Autostopem przez gALAIktykę: Intuicyjne omówienie zagadnień

Tom I: Optymalizacja

Nie panikuj!

Autorzy: **Iwo Błądek**
Agnieszka Mensfelt
Konrad Miazga

Oświadczamy, że w trakcie produkcji tego tutoriala nie zginęły żadne zwierzęta, nie została obrażona żadna opcja polityczna i nie stwierdzono ataku filozoficznych zombie.

1 Wprowadzenie

Studencie lub studentko kognitywistyki, przeznaczenie chciało, żebyś uczestniczył/a w zajęciach ze Sztucznej Inteligencji i Sztucznego Życia. Przedmiot to zacny i głęboki, acz dla nieprzygotowanego umysłu ciężki do zgłębienia. Mając tedy powyższe na uwadze, ten dokument w Twoje ręce oddajemy. Staraliśmy się w nim, w sposób możliwie intuicyjny i prosty, przedstawić najważniejsze idee optymalizacji dotyczące. Również często popełniane błędy tu przedstawiliśmy, bo uchronić Cię przed tego typu potknięciami na zaliczeniach czy też egzaminie.

1.1 Prawa autorskie i tym podobne

Niektóre obrazki w tym dokumencie pochodzą z `http://www.freepik.com`.

2 Optymalizacja – informacje wstępne

W codziennym życiu cały czas mamy do czynienia z sytuacjami, w których zależy nam na osiągnięciu w czymś jak najlepszego rezultatu. Możemy na przykład być na zakupach i chcieć kupić dobre buty w jak najniższej cenie. W innym wypadku mamy egzamin i musimy tak zaplanować naukę, by osiągnąć jak najwyższą ocenę (albo po prostu zdać – cele zależą od osoby). W tej części przedmiotu zajmiemy się spojrzeniem na problem szeroko pojętej **optymalizacji** z perspektywy matematyki i informatyki. Takie spojrzenie poza wiedzą teoretyczną dostarcza również „praktyczne”[1] metody rozwiązywania tego typu problemów.

Wartością **optymalną** dla pewnego konkretnego problemu optymalizacji nazywamy taką wartość (np. ceny), której nie bylibyśmy w stanie poprawić nawet gdybyśmy bardzo się starali. Inaczej mówiąc, mając do wyboru nasz optymalny element i jakikolwiek inny, zawsze wybralibyśmy ten optymalny. Wartość optymalna w danym problemie zazwyczaj będzie minimalną lub maksymalną możliwą do osiągnięcia wartością. Czasami jednak optimum wypada po środku, jak to ma miejsce na przykład w przypadku temperatury: nie powinna być ani zbyt niska ani zbyt wysoka. W takich wypadkach możemy jednak przeformułować problem jako różnicę między aktualną a naszą ulubioną temperaturą.

2.1 Problem imprezy

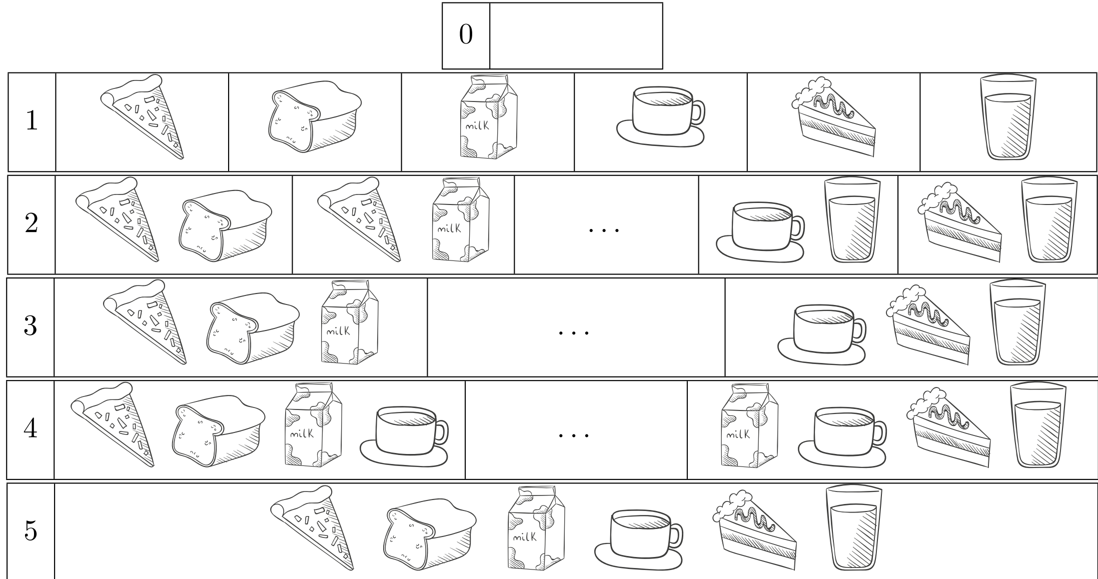
Wyobraźmy sobie pewien konkretny problem, na przykładzie którego omówimy podstawowe pojęcia związane z optymalizacją. Będzie to problem organizacji imprezy. Zapraszamy znajomych i chcemy ich ugościć jakimś jedzeniem i piciem. Wiemy, jak bardzo znajomych ucieszy obecność pewnego produktu podczas imprezy – w poniższej tabelce informacja ta zawarta jest w kolumnie 'aprobata gości'. Niestety, nasz budżet jest ograniczony do 60 zł (resztę pensji wydaliśmy na wcześniejsze imprezy). **Zakładamy, że albo zapewniamy dany produkt wszystkim gościom, albo nikomu, tak by nikt nie czul się poszkodowany**. Ceny w tabelce są sumarycznymi kwotami wymaganymi do zapewnienia każdemu gościowi danego jedzenia lub picia. Chcemy zapewnić gościom takie produkty, które sumarycznie uczynią ich najbardziej szczęśliwymi i nie przekroczą naszego budżetu 60 zł.

produkt	cena [zł]	aprobata gości [1-5]
	30	5
	5	2
	10	1
	12	3
	18	4
	20	5

2.2 Rozwiązanie problemu optymalizacji

Rozwiązaniem problemu optymalizacyjnego jest pewne podstawienie wartości do **zmiennych decyzyjnych** w nim występujących. Intuicyjnie, zmienne decyzyjne to te „elementy” w

[1] W ich praktycznym zastosowaniu pomaga często umiejętność programowania, choć nie zawsze jest wymagana (patrz: programowanie liniowe) :)



Rysunek 1: Wizualizacja przestrzeni przeszukiwania dla problemu imprezy. W każdym rzędzie przedstawione są wszystkie możliwe zbiory (produktów) o zadanym rozmiarze. Każde pole w tabeli to pewne rozwiązanie – zbiór produktów do kupienia. Chcemy wybrać spośród nich wszystkich ten najlepszy ze względu na nasze kryterium (funkcja celu) oraz spełniający ograniczenia.

problemie, które możemy wedle życzenia zmieniać.

W naszym problemie imprezy zmiennymi decyzyjnymi ***nie*** są ani cena, ani aprobata gości przypisana do konkretnego produktu, ponieważ są one zadane z góry, stałe. Zastanówmy się, co możemy zmieniać w tym problemie. Hmm... produkty, które zakupimy? Dokładnie tak! Jedyne na co mamy wpływ to zakupienie lub też nie pewnego produktu. Kiedy zmienne są nie tyle liczbowe, ile udzielają raczej odpowiedzi 'tak' lub 'nie' na pewne pytanie, to nazywamy je **zmiennymi binarnymi**. W tym wypadku nasze zmienne są właśnie tego typu. We wszelkich równaniach matematycznych będziemy je interpretować jako 1 w przypadku 'tak' i jako 0 w przypadku 'nie'.

2.3 Przestrzeń przeszukiwania

Jednym z podstawowych pojęć w optymalizacji jest **przestrzeń przeszukiwania**. Zawiera ona w sobie wszystkie możliwe potencjalne rozwiązania dla naszego problemu optymalizacji. Intuicyjnie, są to wszystkie opcje spomiędzy których możemy wybierać.

Na Rysunku 1 przedstawione są wszystkie możliwe opcje w naszym problemie organizacji imprezy. W celu wizualizacji pogrupowane zostały one w rzędy, a każdy rząd zawiera opcje reprezentujące zakupienie pewnej liczby produktów (0, 1, 2, itd.). Jedna z nich (a być może nawet kilka w przypadku „remisu") jest opcją optymalną. Nie wiemy jednak, która to spośród tych przedstawionych na rysunku. Chcemy ją znaleźć!

W algorytmach dokładnych zasadniczo przeszukujemy wszystkie rozwiązania z przestrzeni rozwiązań, gdyż zależy nam na gwarancji optymalności. Często jednak „struktura" problemu pozwala na niesprawdzanie części rozwiązań. Gdy jakiś zbiór produktów przekracza maksymalną kwotę, którą jesteśmy w stanie wydać, to dodawanie do niego kolejnych produktów nie poprawi nam sytuacji, prawda?

2.4 Funkcja celu

Funkcja celu wskazuje na to, co chcemy osiągnąć. Dokonuje ona tego poprzez przypisywanie pewnej wartości (oceny) każdemu rozwiązaniu problemu. Szukamy takiego rozwiązania, które będzie miało jak najlepszą ocenę zwróconą przez funkcję celu. Ocenę zawsze chcemy albo maksymalizować albo minimalizować, zależnie od rozważanego problemu.

W naszym problemie imprezy funkcja celu przypisuje każdemu rozwiązaniu sumę aprobaty gości. Chcemy, by aprobata była jak najwyższa, więc jest to problem **maksymalizacji**. Zauważ, że cena nie jest brana pod uwagę w funkcji celu (jest ona ograniczeniem, o czym za chwilę). Przykładowe wartości, które zwróciłaby nasza funkcja celu, przedstawione są w poniższej tabelce:

nr	rozwiązanie	wartość funkcji celu
1		$4 + 3 = 8$
2		$4 + 5 + 5 = 14$
3		0

Zwróć uwagę, że niekupienie żadnych produktów (rozwiązanie nr 3 w tabelce) też jest jednym z możliwych rozwiązań tego problemu, jednak widać, że na pewno nie będzie to rozwiązanie optymalne (chyba że nie byłoby nas stać na żaden produkt). Ważne: **praktycznie nigdy w momencie rozpoczynania przeszukiwania nie znamy wartości funkcji celu dla optymalnego rozwiązania**.

2.5 Ograniczenia

Zazwyczaj nie jest tak, że wszystkie rozwiązania są dozwolone. Praktycznie zawsze pojawiają się pewne **ograniczenia**, czyli stwierdzenia dotyczące tego, które rozwiązania są dopuszczalne, a które nie. Ważne: **ograniczenia są funkcją (mówiąc matematycznie) zmiennych decyzyjnych, czyli muszą je w jakiś sposób uwzględniać**. Cała idea ograniczeń polega na tym, że pewne zakresy wartości zmiennych decyzyjnych stają się „niedopuszczalne” z perspektywy rozważanego problemu.

