General Scheduling Problem

Realizacja na podstawie Simulated Annealing

Spis Treści:

1 Wstęp 1
2 Architektura programu 1
3 General Scheduling Problem 2
3.1 Podstawowe Klasy 2
3.2 Funkcja Main 2
3.3 Algorytm przydziału prac 3

Rozdział 1

Wstęp

General Scheduling Problem jest zagadnieniem dotyczącym problematyki przydziału prac (zadań) poszczególnym procesom, który na podstawie zadanych prac i ilości procesów ma za zadanie wyznaczyć najlepszą kolejność rozwiązywania zadań przez poszczególne procesy tak, by czas, jaki należy czekać na zrealizowanie wszystkich zadań był jak najkrótszy (optymalny).

Cały program składa się z trzech części i został zrealizowany w podobny sposób jak program dotyczący zagadnienia „Travelling Salesman". Pierwsza część programu ma za zadanie przygotować potrzebne dane dla algorytmu symulowanego wyżarzania (Simulated Annealing). Drugą częścią programu jest już sam algorytm symulowanego wyżarzania, który ma za zadanie znaleźć optymalne rozwiązanie problemu. W trzeciej części programu można zobaczyć animację pracy procesów dzień po dniu dla najlepszych podziałów prac.

W dokumentacji można znaleźć tylko omówienie pierwszej części programu, ponieważ algorytm symulowanego wyżarzania został omówiony w osobnym projekcie a trzecia część jest tylko dodatkiem ułatwiającym korzystanie z programu niewymagająca komentarza.

Rozdział 2

Architektura programu

Ze względu na mnogość linii kodu, zdecydowałem się podzielić program na klasy umieszczone w osobnych plikach. Każda z tych klas zawiera posegregowane komponenty należące do odpowiednich etapów w programie. Wyróżniłem 8 klas (osobnych plików).:

1. AllClasses – plik zawierający zbiór używanych przeze mnie klas na potrzebę pierwszej części programu.

2. CreatingComponents – klasa zawierająca krótkie metody potrzebne do utworzenia poszczególnych list w części Main programu.

3. Parameters – klasa zawierająca statyczne zmienne sterujące programem. Jest to ważna klasa ponieważ zmienne w niej się znajdujące sterują całym programem. Z tego powodu omówię pokrótce każdą z nich:

3a) alphabet – tablica znaków zawierająca uproszczony alfabet, służąca do przypisania nazw odpowiednim zadaniom

3b) names – tablica łańcuchów zawierająca nazwy fabryk, służąca do przypisania nazw odpowiednim procesom (fabrykom)

3c) Duration – tablica czasów trwania poszczególnych zadań. Proszę zauważyć, że z logicznego punktu widzenia nie może być ona większa niż pojemność tablicy alphabet

3d) N – zmienna pobierająca informację o wielkości tablicy Duration

3e) K – zmienna z informacją o ilości procesów (fabryk) używanych w programie. Proszę zauważyć, że z logicznego punktu widzenia nie może być ona większa niż pojemność tablicy names

3f) animation – zmienna służąca do prezentacji pracy przez procesy w trzeciej części programu

4. Permutation – klasa zawierająca metody potrzebne do przeprowadzenia procesu permutacji przy zadanym łańcuchu znaków, które mają mu ulec oraz kilku innych danych sterujących (możliwości dokonania permutacji dla wyciętego podłańcucha)

5. PrintingMethods – klasa zawierająca metody odpowiedzialne za animowaną prezentację wyników w trzeciej części programu

6. Program – klasa zawierająca startową funkcję rozpoczynającą Main programu

7. SchedulingAlgorithm – klasa zawierająca algorytm przydziału prac do poszczególnych procesów (fabryk) oraz pomocnicze mu metody

8. SimulatedAnnealing – klasa zawierająca wszystkie metody algorytmu symulowanego wyżarzania używanego w drugiej części programu

Rozdział 3

General Scheduling Problem

3.1 Podstawowe Klasy

W tym miejscu pozwolę sobie omówić komponenty, które stworzyłem na potrzeby pierwszej części programu.:

Klasa Task:

```
Task: {
  Job
  Duration
}
```

Klasa ta zbiera informacje na temat zadań wprowadzanych w modyfikowalnych zmiennych statycznych w klasie Parameters.

