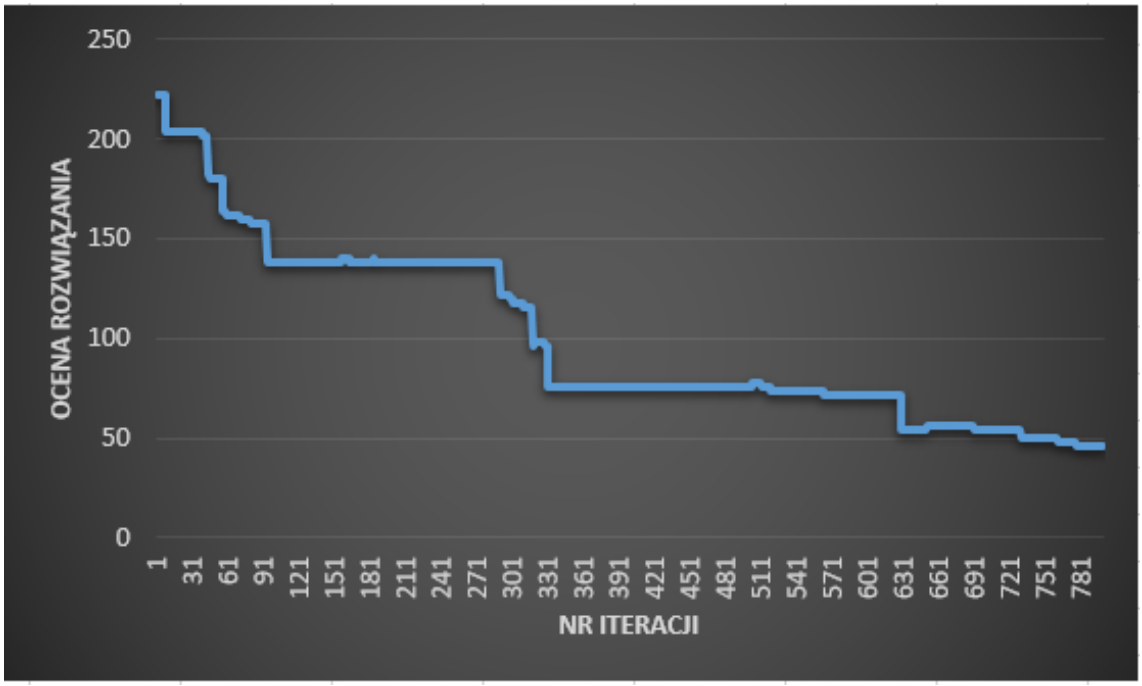
Rysunek 7.4: Przebieg pracy algorytmu dla czasu trwania tabu = 0.

