Wydział Matematyki i Nauk Informacyjnych
Politechniki Warszawskiej

Teoria Algorytmów i Obliczeń
Laboratorium - Etap 3

Dokumentacja końcowa

Adrian Bednarz,
Bartłomiej Dach,
Tymon Felski

Wersja 1.0

14.12.2017

Spis treści

1. Problem podstawowy

Poniższy rozdział zawiera informacje dotyczące podstawowej wersji problemu, które zostały przygotowane w ramach pierwszego etapu laboratorium. Niniejszy problem został szczegółowo opisany, sformułowano algorytm pozwalający na rozwiązanie dowolnego zadania w tym problemie i przedstawiono dowód poprawności. Implementacja tej wersji problemu została stworzona na potrzeby drugiego etapu laboratorium.

1.1. Opis problemu

Niniejszy rozdział poświęcony jest dokładnemu opisaniu podstawowej wersji zadanego problemu.

Dane są zbiory:

- E - **ekspertów** realizujących projekty,
- U - **umiejętności** posiadanych przez ekspertów,
- P - **projektów** do zrealizowania.

Każdemu ekspertowi przypisany jest wektor binarny opisujący posiadane przez niego umiejętności. Przykładowo, jeżeli ekspert posiada umiejętność i, to w wektorze umiejętności odpowiadającemu temu ekspertowi na i-tym miejscu znajduje się znak 1, w przeciwnym wypadku — 0.

Przykład — wektory ekspertów

Załóżmy, że liczność zbioru umiejętności U jest równa 5. Ponumerujmy umiejętności rozważane w problemie liczbami z zakresu $[1, 5]$. Niech pewien ekspert ze zbioru E posiada umiejętności 2, 3 i 5. Wówczas wektor opisujący jego umiejętności to:

$$[0, 1, 1, 0, 1]$$

Każdemu projektowi przypisany jest wektor liczbowy opisujący zapotrzebowanie na ekspertów posiadających dane umiejętności. Przykładowo, jeżeli do realizacji projektu potrzeba trzech ekspertów posiadających umiejętność i, to w wektorze umiejętności odpowiadającemu temu projektowi na i-tym miejscu znajduje się liczba 3.

Przykład — wektory zapotrzebowania projektów

Utrzymując założenie o liczności zbioru umiejętności z poprzedniego przykładu, rozważmy pewien projekt ze zbioru P. Niech jego zapotrzebowanie na ekspertów posiadających umiejętności 1 i 4 wynosi odpowiednio 4 i 3, a na pozostałe — 0. Wówczas wektor opisujący zapotrzebowanie tego projektu to:

$$[4, 0, 0, 3, 0]$$

Wszystkie projekty realizowane są w tym samym oknie czasowym, tzn. prace nad każdym z nich rozpoczynają się w momencie t_0 i kończą w późniejszym momencie t_k. Oznacza to, że jeżeli dany ekspert zostanie zatrudniony do pracy nad projektem P_1, to nie będzie mógł brać udziału w równoległym projekcie P_2. Ponadto każdy ekspert podczas pracy nad projektem może wykorzystywać tylko jedną z posiadanych umiejętności i nie może jej zmienić w trakcie trwania prac.

Prace nad danym projektem zostaną zakończone nawet jeżeli nie zostanie mu przydzielona wymagana liczba ekspertów posiadających potrzebne umiejętności, określona przez wektor liczbowy odpowiadający temu projektowi. Będzie on natomiast zrealizowany gorzej niż w przypadku, gdyby przypisana została odpowiednia liczba ekspertów. Może się również zdarzyć, że najbardziej optymalne okaże się takie przypisanie ekspertów, że nad pewnym projektem nie będzie pracował nikt.

Jeżeli zapotrzebowanie projektu nie zostanie wypełnione w całości, mamy do czynienia z brakami. Poprzez braki rozumiemy różnicę pomiędzy zapotrzebowaniem projektu na ekspertów o danych umiejętnościach a rzeczywistym przydziałem. Aby wyznaczyć braki w danym projekcie, należy odjąć wektor zapotrzebowania projektu na ekspertów od wektora zawierającego informację o ekspertach przydzielonych do tego projektu i zsumować elementy uzyskanej różnicy. Dokładna definicja tego pojęcia znajduje się w rozdziale zawierającym dowód poprawności (definicja 8).

Przykład — obliczanie liczby braków

Niech wektorem opisującym zapotrzebowanie pewnego projektu na ekspertów będzie:

$$[4, 0, 0, 3, 0]$$

Załóżmy, że do tego projektu zostali przypisani eksperci opisani przez wektory:

$$\begin{aligned}
&[1, 0, 1, 0, 1] && \text{(wykorzystuje umiejętność 1)}\\
&[1, 0, 0, 0, 1] && \text{(wykorzystuje umiejętność 1)}\\
&[1, 1, 0, 1, 0] && \text{(wykorzystuje umiejętność 4)}\\
&[0, 0, 0, 1, 1] && \text{(wykorzystuje umiejętność 4)}
\end{aligned}$$

Wówczas przydział ekspertów do tego projektu można opisać wektorem:

$$[2, 0, 0, 2, 0]$$

Braki w tym projekcie obliczymy następująco:

$$\sum([4, 0, 0, 3, 0] - [2, 0, 0, 2, 0]) = \sum([2, 0, 0, 1, 0]) = 3$$

Naszym celem jest zminimalizowanie braków w obrębie wszystkich projektów (sumy wszystkich braków), czyli znalezienie optymalnego przydziału ekspertów do projektów.

1.2. Problem znajdowania maksymalnego przepływu

Okazuje się, że problem opisany w sekcji 1.1. można uogólnić do znanego problemu znajdowania maksymalnego przepływu w sieciach. W tej sekcji zdefiniowane są podstawowe pojęcia potrzebne do opisu tego problemu.

Definicja 1. Siecią nazywamy czwórkę uporządkowaną $S = (G, c, s, t)$, gdzie:

- $G = (V, E)$ jest grafem skierowanym,
- $c : E \to \mathbb{N}$ to tzw. funkcja przepustowości,
- $s, t \in V, s \neq t$ są dwoma wyróżnionymi wierzchołkami grafu G — kolejno źródłem i ujściem sieci.

Definicja 2. Przepływem w sieci S nazywamy funkcję $f : E \to \mathbb{N}$ spełniającą następujące warunki:

1. $\forall_{e \in E} \quad 0 \leqslant f(e) \leqslant c(e)$,
2. $(\forall v \in V - \{s, t\}) \sum\limits_{u:\ uv \in E} f(uv) = \sum\limits_{u:\ vu \in E} f(vu)$ — tzw. prawo Kirchhoffa.

W ogólniejszym przypadku funkcje przepustowości i przepływu mogą mieć wartości nieujemne rzeczywiste, lecz założenie o całkowitości zapewnia, że algorytmy wyznaczające maksymalny przepływ zawsze zakończą działanie.

Prawo Kirchhoffa stanowi, że suma wartości przepływu na krawędziach wchodzących do danego wierzchołka musi być równa sumie wartości przepływu na krawędziach wychodzących z tego wierzchołka.

Definicja 3. Wartością przepływu f w sieci S nazywamy liczbę

$$W(f) = \sum_{u:\ su \in E} f(su) - \sum_{u:\ us \in E} f(us)$$

Powyższe definicje wystarczą, aby zdefiniować problem maksymalnego przepływu.

Definicja 4 (Problem maksymalnego przepływu)**.** Dana jest sieć $S = (G, c, s, t)$. Szukamy przepływu f o maksymalnej wartości $W(f)$, zwanego również **przepływem maksymalnym**.

Zagadnienie znajdowania maksymalnego przepływu jest rozwiązywalne przez wiele zachłannych algorytmów opartych na metodzie Forda-Fulkersona, polegającej na znajdowaniu ścieżek w tzw. sieci rezydualnej. Szczegółowy opis jednego z takich algorytmów znajduje się w następnej sekcji.

1.3. Algorytm

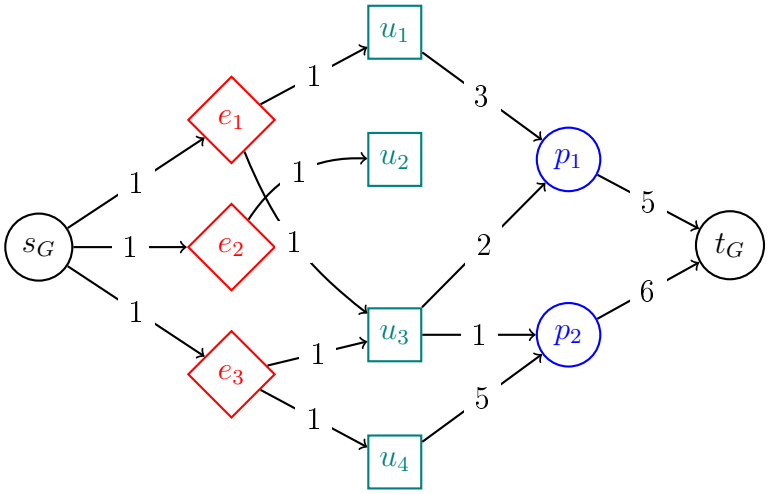
W poniższym rozdziale precyzyjnie sformułowano algorytm pozwalający na rozwiązanie dowolnego zadania w postawionym problemie. Pseudokod został podzielony na fragmenty, z którego każdy będzie rozwiązywał pewien podproblem, w celu ułatwienia opisu głównej części algorytmu.

