

Projekt  
drogi dla



„Kierunki –  
gospodarki”

Tytuł: Algorytmy wykorzystywane w sztucznej  
inteligencji

Przedmiot: Algorytmy i struktury danych

Kierunek studiów: Matematyka

Autor: Marcin Kowalewski



Fundusze Europejskie  
dla Rozwoju Społecznego



Dofinansowane przez  
Unię Europejską

# Spis treści

|          |                                                                                      |          |
|----------|--------------------------------------------------------------------------------------|----------|
| <b>1</b> | <b>Wprowadzenie</b>                                                                  | <b>4</b> |
| 1.1      | Rola algorytmów w sztucznej inteligencji . . . . .                                   | 4        |
| 1.2      | Krótki rys historyczny . . . . .                                                     | 5        |
| <b>2</b> | <b>Algorytmy przeszukiwania przestrzeni stanów</b>                                   | <b>6</b> |
| 2.1      | Problem przestrzeni stanów . . . . .                                                 | 6        |
| 2.1.1    | Stan początkowy i stany docelowe . . . . .                                           | 7        |
| 2.1.2    | Operatory przejścia . . . . .                                                        | 7        |
| 2.2      | Przeszukiwanie wszerz (Breadth-First Search, BFS) . . . . .                          | 8        |
| 2.2.1    | Pseudokod algorytmu BFS . . . . .                                                    | 8        |
| 2.2.2    | Złożoność czasowa . . . . .                                                          | 9        |
| 2.2.3    | Złożoność pamięciowa . . . . .                                                       | 10       |
| 2.2.4    | Przykład . . . . .                                                                   | 10       |
| 2.3      | Przeszukiwanie w głąb (Depth-First Search, DFS) . . . . .                            | 11       |
| 2.3.1    | Idea algorytmu . . . . .                                                             | 11       |
| 2.3.2    | Implementacja algorytmu . . . . .                                                    | 11       |
| 2.3.3    | Pseudokod algorytmu DFS . . . . .                                                    | 12       |
| 2.3.4    | Złożoność czasowa . . . . .                                                          | 12       |
| 2.3.5    | Złożoność pamięciowa . . . . .                                                       | 13       |
| 2.3.6    | Przykład . . . . .                                                                   | 13       |
| 2.4      | Przeszukiwanie z ograniczeniem głębokości i iteracyjne pogłębianie (IDDFS) . . . . . | 14       |
| 2.4.1    | Przeszukiwanie z ograniczeniem głębokości (DLS) . . . . .                            | 14       |
| 2.4.2    | Idea iteracyjnego pogłębiania . . . . .                                              | 15       |
| 2.4.3    | Pseudokod algorytmu IDDFS . . . . .                                                  | 15       |

|       |                                                           |    |
|-------|-----------------------------------------------------------|----|
| 2.4.4 | Złożoność czasowa . . . . .                               | 16 |
| 2.4.5 | Złożoność pamięciowa . . . . .                            | 16 |
| 2.4.6 | Porównanie DLS, DFS i IDDFS . . . . .                     | 17 |
| 2.5   | Przeszukiwanie z kosztami (Uniform Cost Search) . . . . . | 18 |
| 2.5.1 | Idea algorytmu . . . . .                                  | 18 |
| 2.5.2 | Model kosztu . . . . .                                    | 18 |
| 2.5.3 | Pseudokod algorytmu UCS . . . . .                         | 19 |
| 2.5.4 | Złożoność czasowa . . . . .                               | 19 |
| 2.5.5 | Złożoność pamięciowa . . . . .                            | 20 |
| 2.5.6 | Własności algorytmu . . . . .                             | 20 |
| 2.5.7 | Porównanie UCS z BFS i DFS . . . . .                      | 20 |
| 2.6   | Heurystyki i algorytm A* . . . . .                        | 22 |
| 2.6.1 | Pojęcie heurystyki . . . . .                              | 22 |
| 2.6.2 | Własności heurystyk . . . . .                             | 22 |
| 2.6.3 | Idea algorytmu A* . . . . .                               | 23 |
| 2.6.4 | Pseudokod algorytmu A* . . . . .                          | 23 |
| 2.6.5 | Złożoność czasowa i pamięciowa . . . . .                  | 24 |
| 2.6.6 | Znaczenie algorytmu A* w sztucznej inteligencji . . . . . | 24 |

### **3 Algorytmy ewolucyjne 25**

|       |                                                      |    |
|-------|------------------------------------------------------|----|
| 3.1   | Inspiracja biologiczna . . . . .                     | 25 |
| 3.2   | Podstawowe pojęcia algorytmów ewolucyjnych . . . . . | 26 |
| 3.3   | Reprezentacja osobnika . . . . .                     | 26 |
| 3.3.1 | Funkcja przystosowania . . . . .                     | 26 |
| 3.3.2 | Populacja . . . . .                                  | 26 |
| 3.4   | Operatory ewolucyjne . . . . .                       | 27 |

|       |                                                                      |    |
|-------|----------------------------------------------------------------------|----|
| 3.4.1 | Selekcja . . . . .                                                   | 27 |
| 3.4.2 | Krzyżowanie . . . . .                                                | 27 |
| 3.4.3 | Mutacja . . . . .                                                    | 27 |
| 3.5   | Algorytm genetyczny . . . . .                                        | 28 |
| 3.5.1 | Pseudokod algorytmu genetycznego . . . . .                           | 28 |
| 3.5.2 | Własności algorytmów genetycznych . . . . .                          | 28 |
| 3.6   | Szczegółowy przykład algorytmu genetycznego dla problemu plecakowego | 29 |
| 3.6.1 | Sformułowanie problemu . . . . .                                     | 29 |
| 3.6.2 | Reprezentacja chromosomu . . . . .                                   | 30 |
| 3.6.3 | Funkcja przystosowania . . . . .                                     | 30 |
| 3.6.4 | Populacja początkowa . . . . .                                       | 30 |
| 3.6.5 | Selekcja (turniejowa) . . . . .                                      | 31 |
| 3.6.6 | Krzyżowanie jednopunktowe . . . . .                                  | 31 |
| 3.6.7 | Ocena potomstwa po krzyżowaniu . . . . .                             | 31 |
| 3.6.8 | Mutacja . . . . .                                                    | 32 |
| 3.6.9 | Nowa populacja i poprawa optimum . . . . .                           | 32 |

