Sztuczna inteligencja i inżynieria wiedzy
Lista 1

6.03.2025/7.03.2025

1 Cel listy

Celem ćwiczenia jest praktyczne zapoznanie się z problemami optymalizacyjnymi oraz praktyczne przećwiczenie omawianych na wykładzie metod rozwiązywania pewnej podklasy tych problemów. Po wykonaniu listy, student powinien wiedzieć czym jest problem optymalizacyjny, jakie trudności mogą się wiązać z uzyskaniem dokładnego rozwiązania przedstawionego problemu oraz jak poradzić sobie z rozwiązaniem problemu przy ograniczonych zasobach (np. moce obliczeniowe, czas). W szczególności znane powinny być różnice między aproksymacją a heurystyką oraz przykłady podejścia heurystycznego do problemu przeszukania, oraz znajdowania ścieżek.

2 Wprowadzenie teoretyczne

Poniżej znajdują się informacje pomocne w wykonaniu listy zadań. Zakłada się, że po wykonaniu student opanował zagadnienia teoretyczne.

2.1 Definicje

Definicja 1 (Problem Optymalizacyjny)**.** *Problemem optymalizacyjnym nazywamy zbiór ograniczeń (zadanych w postaci nierówności lub równości na zmiennych decyzyjnych) wraz z funkcją celu.*
Niech S będzie zbiorem rozwiązań dopuszczalnych w założonym problemie, tzn. zbiór takich $\bar{x}$, które spełniają wszystkie ograniczenia, a $f(\bar{x}) : \mathbb{K}^n \to \mathbb{K}$ funkcją oceny jakości rozwiązania. Wtedy problemem minimalizacyjnym nazywamy znalezienie takiego $s^* \in S$, dla którego $s^* = \min_{s \in S} f(s)$ a $f(s)$ jest funkcją kosztu. Alternatywnie, problem nazywamy maksymalizacyjnym jeśli celem jest znalezienie takiego $s^* \in S$, dla którego $s^* = \max_{s \in S} f(s)$ a $f(s)$ jest funkcją użyteczności.

Łatwo zauważyć, że dowolny problem minimalizacyjny można zamienić na maksymalizacyjny.

Definicja 2 (Rozwiązanie optymalne)**.** *Niech S będzie zbiorem rozwiązań dopuszczalnych dla problemu (minimalizacyjnego) $\mathcal{P}$ o funkcji kosztu f. Rozwiązanie $s^\star \in S$ jest optimum globalnym* ***iff*** $\forall_{s \in S} f(s^\star) \leq f(s)$.

Definicja 3 (Aproksymacja)**.** *Niech* $\mathrm{Opt}(x) = \min_{s \in S(x)} f(s)$. *Dla algorytmu M, który rozwiązuje problem optymalizacyjny, tz. $M(x) \in S(x)$, mówimy, że jest ε–aproksymacją dla $\varepsilon \geq 0$, jeśli dla każdego x mamy*

$$\frac{|f(M(X)) - \mathrm{Opt}(x)|}{\max\{\mathrm{Opt}(x), f(M(x))\}} \leq \varepsilon \ .$$

Definicja 4 (Heurystyka)**.** *Niech P będzie problemem optymalizacyjnym, a S oznacza zbiór rozwiązań dla problemu P. Niech $f : S \to \mathbb{R}$ będzie funkcją celu oceniającą wartość dla danego rozwiązania. Niech s^* oznacza rozwiązanie optymalne.*

Heurystyka H dla problemu P to funkcja $H : S \to S$, która zwraca rozwiązanie $s' \in S$ dla problemu P, takie, że dla dostatecznie wielu przypadków zwraca rozwiązanie dość dobre, tzn. takie, że różnica $|s^* - s'|$ nie jest istotnie duża.
Heurystyka może wykorzystywać uproszczenia, przybliżenia lub reguły empiryczne w celu zmniejszenia złożoności obliczeniowej, lub zwiększenia prawdopodobieństwa znalezienia dobrego rozwiązania w krótkim czasie.

