POP 2023Z - Dokumentacja Końcowa

Temat: Traveling Santa 2021 - The Merry Movie Montage

Zespół:

- Michał Laskowski
- Konrad Meysztowicz-Wiśniewski

Spis treści:

Treść zadania

Tym razem, Mikołaj potrzebuje pomocy przy optymalizacji harmonogramu siedmiu świątecznych filmów prezentowanych przez SantaTV. Zadanie polega na znalezieniu trzech najkrótszych harmonogramów które spełniają 2 założenia - wszystkie możliwe permutacje filmów muszą znajdować się w przynajmniej jednym harmonogramie. Wszystkie permutacje rozpoczynające się od filmów 1 i 2 muszą znajdować się w każdym harmonogramie. Dla ułatwienia zadania, możliwe jest zastosowanie dwóch blanków - sytuacji w której wszystkie filmy grane są jednocześnie. (https://www.kaggle.com/competitions/santa-2021/overview)

Definicja problemu

Zadanie sprowadza się do znalezienia kombinacji trzech ciągów składających się z zadanych symboli i spełniających zadane warunki, takiej, żeby najdłuższy z nich był jak najkrótszy.

W oryginalnym problemie wymagane symbole są zadane w formie emotikon - 🎅, 🤶, 🦌, 🤷, 🎄, 🎁, 🎀 - dla uproszczenia w dokumentacji będziemy się odwoływać do nich jako 1, 2, 3 itd. (przykładowo sekwencja ‘231’ odpowiada sekwencji 🤶🦌🎅).

Dodatkowo dostępny jest opcjonalny symbol 🌟 (dalej blank lub ‘*’), który może być interpretowany jako wiele symboli jednocześnie. Przykładowo sekwencja ‘1*3’ odpowiada wszystkim sekwencjom ‘113’, ‘123’, ‘133’ itd.

Warunki poprawności rozwiązania:

- Każda permutacja zadanych symboli musi zostać zawarta w przynajmniej jednym ciągu
- Każdy ciąg musi zawierać wszystkie permutacje znaków zaczynające się od sekwencji ‘12’
- Każdy ciąg może zawierać maksymalnie dwa blanki
- Sekwencje siedmiu znaków zawierające więcej niż jeden blank nie liczą się jako permutacje

Analiza danych wejściowych

- Wszystkich permutacji zbioru n-elementowego jest n!
    - w naszym wypadku n=7, 7! = 5040.
- Najkrótsza znaleziona do tej pory super-permutacja dla 7 elementów ma długość 5906 znaków [https://www.gregegan.net/SCIENCE/Superpermutations/Superpermutations.html].
- Najlepsze opublikowane rozwiązanie konkursu osiąga wynik 2428 znaków w najdłuższym ciągu

Zadanie jako problem komiwojażera

Znalezienie super-permutacji zbioru (ciągu zawierającego wszystkie permutacje elementów zbioru) można rozważać jako asymetryczny wariant problemu komiwojażera.

Rozważmy reprezentację problemu w formie ważonego pełnego grafu skierowanego. Węzłami w grafie będą wszystkie permutacje elementów zbioru, a wagi przejść będą zdefiniowane jako liczba znaków, które trzeba dodać do pierwszej permutacji, aby powstały ciąg zawierał też drugą permutację.

Przykład dla dwóch permutacji p='**123**' i q='312' 3-elementowego zbioru:

- d(p,q) = 2
  - **123** => **123**1 => **12**3**12**
- d(q,p) = 1
  - 312 => 3**12**3

Jak widać funkcja odległości jest asymetryczna.

Przy takiej reprezentacji znalezienie super-permutacji jest równoważne ze znalezieniem ścieżki odwiedzającej każdy węzeł przynajmniej raz. Ponieważ szukamy ścieżki o możliwie minimalnym koszcie preferowane jest odwiedzenie każdego węzła dokładnie raz, czyli znalezienie ścieżki Hamiltona. Zadanie znalezienia ścieżki Hamiltona o minimalnym koszcie w grafie skierowanym jest wariantem problemu komiwojażera.