1.3.1. Konstrukcja sieci podstawowej

Niektóre podproblemy opisane w dalszej części rozdziału będą wymagać sieci reprezentowanej przez graf skierowany, który można utworzyć na podstawie danych z zadania.

```
3 // Liczba ekspertów
4 // Liczba umiejętności
2 // Liczba projektów
// Wektory ekspertów
[1, 0, 1, 0]
[0, 1, 0, 0]
[0, 0, 1, 1]
// Zapotrzebowanie
// projektów
[3, 0, 2, 0]
[0, 0, 1, 5]
```

(a) Przykładowy format pliku wejściowego programu



(b) Wygląd sieci skonstruowanej na podstawie dostarczonych danych

Rysunek 1: Przykład skonstruowanej sieci na podstawie określonego zadania problemu

Proponowany graf `G` będzie posiadał $|E|+|U|+|P|+2$ wierzchołków, które utworzą w nim pięć warstw. Wyróżnione zostaną dwa wierzchołki — źródło `s` i ujście `t` sieci, z którego każdy będzie jedynym w swojej warstwie. Pozostałe trzy warstwy pomiędzy nimi będą zawierać wierzchołki reprezentujące odpowiednio ekspertów, umiejętności i projekty. Poniżej znajduje się pseudokod pozwalający na stworzenie takiego grafu.

Konstrukcja sieci podstawowej

```
G <- graf skierowany o liczbie wierzchołków równej |E|+|U|+|P|+2
dla każdego wierzchołka e reprezentującego eksperta w G:
    dodaj krawędź (s, e) do G
    G.c[s, e] <- 1
dla każdego wierzchołka e reprezentującego eksperta w G:
    dla każdej umiejętności u posiadanej przez eksperta e:
        dodaj krawędź (e, u) do G
        G.c[e, u] <- 1
dla każdego wierzchołka p reprezentującego projekt w G:
    dla każdej umiejętności u wymaganej przez projekt p:
        dodaj krawędź (u, p) do G
        G.c[e, u] <- liczba wymaganych ekspertów z u
dla każdego wierzchołka p reprezentującego projekt w G:
    dodaj krawędź (p, t) do G
    G.c[p, t] <- suma przepustowości krawędzi wchodzących do p
```

1.3.2. Konstrukcja sieci rezydualnej

W celu konstruowania w sieci ścieżek rozszerzających niezbędne jest utworzenie pomocniczej sieci rezydualnej. Przepustowość krawędzi w tej sieci zależy od wartości przepływu na krawędziach oryginalnej sieci.

Niech dana będzie krawędź uv i przepływ f. Wówczas w sieci rezydualnej istnieją krawędzie:

- uv o przepustowości $c(uv) - f(uv)$,
- vu o przepustowości $f(uv)$.

Sieć rezydualna będzie aktualizowana po każdym zwiększeniu przepływu wzdłuż ścieżki powiększającej. Na początku działania algorytmu (przy zerowym przepływie) będzie ona wyglądać prawie tak samo, jak wyjściowa sieć. Wystarczy stworzyć sieć opisaną w poprzednim punkcie i rozszerzyć ją w sposób następujący:

Rozszerzenie konstrukcji sieci podstawowej

```
dla każdej krawędzi (u, v) w G:
    dodaj krawędź (v, u) do G
    G.c[v, u] <- 0
```

1.3.3. Wyszukiwanie ścieżek

Do działania algorytmu potrzebna jest podprocedura wyszukująca ścieżki rozszerzające między dwoma wierzchołkami grafu, co można dość prosto zaimplementować poprzez modyfikację przeszukiwania wszerz (ang. breadth-first search, BFS). Poniżej znajduje się pseudokod żądanej podprocedury, wyszukującej ścieżki w grafie `G` od wierzchołka `s` do `t`.

Wyszukiwanie ścieżek rozszerzających w grafie

```
findAugmentingPath(G, s, t):
    Q <- pusta kolejka
    dodaj s na koniec Q
    visited <- tablica o długości równiej liczbe wierzchołków grafu G
               ze wszystkimi elementami zainicjowanymi na False
    visited[s] <- True
    parent <- pusty słownik
    dopóki kolejka Q jest niepusta:
        u <- piewrszy element z kolejki Q
        jeżeli s = t:
            zwróć tracePath(parent, s, t)
        dla każdnej krawędzi (u, v) wychodzącej z u w G:
            jeżeli visisted[v] = False i G.c[u, v] > 0:
                visited[v] <- True
                parent[v] <- u
                dodaj v na koniec Q
```

Powyższe wyszukiwanie używa pomocniczych funkcji `tracePath` oraz `constructPath`, które służą odpowiednio do wyznaczenia listy wierzchołków znajdujących się na ścieżce z `s` do `t` na podstawie słownika `parent` i zbudowania ścieżki na podstawie listy wierzchołków `L`. Są one zdefiniowane następująco:

Funkcje pomocnicze do budowania ścieżki

```
tracePath(s, t, parent):
    L <- pusta lista
    dodaj t na koniec L
    dopóki ostatni element listy L jest różny od s:
        dodaj parent[ostatni element listy L] na koniec L
    odwróć kolejność elementów listy L
    zwróć constructPath(L)

constructPath(L):
    E <- pusta lista
    u <- pierwszy element listy L
    dla każdego wierzchołka v poza pierwszym z L:
        dodaj krawędź (u, v) na koniec E
        u <- v
    zwróć E
```

Przeszukiwanie wszerz jest sposobem konstrukcji ścieżek powiększających wykorzystywanym w algorytmie Edmondsa-Karpa, będącym implementacją metody Forda-Fulkersona.

1.3.4. Wyznaczanie przepływu maksymalnego

Poniżej znajduje się zapis algorytmu wyznaczającego przepływ maksymalny zgodnie z zasadą opisaną przez Forda i Fulkersona. `G_res` jest w tym przypadku siecią rezydualną wyznaczoną na podstawie sieci podstawowej skonstruowanej w pierwszym podrozdziale.

Wyznaczanie przepływu maksymalnego

```
maxFlow(G_res):
    f <- 0
    powtarzaj:
        L <- findPath(G_res, s, t)
        df <- inf
        dla każdej krawędzi (u, v) z L:
            jeśli G_res.c[u, v] < df:
                df <- G_res.c[u, v]
        dla każdej krawędzi (u, v) z L:
            G_res.c[u, v] -= df
            G_res.c[v, u] += df
        f += df
    dopóki lista L jest niepusta
    zwróć f
```

Algorytm ten jednak nie rozwiązuje zadania wyjściowego. Optymalną wartość braków i przyporządkowanie ekspertów do projektów należy wyznaczyć z uzyskanego przepływu.

1.3.5. Konstrukcja rozwiązania

Poniższa konstrukcja przyporządkowania ekspertów do projektów korzysta z sieci podstawowej `G` (nie z sieci rezydualnej `G_res`) i z wyznaczonego przepływu `f`.

Wyznaczanie przydziału

```
constructAssignment(G, f):
    skills <- słownik pustych kolejek dla poszczególnych umiejętności
    L <- pusta lista
    dla każdego wierzchołka e reprezentującego eksperta w G:
        dla każdej krawędzi (e, u) wychodzącej z e w G:
            jeśli f[e, u] = 1:
                skills[u].push(e)
    dla każdego wierzchołka u reprezentującego umiejętność w G:
        dla każdej krawędzi (u, p) wychodzącej z u w G:
            dopóki f[u, p] > 0:
                e <- skills[u].pop()
                dodaj krotkę (e, u, p) na koniec L
                f[u, p] -= 1
    zwróć L
```

Poniższa funkcja oblicza ostateczną liczbę braków w wyznaczonym przyporządkowaniu `L`.

Wyznaczanie braków

```
calcLosses(G, L):
    need <- 0
    dla każdej krawędzi (p, t) wchodzącej do ujścia t w G:
        need += G.c[p, t]
    flow <- liczba elementów w liście L
    zwróć (need - flow)
```

1.4. Dowód poprawności

W tej sekcji wykażemy związek między postawionym problemem a zagadnieniem wyznaczania przepływu maksymalnego oraz równoważność rozwiązań obu zadań.

Na początek zdefiniujmy w sposób formalny pojęcia użyte w oryginalnym zadaniu. Załóżmy, że dane są następujące zbiory:

- **zbiór ekspertów**, oznaczony E,
- **zbiór umiejętności**, oznaczony U,
- **zbiór projektów**, oznaczony P.

Definicja 5. Funkcją umiejętności nazywamy funkcję

$$\text{ability} : E \times U \to \{0, 1\}$$

gdzie dla eksperta $e \in E$ oraz umiejętności $u \in U$ zachodzi $\text{ability}(e, u) = 1$ wtedy i tylko wtedy, gdy ekspert e posiada umiejętność u, zaś 0 w przeciwnym przypadku.

Definicja 6. Zapotrzebowaniem projektu nazywamy funkcję

$$\text{need} : P \times U \to \mathbb{N}$$

gdzie dla projektu $p \in P$ i umiejętności $u \in U$ zachodzi $\text{need}(p, u) = n$ wtedy i tylko wtedy, gdy w projekcie p liczba potrzebnych ekspertów w dziedzinie umiejętności u wynosi n.

Zauważmy, że funkcje umiejętności i zapotrzebowania projektu są tożsame z wektorami wejściowymi zadania problemu (odpowiadają wzięciu odpowiedniej ich współrzędnej).

Definicja 7. Przyporządkowaniem eksperta nazywamy relację

$$\text{assign} \subseteq E \times U \times P$$

gdzie projekt $p \in P$, umiejętność $u \in U$ oraz ekspert $e \in E$ są ze sobą w relacji assign wtedy i tylko wtedy, gdy

- ekspert e posiada umiejętność u (tj. $\text{ability}(e, u) = 1$),
- ekspert e został przyporządkowany do pracy w projekcie p w dziedzinie umiejętności u.