Z formalnej definicji wynika, że heurystyki nie są algorytmami dokładnymi, a jedynie metodami poszukiwania dobrych rozwiązań przybliżonych. Heurystyki mogą być stosowane samodzielnie lub w połączeniu z algorytmami dokładnymi w celu poprawy ich wydajności i jakości rozwiązań. Można zauważyć, że heurystyki mogą istnieć dla problemów nieaproksymowalnych.

Jednym z zastosowań podejścia heurystycznego jest rozwiązanie problem komiwojażera (Travelling Salesman Problem), którego celem jest znalezienie najkrótszej zamkniętej trasy łączącej wszystkie miasta z podanego zbioru V.

Definicja 5 (Problem komiwojażera (TSP))**.** *Niech $G = (V, E)$ będzie grafem, w którym V jest zbiorem wierzchołków (miast do odwiedzenia) o rozmiarze n, a E jest zbiorem krawędzi między parą miast. Każda krawędź $(u, v) \in E$ ma przypisaną nieujemną wagę $d(u, v)$ reprezentującą odległość między miastami u i v.*
Celem jest znalezienie najkrótszej ścieżki zamkniętej, która odwiedza każde miasto dokładnie raz, po czym wraca do miasta początkowego. Formalnie, należy znaleźć taką permutację $\pi = (v_1, v_2, \ldots, v_n)$ zbioru V, że suma wag krawędzi $D = d((\pi_1, \pi_2)) + d((\pi_2, \pi_3)) + \cdots + d((\pi_{n-1}, \pi_n)) + d((\pi_n, \pi_1))$ jest minimalna.

2.2 Algorytm przeszukiwania A*

Algorytm A* jest heurystycznym algorytmem służącym do znajdowania najkrótszej ścieżki w grafie i jest rozbudowaniem algorytmu Dijkstry o dodatkowy moduł estymacji kosztu ścieżki od celu. Algorytm ten przy nieprawidłowo zaprojektowanej heurystyce może osiągnąć wyniki dalekie od optymalnych. Algorytm

ten został po raz pierwszy zaproponowany przez Petera E. Harta, Nilsa Johna Nilssona i Bertrama Raphaela w 1968 r.

Algorytm A* oparty jest na optymalizacji funkcji kosztu, która jest zdefiniowana jako $f(curr, next) = g(curr, next) + h(next, end)$, gdzie $g(curr, next)$ oznacza funkcję kosztu przejścia do danego wierzchołka a $h(next, end)$ odpowiada za estymację kosztu z wierzchołka do celu.

Powszechnie stosowanymi estymacjami kosztu są:

- odległość Manhattan: $h(curr, next) = |curr.x - next.x| + |curr.y - next.y|$
- odległość Euklidesowa: $h(curr, next) = \sqrt{(curr.x - next.x)^2 + (curr.y - next.y)^2}$

Algorytm opiera się na działaniu dwóch list: otwartej - zawierającej listę węzłów odwiedzonych, ale których nie wszyscy sąsiedzi zostali odwiedzeni oraz listę zamkniętą - węzłów których wszyscy sąsiedzi zostali już sprawdzeni.

Algorithm 1 Algorytm A*

Input: poczatek, koniec
Output: Lista krawędzi tworzących znalezioną ścieżkę

```
1: poczatek.g ← 0
2: poczatek.h ← 0
3: poczatek.f ← poczatek.g + poczatek.h
4: otwarte ← list([poczatek])
5: zamkniete ← list()
6: while len(otwarte) > 0 do
7:    wezel ← null
8:    koszt_wezla ← +Inf
9:    for wezel_testowy in otwarte do
10:      if f(wezel_testowy) < koszt_wezla then
11:         wezel ← wezel_testowy
12:         koszt_wezla ← f(wezel_testowy)
13:   if wezel == koniec then
14:      Rozwiązanie znalezione
15:   otwarte ← (otwarte − wezel)
16:   zamkniete ← (zamkniete + wezel)
17:   for wezel_nastepny in sasiedztwo(wezel) do
18:      if wezel_nastepny not in otwarte and wezel_nastepny not in zamkniete
         then
19:         otwarte ← (otwarte + wezel_nastepny)
20:         wezel_nastepny.h = h(wezel_nastepny, koniec)
21:         wezel_nastepny.g = wezel.g + g(wezel, wezel_nastepny)
22:         wezel_nastepny.f = wezel_nastepny.g + wezel_nastepny.h
23:      else
24:         if wezel_nastepny.g > wezel.g + g(wezel, wezel_nastepny) then
25:            wezel_nastepny.g = wezel.g + g(wezel, wezel_nastepny)
26:            wezel_nastepny.f = wezel_nastepny.g + wezel_nastepny.h
27:            if wezel_nastepny in zamkniete then
28:               otarte ← (otwarte + wezel_nastepny)
29:               zamkniete ← (zamkniete − wezel_nastepny)
```