# 1 Wprowadzenie

## 1.1 Rola algorytmów w sztucznej inteligencji

Algorytmy stanowią fundament każdego systemu sztucznej inteligencji. To one określają, w jaki sposób dane są reprezentowane, przetwarzane oraz wykorzystywane do podejmowania decyzji. W literaturze przedmiotu algorytm AI rozumiany jest nie tylko jako procedura obliczeniowa, lecz jako kompletna metoda rozwiązywania problemu, obejmująca:

- reprezentację wiedzy lub danych,
- mechanizm wnioskowania lub uczenia,
- kryterium oceny jakości rozwiązania.

Hurbans [HR] zwraca uwagę, że algorytmy sztucznej inteligencji różnią się od klasycznych algorytmów informatycznych tym, iż często nie gwarantują rozwiązania optymalnego w sensie ścisłym, lecz dążą do rozwiązań *wystarczająco dobrych* w rozsądnym czasie obliczeń. Ma to szczególne znaczenie w problemach o bardzo dużej przestrzeni stanów, takich jak gry, planowanie tras czy analiza dużych zbiorów danych.

W przypadku algorytmów uczących się istotną rolę odgrywa również matematyczny aparat opisu modeli, funkcji kosztu oraz metod optymalizacji. Jak podkreślają Deisenroth, Faisal i Ong [DPFAOCS], zrozumienie matematycznych podstaw algorytmów uczenia maszynowego pozwala świadomie dobierać modele, analizować ich ograniczenia oraz interpretować uzyskane wyniki.

## 1.2 Krótki rys historyczny

Historia sztucznej inteligencji sięga lat 50. XX wieku, kiedy to pojawiły się pierwsze próby formalizacji ludzkiego rozumowania w postaci algorytmów. Za symboliczny początek tej dziedziny uznaje się konferencję w Dartmouth w 1956 roku, podczas której zaproponowano nazwę *artificial intelligence*.

W początkowym okresie dominowały podejścia symboliczne, oparte na logice matematycznej, systemach regułowych oraz przeszukiwaniu przestrzeni stanów.

Powstały wówczas klasyczne algorytmy przeszukiwania, takie jak BFS, DFS czy A\*, które do dziś stanowią ważny element kanonu sztucznej inteligencji.

Kolejne dekady przyniosły rozwój metod probabilistycznych oraz algorytmów uczących się na podstawie danych. Szczególnie istotny okazał się gwałtowny wzrost znaczenia uczenia maszynowego i sieci neuronowych, możliwy dzięki zwiększeniu mocy obliczeniowej komputerów oraz dostępności dużych zbiorów danych.

Współczesna sztuczna inteligencja łączy elementy klasycznych algorytmów z zaawansowanymi metodami matematycznymi i statystycznymi.

## 2 Algorytmy przeszukiwania przestrzeni stanów

### 2.1 Problem przestrzeni stanów

Jednym z fundamentalnych sposobów formułowania problemów w sztucznej inteligencji jest ich opis w postaci *przestrzeni stanów*. Takie podejście pozwala na jednolite ujęcie bardzo różnych zagadnień, takich jak rozwiązywanie łamigłówek, planowanie działań, nawigacja czy gry strategiczne.

**Stanem** nazywamy jednoznaczny opis sytuacji, w jakiej znajduje się rozważany system w danym momencie. Opis ten powinien zawierać wszystkie informacje niezbędne do podjęcia dalszych decyzji, przy czym pomija się elementy nieistotne z punktu widzenia rozwiązywanego problemu.

**Przestrzeń stanów** jest zbiorem wszystkich możliwych stanów, jakie mogą wystąpić w danym problemie, wraz z relacjami przejścia pomiędzy nimi. W praktyce przestrzeń ta bywa bardzo duża, a często nawet nieskończona, co stanowi jedno z głównych wyzwań algorytmów sztucznej inteligencji.

Przestrzeń stanów najczęściej modeluje się za pomocą **grafu**, w którym:

- wierzchołki odpowiadają stanom systemu,
- krawędzie odpowiadają możliwym przejściom pomiędzy stanami.

Graf taki może być skierowany lub nieskierowany, ważony lub nieważony, w zależności od charakteru problemu. Model grafowy umożliwia wykorzystanie algorytmów przeszukiwania do systematycznego badania przestrzeni rozwiązań.

### 2.1.1 Stan początkowy i stany docelowe

Każdy problem przeszukiwania definiuje co najmniej jeden **stan początkowy**, od którego rozpoczyna się eksploracja przestrzeni stanów. Jest to stan odpowiadający sytuacji wyjściowej problemu, na przykład początkowemu ułożeniu elementów łamigłówki lub aktualnej pozycji robota.

Z kolei **stany docelowe** (zwane również stanami końcowymi lub stanami celu) reprezentują sytuacje uznawane za rozwiązanie problemu. Zadaniem algorytmu przeszukiwania jest odnalezienie sekwencji przejść prowadzącej od stanu początkowego do jednego ze stanów docelowych, o ile taka sekwencja istnieje.

### 2.1.2 Operatory przejścia

Operatory przejścia opisują dozwolone działania, które mogą zostać wykonane w danym stanie i prowadzą do innego stanu. Formalnie operator przejścia jest funkcją, która dla zadanego stanu generuje jeden lub więcej stanów następnych.