Job – nazwa zadania
Duration – czas trwania zadania (określany w dniach)

Klasa Way:

```
Way: {
  Combination
  Worktime
  Index
  History
}
```

Klasa ta zbiera informacje na temat określonej ścieżki zadań rozwiązywanych przez procesy.

Combination – ściśle określona kolejność wykonywania zadań
Worktime – czas trwania danej ścieżki (podejścia)
Index – sztywno przypisany indeks, który na wzór rozwiązania problemu „Travelling Salesman" pełni rolę argumentu dla funkcji „f" z części Simulated Annealing
History – historia podejścia, która jest swojego rodzaju animacją wyświetlaną w części trzeciej programu.

Klasa Factory:

```
Factory: {
  Job
  Duration
  Name
  IsWorking
  Progress
}
```

Klasa ta zbiera informacje o całym procesie produkcyjnym dla jednego procesu (fabryki – stąd nazwa). Programistycznie dziedziczy ona pola „Job" i „Duration" z klasy Task. Jest to bardzo pomocny zabieg sprzyjający prostocie kodu w późniejszej części programu. Odziedziczonych pól nie będę zatem omawiać.

Name – nazwa fabryki (coś co rozróżni jedną fabrykę od innej)
IsWorking – informacja czy dana fabryka pracuje w jednostce czasu
Progress – informacja o postępie pracy w przypadku pracującej fabyrki

3.2 Funkcja Main

W tym miejscu omówię cały proces przygotowania danych do pracy w drugiej części programu. Zajmiemy się linijkami kodu odpowiadającymi za pierwszą część programu w klasie Program.

1. Tworzę listę zadań, którą uzupełnię później danymi ze zmiennych statycznych z klasy Parameters (alphabet, Duration).

2. Tworzę listę kombinacji (permutacji) – listę stringów którą uzupełnię danymi na podstawie listy wprowadzonych zadań.

3. Tworzę listę ścieżek (sposobów kolejno rozwiązywanych zadań), którą uzupełnię odpowiednimi danymi.

4. Tworzę listę najlepszych ścieżek, która zostanie uzupełniona po pracy drugiej części programu.

5. Tworzę listę procesów (fabryk), która zawierać będzie wszystkie niezbędne informacje na temat fabryk i ich pracy.

6. Uzupełniam listę zadań danymi ze zmiennych statycznych za pomocą metody z klasy CreatingComponents.

7. Uzupełniam listę permutacji za pomocą metod z klasy Permutation.

8. Uzupełniam listę fabryk danymi o ich nazwach za pomocą metody z klasy CreatingComponents. Na tą chwilę mam fabryki gotowe do pracy. Nie mają jeszcze żadnych przydzielonych zadań oraz czekają na pracę ze zmienną „IsWorking" ustawioną na false.

9. Rozpoczynam pętlę, która będzie miała za zadanie Uzupełnić listę ścieżek bazując na konkretnych metodach z odpowiednich klas.

10,11. Tworzę tymczasową zmienną typu Way, która będzie dodawana kolejno w pętli do listy ścieżek.

12. Zeruję zmienną statyczną „animation" - zabieg ten istnieje w tym miejscu tylko i wyłącznie ze względu na strukturę programu oraz chęć otrzymania animacji przedstawiającej przebieg pracy fabryk dzień po dniu w trzeciej części programu.

13. Zmienną tymczasową typu Way uzupełnioną o prawidłowe dane dodaję do listy ścieżek.

Realizując program w ten sposób uzyskam potrzebne dane do pracy w drugiej części programu. Proszę zauważyć, że to lista ścieżek (ListOfWays) stanie się właśnie tą pseudo funkcją na której będzie bazować algorytm symulowanego wyżarzania. Spójrzmy na jej strukturę:

Ways[0]: { Combination: ABCDEF, Worktime: 16, index: 0, History: „ … ” }

Ways[1]: { Combination: ABCDFE, Worktime: 17, index: 1, History: „ … ” }

Ways[2]: { Combination: ABCEDF, Worktime: 16, index: 2, History: „ … ” }

Ways[3]: { Combination: ABCEFD, Worktime: 16, index: 3, History: „ … ” }

Ways[4]: { Combination: ABCFDE, Worktime: 15, index: 4, History: „ … ” }

…

Mamy do czynienia z pewnego rodzaju funkcją o argumentach należących do zbioru liczb naturalnych oraz odpowiednimi dla nich wartościami w zmiennej „Worktime”.

