**Piotr Wilczyński**

**Temat: Problem plecakowy z wykorzystaniem algorytmu genetycznego na przykładzie kelnera.**

**Wstęp teoretyczny**

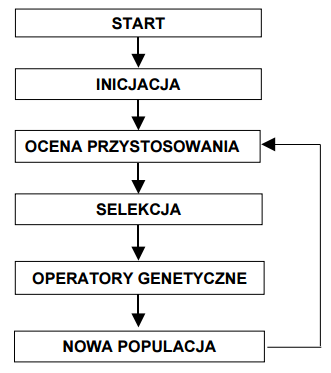
**Problem plecakowy** (ang. discrete knapsack problem) jest jednym z najczęściej poruszanych problemów optymalizacyjnych. Nazwa zagadnienia pochodzi od maksymalizacyjnego problemu wyboru przedmiotów, tak by ich sumaryczna wartość była jak największa i jednocześnie mieściły się w plecaku. Przy podanym zbiorze elementów o podanej wadze i wartości, należy wybrać taki podzbiór by suma wartości była możliwie jak największa, a suma wag była nie większa od danej pojemności plecaka.

**Algorytm genetyczny** - rodzaj heurystyki przeszukującej przestrzeń alternatywnych rozwiązań problemu w celu wyszukania rozwiązań najlepszych. W algorytmach genetycznych stosuje się pojęcia zapożyczone z genetyki naturalnej. Istnieje populacja rozwiązań. Każde rozwiązanie jest nazywane osobnikiem. Na populacji dokonujemy *selekcji* (słabe osobniki giną), *krzyżowania* (czyli tworzenia nowych osobników) i *mutacji* (dzika karta).

Używane terminy:  
• **Populacja** – zbiór osobników o określonej liczebności.   
• **Osobniki** populacji w algorytmach genetycznych to zakodowane w postaci chromosomów zbiory parametrów zadania, czyli rozwiązania, określane też jako punkty przestrzeni poszukiwań.   
• **Chromosomy** – inaczej łańcuchy lub ciągi kodowe – to uporządkowane ciągi genów.   
• **Gen** – nazywany też cecha, znakiem, detektorem – jest to pojedynczy element genotypu, w szczególności chromosomu.   
• **Genotyp**, czyli struktura - to zespół chromosomów danego osobnika. Osobnikami populacji mogą być genotypy albo pojedyncze chromosomy.   
• **Fenotyp** - zestaw wartości odpowiadający danemu genotypowi, czyli zdekodowana struktura. Jest to zbiór parametrów zadania (rozwiązanie, punkt przestrzeni poszukiwań).   
• **Allel** to wartość danego genu (określana też jako wartość cechy lub wariant cechy).   
• **Locus** to położenia danego genu w łańcuchu czyli w chromosomie.   
• **Funkcja przystosowania** (ang. fitness function) nazywana też funkcją dopasowania lub funkcją oceny. Stanowi ona miarę przystosowania (dopasowania) danego osobnika w populacji. Funkcja przystosowania pozwala ocenić stopień przystosowania poszczególnych osobników w populacji. Na tej podstawie wybiera się osobniki najlepiej przystosowane (o największej wartości funkcji przystosowania). Jest to zgodne z ewolucyjną zasadą przetrwania „najsilniejszych” (najlepiej przystosowanych). Ma ona duży wpływ na działanie algorytmów genetycznych i musi być odpowiednio zdefiniowana.   
• **Generacja** – to kolejna iteracja w algorytmie genetycznym.   
• **Pokolenie** (nowe pokolenie lub pokolenie potomków) – to nowoutworzona populacja osobników.