W naszym problemie imprezy mamy ograniczenie na ilość pieniędzy, które możemy wydać (60 zł). Interesują nas wyłącznie te rozwiązania, których suma kosztów jest mniejsza niż 60 zł. **Dopuszczalne** będą więc te zbiory produktów, które zmieszczą się poniżej tej kwoty. W poniższej tabelce znajdują się wyliczone sumaryczne ceny dla przykładowych zbiorów.

nr	rozwiązanie	sumaryczna cena [zł]
1		$18 + 12 = 30$
2		$18 + 30 + 20 = 68$
3		0

Jak można zauważyć, zbiory produktów 1 i 3 mają sumaryczną cenę mniejszą od 60 zł. Inaczej sytuacja wygląda dla zbioru produktów 2, którego cena wynosi 68 zł. Tak więc pomimo tego, że zbiór produktów 2 ma bardzo dobrą wartość funkcji celu, nie może on być rozwiązaniem optymalnym, gdyż łamie nasze ograniczenie.

3 Programowanie matematyczne

Programowanie matematyczne jest metodą optymalizacji, w której konstruowany jest model matematyczny problemu.

3.1 Zmienne decyzyjne

Rozwiązanie problemu modelowane jest przez zmienne decyzyjne. Różne zestawy wartości jakie mogą przyjąć zmienne decyzyjne odpowiadają różnym rozwiązaniom problemu. Dla rozpatrywanego powyżej problemu imprezy rozwiązaniem jest zbiór produktów, które zostaną zakupione. Zauważyliśmy, że dla każdego produktu musimy podjąć decyzję czy kupimy go czy nie. W związku z tym, możemy zdefiniować następujące zmienne decyzyjne:

x_i - decyzja czy i-ty produkt zostanie zakupiony, $x_i \in \{0, 1\}$, $i = 1, .., 6$

Przykładowe rozwiązania to np.:

(a) $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1, x_6 = 0$: kupujemy pizzę, chleb i tort

(b) $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 0$: kupujemy chleb i mleko

(c) $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 1, x_6 = 0$: kupujemy kawę i tort

W rozpatrywanych przez nas problemach będziemy mieć najczęściej do czynienia z następującymi typami zmiennych:

rzeczywiste zmienna może przyjąć dowolną wartość, nie nakładamy żadnych ograniczeń na wartości przyjmowane przez zmienną

całkowitoliczbowe, $x_i \in \mathbb{Z}$, zmienna może przyjmować tylko wartości całkowite, np. problem produkcji, gdzie nie mamy możliwości wyprodukowania $2\frac{3}{4}$ stołu

binarne, $x_i \in \{0, 1\}$, wartość zmiennej należy do zbioru $\{0, 1\}$, np. problem w którym odnośnie każdego z elementów rozwiązania musimy podjąć decyzję tak (1) lub nie (0)

3.2 Funkcja celu

Aby odnaleźć rozwiązanie najlepsze pod względem przyjętych przez nas kryteriów niezbędna jest funkcja celu. Funkcja ta przypisuje każdemu rozwiązaniu wartość określającą na ile jest ono dobre. Skoro funkcja celu ma pozwolić porównać pomiędzy sobą rozwiązania, a rozwiązania są modelowane za pomocą zmiennych decyzyjnych, funkcja ta musi być funkcją zmiennych decyzyjnych. Ustaliliśmy, że w problemie imprezy chcemy maksymalizować zadowolenie gości. Zadowolenie gości z konkretnego rozwiązania, np. rozwiązania (a) z poprzedniego rozdziału, możemy policzyć jako:

$$5 + 2 + 4 = 11$$

Zadowolenie z dowolnego rozwiązania obliczmy w następujący sposób:

$$z = x_1 \cdot a_1 + x_2 \cdot a_2 + x_3 \cdot a_3 + x_4 \cdot a_4 + x_5 \cdot a_5 + x_6 \cdot a_6$$

gdzie a_i to stała określająca aprobatę dla i-tego produktu. Bardziej zwięźle można to zapisać jako:

$$z = \sum_{i=1}^{6} x_i \cdot a_i$$

3.3 Ograniczenia

W większości rzeczywistych problemów pojawiają się ograniczenia, np. czas, powierzchnia, czy, jak w problemie imprezy, pieniądze. Ograniczenia, podobnie jak funkcja celu, są funkcjami zmiennych decyzyjnych. Rozwiązania, które spełniają ograniczenia nazywamy *rozwiązaniami dopuszczalnymi*, takie które ich nie spełniają, *rozwiązaniami niedopuszczalnymi*. W problemie imprezy liczbę produktów, które możemy zakupić, ogranicza ich sumaryczna cena – nie możemy wydać więcej niż 60 zł. Koszt konkretnego rozwiązania, np. rozwiązania (a) z poprzedniego rozdziału możemy policzyć jako:

$$30 + 5 + 18 = 53$$

Koszt dowolnego rozwiązania obliczmy w następujący sposób:

$$koszt = x_1 \cdot c_1 + x_2 \cdot c_2 + x_3 \cdot c_3 + x_4 \cdot c_4 + x_5 \cdot c_5 + x_6 \cdot c_6$$

gdzie c_i to stała określająca cenę i-tego produktu. Bardziej zwięźle można to zapisać jako:

$$koszt = \sum_{i=1}^{6} x_i \cdot c_i$$

Ponieważ koszt nie może być większy niż 60 zł ostatecznie ograniczenie będzie wyglądać następująco:

$$\sum_{i=1}^{6} x_i \cdot c_i \leqslant 60$$

W standardowej postaci problemu programowania liniowego[2] dodawane jest jeszcze ograniczenie na nieujemność zmiennych decyzyjnych:

$$x_i \geqslant 0, \quad i = 1, ..., n$$

W praktycznych problemach jest to też często ograniczenie zdroworozsądkowe, gdyż nie jesteśmy przecież w stanie wyprodukować -2 stołów.

3.4 Definicja problemu programowania matematycznego

Podsumowując, ogólna definicja problemu programowania matematycznego wygląda następująco:

$$\max/\min\ z = f(x_1, x_2, ..., x_n)$$

$$\text{przy ograniczeniach:}$$

$$g_1(x_1, x_2, ..., x_n) \leqslant 0$$

$$g_2(x_1, x_2, ..., x_n) \leqslant 0$$

$$...$$

$$g_m(x_1, x_2, ..., x_n) \leqslant 0$$

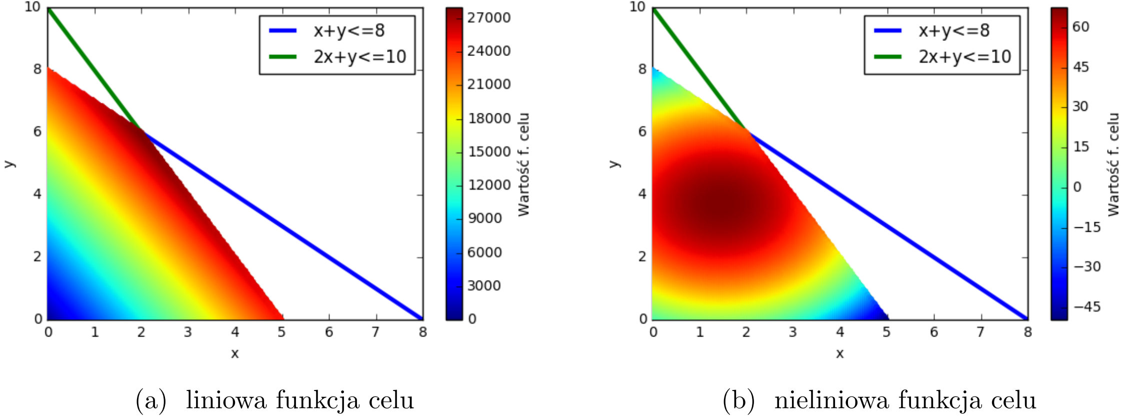
Jeśli funkcja celu i wszystkie ograniczenia są funkcjami liniowymi to mamy do czynienia z problemem programowania matematycznego liniowego, jeśli natomiast chociaż jedna z tych funkcji jest nieliniowa, to mamy do czynienia z problemem nieliniowym. Rysunek 2 przedstawia problem z liniową i nieliniową funkcją celu. W przypadku problemów programowania liniowego rozwiązanie, o ile istnieje, znajduje się w wierzchołku obszaru rozwiązań dopuszczalnych.

4 Modelowanie problemów optymalizacji

4.1 Optymalizacja kombinatoryczna

W poprzednim rozdziale zamodelowaliśmy problem ustalenia menu imprezy jako problem programowania matematycznego. Nasz problem moglibyśmy zamodelować również jako problem optymalizacji kombinatorycznej. Problem optymalizacji kombinatorycznej cechuje się zbiorem instancji, gdzie instancja to para: (S, f), S to zbiór wszystkich możliwych rozwiązań, a f to funkcja celu przypisująca rozwiązaniom wartość funkcji oceny. Nasz problem ułożenia menu z 5 produktów jest instancją problemu układania menu, rozmiar zbioru wszystkich rozwiązań S określimy w punkcie 4.3, funkcję oceny rozwiązania zdefiniowaliśmy w punkcie 3.2.

[2] pl.wikipedia.org/wiki/Programowanie_liniowe



(a) liniowa funkcja celu

(b) nieliniowa funkcja celu

Rysunek 2: Problem z liniową (a) i nieliniową (b) funkcją celu.

4.2 Określenie czym jest rozwiązanie w problemie optymalizacji

Modelowanie problemu rozpoczynamy od określenia co jest **rozwiązaniem** tego problemu. W przypadku problemu imprezy rozwiązaniem jest lista produktów, które zakupimy na imprezę. Wyobraźmy sobie, że naszym pomysłem na startup jest aplikacja układająca optymalne menu na różne okazje (Rysunek 3). Użytkownik aplikacji wprowadzałby maksymalny koszt zakupów oraz preferencje swoje i gości, których postanowił zaprosić (np. słodkie, słone, obiadowe, tradycyjne, niskokaloryczne, chińskie, tajskie, wegetariańskie, bezglutenowe itd.). Następnie aplikacja wykluczałaby ze zbioru produktów te które wykluczają preferencje użytkownika, np. produkty mięsne w przypadku menu wegetariańskiego. Każdemu z pozostałych produktów, byłaby przypisywana aprobata, wyliczana na podstawie preferencji użytkownika. Po odnalezieniu rozwiązania posiadającego największą wartość funkcji celu (sumarycznej aprobaty) byłoby ono prezentowane użytkownikowi w postaci listy zakupów. Zwróćmy uwagę, że tym co interesuje użytkownika nie jest koszt (jest on ustalony), ani wartość funkcji celu, tylko **rozwiązanie** – lista zakupów do zrobienia na imprezę.

Rysunek 3: Interfejs aplikacji układającej optymalne menu, wersja pre-alfa.