Każdy ekspert $e \in E$ może być w relacji z co najwyżej jedną parą postaci (u, p), gdzie $u \in U, p \in P$.
Ponadto, dla każdego projektu p i umiejętności u musi zachodzić

$$\text{assigned}(p, u) \stackrel{\text{def}}{=} |\{e \in E : (e, u, p) \in \text{assign}\}| \leq \text{need}(p, u)$$

Definicja 8. Liczbą braków w projekcie p dla danego przyporządkowania assign nazywamy liczbę

$$\text{missing}(p, \text{assign}) = \sum_{u \in U} (\text{need}(p, u) - \text{assigned}(p, u))$$

Definicja 9. Całkowitą liczbą braków dla danego przyporządkowania assign nazywamy liczbę

$$M(\text{assign}) = \sum_{p \in P} \text{missing}(p, \text{assign})$$

Widoczne jest, że M jest parametrem minimalizowanym w postawionym problemie, zależnym od końcowego przyporządkowania.

Na podstawie powyższych definicji skonstruujemy teraz sieć, której użyjemy do wyznaczenia rozwiązań problemu.

Definicja 10. Siecią przydziałów nazwiemy sieć $S = (G, c, s, t)$, gdzie:

- $G = (V_G, E_G)$ jest grafem skierowanym takim, że:
  - $V_G = E \cup U \cup P \cup \{s, t\}$,
  - $E_G = \{(e, u) : \text{ability}(e, u) = 1, e \in E, u \in U\} \cup \{(u, p) : \text{need}(u, p) > 0, u \in U, p \in P\}$, tj. krawędziami połączeni są eksperci z ich opanowanymi umiejętnościami, oraz projekty z potrzebnymi do ich realizacji umiejętnościami.
- $c : E_G \to \mathbb{N}$ jest funkcją pojemności zdefiniowaną dla krawędzi e_G następująco:
  - jeżeli $e_G = se, e \in E$, to $c(e_G) = 1$,
  - jeżeli $e_G = eu, e \in E, u \in U$, to $c(e_G) = \text{ability}(e, u) = 1$,
  - jeżeli $e_G = up, u \in U, p \in P$, to $c(e_G) = \text{need}(p, u)$,
  - jeżeli $e_G = pt, p \in P$, to
$$c(e_G) = \sum_{sp \in E_G} c(up)$$
(tj. pojemność tej krawędzi jest równa sumie pojemności krawędzi wchodzących do wierzchołka p).
- s, t są wyróżnionymi wierzchołkami z V_G — kolejno źródłem i ujściem.

Definicja 11. Odległością $\text{dist}(u, v)$ wierzchołka u od wierzchołka v w grafie G nazywamy:

- liczbę krawędzi w najkrótszej ścieżce od u do v, jeśli taka istnieje,
- 0, jeśli $u = v$,
- ∞, jeśli $u \neq v$ i nie istnieje ścieżka od u do v.

Twierdzenie 1. *Przepływ maksymalny w sieci przydziałów wyznacza przyporządkowanie o minimalnej możliwej wartości parametru M.*

Dowód. Aby dowieść to twierdzenie, wykażemy kolejno, że:

1. Każde zadanie problemu wyjściowego jest równoważne z pewną siecią przydziałów S.
   - ($\Rightarrow$) Niech dane będzie pewne zadanie problemu wyjściowego (tj. dane będą zbiory E, U, P oraz funkcje ability i need). Wówczas można dla tego zadania skonstruować sieć przydziałów za pomocą konstrukcji pokazanej w sekcji 1.3. i definicji 10.
   - ($\Leftarrow$) Niech dana będzie pewna sieć przydziałów $S = (G, c, s, t)$. Zauważmy, że wierzchołki sieci przydziałów dzielą się z definicji sieci na pięć zbiorów, określonych przez ich odległość od źródła:
     - $\text{dist}(u, v) = 0$ — singleton $\{s\}$,
     - $\text{dist}(u, v) = 1$ — zbiór ekspertów E,
     - $\text{dist}(u, v) = 2$ — zbiór umiejętności U,
     - $\text{dist}(u, v) = 3$ — zbiór projektów P,
     - $\text{dist}(u, v) = 4$ — singleton $\{t\}$,

co daje nam wyjściowe zbiory E, U, P.

Na podstawie powyższych zbiorów i funkcji przepustowości c można zrekonstruować również funkcje ability i need:

- Dla każdego $e \in E$ i $u \in U$ funkcję ability możemy zdefiniować jako

$$\text{ability}(e,u) = \begin{cases} 1, & eu \in E_G \\ 0, & eu \notin E_G \end{cases}$$

- Dla każdego $u \in U$ i $p \in P$ funkcję need możemy zdefiniować jako

$$\text{need}(p,u) = \begin{cases} c(up), & up \in E_G \\ 0, & up \notin E_G \end{cases}$$

Z każdej sieci przydziałów można skonstruować więc zadanie oryginalnego problemu.

2. Dowolny przepływ w sieci przydziałów wyznacza ilość wykonanych podzadań przy danym przyporządkowaniu.

   Niech dany będzie pewien przepływ f w sieci przydziałów S. Przyporządkowanie ekspertów do projektów assign_f wyznaczamy w następujący sposób:

   (a) Pewnego eksperta $e \in E$ przypisujemy do umiejętności $u \in U$, jeżeli $f(e,u) = 1$.

   (b) Niech dana będzie pewna umiejętność $u \in U$. Oznaczmy zbiór ekspertów przypisanych do tej umiejętności w punkcie (a) jako E_u.
   Zbiór E_s dzielimy na rozłączne podzbiory $E_{u,p}$ takie, że $|E_{u,p}| = f(u,p)$.

   (c) Dla każdego z uzyskanych podzbiorów $E_{u,p}$, gdzie $u \in U, p \in P$, do relacji assign_f dodajemy krotki

$$\{(e,u,p) : e \in E_{u,p}\}$$

   Zauważmy następujące fakty:

   - Rozważmy wierzchołek $e \in E$ odpowiadający pewnemu ekspertowi.
     Z definicji zbioru krawędzi sieci i funkcji przepustowości, do wierzchołka tego wchodzi dokładnie jedna krawędź o pojemności 1, a wychodzi z niego co najwyżej $|U|$ krawędzi o pojemności 1.
     Stąd w przepływie f tylko jedna z krawędzi wychodzących może mieć przepływ 1, a więc każdy ekspert może być przyporządkowany do co najwyżej jednej umiejętności.
   - Rozważmy dowolny wierzchołek $u \in U$ odpowiadający pewnej umiejętności.
     Z własności przepływu mamy

$$\sum_{wu \in E_G} f(wu) = \sum_{uv \in E_G} f(uv)$$

     Wiedząc, że wszystkie krawędzie wchodzące do s wychodzą ze zbioru E, oraz że wszystkie krawędzie wychodzące z s wchodzą do zbioru P, mamy

$$\sum_{e \in E} f(eu) = \sum_{p \in P} f(up)$$

Krawędzie o niezerowym przepływie wchodzące do e reprezentują ekspertów przydzielonych do danej umiejętności, zaś krawędzie o niezerowym przepływie wychodzące z e reprezentują zapotrzebowanie projektów na ekspertów z umiejętnością u.

Ponieważ suma przepływów krawędzi wchodzących i wychodzących jest taka sama, każdego eksperta przydzielonego do u można przypisać do dokładnie jednego podzadania (do dokładnie jednego projektu w dziedzinie umiejętności u), a więc można wykonać punkt (b) konstrukcji.

- Rozważmy dowolne dwa wierzchołki $u \in U, p \in P$ takie, że $up \in E_G$. Z definicji sieci mamy $c(u,p) = \text{need}(u,p)$, a z konstrukcji rozwiązania wynika, że $f(u,p) = \text{assigned}_f(u,p)$. Stąd na mocy definicji przepływu mamy

$$\text{assigned}_f(u,p) = f(u,p) \leq c(u,p) = \text{need}(u,p)$$

- Rozważmy dowolny wierzchołek $p \in P$. Z definicji funkcji pojemności, jeśli wszystkie krawędzie wchodzące do p będą wysycone przepływem (tj. $f(e) = c(e)$), to przepływ ten można przekazać w całości do ujścia krawędzią pt, bo

$$c(pt) = \sum_{up \in E_G} c(up)$$

Stąd pojemność krawędzi pt nie ogranicza wartości maksymalnego przepływu.

Wyznaczone przyporządkowanie assign_f spełnia więc wszystkie warunki prawidłowego przyporządkowania ekspertów do projektów, a ilość elementów w tej relacji odpowiada liczbie wykonanych podzadań.

3. Przepływ maksymalny wyznacza minimalną wartość parametru M.

Na mocy punktu 1., każde zadanie oryginalnego problemu jest równoważne pewnej sieci przydziałów, zaś na mocy punktu 2 dowolny przepływ w sieci przydziałów wyznacza ilość wykonanych podzadań. Wobec tego przepływ maksymalny $f_{\max}$ wyznacza maksymalną ilość wykonanych podzadań, równą $\left|\text{assign}_{f_{\max}}\right|$.

Zauważmy, że

$$\begin{aligned} M(\text{assign}) &= \sum_{p \in P} \text{missing}(p, \text{assign}) = \\ &= \sum_{p \in P} \sum_{u \in U} (\text{need}(p,u) - \text{assigned}(p,u)) = \\ &= \left(\sum_{p \in P} \sum_{u \in U} \text{need}(p,u) \right) - \left(\sum_{p \in P} \sum_{u \in U} \text{assigned}(p,u) \right) = \\ &= \left(\sum_{p \in P} \sum_{u \in U} \text{need}(p,u) \right) - |\text{assign}|, \end{aligned}$$

gdzie ostatnia równość wynika z definicji 7 (przyjęto, że jeden ekspert może być w relacji z co najwyżej jedną parą (u,p)).