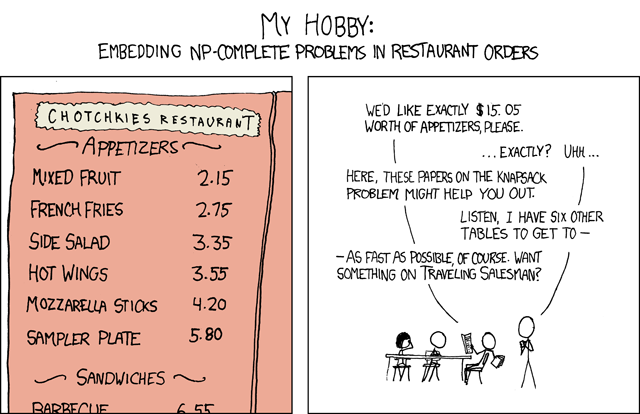
W algorytmie genetycznym, w każdej jego iteracji, jest oceniane przystosowanie każdego osobnika danej populacji za pomocą funkcji przystosowania. Na tej podstawie tworzona jest nowa populacja osobników, którzy stanowią zbiór potencjalnych rozwiązań problemu, np. zadania optymalizacji.

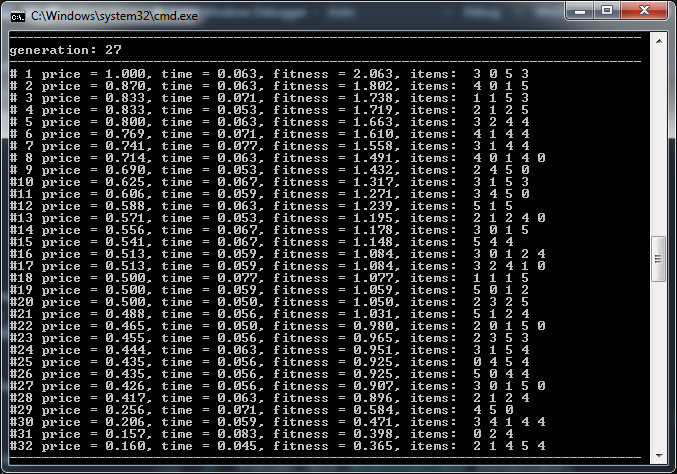
**Mechanizm działania klasycznego** (elementarnego, prostego) algorytmu genetycznego jest prosty. Polega na kopiowaniu ciągów i wymianie podciągów. Na rysunku poniżej pokazano schemat działania algorytmu genetycznego, które składa się z następujących kroków:   
• **Inicjacja** – utworzenie populacji początkowej, poprzez losowy wybór ustalonej liczby chromosomów.   
• **Ocena przystosowania** – obliczenie wartości funkcji przystosowania dla każdego chromosomu.   
• **Selekcja chromosomów** – wybór chromosomów, które biorą udział w tworzeniu nowej populacji.   
• **Zastosowanie operatorów genetycznych** – na grupie chromosomów, wybranej drogą selekcji działają operatory genetyczne. W klasycznym algorytmie genetycznym są operatory krzyżowania i mutacji. Przy czym, krzyżowanie powinno zachodzić znacznie częściej niż mutacja.   
• **Utworzenie nowej populacji** – Chromosomy otrzymane jako rezultat działania operatorów genetycznych na chromosomy tymczasowej populacji wchodzą w skład nowej populacji. W klasycznym algorytmie genetycznym cała poprzednia populacja chromosomów jest zastępowana przez tak samo liczną nowa populacje potomków.   
• **Wyprowadzenie „najlepszego” chromosomu** – najlepszym rozwiązaniem jest chromosom o największej wartości funkcji przystosowania. Nową populację tworzą chromosomy powstałe w wyniku selekcji i działania operatorów genetycznych. Nowa populacja zastępuje w całości starą, i staje się bieżącą w kolejnej generacji (iteracji) algorytmu genetycznego.



**Przykład**

Ten przykład pokazuje wykorzystanie algorytmu genetycznego do rozwiązania problemu przedstawionego poniżej. Algorytm genetyczny zaimplementowany przy wykorzystaniu biblioteki GALex.





O problemie

Grupa ludzi wchodzi do restauracji i chcą wydać 15,05$ na przekąski. Chcą również zjeść tak szybko jak to możliwe. Pozostawia to kelnera z NP-ciężkim problemem do rozwiązania, odmiana problemu plecakowego.