Po określeniu czym jest rozwiązanie w rozpatrywanym problemie optymalizacji możemy za-

stanowić się jak reprezentować rozwiązanie, tak aby można było zastosować do niego algorytmy optymalizacji. Pomoże nam to też określić ile jest wszystkich potencjalnych rozwiązań danego problemu. Jak już zauważyliśmy, w problemie menu dla każdego z rozpatrywanych produktów możemy podjąć dwie decyzje - tak lub nie. Rozwiązanie możemy więc reprezentować za pomocą ciągu binarnego, gdzie 1 odpowiada wybraniu danego produktu, a 0 niewybraniu go, np.:

pizza	chleb	mleko	kawa	tort	napój
1	0	1	1	0	1

4.3 Określenie liczby rozwiązań

Dla pierwszego produktu (pizzy) mamy dwie możliwości wyboru (tak lub nie), dla drugiego (chleba) również dwie, itd. Ponadto, wybór danego produktu nie wpływa na możliwość wyboru innego. Ostatecznie, liczba wszystkich potencjalnych rozwiązań to:

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^6$$

Rozpatrzmy inny problem – problem komiwojażera. Ma on odwiedzić n miast, trasa musi przechodzić przez każde miasto dokładnie raz. Zatem rozwiązaniem w tym problemie jest trasa – lista miast, które po kolei odwiedzi komiwojażer. Reprezentacją rozwiązania jest permutacja n-elementowa, gdzie n to liczba miast. Ile jest wszystkich permutacji bez powtórzeń n elementów? Pierwsze miasto możemy wybrać spośród n miast, kolejne już tylko spośród $n-1$, ponieważ jedno z miast już znalazło się na trasie, następne spośród $n-2$, itd. Ostatecznie, liczba permutacji n elementów to:

$$n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1 = n!$$

Zastanówmy się jednak, czy wszystkie z $n!$ permutacji reprezentują różne trasy? Zakładając, że wracamy do miasta początkowego trasa Poznań-Warszawa-Kraków-Wrocław jest taką samą trasą jak Warszawa-Kraków-Wrocław-Poznań. Takich tras, gdzie miasta są w tej samej kolejności, ale różnią się pozycją miast na liście jest n. Dodatkowo, jeżeli jako miarę odległości przyjmujemy dystans w kilometrach, to problem jest symetryczny i trasa Poznań-Warszawa-Kraków-Wrocław jest taką samą trasą jak Wrocław-Kraków-Warszawa-Poznań. Zatem każda trasa jest wyrażona na $2 \cdot n$ różnych sposobów, więc liczbą różnych rozwiązań będzie $\frac{n!}{2n}$.

4.4 Zdefiniowanie funkcji oceny i ograniczeń

Funkcja oceny pozwala na porównanie pomiędzy sobą rozwiązań i wybór najlepszego. Funkcję celu dla problemu ułożenia menu zdefiniowaliśmy w punkcie 3.2 jako sumaryczne zadowolenie gości. W pierwotnej wersji problemu założyliśmy, że mamy daną z góry aprobatę gości, zatem rozwiązania oceniamy tylko na jednym kryterium - aprobacie gości, więc funkcja oceny jest jednokryterialna.

Projektując aplikację, chcemy obarczyć użytkownika jak najmniejszą ilością pracy przy układaniu menu. W związku z tym nie zakładamy, że dostarczy listę wszystkich możliwych produktów, nasza aplikacja będzie w nią wyposażona. Nie oczekujemy też, że użytkownik będzie obliczał aprobatę dla każdego z możliwych produktów, ma on tylko wprowadzić swoje preferencje odnośnie menu (np. słodkie, słone, obiadowe, tradycyjne, niskokaloryczne, chińskie, tajskie, wegetariańskie, bezglutenowe itd.)[3]. Widzimy więc, że rozwiązania będą ocenianie na wielu kryteriach. Zastanówmy się nad innymi udogodnieniami w organizacji imprezy jakie mogłaby oferować aplikacja. Jeśli dysponowalibyśmy informacją o tym czy dany produkt znajduje się w konkretnym sklepie moglibyśmy dodać kolejne kryterium oceny rozwiązania – sumaryczna odległość do najbliższych sklepów, w jakich znajdują się produkty.

[3] W wersji płatnej można by rozważyć zastosowanie metod uczenia maszynowego do ustalenia preferencji użytkownika na podstawie jego aktywności na urządzeniu mobilnym.

Definicja funkcji oceny i ograniczeń zależy od celu. W przypadku organizacji imprezy założyliśmy, że chcemy maksymalizować zadowolenie gości mierzone przez aprobatę dla produktów spożywczych. Nie możemy kupić każdego ze wszystkich rozpatrywanych produktów, ponieważ ogranicza nas budżet. Chcemy, aby projektowana przez nas aplikacja miała jak najszerszy zakres użytkowników, więc zastanawiamy się czy może są jacyś potencjalni użytkownicy o odmiennych celach. Rozpatrzmy problem osoby, która organizuje tradycyjne wesele. Ma ona już bardzo ograniczone środki, ponieważ dużo pieniędzy przeznaczyła na salę, orkiestrę itd. Zależy jej, aby na stole znalazły się potrawy spełniające określone kryteria, ale chce przeznaczyć na ten cel jak najmniej środków. W tym przypadku funkcją oceny mógłby być sumaryczny koszt menu, a ograniczeniem sumaryczna aprobata wynikająca z preferencji co do potraw. Rozważamy więc wprowadzenie trybu maksymalizacji aprobaty i trybu minimalizacji kosztu, w zależności od celu użytkownika.

5 Algorytmy

Po stworzeniu modelu problemu nadchodzi czas na rozwiązanie go za pomocą wybranego algorytmu/algorytmów. W tym rozdziale krótko omówimy ogólną ideę heurystyki i algorytmu przeszukiwania, natomiast w rozdziale 6 i 7 dokładniej zostaną omówione algorytmy przeszukiwania lokalnego i algorytmy ewolucyjne. **Heurystyki** „konstruujące” odróżniamy od **metaheurystyk**. Czasem można spotkać się z użyciem określenia „heurystyka” w odniesieniu do metaheurystyk, dlatego dla ujednoznacznienia dodajemy słowo „konstruująca”.

5.1 Heurystyki „konstruujące”

Heurystyka jest algorytmem, który buduje rozwiązanie wykorzystując wiedzę na temat problemu. Jaką heurystykę moglibyśmy zaproponować dla problemu ułożenia menu? Wiadomo że naszym celem jest maksymalizacja aprobaty gości. Można by więc posortować produkty po aprobacie gości, wtedy lista wyglądałaby następująco:

1. napój – 5
2. pizza – 5
3. tort – 4
4. kawa – 3
5. chleb – 2
6. mleko – 1

Zauważmy, że pizza i napój mają taką samą aprobatę gości – możemy np. przyjąć zasadę, że jeśli produkty mają taką samą aprobatą jako pierwszy na liście umieszczamy ten, który ma niższą cenę. Teraz wybieramy produkty zaczynając od początku listy tak długo, aż nie przekroczymy budżetu. Na początku umieszczamy w menu napój, ponieważ ma największą aprobatę, następnie dodajemy pizzę. Nie możemy już dodać trzeciego pod względem aprobaty tortu ani czwartej kawy, ponieważ przekroczylibyśmy budżet. Możemy natomiast dodać chleb. To już ostatni produkt, który możemy dodać nie przekraczając budżetu. Rozwiązaniem wygenerowanym przez heurystykę jest zbiór {napój, pizza, chleb}. Wartość funkcji oceny dla tego rozwiązania to 12, a koszt to 55 zł.

Oczywiście, możemy zaproponować wiele innych heurystyk generujących rozwiązanie dla tego problemu. Zaletą heurystyki „konstruującej” jest szybkość działania, wadą brak gwarancji znalezienia rozwiązania optymalnego i konieczność znajomości dziedziny problemu.

5.2 Algorytm przeszukiwania

W algorytmach przeszukiwania generowane i oceniane są rozwiązania ze zbioru wszystkich możliwych rozwiązań.

Algorytm przeszukiwania dokładnego Pierwszym pomysłem na algorytm przeszukiwanie może być przeszukiwanie dokładne – wygenerowanie i ocena wszystkich potencjalnych rozwiązań. Niewątpliwą zaletą tego algorytmu jest gwarancja znalezienia rozwiązania optymalnego. Niestety, ze względu na rozmiar przestrzeni rozwiązań dla niektórych problemów niemożliwe może być uzyskanie rozwiązania w sensownym czasie.

Algorytm przeszukiwania losowego Kolejnym podejściem do przeszukiwania przestrzeni rozwiązań może być generowanie losowych rozwiązań tak długo aż nie upłynie określony czas. Każdorazowo po wygenerowaniu losowego rozwiązania sprawdzamy, czy wartość jego funkcji celu jest lepsza niż dla najlepszego dotąd znalezionego rozwiązania. Jeśli tak, staje się ono nowym najlepszym dotąd znalezionym rozwiązaniem.

Metaheurystyki Metaheurystyki są strategiami przeszukiwania przestrzeni rozwiązań, które nie wymagają, w przeciwieństwie do heurystyk „konstruujących", wiedzy o problemie. Np. tę samą metaheurystykę lokalnego przeszukiwania możemy zastosować do problemu układania menu na imprezę i dla problemu komiwojażera, po uprzednim zdefiniowaniu reprezentacji rozwiązania i relacji sąsiedztwa. Metaheurystyki nie dają gwarancji znalezienia rozwiązania optymalnego, ale zwykle pozwalają na znalezienie rozwiązania bliskiego optymalnemu.

6 Przeszukiwanie lokalne

Niektóre problemy w praktyce okazują się zbyt trudne aby móc je opisać metodą programowania liniowego, bądź też stworzyć algorytm gwarantujący znalezienie rozwiązania optymalnego w rozsądnym czasie. Przykładem takiego problemu może być **problem komiwojażera (TSP, traveling salesman problem)**.

Przeszukiwanie lokalne (LS, local search) jest metaheurystyką. Oznacza to dwie rzeczy:

a) nie daje ono gwarancji znalezienia rozwiązania optymalnego,

b) można je zastosować do praktycznie dowolnego problemu.

Jednocześnie jednak, LS będzie zazwyczaj działać szybko i często będzie znajdywać stosunkowo dobre rozwiązania, w związku z czym jest to metoda optymalizacji często stosowana w praktyce.

Zanim przejdziemy dalej, warto w paru zdaniach podsumować całą ideę przeszukiwania lokalnego. Przyjmijmy najpierw trzy założenia:

a) jesteśmy w stanie ocenić jakość każdego rozwiązania (jego **fitness**),

b) dla każdego rozwiązania jesteśmy w stanie podać zbiór rozwiązań podobnych (jego **sąsiedztwo**),

c) rozwiązania podobne będą podobne nie tylko pod względem zapisu, ale również wartości funkcji celu.

Zaczynamy poszukiwania naszego dobrego rozwiązania od pewnego losowego, niekoniecznie dobrego rozwiązania. Sprawdzamy jakość każdego z jego sąsiadów i przechodzimy do najlepszego z nich jeśli jest lepszy od naszego rozwiązania aktualnego. Rozwiązanie do którego przechodzimy staje się naszym nowym rozwiązaniem aktualnym. Powtarzamy ten proces tak długo, aż żaden z sąsiadów naszego rozwiązania aktualnego nie jest od niego lepszy (rozwiązanie takie nazwiemy **optimum lokalnym**, w odróżnieniu od **optimum globalnego** które jest najlepszym możliwym rozwiązaniem) – kończymy wtedy wykonanie algorytmu.

Przydatnym może okazać się wyobrażenie sobie działania LS jako wspinaczkę górską we mgle. Chcemy wejść na szczyt, nie wiemy jednak gdzie się on znajduje – mgła ogranicza nasze pole widzenia jedynie do naszego najbliższego otoczenia (rozwiązania podobne, sąsiedzi). Nie wiedząc nic o geografii okolicy, naturalnym pomysłem będzie więc kierowanie się w stronę gdzie teren się najbardziej podwyższa. Ostatecznie skończymy w punkcie, który jest lokalnie najwyższy. Jako

Rysunek 4: Nie zawsze najbardziej stromy szczyt jest najwyższy.

że jest mgła i nie możemy stwierdzić czy jest to nasz wyśniony szczyt czy też nie – kończymy wędrówkę.

