Wojciech Misztal 127302 wojciech.misztal@student.put.poznan.pl

Tomasz Pecyna 127284 tomasz.pecyna@student.put.poznan.pl

Job Shop Scheduling

Genetic Local Search

1. Wstęp

Tematem niniejszego sprawozdania jest implementacja algorytmu Genetic Local Search dla problemu Job Shop Scheduling.

Jak zostało opisane w sprawozdaniu omawiającym algorytm Branch & Bound, Job Shop Sheduling Problem (dalej zwany JSSP) można przedstawić jako graf skierowany w którym każde zadanie jest osobnym wierzchołkiem z przypisanym do niego czasem i maszyną przetwarzania. Pierwszą zmianą w reprezentacji, która występuje przy implementacji Genetic Local Search (dalej GLS) jest to, że nie było konieczne tworzenie wierzchołka początkowego i końcowego. Drugą zmianą było powołanie do życia osobnej struktury specjalnie na potrzeby algorytmu genetycznego (dalej GA – Genetic Algorithm) repetition job list. Zostanie one omówiona w dalszej części.

**Genetic Local Search jako polączenie Genetic Algorithm i Local Search**

Aby mówić o GLS trzeba najpierw przedstawić algorytmy z których się on składa – GA i Local Search (dalej LS).

**Genetic Algorithm**

Algorytm genetyczny jest to metaheurystyka która została zainspirowana występującym w przyrodzie zjawiskiem ewolucji oraz naturalnej selekcji. Głównymi punktami algorytmu genetycznego są:

* **Tworzenie populacji** – jest to pierwsza część algorytmu, która polega na wygenerowaniu, najczęściej w sposób losowy, P rozwiązań danego problemu. Zbiór ten jest nazywany populacją, a poszczególne rozwiązania nazywane są osobnikami. Każdy z osobników posiada swój własny zbiór danych dotyczących rozwiązania, który zwany jest chromosomem. To właśnie chromosom determinuje jakość poszczególnego rozwiązania danego problemu (minimalizacji lub maksymalizacji). Na populacji oraz chromosomach danych rozwiązań wykonywane są w pętli kolejne trzy operacje, jakimi są selekcja, krzyżowanie i mutacja.
* **Selekcja –** po wygenerowaniu zbioru rozwiązań populacja poddawana jest selekcji, czyli wyboru poszczególnych osobników (rodziców) którzy przejdą do następnego punktu, jakim jest krzyżowanie. Wybór rodziców do krzyżowania może się odbywać kilkoma metodami, na przykład metodą turniejową, losową z różnym prawdopodobieństwem, albo porównaniem jakości osobnika do pewnej zadanej wartości. To tutaj właśnie następuje selekcja naturalna, w której słabsze osobniki zostają wyeliminowane i zastąpione zostają podczas krzyżowania.
* **Krzyżowanie –** po wybraniu dwóch osobników do krzyżowania część chromosomu jednego i drugiego rodzica zostaje przekazana nowemu reprezentantowi populacji – dziecku. Krzyżowanie ma na celu uzyskanie potomka o cechach preferowanych, które występują zarówno w pierwszym jak i w drugim rodzicu.
* **Mutacja** – Po przejściu poprzednich punktów pewna część populacji poddawana jest mutacji, czyli losowym zmianom, mającym na celu wytworzenie w poszczególnych rozwiązaniach cech osobistych, oraz tym samym zapobiegnięcie zjawisku klonowania poszczególnych rozwiązań, czyli skupiania się całej populacji wokół pojedynczego rozwiązania.

Po wystąpieniu mutacji algorytm wraca do punktu selekcji, pracując dalej na starej populacji lub dokładając nowe rozwiązania.

**Local Search**

LS to heurystyka polegająca na przeszukiwaniu sąsiedztwa wokół danego rozwiązania. Algorytm ten wprowadza niewielkie zmiany w rozwiązaniu, po czym porównuje jakość nowego rozwiązania do starego. Jeśli nowy osobnik jest lepszy od starego, zostaje przez niego zastąpiony i proces przeszukiwania sąsiedztwa startuje na nowo.

