**WMH Projekt SK5­**

*Wizualizacja wybranych algorytmów heurystycznych II.*

Sprawozdanie 3

**Prowadzący projekt:**

dr inż. Sebastian Kozłowski

**Autorzy:**

Andrzej Fiedukowicz

Maciej Grzybek

**Architektura aplikacji**

Aplikacja wykonana jest zgodnie z architekturą MVC (Model Widok Kontroler). Architektura ta zapewnia możliwość zrealizowania wizualizacji algorytmu w ramach tego samego Modelu wykonywania algorytmów ewolucyjnych w różnej formie.

Model – Black Box

Model aplikacji odpowiada za przetrzymywanie stanu algorytmu i przetwarzanie jego kolejnych kroków. Model dostarcza także możliwości zmiany poszczególnych operatorów i ich parametrów zgodnie ze sprawozdaniem nr 2.

Model informuje zewnętrze o wydarzeniach w trakcie przetwarzania za pośrednictwem interfejsu obserwatora. Implementujący ten interfejs mogą rejestrować się za pomocą interfejsu Modelu.

Model przeznaczony jest do pracy w osobnym wątku a wszystkie komendy przetwarzane są asynchronicznie. Publiczny interfejs modelu jest bezpieczny wątkowo i posiada właściwość szybkiego powrotu. Przetwarzanie wykonywane jest w wątku modelu a następnie potwierdzenia są wykonywane przez interfejs obserwatorów. Warto zwrócić uwagę, że metody obserwatorów są wykonywane kosztem wątku modelu więc zalecane jest więc, by były one bezpieczne wątkowo i także posiadały właściwość szybkiego powrotu.

Warto zwrócić uwagę, że zmiany we właściwościach modelu wykonywane są leniwie, to znaczy dopiero w momencie uruchomienia wykonywania jakiegokolwiek przetwarzania stosowane są zmiany istotne i możliwe do zastosowania w danym momencie. Możliwe jest jednak wymuszenie natychmiastowego ustawiania możliwych do zmiany właściwości za pomocą dodatkowej komendy interfejsu publicznego. Taka możliwość może być cenna dla niektórych implementacji kontrolera i widoku.

Opcje operatorów wraz z ich rodzajem przekazywane są za pomocą generycznych struktur. Każdy operator ma odpowiadającą mu klasę bazową a także interfejs wizytatora, który umożliwia zastosowanie ich wewnątrz modelu zgodnie ze wzorcem wizytatora. Struktury mają prosty charakter i posiadają opcje i warianty opisane w dokumentacji wstępnej.

Publiczny interfejs modelu przedstawia się następująco:

/\* Wykonywanie komend \*/

void doStep(); // Wykonuje kolejny krok algorytmu

void doGeneration(); // Wykonuje całą iterację algorytmu

void doRestart(); // Powrót algorytmu do stanu przed inicjalizacją

void doCommit(); // Wymuszenie zastosowania zmian

void doExit(); // Zamknięcie wątku modelu po wykonaniu oczekujących operacji

void doImmidiateExit(); // Natychmiastowe zamknięcie wątku modelu

/\* Ustawianie właściwości algorytmu i jego operatorów \*/

void setFitnessFunction(const std::string& formula);

void setInitializationOptions(const common::InitializationOptions& options);

void setReproductionOptions(const common::ReproductionOptions& options);

void setMutationOptions(const common::MutationOptions& options);

void setCrossOverOptions(const common::CrossOverOptions& options);

void setRangeOptions(const common::RangeAlignmentOptions& options);

void setSelectionType(const common::SelectionOptions& options);

void setPopulationSize(const unsigned int& size);

void setGoalValue(const double& goal);

/\* Pobieranie bieżącego, ustabilizowanego stanu \*/

common::PopulationSnapshot getPopulationSnapshot();

unsigned int getGenerationId();

/\* Rejestrowanie i wyrejestrowywanie obserwatorów modelu \*/

void addObserver(common::ModelObserver\* observer);

void removeObserver(common::ModelObserver\* observer);

/\*

\* Operator służący uruchomieniu działania modelu.