6.1 Greedy i Steepest

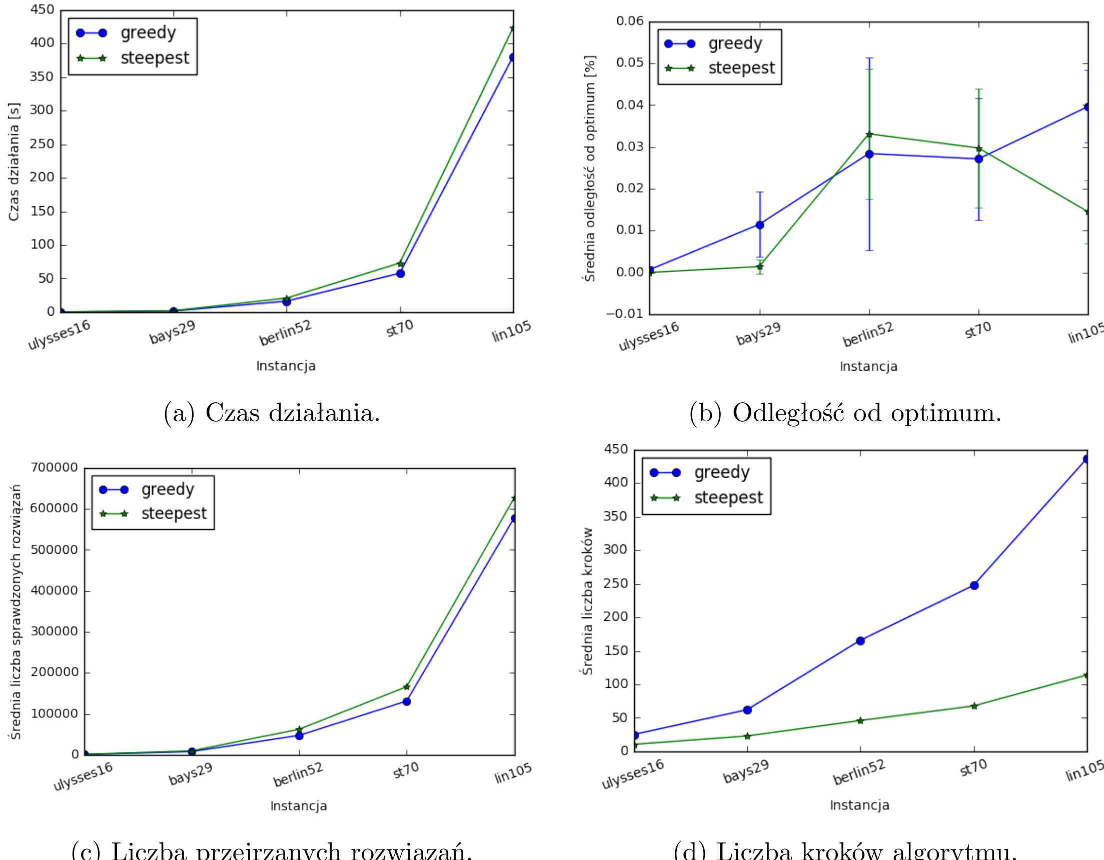
Gwoli ścisłości, przeszukiwanie lokalne występuje w dwóch odmianach: **greedy** i **steepest**. Powyższy opis dotyczy odmiany **steepest**, która (jak sama nazwa wskazuje) zawsze będzie przechodziła do najlepszego rozwiązania sąsiedniego (odwołując się do naszej metafory: wspina się zawsze tam gdzie najbardziej stromo). Istnieje jednak również odmiana **greedy**, która różni się od *steepest*'a tym, że nie przechodzi do najlepszego rozwiązania sąsiedniego, ale do pierwszego przejrzanego rozwiązania sąsiedniego które jest lepsze niż aktualne (a więc jest zachłanna / niecierpliwa). Zauważ, iż w przypadku metody *greedy* istotna może okazać się kolejność przeglądania rozwiązań sąsiednich!

Które z tych dwóch podejść jest lepsze? Można to oceniać na podstawie dwóch kryteriów: prędkości działania i jakości znajdywanych rozwiązań. Na pierwszy rzut oka może się wydawać, że *greedy* będzie szybszy (w końcu nie przegląda wszystkich rozwiązań w sąsiedztwie!) a *steepest* będzie znajdować lepsze rozwiązania (w końcu zawsze idzie w najlepszym możliwym kierunku!). W praktyce okazuje się jednak, iż **nie ma istotnych różnic między tymi podejściami**. Dlaczego?

Najpierw spójrzmy na czas działania. Podczas gdy prawdziwym jest stwierdzenie, iż *greedy* sprawdza w każdym kroku mniej rozwiązań sąsiednich, w praktyce oznacza to, że jego podróż do optimum lokalnego nie idzie najkrótszą ścieżką! W związku z tym, średnia liczba rozwiązań odwiedzonych przez metodę *greedy* w trakcie działania całego algorytmu jest porównywalna do tej uzyskiwanej przez metodę *steepest*.

Jeśli chodzi o jakość znajdowanych rozwiązań, to czy lepsze rozwiązanie znajdzie *greedy* czy *steepest* będzie zależało zarówno od rozwiązywanego problemu, jak i od obranego rozwiązania startowego. Mogłoby się wydawać, iż *steepest* powinien osiągać lepsze rezultaty, ponieważ zawsze wybiera najlepsze rozwiązanie w sąsiedztwie. Zauważ jednak, iż nie ma podstaw aby twierdzić, że przechodząc do rozwiązania które nie jest lokalnie najlepsze pogarszamy swoje szanse na znalezienie optymalnego rozwiązania – tak długo jak zawsze przechodzimy do rozwiązania lepszego niż aktualne, mamy gwarancję, iż rozwiązanie aktualne będzie się poprawiać w ramach naszej podróży! Wracając raz jeszcze do górskiej metafory: czasem *steepest* zaprowadzi nas na jakąś pomniejszą iglicę skalną, podczas gdy *greedy* spokojnym trawersem pozwoli nam dotrzeć na sam szczyt!

Gdyby powyższe rozważania nie wystarczyły aby Cię przekonać, na Rysunku 5 znajdują się wyniki eksperymentów dla różnych zestawów miast (oś X) w problemie komiwojażera.



(a) Czas działania.

(b) Odległość od optimum.

(c) Liczba przejrzanych rozwiązań.

(d) Liczba kroków algorytmu.

Rysunek 5: Porównanie działania metod *greedy* i *steepest*.

6.2 Schemat algorytmu

Poniższy (pythonopodobny) kod pokazuje ogólny schemat algorytmu lokalnego przeszukiwania (w tym wypadku w odmianie **greedy**). Jak widać jest on bardzo prosty (podobnie jak sam algorytm). Dysponując wiedzą na temat sposobu działania LS, spróbuj przeanalizować poniższy kod. Jakie jest znaczenie zmiennej *better*? Co by się stało gdybyśmy pominęli polecenie **break** z linii 10?

```
1  r = losowe_rozwiazanie()
2  better = True
3
4  while better:
5      better = False
6      for n in sasiedzi(r):
7          if fitness(n) > fitness(r):
8              r = n
9              better = True
10             break
```

6.3 Sąsiedztwo

Definicja **sąsiedztwa** jest prawdopodobnie najważniejszym elementem przygotowania problemu do bycia rozwiązanym za pomocą metody LS. Sąsiedztwo rozwiązania r to w praktyce skończony zbiór rozwiązań podobnych (najlepiej semantycznie) do rozwiązania r. Zazwyczaj rozwiązania podobne syntaktycznie będą jednocześnie podobne semantycznie, aczkolwiek nie zawsze musi tak być. Przykładem mogą być programy komputerowe – zmiana jednego warunku *if* (ich syntaktyka) może mieć olbrzymi wpływ na zwracane przez nie wartości (ich semantyka)!

Używając nieco ściślejszego języka: sąsiedztwem rozwiązania r będziemy nazywali taki zbiór rozwiązań $N(r)$, że dla każdego rozwiązania $n \in N(r)$ zachodzi $distance(r, l) < \epsilon$ (dla obranej miary odległości $distance$ między rozwiązaniami i wartości ϵ). W praktyce, zazwyczaj $\epsilon = 1$ ale możemy je zwiększać w celu powiększenia naszego sąsiedztwa (zauważ, że im większe ϵ tym więcej rozwiązań będzie należeć do naszego zbioru $N(r)$). Miara odległości $distance$ może być ustalona w dowolny sposób, należy jednak pamiętać, iż zależy nam aby rozwiązania które są blisko siebie (niska wartość $distance$) miały podobną wartość funkcji celu. Należy też upewnić się, iż nasza miara $distance$ spełnia warunek $distance(a, b) = distance(b, a)$ dla każdej pary rozwiązań a, b (oznacza to tyle, iż odległość z punktu a do punktu b jest równa odległości od punktu b do punktu a).

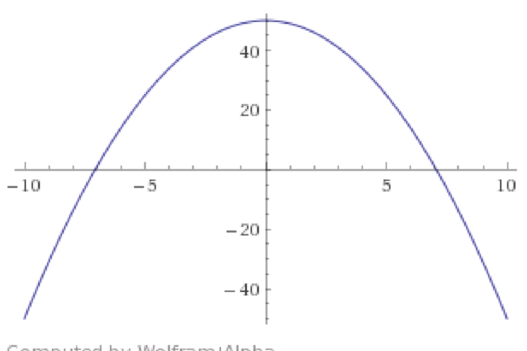
Poprawnie zdefiniowane sąsiedztwo powinno spełniać następujące warunki:

Ograniczenie na rozmiar Każde rozwiązanie powinno zawierać w swoim sąsiedztwie conajmniej jedno rozwiązanie inne niż ono samo - w przeciwnym razie nigdy nie będziemy w stanie go opuścić, a więc cała idea przeszukiwania lokalnego (przechodzenie pomiędzy sąsiadami) przestaje mieć sens!

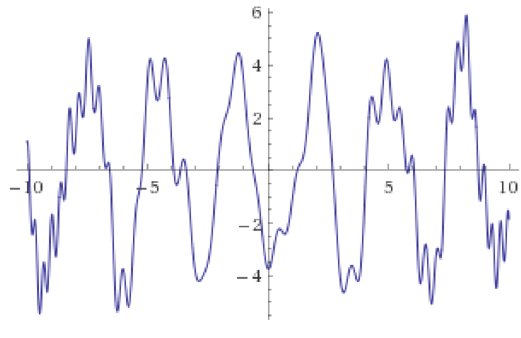
Z drugiej strony, sąsiedztwo nie powinno też nigdy zawierać wszystkich możliwych rozwiązań. Oznaczałoby to bowiem, iż w każdym kroku metody *steepest* dokonujemy pełnego przeglądu wszystkich rozwiązań (a w końcu używamy algorytmu LS aby tego właśnie uniknąć!).

W praktyce, rozmiar sąsiedztwa powinien być zawsze zbalansowany. Mniejsze sąsiedztwo oznacza, iż każdy krok algorytmu będzie trwał krócej (ponieważ będziemy w nim przeglądać mniej rozwiązań), jednak kroków tych będzie więcej, a ponadto mamy większą szanę utknąć w optimum lokalnym (ponieważ mamy niewiele opcji opuszczenia go). Z drugiej strony, duże sąsiedztwo minimalizuje szansę na utknięcie w optimum lokalnym i liczbę potrzebnych kroków, jednak znacznie wydłuża czas potrzebny no wykonanie każdego z nich.