W związku z tym maksymalizacja liczności przyporządkowania assign jest równoważna minimalizacji parametru M, co kończy dowód.

□

2. Rozszerzenie problemu

Niniejszy rozdział dotyczy rozszerzenia podstawowego problemu, które zostało przygotowane i zaimplementowane na porzeby trzeciego etapu laboratorium. Opisano sposób rozszerzenia problemu i przedstawiono algorytm, który pozwoli na rozwiązanie dowolnego zadania w tym problemie. Oszacowano także złożoność czasową wspomnianego algorytmu.

2.1. Opis problemu

Wybrane rozszerzenie problemu wprowadza czas jako czynnik wpływający na przyporządkowanie ekspertów do projektów. Zbiory ekspertów E, umiejętności U i projektów P z poprzedniej definicji pozostają bez zmian. Nowym elementem jest założenie, że każdy projekt trwa określoną liczbę kolejnych jednostek czasu (tzn. proces wytwórczy nie może zostać podzielony), a czas na wykonanie wszystkich projektów jest z góry określony i wszystkie muszą się w nim zmieścić.

Przykład — czas na wykonanie projektów

Jeżeli parametr określający liczbę jednostek czasu dostępnych na realizację projektów wynosi 5, to wszystkie projekty muszą zaczynać się nie wcześniej niż w chwili czasu 0 i kończyć nie później niż w chwili czasu 5.

Definicja wektora przypisanego ekspertowi ze zbioru E pozostaje bez zmian – nadal jest to wektor binarny opisujący umiejętności danego eksperta. Zmianie uległa definicja wektora odpowiadającego projektowi ze zbioru P. Jest on wciąż wektorem liczbowym, jednak jego ostatni element mówi o liczbie jednostek czasu potrzebnych na wykonanie danego projektu.

Przykład — wektor projektu

Załóżmy, że liczność zbioru umiejętności U jest równa 5. Ponumerujmy umiejętności rozważane w problemie liczbami z zakresu $[1, 5]$. Rozważmy pewien projekt ze zbioru P. Niech jego zapotrzebowanie na ekspertów posiadających umiejętności 1 i 3 wynosi odpowiednio 2 i 5, a na pozostałe — 0. Ponadto długość procesu wytwórczego tego projektu została oszacowana na 3 jednostki czasu. Wówczas wektor opisujący zapotrzebowanie tego projektu to:

$$[2, 0, 5, 0, 0, 3]$$

Ważnym jest, że jednostki czasu są niepodzielne i zarówno ogólny czas przeznaczony na wykonanie wszystkich projektów, jak i czas realizacji poszczególnych projektów musi wynosić przynajmniej jedną jednostkę czasu. Dodatkowo zakładamy również, że wymagania każdego projektu są stałe w każdej jednostce czasu jego trwania.

Ekspert zatrudniony do pracy nad projektem P_1 w jednostce czasu od chwili t_k do chwili t_{k+1} nie będzie mógł brać udziału w równoległym projekcie P_2 w tej jednostce czasu. Ponadto każdy ekspert podczas pracy nad projektem może nadal wykorzystywać tylko

jedną z posiadanych umiejętności i nie może jej zmienić w trakcie trwania jednostki czasu. Jest możliwa natomiast zmiana projektu i/lub umiejętności po upływie danej jednostki czasu. Wspomniany ekspert mógłby w jednostce czasu od chwili t_{k+1} do chwili t_{k+2} dalej pracować nad projektem P_1 wykorzystując tę samą lub inną umiejętność albo zmienić projekt na P_2 i pracować na wcześniej wspomnianych zasadach.

Definicja braków i jej założenia są nadal podtrzymywane. Różnica w rozszerzonym problemie dotyczy ich interpretacji, ponieważ braki i przyporządkowania rozpatrujemy teraz w danej jednostce czasu. Oznacza to, że zarówno braki jak i przyporządkowania ekspertów do danego projektu mogą się różnić na przestrzeni czasu, a ostateczna liczba braków jest równa sumie wszystkich braków w czasie trwania procesu wytwórczego projektu.

Przykład — obliczanie ostatecznej liczby braków

Załóżmy, że pewien projekt ze zbioru P jest realizowany przez 3 jednostki czasu. Każdej jednostce czasu odpowiada pewne przyporządkowanie ekspertów do tego projektu i określają one braki odpowiednio 21, 3 i 7. Wówczas ostateczna liczba braków w tym projekcie jest zdefiniowana przez liczbę będącą sumą braków w poszczególnych jednostkach czasu, czyli:

$$21 + 3 + 7 = 31$$

Naszym celem jest zminimalizowanie braków w obrębie wszystkich projektów na przestrzeni wszystkich jednostek czasu, czyli znalezienie takiego planu projektów (chwil czasu, w których projekty mają się rozpoczynać), aby możliwe było optymalne przydzielenie ekspertów do projektów.

2.2. Algorytmy genetyczne

Algorytmami genetycznymi nazywamy metaheurystyki opierające się na ideach genetyki oraz selekcji naturalnej. Algorytmy te swoje główne zastosowanie znajdują w problemach optymalizacyjnych i są formą optymalizacji stochastycznej (tj. opartej na losowości).

Najczęściej w problemach rozwiązywanych tego typu metodami nie jest znana postać ogólna rozwiązań optymalnych lub bliskich optymalnemu oraz przeszukiwana przestrzeń rozwiązań jest duża, lecz obliczenie jakości pewnego rozwiązania jest obliczeniowo mało kosztowne. [1]

Algorytmy genetyczne nie gwarantują odnalezienia rozwiązania optymalnego, lecz zastosowane do odpowiednich problemów przynoszą rozwiązania bliskie optymalnemu w czasie o wiele krótszym niż w przypadku pełnego przeszukania przestrzeni rozwiązań.

Podstawowymi pojęciami w tym rodzaju algorytmów są: osobnik, gen i populacja.

Definicja 12. Osobnikiem (chromosomem) nazywamy k-elementowy wektor $o \in \mathbb{N}^k$.

Definicja 13. Genem nazywamy pewien element wektora o.

Definicja 14. Populacją nazywamy skończony zbiór osobników $P \subseteq \mathbb{N}^k$.

W kontekście rozwiązywanego problemu, celem osobnika jest reprezentowanie pojedynczego potencjalnego rozwiązania. Rozważany wektor musi więc zawierać wystarczająco dużo danych (genów), aby dało się na jego podstawie dokonać jednoznacznej konstrukcji rozwiązania, które później zostanie poddane ocenie.

Podstawą do wyboru, które rozwiązania przetrwają wraz ze wzrostem populacji jest tzw. funkcja przystosowania (ang. *fitness function*).

Definicja 15. Funkcją przystosowania nazywamy funkcję $f : \mathbb{N}^k \to \mathbb{N}$, określającą przystosowanie pojedynczego osobnika populacji.

Zazwyczaj powyższa funkcja jest maksymalizowana (im większa jej wartość, tym lepsze rozwiązanie).

Dodatkowo, w celu wprowadzenia (pozytywnych lub negatywnych) zaburzeń w populacji, definiowane są dwa operatory genetyczne: krzyżowania i mutacji.

Definicja 16. Funkcją krzyżowania nazywamy funkcję $\text{cross} : \mathbb{N}^k \times \mathbb{N}^k \to \mathbb{N}^k \times \mathbb{N}^k$.

Definicja 17. Funkcją mutacji nazywamy funkcję $\text{mut} : \mathbb{N}^k \to \mathbb{N}^k$.

Definicja operacji krzyżowania i mutacji może zależeć od wybranego problemu, lecz w zdecydowanej większości przypadków operacja krzyżowania jest implementowana jako podział obu osobników wyjściowych (rodziców) na identyczne fragmenty, a następnie produkcję nowych dwóch osobników (dzieci) poprzez wymianę parzystych lub nieparzystych fragmentów wektorów rodziców. Najczęstszą implementacją mutacji jest zaś losowa zmiana części elementów osobnika-rodzica.

Po wprowadzeniu powyższych definicji, możliwe jest wyrażenie szkieletu algorytmu w postaci pseudokodu.

Schemat algorytmu genetycznego

```
genetic_algorithm:
    wylosuj populację początkową
    powtarzaj:
        oblicz funkcję przystosowania dla osobników w populacji
        jeżeli osiągnięto warunek zatrzymania:
            zwróć najlepiej przystosowanego osobnika
        wybierz chromosomy do kolejnej generacji
        zastosuj operatory krzyżowania i mutacji
        utwórz kolejną generację populacji
```

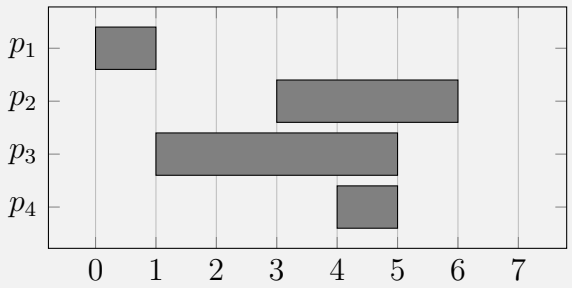
2.3. Algorytm

Wybranie uogólnienie problemu jest optymalizacyjnym problemem planowania. Problemy tego typu są jednym z klasycznych, występujących w literaturze przypadków zastosowania algorytmów genetycznych (przykładem jest praca [2]).