**Genetic Local Search**

Uwaga: wszystkie testy w tej sekcji oraz w części *Strojenie* zostały wykonane na instancji tai20. Czas na osi X podano w sekundach

Zarówno GA jak i LS posiadają wady, które łatwo jest wyeliminować stosując jednocześnie oba algorytmy. Algorytm genetyczny przy małej populacji działa szybko, jednak bez odpowiednio zaimplementowanej mutacji równie szybko zbiega się do lokalnego optimum, co powoduje zastój w posuwaniu się rozwiązania do optimum globalnego. Dobrze przedstawia to następujący wykres przeprowadzony dla populacji P = 20 ze współczynnikiem mutacji m = 0,05

Jak widać z powyższego wykresu algorytm nie zszedł ze swoim rozwiązaniem poniżej wartości 1750, a przez połowę czasu pracy uczynił bardzo mały postęp. Rozwiązaniem tego problemu jest tworzenie większej populacji, która z oczywistych względów później osiągnie optimum lokalne. Wadą tego rozwiązania jest długi czas przetwarzania i postępu, przez co w zadanych 3 minutach algorytm nie dojdzie nawet do swojego optimum lokalnego.

Jak widać, dla większego P algorytm nawet w końcowych sekundach swojej pracy wciąż poprawiał wynik, jednak początkowe wartości minimalne były bardzo wysokie, co nie jest pożądane przy testach z ograniczeniem czasowym.

Algorytm LS osiąga lokalne optimum jeszcze szybciej niż GA. Aby wyjść z lokalnego optimum trzeba porzucić stare rozwiązanie i zacząć pracować na nowym. Poniżej przedstawiono wykres osiągania lokalnego optimum przez LS.

Oba te algorytmy posiadają tendencję do zbiegania się do lokalnego optimum, jednak robią to na inny sposób.

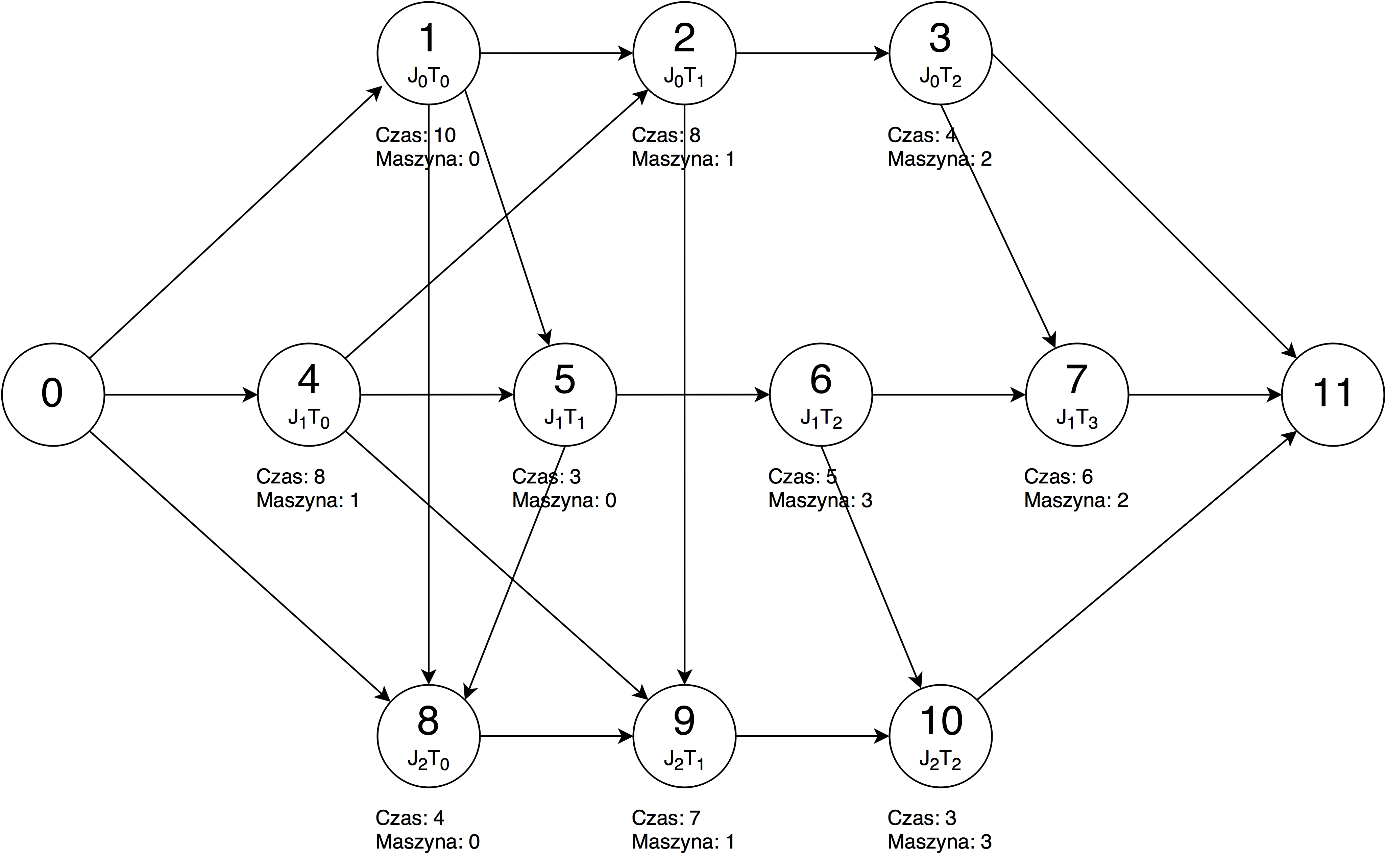
Zastosowanie jednocześnie GA i LS sprawia, że te problemy występujące w nich niwelują się. Jak było wcześniej powiedziane, algorytm genetyczny aby zapobiec zastojowi stosuje mutacje. LS z definicji działania stosuje ciągłe mutacje i zmiany na swoim rozwiązaniu. LS aby uniknąć lokalnego optimum wybiera inne sąsiedztwo – GA podczas krzyżowania tworzy „inne, lepsze sąsiedztwo” tworząc nowego reprezentanta populacji. Metoda ta pozwala również na używanie małej populacji, która szybciej dochodzi do dobrej jakości rozwiązań