Definicja funkcji celu

Funkcja celu w problemie komiwojażera określana jest ogólnie jako suma odległości między odwiedzonymi węzłami. W naszym przypadku tworzymy 3 ciągi, dlatego funkcją celu, którą będziemy minimalizować, jest maksimum z sum odległości między permutacjami dla każdego z ciągów. Możemy opisać ją wzorem:

$$max\left\{\sum_{j=1}^{n_i-1} d_{ij} : i = 1..s\right\}$$

gdzie:

s - liczba ciągów

n_i - liczba permutacji w i-tym ciągu

d_{ij} - dystans między permutacją j, a j+1 w i-tym ciągu

Omówienie algorytmów

W kontraście do zdecydowanej większości rozwiązań konkursu kaggle'owego, które bazują na zewnętrznych bibliotekach i solverach jak LKH, chcieliśmy w ramach projektu stworzyć autorskie rozwiązanie od podstaw, korzystając z wiedzy pozyskanej na wykładach.

Jako rozwiązanie projektu zaimplementowaliśmy w języku Python różne wersje dwóch algorytmów przeszukiwania - najbliższego sąsiada oraz symulowanego wyżarzania. W pierwszej wersji algorytmy pracują bez blanków, aby sprawdzić układanie bazowych permutacji, a w drugiej zostało dodane ich użycie. Na końcu dokonujemy porównań wszystkich wersji algorytmów, żeby sprawdzić czy udało nam się je usprawnić wprowadzając użycie blanków i ocenić, który daje najlepsze rezultaty.

Algorytm najbliższego sąsiada

Algorytm najbliższego sąsiada (Nearest Neighbour Algorithm) jest algorytmem zachłannym, który może służyć do rozwiązywania wyżej opisanego problemu komiwojażera. Najpewniej nie znajdzie on rozwiązania optymalnego, ale otrzymane wyniki powinny być akceptowalne. Działa on na zasadzie doczepiania kolejnych permutacji symboli do ciągu bazując na dystansie od permutacji wybranej w poprzedniej iteracji. Wybierana jest ta permutacja, która ma najkrótszy dystans, czyli wymaga dopisania najmniejszej liczby symboli.

Kolejne kroki działania algorytmu

1. Utworzenie trzech indywidualnych list, po jednej dla każdego harmonogramu, oraz jednej wspólnej listy, zawierających permutacje symboli. Indywidualne listy zawierają wszystkie wymagane permutacje znaków, to znaczy zaczynające się od '12' , a wspólna lista zawiera wszystkie pozostałe.
2. W każdej iteracji algorytmu po kolei dla każdego ciągu wykonywane jest kilka kroków:
    2.1. Znalezienie listy najbliższych sąsiadów
    2.2. Wybór najbliższego lub losowego z najbliższych, jeżeli mają taki sam dystans
    2.3. Doczepienie wybranej permutacji do aktualnego ciągu
    2.4. Usunięcie wybranej permutacji z odpowiedniej listy
3. Po ukończeniu kroku 2 dla wszystkich ciągów sprawdzany jest warunek zakończenia pętli - sprawdzenie czy listy utworzone na początku są już puste. Będzie to oznaczać, że wszystkie permutacje zostały już zawarte w ciągach. Jeśli pozostały jakieś permutacje w listach, to ponownie wykonywany jest krok 2.
4. Na koniec wyświetlane jest rozwiązanie oraz jego wynik

Symulowane wyżarzanie

Algorytm symulowanego wyżarzania polega na wpierw znalezieniu bazowego dozwolonego rozwiązania problemu, a potem modyfikowaniu go w poszukiwaniu lepszego. Jeśli rozwiązania po modyfikacji stało się lepsze, to staje się nowym rozwiązaniem bazowym i proces jest kontynuowany. Jeśli rozwiązanie stało się gorsze, to nadal jest szansa, że stanie się rozwiązaniem bazowym - szansa ta wynika z funkcji prawdopodobieństwa opartej o temperaturę. Temperatura to ustalona na początku wykonywania algorytmu wartość, która będzie maleć wraz z postępem algorytmu - im mniejsza temperatura, tym mniejsze prawdopodobieństwo, że gorsze rozwiązanie zostanie wybrane jako nowe bazowe.

