

POLITECHNIKA WROCŁAWSKA

- INE3802 - Informatyka systemów autonomicznych

Praca zaliczeniowa

Wydział Elektroniki, INF/INS/ISB

Autor:

Grzegorz Studziński 133374

Temat: Heurystyka, co to jest? Praktyczne przedstawienie algorytmu poszukiwania Tabu Search

Wrocław, 31.05.2007




Praca zawiera krótką charakterystykę algorytmów heurystycznych oraz przykład implementacji algorytmu Tabu Search.

Agenda:	Strona
1. CO TO JEST HEURYSTYKA?	2
2. PRZYKŁADY HEURYSTYK	2
3. TABU SEARCH – OPIS	3
4. TABU SEARCH – PRAKTYCZNE ZASTOSOWANIE	3
5. PODSUMOWANIE	6

1. CO TO JEST HEURYSTYKA?

Słowo „heurystyka” pochodzi od greckiego czasownika „*heuriskein*”, oznaczającego „znaleźć” lub „odkryć”. Za heurystykę uważano praktyczną zasadę służącą do generowania dobrych rozwiązań bez ekstensywnego poszukiwania.

W literaturze możemy wyróżnić trzy główne mechanizmy stosowane w algorytmach wyszukujących:




-  **szukanie analityczne** – sterowane za pomocą matematycznych funkcji, np. w optymalizacji algorytmy mogą być zależne od gradientu czy hesjanu, gwarantują odnalezienie istniejącego rozwiązania, ale zwykle w praktyce dostarczają rozwiązań nieglobalnych
-  **ślepe szukanie** – inaczej nazywane poszukiwaniem niesterowalnym
-  **szukanie heurystyczne** – sterowane szukanie, szeroko stosowane w praktyce, dostarcza zwykle satysfakcjonujących rozwiązań (choć nie ma gwarancji, że są to rozwiązania globalne).

Każdy proces poszukiwania można podzielić na trzy główne fazy – wybranie punktu startowego poszukiwań, inicjacja – generacja kolejnych rozwiązań – zakończenie po spełnieniu przez rozwiązanie przyjętego kryterium.

W teorii problemów NP-zupełnych, do których głównie stosuje się metody heurystyczne, dla niektórych przypadków dowodzi się, że dokładne rozwiązania takich problemów mogą być aproksymowane przez rozwiązania, które otrzymujemy wykorzystując metody heurystyczne. Zwykle takie rozwiązania nie różnią się od rozwiązań optymalnych o więcej niż kilka zadanych procentów [1].

2. PRZYKŁADY HEURYSTYK

Do grupy algorytmów heurystycznych możemy zaliczyć:

-  **Algorytmy ewolucyjne** - Bazują na mechanizmach ewolucji naturalnej. Lepiej przystosowane osobniki mają większe szanse na reprodukcję. Cechy osobnika zawarte są w jego chromosomie. Osobniki przekazują swoje cechy potomkom. Ogólny algorytm ewolucyjny składa się z następujących etapów: 1. Wyznaczenie funkcji przystosowania osobników, 2. Wybór puli rodzicielskiej, 3. Krzyżowanie, 4. Mutacja nowych osobników, 5. Selekcja nowej populacji.
-  **Algorytmy lokalnego przeszukiwania** – algorytmy poszukujące minimum. Do tych algorytmów możemy zaliczyć Poszukiwanie Zstępujące, Poszukiwanie Losowe, Symulowane Wyżarzanie oraz **Tabu Search**. Wszystkie te algorytmy poruszają się w sąsiedztwie danego rozwiązania, poszukując lepszego rozwiązania niż dotychczas. Nie wszystkie z tych algorytmów są skuteczne. Przykładem może być Przeszukiwanie zstępujące, którego czas zakończenia jest niedeterministyczny oraz zatrzymuje się w pierwszym minimum lokalnym, co może znacznie zawęzić przestrzeń przeszukiwanych rozwiązań. Aby obronić się przed tymi problemami stworzono Symulowane Wyżarzanie i Tabu Search, które odporne są na minima lokalne a czas ich zakończenia jest deterministyczny [2].
-  **Algorytm mrówkowy** – Bazuje na zachowaniu owadów w środowisku. Mrówka szuka pożywienia żeby je zanieść do domu; porusza się w sposób losowy; zostawia za sobą ślad feromonowy, którego intensywność maleje z czasem (feromon paruje). W algorytmie feromon symbolizuje prawdopodobieństwo wybrania następnego punktu podróży, które jest modyfikowane za każdym ruchem mrówki (w aktualnym punkcie