Algorytm Dijkstry Algorytm Dijkstry to algorytm znajdowania najkrótszych ścieżek w grafie ważonym (o nieujemnych wagach) z jednym źródłem. Algorytm działa poprzez utrzymywanie zbioru wierzchołków o najkrótszej odległości od źródła oraz aktualizację tych odległości wraz z dodawaniem kolejnych wierzchołków do zbioru.
Niech $G = (V, E)$ będzie grafem ważonym z jednym źródłem s, zbiorem wierzchołków V i zbiorem krawędzi E. Niech $w : E \to \mathbb{R}$ będzie funkcją wag krawędzi. Dla każdego wierzchołka $v \in V$, niech $d(v)$ będzie kosztem najkrótszej ścieżki z s do v, a $p(v)$ będzie wierzchołkiem poprzedzającym v na najkrótszej ścieżce z s do v.

Algorytm Dijkstry działa w następujący sposób:

1. Inicjalizuj $d(s) = 0$ oraz $d(v) = \infty$ dla każdego $v \in V \setminus \{s\}$.
2. Utwórz zbiór Q zawierający wszystkie wierzchołki grafu G.
3. Dopóki Q nie jest pusty, wykonuj:
   (a) Wybierz wierzchołek $u \in Q$ o najmniejszej wartości $d(u)$ i usuń go ze zbioru Q
   (b) Dla każdego v takiego, że $\exists_{(u,\,v)\in E}\ d(v) > d(u) + w(u,\,v)$ zaktualizuj $d(v) = d(u) + w(u,\,v)$ oraz $p(v) = u$
4. Zwróć d oraz p

Zbiór Q reprezentuje zbiór wierzchołków, które jeszcze nie zostały dodane do zbioru wierzchołków o najkrótszej odległości od źródła s. W kroku 3a wybierany jest wierzchołek z Q o najmniejszej wartości $d(u)$, co oznacza, że zostanie dodany do zbioru wierzchołków o najkrótszej odległości od źródła. Następnie, aktualizowane są odległości do sąsiadów wierzchołka u, jeśli $d(u) + w(u, v)$ jest mniejsze niż obecna wartość $d(v)$. W ten sposób, algorytm znajduje najkrótsze ścieżki z s do wszystkich innych wierzchołków w grafie.

2.3 Lokalne przeszukiwanie i Tabu Search

Przeszukiwanie lokalne Algorytm lokalnego przeszukiwania (local search) to metaheurystyka, która polega na iteracyjnym przeszukiwaniu sąsiedztwa aktualnego rozwiązania, z zamiarem znalezienia lepszego rozwiązania.

Niech P oznacza problem optymalizacyjny, a s jest aktualnym rozwiązaniem. Niech $N(s)$ będzie sąsiedztwem s, czyli zbiorem rozwiązań, które można osiągnąć przez zmianę ustalonej liczby parametrów (np. innej ewaluacji jednej ze zmiennych) aktualnego rozwiązania s. Niech $f(s)$ oznacza wartość funkcji celu dla rozwiązania s.

Algorytm rozpoczyna się od wygenerowania początkowego rozwiązania s_0. Następnie, w każdej iteracji, algorytm generuje nowe rozwiązanie $s' \in N(s)$. Jeśli $f(s') < f(s)$, to algorytm akceptuje nowe rozwiązanie s' i ustawia je jako aktualne rozwiązanie s. W przeciwnym przypadku odrzuca nowe rozwiązanie i kontynuuje przeszukiwanie sąsiedztwa. Przeszukanie kończy się po osiągnięciu kryterium stopu, które jest jednym z parametrów heurystyki (np. maksymalna liczna iteracji, maksymalna liczba iteracji bez aktualizacji s). Algorytm przeszukiwania lokalnego jest wrażliwy na utknięcie w lokalnym optimum funkcji celu.