7.4 Wpływ czasu trwania tabu na uzyskane wyniki.

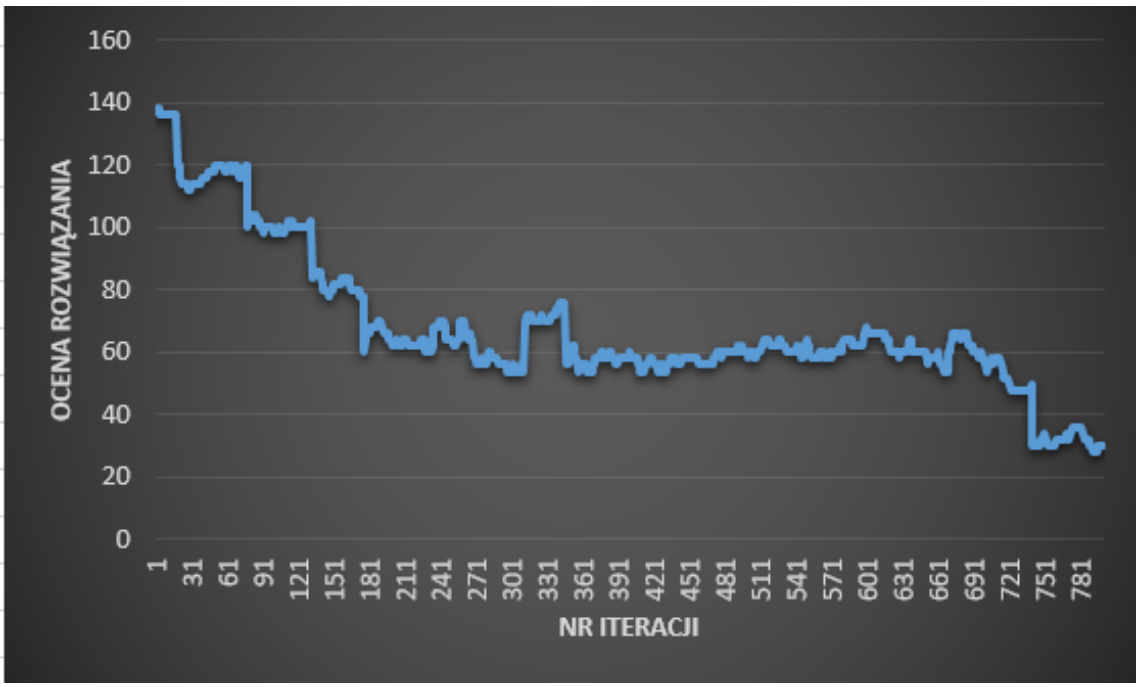
Algorytm wyszukiwania z tabu opiera się na algorytmie przeszukiwania lokalnego, w który stosuje się listę tabu. Parametr określający czas pozostawania ruchów na liście tabu ma duże znaczenie. Zmiana tego parametru powoduje zupełnie inną pracę algorytmu. Potwierdzeniem tego są wykresy przedstawione na rysunkach 7.4 - 7.8.

Na wykresach przedstawiono przebieg pracy algorytmu dla parametrów: 1000 iteracji i rozmiar sąsiedztwa równy 300. Zmieniał się tylko parametr długości tabu. Jak można zauważyć na rysunku 7.3, algorytm w początkowej fazie pracy znacznie poprawia swój wyniki (zmniejsza ocenę rozwiązania z 6000 do ok. 200). Z tego powodu na analizowanych obecnie wykresach przedstawiono tylko ostatnie 800 iteracji pracy algorytmu, by obciąć początkowy „skok” i móc zaobserwować zmiany pracy algorytmu w mniejszej skali.

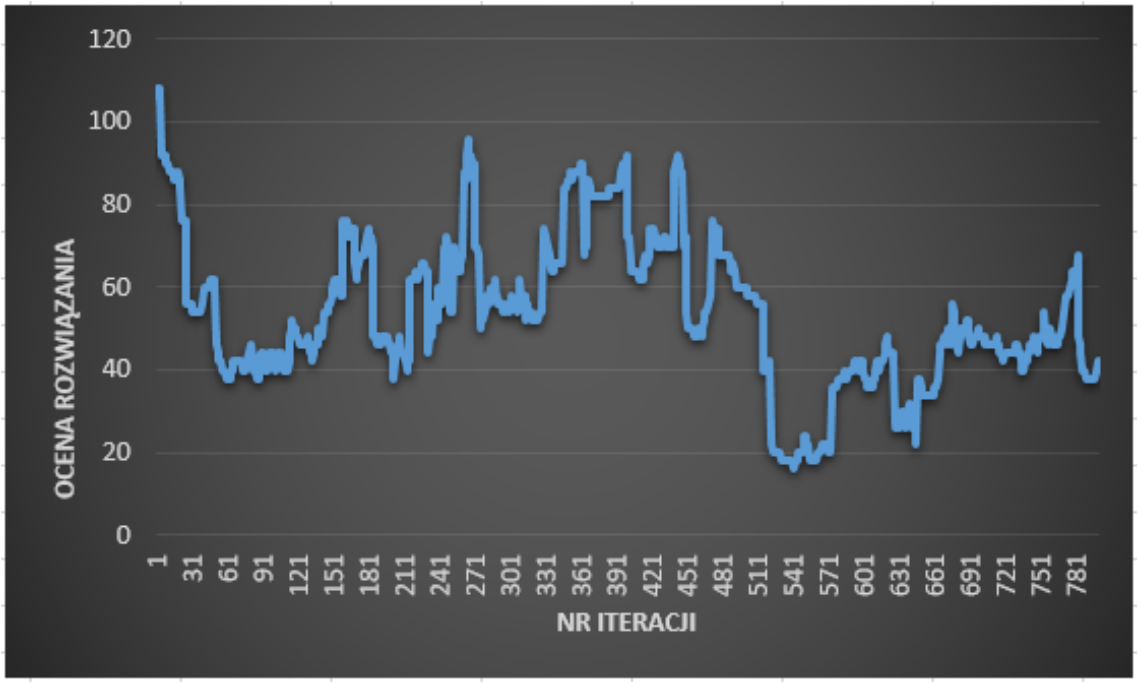
Pierwszy z wykresów (rysunek 7.4) prezentuje pracę algorytmu dla długości tabu