Podobieństwo sąsiadów Rozwiązania sąsiednie powinny być do siebie podobne (najlepiej semantycznie) z dwóch względów.



(a) Gładki krajobraz dostosowania – LS bedzie działał bardzo dobrze.



(b) Poszarpany krajobraz dostosowania – LS szybko ugrzęźnie w optimum lokalnym.

Rysunek 6: Porównanie rodzajów krajobrazu przystosowania (oś pozioma – rozwiązania, podobne są blisko siebie; oś pionowa – jakość rozwiązań).

Po pierwsze, jeśli rozwiązania sąsiednie są do siebie podobne, nasza przestrzeń przeszukiwanych rozwiązań (tzw. **fitness landscape**, **krajobraz dostosowania**) będzie gładka – patrz Rysunek 6a. Gładki krajobraz dostosowania ogranicza liczbę optimów lokalnych, a tym samym podnosi szansę iż LS zaprowadzi nas do optimum globalnego. Sytuacja gdy rozwiązania sąsiednie nie są do siebie podobne (a więc krajobraz dostosowania jest bar-

dzo poszarpany) przedstawiona jest na Rysunku 6b – jak widać w takiej przestrzeni nasz algorytm nie byłby w stanie zbyt wiele zdziałać.

Po drugie, jeśli rozwiązania różnią się jedynie w niewielkim stopniu, możliwe, iż nie będziemy musili obliczać jakości rozwiazań sąsiednich od zera (co dla bardziej złożonych problemów może zająć sporo czasu), a jedynie zmodyfikować wartość funkcji celu naszego aktualnego rozwiązania (co może być znacznie szybsze).

Brak odizolowanych rozwiązań Nasza definicja sąsiedztwa musi zapewniać, iż będziemy w stanie przejść z każdego rozwiązania do każdego innego rozwiązania poruszając się od sąsiada do sąsiada. Jeśli warunek ten nie będzie spełniony, napotkamy problem podobny do tego gdy wielkość sąsiedztwa wynosi 1 – możliwe, iż nie będzie istniała ścieżka od naszego rozwiązania startowego do rozwiązania optymalnego, a więc nie będziemy w stanie go znaleźć!

6.4 Przykład praktyczny – problem komiwojażera

Aby zwizualizować w praktyce działanie przeszukiwania lokalnego, spróbujmy rozwiązać z jego pomocą problem komiwojażera. W tym celu musimy opracować trzy rzeczy:

a) reprezentację rozwiązania,

b) funkcję celu,

c) definicję sąsiedztwa.

Reprezentacja Dla każdego problemu istnieje wiele możliwych sposobów reprezentacji rozwiązania. Każda z tych reprezentacji będzie interpretowana w inny sposób.

W problemie komiwojażera rozwiązaniem jest trasa przechodząca przez wszytkie miasta, przez każde dokładnie raz. Jeśli ponumerujemy miasta (od 1 do n, gdzie n to liczba miast), możemy reprezentować trasę jako ciąg liczb od 1 do n bez powtórzeń. Cała informacja o trasie jest zatem zawarta w kolejności występowania liczb – każda permutacja takiego ciągu będzie osobnym, poprawnym rozwiązaniem. Przykładowo, jeśli mamy problem z pięcioma miastami, możemy zapisać jakieś (losowe) rozwiązanie jako ciąg:

$$\begin{bmatrix} 3 & 4 & 2 & 1 & 5 \end{bmatrix}$$

Rozwiązanie takie możemy interpretować następująco: zaczynajmy od miasta 3, potem odwiedzamy po kolei miasta 4, 2, 1 i 5 po czym wracamy do miasta 3.

W dalszej części naszego rozwiązania bedziemy korzystali z reprezentacji opisanej powyżej. W celu dydaktycznym przedstawimy jednak poniżej reprezentację alternatywaną, aby pokazać, że nie istnieje nigdy „jedyna właściwa reprezentacja" (chociaż niektóre bedą w praktyce lepsze niż inne).

Wyobraźmy sobie tablicę n wartości, gdzie n to liczba miast. i-ta pozycja tablicy odpowiadać bedzie miastu o numerze i. Na każdej pozycji tablicy mamy liczbę z zakresu $< 0; 1 >$. Możemy wtedy interpretować taką reprezentację w następujący sposób: odwiędzamy najpierw miasto o najniższej wartosci w tablicy, następnie miasto o kolejnej wyższej wartości, itd. Przykładowym rozwiązaniem dla pięciu miast mogłoby być:

$$\begin{bmatrix} 0.67 & 0.5 & 0.23 & 0.29 & 0.72 \end{bmatrix}$$

Jego interpretacja jest dokładnie sama jak dla przykładu w reprezentacji permutacyjnej.

Funkcja celu W przypadku problemu komiwojażera funkcją celu będzie długość trasy. Możemy ją obliczyć sumując odległości pomiędzy parami miast między którymi przejeżdżamy. Należy pamiętać jednak o dodaniu ponadto odległości między pierwszym i ostatnim odwiedzonym miastem (droga powrotna).

W problemie komiwojażera będziemy chcieli minimalizować wartość tak zdefiniowanej funkcji celu.

Definicja sąsiedztwa Popularnym sąsiedztwem dla rozwiązań reprezentowanych jako permutacje (jak ma to miejsce w naszym przypadku) jest tzw. **k-zamiana** (**k-opt**, **k-swap**). Polega ona na usunięciu z ciągu k elementów a następnie wstawieniu ich ponownie na wolne miejsca, w zmienionej kolejności. Przykładowo, dla *3-opt* jednym z sąsiadów rozwiązania

$$\begin{bmatrix} 3 & 7 & 4 & 8 & 2 & 1 & 5 & 6 \end{bmatrix}$$

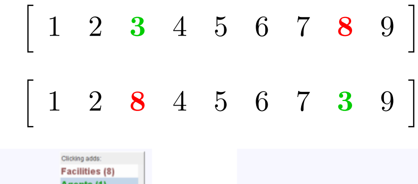
będzie rozwiązanie

$$\begin{bmatrix} 3 & 2 & 4 & 7 & 8 & 1 & 5 & 6 \end{bmatrix}$$

(zmieniliśmy kolejność występowania trzech zaznaczonych elementów).

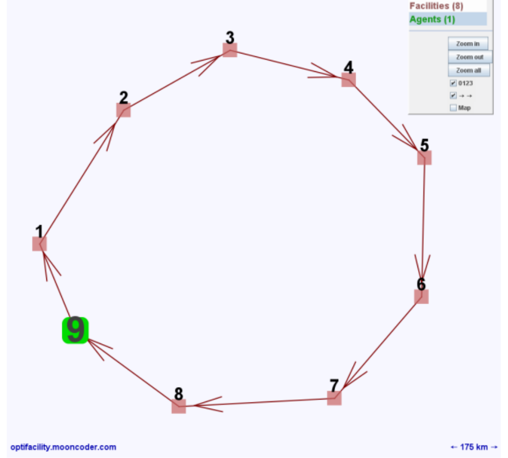
Zastosujemy dla naszego problemu sąsiedztwo **2-opt** (zamiana dwóch elementów miejscami). Jako odległość między dwoma rozwiązaniami przyjmiemy ilość 2-zamian które trzeba zastosować aby przejść między nimi. Ponadto, przyjmiemy $\epsilon = 1$ a więc w sąsiedztwie naszego rozwiązania tymczasowego bedziemy mieli jedynie takie rozwiązania które różnią się elementami na dwóch pozycjach.

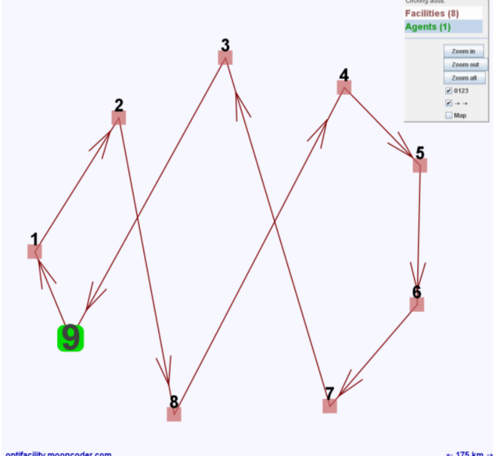
W oczywisty sposób tak zdefiniowane sąsiedztwo spełnia wymagania na rozmiar sąsiedztwa i brak odizolowanych rozwiązań. Jak wygląda jednak kwestia podobieństwa bliskich sobie rozwiązań? Spójrzmy na przykład poniżej:



$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 8 & 4 & 5 & 6 & 7 & 3 & 9 \end{bmatrix}$$



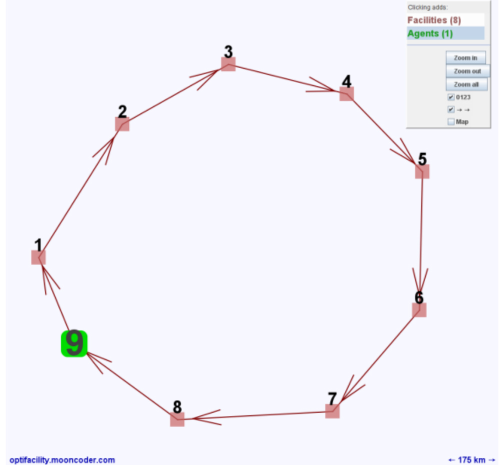


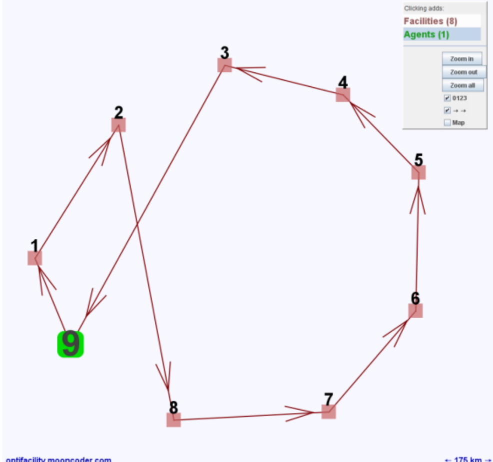
Jak widać jest całkiem dobrze - zmieniła się długość jedynie czterech fragmentów trasy.

Okazuje się jednak, iż dla symetrycznego problemu komiwojażera da się stworzyć lepsze sąsiedztwo (tj. takie gdzie rozwiązania sąsiednie są jeszcze bardziej do siebie podobne)! Jako że w symetrycznej wersji problemu komiwojażera kierunek odwiedzania miast nie ma wpływu na jakość funkcji celu (ponieważ odległość z miasta A do miasta B jest zawsze równa odległości z miasta B do miasta A) możemy zmodyfikować sąsiedztwo *2-opt* i zamiast zamieniać dwa miasta miejscami na trasie, możemy odwracać cały przebieg trasy pomiędzy nimi! Jak widać na poniższym przykładzie, skutkuje to zamianą jedynie dwóch odcinków (w kontraście do klasycznego *2-opt* gdzie musieliśmy zamieniać aż 4 odcinki).