Z wyprowadzonych wcześniej wniosków wynika, że aby uzyskać pełny potencjał z obu algorytmów dobrym rozwiązaniem jest wstawienie algorytmu Local Search zamiast (lub równolegle do) mutacji w algorytmie genetycznym. Z tego połączenia powstaje algorytm GLS Podejście to zostało zaimplementowane w języku C++, a jego wyniki zostaną zaprezentowane w niniejszym sprawozdaniu.

1. Reprezentacja, opis użytych rozwiązań.

Warunkiem używania algorytmu genetycznego jest zapisanie danego rozwiązania w sposób umożliwiający przeprowadzenie krzyżowania. Najczęściej wykonuje się do w formie listy/tablicy liczb w kodzie dziesiętnym lub binarnym. Do JSSP najlepszym wyborem okazała się tzw. repetitive job list. Reprezentacja ta polega na utworzeniu tablicy, której elementami są numery procesów. Numery te powtarzają się dla każdego procesu tyle razy, ile razy ma on wykonywać zadanie na jakiejś maszynie. Kolejność wystąpienia licz reprezentujących poszczególne procesy na liście określa które zadanie ma być wykonane pierwsze. Przekładając to na przykład ze sprawozdania dotyczącego algorytmu B&B:

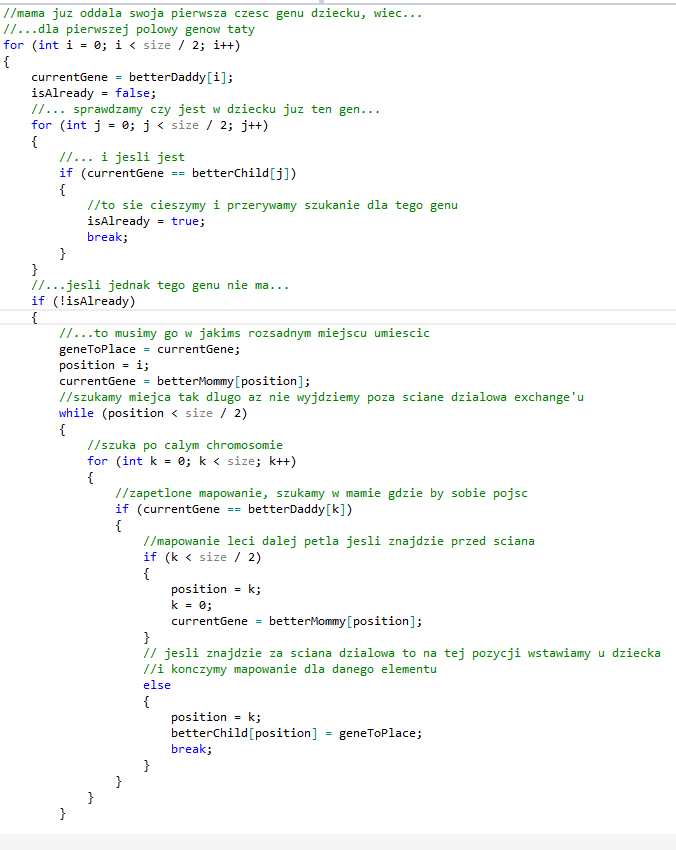
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | T0 | T1 | T2 | T3 |
| J0 | t = 10, mc = 0 | t = 8, mc = 1 | t = 4, mc = 2 | - |
| J1 | t = 8, mc = 1 | t = 3, mc = 0 | t = 5, mc = 3 | t = 6, mc = 2 |
| J2 | t = 4, mc = 0 | t = 7, mc = 1 | t = 3, mc = 3 | - |