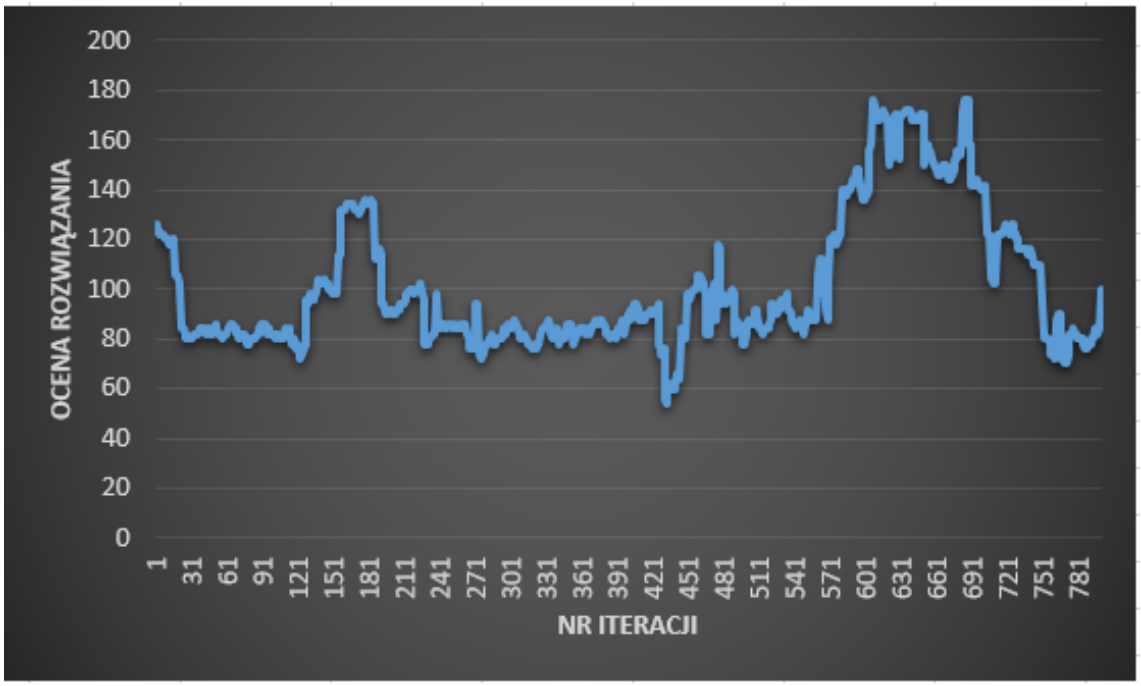
Rysunek 7.5: Przebieg pracy algorytmu dla czasu trwania tabu = 100.



Rysunek 7.6: Przebieg pracy algorytmu dla czasu trwania tabu = 300.



Rysunek 7.7: Przebieg pracy algorytmu dla czasu trwania tabu = 500.