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 8 & 7 & 6 & 5 & 4 & 3 & 9 \end{bmatrix}$$

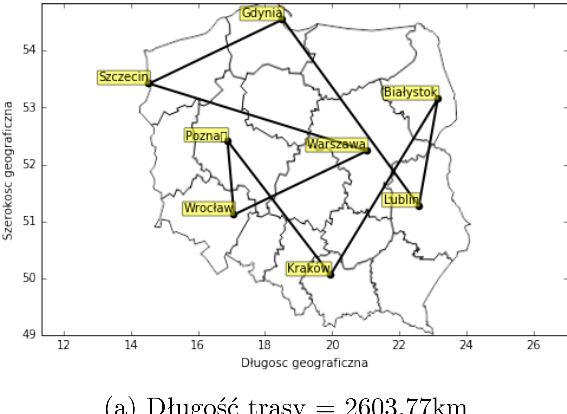




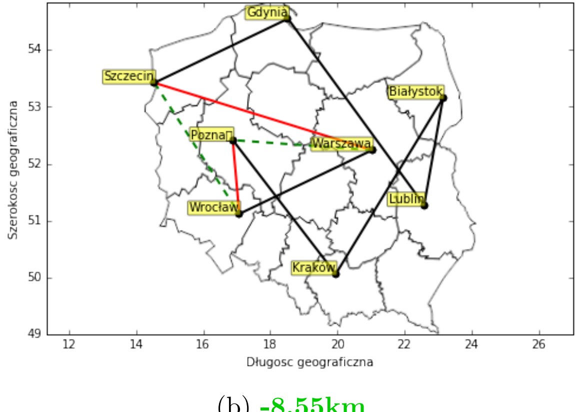
W przypadku symetrycznego problemu komiwojażera bedziemy chcieli korzystać ze zmodyfikowanego sąsiedztwa *2-opt* – sąsiedzi będą tu bardziej podobni do aktualnego rozwiązania, a więc krajobraz dostosowania będzie bardziej gładki, a więc będzie w nim mniej optimów lokalnych, a więc mamy większe szanse na znalezienie rozwiązania optymalnego (optimum globalne)! Jednocześnie jedak zauważ, iż w przypadku asymetrycznej wersji TSP nasz zmodyfikowany *2-opt* będzie działał bardzo źle (czy potrafisz uzasadnić dlaczego?).

6.4.1 Wizualizacja procesu optymalizacji problemu komiwojażera

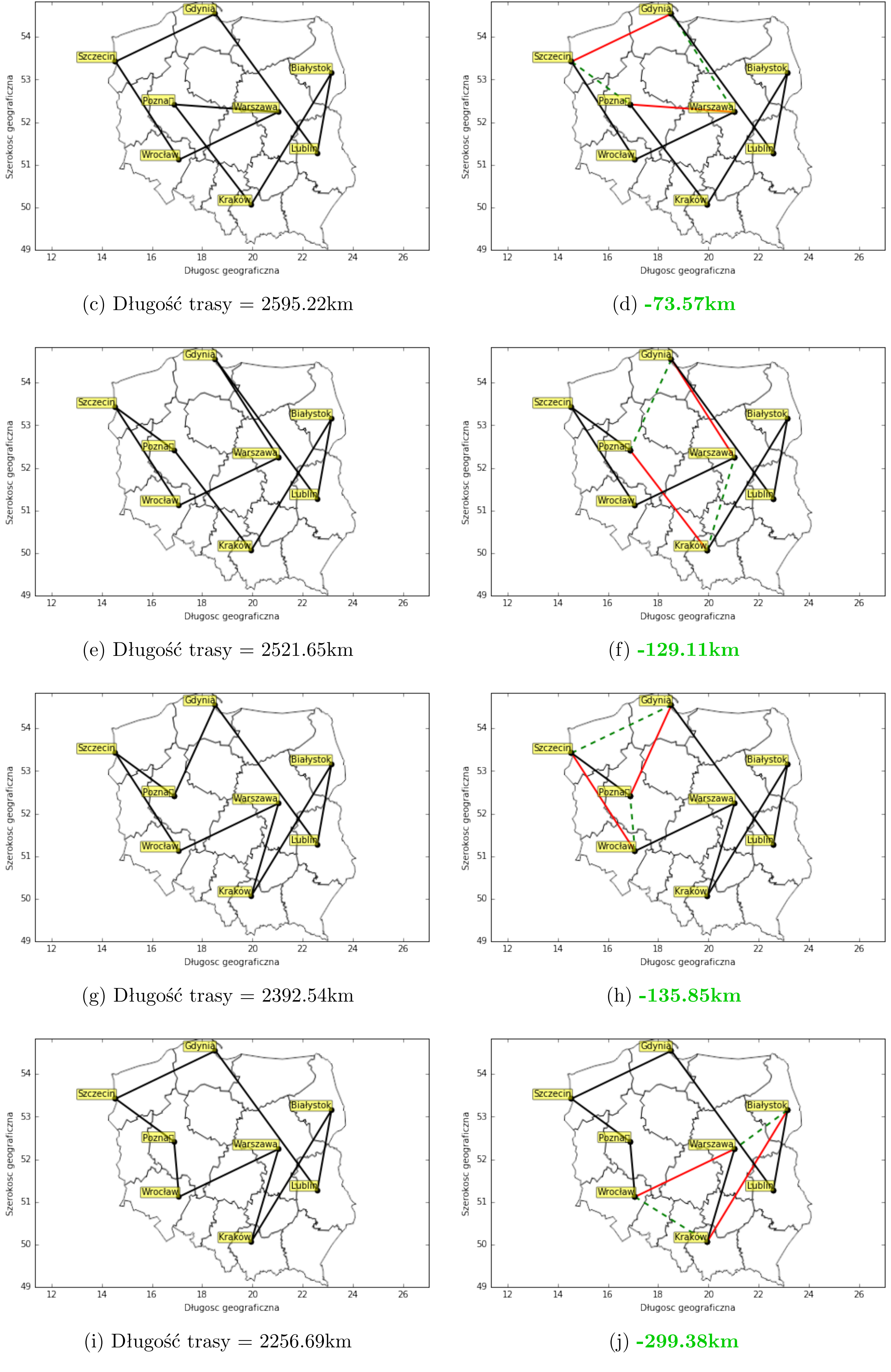
A więc mamy już gotowe wszystko co będzie nam potrzebne aby zminimalizować długość trasy przechodzącej przez polskie miasta. Poniżej, na Rysunku 7 zwizualizowano proces optymalizacji pewnej losowej trasy początkowej, przy użyciu permutacyjnej reprezentacji rozwiązania, zmodyfikowanego sąsiedztwa *2-opt* oraz metody *greedy*. Czerwone odcinki reprezentują odcinki które są w danym kroku usuwane z trasy, a zielone przerywane odcinki – te które są dodawane. Czarne odcinki reprezentują część trasy, która nie podlega zmianie.



(a) Długość trasy = 2603.77km



(b) **-8.55km**



(c) Długość trasy = 2595.22km

(d) **-73.57km**

(e) Długość trasy = 2521.65km

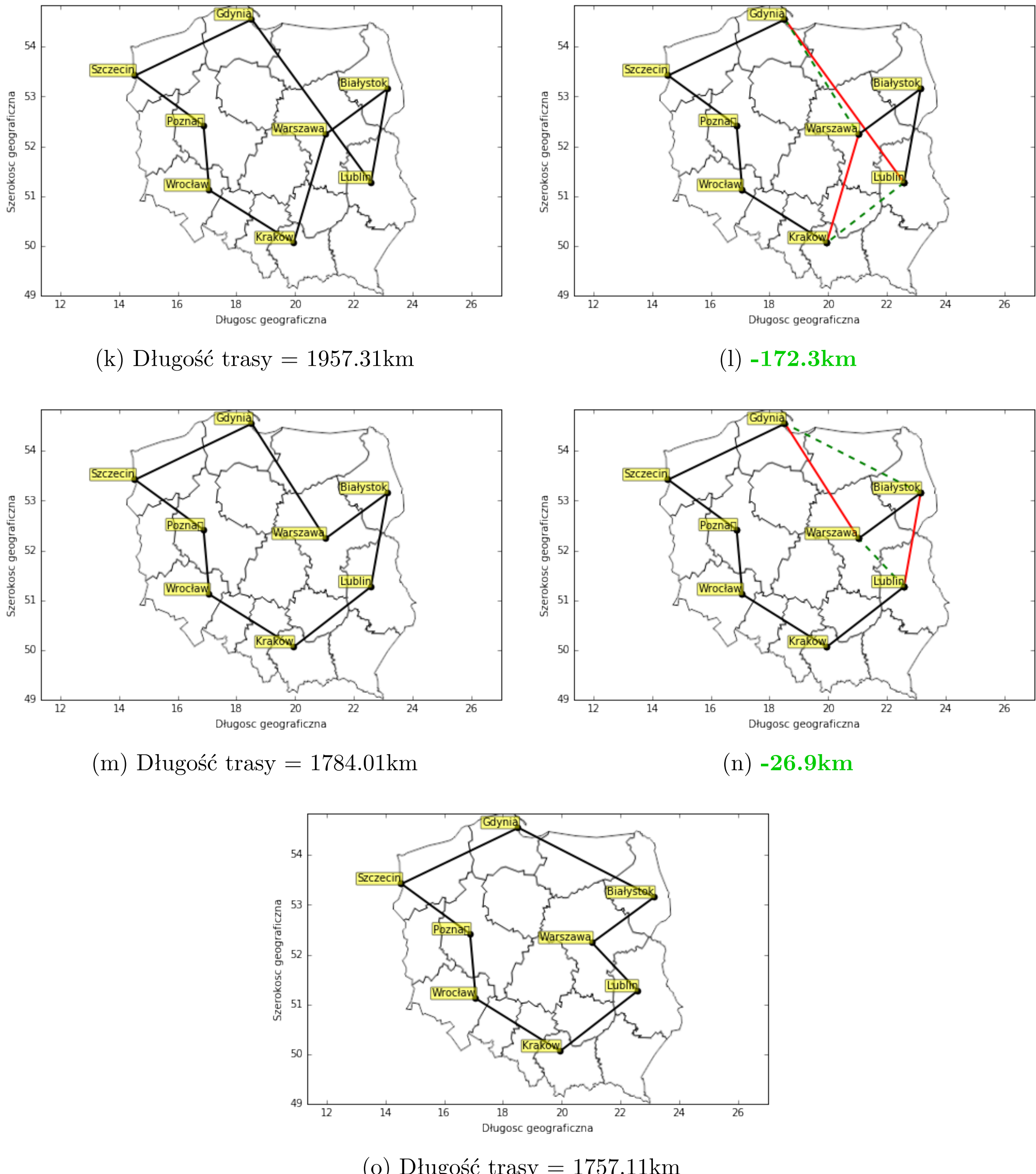
(f) **-129.11km**

(g) Długość trasy = 2392.54km

(h) **-135.85km**

(i) Długość trasy = 2256.69km

(j) **-299.38km**



(k) Długość trasy = 1957.31km

(l) **-172.3km**

(m) Długość trasy = 1784.01km

(n) **-26.9km**

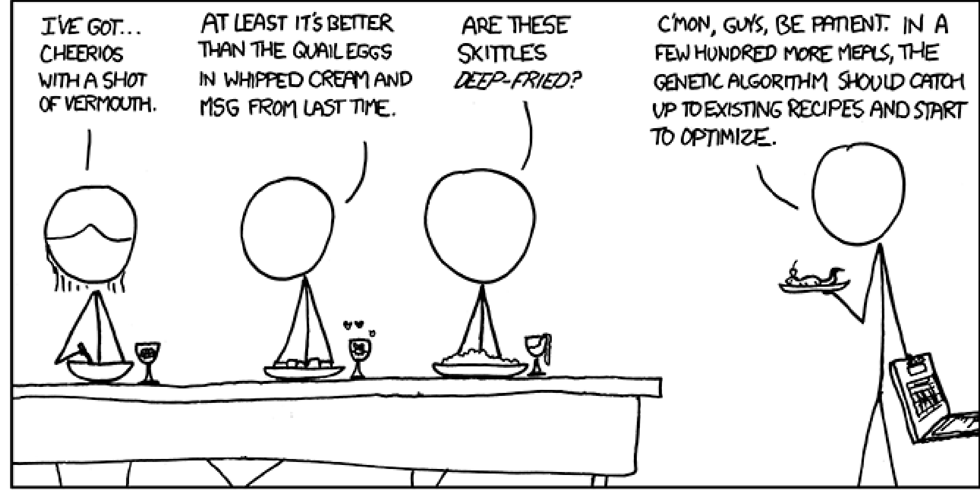
(o) Długość trasy = 1757.11km

Rysunek 7: Proces optymalizacji długości trasy w problemie komiwojażera przy użyciu przeszukiwania lokalnego.

