Projektowanie efektywnych algorytmów

Projekt

Kasper Radom (264023) *Algorytm Tabu Search i Symulowanego Wyżarzania*

1. Wstęp

Zadanie polegało na implementacji i zbadaniu algorytmów przeszukiwania z zakazami oraz symulowanego wyżarzania. Oba algorytmy można opisać jako metaheurystyczne. Algorytm metaheurystyczny to rodzaj algorytmu optymalizacyjnego, który jest zdolny do rozwiązywania problemów optymalizacyjnych poprzez eksplorację przestrzeni rozwiązań w sposób bardziej ogólny niż tradycyjne metody. Metaheurystyki są często używane do rozwiązywania problemów trudnych obliczeniowo, gdzie podejście dokładne jest nieefektywne lub niemożliwe do zastosowania w akceptowalnym czasie.

Z owymi algorytmami wiążą się elementy które warto tu zdefiniować:

- **Sąsiedztwo** – jest to zbiór rozwiązań sąsiednich dla obecnego stanu. Rozwiązanie sąsiednie powinno różnić się nieznacznie od swoich sąsiadów. Możliwe jest zdefiniowanie wielu rodzajów sąsiedztwa, poniżej zaimplementowane zostały jednak wyłącznie trzy. Każda metoda generowania sąsiedztwa pobiera dwa argumenty i na ich podstawie generuje nowe rozwiązanie.
    - Metoda *doInsert*:

```
void Ts::doInsert(int where, int from){
    vector<int> newPath;
    for(int i = 0;i<testPath.size()-1;i++){
        if(i==where) newPath.push_back(testPath[from]);
        else if(i == from)continue;
        newPath.push_back(testPath[i]);
    }
    newPath.push_back(newPath[0]);
    testPath = newPath;
}
```

Metoda ta pobiera element listy o indeksie *from* oraz umieszcza go między elementami o indeksach *where -1* oraz *where*.

- Metoda *doSwap*:

```
void Ts::doSwap(int indexFirst, int indexSecond){
    swap( &: testPath[indexFirst], &: testPath[indexSecond]);
    testPath[testPath.size()-1]=testPath[0];
}
```

Funkcja ta działa podobnie do poprzedniej. Z tą różnicą, że element nie jest umieszczany pomiędzy dwoma innymi, a zamieniany miejscem z innym elementem.

- Metoda *doInvert*:

```
void Ts::doInvert(int indexSmaller, int indexBigger){
    vector<int> inverted;
    vector<int> newPath;
    for(int i = indexBigger; i >=indexSmaller; i--){
        inverted.push_back(testPath[i]);
    }
    for(int i=0; i<testPath.size()-1; i++){
        if(i<=indexBigger and i>=indexSmaller) {
            newPath.push_back(inverted[i-indexSmaller]);
        }else{
            newPath.push_back(testPath[i]);
        }
    }
    newPath.push_back(newPath[0]);
    testPath = newPath;
}
```

Jest to najbardziej skomplikowana metoda generowania sąsiedztwa. Pobiera ona dwa indeksy oraz odwraca kolejność indeksów elementów zawartych w liście pomiędzy nimi.

- **Lista tabu** – w algorytmie Tabu Search lista tabu określa, jak długo dane przejście między sąsiednimi rozwiązaniami jest zabronione. W mojej implementacji, lista tabu zawiera dwa argumenty określające transformację obecnego rozwiązania oraz dekrementowaną, z każdą iteracją programu, wartość tabu. Gdy wartość staje się zerem, przejście wraca do przestrzeni dozwolonych przejść,

- **Dywersyfikacja** – jest to koncepcja zwiększająca różnorodność przeszukiwanej przestrzeni rozwiązań. Zapobiega ona zbyt długie pozostawanie w minimum lokalnym przestrzeni sąsiedztwa. W poniższym programie mechanizm dywersyfikacji polega na wprowadzeniu zmiennej, która co daną liczbę iteracji, zeruje listę tabu i przechodzi do losowego rozwiązania sąsiedniego kilkukrotnie.

```
//mechanizm mający zapewnić dywersyfikacje
if(iterationCounter>iterationStopCondition){
    iterationCounter = 0;
    clearTabu();
    random_device rd;
    mt19937 gen( sd: rd());
    uniform_real_distribution<> dis( a: 0,  b: changes.size()-1);
    pair<int, int> randomchoice;

    for(int i=0;i<matrixSize/2;i++) {
        randomchoice = changes[dis( &: gen)];
        doChange( indexFirst: randomchoice.first,  indexSecond: randomchoice.second);
    }
}
```