Problemem jaki kelner musi rozwiązać jest wybór pozycji z menu, które kosztują tyle ile klienci określili. Klienci chcą zjeść tak szybko jak to możliwe. Kelner nie może wybrać dowolnej kombinacji produktów które zaspokoją ich cenowe wymagania. Musi znaleźć wszystkie kombinacje, aby móc wybrać tą która zajmie mu najmniej czasu do zrealizowania.

Chromosom

Definiowanie reprezentacji chromosomów będących użytych przez algorytm genetyczny tak samo jak zaprojektowanie dobrych krzyżówek i operacji mutacji są ważnymi krokami do rozwiązania tego problemu.

Reprezentacja

Jedną z pierwszych rzeczy która powinna być zrobiona podczas projektowania algorytmu genetycznego jest wybór odpowiedniej dla tego rozwiązania reprezentacji chromosomów. Niektóre z możliwych reprezentacji przedstawiono poniżej

* nieuporządkowana lista gdzie węzły listy zawierają element które mogą być podane
* tablica przechowująca liczbę elementów które mogą być podane

Linked List Representation  
***Nieuporządkowana lista***

  
***Tablica***

Wybór reprezentacji ma wpływ na projektowanie operacji genetycznych takich jak krzyżowanie czy mutacji. W naszym przykładzie wykorzystujemy listę.

Jedną z zalet jest to że reprezentacja pozwala algorytmowi genetycznemu na samodzielną optymalizacje reprezentacji. Krzyżowanie może przetasowywać bloki przedmiotów z całego chromosomu bez wpływu na jego wartość przystosowania. (fitness value). W ten sposób algorytm genetyczny może pogrupować dobre kombinacje elementów razem. Takie kombinacje nazywa się budowaniem bloków. Grupowanie zmniejsza szanse wystąpienia zakłuceń które mogą wpływać na jakość rozwiązania i wartości przystosowań.

Następną rzeczą jakiej potrzebujemy jest zdefiniowanie bloku konfiguracji chromosomu (chromosome configuration block - CCB ), która będzie zawierać listę przekąsek, z ich cenami i czasem potrzebnym do wykonania.

Zdefiniowane operacje genetyczne:

* inicjalizacja chromosomu
* operacje dopasowania
* operacje krzyżowania
* operacje mutacji

Operacje dopasowania (przystosowania)

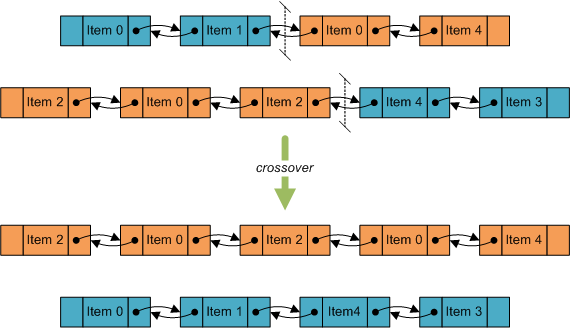
Operacje dopasowania są odpowiedzialne za obliczanie wartości fitness chromosomu. Określony problem ma dwa kryteria: koszt przekąski i czas produkcji, algorytm powinien wziąć oba pod uwagę przy obliczaniu wartości dopasowania chromosomu. W tym przypadku cena jest ważniejszym kryterium od czasu, wagi mogą być przypisane do oby wartości. Biorąc pod uwagę wszytskie wartości i ich wagi, funkcja dopasowania przyjmuje postać:  
Fitness function

***f****– wartość fitness*, ***wp****– waga ceny*, ***wt****– waga czasu*, ***tp****– cena docelowa*, ***n****– liczba przekąsek*, ***api****– cena itej przystawki*, ***ati****– wymagany czas do wyprodukowania itej przystawki*

Ponieważ waga ceny jest ważniejszym kryterium, wyższa waga powinna być przypisana do **wp** niż do **wt**. W tym przypadku **wp** = 2.0 a **wt** = 1.0.