Rysunek 7.8: Przebieg pracy algorytmu dla czasu trwania tabu = 800.

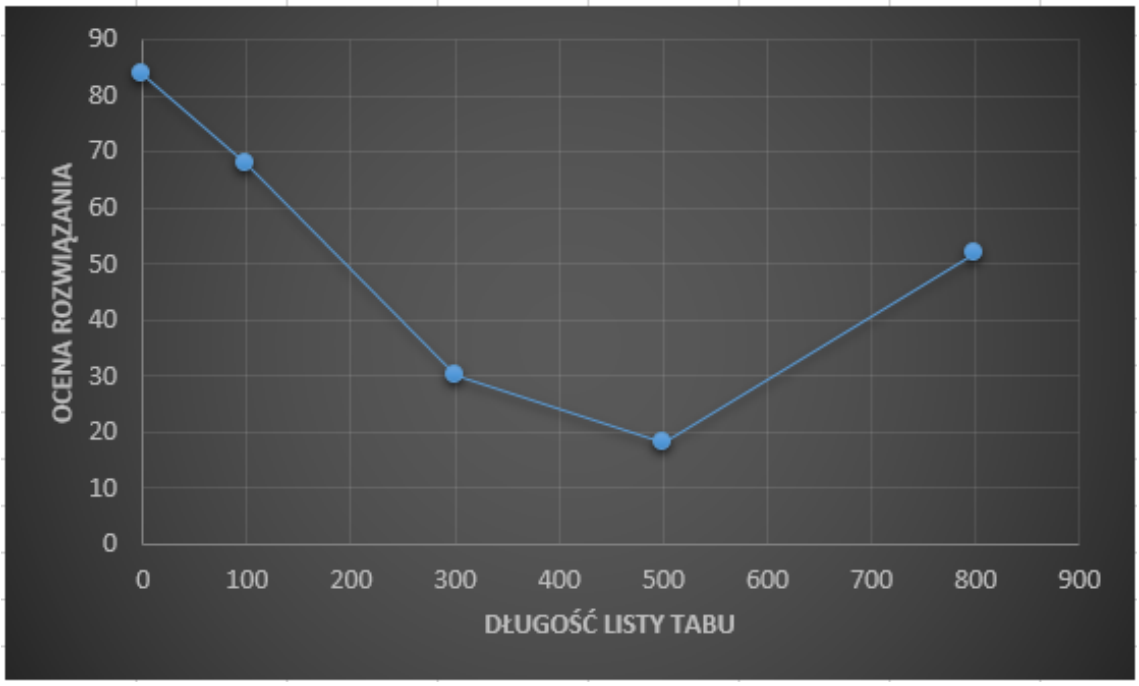
równej 0. Oznacza to, że algorytm w ogóle nie korzysta z listy tabu. Jest to więc klasyczne przeszukiwanie lokalne. Jak widać, algorytm przeszukuje rozwiązania, które mają tą samą ocenę lub wyższą. Na wykresie nie można zauważyć skoku oceny wartości „w górę". Jest to spowodowane wpadaniem algorytmu w lokalne minima, a poprawa rezultatu najczęściej jest skutkiem innego losowego wyboru sąsiadów. ocena planu jaką udało się osiągnąć to 84.

Drugi wykres (rysunek 7.5) przedstawia przebieg pracy algorytmu dla długości tabu równej 100. Otrzymany wykres jest bardzo zbliżony do poprzedniego, jednak można zaobserwować nieliczne próby pogorszenia oceny rozwiązania bierzącego w celu dotarcia do rozwiązania lepszego. Najlepszą uzyskaną oceną rozwiązania była ocena 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 7.6) sytuacja wygląda już zdecydowanie lepiej. Długość tabu równa 300 sprawia, że algorytm przeszukuje obszary, do których można dotrzeć tylko przez pogorszenie bieżącego wyniku. Większe skoki oceny rozwiązań to miejsca, w których udało się wyeliminować konflikt zasobów (każdy konflikt nakłada karę 20). Mniejsze zmiany obrazują próby minimalizacji okienek (każde okno czasowo powiększa ocenę rozwiązania o 2). Najlepszy uzyskany wynik to 30.