Zbiór operatorów przejścia determinuje strukturę przestrzeni stanów oraz stopień jej rozgałęzienia. Zbyt ubogi zbiór operatorów może uniemożliwić osiągnięcie stanu docelowego, natomiast zbyt bogaty prowadzi do gwałtownego wzrostu liczby stanów do rozważenia. Dlatego projektowanie operatorów przejścia stanowi istotny element modelowania problemu w sztucznej inteligencji.

Sformułowanie problemu w postaci przestrzeni stanów jest pierwszym i kluczowym krokiem w projektowaniu algorytmu przeszukiwania. Dopiero po jego wykonaniu możliwe jest świadome dobranie odpowiedniej strategii eksploracji, takiej jak przeszukiwanie wszerz, w głąb lub z wykorzystaniem heurystyk.

## 2.2 Przeszukiwanie wszerek (Breadth-First Search, BFS)

Przeszukiwanie wszerek (ang. *Breadth-First Search*, BFS) jest jednym z najprostszych i jednocześnie najważniejszych algorytmów przeszukiwania przestrzeni stanów. Algorytm ten eksploruje przestrzeń stanów poziomami, badając najpierw wszystkie stany znajdujące się w tej samej odległości od stanu początkowego, a dopiero następnie przechodząc do stanów bardziej odległych.

### Idea algorytmu

Algorytm BFS rozpoczyna działanie w stanie początkowym i umieszcza go w strukturze danych typu kolejka. Następnie w kolejnych krokach:

- pobiera pierwszy stan z kolejki,
- sprawdza, czy jest on stanem docelowym,
- generuje wszystkie jego stany następce,
- dodaje nieodwiedzone stany do końca kolejki.

Dzięki wykorzystaniu kolejki algorytm BFS gwarantuje, że stany są przetwarzane w kolejności rosnącej długości ścieżki od stanu początkowego. W konsekwencji, jeżeli wszystkie przejścia mają jednakowy koszt, BFS znajduje rozwiązanie o minimalnej liczbie kroków.

### 2.2.1 Pseudokod algorytmu BFS

```
1 BFS(stan_początkowy):  
2   utwórz pusta kolejkę Q  
3   utwórz pusty zbiór odwiedzonych  
4  
5   dodaj stan_początkowy do Q  
6   dodaj stan_początkowy do odwiedzonych
```

```

7
8  while Q nie jest pusta:
9      s = usun pierwszy element z Q
10     if s jest stanem docelowym:
11         zwroc s jako rozwiazanie
12
13     for kazdy stan_nastepny generowany z s:
14         if stan_nastepny nie jest w odwiedzonych:
15             dodaj stan_nastepny do odwiedzonych
16             dodaj stan_nastepny do Q
17
18 zwroc brak_rozwiazania

```

Listing 1: Pseudokod algorytmu przeszukiwania w szerz (BFS)

### 2.2.2 Złożoność czasowa

Złożoność czasowa algorytmu BFS zależy od liczby stanów w przestrzeni stanów oraz liczby przejść pomiędzy nimi. W najgorszym przypadku algorytm musi odwiedzić wszystkie stany osiągalne ze stanu początkowego.

Jeżeli oznaczymy:

- $b$  – współczynnik rozgałęzienia (średnia liczba następców stanu),
- $d$  – głębokość najpłytszego stanu docelowego,

to złożoność czasową BFS można oszacować jako:

$$O(b^d).$$

Oznacza to, że czas działania algorytmu rośnie wykładniczo wraz z głębokością rozwiązania.

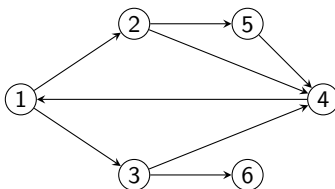
### 2.2.3 Złożoność pamięciowa

Algorytm BFS przechowuje w pamięci wszystkie stany znajdujące się na bieżącym poziomie przeszukiwania oraz ich następców. W efekcie jego złożoność pamięciowa jest również rzędu:

$$O(b^d).$$

W praktyce oznacza to, że BFS bardzo szybko zużywa pamięć dla problemów o dużej przestrzeni stanów, co stanowi jego główne ograniczenie.

### 2.2.4 Przykład



| Krok | Opis                                                                                                                                                  | Odwiedzone       | Kolejka |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------|
| 1    | Zaczynamy od wierzchołka 1, odwiedzamy go i wrzucamy do kolejki wszystkie jego następniki, w kolejności od tego z najmniejszym indeksem.              | 1                | 2, 3    |
| 2    | Bierzemy wierzchołek z kolejki, odwiedzamy go i znów wrzucamy wszystkie jego nieodwiedzone jeszcze następniki do kolejki.                             | 1, 2             | 3, 4, 5 |
| 3    | Bierzemy wierzchołek z kolejki, odwiedzamy go, jedynym następnikiem 3 jest 6, więc wrzucamy go do kolejki.                                            | 1, 2, 3          | 4, 5, 6 |
| 4    | Bierzemy wierzchołek z kolejki, odwiedzamy go, jedynym następnikiem 4 jest 1, ale ten wierzchołek już odwiedzaliśmy, więc nie wrzucamy go do kolejki. | 1, 2, 3, 4       | 5, 6    |
| 5    | Bierzemy wierzchołek z kolejki, odwiedzamy go, jedynym następnikiem 5 jest 4, ale ten wierzchołek już odwiedzaliśmy, więc nie wrzucamy go do kolejki. | 1, 2, 3, 4, 5    | 6       |
| 6    | Bierzemy wierzchołek z kolejki, odwiedzamy go, wierzchołek 6 nie ma następników, więc nie ma czego wrzucić do kolejki.                                | 1, 2, 3, 4, 5, 6 | -       |

## 2.3 Przeszukiwanie w głąb (Depth-First Search, DFS)

Przeszukiwanie w głąb (ang. *Depth-First Search*, DFS) jest podstawowym algorytmem przeszukiwania przestrzeni stanów, który eksploruje możliwe rozwiązania poprzez maksymalne zagłębianie się w jedną ścieżkę, zanim rozważy alternatywne możliwości. W przeciwieństwie do przeszukiwania wszerz, algorytm DFS nie bada przestrzeni stanów poziomami, lecz koncentruje się na pojedynczych ścieżkach.

