POLITECHNIKA WARSZAWSKA

Wydział Elektroniki i Technik Informacyjnych
Instytut Automatyki i Informatyki Stosowanej

GOOL - biblioteka metod optymalizacji globalnej

Opis systemu

Marek Publicewicz

Warszawa 2003

Spis treści

Rozdział 1

Wstęp

Niniejszy podręcznik stanowi szczegółowy opis obiektowej wersji biblioteki metod optymalizacji globalnej GOOL [1], będącej kompleksowym narzędziem służącym do badania oraz rozwiązywania zadań programowania nieliniowego przy wykorzystaniu metod optymalizacji globalnej. Opracowanie to służyć ma jako pomoc przy wykorzystaniu oraz potencjalnej rozbudowie systemu, dlatego pozbawione jest (nieistotnych z punktu widzenia użytkownika) szczegółów dotyczących podstaw teoretycznych zastosowanych algorytmów. Zainteresowanych odsyłamy do pracy [9], która zawiera stosowne informacje oraz prezentuje i omawia wyniki obliczeń, uzyskane przy wykorzystaniu systemu do rozwiązania dwóch przykładowych zadań.

System GOOL, będący następcą programu VSO2 [8], składa się z trzech zasadniczych części:

- Środowisko wizualizacyjne (GOOL/GUI), stanowiące interfejs pomiędzy użytkownikiem, a zaimplementowanymi metodami, umożliwiające definiowanie zadań i nadawanie wartości parametrom poszczególnych metod oraz wygodny sposób prezentacji sposobu działania algorytmów.

- Biblioteka metod optymalizacji globalnej (GOOL/OM [2]), w skład której wchodzą również wykorzystywane przez nie metody lokalne.

- Biblioteka generatorów losowych (GOOL/RG [3]), pozwalająca na korzystanie w metodach optymalizacji z bardziej wyrafinowanych sposobów generowania liczb pseudolosowych, niż zapewnia to standardowy generator losowy zawarty w bibliotece języka C. Możliwe jest również wykorzystanie biblioteki generatorów poza biblioteką GOOL/OM, do generowania punktów o zadanym rozkładzie.

Dodatkowo w skład systemu wchodzi wersja przeznaczona do działania w środowisku tekstowym (GOOL/CON), pobierająca dane od użytkownika z wejściowych plików tekstowych, zapewniająca pełną funkcjonalność z zakresu znajdowania minimum globalnego podanego zadania za pomocą wybranego algorytmu.

Do istotnych zmian w stosunku do VSO2 należą:

- przejście ze zbioru pojedynczych funkcji implementujących poszczególne metody optymalizacji na podejście obiektowe - zdefiniowanie zbioru klas abstrakcyjnych określających zachowanie się systemu, a następnie zaimplementowanie metod jako klas potomnych, realizujących odpowiednie metody wirtualne,

[1] Global Optimization Object-oriented Library

[2] Global Optimization Object-oriented Library/Optimization Methods.

[3] Global Optimization Object-oriented Library/Random Generators.

- utworzenie niezależnej hierachii obiektowej dla celów biblioteki generatorów losowych oraz interfejsu pozwalającego na ich wykorzystanie podczas rozwiązywania zadań,
- lepsze usystematyzowanie metod (dzięki utworzonej hierarchii), umożliwienie rozwiązywania podzadań w obrębie zadania głównego (oraz ich podzadań - zagłębienie do dowolnego poziomu, ustalonego przez zasoby systemowe),
- umożliwienie formułowania i rozwiązywania zadań z funkcyjnymi ograniczeniami nierównościowymi poprzez zastosowanie konfigurowalnej zewnętrznej funkcji kary,
- rozszerzenie sposobu definiowania zadań o nowe tokeny, pozwalające m.in. na lepszą parametryzację zadań, stosowanie funkcji iterowanych (suma, iloczyn), wyrażeń iterowanych (przydatnych m.in. do definiowania grup ograniczeń nierównościowych). Umożliwienie posługiwania się indeksami do tworzonych wyrażeń i parametrów, operowanie aliasami wyrażeń,
- dodanie do modułu obsługi zadań nowych funkcji (sinh, cosh, min, max, sqrt),
- zapewnienie pełnej funkcjonalności wersji nieokienkowej biblioteki w stosunku do wersji z wizualizacją. Tym samym cała funkcjonalność obliczeniowa została przeniesiona do jezyka C++ powodując, iż GOOL można uznać za program napisany zgodnie z trójwarstwowym modelem Model-View-Controller,
- pozbycie się nadmiarowego kodu, który był niezbędny dla strukturalnej postaci biblioteki VSO2/OM. Postać obiektowa umożliwiła m.in. lepsze uporządkowanie metod siatki,
- ułatwienie korzystania z biblioteki generatorów losowych poprzez wprowadzenie odrębnej klasy *zarządcy generatorów*,
- zmiany i rozszerzenia w istniejącej implementacji metod optymalizacji:
  - dopisanie nowego schematu grupowania w algorytmie Törna,
  - modyfikacje istniejących metod siatki aktywnej,
  - implementacja algorytmu ewolucyjnego z kodowaniem rzeczywistoliczbowym oraz odpowiednimi operacjami,
  - dodanie implementacji metody Galperina,
  - udane zaimplementowanie oryginalnego sposobu podziału komórki w metodzie Meewella-Mayne w oparciu o sympleks liniowy [4],
  - rozszerzenie schematów wyżarzania w metodzie SA [5].

W sposób naturalny za język bazowy systemu przyjęto C++. Mając na uwadze kryterium efektywności obliczeniowej, wykorzystany został on głównie do zarządzania strukturą. Świadomie zrezygnowano z niektórych mechanizmów (np. wyjątków, szablonów, przeciążania operatorów), które jakkolwiek eleganckie, mogą stanowić przyczynę obniżenia efektywności przeprowadzanych obliczeń. Podczas projektowania systemu w kilku

[4] W systemie VSO2 metoda ta wykorzystywała wielowymiarowe uogólnienie metody Pijavskiego-Schuberta.

[5] Simulated Annealing

istotnych miejscach posłużono się tzw. *wzorcami projektowymi*, dokładniej przedstawionymi w pozycji [4], powszechnie używanymi podczas tworzenia dużych obiektowych projektów informatycznych. Wprowadzone zmiany zwiększając użyteczność programu jednocześnie w znaczący sposób uprościły potencjalne próby jego rozbudowy.

Praca składa się z sześciu rozdziałów oraz jednego dodatku:

- pierwszy rozdział zawiera opis funkcjonalny systemu, stanowiący przegląd możliwości oferowanych przez program, jak również omawiający założenia jego ogólnej struktury i sposobu jej implementacji,
- rozdział drugi opisuje sposób wykorzystania programu GOOL/CON do rozwiązywania zadań w trybie wsadowym,
- pracy w trybie interakcyjnym w środowisku okienkowym poświęcony jest rozdział trzeci, prezentujący poszczególne okna dialogowe dostępne w programie GOOL/GUI,
- rozdział czwarty, poświęcony implementacji bibliotek może być traktowany jako część dokumentacji programistycznej. Zawiera omówienie najważniejszych klas programu, realizacje najistotniejszych problemów związanych z procesem rozwiązywania zadań przez system. Wyszczególnione są w nim również wszystkie parametry zaimplementowanych algorytmów optymalizacji globalnej i lokalnej,
- rozdział piąty napisany został z myślą o maksymalnym ułatwieniu rozbudowy systemu GOOL o nowe metody optymalizacji. Zawarte są w nim informacje dotyczące rekompilacji jądra, formatu plików konfiguracyjnych oraz przedstawiony jest proces dodania nowo zaimplementowanej metody,
- rozdział szósty pokazuje, w jaki sposób można wykorzystać kod bibliotek GOOL/OM oraz GOOL/RG we własnych programach, bez konieczności wywoływania programu w postaci wsadowej,
- dodatek A omawia pokrótce wykorzystywaną w programie i stworzoną specjalnie na jego potrzeby bibliotekę obsługi wyrażeń symbolicznych.

Rozdział 2

Opis funkcjonalny systemu GOOL

2.1 Wprowadzenie

W swoim zamierzeniu, system GOOL miał być programem, który pozwalałby na efektywne rozwiązywanie zadań optymalizacji funkcji z ograniczeniami oraz jednocześnie pozwalał opcjonalnie na wizualizację przebiegu procesu znajdowania rozwiązania.

Ponieważ wizualizacja wprowadza konieczność dodania kodu komunikacyjnego pomiędzy warstwą modelu, a warstwą prezentacji i przyczynia się do spowolnienia procesu obliczeniowego - stworzono dwie wersje programu. Pierwsza z nich pozwala na rozwiązanie pojedycznego zadania optymalizacji metodą wybraną spośród wszystkich zdefiniowanych w systemie. Druga jest wzbogacona o graficzny interfejs użytkownika, który pozwala na wizualizację przebiegu optymalizacji oraz prezentuje charakterystyki funkcji celu zadania. Dodatkowo wersja druga umożliwia łatwą edycję parametrów definiujących zadanie oraz opcji algorytmów używanych do minimalizacji.

W efekcie obie wersje znakomicie się uzupełniają - wersja okienkowa może służyć do wstępnego rozpoznania charakteru zadania, wersja tekstowa (wsadowa) do efektywnego znalezienia jego rozwiązania.

Projektując program przyjęto, że powinien być on napisany w sposób zapewniający:

- separację logiki programu od warstwy prezentacyjnej oraz warstwy odpowiedzialnej za dialog z użytkownikiem,
- sposób implementacji mający na uwadze możliwie minimalne wykorzystanie zasobów pamięciowych przez metody optymalizacji, przy jednoczesnym zapewnieniu maksymalnej efektywności obliczeniowej,
- prostotę konstruowania i edycji zadań [1] przeznaczonych do optymalizacji - w tym umożliwienie rozwiązywania zadań dla funkcji celu o różnej wymiarowości, parametryzację używanych symboli, łatwość definiowania ograniczeń,
- łatwość modyfikowania zakresów zmiennych, parametrów metod optymalizacji oraz opcji związanych z rozdzielczością rysowanych wykresów,
- przeglądanie i analizę wyników testów metod po ich zakończeniu,
- graficzną komunikację z użytkownikiem,
- prostotę implementacji i rozbudowy programu,

[1] Pod pojęciem zadania rozumiemy funkcję celu wraz z narzuconymi ograniczeniami kostkowymi i funkcyjnymi.

- przenośność programu.

Do kodowania graficznego interfejsu systemu GOOL/GUI użyto języka skryptowego Tcl/Tk, który umożliwia łatwą modyfikację i rozbudowę części prezentacyjnej oraz dialogowej systemu. Zapewnia on również niezależność od konkretnego systemu operacyjnego Jedynym ograniczeniem jest dostępność biblioteki języka Tcl/Tk dla docelowej platformy.

Stworzono dwie wersje programu, przeznaczone do działania odpowiednio pod systemami operacyjnymi rodzin Unix i Windows. Kod źródłowy jest wspólny (w kilku zaledwie miejscach tam, gdzie było to niezbędne, zastosowano kompilację warunkową). Obie wersje programu wymagają natomiast wykorzystania różnych (w sensie implementacji) bibliotek języka Tcl/Tk. Do kompilacji obu wersji środowiska z kodu źródłowego należy użyć odpowiednich plików makefile, dostarczonych razem z wykonanym oprogramowaniem.

Biblioteka metod optymalizacji została zaimplementowana w języku C++ (fragmenty krytyczne - jak np. obliczanie funkcji celu - w języku C) z następujących powodów:

- poziom skomplikowania biblioteki GOOL/OM - wymusił zastosowanie metodologii obiektowej dla poprawienia przejrzystości i łatwości modyfikacji jądra obliczeniowego systemu,
- jest to język efektywny [2], dodatkowo (zdaniem autora jest to zaleta, nie wada) pozwala na bezpośrednie zarządzanie tworzeniem oraz zwalnianiem struktur dynamicznych, [3]
- dołączenie metod optymalizacji zaimplementowanych w C [4] nie wymuszało ich całkowitego przepisania, a jedynie przekonstruowania i dostosowania do utworzonej hierarchii klas (co często prowadziło de facto do uproszczenia metod strukturalnych),
- wydajna implementacja podsystemu obsługi definiowania zadań w języku C zapewnia dużą efektywność obliczeń przy zachowaniu elastyczności w zarządzaniu zadaniami,
- język Tcl/Tk umożliwia rozszerzanie interpretera o nowe polecenia, w tym momencie służące jako pomost pomiędzy systemem okien dialogowych, a jądrem biblioteki. Polecenia te muszą być implementowane w języku C, z którego bardzo łatwo komunikować się z klasami stworzonymi w C++.

System GOOL pozwala na definiowanie zadań, tj. wprowadzanie funkcji celu dowolnej liczby zmiennych, definiowanie ograniczeń nierównościowych oraz wywoływanie metod optymalizacji w celu znalezienia poszukiwanego minimum, z umożliwieniem szczegółowej obserwacji przebiegu działania danej metody (podczas korzystania z interfejsu okienkowego GOOL/GUI). Poszczególne ograniczenia nierównościowe mogą być niezależnie deaktywowane w procesie obliczeniowym.

System przetestowano pod systemami operacyjnymi: Linux (jądro 2.4.22, Active-State Tcl/Tk wersja 8.4.4), Windows 2000 (Active-State Tcl/Tk, wersja 8.4.4), FreeBSD 4.7 (tylko jądro systemu GOOL, tj. biblioteki GOOL/OM i GOOL/RG, bez graficznego interfejsu użytkownika).

[2] Czego jeszcze przez długi czas nie będzie chyba można powiedzieć o języku Java.

[3] W szczególności przy poprawnej obsłudze dealokacji mamy pewność, że struktury cykliczne zostaną NA PEWNO usunięte z pamięci dokładnie tym momencie, w którym tego żądamy. W przypadku np. Javy tej pewności nie ma.

[4] Dostępnych w wersji VSO2/OM.