Czwarty wykres (rysunek 7.7) prezentuje sytuację, w której udało się osiągnąć najlepsze rezultaty (w tym przypadku oceną 18 - rozwiązanie optymalne to ocena 0). Najlepszy wynik udało się osiągnąć już w 740 iteracjach (540 plus 200 iteracji nie przedstawionych na wykresie). Dla długości listy tabu równej 500 przeszukiwanie odbywa się już bardzo „skokowo". Algorytm nie może wykonać 500 ruchów, które w poprzednich iteracjach najbardziej poprawiły wyniki algorytmu. Musi jako bieżące rozwiązanie przyjąć rozwiązanie o gorszej ocenie. Dzięki temu jest w stanie dotrzeć do obszarów, które dla przeszukiwania lokalnego są nieosiągalne.

Ostatni wykres (rysunek. 7.8) przedstawia przebieg algorytmu dla długości tabu równej 800. Przy 1000 iteracjach tylko pierwsze 200 ruchów mogło zostać powtórzone. Niestety, tak duża liczba zabronionych ruchów spowodowała, że algorytm zdecydowanie pogorszył swoje wyniki (najlepszy rezultat to ocena 52). Choć algorytm próbował przejść w inne obszary poszukiwań, to nie otrzymano oczekiwanych



Rysunek 7.9: Stosunek długości listy tabu do uzyskanej oceny rozwiązania - wykres.

Długość listy tabu	0	100	300	500	800
Ocena rozwiązania	84	68	30	18	52

Rysunek 7.10: Stosunek długości listy tabu do uzyskanej oceny rozwiązania - tabela.

rezultatów. Najprawdopodobniej, kluczowe ruchy znajdowały się na liście tabu.

Rysunek 7.9 i 7.10 przedstawiają stosunek parametru długości listy tabu do otrzymanego rozwiązania. Najlepszy rezultat otrzymano dla parametru długości listy tabu równego 500. Ocena rozwiązania optymalnego, którą algorytm starał się osiągnąć jest równa 0.

Rozdział 8

Podsumowanie

Głównym celem niniejszej pracy było zastosowanie algorytmu wyszukiwania z tabu do rozwiązywania problemu układania planu zajęć. Cel ten został zrealizowany. Efektem pracy jest aplikacja, za pomocą której można ułożyć plan zajęć dla problemu sformułowanego z użyciem formatu XHSTT. W aplikacji zaimplementowano cztery ograniczenia planu zajęć spośród dużej liczby dostępnych ograniczeń w formacie XHSTT. Pozostałe ograniczenia nie zostały zaimplementowane z powodu rzadkiego występowania w różnych danych testowych oraz ograniczeń czasowych wykonania niniejszej pracy.

Podczas implementacji algorytmu, autor napotkał wiele trudności. Wszystkie z nich zostały pokonane. Pierwszą z nich było zapoznanie się z formatem danych wejściowych XHSTT, który mimo bardzo dobrej dokumentacji jest formatem stosunkowo złożonym i sprawia, że na problem rozwiązywania planu zajęć należy spojrzeć z innej strony. Kolejną trudnością była implementacja algorytmu ze względu na duży rozmiar danych wejściowych, który powodował kłopoty z weryfikacją poprawności pracy algorytmu. Dlatego zdecydowano się rozpocząć implementację, testując jej działanie na danych nierzeczywistych zawierających małą liczbę zdarzeń i zasobów. Podczas testowania pojawiła się trudność związana z długim czasem oczekiwania na wyniki oraz obiektywnym porównaniem jakości rozwiązań. Zdecydowano, aby generować raporty z pracy algorytmu w celu analizy jego działania w zależności od parametrów wejściowych.