### 2.3.1 Idea algorytmu

Algorytm DFS rozpoczyna działanie w stanie początkowym i przechodzi do jednego z jego stanów następnych. Następnie powtarza ten proces dla kolejnych stanów, dopóki:

- nie zostanie osiągnięty stan docelowy,
- nie wystąpi stan bez następców,
- lub nie zostanie spełniony warunek ograniczający głębokość przeszukiwania.

Po osiągnięciu stanu, z którego nie można już kontynuować eksploracji, algorytm cofa się do poprzedniego stanu i wybiera inną, niezbadaną jeszcze ścieżkę. Mechanizm ten określany jest mianem *backtrackingu*.

### 2.3.2 Implementacja algorytmu

Przeszukiwanie w głąb może być zaimplementowane na dwa równoważne sposoby:

- iteracyjnie, z wykorzystaniem stosu,
- rekurencyjnie, z wykorzystaniem stosu wywołań.

W obu przypadkach zachowana jest ta sama kolejność eksploracji przestrzeni stanów.

### 2.3.3 Pseudokod algorytmu DFS

```
1 DFS(stan):
2   if stan jest stanem docelowym:
3       zwroc stan jako rozwiazanie
4   oznacz stan jako odwiedzony
5   for kazdy stan_nastepny generowany z stan:
6       if stan_nastepny nie jest odwiedzony:
7           wynik = DFS(stan_nastepny)
8           if wynik != brak:
9               zwroc wynik
10  zwroc brak
```

Listing 2: Pseudokod algorytmu przeszukiwania w głąb (DFS)

### 2.3.4 Złożoność czasowa

Podobnie jak w przypadku BFS, w najgorszym przypadku algorytm DFS musi odwiedzić wszystkie stany osiągalne w przestrzeni stanów.

Przyjmując:

- $b$  – współczynnik rozgałęzienia,
- $m$  – maksymalną głębokość przeszukiwania,

złożoność czasową DFS można oszacować jako:

$$O(b^m).$$

W praktyce algorytm może działać bardzo szybko, jeśli rozwiązanie znajduje się głęboko w przestrzeni stanów, jednak w najgorszym przypadku może eksplorować znaczne fragmenty przestrzeni bez znalezienia rozwiązania.

### 2.3.5 Złożoność pamięciowa

Jedną z kluczowych zalet przeszukiwania w głąb są jego niewielkie wymagania pamięciowe. Algorytm DFS przechowuje jedynie aktualną ścieżkę od stanu początkowego do bieżącego stanu oraz informacje pomocnicze.

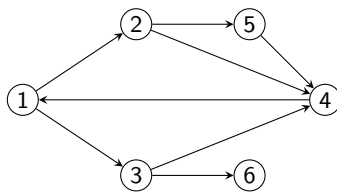
Złożoność pamięciowa DFS wynosi:

$$O(b \cdot m),$$

co w praktyce jest znacznie mniejsze niż w przypadku przeszukiwania wszerz.

Z tych względów DFS znajduje zastosowanie głównie tam, gdzie pamięć stanowi czynnik krytyczny lub gdy interesuje nas samo istnienie rozwiązania, a nie jego optymalność.

### 2.3.6 Przykład



| Krok | Opis                                                                                                                                               | Odwiedzone       | Stos    |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------|------------------|---------|
| 1    | Zaczynamy od wierzchołka 1, odwiedzamy go i wrzucamy na stos wszystkie jego następniki, w kolejności od tego z największym indeksem.               | 1                | 3, 2    |
| 2    | Bierzemy wierzchołek ze stosu, odwiedzamy go i znów wrzucamy wszystkie jego nieodwiedzone jeszcze następniki na stos.                              | 1, 2             | 3, 5, 4 |
| 3    | Bierzemy wierzchołek ze stosu, odwiedzamy go, jedynym następnikiem 4 jest 1, ale ten wierzchołek już odwiedzaliśmy więc nie wrzucamy nic na stos.  | 1, 2, 4          | 3, 5    |
| 4    | Bierzemy wierzchołek ze stosu, odwiedzamy go, jedynym następnikiem 5 jest 4, ale ten wierzchołek już odwiedzaliśmy więc nie wrzucamy nic na stos.  | 1, 2, 4, 5       | 3       |
| 5    | Bierzemy wierzchołek ze stosu, odwiedzamy go, jedynym następnikiem 3 jest 6, ten wierzchołek jeszcze nie jest odwiedzony więc wrzucamy go na stos. | 1, 2, 4, 5, 3    | 6       |
| 6    | Bierzemy wierzchołek ze stosu, odwiedzamy go, wierzchołek 6 nie ma następników, więc nie ma czego wrzucić na stos.                                 | 1, 2, 4, 5, 3, 6 | -       |

## 2.4 Przeszukiwanie z ograniczeniem głębokości i iteracyjne pogłębianie (IDDFS)

Przeszukiwanie w głąb, mimo niewielkich wymagań pamięciowych, posiada istotne ograniczenia, w szczególności brak kompletności w przestrzeniach nieskończonych oraz możliwość utknięcia w bardzo głębokich, nieproduktywnych gałęziach drzewa przeszukiwania. Jednym ze sposobów przezwyciężenia tych problemów jest wprowadzenie ograniczenia głębokości przeszukiwania.

### 2.4.1 Przeszukiwanie z ograniczeniem głębokości (DLS)

Przeszukiwanie z ograniczeniem głębokości (ang. *Depth-Limited Search*, DLS) jest modyfikacją algorytmu DFS, w której wprowadzony zostaje maksymalny dopuszczalny poziom zagłębienia. Algorytm przestaje rozwijać węzły, których głębokość przekracza ustalony limit.

