Dokumentacja programu CPP

Spis treści:

1. Opis problemu

W królestwie rządzącym królewna Śnieżka i książę, liczba krasnoludków stale wzrasta. Królewna jest odpowiedzialna za przydział pracy dla wszystkich krasnoludków, uwzględniając ich specjalizacje w wydobyciu różnych minerałów. Każdy krasnoludek jest ekspertem w określonym zawodzie i może wydobywać różne minerały. Królewna zna preferencje i umiejętności każdego krasnoludka oraz posiada informacje o położeniu złóż mineralnych w królestwie i ich wydajności. Celem królewny jest zapewnienie obywatelom wystarczającej ilości surowców, które są niezbędne do rozwoju królestwa. Aby to osiągnąć, musi dokonać odpowiedniego przydziału pracy dla krasnoludków, biorąc pod uwagę ich preferencje oraz wydajność poszczególnych złóż. Dodatkowo, książę zauważył, że jego ukochana spędza wiele czasu w kuchni, gotując owsiankę dla krasnoludków. Okazało się, że ilość potrzebnej owsianki zależy od odległości, jaką muszą pokonać krasnoludki, aby dotrzeć z domu do miejsca pracy. Im dłuższa odległość, tym więcej owsianki jest potrzebne. Książę chciałby zminimalizować sumaryczną odległość podróży krasnoludków, aby zmniejszyć ilość owsianki potrzebnej do gotowania. Jednak królewna wprowadziła zakaz spożywania jabłek w królestwie po incydencie ze złą królową. Nakazała wyciąć wszystkie jabłonie, a spożywanie owoców jest surowo zabronione. Książę, mając obowiązek egzekwowania tego zakazu, codziennie musi obchodzić wszystkie użytkowane kopalnie, aby upewnić się, że nie ma naruszeń. Chciałby, aby codziennie pokonywana odległość była jak najmniejsza. Rozwiązanie tego problemu polega na opracowaniu programu, który na podstawie preferencji krasnoludków, wydajności złóż, odległości między wyrobiskami a domkami krasnoludków oraz ograniczeń związanych z owsianką i jabłkami, dokona optymalnego przydziału pracy dla krasnoludków, minimalizując sumaryczną odległość podróży i zachowując wartość produkowanych dóbr. Program powinien uwzględniać preferencje krasnoludków, wydajność złóż, odległości między miejscami pracy a domkami, a także uwzględniać zakaz spożywania jabłek na terenie królestwa i zmniejszać odległość podróży księcia w celu egzekwowania tego zakazu. Wynikiem działania programu powinien być optymalny przydział pracy dla krasnoludków, minimalizujący sumaryczną odległość podróży, przy zachowaniu wartości produkowanych dóbr i uwzględnieniu ograniczeń związanych z owsianką i jabłkami.

2. Opis rozwiązania:

Program rozwiązuje problem optymalnego przydziału pracy dla krasnoludków w celu zapewnienia wystarczającej ilości surowców dla królestwa. Program uwzględnia preferencje i umiejętności każdego krasnoludka oraz informacje o wydajności złóż mineralnych i ich położeniu. Celem programu jest minimalizacja sumarycznej odległości podróży krasnoludków oraz minimalizacja odległości podróży księcia, który ma egzekwować zakaz spożywania jabłek.

1. Tworzy się obiekty klasy Hungarian o nazwie dwarfs_to_ores i optima_distances.
2. Obiekt dwarfs_to_ores jest tworzony na podstawie danych dotyczących krasnoludków i kopalń.
3. Wywołuje się metodę make_assignment() na obiekcie dwarfs_to_ores, która wykonuje optymalne przypisanie pracy.
4. Wyświetla się wynik przypisania poprzez wyświetlenie obiektu dwarfs_to_ores.
5. Pobiera się zbiór zer z obiektu dwarfs_to_ores.
6. Obiekt optima_distances jest tworzony na podstawie zbioru zer, danych dotyczących krasnoludków i kopalń.
7. Wywołuje się metodę make_assignment() na obiekcie optima_distances, która wykonuje optymalne przypisanie pracy.
8. Wyświetla się wynik przypisania poprzez wyświetlenie obiektu optima_distances.

Ostatecznym wynikiem działania programu jest optymalny przydział pracy dla krasnoludków, minimalizujący sumaryczną odległość podróży i uwzględniający zakaz spożywania jabłek.

3. Struktura programu

Program można podzielić na następujące pliki:

1. Presentation.cpp: Plik główny programu, zawierający funkcję main. W tym pliku inicjalizowane są dane dotyczące krasnoludków i kopalń, tworzone są obiekty klasy Hungarian i wywoływane metody do optymalnego przypisania pracy i wyświetlenia wyników.
2. Katalog HungarianClass zawierający plik nagłówkowy hungarian.h, hungarian.cpp oraz types.h: W tych plikach znajdują się deklaracje klas Hungarian, mine_t i dwarf_t, które reprezentują struktury danych oraz metody z nimi związane.
3. brut_force.cpp: zawiera funkcję brute_force, która wykonuje przeszukiwanie brutalne w celu znalezienia optymalnych indeksów w macierzy kosztów. Program przyjmuje macierz kosztów jako argument i zwraca parę wektorów: optimal_row_ind (optymalne indeksy wierszy) i optimal_col_ind (optymalne indeksy kolumn).