\* Jest to funkcja zawieszająca, przeznaczona do uruchamiania

\* w osobnym wątku

\*/

void operator()();

Interfejs oserwatora modelu (ModelObserver) wygląda następująco:

class ModelObserver

{

public:

virtual ~*ModelObserver*();

virtual void *onStateChanged*() = 0;

virtual void *onGoalReached*() = 0;

virtual void *onProcessingStarted*() = 0;

virtual void *onProcessingStoped*() = 0;

virtual void *onFitnessFunctionApplied*() = 0;

virtual void *onInitializationOptionsApplied*() = 0;

virtual void *onReproductionOptionsApplied*() = 0;

virtual void *onMutationOptionsApplied*() = 0;

virtual void *onCrossOverOptionsApplied*() = 0;

virtual void *onRangeOptionsApplied*() = 0;

virtual void *onSelectionTypeApplied*() = 0;

virtual void *onPopulationSizeApplied*() = 0;

virtual void *onGoalValueApplied*() = 0;

};

Jak wspominano informuje on o zatwierdzeniu w modelu zmian poszczególnych parametrów, a także o osiągnięciu przez model zadanego celu. O rozpoczęciu i zakończeniu zadanego przetwarzania a także o zmianie aktualnego stabilnego stanu. Warto zwrócić uwagę, że dla każdego wołania metody typu setXXX na modelu można oczekiwać, że najpóźniej przy restarcie modelu (doRestart) zostanie wygenerowane potwierdzenie onXXXApplied.

Model – Wewnętrzna implementacja

Wewnętrznie model opiera się o działanie dwóch klas głównych model::Model oraz model::Evolution, a także wielu klas wspomagających, a w szczególności klas operatorów (model::Initializator, model::Selector, model::Duplicator, model::Mutator, model::Crosser, model::FitnessFunctioner) wraz z odpowiadającymi im fabrykami generującymi poszczególne implementacje operatorów w oparciu o zadane struktury z opcjami wykorzystując wspomniany wzorzec wizytatora.

Klasa model::Model odpowiada za działanie wątku, zarządzanie przychodzącymi operacjami i generowanie powiadomień dla obserwatorów. Jej implementacja opiera się o przerywalną pętlę nieskończoną w ramach operatora(). Pętla ta zawiesza się na kolejce blokującej w oczekiwaniu na komendę (wołania doXXX dodają elementy do tej kolejki). Odwieszenie powoduje zastosowanie zmian wynikających z wcześniejszych wołań setXXX, których polityka zmiany odpowiada bieżącemu stanowi a następnie wykonaniu zadanego polecenia.

Realizacja wołań setXXX opiera się o kolejkowanie zadań w postaci zadania ustawienia wraz z odpowiadającą mu notyfikacją zwrotną dla obserwatorów w jednej z kolejek odpowiadających polityce zmiany. Kolejki te są opróżniane w zależności od aktualnego stanu w momencie wykonywania komend (wspomniane wyżej).

Dostępne polityki zmiany to:

* INSTANT – zmiana może być wykonana w dowolnym momencie, także w trakcie przetwarzania. Funkcjonalność ta nie została zaimplementowana ponieważ nie było wołania setXXX, które mogło by mieć taką politykę zmiany. Polityka ta jest w implementacji równoważna polityce STEP.
* STEP – zmiana może zostać wykonana tylko między kolejnymi krokami algorytmu.
* GENERATION – zmian może zostać wykonana tylko między pokoleniami algorytmu.
* INITIALIZATION – zmiana może być wykonana tylko przy restartowaniu modelu do stanu przed inicjalizacją.

Klasa model::Evolution zawiera implementację algorytmu wraz z maszyną stanową. Wykonuje ona krok lub całą iterację algorytmu. Udostępnia na zewnątrz stan, by można było na jego podstawie podejmować decyzje na temat zmiany własności operatorów algorytmu:

enum State { BEFORE\_INITIALIZATION, BEFORE\_GENERATION,

BEFORE\_REPRODUCTION = BEFORE\_GENERATION,

BEFORE\_MUTATION, BEFORE\_CROSSOVER, BEFORE\_SELECTION };

Jej implementacja opiera się, o wołania zadanych operatorów, do których przekazywana jest referencja na bieżącą populację i ewentualnie dodatkowe potrzebne poszczególnym operatorom dane. Operatory te ustawiane są przez model zgodnie z polityką zmiany na wartości wyprodukowane przez fabryki poszczególnych typów operatorów przyjmujące zadane z zewnątrz struktury z właściwościami.

Taka implementacja pozwala na przejrzysty układ i możliwość oddzielenia kodu poszczególnych operatorów od generalnej struktury samego algorytmu opisanej w Sprawozdaniu nr 2.

Widok

Widok komunikuje się z częścią wykonawczą aplikacji za pośrednictwem kontrolera, z którym połączony jest przy użyciu kolejki komunikatów. W ten sposób zapewniona została dekompozycja warstwy wizualizacji od warstwy logiki (zarówno aplikacji jak i samej modelowanej dziedziny – algorytmów ewolucyjnych).