Przeszukiwanie Tabu Algorytm przeszukiwania Tabu (Tabu search) to metaheurystyka, która polega na iteracyjnym przeszukiwaniu sąsiedztwa aktualnego rozwiązania, z uwzględnieniem zestawu ruchów, które są niedozwolone

(tabu). W porównaniu z przeszukiwaniem lokalnym wymaga on dodatkowej pamięci, jednak pozwala na wyeliminowanie powtórnego testowania tych samych (lub podobnych) rozwiązań, co pozwala na wydostanie się z lokalnego optimum.

Główne kroki przeszukiwania Tabu:

1. Generujemy rozwiązanie początkowe s_0.
2. Ustalamy aktualne rozwiązanie s na s_0. Ustaw zbiór tabu $T = \emptyset$ oraz najlepsze znane rozwiązanie $s^* = s$.
3. Określamy definicję sąsiedztwa $N(s)$.
4. Dopóki nie spełniliśmy warunku końca:
   (a) Generujemy sąsiedztwo aktualnego rozwiązania $N(s)$.
   (b) Przeszukujemy całe lub (deterministycznie) próbkujemy $N(s)$.
   (c) Wybieramy najlepszego z sąsiadów, takiego, że $s_i \in N(s) \setminus T$.
   (d) $T = T \cup N(s)$
   (e) Ustalamy $s = s_i$, jeśli $f(s) \leq f(s^*)$, to podstawiamy $s^* = s$.
5. Zwracamy rozwiązanie s^*.

Można zauważyć, że zależnie od przedstawionego problemu optymalizacyjnego P, algorytm przeszukiwania Tabu należy dostosować do warunków zadania. Przykładem może być czasochłonność przeszukiwania $N(s)$, które na przykład dla punktów w kuli o promieniu r jest nieskończone. W takim wypadku wykonywane jest próbkowanie (np. deterministyczne lub metodą Monte Carlo) rozwiązań leżących w sąsiedztwie s. Dodatkowo łatwo zauważyć, iż wraz z liczbą kroków zwiększa się tablica rozwiązań zakazanych T wymagając coraz większej pamięci. Podstawową metodą rozwiązania tego problemu jest ustalenie rozmiaru tablicy $t = |T|$ i traktowaniu jej jako kolejki FIFO, gdzie w przypadku przepełnienia usuwany jest najstarszy wpis. Wybór t powinien być zależny od charakterystyki $N(s)$, gdyż zbyt małe t dla dużych sąsiedztw powoduje, że łatwo utworzyć cykle długości większej niż t, powodując ponowne odwiedzanie tych samych rozwiązań, a więc problemy podobne do tych z przeszukiwania lokalnego. Natomiast zbyt duże t dla małych sąsiedztw powoduje zużycie nadmiarowej pamięci oraz brak usuwania rozwiązań z T. W celu dopuszczenia rozwiązań z T, które, ze względu na próbkowanie, trafiły do tablicy rozwiązań zakazanych bez sprawdzenia wartości funkcji użyteczności można stosować funkcję (lub tablicę) aspiracji A, która określa, że jeśli rozwiązanie jest dostatecznie dobre, to mimo obecności na T jest rozważane jako możliwe rozwiązanie.

Warto zauważyć, że nie zawsze musimy pamiętać listę kompletnych rozwiązań. Wykorzystując specyfikę problemu (lub przedstawiając zbiór rozwiązań jako n-wymiarową bryłę) możemy modyfikować T:

- zakazujemy wierzchołków – zamiast zakazywać całego rozwiązania, zabraniamy ustalonej wartości na jednej ze współrzędnych, np. w TSP zabraniamy wszystkich rozwiązań, które na pozycji i mają miasto j lub zabraniamy zmieniać pozycję miasta i.

- zakazujemy krawędzi – zabraniamy poruszania się wewnątrz ustalonej płaszczyzny (dwóch lub więcej współrzędnych), np. dla TSP w 2-opt zabraniamy usuwania jednej z krawędzi.