7 Algorytmy ewolucyjne (WORK IN PROGRESS)

Chociaż LS jest w stanie znaleźć w bardzo szybkim czasie rozwiązania optymalne lokalnie (optima lokalne), dla trudniejszych problemów optymalizacji jakość takich rozwiązań może nie być zadowalająca. Możemy wyróżnić dwie istotne cechy (wady?) metody LS które są odpowiedzialne za ten stan rzeczy.

Po pierwsze, metoda ta działa lokalnie, co oznacza, iż fragment przestrzeni rozwiązań, który będzie przeszukany jest niezwykle mały w porównaniu do rozmiaru całej tej przestrzeni (w dowolnym momencie nasze aktualne rozwiązanie jest tylko kroplą w całym oceanie możliwości!). W praktyce skutkuje to tym, iż znalezione rozwiązanie jest bardzo silnie zależne od obranego punktu startowego. W zależności od charakterystyki problemu, procent rozwiązań bedących optimami lokalnymi może się różnić, jednak gdy liczba wszystkich możliwych rozwiązań jest



https://xkcd.com/720/

bardzo duża, nawet niewielki jej ułamek bedzie mógł być bardzo dużą wartością. Oznacza to, że godząc się na zbieżność do najbliższego optimum lokalnego co prawda zmiejszamy rozmiar przestrzeni rozwiązań (a ich średnia jakość bedzie wyższa niż średnia jakość rozwiązań w oryginalnej przestrzeni), jednak w tej nowej przestrzeni wybieramy po prostu losowe rozwiązanie (lub – w przypadku wielokrotnych powtórzeń LS z różnych pozycji startowych – używamy random search'a).

Po drugie, LS nie pozwala nam nigdy na podjęcie akcji które pogorszyłyby jakość naszego aktualnego rozwiązania. Oznacza to, iż optimum lokalne działa jak swego rodzaju pułapka – w momencie gdy je znajdziemy algorytm kończy się, ponieważ nie jesteśmy już w stanie dokonać żadnego innego ruchu.

Algorytmy ewolucyjne (evolutionary algorithms, EA) są metaheurystyką, która próbuje „wyleczyć" obie w wymienionych powyżej „chorób" metody LS. Zamiast skupiać się w dowolnym momencie tylko na jednym z rozwiązań, EA będzie przyglądać się jednocześnie dużej grupie różnych rozwiązań. Ponadto, zamiast przemieszczać się w przestrzeni rozwiązań w systematyczny i deterministyczny sposób jak ma to miejsce w LS, EA będzie przeglądać przestrzeń rozwiązań w sposób znacznie bardziej chaotyczny, co powinno pozwolić mu uniknąć zatrzymania się w większości optimów lokalnych.

7.1 Porównanie z ewolucją biologiczną

Sama nazwa algorytmu ewolucyjnego sugeruje jakiś związek z ewolucją biologiczną. Należy jednak pamiętać, iż ewolucja biologiczna służyła tutaj zaledwie jako inspiracja do stworzenia metody EA, a więc nie wszystkie elementy występujące w biologii musza mieć swoje odzwierciedlenie w algorytmie i vice versa. Algorytm ewolucyjny **NIE** jest symulacją ewolucji biologicznej, a jedynie metaheurystyczną metodą optymalizacji!

Pomimo powyższych zastrzeżeń, skorzystanie z metafory biologicznej może być pomocne dla celu lepszego zrozumienia działania algorytmu EA (zarówno *jak* on działa, jak również *dlaczego* on działa). Poniżej zamieszczona jest tabelka zestawiająca istotne elementy ewolucji biologicznej z ich algorytmicznymi odpowiednikami.

	Ewolucja (biologia)	Algorytm ewolucyjny
osobnik	Pojedynczy unikalny organizm.	Pojedyncze rozwiązanie dopuszczalne.

jak jest opisany?	Genotyp w postaci kodu DNA złożonego z zasad azotowych nukleotydów (litery A, C, T i G).	Genotyp zależny od problemu optymalizacji (np. permutacja miast [1 6 2 5 4 3] dla problemu TSP lub wektor binarny 00110101 dla problemu plecakowego).
co jest opisane?	Struktura ciała i mózgu organizmu, jego predyspozycje oraz pewne wrodzone instynkty (jego zachowanie)	Konkretne rozwiązanie problemu (np. konkretna trasa w problemie TSP) oraz jego jakość (wartość fitnessu)
grupa	Populacja podobnych, jednak różnorodnych organizmów które mogą wchodzić ze sobą w interakcje i się rozmnażać (np. stado antylop).	Populacja różnych rozwiązań (zazwyczaj o pewnym stałym, ustalonym rozmiarze) pokrywająca pewien obszar przestrzeni rozwiązań.
rozmnażanie	W zależności od gatunku, płciowe lub bezpłciowe: stworzenie nowego, niezależnego organizmu podobnego do organizmów rodzicielskich.	Stworzenie nowego rozwiązania w oparciu o inne rozwiązanie (rozwiązania).
mutacja	Losowa zmiana wprowadzana do genotypu w ramach procesu rozmnażania.	Operator mutacji wprowadza losową, drobną modyfikację do istniejącego już rozwiązania. Może być traktowany jak zastąpienie rozwiązania innym, losowym rozwiązaniem z jego sąsiedztwa (zdefiniowanego tak samo jak w przypadku metody LS). Dla problemu TSP może być to zamiana pozycji dwóch miast w permutacji (np. [12345] → [14325]).
krzyżowanie	Połączenie genotypów rodziców poprzez losowy wybór dla każdego z genów rodzica od którego gen ten będzie wzięty. W uproszczeniu: dla osobników **ATTGGCAC** oraz **CTCGGAAT**, przykładowym efektem krzyżowania może być osobnik **ATTGGAAT**.	Operator krzyżowania ma na celu stworzenie nowego rozwiązania na podstawie dwóch (lub więcej) rozwiązań rodzicielskich. Choć implementacja tego operatora będzie się różniła w zależności od problemu, zawsze powinien on spełniać następujące wymagania: a) efekt krzyżowania (dziecko) powinno być poprawnym rozwiązaniem dopuszczalnym, b) dziecko powinno łączyć w sobie cechy obu rozwiązań rodzicielskich (np. w przypadku problemu TSP: część miast jest odwiedzana w kolejności zgodnej z jednym rodzicem, a część w kolejności zgodnej z drugim rodzicem).

presja selekcyjna	Presja selekcyjna wynika z wielu czynników, takich jak zdolność przetrwania i zdobywania pożywienia, ogólne zdrowie i atrakcyjność w oczach osobników płci przeciwnej, troska o przetrwanie potomstwa itd. Możemy powiedzieć, iż środowisko w którym żyją osobniki wywiera na nie presję, która sprawia, iż niektóre osobniki bedą z większym sukcesem niż inne propagować dalej swój materiał genetyczny. Własności populacji osobników bedą się więc zmieniać w taki sposób, aby jak najlepiej dostosować się do warunków otoczenia.	Presja selekcyjna ustalana jest przez operator selekcji, który wybiera z populacji rozwiązania do rozmnażania na podstawie ich jakości (fitnessu). Lepsze rozwiazania powinny mieć większą szansę na zostanie wybranym do rozmnażania niż rozwiązania o niskiej jakości, jednak operator selekcji powinien być stochastyczny, t.j. pozwalać z pewnym prawdopodobieństwem na wybór każdego (nawet słabego) rozwiązania. Istnieją różne wersje operatora selekcji, np. selekcja ruletkowa (proporcjonalna) czy też selekcja turniejowa.

7.2 Schemat algorytmu ewolucyjnego

Możliwych jest wiele implementacji algorytmów ewolucyjnych różniących się szczegółami, jednak jednocześnie zachowujących wszystkie wymienione wyżej elementy i własności. Opiszemy dokładniej tylko dwie z nich – podejście generacyjne (będące de facto najbardziej klasyczną wersją EA) oraz tzw. steady-state.

7.2.1 Podejście pokoleniowe

Podejście **pokoleniowe** można opisać za pomocą poniższego schematu:

1. Stwórz pustą populację startową P o rozmiarze N.
2. Wypełnij populację P rozwiązaniami losowymi.
3. Dopóki nie zostanie spełniony warunek końcowy (określona liczba iteracji, czas obliczeń bądź jakość najlepszego znalezionego rozwiązania) powtarzaj:
   (a) Każdemu rozwiązaniu z P przypisz jego jakość (fitness).
   (b) Stwórz pustą populację P'.
   (c) Dopóki populacja P' ma mniej niż N elementów, powtarzaj:
      i. Losowo zdecyduj którego operatora genetycznego (mutacją bądź krzyżowanie) użyjesz do stworzenia nowego rozwiązania.
      ii. Korzystając z operatora selekcji, losowo wybierz rozwiązanie(a)-rodzica(ów).
      iii. Użyj wybranego uprzednio operatora genetycznego na rodzicach aby stworzyć rozwiazanie-dziecko.
      iv. Dodaj nowo utworzone rozwiązanie do populacji P'.
   (d) Ustaw $P = P'$.

7.2.2 Steady-state

Podejście **steady-state** można opisać za pomocą poniższego schematu:

1. Stwórz pustą populację startową P o rozmiarze N.

2. Wypełnij populację P rozwiązaniami losowymi.
3. Każdemu rozwiązaniu z P przypisz jego jakość (fitness).
4. Dopóki nie zostanie spełniony warunek końcowy (określona liczba iteracji, czas obliczeń bądź jakość najlepszego znalezionego rozwiązania) powtarzaj:
   (a) Korzystając z operatora selekcji negatywnej (np. losowego) wybierz i usuń z populacji jedno z rozwiązań.
   (b) Losowo zdecyduj którego operatora genetycznego (mutacją bądź krzyżowanie) użyjesz do stworzenia nowego rozwiązania.
   (c) Korzystając z operatora selekcji (tym razem pozytywnej), losowo wybierz rozwiązanie(a)-rodzica(ów).
   (d) Użyj wybranego uprzednio operatora genetycznego na rodzicach aby stworzyć rozwiazanie-dziecko.
   (e) Nowo utworzonemu rozwiązaniu przypisz jego jakość (fitness).
   (f) Dodaj nowo utworzone rozwiązanie do populacji P.