Wyżarzanie - wersja pierwsza

Kolejne kroki działania programu:

1. Podzielenie wszystkich permutacji na zbiór permutacji obowiązkowych dla każdego ciągu (tych zaczynających się sekwencją ‘12’) i zbiór permutacji “jednorazowych”, które muszą pojawić się tylko przynajmniej raz w dowolnym ciągu.
2. Równomierne rozdzielenie permutacji “jednorazowych” pomiędzy harmonogramy i dodanie do każdego z nich pełnego zbioru permutacji obowiązkowych
3. Wygenerowanie bazowego ciągu dla każdego z harmonogramów jako losową sekwencję przypisanych mu permutacji
4. W każdej iteracji algorytmu wykonanie następujących czynności dla każdego harmonogramu:
    4.1. Wygenerowanie nowego ciągu poprzez zamianę miejscami dwóch permutacji w starym ciągu
    4.2. Porównaniu sum wag przejść pomiędzy permutacjami w ciągach
    4.3. Wybranie nowego ciągu jeśli jest lepszy od starego
        4.3.1. Jeśli nowy ciąg jest gorszy, to nadal ma szansę zostać wybrany zgodnie z prawdopodobieństwem zależnym od temperatury
    4.4. Zmniejszenie temperatury
    4.5. Sprawdzenie warunku zatrzymania pętli - przekroczona liczba dozwolonych iteracji lub temperatura poniżej ustalonej wartości
5. Na koniec wyświetlane jest rozwiązanie oraz jego wynik

Wyżarzanie - wersja druga

W trakcie implementacji symulowanego wyżarzania według powyższych kroków byliśmy bardzo niezadowoleni z wyników zwracanych przez algorytm. Program nawet po długim czasie pracy produkował rozwiązania znacząco gorsze niż wykorzystując algorytm najbliższego sąsiada, który dawał wynik znacznie szybciej.

Szybko zorientowaliśmy się, że w obecnej formie punkt 4, a konkretniej 4.1, zbytnio ogranicza nam przestrzeń rozwiązań. Rozważając zamianę miejsc dla permutacji tylko w jednym ciągu nie dajemy algorytmowi możliwości poprawy początkowego podziału na harmonogramy, który poza tym, że jest możliwie równomierny, jest przecież całkowicie arbitralny.

W celu rozszerzenia przestrzeni potencjalnych rozwiązań algorytm w tej wersji operuje na wszystkich ciągach jednocześnie. Generowanie nowego zestawu harmonogramów polega na zamianie miejscami dwóch dowolnych permutacji, niezależnie od tego, do którego ciągu należą.

W tej wersji algorytmu ważne jest zachowanie poprawności rozwiązania. Potencjalnie wymieniając dwie zupełnie dowolne permutacje pomiędzy harmonogrami moglibyśmy doprowadzić do sytuacji, gdzie jeden z nich przestaje zawierać permutację wymaganą dla każdego harmonogramu. Aby temu zapobiec, gdy do zamiany zostaną wylosowane permutacje zawarte w dwóch różnych ciągach, program upewnia się, że żadna z nich nie jest permutacją obowiązkową. W przeciwnym przypadku losowanie odbywa się na nowo.

Wprowadzenie blanków

Potencjalnie najbardziej efektywnym wykorzystaniem blanka może wydawać się wstawienie go do sekwencji, w której zastąpi jak najwięcej symboli. Jednym ze sposobów na to dla n-elementowego zbioru jest wygenerowanie ciągu o długości 2n-1, gdzie środkowym elementem będzie blank. Ciąg o długości 2n-1 ma n podciągów, co pozwala potencjalnie interpretować blank jako każdy z symboli dokładnie raz.

Przykładowy ciąg dla n=4:

- 123*412
    - * => 4: 1234
    - * => 1: 2314
    - * => 2: 3241
    - * => 3: 3412

Wygenerowanie podobnych ciągów dla wszystkich harmonogramów tak, żeby każdy z nich rzutował na różny zbiór permutacji pozwoliłoby nam na potencjalnie efektywne wykorzystanie blanków.

Rozpoczęcie harmonogramu od przypisanego mu ciągu z blankiem na starcie eliminuje część permutacji i pozwala dalej wykonać algorytm bez większych zmian.Dodanie drugiego blanka można wykonać analogicznie, generując ciąg zaczynający się od ostatnich n-1 znaków pierwszego ciągu.

Odeszliśmy jednak od takiego rozwiązania ze względu na dwa powody. Pierwszym z nich jest sztywność tego podejścia, które w żaden sposób nie korzysta z zaimplementowanych algorytmów. Drugim jest błędne w naszej opinii założenie, że optymalnym wykorzystaniem blanka jest zastąpienie największej liczby różnych symboli. Zamiast tego celem dodania blanka do harmonogramu powinno być skrócenie go na tyle, na ile to możliwe, poprzez skrócenie przejścia pomiędzy dwoma odległymi od siebie permutacjami.