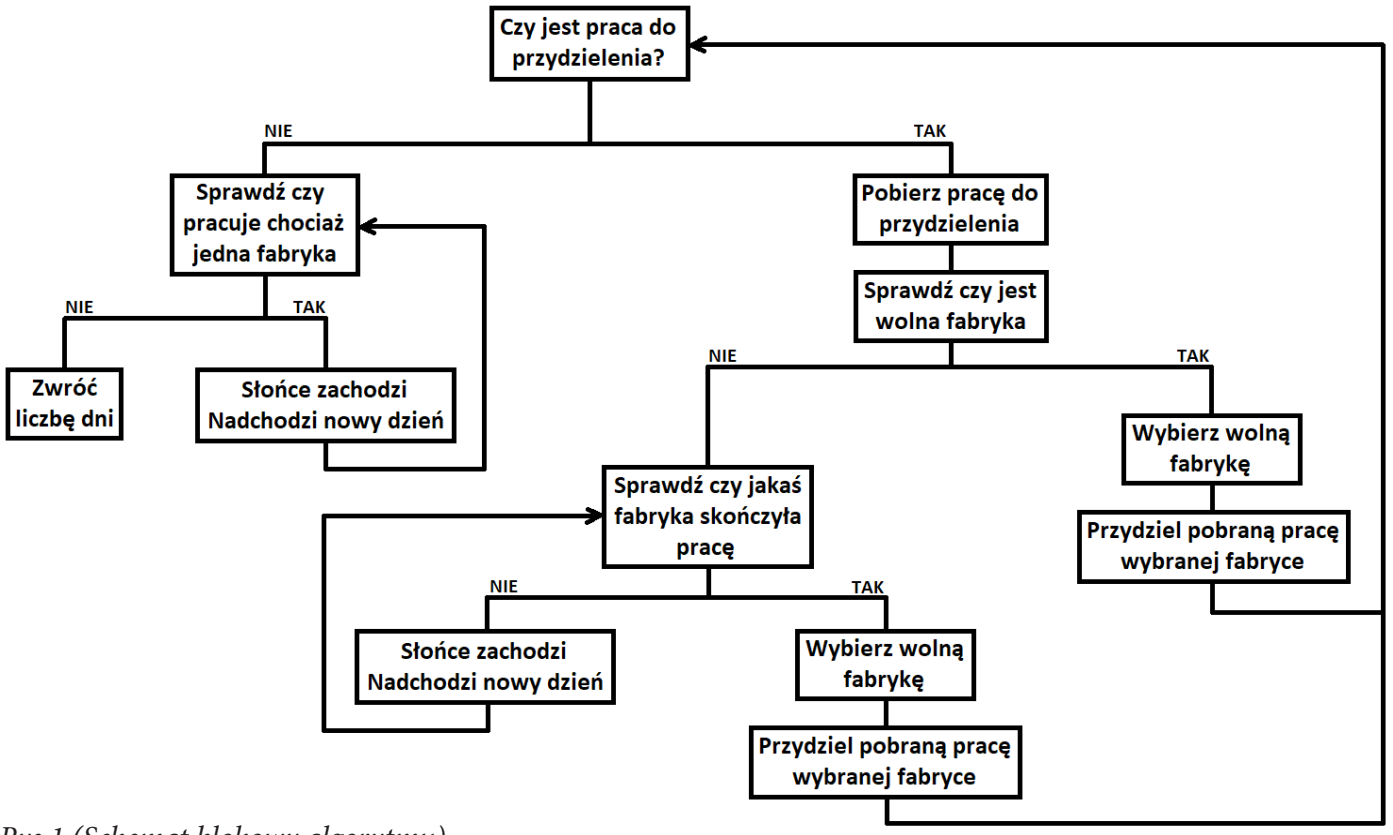
3.3 Algorytm przydziału prac

Algorytm ten jest tak naprawdę esencją całego projektu, ponieważ to on odpowiada za poruszane zagadnienie, które należało zaimplementować. Z poprzednich rozdziałów wiemy, że znajduje się on w klasie SchedulingAlgorithm. Wymagał on starannego zaprojektowania i przemyślenia komponentów tak, aby cały algorytm mógł być przejrzysty, zgrabny i nie wymagający wielu nieczytelnych linii kodu a mogący się składać z dobrze współgrających ze sobą komponentów mających za zadanie wykonanie tylko ściśle określonych kawałków czynności. Algorytm ten tak jak Simulated Annealing ma za zadanie z grupy komponentów zapewnić szybki proces otrzymania pożądanego rezultatu jakim jest wartość zmiennej Worktime dla każdej ścieżki z listy dróg zmiennej ListOfWays.

Chcąc sprostać problemowi przydziału prac można zauważyć, że już po kilku próbach na kartce papieru da się wyodrębnić pewne stałe czynności, które trzeba wykonywać, by wszystko przebiegało sprawnie. Załóżmy, że rysujemy sobie na kartce papieru przykładowo dwie fabryki i listę zadań, które kolejno mają być wykonywane (np. ABFEDC). Czynności, które będziemy musieli wykonać będą następujące: (*Rys 1*)

Polecam przeprowadzić dany schemat blokowy na kartce papieru i sprawdzić jego skuteczność. Widoczne bloczki są nazwane w taki sposób, że opisują co mają zrobić. Postarałem się zastąpić je odpowiednimi nazwami metod, które będą odwzorowaniem ich czynności.

Czy jest praca do przydzielenia – IsThereAnyWork()
Pobierz pracę do przydzielenia – PullWork()
Sprawdź czy jest wolna fabryka – IsThereALazyFactory()
Wybierz wolną fabrykę – GetFreeFactory()
Przydziel pobraną pracę wybranej fabryce – zadanie realizowane w procesie AssignDuration



Rys 1 (Schemat blokowy algorytmu)