prawdopodobieństwo rośnie, w pozostałych maleje). Mrówka idzie do tego punktu, który jest najbardziej prawdopodobny w wyborze (jednak nie może w danej iteracji wybrać już odwiedzonego punktu).

W informatyce istnieje więcej różnorodnych algorytmów heurystycznych. Są one ciągle modyfikowane i optymalizowane w celu uzyskiwania lepszych rozwiązań dla problemów, które nie posiadają algorytmów dokładnego rozwiązania w skończonym czasie.

3. TABU SEARCH – OPIS

Wszystkie algorytmy lokalnego poszukiwania (również Tabu Search) związane są z pojęciem sąsiedztwa punktu. W przestrzeni R^n sąsiedztwem punktu 'x' jest hipersfera o środku w punkcie 'x' i promieniu 'ε'.

Natomiast w permutacji n-elementowej można zdefiniować sąsiedztwa jako następujące typy:

1. Sąsiedztwo typu „**insert**”, 2. Sąsiedztwo typu „**swap**”, 3. Sąsiedztwo typu „**invert**”.

Jeśli mamy zdefiniowaną permutację $\pi = \langle \pi(0), \pi(1), \pi(2), \pi(3), \pi(4), \pi(5), \dots, \pi(n-1) \rangle$ Sąsiedztwa te przybierają przykładowe postacie:

1. **insert**(1 , 4) $\pi^* = \langle \pi(0), \pi(4), \pi(1), \pi(2), \pi(3), \pi(5), \dots, \pi(n-1) \rangle$,
2. **swap**(1 , 4) $\pi^* = \langle \pi(0), \pi(4), \pi(2), \pi(3), \pi(1), \pi(5), \dots, \pi(n-1) \rangle$
3. **invert**(1 , 4) $\pi^* = \langle \pi(0), \pi(4), \pi(3), \pi(2), \pi(1), \pi(5), \dots, \pi(n-1) \rangle$

Tabu Search – zostało wymyślone przez prof. Freda Glovera w latach 1989-90. Algorytm bazuje na poszukiwaniu losowym, jednak w porównaniu do niego posiada dodatkowo „listę tabu”, która pozwala na uniknięcie utknięcia w minimum lokalnym.

Algorytm TS rozpoczyna swoje działanie od pewnego rozwiązania początkowego x_0 i sukcesywnie porusza się po rozwiązaniach sąsiednich. W danej iteracji i , z sąsiedztwa aktualnego rozwiązania $N(x^i)$ wybierane jest „niezakazane” rozwiązanie $x^{(i+1)}$ takie, że

$$f(x^{(i+1)}) = \min \{f(x)\}, \text{ gdzie } x \in N(x^i), \text{ a } f(x) \text{ - funkcja celu.}$$

Rozwiązanie już odwiedzone, jest zapamiętywane i w dalszym procesie poszukiwania jest zakazane (ma status tabu). Lista tabu jest to zazwyczaj kolejka LIFO o długości do 10 elementów (nazwana pamięcią krótkoterminową), może jednak wykorzystywać pamięć długoterminową (o długości do kilkudziesięciu elementów).

4. TABU SEARCH – PRAKTYCZNE ZASTOSOWANIE

Algorytm TS można zastosować przy tworzeniu programu „osobistego agenta turystycznego”. Agent turystyczny, będzie miał za zadanie wybrać trasę podróży mając podane dane na temat miast na świecie. Na podstawie tego agent będzie miał za zadanie wybrać dla nas ciekawą podróż przy niskim koszcie, stosunkowo krótkim czasie, a dużą satysfakcją.