Możliwości i ograniczenia blanków

Rozważając gdzie i jak umieszczać blanki w naszych harmonogramach doszliśmy do wniosku, że próba skrócenia przejścia między dwoma permutacjami ma sens wyłącznie dla par odległych od siebie o maksymalny dystans. Dodanie blanka do bliższych sobie permutacji nie daje żadnego efektu.

Maksymalną redukcją jaką można uzyskać wstawiając pojedynczy blank do harmonogramu jest skrócenie go o n-1 elementów, gdzie n jest liczbą symboli zadanych na początku. Mimo, że nie dla każdej pary n-odległych permutacji da się taki wynik uzyskać, to zawsze możliwe jest skrócenie odległości między nimi o 1, usuwając ostatni element pierwszej permutacji oraz pierwszy element drugiej i zastępując je blankiem.

Rozszerzenie algorytmu najbliższego sąsiada

Do algorytmu najbliższego sąsiada postanowiliśmy dodać blanki w formie post-procesu. Praca samego algorytmu opisanego wyżej krok po kroku jest niezmieniona, ale po uzyskaniu wyniku będziemy starać się go poprawić.

Dla każdego z wygenerowanych harmonogramów będziemy szukać dwóch podstawień, kierując się zasadą “pierwszy najlepszy”:

1. Znajdujemy w harmonogramie pierwszą parę występujących po sobie permutacji, takich że odległość między nimi jest maksymalna
2. Szukamy najlepszego podstawienia dla blanka, to znaczy takiego, które możliwie najbardziej skróci odległość między permutacjami (zawsze skraca o przynajmniej 1)
3. Sprawdzamy czy nowe podstawienie mogłoby skrócić harmonogram bardziej niż chociaż jedno ze znalezionych wcześniej dwóch najlepszych podstawień
4. Jeśli tak, to sprawdzamy czy zastąpienie gorszego z dwóch najlepszych podstawień nowym podstawieniem zachowa warunek poprawności - to znaczy, czy liczba symboli pomiędzy blankami w końcowym harmonogramie będzie większa lub równa liczbie symboli permutacji
5. Jeśli tak, to zastępujemy gorsze z dotychczasowych najlepszych podstawień nowym podstawieniem
6. Powtarzamy 1-5, aż dojdziemy do ostatniej pary sąsiednich permutacji
7. Wykonujemy znalezione podstawienia

Rozszerzenie symulowanego wyżarzania

W przypadku algorytmu symulowanego wyżarzania blanki zostały wprowadzone jako osobne elementy, które algorytm może dowolnie zamieniać miejscami z innymi permutacjami. Umożliwia to ciągłą optymalizację ułożenia blanków podczas działania programu.

Blanki zostały wprowadzone jedynie do drugiej wersji algorytmu ze względu na jej lepsze możliwości optymalizacji. Należy uważać jednak na dalsze zachowanie poprawności rozwiązania, ponieważ w jednym harmonogramie mogą występować tylko dwa blanki oraz nie mogą być one umieszczone zbyt blisko siebie. W momencie, kiedy algorytm wylosuje potencjalne nowe zestawienie harmonogramów sprawdzany jest warunek czy nie został przeniesiony blank między różnymi ciągami oraz czy w obrębie jednego ciągu blanki nie są obok siebie. Jeżeli wykryty zostanie którykolwiek z tych błędów, zestawienie harmonogramów jest losowane ponownie.

Implementacja

Nasze rozwiązanie zostało podzielone na pięć głównych plików z odpowiednimi algorytmami oraz jeden plik zawierający pomocnicze funkcje. W pliku o nazwie NN.py znajduje się implementacja algorytmu najbliższego sąsiada, a w pliku NN_blank.py jego rozszerzenie o dodanie blanków. Podobnie algorytm symulowanego wyżarzania w pierwszej wersji znajduje się w pliku SA.py, w drugiej wersji w pliku SA_v2.py, a rozszerzenie drugiej wersji o blanki w pliku SA_v2_blank.py. Plik santa_utl.py zawiera różne pomocnicze funkcje, z których korzystają pozostałe algorytmy.