Operacje krzyżowania

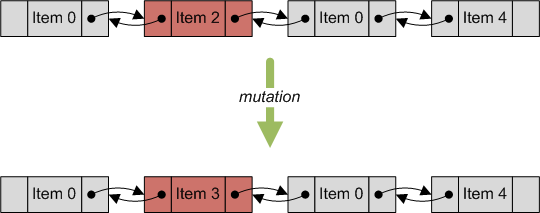
Celem operacji krzyżowania jest stworznie nowego rozwiązania poprzez łączenie istniejących. Implementacja operacji krzyżowania zależy od reprezentacji. Prosta jednopunktowa operacja krzyżowania:

  
***Operacja krzyżowania***

Biblioteka zawiera implementacje operacji krzyżowania ( klasa GaListMultipointCrossover ).

Operacje mutacji

Odpowiedzialne są za utrzymanie różnorodności genetycznej chromosomów w populacji. Odbywa się to poprzez zmianę genów losowa w nowo produkowanych chromosomach:

  
***Operacja mutacji***

Klasa XkcdMutationOperation.

|  |
| --- |
|  |

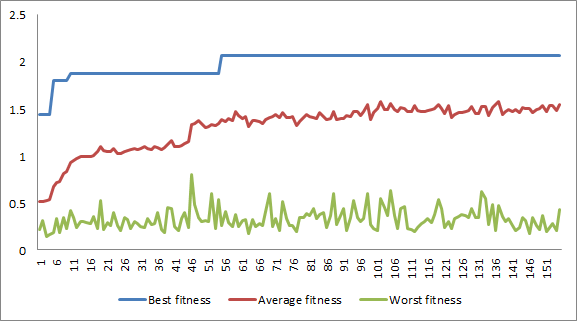
Parametry dla algorytmu

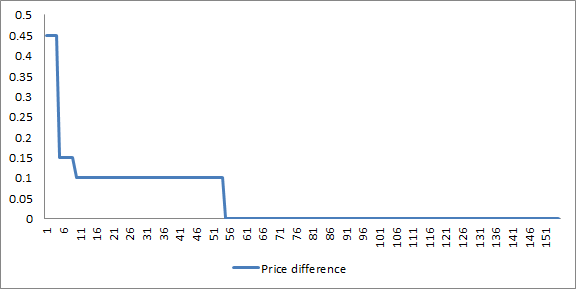
Wybrane parametry algorytmu:

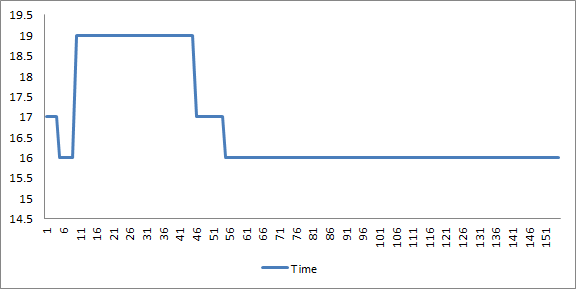
|  |  |
| --- | --- |
| Prawdopodobieństwo mutacji: | 30% |
| Rozmiar mutacji | 1 gen |
| Przyjmuje mutacje polepszające dopasowanie | Tak |
| Prawdopodobieństwo krzyżowania: | 80% |
| Punkty krzyżowania: | 1 |
| Typ algorytmu: | przyrostowy |
| Rozmiar populacji: | 32 chromosomy |
| Sortowanie populacji: | Tak |
| Sortowanie dopasowania: | maksymalizacja |
| Wagi dopasowania: | 2.0, 1.0 |
| Typ wyboru: | koło ruletki |
| Rozmiar wyboru: | 8 |
| Liczba potomstwa: | 8 |
| Typ zamiany: | Wymień najgorsze |

Wyniki

Poniżej przedstawiono wyniki dla pojedyńczego uruchomienia z ceną 15.05$. Możemy zauważyć postęp wartości dopasowania chromosomów w czasie działania. Oś X reprezentuje pokolenie a oś Y reprezentuje wartości dopasowania.