2.2 Opis funkcjonalny systemu GOOL/CON

Ponieważ jest to część bazowa całego systemu (system okienkowy GOOL/GUI jest w dużej mierze jej rozszerzeniem), opis działania systemu jako całości rozpoczniemy właśnie od niej. Jest to aplikacja przeznaczona do uruchomienia w środowiskach Linux/Windows, pozwalająca na definiowanie zadań minimalizacji funkcji wielowymiarowych z ograniczeniami nierównościowymi. Jest w całości zaimplementowana w języku C++ (oraz w C) i do jej uruchomienia nie są potrzebne żadne biblioteki, poza standardowymi bibliotekami systemowymi zawierającymi implementację funkcji standardu ANSI C++.

Wersja GOOL/CON umożliwia wykorzystanie PEŁNEJ funkcjonalności obliczeniowej zaimplementowanych algorytmów. Wykorzystanie tej wersji jest zalecane do rozwiązania złożonych zadań optymalizacji, gdyż jest ona wydajniejsza, m.in. ze względu na nieobecność kodu [5] zapewniającego komunikację metod z oknami dialogowymi.

Użytkownik dzięki systemowi GOOL/CON może:

- definiować zadanie przeznaczone do optymalizacji. Na zadanie składa się: minimalizowana funkcja celu, ograniczenia kostkowe i funkcyjne, definicje symboli pomocniczych oraz pewien zbiór parametrów ogólnych zadania,
- określić algorytm, który ma zostać użyty do rozwiązania zadania oraz ustawić dostępne parametry algorytmu,
- dokonać optymalizacji zdefiniowanego zadania wybranym algorytmem biblioteki GOOL/OM.

2.3 Opis funkcjonalny systemu GOOL/GUI

Program GOOL/GUI jest aplikacją pozwalającą na wizualizację działania różnych metod optymalizacji globalnej, uruchamianą w środowisku X Windows systemu UNIX/Linux lub w systemach Windows 95/98/2000 [6].

W kontekście aktywnego zadania możliwe jest wyświetlenie wartości funkcji celu dla podanego argumentu, jak również wartości odpowiadających mu ograniczeń nierównościowych.

Liczba zadań wczytanych do programu jest dowolna, ale tylko jedno z nich może być w danej chwili aktywne i nie można go zmieniać podczas obliczeń Funkcje celu wprowadzane są na dwa sposoby: w postaci analitycznej - wówczas program automatycznie określa wymiar oraz próbuje wyznaczyć wzory na kolejne pochodne cząstkowe; oraz w postaci procedury napisanej w języku Tcl, zwracajacej wartość funkcji w zależności od przekazanego argumentu.

Dla lepszego zobrazowania funkcji celu dostępne są narzędzia do rysowania wykresów poziomicowych funkcji na płaszczyźnie. Do rysowania poziomic system wykorzystuje jeden z trzech dostępnych w systemie GOOL algorytmów:

- oryginalny algorytm, opisany w [12],
- algorytm zastosowany po raz pierwszy w wersji VSO2 [8, 2],
- algorytm wprowadzony w wersji GOOL, oparty na schemacie zaproponowanym przez Snydera, zaimplementowanym oryginalnie w języku Fortran [10].

[5] Włączanie/wyłączanie fragmentów kodu zrealizowane jest odpowiednimi dyrektywami warunkowymi kompilatora umieszczonymi w kodzie źródłowym.

[6] Po zainstalowaniu interpretera języka Tcl/Tk.

Użytkownik w opcjach programu wybiera, który algorytm jest aktualnie wykorzystywany.

W przypadku funkcji wielowymiarowych ($n > 2$) użytkownik wybiera zmienne do prezentacji (parę). Pozostałym, niewyświetlanym zmiennym, przypisuje natomiast ustalone wartości. Podczas pracy programu można utworzyć dowolną liczbę okien z poziomicami dla różnych funkcji, dowolnie je przeskalowywać i wykonywać na nich różne operacje. Wykresy poziomicowe oprócz dostarczania informacji na temat charakteru funkcji służą również do wizualizacji przebiegu wywoływanych algorytmów optymalizacji - punkty znajdowane w kolejnych krokach metody są umieszczane w aktywnych oknach z wykresami poziomicowymi, pozwalając na lepszą obserwację sposobu działania danego algorytmu w odniesieniu do konkretnej funkcji. Wizualizacja nie sprowadza się tu tylko do pokazywania kolejnego najlepszego rozwiązania znalezionego w danej chwili - w zależności od charakteru aktywnej metody, obserwować możemy zmiany w całych populacjach punktów (np. metody z grupy CRS), przesuwanie pojedynczego punktu (np. metoda Griewanka), czy też położenie oraz rozmiar aktualnie najlepszej komórki w przypadku, gdy badane są metody siatki. Kolejne wyznaczane punkty mogą być umieszczane jednocześnie w kilku oknach graficznych przedstawiających wykresy poziomicowe jednej funkcji dla różnych zakresów zmiennych.

W przypadku metod, w których w każdym kroku otrzymujemy jeden, nowy punkt, wyniki z różnych przebiegów są doklejane do aktywnych okien poziomicowych, tworząc kolejne serie danych, zaś użytkownik poprzez zastosowanie hierarchicznego, dynamicznie tworzonego menu może w każdym oknie dowolnie zarządzać wyświetlaniem poszczególnych serii danych.

Wszystkie dane otrzymywane podczas eksperymentu numerycznego, jak i utworzone w programie funkcje celu, mogą zostać zapisane na dysku, co daje możliwość ich późniejszego odtwarzania. Dodatkowo (tylko dla funkcji dwumiarowych) możliwe jest wykonanie wykresu trójwymiarowego.

System GOOL/GUI może być również wykorzystany do prezentacji działania algorytmu optymalizacji nie wchodzącego w skład dołączonej biblioteki metod. Kolejne punkty są odczytywane z uruchomionego uprzednio programu poprzez przekierowanie strumieni wejścia/wyjścia i umieszczane w aktywnych oknach z wykresem poziomicowym.

Bardzo ważną funkcją GOOL/GUI jest umożliwienie oglądania przebiegu minimalizacji w trybie edukacyjnym. Użytkownik w każdym kroku metody dostaje obraz wartości najważniejszych parametrów związanych z daną metodą, może obserwować krok po kroku przebieg wybranego algorytmu minimalizacji kierunkowej. Po zakończeniu przebiegu symulacji program dostarcza informacji dodatkowych, stanowiących podsumowanie działania algorytmu. Są one prezentowane w postaci graficznej. Informacje te mogą być wykorzystane do ustalenia parametrów metody i dostrojenia jej do danego problemu.

Ponadto w bibliotece zaimplementowane zostały różne generatory losowe oraz sekwencje pseudolosowe [7], które są używane zarówno podczas działania metod optymalizacji o charakterze losowym, jak i mogą być użyte całkowicie od nich niezależnie, w celu wygenerowania określonej liczby punktów (wielowymiarowych) zgodnie z zadanym rozkładem. Punkty te oczywiście można zapisywać do pliku, jak i z plików pobierać, w celu wykreślenia np. wykresu gęstości (jednowymiarowy histogram) czy też populacji utworzonych punktów zrzutowanej na wyspecyfikowane przez użytkownika dwa wymiary (wykres na płaszczyźnie).

[7] Implementacja stanowi w dużej części adaptację procedur stworzonych przez dr inż. J. Arabasa z Instytutu Systemów Elektronicznych Politechniki Warszawskiej, wchodzących w skład biblioteki algorytmów ewolucyjnych *GABI*, opisanej w pracy [1].

Reasumując, użytkownik korzystający z programu ma następujące możliwości:

- wprowadzanie i edycja funkcji celu przeznaczonych do optymalizacji,
- wprowadzanie i edycja związanych z funkcją celu ograniczeń funkcyjnych,
- aktywowanie/deaktywowanie pojedynczych ograniczeń zadania,
- zapisywanie i wczytywanie zadań z plików dyskowych,
- zadawanie funkcji celu zarówno poprzez podanie wzoru analitycznego (wówczas program automatycznie wyznacza wzory na pochodne cząstkowe), jak i poprzez przygotowanie procedury obliczającej wartość minimalizowanej funkcji,
- definiowanie i edycja zakresów zmiennych zadania,
- dla funkcji jednowymiarowych - rysowanie wykresu liniowego funkcji celu,
- dla funkcji dwuwymiarowych - rysowanie trójwymiarowych wykresów wprowadzonych funkcji celu,
- zobrazowanie funkcji celu w postaci wykresów poziomicowych przy wykorzystaniu jednego z trzech dostarczonych algorytmów,
- wykonywanie na oknach graficznych różnych operacji, np. skalowanie, zmiana zakresów zmiennych, nanoszenie siatki na wykres, ukrywanie i zamykanie okien, ukrywanie i pokazywanie danych z optymalizacji, wybór sposobu przedstawiania danych (linia, punkty),
- wybór aktywnych okien przeznaczonych do badania metody - wyniki działania metody można umieszczać w kilku oknach jednocześnie, ale muszą być one związane z jedną funkcją (np. dla różnych zakresów zmiennych, czy też różnych zestawów par zmiennych),
- ustalanie parametrów metod optymalizacji poprzez system okien dialogowych,
- wybór rodzaju eksperymentu: metoda optymalizacji z wbudowanej biblioteki, metoda w postaci pliku wykonywalnego (wysyłającego na standardowe wyjście kolejne punkty), punkty pobierane z pliku tekstowego o ustalonym formacie,
- zapisywanie na dysku wyników optymalizacji z możliwością późniejszego ich odtworzenia,
- tworzenie, zapisywanie oraz odczytywanie zbiorów punktów, będących wynikiem działania określonego generatora losowego; wyświetlanie punktów na płaszczyźnie, wykreślanie histogramów,
- dokładne śledzenie przebiegu optymalizacji poprzez włączenie specjalnego trybu ***Educational Mode***.

Program napisany jest jako skrypt w języku Tcl/Tk i można uruchamiać go na dwa sposoby:

1. Korzystając ze standardowego, zainstalowanego w systemie interpretera języka Tcl/Tk. Przy takim uruchomieniu niedostępne są metody optymalizacji z biblioteki metod, funkcje generacji punktów losowych, automatyczne różniczkowanie funkcji, pełna wizualizacja przebiegów. Niemniej można wówczas wykonywać badania, uruchamiając inne programy lub wczytywać dane z plików.

2. Korzystając z interpretera dostarczonego wraz z programem, zawierającego wkompilowaną bibliotekę metod optymalizacji.

W pierwszym przypadku program jest w pełni przenaszalny pomiędzy systemami, w których jest zainstalowany interpreter języka Tcl/Tk, natomiast w drugim może być konieczne przekompilowanie jądra interpretera.

Rozdział 3

Opis działania programu GOOL/CON

3.1 Uruchomienie programu

Wywołanie programu GOOL/CON polega na wpisaniu komendy w linii poleceń:

gool_con task_file [methods_file]

Argument *task_file* jest obowiązkowy i powinien zawierać nazwę pliku definiującego rozpatrywane przez nas zadanie. Drugi argument, związany z określeniem algorytmu minimalizacji oraz jego parametrów dla tego zadania jest opcjonalny. W przypadku jego braku, zostaje wywołany algorytm *CRS3* .

Program tworzy obiekty zadania oraz odpowiednich metod optymalizacji, a następnie uruchamia proces obliczeniowy. Po jego zakończeniu zostaje wyświetlony raport z przebiegu optymalizacji, na który składają się:

- parametry zadania (dokładność obliczeń, postać funkcji kary, użyte generatory rozkładu),
- znalezione minimum (wektor $\hat{x}$),
- odpowiadająca mu wartość funkcji celu,
- wartość kryterium STOP-u, które spowodowało zakończenie algorytmu,
- numer iteracji, w której znaleziono minimum,
- wartości ograniczeń zadania dla tego punktu,
- liczba uderzeń w ograniczenia,
- całkowita liczba iteracji,
- liczba wywołań funkcji celu,
- czas trwania optymalizacji,
- opcjonalnie, jeśli algorytm operował na grupie punktów - wyświetlenie k (jest to parametr zadania) najlepszych punktów ze zbioru. Umożliwia to np. odczyt k minimów lokalnych, znalezionych przez algorytm Törna.

3.2 Format pliku zadania

Plik **task_file** powinien składać się z następujących sekcji:

- <*NAME*> - nazwa zadania [1],
- <*DESCRIPTION*> - opis słowny zadania,
- <*DIMENSION*> - wymiar zadania,
- <*BOUNDS*> - ograniczenia kostkowe w postaci wektorów,
- <*SYMBOLS*> - definicje symboli pomocniczych, służących do definiowania bardziej skomplikowanych wyrażeń,
- <*OBJECTIVE*> - wzór na wartość funkcji celu lub nazwa programu zwracającego wartość funkcji celu dla przekazanego argumentu wejściowego (symulator),
- <*CONSTRAINTS*> - definicje ograniczeń nierównościowych zadania,
- <*PARAMS*> - wartości parametrów zadania takie, jak: dokładność dla przeprowadzanych obliczeń, poszukiwana wartość funkcji celu (opcjonalna) oraz parametr określający minimalną wartość zbliżenia się do tej wartości, której osiągnięcie kończy zadanie. Do tej grupy należą również parametry związane z konstrukcją funkcji kary zadania oraz informacje o generatorach losowych,
- *GRADIENTS* - postać analityczna gradientu zdefiniowanego wcześniej symbolu. Specjalny symbol *objective* określa funkcję celu zadania,
- <*START_POINTS*> - punkty startowe optymalizacji dla tego zadania (opcjonalne).

Linie zaczynające się od znaku # są traktowane jako komentarz i ignorowane.