Aspektem badawczym pracy była analiza pracy algorytmu dla różnych warto-

ści długości listy tabu. W wyniku przeprowadzonej analizy pokazano, że najlepsze rezultaty algorytm osiągał dla listy tabu o długości stanowiącej około 4-5% liczby wszystkich dostępnych ruchów. Czas pracy algorytmu zależny jest od rozmiaru danych wejściowych oraz od przyjętych wartości parametrów. Dla problemu układania planu zajęć dla szkoły o rozmiarze polskiego liceum o średniej wielkości oraz właściwie dobranych parametrach, czas pracy algorytmu wynosi około 10 minut.

Zaimplementowany algorytm nadaje się do praktycznego użytkowania i może być wykorzystany dla większości problemów układania planu zajęć zapisanych w formacie XHSTT.

Możliwy jest dalszy rozwój aplikacji przez zaimplementowanie kolejnych ograniczeń oraz poprawnego działania algorytmu dla zajęć o długości dłuższej niż trzy okna czasowe. Osiągnięte rezultaty są satysfakcjonujące, jednak można poczynić starania, aby poprawić algorytm wyszukiwania przez dodanie metod dywersyfikacji i intensyfikacji obszaru poszukiwań do opracowanego algorytmu. Należy się jednak liczyć z tym, że każde dodatkowe ograniczenie wydłuży czas działania algorytmu, co spowoduje konieczność jego optymalizacji.

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.

Dodatek A

Wybrane fragmenty kodu źródłowego

```
0 references | 0 changes | 0 authors, 0 changes
public static void UpdateTabuList
(List<TabuItem> tabuList, List<TabuItem> availableList)
{
    List<TabuItem> toDelete = new List<TabuItem>();
    foreach (var item in tabuList)
    {
        item.TabuDuration--;
        if (item.TabuDuration < 0)
            toDelete.Add(item);
    }
    foreach (var item in toDelete)
    {
        tabuList.Remove(item);
        availableList.Add(item);
    }
}
```

Rysunek A.1: Funkcja aktualizująca listę tabu.

```
0 references | 0 changes | 0 authors, 0 changes
public static void GenerateStartSolutionForTimes(Instance instance)
{
    Random r = new Random();
    foreach (var ev in instance.Events)
    {
        int randomIndex = r.Next(instance.Times.Count);
        ev.Time = instance.Times[randomIndex];
    }
}

0 references | 0 changes | 0 authors, 0 changes
public static List<TabuItem> GenerateAvaiableMoveList(Instance instance)
{
    List<TabuItem> avaiableList = new List<TabuItem>();

    foreach (var itemX in instance.Events)
    {
        foreach (var itemY in instance.Times)
        {
            avaiableList.Add(new TabuItem
            (instance.Events.IndexOf(itemX), instance.Times.IndexOf(itemY), 0));
        }
    }

    return avaiableList;
}
```

Rysunek A.2: Funkcje generujące rozwiązanie początkowe oraz dostępne ruchy.

```
1 reference | 0 changes | 0 authors, 0 changes
static Instance SelectBestInstanceFromGeneratedNeighborhood
(List<Instance> neighborhood)
{
    int bestIndex = -1;
    int bestRating = -1;
    foreach (var instance in neighborhood)
    {
        int tempRating = EvaluationFunction.EvaluateInstance(instance);
        if (bestRating < 0 || bestRating > tempRating)
        {
            bestRating = tempRating;
            bestIndex = neighborhood.IndexOf(instance);
        }
    }

    return bestIndex < 0 ? null : neighborhood[bestIndex];
}
```

Rysunek A.3: Funkcja wybierająca najlepsze rozwiązanie z sąsiedztwa.

```
1 reference | 0 changes | 0 authors, 0 changes
public void LoadEntityFromDB(int InstanceId)
{
    instanceToResolve = context.Instances.Where(x => x.Id == InstanceId)
                               .Include(x => x.Metadata)
                               .Include(x => x.Resources
                                       .Select(y => y.Groups))
                               .Include(x => x.ResourceTypes)
                               .Include(x => x.TimeGroups
                                       .Select(y => y.Times))
                               .Include(x => x.Times)
                               .Include(x => x.Events
                                       .Select(y => y.EventResources))
                               .Include(x => x.EventGroups
                                       .Select(y => y.Events))
                               .Include(x => x.ResourceGroups
                                       .Select(y => y.Events))
                               .ToList().FirstOrDefault();

}
```