- **Kryterium aspiracji** – zapobiega ono pozostawaniu w samym rozwiązaniu zbyt długo. Niemożność przejścia do kolejnego rozwiązania może być spowodowana zablokowaniem wszystkich sąsiednich rozwiązań poprzez dodanie ich do listy tabu. W projekcie zaimplementowane zostało zaimplementowane najprostsze kryterium aspiracji; rozwiązanie przechodzi do sąsiada który jest najdłużej na liście tabu, a więc ma w niej najmniejszą wartość.

```
bestPair = { x: -1, y: -1};
int min=INT_MAX;
for(int i=0;i<tabuList.size();i++){
    for(int j=0;j<tabuList.size();j++){
        if(tabuList[i][j]==0)continue;
        if(tabuList[i][j]<min){
            min = tabuList[i][j];
            bestPair.first = i;
            bestPair.second = j;
        }
    }
}
doChange( indexFirst: bestPair.first,  indexSecond: bestPair.second);
calcCost();
currentPath = testPath;
currentLen = testLen;
if(currentLen<bestLen) {
    bestPath = currentPath;
    bestLen = currentLen;
}
```

- **Początkowa temperatura** - Pierwsza temperatura przyjęta na początku procesu symulowanego wyżarzania. Wysoka początkowa temperatura pozwala na akceptację większej liczby ruchów pogarszających jako strategii eksploracyjnej. Można ją obliczać różnymi sposobami. W implementacji liczona jest średnia kosztów stu rozwiązań problemu i wynik mnożony razy 1.5

```
long Sa::calcBeginTemperature() {
    testPath = bestPath;
    testLen = bestLen;
    long sum = testLen;

    for(int i=0; i<99; i++){
        swapPoints( swappedPoints: generateSwapPoints());
        calcLen();
        sum += testLen;
    }

    sum *= 0.015;
    return sum;
}
```

- **Końcowa temperatura** - Temperatura, przy której algorytm kończy działanie. Zwykle jest to niska temperatura, co oznacza, że proces schładzania stopniowo zmniejsza temperaturę do pewnego minimalnego poziomu
- **Kryterium stopu** – warunek przy którym algorytm kończy swoje działanie. Zazwyczaj jest to czas przez jaki algorytm ma działać. Może nim być również temperatura końcowa (w przypadku Symulowanego Wyżarzania) lub osiągnięta maksymalna liczba iteracji
- **Funkcja energii, celu (kosztu) -** Funkcja, którą algorytm stara się zminimalizować. Określa, jak dobrze rozwiązanie spełnia kryteria optymalizacyjne
- **Epoka -** Liczba iteracji lub kroków w ramach jednej wartości temperatury
- **Schemat schładzania** – w Symulowanym Wyżarzaniu jest to sposób zmniejszania temperatury co każdą epokę. Istnieje wiele efektywnych schematów schładzania. W omawianym algorytmie zaimplementowane zostały trzy:

```
//chłodzenie geometryczne
double Sa::calcGeometricTemp(double T){
    return T*coolingFactor; //*0.995 dla 5 minut
}
//chłodzenie logarytmiczne
double Sa::calcLogaritmicTemp(double T, int eraNumber){
    return T/(1+coolingFactor*log( x: eraNumber+1));
}
//chłodzenie wykładnicze
double Sa::calcExpotentialTemp(double T, int eraNumber){
    return pow( x: coolingFactor,  y: eraNumber)*T;
}
```