Sprawdź czy jakaś fabryka skończyła pracę – CheckFactories()

Słońce zachodzi. Nadchodzi nowy dzień – Sunset()

Sprawdź czy pracuje chociaż jedna fabryka – IsThereAnyLazyFactory()

Możemy zatem naliczyć 7 metod potrzebnych do realizacji procesu AssignDuration. Każdą z metod można obejrzeć w klasie SchedulingAlgorithm. Poniżej zamieszczam wersję schematu blokowego w postaci uproszczonego kodu C# bez linii odpowiedzialnych za podłoże do trzeciej części programu. (*Rys 2*)

Proszę zauważyć, że informacje dla poszczególnych komponentów są udzielane lub ustanawiane na podstawie listy fabryk z aktualnymi informacjami o ich stanie, etapie w pracy itp. i aktualnej listy kolejności wykonywania prac (zmienna works). Każdy z 7 komponentów można przeanalizować w kodzie wykorzystując informacje o specyfice działania każdego z nich umieszczonych w tym dokumencie.

```
1.  while (IsThereAnyWork(works))
2.  {
3.      work = PullWork(works, listT);
4.      if (IsThereALazyFactory(listF))
5.      {
6.          Freefactory = GetFreeFactory(listF);
7.          Freefactory.job = work.job;
8.          Freefactory.duration = work.duration;
9.          Freefactory.progress = 0;
10.         Freefactory.isWorking = true;
11.     }
12.     else
13.     {
14.         while (!CheckFactories(listF))
15.         {
16.             Sunset(listF);
17.             day++;
18.         }
19.         Freefactory = GetFreeFactory(listF);
20.         Freefactory.job = work.job;
21.         Freefactory.duration = work.duration;
22.         Freefactory.progress = 0;
23.         Freefactory.isWorking = true;
24.     }
25. }
26. while (IsThereAnyLazyFactory(listF))
27. {
28.     Sunset(listF);
29.     day++;
30. }
31. return day;
```

Przydzielanie pracy

Przydzielanie pracy

Rys 2 (Uproszczony kod algorytmu C#)