Poniżej przedstawiony jest plik dla przykładowego zadania, będącego uproszczoną postacią modelu optymalizacji cen dla wariantu Cobba-Douglasa:

```
<NAME>
Cobb-Douglas 1
<DESCRIPTION>
Zadanie minimalizacji cen w oparciu o model Cobba-Douglasa.
<DIMENSION>
3
<BOUNDS>
[1, 1, 2 ]
[10, 7, 10 ]
<SYMBOLS>
alfa = [ 1, 2, 3 ]
beta = [ -0.2, 0.01, 0.1 ; 0.4 , -0.2, 0.1 ; 0.3, 0.5, -0.3 ]
cost = [ 1, 2, 3 ]
Si_j(1:3) = alfa_j * mul_i(1:3)(pow(x_i,beta_j_i))
<OBJECTIVE>
```

[1] Z opisanego powyżej formatu korzysta również wersja okienkowa GOOL/GUI. W przypadku wersji GOOL/CON sekcje *NAME* oraz *DESCRIPTION* są niewykorzystywane.

```
sum_i(1:3)((cost_i - x_i) * Si_i)
<CONSTRAINTS>
g11 = x_1 - x_2 - 4
<GRADIENTS>
g11 = [1, -1]
<PARAMS>
# dokladnosc obliczen
epsilon=10e-5
# poszukiwana wartosc funkcji celu i zadowalajacy nas prog jej osiagniecia
fval=50;fval_eps=0.01
# skorzystamy z generatora nr 1 dla rozkladu jednostajnego
# oraz generatora nr 2 dla rozkładu normalnego
Uniform=1;Gaussian=2
# funkcja kary bedzie miala postac: 10 * max(g_i(x), 0) ^ 2
penaltyf=10;penaltyp=2
# po zakończeniu działania dla metod wielopunktowych chcemy
# obejrzeć punkt najlepszy oraz 2 najlepsze punkty ze zbioru roboczego
end_solutions=3
```

Choć nazwa pliku jest dowolna, jako konwencję przyjęto, że pliki tego rodzaju w systemie GOOL mają rozszerzenie **.tsk**.

3.3 Format pliku metod

Założenia tu są podobne, jak w przypadku pliku definiującego zadanie. Plik składa się ze zbioru sekcji, które w tym przypadku są następujące:

- <*METHODS*> - powinna zawierać nazwę symboliczną algorytmu biblioteki GOOL/OM, który zostanie wybrany do rozwiązania zadania. Tylko jedna metoda zostanie uwzględniona w ten sposób, tj. istotna jest tylko pierwsza linia pierwszej takiej sekcji w pliku,
- <*PARAMS*> - sekcja powinna składać się z kolejnych linii specyfikujących opcje metody, które chcemy ustawić. Mogą to być opcje ogólne (np. maksymalna liczba iteracji), lub szczególne, związane z konkretną metodą.

Nazwa algorytmu musi odpowiadać jednemu z kodów symbolicznych metod zdefiniowanych w bibliotece GOOL/OM. Wszystkie dostępne kody zostały wyszczególnione w sekcji 5.5, opisującej szczegóły dotyczące implementacji i wykorzystania poszczególnych algorytmów. W przypadku podania błędnego kodu, metoda nie zostanie wybrana i program w efekcie dokona optymalizacji przy użyciu metody domyślnej, jaką jest algorytm *CRS3* .

Przykładowy plik może wyglądać np. tak:

```
<METHOD>
CRS23
<PARAMS>
NP=10
max_iter=20
```

UWAGA: Jeśli chcielibyśmy bardziej szczegółowo wyspecyfikować warunki optymalizacji np. algorytmy lokalne używane przez wybrany algorytm (o ile używa on takowych),

należy skorzystać z wariantu opisanego w rozdziale 7.

Plik metod jest używany tylko przez program GOOL/CON. [2]

[2]Wersja okienkowa ze względu na konieczność dokładnej wizualizacji przebiegu działania metod, posługuje się osobnym formatem.

Rozdział 4

Opis działania programu GOOL/GUI

4.1 Uruchomienie programu

W środowisku X Window System program GOOL/GUI można uruchomić na kilka sposobów. Jednym z nich jest uruchomienie z konsoli shell standardowego interpretera języka Tcl/Tk lub interpretera rozszerzonego o bibliotekę metod optymalizacji (jego nazwa to **gool_gui**). Po pojawieniu się na konsoli znaku zachęty „%” oraz pustego okna aplikacyjnego na ekranie, należy wczytać główny plik programu - **gool.tcl**, używając polecenia *source*:

```
# wish
% source gool.tcl
%
```

Puste okno powinno zamienić się w aplikację. Innym, bardziej naturalnym sposobem, jest umieszczenie w pierwszej linii głównego pliku **gool.tcl**, po znakach "#!", ścieżki dostępu do interpretera, np.

```
#!/usr/local/bin/gool_gui
#!./wish
```

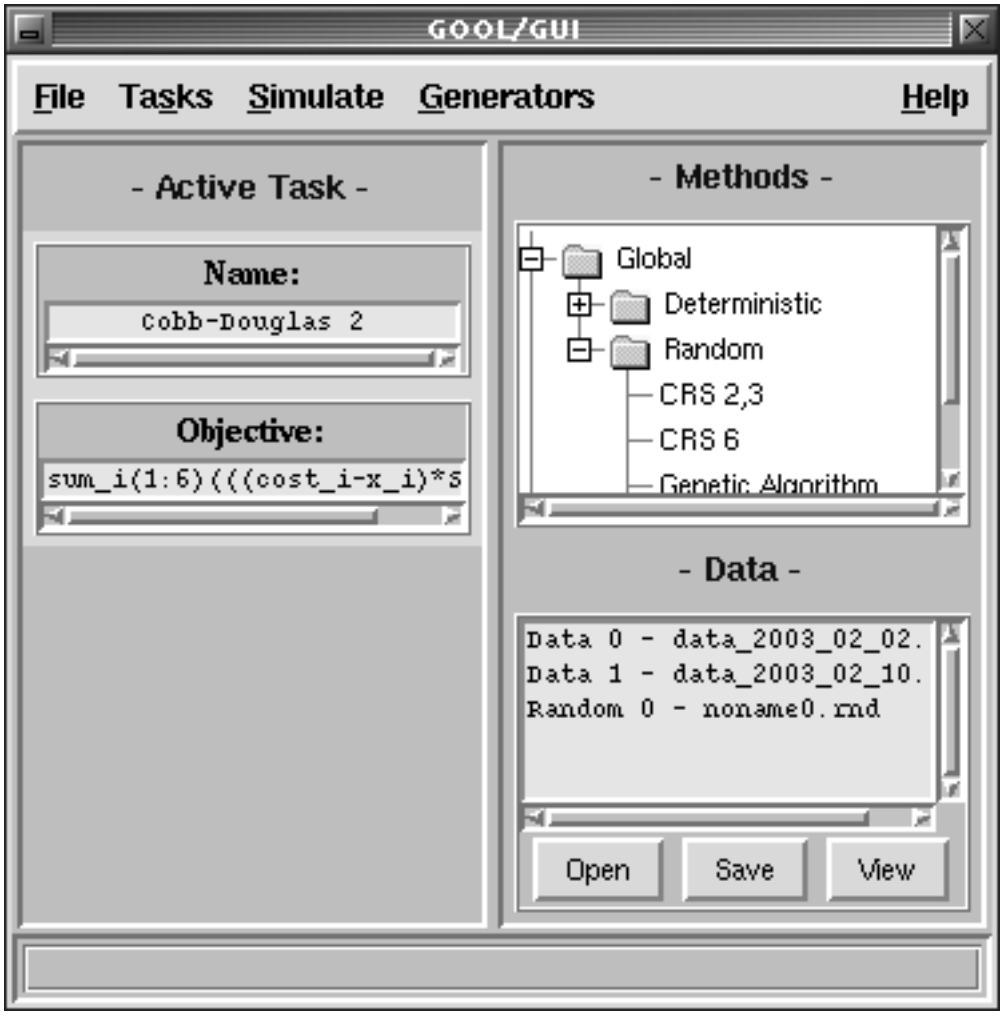
Teraz można już uruchomić aplikację jak zwykły program wykonywalny.

Jeszcze innym sposobem jest podanie w kodzie źródłowym interpretera nazwy pliku głównego programu jako skryptu startowego. Po skompilowaniu i uruchomieniu interpretera, wczytuje on skrypt startowy (czyli program) i aplikacja pojawia się na ekranie.

W systemach rodziny Windows po zainstalowaniu biblioteki Tcl/Tk pliki z rozszerzeniem **.tcl** są automatycznie kojarzone z interpreterem **wish** i można je uruchamiać jak zwykłe programy. Program GOOL może być uruchomiony poprzez standardowy interpreter języka Tcl/Tk - bez metod optymalizacji, dostępu do generatorów losowych i możliwości definiowania zadań - lub interpreter rozszerzony o metody. W drugim przypadku, po rekompilacji jądra języka, należy zamienić istniejący interpreter - **wish.exe** - na interpreter nowo utworzony. Nowo tworzony interpreter rozszerzony nosi nazwę **gool_gui.exe**.

Plik **wish.exe** znajduje się w katalogu, w którym zastał zainstalowany język Tcl/Tk, w podkatalogu *bin*. W przypadku, gdy istnieje potrzeba zachowania standardowego interpretera, program GOOL można uruchomić tworząc skrót do programu MS-DOS i wpisując w wierszu polecenia ścieżkę dostępu do utworzonego interpretera i plik główny programu, np.:

Wiersz polecenia: **gool_gui.exe gool.tcl**



Rysunek 4.1: Główne okno aplikacji.

Po uruchomieniu programu, na ekranie pojawia się główne okno aplikacji, które przedstawione jest na rysunku 4.1 [1]. W lewym panelu okna umieszczane są informacje dotyczące aktywnego zadania (sekcja -*Task*-). Są to: symboliczna nazwa **Name** oraz postać analityczna funkcji celu (o ile została ona zdefiniowana analitycznie) lub ścieżka do programu zwracającego wartość funkcji celu dla zadanego argumentu (przypadek symulatora) - **Objective**. W prawym panelu dostępne są wbudowane metody optymalizacji (sekcja -*Methods*-), tworzące strukturę drzewiastą, stanowiącą hierarchię zstępującą - oraz lista (sekcja -*Data*-) reprezentująca wyniki optymalizacji (uzyskane poprzez uruchomienie metod biblioteki lub wczytane z pliku), jak również dane stanowiące punkty wygenerowane przez zastosowane generatory losowe.

Przed wczytaniem lub utworzeniem pierwszego zadania (zaraz po uruchomieniu aplikacji), pola w lewym panelu są puste, a w menu niedostępne są niektóre opcje (np. uruchomienie metody optymalizacji).

Drzewo metod zawarte w prawym panelu generowane jest podczas uruchomienia programu, na podstawie znajdującego się w tym samym katalogu co program pliku tekstowego. W pliku zapisane są informacje dotyczące wszystkich zaimplementowanych metod. Jego format podany jest w rozdziale 6.4, opisującym rejestrację nowo utworzonej metody

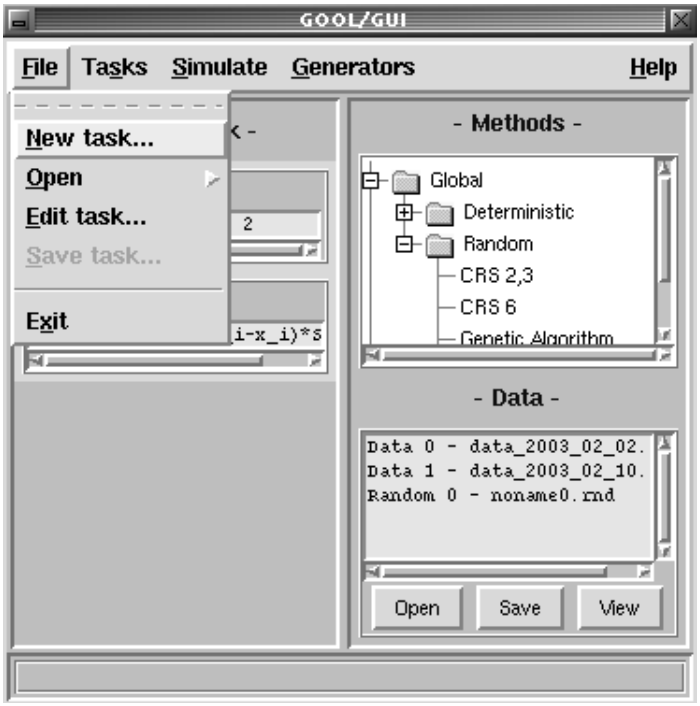
[1] Wszystkie zamieszczone w pracy ekrany pochodzą z wersji programu GOOL uruchomionej w systemie operacyjnym Linux. Wszędzie tam, gdzie uznano, iż nie wpłynie to na zmniejszenie czytelności rysunków, zastosowano konwersję do skali kolorów zdefiniowanych kolejnymi stopniami szarości.

w systemie GOOL.

Program automatycznie rozpoznaje rodzaj interpretera - standardowy lub rozszerzony o metody optymalizacji. W przypadku uruchomienia go za pomocą interpretera standardowego, w prawym panelu pojawia się tylko sekcja *-Data-*, ponieważ metody optymalizacji nie są w nim zaimplementowane.

4.2 Opcje menu głównego okna aplikacji

W menu **File** przedstawionym na rysunku 4.2 użytkownik ma do wyboru następujące opcje:

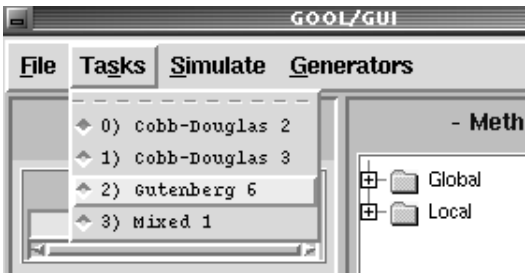


Rysunek 4.2: Menu File.

1. **New task...** - uaktywnia okno dialogowe do definiowania nowego zadania,
2. **Open** - uaktywnia okno dialogowe służące do odszukania na dysku zapisanego zadania lub zarchiwizowanych wyników wcześniej wykonanego eksperymentu,
3. **Edit task** - uaktywnia okno dialogowe służące do edycji aktywnego zadania,
4. **Save task** - uaktywnia okno dialogowe przeszukujące dysk w celu zapisania definicji i parametrów aktywnego zadania,
5. **Exit** - wyjście z programu; w przypadku gdy istnieją nowe lub zmodyfikowane zadania, dane uzyskane w czasie eksperymentów, jak również wygenerowane serie punktów losowych, program wyświetla okno dialogowe z zapytaniem o chęć ich zapisania i w przypadku potwierdzenia zapisuje je.