Pierwszą, podstawową kwestią do ustalenia przy projektowaniu algorytmu genetycznego jest sposób translacji potencjalnych rozwiązań na chromosomy (osobniki). W przypadku tego problemu liczba genów będzie równa liczbie projektów (mocy zbioru P), zaś i-ty gen w pewnym chromosomie $o = (x_1, x_2, \ldots, x_{|P|})$ będzie równy indeksowi jednostki czasu, w której planowane jest rozpoczęcie i-tego projektu.

Przykład — reprezentacja rozwiązania zadania przez chromosom

Załóżmy, że dany jest następujący rozkład projektów w oknie czasowym długości 7 przewidzianym na ich wykonanie:



Rysunek 2: Przykładowy rozkład projektów

Wówczas osobnikiem (chromosomem) odpowiadającym takiemu rozkładowi projektów jest osobnik

$$(0, 3, 1, 4)$$

Zauważmy, że za sprawą tego, że długość poszczególnych jest częścią danych wejściowych, możliwe jest zrekonstruowanie kompletnego rozwiązania na podstawie znajomości tylko czasu rozpoczęcia i długości projektów.

W kolejnych iteracjach (generacjach) algorytmu będzie rozważany pewien zbiór chromosomów, zwany też populacją, tożsamych z potencjalnymi rozwiązaniami problemu, inicjowany losowo i modyfikowany za pośrednictwem operatorów genetycznych.

2.3.1. Generacja populacji początkowej

W większości przykładów algorytmy genetyczne rozpoczynają swoje działanie od inicjacji populacji osobnikami o cechach losowych. Podobny mechanizm został zastosowany w zaimplementowanym rozwiązaniu — wartość każdego genu x_i jest losowana ze zbioru $\{0, 1, 2, \ldots, D - d_i\}$, gdzie D oznacza długość okna czasowego wyznaczonego dla wszystkich projektów, zaś d_i oznacza długość trwania i-tego projektu ze zbioru P.

Taki sposób inicjacji populacji zapewnia, że dla poprawnych danych wejściowych populacja początkowa zawsze będzie składała się z poprawnych rozwiązań (tj. takich, w których okres realizacji żadnego projektu nie wykroczy poza okno czasowe).

Losowanie populacji początkowej

```
init_population:
    population <- []
    dopóki liczność population jest mniejsza od docelowej:
        member <- []
        dla każdego projektu p:
            g <- losowa liczba całkowita z przedziału [0, D - p.d]
            dodaj g na koniec member
        dodaj member do population
```

2.3.2. Funkcja przystosowania

W projekcie algorytmu genetycznego niezbędne jest również wskazanie przyjętej funkcji przystosowania. Aby w sposób formalny zdefiniować tą funkcję, należy poszerzyć definicję niektórych pojęć zdefiniowanych dla podstawowej wersji problemu.

Definicja 18. Przyporządkowaniem eksperta nazywamy relację

$$\text{assign} \subseteq E \times U \times P \times \{0, 1, \ldots, D-1\}$$

gdzie ekspert $e \in E$, umiejętność $u \in U$, projekt $p \in P$ oraz liczba $k \in \{0, 1, \ldots, D-1\}$ są ze sobą w relacji wtedy i tylko wtedy, gdy:

- ekspert e posiada umiejętność u (tj. $\text{ability}(e, u) = 1$),
- ekspert e został przyporządkowany do pracy w projekcie p w dziedzinie umiejętności u w przedziale czasowym $[k, k+1)$.

Każdy ekspert $e \in E$ może być w relacji z co najwyżej jedną trójką postaci (u, p, k), gdzie $u \in U, p \in P, k \in \{0, 1, \ldots, D-1\}$.
Ponadto, dla każdego projektu p, umiejętności u i chwili k musi zachodzić

$$\text{assigned}(p, u, k) \stackrel{\text{def}}{=} |\{e \in E : (e, u, p, k) \in \text{assign}\}| \leq \text{need}(p, u)$$

oraz dla każdej czwórki $(e, u, p_i, k) \in \text{assign}$ musi zachodzić

$$t_i \leq k < t_i + d_i$$

gdzie t_i jest czasem rozpoczęcia projektu p_i, zaś d_i — czasem trwania tego projektu.

Definicja 19. Liczbą braków w projekcie $p \in P$ w chwili k dla danego przyporządkowania assign nazywamy liczbę

$$\text{missing}(p, k, \text{assign}) = \sum_{u \in U} (\text{need}(p, u) - \text{assigned}(p, u, k))$$

gdzie $k \in \{0, 1, \ldots, D-1\}$.

Definicja 20. Liczbą braków w projekcie $p_i \in P$ dla danego przyporządkowania assign nazywamy liczbę

$$\text{missing}(p, \text{assign}) = \sum_{t=t_i}^{t_i+d_i-1} \text{missing}(p, t, \text{assign})$$

gdzie t_i jest czasem rozpoczęcia projektu p_i, zaś d_i — czasem trwania tego projektu.

Definicja 21. Całkowitą liczbą braków dla danego przyporządkowania assign nazywamy liczbę

$$M(\text{assign}) = \sum_{p \in P} \text{missing}(p, \text{assign})$$

Funkcją przystosowania wybraną w tym problemie jest właśnie funkcja obliczająca całkowitą ilość braków w przyporządkowaniu zgodnie z definicją 21. Pod niektórymi względami jest to nieortodoksyjny wybór, ponieważ celem algorytmu jest minimalizacja tej funkcji, a nie jej maksymalizacja (jak postępuje wiele klasycznych przykładów algorytmów genetycznych). Nie ma to jednak wielkiego wpływu na wygląd algorytmu — wystarczy dokonać jedynie kilku przekształceń.

Znając już definicję funkcji przystosowania, przyjrzyjmy się metodzie obliczania tej funkcji oraz oceny poszczególnych osobników. Proces oceny pojedynczego rozwiązania składa się z następujących etapów:

1. sprawdzenie poprawności,
2. podział na podproblemy,
3. ocena podproblemów,
4. wyznaczenie ostatecznej oceny całego problemu.

O ile dany osobnik nie zostanie usunięty z populacji na etapie selekcji, algorytm dokona jego oceny dokładnie raz.

Sprawdzenie poprawności rozwiązania

W procesie tworzenia kolejnych generacji przez algorytm, może się zdarzyć, że zastosowanie operatorów genetycznych powoduje powstanie nieprawidłowego osobnika (tj. takiego, który nie reprezentuje prawidłowego rozwiązania zadania). W rozważanym problemie, jedynym rodzajem takiego błędu może być uzyskanie rozwiązania, w którym czas zakończenia jednego z projektów wykracza poza ustaloną długość okna czasowego.

Detekcja tego typu rozwiązań opiera się na ręcznej weryfikacji genów poszczególnych osobników. Rozwiązania błędne otrzymują specjalną, sztuczną wartość funkcji przystosowania, równą -1, która jest tak naprawdę flagą ustawioną na potrzeby późniejszych kroków algorytmu.

Sprawdzenie poprawności pojedynczego rozwiązania

```
validate_scheduling(m):
    dla każdego genu m[i] w osobniku m:
        jeśli m[i] + p[i].d > D:
            zwróć fałsz
    zwróć prawdę
```

Podział na podproblemy

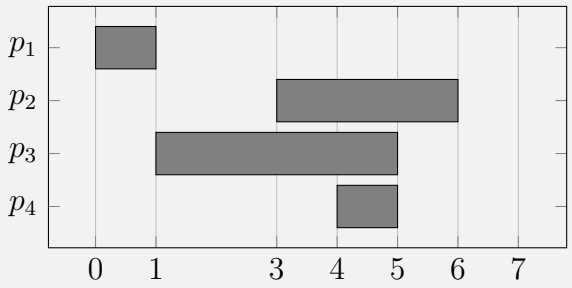
Wyznaczenie wartości funkcji M dla pewnego przyporządkowania ułatwiają założenia przyjęte w definicji rozszerzenia problemu. Założenie pozwalające ekspertom na zmianę projektów po poszczególnych jednostkach czasu pozwala podzielić ocenę rozwiązania uogólnionego problemu i sprowadzić ją do oceny podstawowego rozwiązania.

Podział odbywa się na podstawie czasów rozpoczęcia i zakończenia każdego z projektów w wyznaczonym rozwiązaniu. Jeśli rozważymy zbiór zawierający początki i końce każdego projektu (jako liczby) oraz początek i koniec wyznaczonego okna czasowego i posortujemy ten zbiór, możemy na podstawie posortowanych elementów wyznaczyć przedziały, w obrębie których nie zmienia się zbiór wykonywanych projektów.

Jeżeli przy podziale okaże się, że istnieją przedziały czasowe, podczas których nie jest wykonywany żaden projekt, przedziały te nie są rozważane, jako, że z definicji funkcji braków dla projektu każdy projekt ma w tym przedziale zerowe braki.

Przykład — podział oceny rozwiązania na podproblemy

Rozważmy rozkład projektów z rysunku 2. Opisywany w powyższym akapicie zbiór dla tego rozwiązania ma elementy $\{0, 1, 3, 4, 5, 6, 7\}$ i dzieli oś czasową projektów na przedziały zaznaczone na poniższym rysunku liniami pionowymi:



Rysunek 3: Podział osi czasowej na przedziały

W każdym z tych przedziałów każdy z projektów jest albo wykonywany, albo nie. Przedział $[6, 7]$ nie będzie oceniany, ponieważ nie wykonywany jest żaden projekt, a więc liczba braków w tym przedziale wynosi 0. W związku z tym przedziałami zwróconymi przez odpowiednią funkcję będą

$$\{[0, 1], [1, 3], [3, 4], [4, 5], [5, 6]\}$$

Ocena pojedynczego podproblemu

Rozważmy pojedynczy przedział powstały w wyniku procedury opisanej powyżej. Zauważmy, że w obrębie tego przedziału spełnione są następujące założenia problemu podstawowego:

- zapotrzebowanie każdego projektu jest stałe w czasie,
- wszystkie projekty wykonywane są w tej samej jednostce czasowej.