- zakazujemy krawędzi o jednym z wierzchołków – zabraniamy poruszania się wzdłuż ustalonej prostej (dla dwóch współrzędnych) lub wewnątrz ustalonej płaszczyzny (większa liczba współrzędnych), np. dla TSP zabronione są wszystkie krawędzie, które wychodzą z ustalonego wierzchołka i.

Poniżej znajduje się przykład pseudokodu wykorzystującego przeszukiwanie Tabu dla problemu komiwojażera.

Algorithm 2 Algorytm Knoxa: Przeszukiwanie Tabu dla problemu komiwojażera

```
1:  k ← 0
2:  losuj rozwiązanie początkowe s
3:  s* ← s
4:  T ← ∅
5:  while k < STEP_LIMIT do
6:    i ← 0
7:    while i < OP_LIMIT do
8:      określ N(s), A
9:      wybierz najlepszy s' ∈ (N(s) \ T) ∪ A
10:     zaktualizuj T
11:     if D(s') < D(s) then
12:       aktualizuj lokalne optimum s ← s'
13:     i++
14:   k++
15:   if D(s) < D(s*) then
16:     s* ← s
```

Algorytm Knoxa pozwala rozpatrzeć rozwiązanie sąsiednie, które znajduje się na liście Tabu, jeśli jest ono dostatecznie dobre. Wykorzystuje przy tym tablicę Aspiracji (A).

Przykładem tablicy aspiracji może być pamiętanie historii k ostatnich kroków algorytmu. W takim wypadku, poza tablicą tabu T, przechowujmy historię H opisującą liczbę modyfikacji zmiennej w h ostatnich krokach. Ustalmy $0 < \varepsilon < 1$ oraz $A_i = f(s_i) + \varepsilon(k - H(i))$. Jeśli $f(s) > A_i$, przechodzimy do s_i.

3 Materiały

1. Plik `connection_graph.csv` zawierający przetworzone z rozkładu jazdy komunikacji miejskiej we Wrocławiu dla dnia 1 marca 2023. Dane zostały pobrane wykorzystując informacje udostępniane przez urząd miasta Wrocławia (Otwarte Dane Wrocław) Zawiera on następujące kolumny:

(a) id - identyfikator krawędzi

(b) company - nazwa przewoźnika

(c) line - nazwa linii

(d) departure_time - czas odjazdu w formacie HH:MM:SS

(e) arrival_time - czas przyjazdu w formacie HH:MM:SS

(f) start_stop - przystanek początkowy

(g) end_stop - przystanek końcowy

(h) start_stop_lat - szerokość geograficzna przystanku początkowego w systemie WGS84

(i) start_stop_lon - długość geograficzna przystanku początkowego w systemie WGS84

(j) end_stop_lat - szerokość geograficzna przystanku końcowego w systemie WGS84

(k) end_stop_lon - długość geograficzna przystanku końcowego w systemie WGS84

4 Zadania

1. Wykorzystując udostępniony plik `connection_graph.csv` zaimplementuj algorytm wyszukiwania najkrótszych połączeń pomiędzy zadanymi przystankami `A` i `B`. Jako miarę odległości przyjmij, zależnie od decyzji użytkownika, czas dojazdu z `A` do `B` lub liczbę przesiadek koniecznych do wykonania.
   Program powinien przyjmować na wejściu wyłącznie 4 zmienne:

   (a) przystanek początkowy `A`

   (b) przystanek końcowy `B`

   (c) kryterium optymalizacyjne: wartość t oznacza minimalizację czasu dojazdu, wartość p oznacza minimalizację liczby zmian linii

   (d) czas pojawienia się na przystanku początkowym

   Program powinien zwracać na standardowym wyjściu harmonogram przejazdu, wypisując w kolejnych liniach informacje o kolejno wykorzystanych liniach komunikacyjnych (nazwa linii, czas i przystanek, na którym wsiadamy do danej linii komunikacyjnej oraz czas i przystanek, na którym kończymy korzystać z danej linii). Na standardowym wyjściu błędów powinien wypisywać wartość funkcji kosztu znalezionego rozwiązania oraz czas obliczeń liczony od wczytania danych do uzyskania rozwiązania.