Menu **Tasks** służy do przełączania się pomiędzy wczytanymi lub utworzonymi przez użytkownika zadaniami, ustawiając jedno wybrane jako aktywne. Na rysunku 4.3 przedstawiona jest jego przykładowa zawartość. Zadania identyfikowane są przez ich nazwy symboliczne (dowolny ciąg znaków) oraz przez numery oznaczające kolejność pojawienia się

ich w programie.



Rysunek 4.3: Menu Tasks.

Rysunek 4.4: Menu Simulate.

W menu **Simulate** dostępne są następujące opcje (rysunek 4.4):

- **Graph 3D** - utworzenie nowego okna z wykresem 3D funkcji celu (z ograniczeniami) aktywnego zadania,
- Druga opcja menu jest wariantowa - w przypadku, gdy funkcja celu jest jednowymiarowa, figuruje tu pozycja **Side View**, wywołanie której powoduje wyświetlenie okna z wykresem funkcji jednej zmiennej. W przypadku funkcji o większym wymiarze, wykreślamy wykres poziomicowy - poprzednią opcję w menu zastąpi pozycja **Levels**, wywołanie której pozwala na utworzenie okna z wykresem poziomicowym, będącego podstawą do umieszczania punktów z przeprowadzanych eksperymentów.

UWAGA: przy pierwszym wykreślaniu poziomic, użytkownik ogląda przekrój w układzie odniesienia dwóch pierwszych zmiennych funkcji celu (x_1 i x_2), przy jednoczesnym ustaleniu pozostałych zmiennych (o ile istnieją) na wartość leżącą w połowie przedziału wartości dopuszczalnych dla danego wymiaru. Zmiany układu współrzędnych oraz wartości przypisanych niewyświetlanym zmiennym dokonuje się już poprzez menu okna, przedstawiającego konkretny wykres poziomicowy.

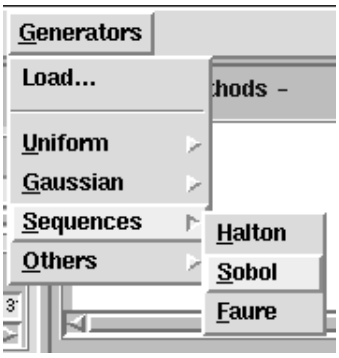
W przypadku minimalizowania funkcji jednej zmiennej do wizualizacji algorytmu (poszukiwania minimum kierunkowego) używane jest osobne, tworzone specjalnie do tego celu okno.

- **Start Simulation** - powoduje uruchomienie, zaznaczonej w prawym panelu okna głównego, metody optymalizacji (sekcja -*Methods*-) lub powtórzenie optymalizacji zapamiętanej (sekcja -*Data*-); w przypadku, gdy nie ma wyznaczonych aktywnych okien z wykresami poziomicowymi (**Active windows** w menu **Simulate**), wyniki działania metody są tylko zapamiętywane w sekcji -*Data*-,

- **Execute program** - uaktywnienia okno dialogowe w celu odszukania na dysku pliku wykonywalnego (skompilowanej metody optymalizacji), który po uruchomieniu przekazuje do programu kolejne punkty będące wynikiem procesu optymaliacji (współrzędne punktów oraz odpowiadające im wartości funkcji celu),

- **Active windows** - zawiera listę wszystkich otworzonych dotychczas okien z wykresami poziomicowymi i umożliwia ustawianie dowolnej ich liczby jako aktywnych okien symulacji (podobnie jak w menu **Tasks**, z tym, że tu można zaznaczyć dowolną ich liczbę). Okna identyfikowane są przez numer oznaczający kolejność pojawienia się oraz ciąg znaków zawierający numer i nazwę zadania, dla którego był rysowany wykres poziomicowy funkcji celu,

- **Options** - powoduje pojawienie się okien dialogowych do wprowadzania zmian opcji związanych z metodami optymalizacji, aktywnym zadaniem, przebiegiem eksperymentów, sposobem rysowania wykresów poziomicowych, itp. Rozwinięcie jego podmenu dotyczącego metod ma postać hierarchiczną, odpowiadającą strukturze drzewa prawego panelu głównego okna aplikacji.



Rysunek 4.5: Menu Generators.

Menu **Generators** z rysunku 4.5 zawiera hierarchiczne menu, pozwalające na wybór odpowiedniego generatora, w celu uzyskania populacji punktów o zadanym rozmiarze i własnościach, związanych z konkretnym, wybranym przez użytkownika, rozkładem (np. wartości oczekiwanej oraz wariancji w przypadku generatorów rozkładu normalnego - podanych oczywiście niezależnie dla każdego wymiaru). Generowane punkty można obserwować na płaszczyźnie, wyświetlić ich histogram oraz zapisywać do pliku.

Menu **Help** zawiera pomoc do programu oraz informacje o nim.

4.3 Wprowadzanie, modyfikowanie i zapisywanie zadania

Środowisko GOOL umożliwia wprowadzanie zadań przeznaczonych do analizy na dwa sposoby: poprzez wyspecyfikowanie wszystkich parametrów w specjalnie do tego przygotowanym oknie dialogowym, oraz wczytując zapisaną definicję zadania z pliku tekstowego.

Wczytanie odbywa się poprzez wybór opcji **Open task** menu **File**, zaś format pliku jest analogiczny do omawianego w przypadku wersji GOOL/CON (rozdział 3.2 pracy) pliku definicji zadania.

Okno dialogowe przeznaczone do wprowadzania nowego zadania oraz modyfikacji istniejącego zadania aktywnego przedstawia rysunek 4.6. Możemy w nim wyróżnić następujące sekcje:

- Name i Description - pierwsza z nich identyfikuje zadanie w systemie okienkowym, druga stanowi informację o charakterze uzupełniającym i istnieje tylko dla wygody użytkowników,

- Minimize - mamy tu do wyboru dwie metody określania funkcji celu zadania. Pierwsza polega na wprowadzeniu analitycznego wzoru funkcji (zgodnie ze składnią akceptowaną przez podsystem obsługi wyrażeń symbolicznych, opisany w dodatku A), druga (po uaktywnieniu tej możliwości poprzez zaznaczenie pola **simulator** pozwala na wybranie programu z dysku, pełniącego rolę „czarnej skrzynki" procesu. W drugim przypadku oczekuje się, iż program pobierając poszczególne współrzędne przekazanego mu punktu w postaci argumentów wywołania, zwraca wartość funkcji. Program ten może być zaimplementowany w dowolnym języku. **Gradient** służy zadaniu lub modyfikacji (przy wcześniejszym wyznaczeniu automatycznym) postaci gradientu funkcji celu (bez ograniczeń).

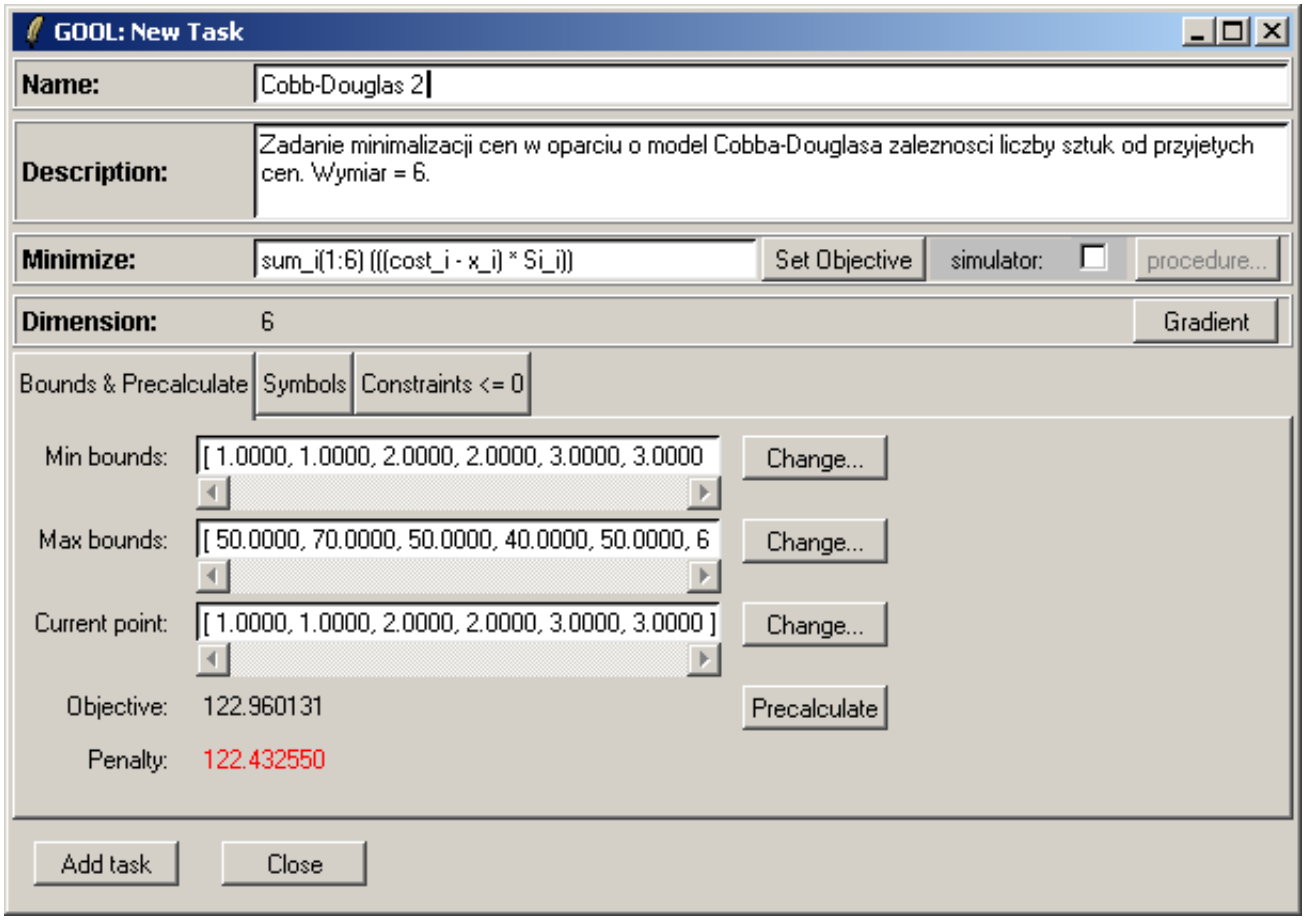
- Bounds & Precalculate - sekcja ta pozwala na modyfikację ograniczeń kostkowych zadania (wektory **min** i **max**), jak również na obliczenie wartości funkcji celu dla zadanego (w polu **point**) wektora. Po naciśnięciu przycisku **Apply** uzyskujemy wartość funkcji celu uwzględniającej karę za przekroczenie ograniczeń aktywnych zadania, jak również wartość samej kary. Wybrany w tym oknie punkt służy również jako podstawa do obliczeń wartości parametrów w pozostałych sekcjach,

- Symbols - sekcja pozwala na przeglądanie definicji i wartości symboli zdefiniowanych w kontekście zadania. Możliwa jest również ich edycja, usuwanie, a także dodawanie nowych symboli. W przypadku części symboli wartość może zależeć od punktu wyspecyfikowanego w sekcji **Bounds & Precalculate**.

- Constraints - ta część okna pozwala na przeglądanie, kasowanie oraz modyfikowanie ograniczeń nierównościowych zadania. Możliwe jest również przeglądanie ich wartości dla punktu wyspecyfikowanego w sekcji **Bounds & Precalculate**. Poprzez przełącznik **Active**, można dokonywać aktywacji oraz deaktywacji poszczególnych ograniczeń, obserwując ich wpływ na wartość funkcji celu zadania. Pole **Gradient** wyświetla postać na gradient dla wybranego ograniczenia z możliwością ręcznej edycji.

Przycisk **Delete task** służy do skasowania zadania z systemu. **Task options...** wywołuje kolejne okno dialogowe, pozwalające na zmianę wartości parametrów ogólnych rozpatrywanego zadania (takich jak np. wartość członu kary).

Użycie opcji **Edit task** jest jednym ze sposobów zmiany zakresów zmiennych dla aktywnej funkcji celu, inne sposoby podane będą w dalszej części opisu programu. Okno edycyjne można również wywołać przez dwukrotne przyciśnięcie lewego klawisza myszy $(< Double - Button1 >)$, gdy kursor znajduje się w lewym panelu głównego okna.



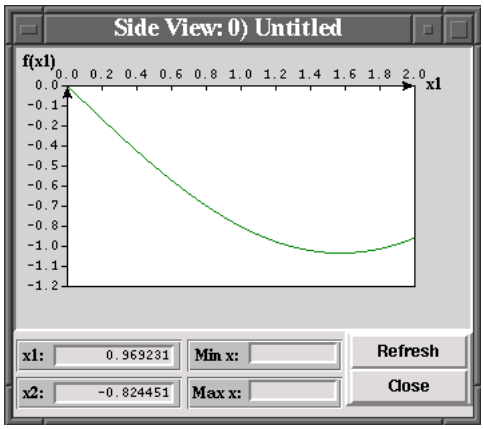
Rysunek 4.6: Okno zarządzania zadaniem.

UWAGA: Należy mieć świadomość, że dla zadań definiujących swoją funkcję celu poprzez podanie pliku zewnętrznego, system GOOL nie dysponuje informacją o jej gradiencie. W efekcie do minimalizacji tych funkcji nie można używać metod gradientowych. Również stosowanie niektórych rodzajów tokenów w analitycznej definicji funkcji celu uniemożliwia w obecnej wersji programu na wyznaczenie wzoru odpowiednich pochodnych.

4.4 Wykresy funkcji celu

4.4.1 Wykres funkcji jednej zmiennej

Wykres tego rodzaju jest wykorzystywany do wizualizacji przebiegu minimalizacji funkcji jednej zmiennej (w szczególności - minimalizacji w zadanym kierunku funkcji wielowymiarowej). Przykład tego typu okna przedstawia rys. (4.7). Za pomocą pól **Min x:** oraz **Max x:** możemy zmieniać zakres prezentowanej dziedziny (odświeżeniu służy przycisk **Refresh**). Dodatkowo pola **x1** oraz **x2** pokazują wartość funkcji celu dla aktualnego położenia kursora (niewidocznego na rysunku).