Definicja uogólnienia nie zapewnia jednak, że spełnione będzie założenie o tym, że żaden z ekspertów nie zmieni swojego projektu w trakcie przedziału. Okazuje się jednak, że można to założenie przyjąć i nadal otrzymać rozwiązanie optymalne pod względem braków, co udowodnimy w sekcji 2.4.

Wobec tego, aby uzyskać łączną ilość braków dla wszystkich projektów w danym przedziale, wystarczy stworzyć zadanie problemu podstawowego, w którym wektorami projektów są wektory z wejścia (jeśli dany projekt jest wykonywany w danym przedziale) lub wektory zerowe (w przeciwnym przypadku). W przypadku, gdy długość przedziału jest większa niż 1, wystarczy uzyskaną liczbę braków przemnożyć przez jego długość.

Ocena podproblemu

```
solve_for_interval(I):
    D <- struktura danych dla problemu podstawowego
    przepisz dane ekspertów dla problemu rozszerzonego do D
    dla każdego projektu p:
        jeżeli p jest wykonywany w przedziale I:
            dodaj wymagania projektu p do D
        w przeciwnym przypadku:
            dodaj zerowy wektor wymagań do D
    b <- base_problem(D)
    zwróć b * I.length
```

Ostateczna ocena

Ocena całego rozwiązania jest równa sumie oceny wyznaczonych podproblemów.

Ocena pojedynczego rozwiązania

```
fitness_function(m):
    jeśli validate_scheduling(m) == fałsz:
        zwróć -1
    I_list <- generate_intervals(m)
    f <- 0
    dla każdego przedziału I w I_list:
        f <- f + solve_for_interval(I)
    zwróć f
```

Dla uproszczenia w pseudokodzie pominięto zapamiętywanie wyznaczonego przyporządkowania. W faktycznym programie przyporządkowania pamiętane są jednak razem z wyznaczoną wartością funkcji przystosowania, aby nie liczyć ich wielokrotnie.

2.3.3. Wybór najlepszego osobnika z populacji

Wartość funkcji przystosowania obliczana jest na początku algorytmu dla całej populacji początkowej oraz w każdej kolejnej generacji dla osobników nowo dodanych do populacji. Program wyznacza najlepszego osobnika znajdującego się w populacji w obecnej generacji i porównuje go z najlepszym osobnikiem znalezionym do tej pory.

Jeżeli nastąpiła poprawa wyniku, osobnik zapamiętywany jest jako nowy najlepszy. Zerowany jest również licznik, który przechowuje informację, ile generacji temu nastąpiła ostatnia zmiana najlepszego rozwiązania.

2.3.4. Warunki stopu

Na tym etapie głównej pętli algorytmu sprawdzane są warunki zatrzymania obliczeń. Wyszczególnione zostały trzy warunki:

- znalezione zostało rozwiązanie optymalne, tj. najlepsze znalezione rozwiązanie ma liczbę braków równą 0,
- upłynęła określona liczba generacji od ostatniej zmiany najlepszego osobnika,
- upłynęła określona dla całego algorytmu górna granica liczby generacji.

W przypadku dwóch ostatnich warunków stopu, liczby generacji są parametrem programu modyfikowalnym z poziomu kodu źródłowego.

2.3.5. Kontrola populacji

W kolejnych iteracjach algorytmu powstają chromosomy nowej populacji. Kluczowym zjawiskiem występującym w genetyce jest stworzenie nowej generacji bazując na osobnikach już istniejących. W świecie algorytmów genetycznych sprowadza się to do:

1. wyboru kandydatów biorących udział w stworzeniu nowej populacji,
2. zastosowania operatorów genetycznych do osobników aktualnej populacji,
3. ograniczenia rozmiaru nowopowstałej populacji.

W zależności od wartości funkcji przystosowania osobnik ma większe lub mniejsze szanse na udział w tworzeniu kolejnego pokolenia. Istnieje wiele podejść umożliwiających wybór osobników, jednym z nich jest **metoda koła ruletki**.

Metoda koła ruletki

Polega na wielokrotnym losowaniu chromosomu z dotychczasowej populacji uwzględniając fakt, że nie każdy osobnik ma jednakową szansę na bycie wylosowanym. Każdy z nich ma przypisane prawdopodobieństwo bycia wybranym. W przypadku opracowanego algorytmu jest ono odwrotnie proporcjonalne do wartości funkcji przystosowania (ponieważ funkcja przystosowania jest minimalizowana). Warto zauważyć, że nie zapewnia to zachowania osobników o najniższej wartości funkcji przystosowania – jedynie zwiększa prawdopodobieństwo ich wyboru.

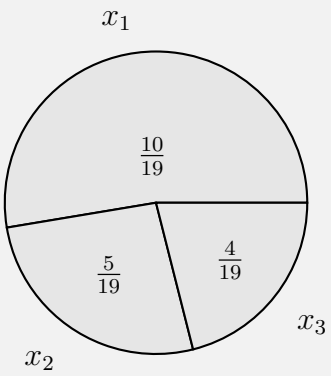
Przykład — metoda koła ruletki

Załóżmy, że w populacji znajdują się trzy rozwiązania danego problemu o wartościach funkcji przystosowania kolejno 2, 4 i 5. Wówczas proces obliczania prawdopodobieństwa wyboru każdego z tych rozwiązań na etapie redukcji populacji przedstawia poniższa tabela:

	x_1	x_2	x_3	Suma
$\frac{1}{M(x)}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{19}{20}$
$p(x)$	$\frac{10}{19}$	$\frac{5}{19}$	$\frac{4}{19}$	1

Tablica 1: Sposób obliczania prawdopodobieństw wyboru osobników

Prawdopodobieństwa te można intuicyjnie zobrazować za pomocą wykresu kołowego:



Rysunek 4: Wykres kołowy ilustrujący działanie ruletki. Stosunek pól poszczególnych wycinków kołowych do całości pola całego koła reprezentuje prawdopodobieństwo przeżycia osobnika.

Metoda koła ruletki – pseudokod

```
roulette(population, new_population_size):
    probabilities <- []
    dla osobnika x populacji population:
        fitness_value <- ustal wartość funkcji przynależności dla
            osobnika x
        dodaj 1 / fitness_value na koniec probabilities
    probabilities <- probabilities / sum(probabilities)
    indices <- wybierz indeksy new_population_size elementów wektora
        probabilities zgodnie z określonymi w nim wartościami
        prawdopodobieństwa
    new_population = []
    dla indeksu i w indices:
        dodaj osobnika population[i] do new_population
    zwróć new_population
```

Uwagi

Po zastosowaniu przede wszystkim operatora mutacji może okazać się, że nowo powstały osobnik nie jest poprawnym rozwiązaniem. Przykładowo, patrząc na rozpatrywany problem byłby tak, gdyby chwila rozpoczęcia wykonania projektu wypadłaby zbyt późno tzn. projekt mógłby zakończyć się po upływie ustalonego na wykonanie wszystkich projektów okna czasu.

W zastosowanym podejściu ustalono, że najlepsze wyniki można uzyskać przy zastosowaniu metody koła ruletki dopiero po przekroczeniu z góry ustalonej wielkości populacji. W innych przypadkach następna populacja jest inicjowana osobnikami populacji poprzedniej.

W szczególnych przypadkach po zastosowaniu operatorów mutacji i krzyżowania mogłoby dojść do sytuacji, gdzie wszystkie osobniki nie odpowiadałyby poprawnej definicji problemu. Taka sytuacja może wyniknąć m.in. ze złego wyboru parametrów mutacji. Jeśli tak się stanie to algorytm zakończy działanie.

2.3.6. Krzyżowanie osobników

Jedną z idei algorytmów genetycznych jest symulacja ewolucji, którą można zaobserwować w przyrodzie. Odbywa się to za pomocą operatorów genetycznych: krzyżowania i mutacji.

Pierwsza z operacji – krzyżowanie dwóch osobników w populacji – działa tak jak to było opisywane w rozdziale 2.2. o algorytmach genetycznych. Dwa osobniki dzielone są na $n + 1$ odcinków równej długości poprzez ich podział w n punktach. Następuje wymiana fragmentów genów pomiędzy nimi, której wynikiem jest dwóch potomków osobników-rodziców.

Poniżej znajduje się pseudokod funkcji realizującej `n`-punktowe krzyżowanie dwóch osobników `parent1` oraz `parent2`, gdzie `n` jest liczbą z przedziału [1, długość osobnika − 1].

Krzyżowanie dwóch osobników

```
n_point_crossover(parent1, parent2, n):
    offspring1 <- kopia parent1
    offspring2 <- kopia parent2
    jeżeli przeprowadzenie krzyżowania nie ma sensu:
        zwróć offspring1 i offspring2
    cuts <- tablica n losowych liczb z zakresu [1, parent1.len - 1]
    dodaj parent1.len na koniec cuts
    dla każdej kolejnej pary (cut_from, cut_to) ze zbioru cuts:
        zamień offspring1[cut_from:cut_to] na parent2[cut_from:cut_to]
        zamień offspring2[cut_from:cut_to] na parent1[cut_from:cut_to]
    zwróć offspring1 i offspring2
```

Krzyżowanie może nie być wykonane w przypadku, jeżeli nie wyspecyfikowano projektów lub projekt jest tylko jeden, więc osobniki mają długość równą 1 (posiadają jeden gen).