Takie podejście:

- zapobiega nieskończonemu zagłębianiu się algorytmu,
- pozwala kontrolować czas działania,
- nie gwarantuje jednak odnalezienia rozwiązania, jeśli limit głębokości jest zbyt mały.

Kluczowym problemem DLS jest konieczność wcześniejszego dobrania odpowiedniego ograniczenia głębokości, co w praktyce często nie jest możliwe.

### 2.4.2 Idea iteracyjnego pogłębiania

Iteracyjne pogłębianie (ang. *Iterative Deepening Depth-First Search*, IDDFS) łączy zalety przeszukiwania wszerz i w głąb. Algorytm wykonuje kolejne przeszukiwania w głąb z rosnącym limitem głębokości:

$$0, 1, 2, \dots, d$$

gdzie  $d$  jest głębokością najpłytszego stanu docelowego.

### 2.4.3 Pseudokod algorytmu IDDFS

```
1 IDDFS(stan_początkowy):
2   for limit = 0, 1, 2, ...:
3       wynik = DLS(stan_początkowy, limit)
4       if wynik != brak:
5           zwroc wynik
6
7 DLS(stan, limit):
8   if stan jest stanem docelowym:
9       zwroc stan
10
11  if limit == 0:
12      zwroc brak
13
14  for każdy stan_następny generowany z stan:
15      wynik = DLS(stan_następny, limit - 1)
16      if wynik != brak:
17          zwroc wynik
18
19  zwroc brak
```

Listing 3: Pseudokod algorytmu iteracyjnego pogłębiania (IDDFS)

#### 2.4.4 Złożoność czasowa

Na pierwszy rzut oka algorytm IDDFS wydaje się nieefektywny, ponieważ wielokrotnie odwiedza te same stany. W praktyce jednak koszt ten jest relatywnie niewielki, gdyż większość węzłów znajduje się na najgłębszych poziomach drzewa przeszukiwania.

Złożoność czasową IDDFS można oszacować jako:

$$O(b^d),$$

gdzie:

- $b$  – współczynnik rozgałęzienia,
- $d$  – głębokość najpłytszego rozwiązania.

Jest to ta sama rząd wielkości co dla BFS, przy znacznie mniejszych wymaganiach pamięciowych.

#### 2.4.5 Złożoność pamięciowa

IDDFS, podobnie jak DFS, przechowuje w pamięci jedynie aktualną ścieżkę przeszukiwania. Złożoność pamięciowa wynosi:

$$O(b \cdot d),$$

co czyni ten algorytm szczególnie użytecznym w problemach o dużej przestrzeni stanów.

### 2.4.6 Porównanie DLS, DFS i IDDFS

Wprowadzenie iteracyjnego pogłębiania pozwala połączyć zalety wcześniej omówionych algorytmów:

- w porównaniu z DFS, IDDFS jest algorytmem kompletnym,
- w porównaniu z BFS, IDDFS zużywa znacznie mniej pamięci,
- w przeciwieństwie do DLS, IDDFS nie wymaga znajomości odpowiedniego limitu głębokości.

Z tych względów iteracyjne pogłębianie jest często stosowane w praktycznych systemach sztucznej inteligencji, zwłaszcza w problemach planowania i przeszukiwania o dużej przestrzeni stanów, gdzie pamięć stanowi istotne ograniczenie.

## 2.5 Przeszukiwanie z kosztami (Uniform Cost Search)

Przeszukiwanie wszerek zakłada, że wszystkie przejścia pomiędzy stanami mają jednakowy koszt. W wielu rzeczywistych problemach założenie to nie jest jednak spełnione. Przejścia mogą różnić się czasem trwania, zużyciem energii, ryzykiem lub innym kryterium kosztu. W takich przypadkach konieczne jest zastosowanie algorytmu, który uwzględnia zróżnicowane koszty przejść.

### 2.5.1 Idea algorytmu

Przeszukiwanie z kosztami (ang. *Uniform Cost Search*, UCS) jest uogólnieniem przeszukiwania wszerek, w którym kolejność eksploracji stanów nie zależy od ich głębokości, lecz od skumulowanego kosztu dotarcia do danego stanu.

Algorytm UCS:

- zawsze rozwija stan o najmniejszym dotychczasowym koszcie,
- wykorzystuje kolejkę priorytetową zamiast zwykłej kolejki,
- gwarantuje znalezienie rozwiązania o minimalnym koszcie całkowitym.

Jeżeli wszystkie koszty przejść są jednakowe, algorytm UCS zachowuje się identycznie jak BFS.

### 2.5.2 Model kosztu

Każdemu przejściu pomiędzy stanami przypisywany jest nieujemny koszt. Całkowity koszt ścieżki od stanu początkowego do danego stanu  $n$  oznaczany jest jako:

$$g(n) = \sum_{i=1}^k c_i,$$

gdzie  $c_i$  oznacza koszt  $i$ -tego przejścia na rozważanej ścieżce.

### 2.5.3 Pseudokod algorytmu UCS

```
1 UCS(stan_poczatkowy):
2   utworz pusta kolejke priorytetowa Q
3   utworz pusty zbior odwiedzonych
4
5   dodaj stan_poczatkowy do Q z priorytetem 0
6
7   while Q nie jest pusta:
8       (s, koszt) = usun element o najmniejszym priorytecie z Q
9
10      if s jest stanem docelowym:
11          zwroc s jako rozwiazanie
12
13      if s nie jest w odwiedzonych:
14          dodaj s do odwiedzonych
15
16      for kazdy stan_nastepny generowany z s:
17          nowy_koszt = koszt + koszt_przejscia(s, stan_nastepny)
18          dodaj stan_nastepny do Q z priorytetem nowy_koszt
19
20   zwroc brak_rozwiazania
```

Listing 4: Pseudokod algorytmu przeszukiwania z kosztami (UCS)

### 2.5.4 Złożoność czasowa

Złożoność czasowa algorytmu UCS zależy od liczby stanów oraz liczby możliwych przejść pomiędzy nimi. W najgorszym przypadku algorytm eksploruje wszystkie stany o koszcie nieprzekraczającym kosztu optymalnego rozwiązania.