Rysunek 4.7: Wykres funkcji $f(x) = -\sin(x)$.

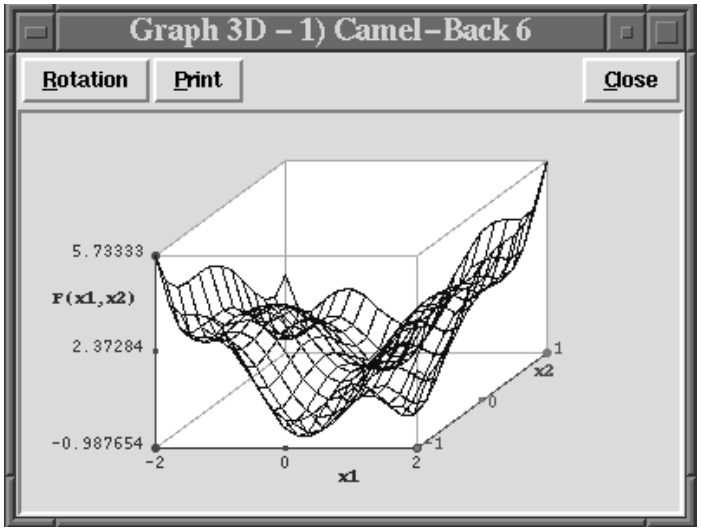
4.4.2 Trójwymiarowy wykres funkcji celu

W celu narysowania wykresu 3D wybranej funkcji celu, należy ustawić odpowiednie zadanie jako aktywne (menu **Tasks**), a następnie wybrać polecenie **Graph 3D** w menu **Simulate**. Wykres jest rysowany w oddzielnym, niezależnym od pozostałych, oknie. Rozmiar okna można modyfikować, powodując w ten sposób automatyczne przeskalowanie rysunku. Przed narysowaniem wykresu można ustawić gęstość siatki, wywołując okno dialogowe za pomocą polecenia **Options->Graph3D** w menu **Simulate**.

Na rysunku (4.8) przedstawiony jest przykładowy trójwymiarowy wykres funkcji „wielbłąda sześciogarbnego", która opisana jest następującym wzorem:

$$F(\mathbf{x}) = \frac{10x_1^5 - 42x_1^3 + 40x_1 + 5x_2}{5} \tag{4.1}$$

Wykresy mogą być obracane o dowolny kąt wokół każdej z trzech osi za pomocą okna dialogowego wywoływanego poleceniem **Rotation**. Wielokrotne naciśnięcie przycisku **Apply** w tym oknie powoduje cykliczne obracanie wykresu o tę samą, wpisaną wcześniej, wartość kątów.



Rysunek 4.8: Wykres trójwymiarowy funkcji Camel6.

Polecenie **Print** umożliwia drukowanie zawartości okna do pliku w formacie Postscript (format kolorowy lub czarno-biały).

Opcja tworzenia wykresu trójwymiarowego dostępna jest tylko dla funkcji dwuwymiarowych.

4.4.3 Poziomicowy wykres funkcji celu

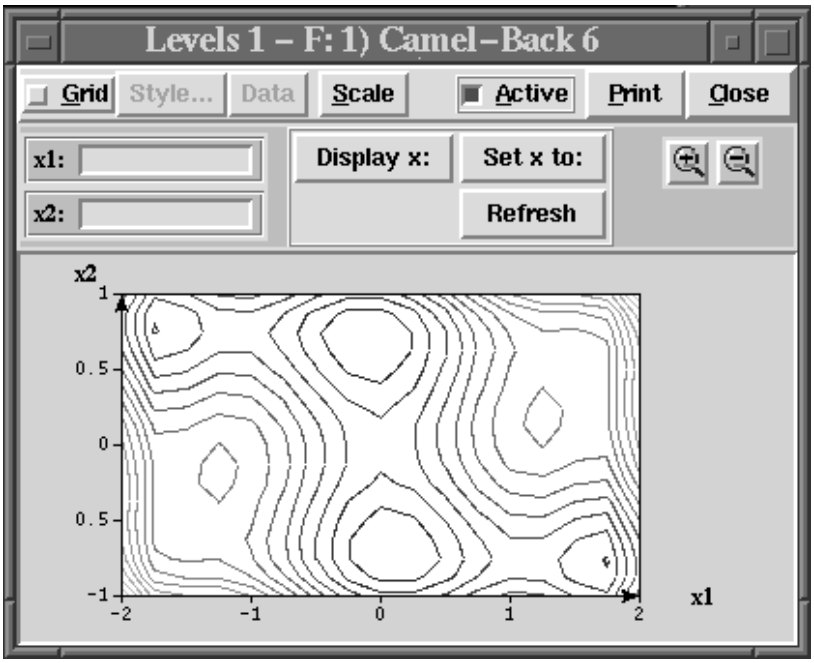
Wykresy poziomicowe funkcji celu są podstawą do prezentowania danych otrzymywanych podczas eksperymentów numerycznych. Wykresy rysowane są w niezależnych oknach. Umożliwia to między innymi utworzenie dla jednej funkcji celu wielu wykresów dla różnych zakresów zmiennych, co pozwala na dokładniejszą analizę wyników optymalizacji. Możliwe jest również oglądanie tego samego obszaru w różnych, wybranych przez użytkownika układach współrzędnych (zawsze dwuwymiarowych), przy ustalonej wartości zmiennych pozostałych wymiarów.

Do rysowania wykresów poziomicowych aktywnej funkcji celu służy polecenie **Levels** w menu **Simulate** (patrz rysunek (4.4)). Po wybraniu tego polecenia na ekranie pojawia się okno informujące o stanie zaawansowania obliczeń oraz okno z wykresem. Długość trwania obliczeń uzależniona jest od opcji dotyczących dokładności rysowania, ilości poziomic na wykresie (polecenie **Options->Levels** w menu **Simulate**) oraz złożoności funkcji celu.

Rysunek 4.9: Opcje rysowania poziomic.

Okno dialogowe służące do zmiany opcji przedstawione jest na rys. 4.9. Ostatnia opcja: **View point (Top / Bottom)** służy do zmiany kolorystyki rysowania wykresów:

- Top - poziomice położone najniżej rysowane są w kolorze ciemnozielonym, następnie przechodzą stopniowo do koloru żółtego (poziomice położone najwyżej).
- Bottom - poziomice położone najniżej rysowane są w kolorze żółtym, najwyżej: w kolorze ciemnozielonym.

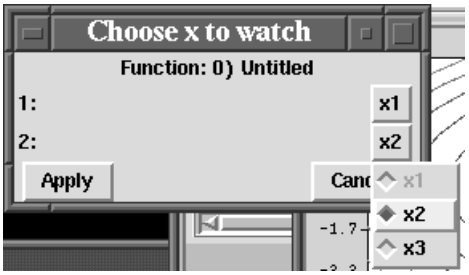


Rysunek 4.10: Wykres poziomicowy funkcji Camel6.

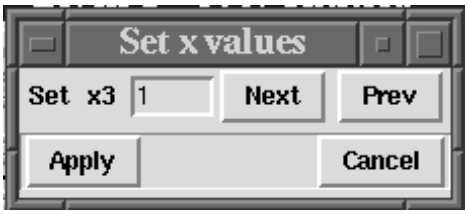
Na rysunku 4.10 przedstawione jest okno z wykresem poziomicowym funkcji Camel, definiowanej wzorem (4.1). Tytuł okna składa się z numeru będącego jego identyfikatorem oraz numeru i nazwy symbolicznej funkcji celu. Tytuł ten jest dodawany do listy utworzonych okien, która znajduje się w menu **Simulate->Active windows** głównego okna aplikacji.

W menu utworzonego okna użytkownik ma do wyboru następujące opcje:

- **Grid** - wyświetlenie lub ukrycie siatki,
- **Style** - wybór sposobu prezentacji przeprowadzonych eksperymentów - do wyboru mamy tutaj **Lines** (kolejne punkty będą łączone linią) oraz **Points**; wykorzystywane tylko dla wyników z metod jednopunktowych,
- **Data** - menu hierarchiczne, tworzone dynamicznie w trakcie pojawiania się kolejnych serii z punktami danych na wykresie, służy do zarządzania wyświetlaniem poszczególnych serii oraz umożliwia ich usuwanie; wykorzystywane tylko dla wyników z metod jednopunktowych,
- **Scale** - opcje związane z przeskalowywaniem rysunku; pozwalają odświeżyć przeskalowane okno (**Scale -> Refresh**) oraz utworzyć nowe okno na bazie zadanego obszaru przeskalowanego **Scale -> New Window**,
- **Active** - ustawia dane okno jako aktywne, czyli gotowe do umieszczania w nim punktów bądących rezultatem wykonywanych eksperymentów,
- **Print** - zapisuje wydruk okna do pliku w formacie Postscript,



Rysunek 4.11: Okno wyboru układu współrzędnych.



Rysunek 4.12: Nadawanie wartości zmiennym ustalonym funkcji.

- **Close** - zamknięcie okna.

W pasku znajdującym się poniżej menu wyświetlane są wartości aktualnie oglądanych zmiennych, które są związane z aktualnym położeniem kursora myszki. Z prawej strony wykresu umieszczane są identyfikatory oraz kolory i kształty punktów dla danych z eksperymentów, które były przeprowadzane w tym oknie. Przyciski zawarte wewnątrz paska mają następujące przeznaczenie:

- **Display x:** - pozwala na wybór wyświetlanych zmiennych,
- **Set x to:** - pozwala na nadanie wartości pozostałym zmiennym (domyślnie nadawana jest średnia przedziału dopuszczalnego),
- **Refresh** - powoduje przerysowanie wykresu po zmianie układu współrzednych (**Display x:**) oraz nadaniu wartości zmiennym spoza układu (**Set x to**); funkcja analogiczna do pozycji z menu **Scale->Refresh**.

Dodatkowo widoczne w pasku lupy pozwalają w prosty sposób powiększać dowolny podobszar aktualnie obserwowanego obszaru - przeciągając myszką po wykresie tworzymy prostokątny wycinek, który następnie po wciśnięciu symbolu lupy (plus) zdefiniuje nowy, przeskalowany obszar. Symbol lupy z minusem służy do powrotu do obszaru sprzed powiększenia.

Rysunki (4.12) oraz (4.11) pokazują przykładową zawartość okien związanych z powyższymi opcjami dla funkcji trzech zmiennych.

Wartość funkcji w punkcie wskazywanym przez kursor można uzyskać wciskając lewy klawisz myszy (<Button1>) - chwilowe wskazanie, lub środkowy (<Button2>) - oznaczenie na stałe. Należy przy tym pamiętać o tym, iż wartość ta zależy od aktualnego położenia kursora myszki *oraz* od wartości stałych przypisanych zmiennym niezwiązanym z oglądanym układem współrzędnych. Dwukrotne wciśnięcie środkowego przycisku (<Double-Button2>) powoduje wyczyszczenie wcześniej utworzonych oznaczeń wartości funkcji celu. „Przeciągając" kursor w oknie przy wciśniętym prawym przycisku myszy (<Button3>) można oznaczyć część rysunku i następnie powiększyć go w tym samym oknie (menu **Scale->Refresh** lub wykorzystanie symbolu lupy) lub w oknie nowo utworzonym (menu **Scale->New Window**).

4.5 Praca z metodami optymalizacji z wbudowanej biblioteki

Wykonywanie eksperymentów numerycznych z metodami optymalizacji wiąże się z częstą zmianą funkcji celu, zakresów zmiennych oraz ustawianiem parametrów testowanej metody. W przypadku korzystania z metod znajdujących się w wbudowanej do programu bibliotece, wszystkie te operacje można wykonywać z poziomu aplikacji za pomocą czytelnych okien dialogowych. Ponadto, podczas wykonywania przez metodę obliczeń można na bieżąco śledzić postępy dzięku umieszczanym automatycznie kolejnym, wyznaczanym przez metodę punktom na wcześniej przygotowanych wykresach poziomicowych funkcji celu.

Reasumując, pracę z metodami bibliotecznymi można podzielić na następujące etapy:

1. wczytanie lub utworzenie odpowiednich funkcji celu,
2. przygotowanie środowiska graficznego,
3. ustalanie parametrów metod optymalizacji,
4. wykonywanie eksperymentów.

Po wprowadzeniu przynajmniej jednego zadania, w oknie głównym aplikacji udostępnione są polecenia z menu **Simulate**. Oznacza to, że już w tej chwili można uruchamiać metodę z biblioteki lub wprowadzać dane z innego źródła. Wszystkie otrzymywane dane są zapamiętywane przez program w sekcji -*Data*- w oknie głównym, jednak sugerowane jest umieszczanie tych danych na bieżąco na wcześniej przygotowanych wykresach.

4.5.1 Przygotowanie środowiska graficznego

Program umożliwia utworzenie dowolnej liczby okien z wykresami trójwymiarowymi oraz poziomicowymi wielowymiarowych funkcji celu. Po wczytaniu odpowiedniej liczby funkcji celu, użytkownik może obejrzeć ich wygląd rysując wykresy 3D oraz przygotować po kilka wykresów poziomicowych dla każdej funkcji (np. dla różnych zakresów zmiennych lub przekrojów). Jeżeli okna zajmują zbyt dużo miejsca na ekranie, można je przenieść do ikony i w danym momencie korzystać tylko z takich, które przedstawiają aktualnie optymalizowaną funkcję celu.