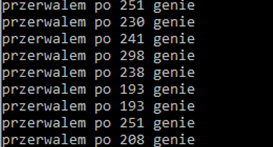
Repetitive job list mógłby wyglądać w sposób następujący: [1, 0, 1, 2, 0, 2, 1, 0, 1, 2], co oznacza, że najpierw wykonuje się pierwsze (jeśli chodzi o kolejność przetwarzania w przypadku konfliktu) zadanie dla procesu nr 1, następnie wykona się pierwsze zadanie dla procesu nr 0, potem drugie zadanie dla procesu nr 1, pierwsze zadanie dla procesu nr 2 itd. Niewątpliwą zaletą tej reprezentacji jest fakt, że uniemożliwia ona powstawanie cykli w grafie i zawsze generuje ona poprawne rozwiązania.

Reprezentacja ta umożliwia również przeprowadzenie w prosty sposób mutacji lub zamian, które polegają na zamianie liczb znajdujących się na określonych indeksach. Dzięki temu łatwa okazała się w implementacji funkcja zmieniająca poszczególne geny, która polegała na losowym wybraniu dwóch indeksów z pojedynczego chromosomu i jeśli liczby znajdujące się na tych indeksach były różne – zamianie ich miejscami. Funkcja ta była używana głównie przez algorytm LS, który działał na zasadzie: Próbuj zamiany tak długa, jak lista zamian nie przynoszących korzyści będzie mniejsza od pewnej liczby maksymalnych iteracji. Jeśli nastąpiła zmiana podczas której

Większy problem jest z krzyżowaniem, ponieważ podczas niego zachodzi możliwość uzyskania niepoprawnego rozwiązania, dlatego do tej części GA został zastosowany algorytm PMX, który w dużym skrócie polega na skopiowaniu części genów od jednego rodzica, a następnie sprawdzaniu czy odpowiadające temu fragmentowi chromosomu miejsca w drugim rodzicu występują w chromosomie pochodnym – jeśli tak, to nic nie robimy, jeśli natomiast nie występują – mapujemy na siebie wartości tak długo, aż nie wyjdziemy poza obszar zamieniany. Na miejsca w potomku na które nie zostało nic wpisane, przepisujemy geny z rodzica drugiego. Drugim problemem jest to, że w reprezentacji liczby się powtarzają, a algorytm PMX działa tylko wtedy, kiedy operuje na różnych wartościach, dlatego aby rozwiązanie było poprawne dla każdej liczby o danej wartości zostaje przypisany indeks oznaczający numer jej wystąpienia w chromosomie. Przykładowo biorąc poprzedni chromosom będzie to: [1(0), 0(0), 1(1), 2(0), 0(1), 2(1), 1(2), 0(2), 1(3), 2(2)], gdzie liczby w nawiasach odpowiadają odpowiednim indeksom dla danego joba. Implementację kluczowego dla każdego algorytmu genetycznego krzyżowania, przedstawia wyciąg z kodu, gdzie zawsze następuje kopiowanie pierwszej połowy pierwszego rodzica i drugiej połowy drugiego rodzica.



Metoda programowania pozwoliła również w łatwy sposób utworzyć funkcję obliczania makespanu. Dzięki wyrażeniu pojedynczego wierzchołka w grafie jako obiektu trzymającego o sobie informacje, nie trzeba było tworzyć listy poprzedników ani następników, jak to zwykle jest w reprezentacjach typu macierz grafu, a jedynie trzymać informację o poprzedniku z największym czasem przetwarzania, która była aktualizowana wraz iteracją po liście repetitive jobs. Wykorzystując doświadczenie zdobyte przy implementowaniu algorytmu B&B udało się również znacznie skrócić czas działania algorytmu Local Search, ponieważ jeśli zamiana genów okazywała się pogarszać wynik, można było wychwycić ją znacznie wcześniej obliczając minimalny czas potrzebny na wykonanie wszystkich zadań i jeśli był on większy, przerwać aktualną pętlę LS. Zastosowanie tej metody pozwoliło skrócić w niektórych przypadkach liczbę przetworzonych wierzchołków nawet o 1/3.