  
***Ważona wartość dopasowania najlepszego i najgorszego chromosomu w populacji i średnia wartość***

  
***Cena składników najlepszego chromosomu w populacji –*** *jak daleko jest od ceny docelowej*

  
***Czas składników najlepszego chromosomu w populacji*** *– czas potrzebny do zrobienia przekąski*

**Kod**

// create mating operation with:

// crossover - 80% probability, 2 offspring, 1 crossover point

// mutation - 30% probability, accept only improving mutations, 1 gene

Chromosome::CrossoverOperations::GaListMultipointCrossover crossover;

Problems::XKCD::XkcdMutationOperation mutation;

Chromosome::MatingOperations::GaBasicMatingOperation mating;

Chromosome::GaMatingConfig matingConfiguration(

  Chromosome::GaCrossoverSetup( &crossover,

    &Chromosome::GaCrossoverPointParams( 0.8f, 2, 1 ), NULL ),

  Chromosome::GaMutationSetup( &mutation,

    &Chromosome::GaMutationSizeParams( 0.3f, true, 1L ), NULL ) );

Problems::XKCD::XkcdConfigBlock::Appetizer appetizers[] =

{

  Problems::XKCD::XkcdConfigBlock::Appetizer("mixed fruit", 2.15f, 3),

  Problems::XKCD::XkcdConfigBlock::Appetizer("french fries", 2.75f, 2),

  Problems::XKCD::XkcdConfigBlock::Appetizer("side salad", 3.35f, 5),

  Problems::XKCD::XkcdConfigBlock::Appetizer("hot wings", 3.55f, 3),

  Problems::XKCD::XkcdConfigBlock::Appetizer("mozzarella sticks", 4.20f, 4),

  Problems::XKCD::XkcdConfigBlock::Appetizer("sampler plate", 5.80f, 7),

};

// create chromosome initializator

Problems::XKCD::XkcdInitializator initializator;

Chromosome::GaInitializatorSetup initializatorSetup( &initializator,

  NULL, &Chromosome::GaInitializatorConfig(

    &Problems::XKCD::XkcdConfigBlock(

      Common::Data::GaSingleDimensionArray&lt;

        Problems::XKCD::XkcdConfigBlock::Appetizer&gt;( appetizers, 6 ) ) ) );

// create fitness comparator for maximizing fitness value

Fitness::Comparators::GaSimpleComparator fitnessComparator;

Fitness::GaFitnessComparatorSetup fitnessComparatorSetup(

  &fitnessComparator,

  &Fitness::Comparators::GaSimpleComparatorParams(

    Fitness::Comparators::GACT\_MAXIMIZE\_ALL ), NULL );

// fitness operation

Problems::XKCD::XkcdFitnessOperation fitnessOperation;

Population::GaCombinedFitnessOperation populationFitnessOperation(

  &fitnessOperation );

// create population statistics trackers

// for fitness values and population size

Population::GaPopulationSizeTracker sizeTracker;

Population::GaRawFitnessTracker rawTracker;

Algorithm::Stubs::GaSimpleGAStub::GaStatTrackersCollection trackers;

trackers[ Population::GaPopulationSizeTracker::TRACKER\_ID ] =  &sizeTracker;

trackers[ Population::GaRawFitnessTracker::TRACKER\_ID ] =  &rawTracker;

trackers[ Population::GaScaledFitnessTracker::TRACKER\_ID ] =  &scaledTracker;

// create selection operation that:

// uses roulette wheel method and

// selects 8 parents and produces 8 offspring chromosomes

// using defined mating

Population::SelectionOperations::GaRouletteWheelSelection selection;

Population::GaSelectionSetup selectionSetup( &selection,

  &Population::SelectionOperations::GaDuplicatesSelectionParams( 8, 1, 2 ),

  &Population::GaCouplingConfig(

    Chromosome::GaMatingSetup( &mating, NULL, &matingConfiguration ) ) );

// create replacement operation that:

// replaces 8 worst chromosomes in the population

Population::ReplacementOperations::GaWorstReplacement replacement;