Sposób uruchomienia

Aby wykonać dany algorytm należy uruchomić plik zawierający jego implementację jako skrypt języka python i podać mu odpowiednie parametry.

NN.py oraz NN_blank.py

Implementacje algorytmu najbliższego sąsiada wymagają następujących parametrów do uruchomienia:

- liczba symboli w permutacji
- liczba harmonogramów

Przykładowe uruchomienie algorytmów z konsoli w folderze zawierającym pliki:

```
>python NN.py 7 3
>python NN_blank.py 7 3
```

SA.py, SA_v2.py oraz SA_v2_blank.py

Implementacje algorytmu symulowanego wyżarzania wymagają następujących parametrów do uruchomienia:

- liczba symboli w permutacji
- liczba harmonogramów
- początkowa temperatura
- tempo spadku temperatury
- liczba iteracji do wykonania

Przykładowe uruchomienie algorytmów z konsoli w folderze zawierającym pliki:

```
>python SA.py 7 3 100 0.99 30000
>python SA_v2.py 7 3 100 0.99 30000
>python SA_v2_blank.py 7 3 100 0.99 30000
```

Wyniki i obserwacje

Sposób mierzenia jakości rozwiązania

Aby móc porównać różne rozwiązania oraz sprawdzić, które jest lepsze musimy mieć sposób mierzenia jakości rozwiązania. Jakość rozwiązania będziemy opisywać za pomocą dwóch parametrów: ostatecznej wartości funkcji celu z otrzymanych ciągów permutacji oraz czasu potrzebnego na obliczenia. Funkcja celu sprowadza się do długości (liczba znaków) najdłuższego ciągu znaków i jest to najważniejszy parametr, który staramy się minimalizować. Czas obliczeń jest drugorzędny i ważniejsze dla nas jest otrzymanie lepszego wyniku, ale w momencie kiedy będzie zdecydowanie zbyt długi, to rozwiązanie może stać się nieakceptowalne. W ostateczności zmniejszymy liczbę znaków z 7 na 6, aby zmniejszyć liczbę koniecznych obliczeń.

Jako punkt odniesienia dla wartości funkcji celu mamy dodatkowo wynik 2428 - najlepsze znalezione w konkursie rozwiązanie.

Algorytm najbliższego sąsiada

Oczywistymi i największymi zaletami algorytmu najbliższego sąsiada są prostota jego implementacji i szybkość wykonania. Nawet dla permutacji siedmio elementowych pojedyncze wykonanie algorytmu zajmowało nam zaledwie kilkanaście sekund.

Co było dla nas dość zaskakujące, to relatywnie wysoka jakość rozwiązań zwracanych przez algorytm. W najlepszym przypadku podstawowy algorytm zwrócił rozwiązanie o wyniku 2633. Jest to wynik gorszy jedynie o niecałe 8% od najlepszego znalezionego do tej pory rozwiązania konkursu - 2428.

Ze względu na zachłanność tego algorytmu wiadomo, że nie znajduje rozwiązań optymalnych, ponieważ nie dokonuje szerszej eksploracji przestrzeni potencjalnych rozwiązań.

Symulowane wyżarzanie

Algorytm ten wymaga zdecydowanie większej ilości obliczeń. Z tego powodu zdecydowaliśmy się zmniejszyć liczbę symboli w permutacji do 6. Mimo wszystko symulowane wyżarzanie do znalezienia sensownego rozwiązania potrzebuje przynajmniej kilku godzin. Milion iteracji dla 6 symboli zajmuje ponad godzinę pracy programu.

Szybko odrzuciliśmy pierwszą wersję algorytmu ze względu na zbyt duże uzależnienie jej od początkowego rozstawienia permutacji. Dzięki użyciu zamiany permutacji z różnych ciągów w wersji drugiej algorytm jest w stanie dużo bardziej eksplorować przestrzeń potencjalnych rozwiązań, a co za tym idzie ma również większą szansę na znalezienie rozwiązania bliżej optymalnego.

Dla pierwszej wersji algorytmu znalezione rozwiązanie po 3 milionach iteracji ma wynik 776, a dla drugiej 690. Jest to poprawa o prawie 1%, ale wciąż mimo tak dużej ilości obliczeń wynik jest gorszy od rozwiązania algorytmu najbliższego sąsiada, ponieważ dla zmniejszonej liczby symboli uzyskuje on wynik 405.