Wynik konsoli dla instancji przetwarzanej algorytmem LS z liczbą wierzchołków równą 300.

Mając funkcję obliczającą makespan oraz funkcję można przeprowadzić wybór osobników do krzyżowania. Selekcja odbywała się w sposób turniejowy: (Populacja / 4) razy zostaje wybranych 4 osobników którzy będą porównywani w ten sposób, aby wybrać dwóch najlepszych, których geny będą kopiowane. Dwóch osobników którzy przegrali turniej zostaje całkowicie wyeliminowanych z populacji.

1. Strojenie

Przed przejściem do strojenia najważniejsze pytanie które musi zostać zadane brzmi: Co dany algorytm będzie optymalizował? W tym projekcie optymalizacyjnym celem GLS było znalezienie jak najlepszego rozwiązania w pierwszych trzech minutach od rozpoczęcia działania, co zostało realizowane poprzez uniemożliwienie zastoju algorytmu w optimum lokalnym.

Algorytm GLS charakteryzuje się dużą liczbą zmiennych sterujących, takich jak wcześniej wymieniona wielkość populacji, współczynnik mutacji, liczba przejścia pętli GA czy liczba iteracji dla LS. Do tego dochodzą bardziej koncepcyjne kwestie, takie jak: czy przeprowadzić wstępny LS na świeżo zainicjalizowanej populacji? Co ile populacji przeprowadzać LS? Czy stosować LS równolegle z mutacją? Jakie warunki posiada mutacja, czy może ona wystąpić na najlepszym osobniku i tym samym, czy może go pogorszyć? Możliwości jest tak wiele, że bardzo ciężko jest przeanalizować je wszystkie i znaleźć optymalne wartości dla algorytmu. Z tego powodu ograniczymy sterowanie do trzech wartości: liczby iteracji LS, liczby populacji oraz współczynnika mutacji LS.

Jako, że optymalizujemy jakość rozwiązania a nie czas, liczba przejść pętli w algorytmie genetycznym będzie nieskończona, a czas trwania algorytmu będzie wynosił 180 sekund. Żeby jak najlepiej uchronić się od zatrzymania się w lokalnym optimum LS był przeprowadzany na losowych osobnikach co populację z danym współczynnikiem mutacji. Aby wprowadzić powiew świeżości do populacji oryginalna mutacja polegająca była przeprowadzana na drugim rodzicu (drugim w sensie uszeregowania jakościowego), przez co nie występowała sytuacja w której najlepszy osobnik zostanie „zepsuty” przez złą mutację.

Kwestię przeprowadzenia wstępnego LS wyjaśnia wykres:

Z którego wynika interesujący wniosek – jeśli jakiś osobnik „urodził się” silniejszy, to mimo że się go zoptymalizuje przez użycie algorytmu LS, wciąż będzie słabszy od pozostałych, co może potem przynieść skutki w gorszym makespanie. Zależność ta pojawiała się, z paroma wyjątkami, w mniejszym lub większym stopniu, dosyć często. Z tego powodu algorytm GLS zaprezentowany w tym sprawozdaniu nie przewiduje wstępnej optymalizacji. Epigenetyka nie pomaga ;<

1. **Wyznaczanie liczby iteracji LS dla mutacji**

Aby wyznaczyć optymalną liczbę iteracji dla LS weźmy stałą populację P = 20 oraz stały współczynnik mutacji m = 0,05. Wykres przedstawia zależność makespanu w zależności od liczby iteracji algorytmu LS w zakresie od 25 do 400 iteracji z krokiem co 25

Jak wynika z powyższego wykresu największa częstotliwość najlepszych wyników występuje w zakresie od 75 do 225 iteracji. Najlepiej jest przyjąć wartość mieszczącą się w tym przedziale. Jako, że w naszym problemie czas jest ograniczony do 3 minut do dalszych badań przyjęto liczbę iteracji równą 150.

1. **Wyznaczenie populacji**

Wyznaczenie liczby osobników populacji odbyło się przy ustalonej liczbie iteracji LS = 150 oraz współczynniku m = 0,05. Populacja od 10 do 60, krok 5.