Population::GaReplacementSetup replacementSetup( &replacement,

  &Population::GaReplacementParams( 8 ),

  &Population::GaReplacementConfig(

    Chromosome::GaChromosomeComparatorSetup(

      &chromosomeComparator, NULL, NULL ) ));

// creates scaling operation that just copies raw fitness value

Population::ScalingOperations::GaNoScaling scaling;

Population::GaScalingSetup scalingSetup( &scaling, NULL,

  &Population::GaScalingConfig() );

// weights used to calculate fitness value

float fitnessWights[] = { 2.0f, 1.0f };

// create fitness operation that uses weighted fitness

// with specified weights

Population::GaPopulationFitnessOperationSetup fitnessSetup(

  &populationFitnessOperation,

  &Problems::XKCD::XkcdFitnessOperationParams( targetPrice ),

  &Fitness::GaFitnessOperationConfig(

    &Fitness::Representation::GaWeightedFitnessParams&lt;float&gt;(

      fitnessWights, 2 ) ) );

// population - 32 chromosomes, 0 crowding size

Population::GaPopulationParams populationParams( 32, 0,

    Population::GaPopulationParams::GAPFO\_FILL\_ON\_INIT );

// create criteria for sorting population

Population::GaFitnessComparatorSortingCriteria sortCriteria(

    fitnessComparatorSetup,

    Population::GaChromosomeStorage::GAFT\_RAW );

// create simple incremental algorithm stub:

Algorithm::Stubs::GaSimpleGAStub simpleGA(

  WDID\_POPULATION,

  WDID\_POPULATION\_STATS,

  initializatorSetup,

  fitnessSetup,

  fitnessComparatorSetup,

  populationParams,

  trackers,

  Chromosome::GaMatingSetup(),

  selectionSetup,

  Population::GaCouplingSetup(),

  replacementSetup,

  Population::GaScalingSetup(),

  sortCriteria);

// create workflow

Common::Workflows::GaWorkflow workflow( NULL );

workflow.RemoveConnection(

  \*workflow.GetFirstStep()-&gt;GetOutboundConnections().begin(), true );

// connect algorithm stub to workflow

Common::Workflows::GaWorkflowBarrier\* br1 =

  new Common::Workflows::GaWorkflowBarrier();

simpleGA.Connect( workflow.GetFirstStep(), br1 );

Common::Workflows::GaBranchGroup\* bg1 =

  (Common::Workflows::GaBranchGroup\*)workflow.ConnectSteps(

    br1, workflow.GetLastStep(), 0 );

// create stop criteria that will stop the algorithm if:

// raw fitness of the best chromosome in the population

// has not been changed for the last 1000 generations.

Algorithm::StopCriteria::GaStatsChangesCriterion stopCriterion;

Algorithm::StopCriteria::GaStopCriterionStep\* stopStep =

  new Algorithm::StopCriteria::GaStopCriterionStep(

    Algorithm::StopCriteria::GaStopCriterionSetup(

      &stopCriterion,

      &Algorithm::StopCriteria::GaStatsChangesCriterionParams(

        Population::GADV\_BEST\_FITNESS, 1000), NULL ),

    workflow.GetWorkflowData(), WDID\_POPULATION\_STATS );

// connect stop criterion to workflow and algorithm stub

Common::Workflows::GaBranchGroupTransition\* bt1 =

  new Common::Workflows::GaBranchGroupTransition();

bg1-&gt;GetBranchGroupFlow()-&gt;SetFirstStep( stopStep );

bg1-&gt;GetBranchGroupFlow()-&gt;ConnectSteps( stopStep, bt1, 0 );

workflow.ConnectSteps( bt1, simpleGA.GetStubFlow().GetFirstStep(), 1 );

// subscribe handler to event raised before new generation cycle begins

Common::Workflows::GaDataCache&lt;Population::GaPopulation&gt; population(

  workflow.GetWorkflowData(), WDID\_POPULATION );

population.GetData().GetEventManager().AddEventHandler(

  Population::GaPopulation::GAPE\_NEW\_GENERATION, &newGenHandler );

// start algorithm and wait for it to finish

workflow.Start();

workflow.Wait();