Dla pierwszej wersji algorytmu wynik ten zatrzymał się i nie zmieniał przez ostatni milion iteracji. Oznacza to najpewniej, że ze względu na niefortunny początkowy podział permutacji na samym początku nie byłby w stanie znaleźć lepszego rozwiązania. Za to wersja druga algorytmu co jakiś czas znajdowała lepszy wynik. Można po tym wnioskować, że w nieograniczonym czasie znalazłaby jeszcze lepsze rozwiązanie.

Połączenie algorytmów

Oba algorytmy mają swoje wady oraz zalety. Algorytm najbliższego sąsiada znajduje całkiem dobre rozwiązanie bardzo szybko, ale możemy mieć niemalże pewność, że nie jest ono optymalne. Symulowane wyżarzanie za to wymaga zdecydowanie większej ilości obliczeń, ale potencjalnie ma szansę na znalezienie optymalnego rozwiązania poprzez ciągłe udoskonalanie wyniku w każdej iteracji.

Naturalnym rozwiązaniem jest połączenie działania obu algorytmów, aby przetestować czy jesteśmy w stanie zachować zalety obu rozwiązań, przy jednoczesnej niwelacji ich wad. W tym celu jako początkowy harmonogram w algorytmie symulowanego wyżarzania w naszej drugiej wersji ustawiliśmy rozwiązanie z podstawowego algorytmu najbliższego sąsiada. Oznacza to, że symulowane wyżarzanie już na starcie ma całkiem dobre rozwiązanie i jego zadaniem jest jedynie ulepszanie go. Przykładowy początkowy ciąg dla sześciu symboli ma długość 405. Po działaniu algorytmu przez 3 miliony iteracji udało się zmniejszyć ten wynik do 403. Poprawa jest znikoma, ale im lepszy wynik tym ciężej go poprawić.

Po tych eksperymentach można wnioskować, że wykorzystanie jednocześnie dwóch algorytmów umożliwia uzyskanie jeszcze lepszych wyników.

Wpływ dodania blanków na wyniki algorytmów

Tak jak zostało to opisane przy omawianiu wprowadzenia blanków do algorytmów, ich wpływ na rozwiązanie jest ograniczony. W kontekście najlepszych rozwiązań znalezionych w konkursie optymalne dodanie dwóch blanków dało poprawę wyniku o zaledwie pół procenta. Ponieważ nasze algorytmy nie zwracają tak dobrych wyników, dodanie blanków ma jeszcze mniej zauważalny efekt.

Dodanie blanków ma większe znaczenie dla mniejszej liczby symboli, co mogłoby być istotne dla naszych testów na sześciu symbolach. Mimo to, jednak nawet dla najlepszego rozwiązania znalezionego przez nasz algorytm NN z blankami (403), różnica i tak nie mogła sięgnąć nawet trzech procent. Jest to mniej niż wynoszą wahania w rezultatach wynikające z losowości wyboru sąsiada z najbliższych w algorytmie.

Dla wersji z oryginalną liczbą symboli samo dodanie blanków zmniejszyło nam najlepszy osiągnięty wynik z 2633 na 2631 dla algorytmu najbliższego sąsiada, co mieści się w oczekiwaniach.

Poprawę również można zaobserwować w przypadku algorytmu symulowanego wyżarzania. Po takiej samej ilości iteracji algorytm z zaimplementowaną optymalizacją blanków był w stanie zmniejszyć wynik z 690 do 679. Oznacza to, że ich wprowadzenie pozwoliło na efektywne zmniejszenie wyniku, a w przypadku nieograniczonego czasu obliczeń wynik mógłby być jeszcze lepszy.

Znikoma poprawa w algorytmie NN wynika z zachłanności rozwiązania, które nie próbuje już usprawnić go poprzez przykładowo dalsze przemieszanie permutacji, tylko znajduje miejsca do poprawy w już gotowych harmonogramach.

Podsumowanie

Ostatecznie najlepsze rozwiązanie dla problemu konkursowego jakie udało nam się uzyskać, otrzymaliśmy wykorzystując algorytm najbliższego sąsiada i dodając do niego blanki, z wynikiem 2631.

Próby poprawienia wyników algorytmu NN dla 7 symboli wykorzystując wyżarzanie nie powiodły się ze względu na zbyt długi czas trwania obliczeń.