Rysunek A.4: Funkcja wczytująca instancje z bazy danych.

```
public class TabuItem
{
    public int IndexX;
    public int IndexY;
    public int TabuDuration;

    2 references | Michał Sz, 38 days ago | 1 author, 1 change
    public TabuItem(int x, int y, int duration)
    {
        IndexX = x;
        IndexY = y;
        TabuDuration = duration;
    }
}
```

Rysunek A.5: Klasa reprezentująca pojedynczy element na liście tabu.

```
8 references | reathion69, 26 days ago | 2 authors, 2 changes
public static int EvaluateInstance(Instance instance)
{
    int rating = 0;

    rating += CheckResourceConflict(instance);
    rating += CheckWidnowsExistForClass(instance);

    return rating;
}
```

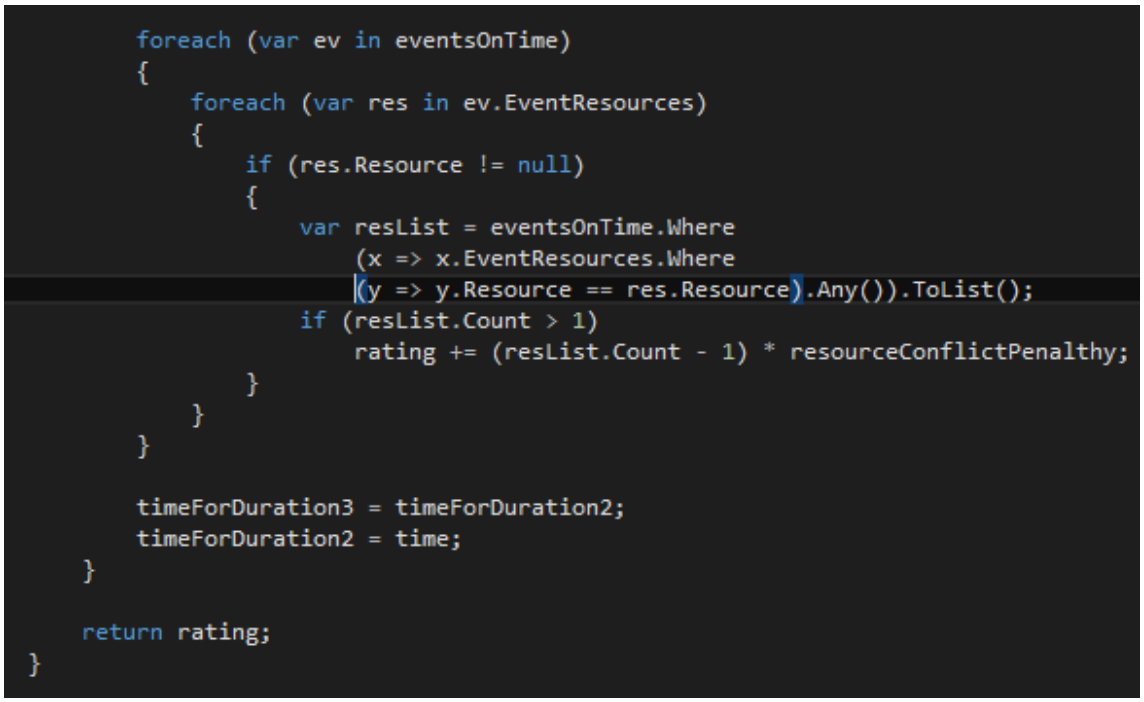
Rysunek A.6: Funkcja oceniająca rozwiązanie.

```
static int CheckWidnowsExistForClass(Instance instance)
{
    List<Resource> Classes
        = instance.Resources.Where(x => x.Type.Name.Contains("Class")).ToList();
    List<TimeGroup> days
        = instance.TimeGroups.Where(x => x.Type == TimeGroupsType.Day).ToList();
    int rating = 0;

    foreach (var classObj in Classes)
    {
        foreach (var day in days)
        {
            List<Time> times
                = instance.Times.Where(x => x.TimeGroups.Any(y => y == day)).ToList();
            bool start = false;  bool end = false;  int size = 0;
            int durationIterator = 0;
            foreach (var time in times)
            {
                if (durationIterator <= 0)
                {
                    Event ev
                        = instance.Events.Where(x => x.Time == time && x.EventResources
                                    .Any(y => y.Resource == classObj)).FirstOrDefault();
                    if (ev != null)
                    {
                        durationIterator = ev.Duration;
                        start = true;
                        if (end)
                        {
                            rating += size * isWindowPenalthy;
                            size = 0;  end = false;
                        }
                    }
                    else
                    {
                        if (start)
                        {
                            end = true;  size++;
                        }
                    }
                }
                durationIterator--;
            }
        }
    }

    return rating;
}
```