2. Opis klas w projekcie

2.1. Sa

Klasa Sa odpowiada za sterowanie algorytmem Symulowanego Wyżarzania. W jej skład wchodzą, między innymi metoda ***start*** - główna metoda klasy. Rozpoczyna działanie algorytmu i zarządza wszystkimi jego elementami. Na początku algorytm chciwy (*greedyAlg*) do wyznaczenia rozwiązania początkowego. Wyznacza początkową temperaturę (*calcBeginTemperature*) i ilość iteracji w jednej epoce. Następnie tworzy obiekt *Time* oraz rozpoczyna mierzenie czasu. Tu zaczyna się właściwa część algorytmu wykonywana w pętli aż do wystąpienia kryterium stopu. Dla każdej epoki losowane są przejścia między sąsiadami tyle razy, ile wynosi długość epoki. Algorytm losuje dwa różne indeksy w liście (*generateSwapPoints*), które następnie określą kolejne potencjalne rozwiązanie. Do testowej listy wierzchołków przypisywany jest nowy sąsiad (*swapPoints*), w zależności od wybranej wcześniej definicji sąsiedstwa i liczony jest jego koszt (*calcLen)*. Jeśli sąsiad ma mniejszy koszt, jest przypisywany do obecnego rozwiązania, jeśli obecne rozwiązanie jest lepsze od najlepszego znalezionego do tej pory, również następuje takie nadpisanie. W przypadku gdy nowe rozwiązanie jest gorsze, sprawdzane jest czy różnica energii (kosztów) rozwiązań jest w granicach akceptacji. W ów czas następuje podobna procedura przejścia do nowego rozwiązania. Pod koniec każdej epoki inkrementowana jest liczba przebytych epok oraz liczona nowa temperatura (*calcNewTemperature*). W zależności od wybranego wcześniej schematu chłodzenia, temperatura jest liczona według jednego z trzech wzorów.

2.2. Ts

Klasa *Ts* odpowiedzialna jest za zarządzanie i przechowywanie wyników algorytmu Tabu Search. W jej skład wchodzą między innymi dwie najważniejsze funkcje:

- ***startSearching*** – funkcje główna algorytmu. Na początku oblicza pierwsze rozwiązanie metodą zachłanną (*greedyAlg*) i generowane są wszystkie możliwe transformacje jakie mogą nastąpić w liście (*getChanges*). Zapisane są one w postaci listy par. Rozpoczyna się mierzenie czasu. Zaraz po tym rozpoczyna się pętla wykonująca się do momentu wystąpienia kryterium stopu. W pętli znajdowany jest sąsiad, który ma najmniejszy koszt (*bestChange*). Sąsiad ów zapisywany jest jako para indeksów deklarujących transformację obecnego rozwiązania. Lista tabu odnotowuje ta transformacje. Elementy w liście tabu są dekrementowane po czym następuje uruchomienie mechanizmu dywersyfikacji. Jeśli pętla wykonała określoną liczbę iteracji, lista tabu jest zerowana (*clearTabu*) i następuje losowa transformacje obecnego rozwiązania (*doChange*).
- **bestChange** – metoda używana jest do znajdywania sąsiada o najmniejszym koszcie. Dla każdej transformacji sprawdzane jest czy nie występuje ona na liście tabu. Jeśli nie występuje, dokonuje się jej na liście testowej (*doChange*) oraz liczony jest koszt (*calcCost*) w celu sprawdzenia czy dany sąsiad ma mniejszy koszt niż poprzedni rozważany. Mechanizm ten pozwala na znalezienie optymalnego rozwiązania wśród sąsiadów. Jeśli cel został wypełniony, zapisujemy nowe rozwiązanie oraz zwracana jest para

znalezionych indeksów. Jeśli natomiast nowe, lepsze rozwiązanie nie zostało znalezione, uruchamiany jest mechanizm obsługujący kryterium aspiracji. Pozwala ono na przejście do rozwiązania, które jest najdłużej na liście tabu.

2.3. Time

Klasa Time zaimplementowana została identycznie jak w poprzednim projekcie. Pozwala ona na zapisanie czasu rozpoczęcia algorytmu i dzięki temu na uzyskanie czasu w dowolnej chwili jego funkcjonowania.