Każde miasto może mieć zatem następujące parametry w języku C++:

```
class Miasto {
public:
    int indeks;                // nr indeksowy tego miasta
    float czas_zwiedzania;     // czas zwiedzania miasta
    float satysfakcja_uzytkownika; // jeśli nie jest ustawiona wynosi = 0
    float walory;              // walory zabytkowe i rozrywkowe miasta
    float koszt_pobytu;         // koszt pobytu w danym mieście
    bool czy_zwiedzzone;       // czy użytkownik był już wcześniej w danym mieście
};
```

Dodatkowo musi być określony zbiór miast oraz czas i koszt podróży z jednego miasta do drugiego, co można przedstawić poniżej:

```
struct Polaczenie {
    float czas_podrozy;
    float koszt_podrozy;
};

class ZbiorMiast{
public:
    Miasto *perm;           // zbior Miast
    Polaczenie **polaczenia; // polaczenia zawierające koszty podróży
    int liczba_miast;       // n liczba miast
    .....
};
```

Do wyboru najlepszego rozwiązania obliczana jest funkcja celu, która oblicza funkcja zgodnie z parametrami przynależącym do każdego z miast i do parametrów połączenia między tymi miastami.

Ważne jest dobre określenie funkcji celu, w zależności co jest dla użytkownika ważne. Najprostszym, chociaż nieoptymalnym, podejściem jest zsumowanie parametrów z wagami tych parametrów.

Cały algorytm zamieścimy w klasie:

```
class TSearch : public ZbiorMiast {
protected:
    int *indexSasiad; // tablica indeksow Sasiada
    int numSasiad;    // liczba Sasiadów
    int *bestPerm;    // tablica najlepszej permutacji
    void Poczatek();  // wygenerowanie początkowego rozwiązania
    void Sasiedztwo(); // Włosowanie sasiedztwa dla aktualnego rozwiązania
    void Swap(int rand1, int rand2); // sasiedztwo Swap-2 permutacji
    void Invert(int rand1, int rand2); // sasiedztwo Invert permutacji
    float FunkcjaCelu();

public:
    float bestMin; // wartosc najlepszej permutacji
    TSearch(const ZbiorMiast &temp, int numSasiad);
    ~TSearch();
    void Results();
    void Algorithm(int numPetli); // wywołanie funkcji algorytmu
};
```

TS potrzebuje również listy Tabu, która możemy zaprojektować następująco:

```
class TabuList {
private:
    Moves *list; // lista zakaznych ruchow
    int maxList; // rozmiar listy
    int akt;     //aktualny wskaznik listy

public:
    TabuList(int maxList);
    ~TabuList();
    void Add(int i1, int i2); //dodanie zakazanego ruchu do listy
    char Search(int i1, int i2); //szukanie danego ruchu 1=if(true)
};
```

Posiadając już zarysy klas możemy zacząć implementację. Kroki całego algorytmu można opisać w kilku punktach.

Krok 1. SZUKANIE POCZĄTKOWEGO ROZWIĄZANIA

W celu znalezienia początkowego rozwiązania (dla którego przeznaczona jest tablica indeksów ***bestPerm**), które dziedziczy skopiowany zbiór miast. Posortowaliśmy indeksy zadań wg parametru **koszt_podróży**, który ma znaczny wpływ na całkowitą wartość zwracaną przez **TSearch::FunkcjaCelu()**.

Realizacją tej części algorytmu zajmuje się funkcja **void TSearch::Poczatek()**.

Krok 2. GENEROWANIE I SPRAWDZANIE SĄSIADÓW

W celu realizacji przeszukiwań dla zadanej liczby **numSasiad** generuje się sąsiedztwo, wykonane losowymi ruchami

- Swap – fn. **void Swap(int rand1, int rand2)**
- Invert – fn. **void Invert(int rand1, int rand2)** - bardziej efektywne

Gdzie indeksy **rand1, rand2** są losowane i sprawdzane na liście **TabuList**.