4.5.2 Ustalenie wartości parametrów symulacji

Opcja ta umożliwia ustalenie wartości parametrów eksperymentu, niezależnych od wybranej metody minimalizacji. Do opcji tego rodzaju należą:

- **Delay (ms)** - opóźnienie w milisekundach każdej iteracji metody; wykorzystywane wówczas, gdy istnieje potrzeba wolniejszego generowania punktów w celu czytelniejszej prezentacji działania metody.
- **Uniform generator** - numer generatora rozkładu jednostajnego, który będzie używany w trakcie przeprowadzania eksperymentu. Właściwie każda metoda (nawet metody deterministyczne - np. w momencie wyboru - o ile odbywa się to drogą losowania - punktu startowego) w jakiś sposób korzysta z generatora liczb pseudolosowych. Ta pozycja menu pozwala wybrać, który generator z zestawu zaimplementowanych w bibliotece GOOL/RG będzie użyty. W tym przypadku termin „generator rozkładu jednostajnego” obejmuje również sekwencje pseudolosowe.

Rysunek 4.13: Ogólne parametry symulacji.

- **User provides start point** - flaga decydująca o tym, czy punkt startowy (o ile metoda z takiego korzysta), będzie podany przez użytkownika, czy też zostanie wyznaczony losowo. O tym, czy konieczne jest jego wyznaczenie decyduje charakter metody [2]; można również narzucić to w sposób jawny, dokonując odpowiedniego wpisu w pliku konfiguracyjnym **methods.tcl**, który opisuje wszystkie metody optymalizacji.
- **Educational mode** - flaga wskazująca na to, czy chcemy uruchamiać proces optymalizacji w specjalnym trybie, pozwalającym na dokładniejsze śledzenie poszczególnych faz działania metody.
- **Direction minimum search** - specyfikacja zastosowanego algorytmu znajdowania mimimum w kierunku (o ile metoda optymalizacji będzie taki wykorzystywać), wraz z podaniem wartości dla niego charakterystycznych. Opis zaimplementowanych metod wraz z podaniem parametrów indywidualnych znajduje się w rozdziale 5.6.

Rysunek 4.13 pokazuje przykładowe ustawienie parametrów eksperymentu - pozycje, umożliwiające ustalanie parametrów metody minimalizacji w kierunku zmieniają się (pod względem opisu oraz liczby w sensie pozycji edytowalnych) w zależności od wybranej z listy **Direction minimum search** metody; aktywna metoda *Golden division* jest parametryzowana dwiema wielkościami (przy czym *maxTau* jest parametrem opcjonalnym), podczas gdy np. aproksymacja paraboloidą - trzema.

Uwaga 1: Należy pamiętać o tym, iż korzystanie z metod gradientowych (czyli - np. testu dwuskośnego Goldsteina) znajdowania minimum w kierunku nie zawsze jest możliwe - gdy minimalizowana funkcja nie jest zadana w sposób analityczny. W takim przypadku

[2] Wymagają tego wszystkie metody, w których w każdym kroku wyznaczany jest jeden nowy punkt, zwane dalej *jednopunktowymi*.

środowisko przed uruchomieniem metody dokona przełączenia metody na domyślną metodę bezgradientową, jaką z definicji jest algorytm złotego podziału, informując użytkownika o tym fakcie.

Uwaga 2: W przypadku badania funkcji jednej zmiennej - ze względu na fakt, iż w ogólnym przypadku test Goldsteina jest metodą niedokładną (akceptuje wartości z szerokiego przedziału, spełniającego warunek testów) - środowisko dokona analogicznego przełączenia, zapewniając zwiększoną dokładność znalezionego rozwiązania.

4.5.3 Ustalanie parametrów metod optymalizacji

Podczas testowania metod optymalizacji bardzo ważny jest dobór parametrów związanych z daną metodą. W przypadku metod z biblioteki, zmianę parametrów metod można wykonywać z poziomu aplikacji za pomocą okien dialogowych.

Zarówno liczbę jak i rodzaj parametów ustala się podczas kompilacji procedury realizującej metodę. Informacje o metodzie i jej parametrach program pobiera ze znajdującego się w tym samym katalogu pliku tekstowego **methods.tcl**. Poniżej przedstawiony jest fragment tego pliku opisujący metodę CRS2(3):

```
          ...

crs2 "CRS 2,3" Global Random Population
"Local search(CRS3)"                  int      0     0 1
"NP"                                  int      10    1 -
          ...
```

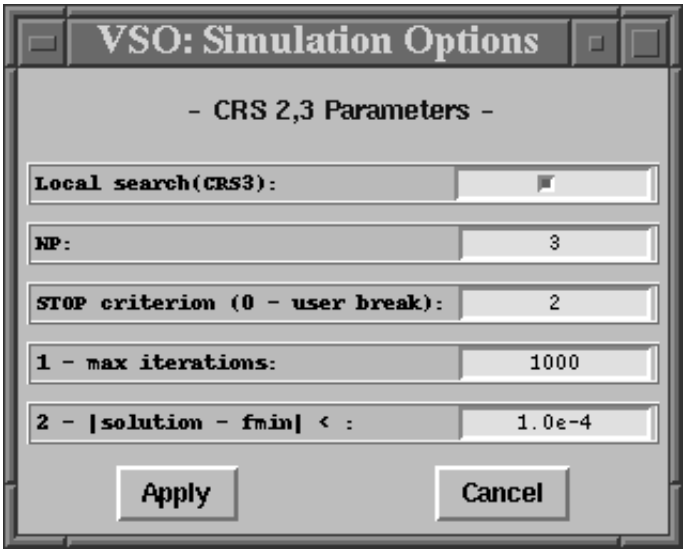
Format tego pliku wraz ze szczegółowym opisem znaczenia poszczególnych parametrów można znaleźć w rozdziale 6.4 poświęconemu rozbudowie biblioteki metod.

Po uruchomieniu programu GOOL/GUI, podane w pierwszej linii nazwy symboliczne metod widoczne są w drzewie -*Methods*- w prawym panelu okna głównego. Parametry można zmieniać wywołując okno dialogowe dostępne poprzez menu hierarchiczne **Options->Method settings** lub, po zaznaczeniu danej metody w drzewie metod, poprzez dwukrotne wciśnięcie prawego przycisku myszy. Okno dialogowe służące do zmiany parametrów metody, której opis przedstawiono powyżej, prezentuje rys 4.14.

Należy zauważyć, iż choć parametry decydujące o spełnieniu kryterium STOP-u algorytmu nie są ściśle związane z konkretną metodą (dokładniej - część parametrów opisujących kryterium STOP-u jest ogólna, np. maksymalna liczba iteracji), to ze względu na możliwość specyfikowania przez metodę dodatkowo własnych kryteriów, parametry tego typu zostały zaklasyfikowane jako przypisane konkretnemu algorytmowi.

4.5.4 Wykonywanie eksperymentów

Metody optymalizacji z biblioteki można uruchomić wówczas, gdy do programu wczytane jest przynajmniej jedno zadanie. Jeśli jest ich więcej, to optymalizacja wykonywana jest dla zadania aktualnie aktywnego, widocznego w lewym panelu okna głównego aplikacji. Metody uruchamia się poprzez zaznaczenie w sekcji -*Methods*- testowanego algorytmu i wywołanie polecenia **Start simulation** dostępnego w menu **Simulate** lub poprzez dwukrotne wciśnięcie lewego przycisku myszy (<Double-Button1>) po skierowaniu kursora na identyfikator metody. W przypadku korzystania z metody wymagającej podania punktu



Rysunek 4.14: Parametry metody *CRS2-3*.

startowego oraz uprzedniego uaktywnienia opcji zadawania jego współrzędnych przez użytkownika, otwiera się okno dialogowe pozwalające na ich wprowadzenie.

Użytkownik może otrzymać dwa rodzaje raportów z przebiegu obliczeń: bardziej i mniej szczegółowy, w zależności od tego, czy wykorzystujemy tryb edukacyjny, czy też nie. W pierwszym przypadku otrzymujemy informację pełniejszą, na którą składają się:

- nazwa aktywnej metody,
- wskaźnik trybu pracy metody,
- numer aktualnej iteracji,
- liczba wywołań funkcji celu,
- najlepsze znalezione dotychczas rozwiązanie wraz z wartością funkcji celu oraz numerem iteracji, w której punkt ten został wyznaczony,
- wartości parametrów, które uznane zostały za istotne w kontekście zastosowanej metody (widoczne tylko w trybie edukacyjnym),
- informacja tekstowa, mówiąca o stanie algorytmu (status).

Rysunek 4.15 przedstawia przykładowe okno z raportu działania metody siatki Gourdina; parametrami, których wartość jest podawana przez GOOL/OM są tutaj: stała Lipschitza funkcji celu (parametr istotny w przypadku, gdy algorytm sam zmuszony był do oszacowania jej wielkości) oraz oszacowanie górne (f_{opt}) i dolne (F_{opt}) aktualnie rozpatrywanej komórki siatki.

Dane (tj. współrzędne punktów i wartości funkcji celu), otrzymywane podczas optymalizacji zapamiętywane są w sekcji *-Data-* w prawym panelu okna głównego. Przydzielany jest im identyfikator data_nr oraz pierwotna nazwa pliku, do którego można je zapisać. Dodatkowo program zapamiętuje informacje dotyczące eksperymentu, czyli dane o optymalizowanej funkcji celu oraz metodzie optymalizacji. Dane te zapisywane są do pliku w formie komentarza (po znakach „#”).

Wyniki działania algorytmów optymalizacji są prezentowane w trybie tekstowym (tabela wartości) i graficznym. Prezentacja graficzna ściśle wiąże się z typem stosowanego

Rysunek 4.15: Okienko raportu symulacji.

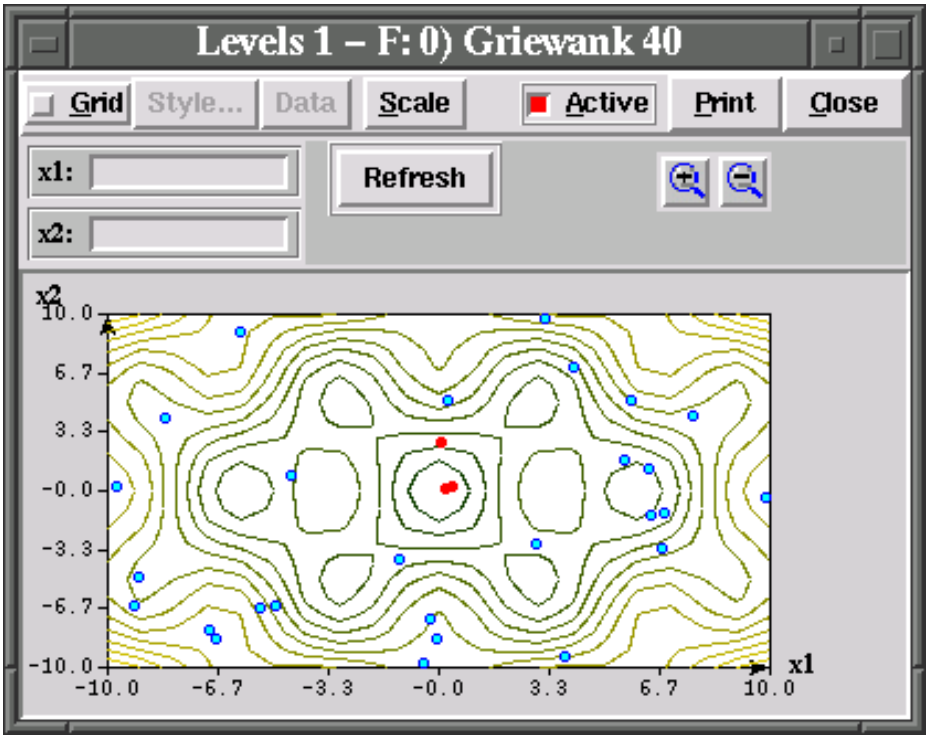
algorytmu minimalizacji. W przypadku metod jednopunktowych (tzn. takich, których „zbiór roboczy” w każdej iteracji składa się tylko z jednego, bieżącego punktu) - czyli np. metoda Griewanka, symulowanego wyżarzania - obserwować możemy kolejne położenia aktualnego punktu, z jednoczesnym wskazaniem na położenie punktu dotychczas najlepszego. Metody wielopunktowe (np. CRS, algorytm genetyczny) pozwalają nam obserwować dynamikę zmian całego ZBIORU punktów roboczych, zaś w przypadku uaktywnienia trybu edukacyjnego - jednocześnie wyróżnianie punktów charakterystycznych (np. grupy $n+1$ najlepszych w metodzie CRS3, dla których uruchamiana jest lokalna metoda sympleksu nieliniowego). Wizualizacja metod siatki (np. Gourdina, Meewella-Mayne), umożliwia obserwację procesu tworzenia oraz usuwania kolejnych komórek tworzonych podczas przeszukiwania dziedziny funkcji celu, z zaznaczeniem komórki aktualnie rozpatrywanej. Rysunki 4.16 oraz 4.17 pokazują przykładowy przebieg algorytmów: CRS23 (z zaznaczonym zbiorem punktów podlegających optymalizacji lokalnej) oraz Gourdina (czerwonym kolorem zaznaczona jest aktualnie rozpatrywana komórka).

Wizualizacja procesu optymalizacji odbywa się w oknach z wykreślonymi poziomicami funkcji celu (z rzutowaniem do aktywnych wymiarów w przypadku funkcji wielowymiarowych). Przed uruchomieniem metody należy oznaczyć interesujące okna jako aktywne (przycisk **Active** w oknie z wykresem poziomicowym lub zestaw wszystkich okien w menu **Simulate->Active windows** w oknie głównym aplikacji). Użytkownik sam musi zadbać o to, by okna ustawione jako aktywne dotyczyły testowanej funkcji celu.