Poniżej znajduje się przykład wyniku operacji krzyżowania dla konkretnych osobników.

Przykład — operacja krzyżowania

Niech dane będą dwa osobniki z populacji:

$$\texttt{p1} = [0, 1, 2, 3, 4, 5],$$
$$\texttt{p2} = [5, 4, 3, 2, 1, 0].$$

Naszym celem jest ich skrzyżowanie w 3 punktach i uzyskanie dwóch potomków. Załóżmy, że wylosowane trzy indeksy punktów przecięcia to 1, 3 i 4. Wówczas nasza tablica `cuts` wygląda następująco:

$$\text{cuts} = [1, 3, 4, 6],$$

ponieważ długość osobnika wynosi w tym przypadku 6. Kolejne pary `(cut_from, cut_to)`, czyli granice odcinków wymiany genów, które możemy tutaj wyróżnić to `(1, 3)` oraz `(4, 6)`. Jeżeli liczba elementów `cuts` byłaby nieparzysta, ostatni element zostałby pominięty, ponieważ nie utworzyłby z innym pary.

Zgodnie z algorytmem, na dwóch wymienionych odcinkach (przy czym lewy brzeg odcinka jest inkluzywny, a prawy – ekskluzywny) następuje wymiana genów pomiędzy osobnikami-rodzicami i otrzymujemy dwóch nowych osobników-potomków:

$$\texttt{o1} = [0, 4, 3, 3, 1, 0],$$
$$\texttt{o2} = [5, 1, 2, 2, 4, 5].$$

Jeżeli nowe osobniki otrzymane w wyniku tej operacji nie występują jeszcze w populacji, są do niej dodawane. Liczba osobników poddanych krzyżowaniu a konkretniej szansa na to, że dany osobnik zostanie wylosowany do krzyżowania jest jednym z parametrów algorytmu genetycznego, który zależy od rozwiązywanego zadania i jego rozmiaru.

2.3.7. Mutacja osobników

Drugim operatorem genetycznym biorącym udział w procesie ewolucji jest mutacja. Pewien osobnik poddawany jest zamianie wartości `n` losowych genów na nowe – także losowe.

Poniżej znajduje się funkcja przeprowadzająca mutację `n` genów u osobnika `member` zapisana w formie pseudokodu, gdzie `n` jest liczbą z przedziału [1, długość osobnika].

Krzyżowanie dwóch osobników

```
n_point_mutation(member, n):
    mutated <- kopia member
    jeżeli przeprowadzenie mutacji nie ma sensu:
        zwróć mutated
    mutation <- tablica n losowych liczb z zakresu [0, member.len - 1]
    possible_genes <- zbiór wszystkich możliwych wartości genów
    dla każdego indeksu genu i z mutation:
        original_gene <- member[i]
        usuń original_gene z possible_genes
        mutated[i] <- losowy element ze zbioru possible_genes
        dodaj original_gene do possible_genes
    zwróć mutated
```

Wykonanie mutacji, podobnie jak krzyżowania, może w pewnych przypadkach nie przynieść efektu, więc nie zostanie przeprowadzone (na przykład, jeżeli określona liczba projektów wynosi 0 lub dozwolony jest tylko jeden gen – zerowy). Poniżej zamieszczono przykład wyniku mutacji konkretnego osobnika populacji.

Przykład — operacja mutacji

Niech dany będzie osobnik z populacji:

$$\texttt{m1} = [0, 1, 2, 3, 4, 5].$$

Naszym celem jest przeprowadzenie mutacji w 3 punktach i otrzymanie nowego osobnika. Załóżmy, że wylosowane trzy indeksy punktów mutacji to 1, 3 i 4. Dla każdego genu zostanie wylosowany nowy – inny niż poprzedni, aby mutacja przyniosła efekt. Niech nowymi genami będą odpowiednio 3, 0, 5. Wówczas nowy osobnik będzie postaci:

$$\texttt{m2} = [0, 3, 2, 0, 5, 5].$$

Analogicznie do operacji krzyżowania, zmutowany osobnik zostanie dodany do populacji tylko, jeżeli jeszcze w niej nie występuje. Szansa na to, że dany osobnik zostanie poddany mutacji jest także parametrem algorytmu genetycznego, który zależy od rozwiązywanego zadania i jego rozmiaru.

2.3.8. Działanie

Ogólny zarys algorytmu wygląda następująco:

Pseudokod algorytmu

```
solution(max_pop, max_it, max_it_without_change, E, S, P, D):
    population <- wygeneruj populację początkową przy pomocy
                  init_population
    best, best_fitness <- None, INF
    it_without_change <- -1
    powtarzaj max_it razy:
        it_without_change += 1
        fitness_values <- []
        dla osobnika x populacji population:
            fitness_value <- oblicz wartość funkcji fitness_function
                             dla x
            dodaj fitness_value na koniec fitness_values
            jeśli fitness_value > 0 && best_fitness < fitness_value:
                best, best_fitness <- x, fitness_value
                it_without_change <- 0
        jeśli best_fitness == 0 lub
              it_without_change == max_it_without_change:
           zwróć best
        jeśli len(population) > max_pop:
            population <- roulette(population, max_pop)
        population <- zastosuj krzyżowanie do populacji
        population <- zastosuj mutację do populacji
    zwróć best
```

Zaprezentowany ogólny opis algorytmu stanowi punkt odniesienia dla analizy złożoności w sekcji 2.5.

2.4. Dowód poprawności

Z uwagi na fakt, że zaimplementowany algorytm genetyczny ma w dużej mierze charakter losowy i nie gwarantuje znalezienia rozwiązania optymalnego, nie jest możliwe przeprowadzenie konwencjonalnego dowodu jego poprawności. Jednakże w przypadku uproszczeń zastosowanych w rozwiązaniu, szczególnie na etapie oceny pojedynczego osobnika, należy wykazać, że nie mają one negatywnego wpływu na całościowe działanie algorytmu.

W tej podsekcji wykażemy dwie własności rozwiązywanego problemu, które pozwalają na zastosowanie wyżej wymienionych uproszczeń bez pogorszenia jakości algorytmu.

Definicja 22. Niech dane będzie pewne przyporządkowanie assign dla problemu rozszerzonego oraz wektor $t_1, t_2, \ldots, t_{|P|}$ oznaczający czasy rozpoczęcia poszczególnych projektów. Wówczas dla dowolnej chwili $t \in \{0, 1, \ldots, D-1\}$ przyporządkowanie assign_t zdefiniowane następująco:

$$\text{assign}_t = \{(e, u, p) : e \in E, u \in U, p \in P, (e, u, p, t) \in \text{assign}\}$$

nazywamy **przyporządkowaniem chwilowym**.

Twierdzenie 2. *Dla każdego* $t \in \{0, 1, \ldots, D-1\}$ *przyporządkowanie chwilowe* assign_t *jest poprawnym rozwiązaniem problemu podstawowego dla danych wejściowych* ability_t, need_t *postaci*

- $(\forall e \in E)(\forall u \in U) \quad \text{ability}_t(e, u) = \text{ability}(e, u)$,
- $(\forall p_i \in P)(\forall u \in U) \quad \text{need}_t(p_i, u) = \begin{cases} \text{need}(p_i, u) & \textit{jeżeli } t_i \leq t < t_i + d_i, \\ 0 & \textit{w przeciwnym przypadku}, \end{cases}$

gdzie d_i *jest długością projektu* p_i, *a* t_i *jego czasem rozpoczęcia.*

Powyższe twierdzenie ma na celu uzasadnienie możliwości użycia algorytmu podstawowego do oceny pojedynczego osobnika w algorytmie genetycznym.

Dowód. Aby udowodnić to twierdzenie, sprawdźmy, czy zaproponowane przyporządkowania spełnia warunki z definicji 7 dla problemu podstawowego.

- Z definicji przyporządkowania, jeśli $(e, u, p, t) \in \text{assign}$, to $\text{ability}(e, u) = 1$.
- Każdy ekspert $e \in E$ mógł być w relacji z co najwyżej jedną trójką (u, p, t), a więc w obliczu ustalenia wartości t, w przyporządkowaniu assign_t może być w relacji z co najwyżej jedną parą (u, p).
- Aby sprawdzić ostatni warunek, musimy rozpatrzyć dwa przypadki:
  1. Jeśli projekt p_i jest wykonywany w chwili t (tj. $t_i \leq t < t_i + d_i$), to z definicji przyporządkowania mamy
  $$\text{assigned}_t(p_i, u) = \text{assigned}(p_i, u, t) \leq \text{need}(p_i, u)$$
  2. Jeśli projekt p_i nie jest wykonywany w chwili t, to w przyporządkowaniu assign nie może istnieć żadna czwórka postaci (e, u, p_i, t), $e \in E, u \in U$, a więc
  $$\text{assigned}_t(p_i, u) = \text{assigned}(p_i, u, t) = 0 \leq \text{need}(p_i, u)$$

Przyporządkowanie assign_t spełnia więc wszystkie warunki z definicji. □

Definicja 23. Ciągiem zdarzeń dla rozwiązania problemu rozszerzonego będziemy nazywać rosnący ciąg $\{e_i\}_0^k$ zawierający wszystkie elementy ze zbioru

$$\{t_i, t_i + d_i : i = 1, 2, \ldots, |P|\} \cup \{0, D\}$$

gdzie t_i oznacza początek wykonywania projektu p_i, d_i — długość projektu p_i, zaś D — długość okna czasowego przeznaczonego na wykonanie projektów.