Następnie zgodnie z **TSearch::FunkcjaCelu()**, liczona jest tymczasowa wartość permutacji, która jest porównywana z wartością **bestMin**, dotychczasowego najlepszego rozwiązania. Jeśli jest większa(lepsza) zastępuje dotychczasową najlepszą permutację.

Generacją i sprawdzeniem sąsiedztwa zajmuje się fn. **void TSearch::Sasiedztwo()**

Krok 3. SPRAWDZANIE TABU-LISTY

Przed każdym ruchem typu Swap lub Invert sprawdzana jest para indeksów na liście Tabu(**TabuList::Moves *list**) - lista zabronionych ruchów, która jest kolejką typu LIFO.

Przeszukanie kolejki dokonuje się przez funkcję **char TabuList::Search(int i1, int i2)**, która zwraca '1' gdy znajdzie taką parę.

Jeśli zadanych indeksów nie będzie na tej liście można wygenerować kolejną permutację sąsiedztwa (poprzez ruchy Invert lub Swap) oraz dodaje tę parę indeksów do listy Tabu za pomocą fn. **void TabuList::Add(int i1, int i2)**

Krok 4. ZAKOŃCZENIE ALGORYTMU

Algorytm kończy się po kilkakrotnym = **numSasiad**, przeszukaniu danych sąsiedztw. A rozwiązaniem jest permutacja indeksów danego zbioru ***bestPerm**, oraz wartość rozwiązania **TSearch::bestMin**.

Ogólna postać funkcji wywołującej cały algorytm Tabu Search.

```
void TSearch::Algorithm(int numSasiad)
{
    Poczatek();

    for(int i=0 ; i<numSasiad ; i++)
    {
        Sasiedztwo();
    }
    Results();
}
```

Aby przetestować jakość algorytmu (z jakim błędem wyliczana jest ostateczna wartość **bestMin**), należy porównać wyniki z najlepszą wartością (wyznaczoną po przeglądzie pełnym permutacji lub po użyciu algorytmu Branch and Bound).

Algorytm TS jest jednym z najlepszych podejść do trudnych problemów kombinatorycznych, jednak skuteczność i praktyczność działania zależy od dobrego doboru parametrów elementów zbioru(Miast), długości listy Tabu, typu sąsiedztwa oraz wyznaczenia właściwej funkcji celu.

5. PODSUMOWANIE

Algorytmy heurystyczne pozwalają rozwiązać zagadnienia i problemy z dziedziny Sztucznej Inteligencji. Mogą być stosowane w systemach agentowych, tam gdzie szuka się optymalnego rozwiązania. Heurystyki dają dużą elastyczność w programowaniu i łatwo dają się przystosować do konkretnego problemu.

Przy projektowaniu algorytmów poszukujących takich jak Tabu Search należy rozważyć cały szereg spraw mających decydujący wpływ na postać algorytmu, stosowanych mechanizmów i metod. Począwszy od wybrania właściwego punktu startowego przez określenie skali naszego problemu, jego związków z rzeczywistością do zdefiniowania typu wiedzy, musimy przeanalizować dany problem, tak, aby otrzymać satysfakcjonujące nas rozwiązanie. Dla przykładu tzw. wiedza pozytywna oznacza, że algorytm „nagradza” dobre rozwiązania, a “karze” złe. Zwykle w praktycznych zastosowaniach mamy do czynienia z problemami wielkiej skali i właśnie dla takich problemów idealne wydaje się być stosowanie metod szukania heurystycznego.

Nowoczesne algorytmy heurystyczne są inspirowane wiedzą z innych dziedzin takich, jak biologia, mechanika statystyczna, neurologia czy fizyka oraz wiele innych. Zwykle jądrem tych metod jest generowanie kolejnych rozwiązań z lokalnego sąsiedztwa wcześniejszych rozwiązań.

6. LITERATURA

- [1] Lewandowski K., Jedynek M., „Simulated Annealing – 30 Lat Później, Czyli Jak Daleko Nam Do Myślących Maszyn”
- [2] wykłady dr inż. Macieja Lichtensteina
- [3] http://en.wikipedia.org/wiki/Taboo_search