Rysunek A.7: Funkcja sprawdzająca występowanie okienek.

```
static int CheckResourceConflict(Instance instance)
{
    int rating = 0;
    Time timeForDuration2 = null;
    Time timeForDuration3 = null;
    foreach (var time in instance.Times)
    {
        List<Event> eventsOnTime = instance.Events.Where(x => x.Time == time).ToList();

        // Implementacja eventów o długości dłuższej niż 1
        // ***********************************************
        List<Event> eventsToAddOnDuration2 = null;
        List<Event> eventsToAddOnDuration3 = null;

        if (timeForDuration2 != null)
            eventsToAddOnDuration2 =
                (instance.Events.Where
                (x => x.Time == timeForDuration2 &&
                (x.Duration == 2 || x.Duration == 3)).ToList());
        if (timeForDuration3 != null)
            eventsToAddOnDuration3 =
                (instance.Events.Where
                (x => x.Time == timeForDuration3 && x.Duration == 3).ToList());


        if (timeForDuration2 != null && eventsToAddOnDuration2.Any() &&
            time.TimeGroups.Where(x => x.Type == TimeGroupsType.Day).
            FirstOrDefault() != timeForDuration2.TimeGroups.Where
            (x => x.Type == TimeGroupsType.Day).FirstOrDefault())

            rating += eventsToAddOnDuration2.Count * eventIsSplitPenalthy;

        if (timeForDuration3 != null && eventsToAddOnDuration3.Any() &&
            time.TimeGroups.Where(x => x.Type == TimeGroupsType.Day).
            FirstOrDefault() != timeForDuration3.TimeGroups.Where
            (x => x.Type == TimeGroupsType.Day).FirstOrDefault())

            rating += eventsToAddOnDuration3.Count * eventIsSplitPenalthy;
        if (eventsToAddOnDuration2 != null)
            eventsOnTime.AddRange(eventsToAddOnDuration2);
        if (eventsToAddOnDuration3 != null)
            eventsOnTime.AddRange(eventsToAddOnDuration3);
        // ***********************************************
```

Rysunek A.8: Funkcja sprawdzająca konflikt zasobów oraz ich niepodzielność. Część 1.



```
        foreach (var ev in eventsOnTime)
        {
            foreach (var res in ev.EventResources)
            {
                if (res.Resource != null)
                {
                    var resList = eventsOnTime.Where
                        (x => x.EventResources.Where
                        (y => y.Resource == res.Resource).Any()).ToList();
                    if (resList.Count > 1)
                        rating += (resList.Count - 1) * resourceConflictPenalthy;
                }
            }
        }

        timeForDuration3 = timeForDuration2;
        timeForDuration2 = time;
    }

    return rating;
}
```

Rysunek A.9: Funkcja sprawdzająca konflikt zasobów oraz ich niepodzielność. Część 2.