4. Opis klas i funkcji:

Funkcja `brute_force`: - Opis: Funkcja wykonuje przeszukiwanie brutalne w celu znalezienia optymalnych indeksów w macierzy kosztów. - Parametry: - `cost_matrix`: Macierz kosztów, na podstawie której obliczane są optymalne indeksy. - `maximize` (opcjonalny): Określa, czy należy maksymalizować koszt (domyślnie: false). - Zwraca: - Para wektorów zawierających optymalne indeksy wierszy (first) i kolumn (second).

Funkcja `main`: - Opis: Funkcja główna programu. Wykorzystuje funkcję `brute_force` do znalezienia optymalnych indeksów w macierzy kosztów. Wyświetla wynik, czyli zestawienie optymalnych indeksów oraz minimalny koszt osiągnięty dla tych indeksów. - Zwraca: Wartość 0 oznaczająca poprawne zakończenie programu.

Klasa `mine_t`: - `MineTypes`: Typ wyliczeniowy definiujący różne typy kopalń (zakomentowany w kodzie, ale nie używany). - `type`: Typ kopalni. - `position`: Para liczb zmiennoprzecinkowych reprezentujących pozycję kopalni.

Klasa `dwarf_t`: - `position`: Para liczb zmiennoprzecinkowych reprezentujących pozycję krasnoluda. - `skills`: Mapa typów kopalń i umiejętności krasnoluda w tych kopalniach.

Funkcja `main()`: - Tworzy wektor `ores` zawierający obiekty `mine_t` reprezentujące kopalnie. - Tworzy wektor `dwarfs` zawierający obiekty `dwarf_t` reprezentujące krasnoludów. - Tworzy obiekt klasy `Hungarian` o nazwie `dwarfs_to_ores` z wykorzystaniem krasnoludów i kopalń. - Wyświetla obiekt `dwarfs_to_ores` na standardowym strumieniu wyjścia. - Wywołuje metodę `make_assignment()` na obiekcie `dwarfs_to_ores` w celu wykonania przypisania. - Ponownie wyświetla obiekt `dwarfs_to_ores` na standardowym strumieniu wyjścia. - Pobiera zbiór zer z obiektu `dwarfs_to_ores`. - Tworzy obiekt klasy `Hungarian` o nazwie `optima_distances` z wykorzystaniem zbioru zer, krasnoludów i kopalń. - Wywołuje metodę `make_assignment()` na obiekcie `optima_distances` w celu wykonania przypisania. - Wyświetla obiekt `optima_distances` na standardowym strumieniu wyjścia.

Klasa Hungarian: Główna klasa implementująca algorytm węgierski. Zawiera metody do rozwiązywania problemów przypisania, aktualizacji macierzy kosztów i innych operacji pomocniczych.

Funkcje składowe klasy Hungarian:

subtract_min_col: Odejmuje minimum z każdej kolumny w macierzy kosztów.
subtract_min_row: Odejmuje minimum z każdego wiersza w macierzy kosztów.
mark_starred_zeros: Oznacza zerowe elementy w macierzy kosztów gwiazdkami.
check_cover_is_min: Sprawdza, czy pokrycie jest minimalne.

exists_uncovered_zero: Sprawdza, czy istnieje niepokryte zero w macierzy kosztów.
zero_in_row: Sprawdza, czy w danym wierszu istnieje zero z odpowiednim oznaczeniem.
zero_in_col: Sprawdza, czy w danej kolumnie istnieje zero z odpowiednim oznaczeniem.
augment_path: Powiększa ścieżkę.

clear_covers: Usuwa pokrycia.

erase_primes: Usuwa oznaczenia typu Prime.

find_min_uncovered: Znajduje najmniejszą niepokrytą wartość w macierzy kosztów.
step_four: Wykonuje czwarty krok algorytmu węgierskiego.

step_five: Wykonuje piąty krok algorytmu węgierskiego.

update_costs_with_min: Aktualizuje koszty w macierzy za pomocą minimalnej wartości.
distance: Oblicza odległość między dwoma punktami na płaszczyźnie.

5. Instrukcje użytkowania:

Przygotowanie danych: Zdefiniuj wektor krasnoludów (std::vector<dwarf_t>) reprezentujących krasnoludów. Dla każdego krasnoluda zdefiniuj pozycję (x, y) oraz umiejętności w postaci mapy skills, gdzie kluczem jest typ kopalni (enum MineTypes) a wartością jest umiejętność (liczba zmiennoprzecinkowa). Zdefiniuj wektor kopalń (std::vector<mine_t>) reprezentujących kopalnie. Dla każdej kopalni zdefiniuj typ (enum MineTypes) oraz pozycję (x, y).