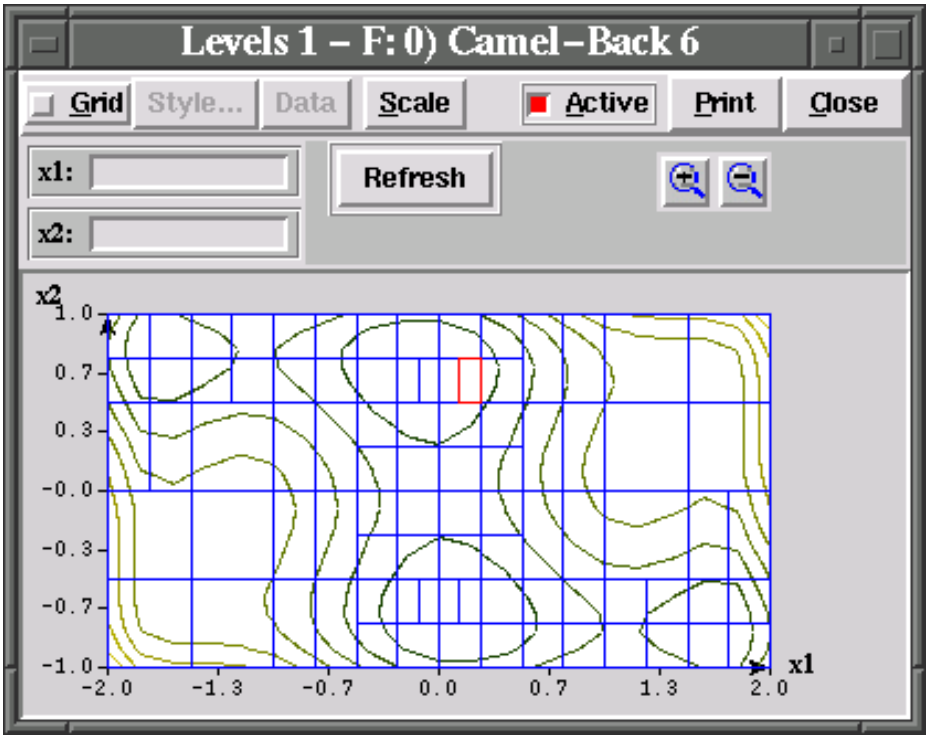
Zarówno funkcja celu jak i okna przedstawiające jej wykresy poziomicowe posiadają ograniczenia na zmienne (okna mogą przedstawiać fragmenty dziedziny danej funkcji). Podczas wykonywania eksperymentów ograniczenie zmiennych dla metody rozumiane jest jako minimalny zakres obejmujący zakresy wszystkich aktywnych okien oraz zakres widoczny w sekcji -*Tasks*- głównego okna aplikacji.

Działanie metody można przerwać przed spełnieniem kryterium stopu poprzez wciśnięcie przycisku **Stop** w oknie przedstawiającym raport z eksperymentu (rys. 4.15).

Po zakończeniu działania metody w trybie edukacyjnym, obejrzeć można wykresy stanowiące podsumowanie procesu optymalizacji. Obecnie prezentacja tego typu jest wyko-



Rysunek 4.16: Wizualizacja działania metody CRS23.



Rysunek 4.17: Wizualizacja działania metody siatki Gourdina.

rzystywana w przypadku dwóch algorytmów - symulowanego wyżarzania (rozdział 5.5.5) oraz algorytmu ewolucyjnego (rozdział 5.5.6).

4.6 Wizualizacja poszukiwania minimum w kierunku

Jeśli wybrany algorytm optymalizacji należy do grupy metod poprawy, w trybie edukacyjnym mamy możliwość dokładnego śledzenia jego działania. Sposób prezentacji jest ścisle związany z charakterem wybranej metody poszukiwania optymalnej długości kroku. Rysunki 4.18, 4.19, 4.20 pokazują przykładowe stany metod minimalizacji w kierunku (w przypadku pierwszej z nich ze względu na czytelność zdecydowano się zamieścić wersję kolorową). Obok obszaru roboczego wykresu, na którym obserwujemy przebieg metody na tle wykresu funkcji celu (konkretnie przekroju bocznego w zadanym kierunku), umieszczone są wielkości charakterystyczne oraz dodatkowe informacje procesu poszukiwania. Są nimi: kierunek poszukiwań, punkt startowy, iteracja metody. Funkcje przycisków znajdujących się poniżej obszaru roboczego są następujące:

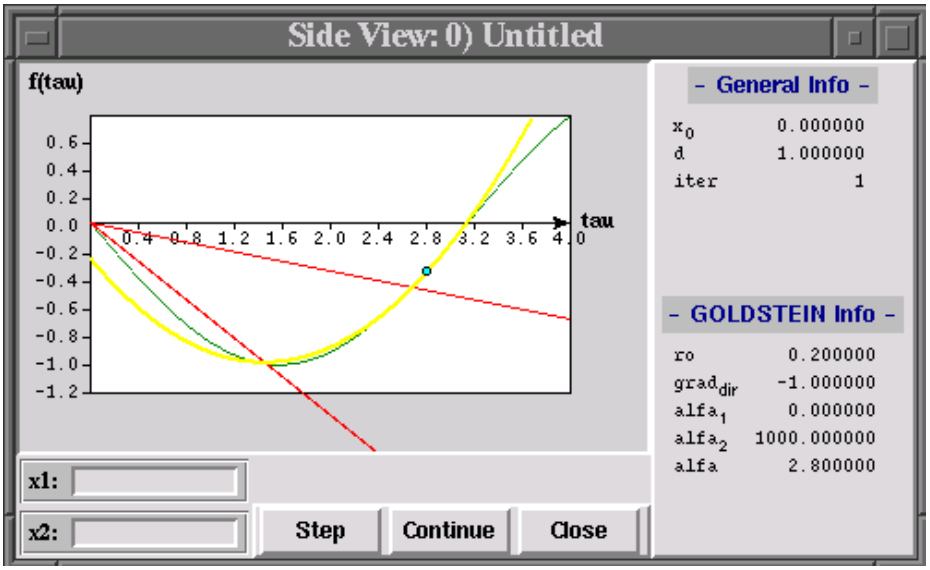
- **Step** - tryb pracy krokowej, pozwalający dokładnie śledzić przebieg działania metody,
- **Continue** - wyłączenie pracy krokowej - algorytm poszukiwania minimum kierunkowego nadal prezentuje wyniki, ale nie czeka już na interwencję użytkownika; ponowne wywołanie metody kierunkowej przez metodę nadrzędną spowoduje powtórne przełączenie w tryb pracy krokowej,
- **Close** - zamyka okno wizualizacyjne; metoda kontynuuje działanie aż do momentu znalezienia optymalnego współczynnika kroku $\hat{\tau}$; ponowne wywołanie metody nie będzie już prezentowane na ekranie.

4.7 Archiwizacja i odtwarzanie danych

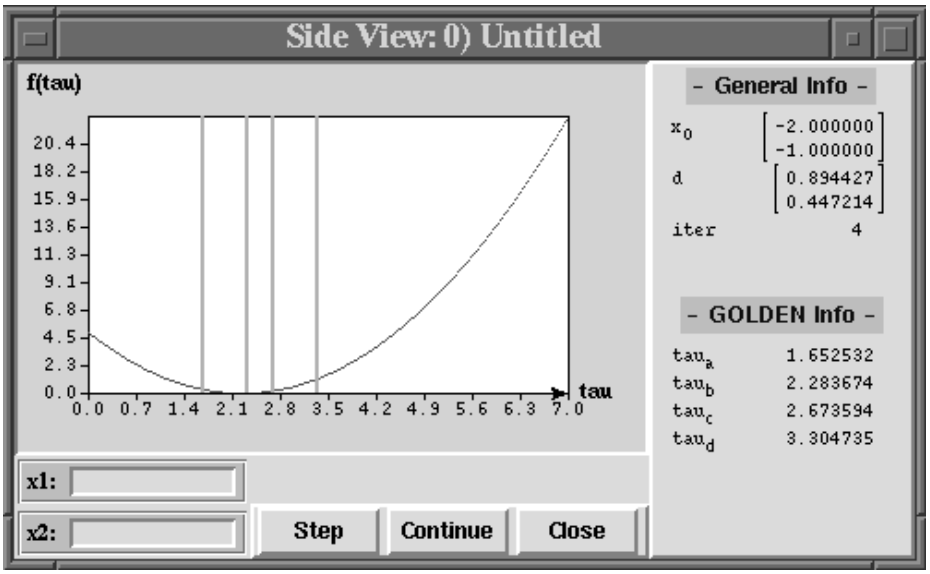
Dane otrzymywane podczas wykonywania eksperymentów są zapamiętywane przez program w sekcji *-Data-*, w prawym panelu okna głównego. Można je zapisać do pliku, obejrzeć w postaci tekstowej lub wykorzystać do odtworzenia przebiegu symulacji, w której wyniku zostały wygenerowane. Okno przedstawiające tekstową postać danych z eksperymentu uzyskuje się poprzez zaznaczenie w sekcji *-Data-* identyfikatora danych (data_nr nazwa_pliku) oraz wciśnięcie przycisku **View**; innym sposobem jest dwukrotne wciśnięcie prawego przycisku myszy (<Double-Button3>).

Informacje zapamiętywane w oknie pozwalają na całkowite odtworzenie przebiegu symulacji, również pod względem pełnej wizualizacji zgodnej z pracą w trybie ***Educational Mode***. Aby było to możliwe, rejestracji podlegają zarówno wartości parametrów metody, współrzędne kolejnych punktów generowanych przez algorytm jak i komendy specjalne, służące tylko do ilustracji wykonywanych kroków.

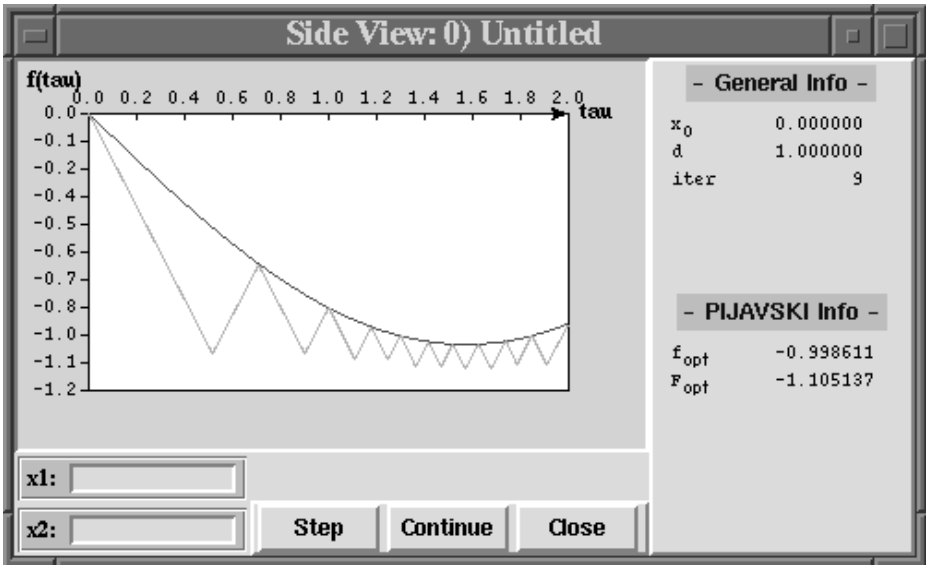
W przypadku, gdy wyniki testów nie były na bieżąco przedstawiane na wykresie poziomicowym, prezentacja może być wykonywana po zakończeniu eksperymentu. Umieszczenie punktów na jednym lub kilku wykresach poziomicowych jest analogiczne do uruchomienia metody optymalizacji. Należy zaznaczyć w sekcji *-Data-* odpowiedni identyfikator danych i wywołać polecenie **Start simulation** dostępne w menu **Simulate**, lub po skierowaniu kursora na identyfikator danych, podwójnie wcisnąć lewy przycisk myszy (<Double-Button1>).



Rysunek 4.18: Wizualizacja działania metody gradientowej. Kolorem zielonym zaznaczono wykres oryginalnej funkcji celu, kolorem czerwonym proste definiujące obszar dopuszczalny zgodnie z testem dwuskośnym Goldsteina. Niebieskie kółko obrazuje (nieakceptowalną) wartość parametru *alfa* - brak akceptacji (niespełnienie testu jednoskośnego) powoduje wyznaczenie nowego punktu *alfa* jako minimum interpolacji kwadratowej przy wykorzystaniu informacji o pochodnych funkcji. Przebieg funkcji interpolującej zaznaczono na wykresie kolorem żółtym.



Rysunek 4.19: Przykład wizualizacji działania bezgradientowej metody złotego podziału odcinka. Kreski pionowe zaznaczone jaśniejszym kolorem odpowiadają punktom charakterystycznym metody τ_a, τ_b, τ_c, τ_d, których dokładne wartości podane są w prawym panelu okna.



Rysunek 4.20: Przykład działania metody Pijavskiego-Shuberta dla funkcji $f(x) = -\sin(x)$. Kolorem jaśniejszym zaznaczony został aktualny obraz funkcji podpierającej dla funkcji oryginalnej; zauważyć można, iż już na etapie tej iteracji oszacowania górne oraz dolne dość dobrze przybliżają wartość optimum w przedziale.

Dane można zapisać do pliku wciskając przycisk **Save** w sekcji *-Data-*, natomiast wczytać do programu za pomocą polecenia **Open-data file...** file w menu **File** lub wciskając podwójnie środkowy przycisk myszy (<Double-Button2>). Możliwa jest prezentacja dowolnych danych z pliku pod warunkiem, że spełnia on następujące wymagania:

1. w pliku umieszcza się wartości liczbowe reprezentujące współrzędne kolejnych punktów oraz odpowiadające im wartości funkcji celu,
2. współrzędne kolejnych punktów wraz z wartościami funkcji celu muszą być umieszczone w osobnych liniach, tworząc w ten sposób kolumny, które kolejno oznaczają: $x_1, x_2, \ldots, x_n,\ f(x_1, x_2, ..., x_n)$;,
3. minimalna liczba kolumn wynosi 2, czyli wymiar zadania wynosi wówczas 1 (dla danych bez wartości funkcji celu lub n=1, ważna jest tylko pierwsza lub dwie pierwsze kolumny, w pozostałych kolumnach mogą być dowolne wartości),
4. w dowolnej linii można umieścić komentarz poprzedzając go znakiem „#”.

Wizualizacja jest w tym przypadku realizowana w trybie standardowym, zaś symulowana metoda przybiera jednopunktowy charakter.