2.4. **Test**

Klasa tworzona na początku działania programu. Przechowuje podane przez użytkownika parametry, zarządza pamięcią oraz pozwala w łatwy sposób zarządzać eksperymentami na instancjach problemu. Poza metodami odpowiedzialnymi za komunikacje z użytkownikiem, warto dodać kilka komentarzy do reszty funkcji:

- **chechPath** – funkcja odpowiedzialna za obliczanie kosztu ścieżko zapisanej w pliku tekstowym dla odpowiedniego grafu. Funkcja po otwarciu pliku do którego ścieżka podana jest jako argument funkcji, przechodzi linijka po linijce między wierzchołkami grafu sumując koszty tych przejść. Po przejściu do ostatniego wierzchołka i zakończeniu cyklu Hamiltona zwracana jest jego długość.
- **saveSolution –** funkcja tworzy plik z unikalną nazwę i zapisuje w nim liczbę wierzchołków grafu dla którego klasa *Test* znalazła ostatnio rozwiązanie, w kolejnych liniach zaś następujące po sobie elementy listy będącej najlepszym znalezionym rozwiązaniem.

3. **Wyniki pomiarów**

Poniższe tabele zawierają wyniki testów przeprowadzonych na algorytmach. Zaznaczonymi kolorem wartościami są koszty najlepszego rozwiązania w poszczególnej instancji eksperymentu. Poniżej znajduje się najlepsze znalezione rozwiązanie i czas w którym zostało ono odkryte oraz błąd względny w procentach.

3.1. Tabu Search

	tabu search								
	ftv55			ftv170			rbg358		
nr	1	2	3	1	2	3	1	2	3
1	1716	1760	1768	3352	3699	3887	1324	1319	1763
2	1716	1760	1768	3352	3699	3887	1324	1319	1763
3	1716	1760	1768	3352	3699	3887	1324	1319	1763
4	1716	1760	1768	3352	3699	3887	1324	1319	1763
5	1716	1760	1768	3352	3699	3887	1324	1319	1763
6	1716	1760	1768	3352	3699	3887	1324	1319	1763
7	1716	1760	1768	3352	3699	3887	1324	1319	1763
8	1716	1760	1768	3352	3699	3887	1324	1319	1763
9	1716	1760	1768	3352	3699	3887	1324	1319	1763
10	1716	1760	1768	3352	3699	3887	1324	1319	1763
minimum	1716	1760	1768	3352	3699	3887	1324	1319	1763
t [s]	0	0	0	0	0	0	0	0	0
błąd [%]	6,716418	9,452736	9,950249	21,66969	34,26497	41,08893	13,84351	13,41359	51,59071

3.2. Symulowane wyżarzanie

	symulowane wyżarzanie								
	ftv55			ftv170			rbg358		
nr	1	2	3	1	2	3	1	2	3
1	1792	1820	1852	3817	3923	3923	1322	1346	1359
2	1792	1834	1858	3901	3923	3923	1274	1349	1314
3	1769	1710	1755	3923	3923	3923	1249	1312	1305
4	1817	1710	1835	3569	3923	3923	1310	1369	1326
5	1832	1790	1923	3628	3923	3923	1300	1332	1320
6	1718	1769	1783	3873	3923	3923	1277	1350	1334
7	1852	1740	1714	3923	3923	3900	1304	1352	1363
8	1761	1812	1759	3923	3923	3923	1302	1377	1327
9	1727	1795	1843	3923	3923	3923	1281	1334	1321
10	1703	1791	1821	3923	3923	3923	1323	1329	1354
minimum	1703	1710	1714	3569	3923	3900	1249	1312	1305
min index	10	3	7	4	1	7	3	3	3
t [s]	103	92	77	158	0	88	360	98	146
Błąd[%]	5,90796	6,343284	6,59204	29,54628	42,39564	41,5608	7,394669	12,81169	12,2098

4. **Podsumowanie**

Projekt utrwalił wiedzę z zakresu dwóch algorytmów metaheurystycznych. Pozwolił dogłębnie poznać ich zasadę działania oraz możliwości. Mimo (zapewne) poprawnego zaimplementowania algorytmów, tabu search okazał się nieefektywny. Jest to spowodowane nieoptymalnym dobraniem parametrów algorytmu.