Komunikaty i ich rozpoznawanie zostało wykonane przy użyciu wzorca wizytatora. Oczekujący na komunikaty zostają powiadomieni o ich nadejściu przy użyciu mechanizmu bezpiecznej wątkowo kolejki blokującej.

Wizualizacja odbywa się przy użyciu biblioteki Qt, w wersji 5.2.0, która umożliwia wyabstrahowanie aspektów tworzenia aplikacji typu GUI zależnych od platformy programowej i sprzętowej. Taka decyzja pozwala niewielkim nakładem pracy stworzyć wersję aplikacji zdatną do uruchomienia na wielu platformach.

Widok wyświetla funkcję przystosowania osobników na siatce dwuwymiarowej. Poprzez zwiększanie udziału składowej koloru, którym rysowana jest funkcja, oznaczane są jej rosnące wartości, tak by wraz ze wzrostem wartości rósł udział koloru. Np. kolorem czarnym oznaczona jest najniższa wartość funkcji przystosowania w określonej dziedzinie. Na siatce uwzględnione są pozycje osobników z danej populacji. Osobnicy rysowani są odmiennym kolorem, spoza palety barw używanej do rysowania tła (funkcji przystosowania).

Obok bieżącego stanu populacji na tle funkcji przystosowania, znajdują się kontrolki służące do zmiany ustawień operatorów algorytmu, zgodnie ze sprawozdaniem nr 2.

Kontroler

Zadaniem kontrolera jest koordynowanie pracy zleconej przez użytkownika za pośrednictwem Widoku oraz informacji zwrotnej z modelu.

Przechowuje on stan aplikacji i reaguje na zdarzenia użytkownika. Powoływany jest do życia w osobnym wątku, by móc zawieszać się w oczekiwaniu na zdarzenia od widoku bądź modelu – działa reaktywnie.

Kontroler synchronizuje pracujący w osobnym wątku Widok, z wykonującym asynchronicznie obliczenia Modelem.

**Zmiany w stosunku do założeń wstępnych (Sprawozdania 2.)**

* Ze względu na brak dostarczenia przez Microsoft standardu opisu plików obiektów o dużej ilości sekcji, nie było możliwe skompilowanie biblioteki exprtk (dla wyrażeń funkcyjnych) w ramach systemów Windows przy użyciu kompilatora innego niż Microsoft Visual Studio. Ponieważ jako kompilator do tworzenia dystrybucji aplikacji został wybrany MinGW (ze względu na względną kompatybilność z GCC i ułatwione dystrybuowanie jego runtime’u współpracującego z biblioteką Qt), wspomniana biblioteka została zmieniona na bibliotekę muParser oferującą podobne funkcjonalności.
* Ze względu na mało wygodne i intuicyjne dla użytkowników założenia dotyczące rozkładów w operatorach krzyżowania opartych o losowość, zostały one zmienione tak, że podaje się rozkład niezależnie dla poszczególnych argumentów funkcji celu a wylosowane wartości są wagami krzyżowania (dotyczy rozkładu Uniwersalnego i Gaussa)
* Zostały dodane dwa dodatkowe operatory krzyżowania, które wybierają poszczególne cechy osobników w całości od jednego z rodziców – z prawdopodobieństwem ½ lub z prawdopodobieństwem zależnym od jakości rodziców.
* Zrezygnowano z wykorzystania biblioteki evol do implementacji modelu aplikacji ze względu na brak dostosowania implementacji tej biblioteki do przewarzania krokowego. Zgodnie z dokumentacją tejże biblioteki (jako, że problem wymaga nadpisania większości jej funkcjonalności bazowych) jest to przesłanką do zrezygnowania z jej użycia.
* W ramach testów, które miały być oparte o testy biblioteki evol (z której zrezygnowano) wykonano zestaw smoke testów, które zakończyły się powodzeniem. Ze względu na kompleksowość przetwarzania nie zostały stworzone testy integracyjne, gdyby jednak aplikacja miała być dalej rozwijana należało by stworzyć dodatkowo testy regresji.

**Załączniki**

* Repozytorium kodu – <http://github.com/maciejgrzybek/EvolViz>
* Gotowa paczka uruchomieniowa zawierająca komplet bibliotek niezbędnych do bezproblemowego uruchomienia aplikacji EvolViz.exe – <http://fiedukowicz.abajt.pl/EvolViz.zip>