Przyjmując:

- $b$  – współczynnik rozgałęzienia,
- $C^*$  – koszt optymalnego rozwiązania,

- $\epsilon$  – minimalny dodatni koszt przejścia,

złożoność czasową UCS można oszacować jako:

$$O\left(b^{\frac{C^*}{\epsilon}}\right).$$

### 2.5.5 Złożoność pamięciowa

Podobnie jak BFS, algorytm UCS przechowuje w pamięci dużą liczbę wygenerowanych stanów, co prowadzi do wysokich wymagań pamięciowych. Złożoność pamięciowa jest tego samego rzędu co złożoność czasowa.

### 2.5.6 Własności algorytmu

Algorytm przeszukiwania z kosztami posiada następujące własności:

- jest algorytmem kompletnym, o ile wszystkie koszty są nieujemne,
- gwarantuje znalezienie rozwiązania o minimalnym koszcie,
- nie wykorzystuje informacji heurystycznej o celu,
- może być bardzo kosztowny obliczeniowo dla dużych przestrzeni stanów.

### 2.5.7 Porównanie UCS z BFS i DFS

W porównaniu z BFS algorytm UCS:

- uwzględnia zróżnicowane koszty przejść,
- nie opiera się wyłącznie na głębokości przeszukiwania,
- zachowuje optymalność rozwiązania.

W porównaniu z DFS algorytm UCS:

- jest algorytmem kompletnym,
- nie ryzykuje utknięcia w nieoptymalnych, głębokich ścieżkach,
- wymaga znacznie większych zasobów pamięciowych.

Algorytm UCS stanowi naturalny etap przejściowy pomiędzy klasycznymi algorytmami przeszukiwania a algorytmami heurystycznymi, w szczególności algorytmem  $A^*$ , który zostanie omówiony w kolejnym podrozdziale.

## 2.6 Heurystyki i algorytm A\*

W klasycznych algorytmach przeszukiwania, takich jak BFS, DFS czy UCS, kolejność eksploracji przestrzeni stanów nie uwzględnia informacji o tym, jak blisko znajduje się dany stan od rozwiązania. W praktyce prowadzi to do badania wielu stanów, które z punktu widzenia celu są mało obiecujące. Rozwiązaniem tego problemu jest zastosowanie heurystyk.

### 2.6.1 Pojęcie heurystyki

**Heurystyka** jest funkcją, która dla danego stanu  $n$  dostarcza przybliżonej informacji o koszcie dotarcia z tego stanu do najbliższego stanu docelowego. Funkcję heurystyczną oznacza się zazwyczaj jako  $h(n)$ .

Heurystyki nie muszą być dokładne ani gwarantować poprawnego oszacowania kosztu. Ich celem jest ukierunkowanie algorytmu przeszukiwania w stronę obszarów przestrzeni stanów, które z większym prawdopodobieństwem prowadzą do rozwiązania.

### 2.6.2 Własności heurystyk

Szczególnie istotne znaczenie mają dwie własności funkcji heurystycznych:

- *dopuszczalność* – heurystyka nigdy nie przeszacowuje rzeczywistego kosztu dotarcia do celu, czyli:

$$h(n) \leq h^*(n),$$

gdzie  $h^*(n)$  oznacza rzeczywisty minimalny koszt dotarcia do celu,

- *spójność* (monotoniczność) – dla każdego przejścia ze stanu  $n$  do stanu  $n'$  spełniona jest nierówność:

$$h(n) \leq c(n, n') + h(n').$$

Heurystyki spełniające powyższe warunki umożliwiają zachowanie optymalności algorytmów heurystycznych.

### 2.6.3 Idea algorytmu A\*

Algorytm A\* (czyt. *A-star*) jest jednym z najważniejszych algorytmów przeszukiwania heurystycznego. Łączy on zalety algorytmu UCS z informacją heurystyczną o celu.

Dla każdego stanu  $n$  algorytm A\* oblicza wartość funkcji oceny:

$$f(n) = g(n) + h(n),$$

gdzie:

- $g(n)$  – rzeczywisty koszt dotarcia do stanu  $n$ ,
- $h(n)$  – heurystyczne oszacowanie kosztu dotarcia z  $n$  do celu.

Algorytm zawsze rozwija stan o najmniejszej wartości funkcji  $f(n)$ , traktując ją jako przybliżony koszt całkowitej ścieżki prowadzącej do rozwiązania.

### 2.6.4 Pseudokod algorytmu A\*

```
1 A(stanPoczątkowy):
2   utwórz pusta kolejkę priorytetowa Q
3   utwórz pusty zbiór odwiedzonych
4
5   dodaj stanPoczątkowy do Q z priorytetem h(stanPoczątkowy)
6
7   while Q nie jest pusta:
8     (s, fs) = usuń element o najmniejszym priorytecie z Q
9
10    if s jest stanem docelowym:
11      zwroc s jako rozwiązanie
```

```

12
13     if s nie jest w odwiedzonych:
14         dodaj s do odwiedzonych
15
16     for kazdy stanNastepny generowany z s:
17         g = kosztDotarcia(s) + kosztPrzejscia(s, stanNastepny)
18         f = g + h(stanNastepny)
19         dodaj stanNastepny do Q z priorytetem f
20
21     zwroc brakRozwiazania

```

Listing 5: Pseudokod algorytmu A\*

### 2.6.5 Złożoność czasowa i pamięciowa

Złożoność algorytmu A\* w najgorszym przypadku jest wykładnicza i porównywalna z algorytmem UCS. W praktyce jednak odpowiednio dobrana heurystyka może znacząco ograniczyć liczbę odwiedzanych stanów.