Utworzenie obiektu klasy Hungarian: Skorzystaj z jednego z konstruktorów klasy Hungarian: Hungarian(std::vector<dwarf_t>& dwarfs, std::vector<mine_t>& ores): Przekazuje referencję do wektora krasnoludów i wektora kopalń. Hungarian(float matrix[], size_t size): Przekazuje tablicę i rozmiar, gdzie tablica zawiera koszty przypisania. Hungarian(std::set<std::pair<size_t, size_t>> const& initial_assignement, std::vector<dwarf_t> const& dwarfs, std::vector<mine_t> const& ores): Przekazuje początkowe przypisanie, wektor krasnoludów i wektor kopalń.

Wykonanie algorytmu przypisania: Wywołaj metodę make_assignment() na obiekcie Hungarian. Metoda make_assignment() zwraca wektor par (std::vector<std::pair<int, int>>) reprezentujących przypisanie krasnoludów do kopalń.

Odczytanie wyników: Jeśli wykonano krok 3 i otrzymano wektor przypisań, można skorzystać z metody get_assignemnt() na obiekcie Hungarian, aby odczytać przypisanie. Metoda get_assignemnt() zwraca wektor par (std::vector<std::pair<int, int>>) reprezentujących przypisanie krasnoludów do kopalń.

Obliczenie kosztu przypisania: Można skorzystać z metody calc_cost() na obiekcie Hungarian, aby obliczyć koszt przypisania. Metoda calc_cost() zwraca koszt przypisania jako wartość zmiennoprzecinkową.

Dodatkowe funkcje: Metoda get_zeros() na obiekcie Hungarian zwraca zbiór par (std::set<std::pair<size_t, size_t>>) reprezentujących pozycje zer w macierzy kosztów.

6. Testy:

Algorytm brute force, który został zaimplementowany w podanym kodzie, przegląda wszystkie możliwe kombinacje przypisań krasnoludów do kopalń w celu znalezienia optymalnego rozwiązania. Działa to poprzez generowanie wszystkich permutacji indeksów wierszy i kolumn macierzy kosztów, a następnie obliczanie kosztu dla każdej permutacji. Po przeglądnięciu wszystkich możliwości, algorytm wybiera permutację o najniższym koszcie. W pierwszym teście, gdzie macierz kosztów była stosunkowo mała (rozmiar 5x5), algorytm znalazł idealne dopasowanie, które polegało na przypisaniu każdemu wierszowi i kolumnie indeksu odpowiadającemu ich pozycji w macierzy kosztów. W rezultacie koszt wyniósł 0, co oznacza, że nie było kosztów przypisania. Jednakże, dla większej macierzy kosztów w drugim teście (rozmiar 8x8), algorytm brute force musiał przeglądać znacznie większą liczbę permutacji, co prowadzi do wydłużonego czasu działania. W tym przypadku, algorytm znalazł kombinację, która minimalizowała koszt przypisania. Wynik wskazuje, że krasnoludy zostały przypisane do kopalń w sposób, który minimalizuje całkowity koszt, a wynikowy koszt wyniósł 6.

Test 1:

```
std::vector<std::vector<int>> cost_matrix = {
    {0, 0, 0, 3, 3},
    {2, 2, 2, 0, 0},
    {0, 0, 0, 3, 3},
    {5, 5, 5, 5, 5},
    {5, 5, 5, 5, 5},
};
```

Result:

[0,0]

[1,1]

[2,2]

[3,3]

[4,4]

min cost = 0

Wynik testu wskazuje, że najlepsze dopasowanie zostało znalezione poprzez przypisanie każdemu wierszowi i kolumnie indeksu odpowiadającemu ich pozycji w macierzy kosztów. Koszt wynosi 0, co oznacza, że nie ma kosztów przypisania.

Test 2:

```
std::vector<std::vector<int>> cost_matrix = {
    {0, 0, 0, 3, 3, 6, 3, 3},
    {2, 2, 2, 0, 0, 3, 5, 9},
    {0, 0, 0, 3, 3, 4, 5, 5},
    {5, 5, 5, 5, 5, 1, 1, 5},
    {5, 5, 5, 5, 5, 0, 0, 3},
    {1, 0, 8, 2, 2, 7, 4, 2},
    {9, 1, 2, 3, 7, 8, 4, 4},
    {4, 7, 2, 3, 3, 1, 0, 5}
};
```

Result:

[0,0]

[1,3]

[2,2]

[3,1]

[4,4]

[5,7]

[6,6]

[7,5]

min cost = 6

Wynik testu wskazuje, że najlepsze dopasowanie zostało znalezione poprzez przypisanie odpowiednich wierszy i kolumn w celu minimalizacji kosztu. Koszt wynosi 6.