Twierdzenie 3. *Niech dane będzie pewne optymalne rozwiązanie problemu rozszerzonego oraz odpowiadające mu: przyporządkowanie* assign *i ciąg zdarzeń* $\{e_i\}_0^k$. *Wówczas*

$$M(\text{assign}) = \sum_{i=0}^{k-1} M(\text{assign}'_{e_i}) \cdot (e_{i+1} - e_i)$$

gdzie assign′ *oznacza rozwiązanie optymalne problemu podstawowego.*

Powyższe twierdzenie ma zaś na celu udowodnić poprawność metody obliczania braków w algorytmie. Mówi ono, że możemy zgodnie z opisem podzielić okno czasowe z pomocą początków i końców projektów i zastosować na nich algorytm podstawowy.

Dowód. Założenie poczynione w definicji zadania, mówiące o tym, że eksperci mogą zmieniać projekty po zakończeniu jednostki czasu, zapewnia, że przyporządkowanie eksperta e do podzadania (u, p) w chwili t nie ma wpływu na jego przyporządkowanie w żadnej innej chwili. Wobec tego, na mocy twierdzenia 2 oraz możemy podzielić przyporządkowanie assign na D przyporządkowań chwilowych, gdzie D jest dostępną długością okna czasowego. Wówczas uzyskujemy

$$M(\text{assign}) = \sum_{i=0}^{D-1} M(\text{assign}'_i)$$

gdzie assign'_i jest optymalnym przyporządkowaniem chwilowym w chwili i.

Zauważmy jednak, że może się zdarzyć, że w chwilach t i $t+1$ wykonywane są te same projekty (tj. dla każdego projektu p_i zachodzi $t_i \leq t \leq t+1 < t_i + d_i$). Wówczas, algorytm podstawowy zostanie wykonany dwukrotnie dla tych samych danych wejściowych; z optymalności algorytmu podstawowego otrzymujemy więc

$$M(\text{assign}'_t) = M(\text{assign}'_{t+1})$$

Sytuacje opisane powyżej mają miejsce dla wszystkich przyporządkowań chwilowych w przedziałach $[e_i, e_{i+1})$, $i = 0, 1, \ldots, k-1$, a więc wobec powyższej równości otrzymujemy

$$M(\text{assign}) = \sum_{i=0}^{k-1} M(\text{assign}'_{e_i}) \cdot (e_{i+1} - e_i)$$

□

2.5. Oszacowanie złożoności czasowej

Określając złożoność algorytmu musimy wykonać kilka spostrzeżeń:

1. Ilość iteracji wykonania algorytmu genetycznego jest z góry ograniczona ustalonym parametrem - nazwijmy go MAX_{IT}. Pozostałe warunki stopu jedynie zmniejszają tę wartość.

2. Operacja krzyżowania wykonuje się w czasie $O(|P|)$, podobnie jak operacja mutacji.

3. Dla stałej MAX_{POP} określającej największy rozmiar populacji, wybór osobników przy zastosowaniu metody koła ruletki odbędzie się w czasie $O(MAX_{POP})$. Pesymistycznie trzeba założyć, że metoda ta będzie stosowana w każdej iteracji algorytmu.

4. Złożoność funkcji rozwiązującej podproblem dla danego przedziału jest równa złożoności rozwiązania podstawowego czyli $O((|E|+|S|+|P|)^3)$ (uzasadnienie w sekcji 2.5.1.)

5. Z poprzedniego punktu wynika, że czas wyznaczenia wartości funkcji przystosowania dla chromosomu jest rzędu $O(D \cdot (|E| + |S| + |P|)^3)$

6. Wyznaczenie elementu o najwyższej wartości funkcji przystosowania zajmie $O(MAX_{POP})$.

Zatem w każdej iteracji najbardziej kosztownym obliczeniem będzie wyznaczenie wartości funkcji przystosowania dla każdego elementu nowej populacji. Czas wykonania tej operacji jest co najwyżej rzędu $O(MAX_{POP} \cdot D \cdot (|E| + |S| + |P|)^3)$, a czas wykonania całego algorytmu:

$$O(MAX_{IT} \cdot MAX_{POP} \cdot D \cdot (|E| + |S| + |P|)^3)$$

2.5.1. Złożoność problemu podstawowego

Uzasadnienie, że złożoność czasowa rozwiązania problemu podstawowego jest rzędu $O((|E|+|S|+|P|)^3)$ wymaga zauważenia dwóch faktów. Oznaczmy graf reprezentujący pewne zadanie problemu podstawowego jako G, G_E będzie zbiorem jego krawędzi, a G_V zbiorem wierzchołków.

1. W każdej iteracji algorytmu Edmondsa-Karpa (czyli po znalezieniu ścieżki rozszerzającej w czasie $O(|G_E|)$) monotonicznie zwiększa się wartość bieżącego przepływu. Fakt ten wynika bezpośrednio z definicji ścieżki rozszerzającej.

2. Ograniczeniem górnym na wartość przepływu jest liczba ekspertów. Wynika to z konstrukcji grafu w problemie podstawowym - źródło jest połączone jedynie z wierzchołkami odpowiadającymi ekspertom. Ponadto, są to krawędzie o przepustowości 1. Zatem największą możliwą wartością przepływu jest $|E|$.

Fakt pierwszy stanowi przypomnienie ogólnej koncepcji algorytmów opartych o pomysł Forda-Fulkersona. Fakt drugi nakłada ograniczenie na wartość przepływu czyli liczbę iteracji algorytmu lub też na liczbę wykonań operacji wyszukiwania ścieżki rozszerzającej. Ponadto liczbę ekspertów możemy ograniczyć przez liczność zbioru wierzchołków grafu. Ostatecznie otrzymujemy:

$$O(|G_E|) \cdot O(|E|) = O(|G_V|^2) \cdot O(|G_V|) = O(|G_V|^3) = O((|E| + |S| + |P|)^3)$$

3. Spis zawartości załączonej płyty CD

Zgodnie z wymaganiami przedmiotu, podczas oddawania ostatniego etapu projektu do dokumentacji końcowej została załączona płyta CD. Opisane w niniejszej dokumentacji algorytmy zostały zaimplementowane w języku skryptowym Python, więc nie zostały załączone skompilowane pliki wykonywalne.

W skład rzeczy zamieszczonych na dołączonej płycie CD wchodzi:

- wersja elektroniczna niniejszej dokumentacji w pliku `dokumentacja.pdf` znajdująca się w katalogu `/doc`,
- kod źródłowy programu rozwiązującego podstawową wersję problemu (wraz z przykładowym wejściem) znajdujący się w katalogu `/src/lab2`,
- kod źródłowy programu rozwiązującego rozszerzoną wersję problemu (wraz z przykładowym wejściem) znajdujący się w katalogu `/src/lab3`.

Pliki tekstowe zawierające przykładowe wejścia dla podstawowej i rozszerzonej wersji problemu noszą nazwę `input.txt` i znajdują się w głównych katalogach projektów, czyli odpowiednio `/src/lab2` i `/src/lab3`.

Plik wejściowy dla podstawowej wersji problemu powinien zawierać dodatnią liczbę umiejętności (ozn. `u`), nieujemną liczbę ekspertów (ozn. `e`) i nieujemną liczbę projektów (ozn. `p`) rozdzielone przecinkami w pierwszej linii. Kolejne `e` linii powinno zawierać wektory umiejętności ekspertów (binarne), a następujące po nich `p` linii – wektory zapotrzebowań projektów (naturalne). Wszystkie wektory powinny być długości `u` i zawierać elementy rozdzielone przecinkami.

Plik wejściowy dla rozszerzonej wersji problemu został poszerzony o czwarty parametr w pierwszej linii – dodatnią liczbę jednostek czasu przeznaczoną na wykonanie wszystkich projektów (ozn. `D`). Wektory zapotrzebowania projektów zostały rozszerzone o `u+1` element, którym jest czas trwania danego projektu w jednostkach czasu (ozn. `d`). Ponadto powinno zachodzić $0 < \texttt{d} \leq \texttt{D}$.

Przed uruchomieniem należy się upewnić, że na komputerze znajduje się interpreter Pythona w wersji 3.x i moduł `networkx`, który dostarcza graf skierowany wykorzystywany w przygotowanych rozwiązaniach. Bibliotekę `networkx` można pobrać przy pomocy Pythonowego menedżera paczek `pip` wpisując w konsoli polecenie:

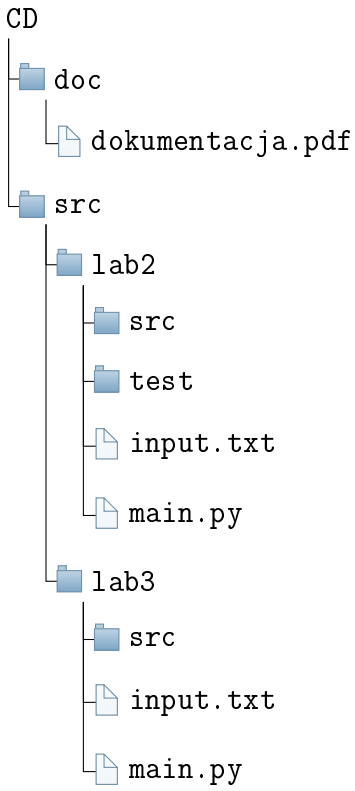
```
$ pip3 install networkx
```

Wówczas możliwe będzie uruchomienie każdego z programów poleceniem

```
$ python3 main.py input.txt
```

wywołanym katalogu `/src/lab2` lub `/src/lab3`.

Poniższy schemat ilustruje strukturę zawartości płyty CD.



Literatura

[1] Sean Luke, *Essentials of Metaheuristics*, Lulu, second edition, 2013, dostępne bezpłatnie pod adresem `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[2] Matthew Bartschi Wall, *A Genetic Algorithm for Resource-Constrained Scheduling*, Massachusetts Institute of Technology, 1996.