Złożoność pamięciowa A\* jest wysoka, ponieważ algorytm przechowuje w pamięci wszystkie wygenerowane stany wraz z ich ocenami.

### 2.6.6 Znaczenie algorytmu A\* w sztucznej inteligencji

Algorytm A\* znalazł szerokie zastosowanie w wielu dziedzinach sztucznej inteligencji, takich jak planowanie tras, gry komputerowe, robotyka czy systemy nawigacyjne. Stanowi on punkt odniesienia dla licznych modyfikacji i rozszerzeń, w tym algorytmów opartych na uczeniu maszynowym oraz metodach probabilistycznych.

Wprowadzenie heurystyk oraz algorytmu A\* zamyka klasyczny rozdział algorytmów przeszukiwania przestrzeni stanów i tworzy naturalne przejście do bardziej zaawansowanych metod optymalizacji i uczenia.

## 3 Algorytmy ewolucyjne

Algorytmy ewolucyjne stanowią klasę metod optymalizacyjnych inspirowanych procesami zachodzącymi w przyrodzie, w szczególności mechanizmami ewolucji biologicznej. W przeciwieństwie do klasycznych algorytmów przeszukiwania, nie operują one na pojedynczym rozwiązaniu, lecz na całej populacji potencjalnych rozwiązań, które ulegają stopniowym modyfikacjom w kolejnych iteracjach algorytmu.

Zastosowanie algorytmów ewolucyjnych jest szczególnie uzasadnione w problemach, dla których:

- przestrzeń rozwiązań jest bardzo duża lub ciągła,
- funkcja celu jest nieliniowa, nieciągła lub nieznana analitycznie,
- klasyczne metody optymalizacji zawodzą lub są zbyt kosztowne obliczeniowo.

### 3.1 Inspiracja biologiczna

Podstawą algorytmów ewolucyjnych są obserwacje dotyczące ewolucji organizmów żywych. W środowisku naturalnym osobniki lepiej przystosowane do warunków otoczenia mają większe szanse na przetrwanie i przekazanie swoich cech potomstwu.

Algorytmy ewolucyjne przenoszą te idee na grunt informatyki, traktując potencjalne rozwiązania problemu jako „osobniki”, które podlegają procesom selekcji i modyfikacji.

## 3.2 Podstawowe pojęcia algorytmów ewolucyjnych

### 3.3 Reprezentacja osobnika

Każdy osobnik w populacji reprezentuje jedno potencjalne rozwiązanie problemu.

Reprezentacja ta może przyjmować różne formy, w zależności od charakteru zadania:

- ciągi binarne,
- wektory liczb rzeczywistych,
- permutacje,
- struktury drzewiaste.

Dobór odpowiedniej reprezentacji ma kluczowe znaczenie dla skuteczności algorytmu.

#### 3.3.1 Funkcja przystosowania

Funkcja przystosowania (ang. *fitness function*) określa jakość danego osobnika, czyli stopień, w jakim reprezentowane przez niego rozwiązanie spełnia kryteria optymalności.

Algorytmy ewolucyjne dążą do maksymalizacji lub minimalizacji funkcji przystosowania, w zależności od sformułowania problemu.

#### 3.3.2 Populacja

Populacja jest zbiorem osobników przetwarzanych równolegle w ramach jednej iteracji algorytmu. Rozmiar populacji wpływa zarówno na jakość znalezionych rozwiązań, jak i na koszt obliczeniowy algorytmu.

## 3.4 Operatory ewolucyjne

### 3.4.1 Selekcja

Selekcja polega na wyborze osobników, które będą brały udział w procesie tworzenia nowego pokolenia. Osobniki o wyższym przystosowaniu mają większe szanse na wybór.

Do najczęściej stosowanych metod selekcji należą:

- selekcja turniejowa,
- selekcja ruletkowa,
- selekcja rankingowa.

### 3.4.2 Krzyżowanie

Krzyżowanie (ang. *crossover*) polega na łączeniu cech dwóch osobników rodzicielskich w celu utworzenia nowego potomka. Operator ten odpowiada za eksplorację przestrzeni rozwiązań.

### 3.4.3 Mutacja

Mutacja wprowadza losowe, niewielkie zmiany w strukturze osobnika. Jej zadaniem jest utrzymanie różnorodności populacji oraz zapobieganie przedwczesnej zbieżności algorytmu.

## 3.5 Algorytm genetyczny

Algorytm genetyczny jest najpopularniejszym przedstawicielem algorytmów ewolucyjnych. Jego ogólny schemat można przedstawić w następującej postaci.

### 3.5.1 Pseudokod algorytmu genetycznego

```
1   inicjalizuj populacje P
2   ocen przystosowanie osobnikow w P
3
4   while warunek stopu nie jest spelniony:
5       wybierz osobniki do reprodukcji
6       zastosuj krzyzowanie
7       zastosuj mutacje
8       ocen przystosowanie nowej populacji
9       zastap stara populacje nowa
10
11  zwroc najlepszy osobnik
```

Listing 6: Pseudokod algorytmu genetycznego

### 3.5.2 Własności algorytmów genetycznych

Algorytmy genetyczne:

- są odporne na lokalne ekstrema,
- nie wymagają informacji o pochodnych funkcji celu,
- łatwo poddają się równoległej implementacji,
- mogą być kosztowne obliczeniowo.

## 3.6 Przykład algorytmu genetycznego dla problemu plecakowego

Problem plecakowy (ang. *Knapsack Problem*) jest klasycznym problemem optymalizacyjnym, często wykorzystywanym do ilustracji działania algorytmów ewolucyjnych. Polega on na wyborze podzbioru przedmiotów w taki sposób, aby maksymalizować całkowitą wartość przy jednoczesnym ograniczeniu całkowitej wagi.