Punktacja:

(a) algorytm wyszukiwania najkrótszej ścieżki z `A` do `B` za pomocą algorytmu algorytmem Dijkstry w oparciu o kryterium czasu (10 punktów)

(b) algorytm wyszukiwania najkrótszej ścieżki z `A` do `B` za pomocą algorytmu A* w oparciu o kryterium czasu (25 punktów)

(c) algorytm wyszukiwania najkrótszej ścieżki z `A` do `B` za pomocą algorytmu A* w oparciu o kryterium przesiadek (25 punktów)

(d) modyfikacja algorytmu A* z punktów (b) lub (c), który pozwoli na zmniejszenie wartości funkcji kosztu uzyskanego rozwiązania lub czasu obliczeń (10 punktów)

2. Wykorzystując udostępniony plik `connection_graph.csv` zaimplementuj algorytm, który dla przystanku początkowe `A` oraz listy przystanków `L`= $A_2, \ldots, A_n$ wyszuka najkrótszą trasę rozpoczynającą a `A`, przebiegającą przez wszystkie przystanki z `L` i wracającą do `A`. Jako funkcję kosztu trasy przyjmij, zależnie od decyzji użytkownika, łączny czas przejazdu lub liczbę przesiadek koniecznych do wykonania.
Program powinien przyjmować na wejściu 4 linie:

(a) przystanek początkowy `A`

(b) listę oddzielonych średnikiem przystanków do odwiedzenia

(c) kryterium optymalizacyjne: wartość t oznacza minimalizację czasu dojazdu, wartość p oznacza minimalizację liczby zmian linii

(d) czas pojawienia się na przystanku początkowym

Program powinien zwracać na standardowym wyjściu harmonogram przejazdu, wypisując w kolejnych liniach informacje o kolejno wykorzystanych liniach komunikacyjnych (nazwa linii, czas i przystanek, na którym wsiadamy do danej linii komunikacyjnej oraz czas i przystanek, na którym kończymy korzystać z danej linii). Na standardowym wyjściu błędów powinien wypisywać wartość funkcji kosztu znalezionego rozwiązania oraz czas obliczeń liczony od wczytania danych do uzyskania rozwiązania.

Punktacja:

(a) algorytm rozwiązujący problem odwiedzenia wierzchołków oparty na przeszukiwaniu Tabu bez ograniczenia na rozmiar tablicy T (10 punktów)

(b) modyfikacja (a) o dobór długości tablicy T w zależności od długości listy `L` w celu minimalizacji funkcji kosztu (5 punktów)

(c) modyfikacja (a) o aspirację w celu minimalizacji funkcji kosztu (5 punktów)

(d) rozszerzenie (a) poprzez dobór strategii próbkowania sąsiedztwa bieżącego rozwiązania, które pozwoli na minimalizację funkcji kosztu i skrócenie czasu działania algorytmu (10 punktów)

Do każdego z zadań przygotuj raport zawierający opis teoretyczny metody, przykładowe zastosowania, wprowadzone modyfikacje, materiały dodatkowe oraz krótkie opisy bibliotek wykorzystanych przy implementacji. W podsumowaniu raportu dodatkowo opisz napotkane problemy implementacyjne przy wykonywaniu zadania. Raport wyślij prowadzącemu przynajmniej na 24 godziny przed oddaniem listy.

5 Literatura

- Blog opisujący działanie A* z interaktywną animacją
- Prezentacja firmy Forbot
- Wpis na blogu nt. A* wraz z objaśnieniem poszczególnych kroków
- Film obrazujący działanie algorytmu A*
- Oryginalna praca opisująca działanie algorytmu A*
- E. Taillard – praca przeglądowa dotycząca przeszukiwania Tabu
- Knox – przeszukiwanie Tabu dla symetrycznego problemu komiwojażera
- Lai, Demirag, Leung – przeszukiwanie Tabu dla problemów routingu pojazdów
- Silvestrin, Ritt – przeszukiwanie Tabu dla problemów routingu pojazdów
- Soto et al. – przeszukiwanie Tabu dla problemów routingu pojazdów
- Lin, Bian, Liu – algorytm dla TSP łączący przeszukiwanie Tabu i symulowane wyżarzanie z dynamicznym doborem sąsiedztwa