Wartości najlepsze występowały w zakresie 45-65, więc od tej pory wszystkie testy będziemy rozważać dla populacji o wielkości P = 50.

1. **Wyznaczenie współczynnika mutacji**

Ostatni ze współczynników zostanie wyznaczony przy P = 50 oraz LS = 150 iteracji.

Jak widać najlepszy wynik osiągnięto dla m = 0,06, której to wartości będziemy używali w badaniach.

1. Testy

Przetestowane zostaną instancje od tai20 do tai25, tai20 zostanie dodatkowo przetestowana dla zwiększającej się liczby procesów. Wszystkie testy odbywać się będą na wyprowadzonych wcześniej parametrach P = 50, m = 0,05, LS = 150.

Z testów tych wynika, że algorytm szybko znajdował dosyć dobre rozwiązanie, a następnie powoli zmniejszał makespan. Dla tai20 po 3 minutach zakończył na make spanie 1608, dla tai21 na makespanie 2042, dla tai22 makespan końcowy wynosi 2045, dla tai23 1924, dla tai24 2001, a dla tai25 1953. Należy zauważyć, że zmiany nie następują płynnie tylko skokowo, a im bardziej średnia populacji zbliża się do minimum tym mniejsze prawdopodobieństwo poprawienia wyniku, a większe prawdopodobieństwo zbiegnięcia się populacji do lokalnego optimum.

Dla pierwszych trzech procesów w instancji tai25 algorytm znalazł rozwiązanie poniżej sekundy dla pierwszych 5 procesów zajęło mu już to 2,84 sekundy, a dla 10 zadań osiągnął optimum w czasie 23,18 sekundy. Dla 15 i 20 procesów algorytm w ciągu pierwszych 180 sekund nie zdążył znaleźć optimum.

1. Porównanie metod B&B oraz GLS

Porównajmy oba przetestowane algorytmy dla instancji tai20

Algorytm B&B znalazł pierwsze rozwiązanie równe 1677 w czasie t = 18,87 s, a następnie przez pozostały czas nie poprawiał wyniku. W algorytmie GLS pierwszym najlepszym rozwiązaniem zaraz po wygenerowaniu populacji w czasie t = 0,35 s był makespan równy 2386, jednak już po pierwszym przejściu pętli głównej GLS, w czasie t = 5,04 s rozwiązanie wynosiło 1867. Jeśli chodzi o szybkość znalezienia pierwszych rozwiązań GLS wypada tutaj lepiej, jednak w momencie w którym B&B znalazł swoje pierwsze rozwiązanie GLS był dopiero na rozwiązaniu równym 1849, czyli algorytm B&B znalazł o 172 jednostki czasowe lepsze rozwiązanie. Jest to rozwiązanie, w bardzo dużym przybliżeniu o 10% lepsze od znalezionego w tym czasie przez GLS. Ciekawym punktem rozważań jest również intersekcja obu krzywych, która nastąpiła w czasie t = 135,8s W czasie gdy B&B sprawdzał kolejne permutacje porównując je do Upper Bounda nie mogąc znaleźć lepszego rozwiązania GLS ciągle krzyżował populacje szukając lepszych wyników tym samym wyprzedzając w rozwiązaniu algorytm B&B, a następnie dalej poprawiając wynik. Mimo tej zalety jednak, w dłuższym czasie poprawy algorytmu GLS stawały się coraz mniejsze, skupiając się wokół lokalnego optimum. Algorytm GLS jako metaheurystyka nie gwarantuje znalezienia optymalnego rozwiązania w porównaniu do algorytmu B&B, więc po czekaniu dostatecznie długiego czasu B&B znajdzie lepsze rozwiązanie od GLS, dążąc, a nawet osiągając optymalne.

1. Wnioski

Algorytm GLS jest algorytmem lepszym od samego GA oraz LA, ponieważ one szybko zbiegają się do lokalnego optimum, jednak wciąż jest to metaheurystyka, więc nie daje nam ona gwarancji optymalnego rozwiązania, ponieważ w dużym stopniu polega ona na losowości. Kluczowe w algorytmie jest dobranie odpowiednich wartości sterujących, które jednak mogą się różnić w zależności od charakterystyki instancji. Jest możliwe jednak dobranie tak wartości sterujących aby algorytm w większości przypadków osiągał bardzo dobre wyniki, a nawet w bardzo wielu przypadkach był lepszy od algorytmów pełnego przeszukiwania takich jak B&B