### 3.6.1 Sformułowanie problemu

Rozważmy plecak o maksymalnej pojemności:

$$W_{\max} = 15.$$

Danych jest 10 przedmiotów:

| Przedmiot $i$ | Waga $w_i$ | Wartość $v_i$ |
|---------------|------------|---------------|
| 1             | 2          | 3             |
| 2             | 3          | 4             |
| 3             | 4          | 8             |
| 4             | 5          | 8             |
| 5             | 9          | 10            |
| 6             | 7          | 9             |
| 7             | 6          | 7             |
| 8             | 3          | 5             |
| 9             | 1          | 2             |
| 10            | 4          | 6             |

### 3.6.2 Reprezentacja chromosomu

Chromosom binarny:

$$x = (x_1, x_2, \dots, x_{10}), \quad x_i \in \{0, 1\}.$$

### 3.6.3 Funkcja przystosowania

$$f(x) = \begin{cases} \sum_{i=1}^{10} v_i x_i, & \text{jeśli } \sum_{i=1}^{10} w_i x_i \leq W_{\max}, \\ 0, & \text{w przeciwnym przypadku.} \end{cases}$$

### 3.6.4 Populacja początkowa

Przyjmijmy populację początkową czterech osobników:

| Osobnik | Chromosom $(x_1, \dots, x_{10})$ | Waga                     | Przystosowanie           |
|---------|----------------------------------|--------------------------|--------------------------|
| A       | $(1, 1, 1, 0, 0, 0, 0, 1, 1, 0)$ | $2 + 3 + 4 + 3 + 1 = 13$ | $3 + 4 + 8 + 5 + 2 = 22$ |
| B       | $(0, 1, 1, 1, 0, 0, 0, 0, 1, 0)$ | $3 + 4 + 5 + 1 = 13$     | $4 + 8 + 8 + 2 = 22$     |
| C       | $(1, 0, 1, 1, 0, 0, 0, 1, 0, 0)$ | $2 + 4 + 5 + 3 = 14$     | $3 + 8 + 8 + 5 = 24$     |
| D       | $(1, 1, 0, 1, 0, 1, 0, 0, 0, 0)$ | $2 + 3 + 5 + 7 = 17$     | 0                        |

Najlepszym osobnikiem populacji początkowej jest:

$$C = (1, 0, 1, 1, 0, 0, 0, 1, 0, 0), \quad f(C) = 24.$$

### 3.6.5 Selekcja (turniejowa)

Wybieramy dwóch najlepszych rodziców:

$$\text{Rodzic 1: } C = (1, 0, 1, 1, 0, 0, 0, 1, 0, 0), \quad \text{Rodzic 2: } A = (1, 1, 1, 0, 0, 0, 0, 1, 1, 0).$$

### 3.6.6 Krzyżowanie jednopunktowe

Wykonujemy krzyżowanie po 6-tym genie:

$$\text{Rodzic 1: } (1, 0, 1, 1, 0, 0 \mid 0, 1, 0, 0)$$

$$\text{Rodzic 2: } (1, 1, 1, 0, 0, 0 \mid 0, 1, 1, 0)$$

Potomstwo:

$$P_1 = (1, 0, 1, 1, 0, 0, 0, 1, 1, 0)$$

$$P_2 = (1, 1, 1, 0, 0, 0, 0, 1, 0, 0)$$

### 3.6.7 Ocena potomstwa po krzyżowaniu

$$w(P_1) = 2 + 4 + 5 + 3 + 1 = 15, \quad v(P_1) = 3 + 8 + 8 + 5 + 2 = 26.$$

$$w(P_2) = 2 + 3 + 4 + 3 = 12, \quad v(P_2) = 3 + 4 + 8 + 5 = 20.$$

Po krzyżowaniu otrzymujemy osobnika  $P_1$  o przystosowaniu:

$$f(P_1) = 26,$$

co jest wynikiem lepszym niż dotychczasowe optimum  $f(C) = 24$ .

### 3.6.8 Mutacja

Założmy mutację w osobniku  $P_2$  (odwrócenie bitu  $x_{10}$  z 0 na 1):

$$P'_2 = (1, 1, 1, 0, 0, 0, 0, 1, 0, 1).$$

Ocena:

$$w(P'_2) = 2 + 3 + 4 + 3 + 4 = 16 > 15 \Rightarrow f(P'_2) = 0.$$

### 3.6.9 Nowa populacja i poprawa optimum

Zakładając strategię zastępowania najgorszego osobnika (eliminacja D), nowa populacja może przyjąć postać:

$$\{A, B, C, P_1\}.$$

Najlepszym osobnikiem po jednej iteracji jest:

$$P_1 = (1, 0, 1, 1, 0, 0, 0, 1, 1, 0), \quad w(P_1) = 15, \quad f(P_1) = 26.$$

Zatem w wyniku jednego cyklu (selekcja–krzyżowanie–mutacja) algorytm genetyczny doprowadził do *poprawy* najlepszego rozwiązania w populacji z 24 do 26.

# Bibliografia

- [1] Timofiejew Aleksander, *Algorytmy i struktury danych w językach programowania*, Wydawnictwo Akademii Podlaskiej, 2008.
- [2] Debudaj-Grabysz Agnieszka, Deorowicz Sebastian, Widuch Jacek, *Algorytmy i struktury danych : wybór zaawansowanych metod*, Wydawnictwo Politechniki Śląskiej, 2012.
- [3] Hurbans Rishal, *Algorytmy sztucznej inteligencji. Ilustrowany przewodnik*, Helion, 2021.
- [4] Deisenroth Marc Peter, Faisal A. Aldo, Ong Cheng Soon, *Matematyka w uczeniu maszynowym*, Helion, 2022.
- [5] Lannelongue Loïc, Grealey Jason, Inouye Michael, *Green Algorithms: Quantifying the carbon footprint of computation*, <https://arxiv.org/pdf/2007.07610>.