Jak widzisz, w podejściu steady-state nigdy nie tworzymy od nowa całej populacji, lecz raczej w każdej jego iteracji usuwamy i dodajemy tylko jedno rozwiązanie. Oznacza to, iż informacja o nowo przebadanym punkcie przestrzeni rozwiązań może być użyta przez nasz algorytm tak szybko jak to możliwe (tj. już w kolejnej jego iteracji). W praktyce skutkuje to często tym, iż by odnaleźć optimum globalne EA w odmianie steady-state potrzebuje sprawdzić mniejszą liczbę rozwiązań niż EA w odmianie pokoleniowej.

7.3 Eksploracja i eksploatacja

Dwoma bardzo istotnymi hasłami związanymi z algorytmami ewolucyjnymi jest eksploracja oraz eksploatacja.

O eksploracji możemy mówić w momencie gdy próbujemy testować nowe obszary w przestrzeni rozwiazań. W algorytmach ewolucyjnych za eksplorację odpowiedzialne są operatory genetyczne (sekcja 7.4) pozwalające tworzyć nowe rozwiązania. Zauważ, iż same operatory genetycze są „ślepe” – tworząc nowe rozwiązanie nie wiedzą czy będzie ono lepsze, gorsze czy też równie dobre co rozwiązania-rodzice! Oznacza to, iż operatory genetyczne służą do próbkowania przestrzeni rozwiązań w celu znalezienia czegoś (jakiegoś rozwiązania) które może okazać się przydatne w przyszłości (być wysokiej jakości).

Niestety, eksploatacja bez eksploracji nie będzie w stanie odkryć żadnych ponadprzeciętnie dobrych rozwiązań, ponieważ nie zważa ona na jakość tworzonych rozwiązań. O eksploracji możemy mówić w przypadku gdy skupiamy się na poszukiwaniu rozwiązania optymalnego w obiecujących okolicach przestrzeni rozwiązań, tj. w sąsiedztwie znanych nam już rozwiązań o wysokiej wartości fitnessu. Do tego celu służyć nam będzie w EA operator selekcji, omówiony dokładniej w sekcji 7.5. Przykładem metaheurystyki, która skupia się jedynie na aspekcie eksploatacji (jednocześnie radośnie ignorując aspekt eksploracji) jest znany Ci już algorytm LS.

Aby skutecznie znaleźć dobre rozwiązanie musimy utrzymywać balans pomiędzy aspektami eksploracji i eksploatacji. Wyobraź sobie, iż stoisz w lodziarni przed ladą z lodami o różnych, nieznanych ci smakach i próbujesz znajeźć taka kombinację, która będzie dla Ciebie jak najsmaczniejsza. Zaczniesz od sprawdzenia losowych smaków, jednak jeśli przy kolejnych wizytach bedziesz zamawiać jedynie te które najbardziej Ci do tej pory smakowały, możesz nigdy nie znaleźć tych które w rzeczywistości były by dla ciebie najlepsze. Z drugiej strony, nie byłoby rozsądnym zawsze zamiawiać jakieś nowe, nieznane ci smaki, ponieważ istnieje ryzyko, że większość z nich by ci nie pasowała. Najlepszą strategią byłoby więc zamawiać kombinacje uwględniając te smaki o których już wiesz, że ci odpowiadają, jednocześnie jednak od czasu do czasu w ramach eksperymentów testując nowe, nieznane smaki. Innymi słowy, najlepszą strategią byłoby zbalansowanie aspektów eksploracji i ekploatacji.

7.4 Rola operatorów genetycznych

Aby zrozumieć rolę mutacji i krzyżowania, warto przeanalizować działanie algorytmu ewolucyjnego w którym brakuje jednego z tych elementów.

Jeśli zabraknie nam operatora mutacji (będziemy mieć tylko krzyżowanie), różnorodność genotypów w naszej populacji będzie bardzo szybko spadać, aż w końcu skończymy z populacją pełną klonów! Dlaczego? Zauważ, iż krzyżowanie nie wprowadza nowych rozwiązań do puli genowej, a jedynie łaczy te już istniejące. W każdym pokoleniu niektóre z fragmentów genotypów z poprzedniego pokolenia zostaną „zagubione”, tj. nie znajdą się one w żadnym z genotypów występujących w nowej populacji. Takie zagubione fragmenty genotypów są dla nas niestety nieodwracalnie utracone, a co gorsza w każdym pokoleniu będziemy tracić kolejne. Oznacza to, iż różnorodność genetyczna populacji bedzie bardzo szybko maleć (w szczególności jeśli ustalimy wysoką presję ewolucyjną), aż w końcu wszystkie osobniki w naszej populacji będą dokładnie takie same! Zwróć ponadto uwagę, że bez operatora mutacji nie jesteśmy w stanie znaleźć żadnego rozwiązania które nie jest kombinacją rozwiązań istniejących w populacji początkowej – oznacza to, że najprawdopodobniej nasz algorytm nie będzie nawet w stanie odkryć żadnego optimum lokalnego!

Przyjrzyjmy się teraz sytuacji alternatywnej – posiadamy operator mutacji, jednak brakuje nam operatora krzyżowania. Tym razem różnorodność genetyczna nie będzie aż tak istotnym problemem, ponieważ każda z mutacji może wprowadzić do naszej puli genowej zupełnie nowe „geny”. Będziemy również mogli odkryć rozwiązania w okolicy rozwiązań optymalnych lokalnie. Czy oznacza to, że EA nie potrzebuje operatora krzyżowania? Nie całkiem. Co prawda EA bez krzyżowania jest w stanie znajdować całkiem dobre rozwiazania, jednak w takim wypadku nie będziemy niestety wykorzystywać pełnego potencjału tej metaheurystyki. Wyobraź sobie następującą sytuację. Posiadamy w populacji dwa różne, dobre rozwiązania. Każde z nich jest dobre w inny sposób – odwołując się do przykładu TSP, być może jedno z nich zawiera „rozplątany” fragment trasy na północy Polski, podczas gdy drugie rozwiązanie zawiera „rozplątany” fragment trasy na południu. Gdybyśmy posiadali operator krzyżowania, ich dziecko mogłoby połączyć dobre cechy obu rodziców – w naszym wypadku cała trasa byłaby „rozplątana”. Niestety, bez operatora krzyżowania nie istnieje żaden sposób aby te dwa rozwiązania ze sobą połączyć. Oznacza to, że prędzej czy później jedno z tych rozwiązań wyginie, a drugie będzie musiało rozplątać drugą część trasy samemu, przy użyciu tysięcy mutacji. Przyznasz chyba, iż brzmi to bardzo nieefektywnie!

Podsumowujac: aby stworzyć skuteczny algorytm ewolucyjny potrzebujemy zarówno operatora mutacji jak i krzyżowania. Mutacja działa jak paliwo dorzucane do ognia ewolucji, pozwalające ewolucji działać dłuższy czas i znajdywać lepsze rozwiązania. Krzyżowanie pełni rolę wymiany wiedzy między różnymi rozwiązaniami, więc bez niego każde z rozwiązań musi się uczyć samemu (a wyobraź sobie jak wyglądałby dzisiejszy stan nauki gdyby ludzie przez ostatnie kilka tysięcy lat nie wymieniali się między sobą pomysłami i obserwacjami!). Ponadto, mutacja zwiększa różnorodność genetyczną populacji, podczas gdy krzyżowanie ją zmniejsza (tym niejmniej oba z nich podpadają pod aspekt eksploracji, nie eksploatacji).

7.5 Metody selekcji

Operator selekcji ma na celu w losowy sposób wybierać rozwiązania na podstawie których będą tworzone nowe rozwiązania, zapewniając jednocześnie, że rozwiązania o wyższej jakości będą miały wyższą szansę na zostanie wybranym. Istnieje wiele sposobów na zaimplementowanie tego operatora, jednak omówimy tutaj jedynie dwa z nich: selekcję ruletkową (proporcjonalną) i selekcję turniejową.

7.5.1 Selekcja ruletkowa

Selekcja ruletkowa przypisuje każdemu osobnikowi (rozwiązaniu) w populacji prawdopodobieństwo bycia wylosowanym proporcjonalne do jego wartości fitnessu. Poniżej znajduje się przykład wyznaczania prawdopodobieństw dla populacji zawierającej 5 osobników:

nr	*ranga*	*fitness*	*prawdopodobienstwo*
1	2	5.5	$5.5/22 = 25\%$
2	5	1.5	$1.5/22 \approx 7\%$
3	1	8	$8/22 \approx 36\%$
4	3	5	$5/22 \approx 23\%$
5	4	2	$2/22 \approx 9\%$
	suma	22	100%

W praktyce selekcja ruletkowa jest używana bardzo rzadko z kilku powodów.

Po pierwsze, jeśli jedno z rozwiązań jest znacznie lepsze od pozostałych, może ono szybko zdominować całą populację. Ryzyko zajścia tego zjawiska jest szczególnie wysokie na początku ewolucji, gdy większość rozwiązań jest jeszcze bardzo słaba. Oznacza to, że w przeciągu kilku pierwszych pokoleń różnorodność dostępnej nam puli genetycznej może zostać bardzo szybko zredukowana co jest z naszej perspektywy bardzo niepożądane.

Po drugie, w momencie gdy fitness wszystkich osobników w populacji jest równocześnie wysoki i bardzo podobny (np. 999, 1000, 995 i 1003), prawdopodobieństwo bycia wybranym dla każdego z nich bedzie niemal identyczne, co oznacza, iż presja selekcyjna w naszym algorytmie bedzie niemal niezauważalna. To z kolei będzie skutkować tym, że poprawa rozwiązań będzie następować bardzo powoli.

7.5.2 Selekcja turniejowa

Selekcja turniejowa różni się od selekcji turniejowej tym, że zamiast bazować prawdopodobieństwa wyboru osobników bezpośrednio na ich jakości, bedziemy je uzależniać od ich rangi w obecnej populacji. Jako rangę rozumiemy miejsce w „rankingu jakości” sporządzonego dla pełnej populacji, a więc najlepszy osobnik będzie miał rangę 1, drugi najlepszy rangę 2 itd. W ten sposób będziemy mogli uwolnić się od problemów opisanych powyżej dla selekcji ruletkowej (czy potrafisz wytłumaczyć dlaczego?).

W praktyce, selekcja turniejowa realizowana jest w następujacy sposób. Najpierw ustalamy wartosć parametru t, nazywanego rozmiarem turnieju. Im wyższa wartość t, tym wyższa presja selekcyjna w naszej populacji – innymi słowy tym bardziej będziemy preferowali rozwiązania dobre i tym bardziej będziemy dyskryminować rozwiązania o niskiej jakości (np. $t = 1$ oznacza brak presji selekcyjnej, $t = 3$ średnią presję, a $t = 20$ bardzo silną presję; w praktyce zależy to również w pewnym stopniu od wielkości naszej populacji, większe populacje wymagają większych rozmiarów turnieju). Następnie wybieramy z populacji w sposób **zupełnie losowy** t rozwiązań. Na koniec wybieramy do rozmnażania najlepsze z tych t rozwiązań. Dla $t = 2$ wykres zależności prawdopodobieństwa selekcji od rangi rozwiązania jest zależnością liniową, dla wyższych wartości t staje się on coraz bardziej wygięty w stronę punktu $(0, 0)$ (a więc rośnie prawdopodobieństwo wyboru osobników bardzo dobrych, spada prawdopodobieństwo wyboru pozostałych osobników).

8 Branch and Bound (WORK IN PROGRESS)

Tymczasowo odsyłamy do prezentacji pod adresem `http://www.cs.put.poznan.pl/mkomosinski/materialy/optymalizacja/BB_DP.pdf`