4.8 Importowanie danych z innych programów

Program GOOL/GUI umożliwia pobieranie danych (wartości punktów i odpowiadające im wartości funkcji celu) z zewnętrznych programów wykonywalnych. Na danych otrzymywanych w ten sposób można wykonywać takie same operacje jakie były opisane w poprzednich rozdziałach, czyli:

1. umieszczać je na jednym lub wielu wykresach poziomicowych w trakcie wykonywania zewnętrznego programu lub po jego zakończeniu,

2. przeglądać je w postaci tekstowej,

3. archiwizować je i ponownie wczytywać do programu GOOL/GUI.

Program zewnętrzny musi być programem wykonywalnym. Komunikacja z programem GOOL/GUI odbywa się za pośrednictwem przekierowania strumieni we/wy. Program zewnętrzny powinien na standardowe wyjście (stdout) wysyłać dane w następującym formacie:

- dane dotyczące jednego punktu (współrzędne: $x_1, x_2, ..., x_n$ oraz wartość funkcji celu: $f(x_1, x_2, \ldots, x_n)$) muszą być oddzielone od siebie przynajmniej jedną spacją i zakończone znakiem końca linii,
- minimalna ilość danych liczbowych w jednej linii wynosi 2, czyli n=1,
- jeśli nie powiedzie się konwersja danych z całej linii na liczby rzeczywiste, lub gdy liczba danych w linii jest za mała, to cała linia zapamiętywana jest jako komentarz.

Format ten jak widać jest analogiczny do formatu podanego w poprzednim rozdziale. W programie realizującym metodę optymalizacji zaimplementowanym w języku C, kolejne punkty optymalizacji można wysyłać wykorzystując funkcję biblioteczną *printf()*, np.

```
...
printf("Komentarz 1\n");
printf("Komentarz 2\n");
...
    for(;;) {
        ...
        printf("%f %f %f %f \n", x[0], x[1], x[2], fun(x));
        ....
        if(..)
            printf("Komentarz warunkowy\n");
        ....
    }
...
printf("Komentarz - raport symulacji\n");
...
```

gdzie fun(x) jest procedurą obliczającą wartość funkcji celu w punkcie x.
Komentarze (ciągi znaków zakończone znakiem końca linii) są zapamiętywane przez program GOOL/GUI i można je obejrzeć lub zapisać do pliku.

Program zewnętrzny uruchamia się poleceniem **Execute program** dostępnym w menu **Simulate** głównego okna aplikacji GOOL/GUI. Po wybraniu tego polecenia wyświetlane jest okno dialogowe do odszukania właściwego pliku, na który należy wskazać. W następnym oknie dialogowym, należy podać ewentualne parametry, które mają być przekazane do uruchamianego programu (rys. 4.21). Po potwierdzeniu program jest wykonywany i na ekranie pojawia się okno umożliwiające jego wcześniejsze zatrzymanie (np. gdy wystąpią błędy).

4.9 Generowanie punktów o zadanym rozkładzie

W bibliotece GOOL/RG zaimplementowano trzy typy rozkładów oraz trzy sekwencje pseudolosowe.

Rysunek 4.21: Okno wprowadzania parametrów wywołania programu zewnętrznego.

W tryb tworzenia punktow o zadanym rozkładzie wchodzimy poprzez menu **Generators**, wybierając następnie wariant nas interesujący w obrębie danej grupy. Następnie korzystając z okienka **„Parameters”** - rysunek 4.22 - specyfikujemy liczbę punktów do wygenerowania, wymiar oraz parametry wybranego wariantu generatora.

UWAGA: niektóre generatory narzucają ograniczenia co do wymiaru, co wynika bądź z samej ich natury lub nietypowych metod inicjalizacji. W takich przypadkach program GOOL, jeśli zostanie podany za duży wymiar, ustala jego wartość na maksymalną dopuszczalną w danym kontekście.

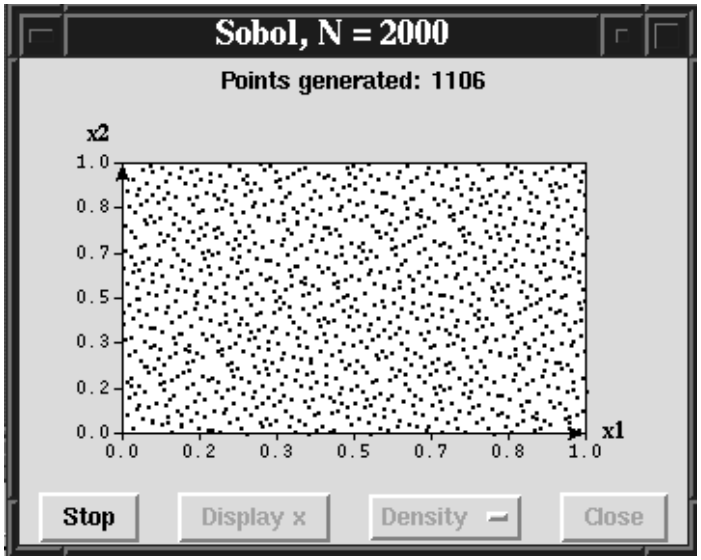
Naciśnięcie klawisza **Ok** okna ustalenia parametrów powoduje rozpoczęcie procedury generowania:

Rysunek 4.22: Wprowadzanie parametrów generatora rozkładu normalnego.

- punktów na wykresie w układzie współrzędnych pierwszych dwóch zmiennych (w przypadku generatorów wielowymiarowych),
- wykresu histogramu gęstości danego rozkładu w przypadku rozkładu jednowymiarowego (np. rozkład β).

Alternatywą jest wybranie pozycji **Load...** z menu **Generators** umożliwiającej wczytanie zbioru punktów ze stworzonego uprzednio pliku.

Skorzystanie z obu sposobów kończy się wykreśleniem zbioru wylosowanych punktów zrzutowanego do przestrzeni x_1, x_2. którą to następnie przestrzeń możemy zmodyfikować, wybierając (po zakończeniu generacji) poprzez przycisk **Display** opcję pozwalającą na sprecyzowanie, które zmienne mają zostać wyświetlone. Przycisk **Density** umożliwia wyświetlenie histogramu tworzącego wykres gęstości rozkładu w zadanym wymiarze.



Rysunek 4.23: Sekwencja Sobola

Przykładowy efekt działania zamieszczono na rysunku 4.23. Pokazuje on przebieg generowania punktów rozkładu jednostajnego przy wykorzystaniu sekwencji Sobola.

UWAGA: W przypadku losowania punktów o rozkładzie normalnym, wartości brzegowe obszaru w $k-tym$ wymiarze definiuje się jako:

$$(EX_k - 3 \cdot VAR_k, EX_k + 3 \cdot VAR_k),$$

gdzie EX_k oraz VAR_k oznaczają odpowiednio wartość oczekiwaną i wariancję rozkładu w $k-tym$ wymiarze.

Rozdział 5

Opis implementacji bibliotek

5.1 Typy danych i klasy fundamentalne

Obliczenia algorytmów optymalizacji są wykonywane na zmiennych zdefiniowanego typu *numType*, którego definicja znajduje się w pliku **gool_defs.h**. Umożliwia to dokonywanie zmiany podstawowego typu danych, co ma wpływ na dokładność wykonywanych obliczeń. W wersji obecnej programu [1] dostosowano również wykorzystywany kod z biblioteki *Numerical Recipes*[6][2], aby prawidłowo uwzględniał definicję typu dokładności obliczeń.

```
typedef float   numType
typedef numType *goolVector
```

goolVector jest podstawowym typem danych definiującym wektor *n-wymiarowy*. Do jego tworzenia oraz usuwania zaleca się używania funkcji z modułu **vector_matrix.cc**, zawierającego pomocnicze funkcje przeznaczone do operacji na wektorach i macierzach:

```
goolVector newGoolVector(unsigned int dim);
void       freeGoolVector(void *p);
```

5.1.1 Task

Klasa **Task** służy do reprezentacji zadania w systemie GOOL. W danej chwili może być utworzonych dowolnie wiele obiektów tego typu, mogą one również tworzyć hierarchię, tj. możliwe jest definiowanie i rozwiązywanie podzadań w stosunku do zadań nadrzędnych. Najważniejsze funkcje tej klasy to:

- **zarządzanie postacią zadania**

Umożliwia definiowanie wyrażenia w postaci symbolicznej, zadawanie ograniczeń nierównościowych, ich aktywację i deaktywację, określanie parametrów związanych z optymalizacją Klasa pozwala również na pobranie informacji o definicji zadania, co pozwoliło m.in. znacznie uprościć kod związany z obsługą systemu dialogowego GOOL/GUI.

[1] W wersji VSO2 wykorzystywane funkcje biblioteki *Numerical Recipes* operowały na typie *float*, niezależnie od definicji typu *numType*.

[2] Do biblioteki GOOL włączono pewne procedury z NR. Są to: jacobi, pythag, simplx, simp1, simp2, simp3, tred2, tqli.

- **obliczanie wartości funkcji celu**

Poprzez metodę *calculate_f(goolVector)* algorytmy pobierają informację o wartości funkcji celu dla zadanego punktu (z uwzględnieniem aktywnych ograniczeń). Jej implementacja uwalnia również metody optymalizacji od jawnego zapamiętywania dotychczasowego najlepszego rozwiązania, czyniąc to w sposób automatyczny, podczas każdego obliczenia wartości funkcji celu.

Uwzględnianie ograniczeń odbywa się poprzez dosumowywanie kolejnych członów kary związanych z naruszaniem ograniczeń aktywnych bieżącego zadania (jest to tzw. algorytm zewnętrznej funkcji kary). Postać członu wynika z parametrów przekazanych klasie **Task**, a ogólną jej postać przedstawia wzór:

$$\phi(x) = M \cdot \max(0, g_i(x))^p \tag{5.1}$$

,

gdzie M oraz p są wprowadzanymi parametrami, $g_i(x)$ reprezentują ograniczenia nierównościowe w postaci $g_i(x) \leq 0$. Tym samym metody optymalizacji rozwiązują zmodyfikowane zadanie bez ograniczeń, zaś o prawidłowe budowanie postaci zmodyfikowanej funkcji dba część systemu GOOL zarządzająca zadaniami.

Klasa umożliwia także pobranie wartości wynikającej z przekroczenia ograniczeń dla danego punktu. Tym samym możemy określić, czy dany punkt jest dopuszczalny i otrzymać ilościową informację o niespełnieniu warunków zadania. Dla dokładniejszej analizy możliwe jest również pobranie wartości poszczególnych ograniczeń dla zadanego punktu.

Kilku słów komentarza wymaga budowanie funkcji kary dla ograniczeń nierównościowych zadawanych w postaci rodzin wyrażeń indeksowanych. W takiej sytuacji tworzone jest sztuczne wyrażenie sumujące wartości ograniczeń należących do jednej rodziny indeksowanej i ta sumaryczna wartość zostaje uwzględniana w członie związanym z karą.

```
g1_i(1:5)( x_i * cost_i - ogrmax_i)
```

zostaje przekształcone do

```
g1Total = sum_i(1:5)(max(0, g1_i))
```

Wartość wyrażenia **g1Total** jest następnie podnoszona do wykładnika *penaltyp* oraz przemnażana przez współczynnik kary (obie wielkości to parametry zadania).

- **obsługa kryteriów stopu zadania**

W obecnej wersji kryterium tym jest zbliżenie się znalezionego rozwiązania do zadanej na wstępie, poszukiwanej wartości. Klasa reaguje również na „ręczne” zatrzymanie optymalizacji (przez metodę lub np. jako reakcja na zdarzenie zewnętrzne).

Klasa **Task** dostarcza również metody niezbędne do przesyłania informacji związanych ze stanem optymalizacji przez metody do programu GOOL/GUI. Są one dostępne jedynie wówczas, jeśli budujemy system okienkowy, dlatego ich wywołanie powinno być otoczone odpowiednimi dyrektywami warunkowymi preprocesora. Komunikacja ze środowiskiem GOOL/GUI opisana jest w dalszej części pracy.

- **problem podzadania**

Obiekt klasy **Task** może reprezentować zarówno zadanie główne, jak i zadanie działające na rzecz innego zadania wyższego poziomu. W tym drugim wypadku do obiektu zadania nadrzędnego zostaje delegowana obsługa wywołań niektórych metod, jak np. obliczenie wartości funkcji celu, sprawdzenie globalnych kryteriów stopu. Alternatywny sposób tworzenia i rozwiązywania podzadań (np. zadań lokalnych) umożliwia klasa **AlgUsingLocal**.

5.1.2 Algorithm

Klasą bazową wszelkich metod optymalizacji jest klasa **Algorithm**. Jest to klasa abstrakcyjna, definiująca pewien scenariusz zachowania, wg. którego muszą działać klasy potomne (odpowiadające w większości przypadków konkretnym metodom). Przejmuje na siebie następujące funkcje:

- **inicjalizacja punktów startowych algorytmu**

Metoda *init_start_points()* umożliwia wykorzystanie przekazanych (o ile są) punktów początkowych z dolosowaniem punktów do liczby wymaganej przez aktualny algorytm. Rozkład punktów dolosowywanych zależy od tego, czy przekazano punkty startowe, oraz od tego, czy algorytm jest metodą nadrzędną, czy też został wywołany na rzecz jakiejś innej metody. W tym drugim przypadku, jeśli algorytm nadrzędny przekazał jakiś punkt startowy, pozostałe (o ile są potrzebne) punkty zostają dolosowane zgodnie z ROZKŁADEM NORMALNYM wokół pierwszego z przekazanych punktów. W pozostałych przypadkach (tj. algorym jest nadrzędny lub jest podrzędny, ale nadrzędny nie przekazał żadnych punktów początkowych), niezbędne punkty zostają wylosowane zgodnie z rozkładem jednostajnym na całym obszarze dziedziny zadania. Użyty zostaje aktywny generator rozkładu jednostajnego (w szczególności - sekwencja pseudolosowa) dla tego algorytmu. Dla każdego obiektu klasy **Algorithm** można ten generator określić indywidualnie.

Taki scenariusz zakłada, że algorytm nadrzędny wymaga od obiektu podrzędnego działania wokół pewnego obszaru lokalnego - stąd taki mechanizm dolosowywania punktów. Przykładem z biblioteki GOOL/OM może być algortym SA, który w pewnych przypadkach wywołuje metodę optymalizacji lokalnej, przekazując jej jeden punkt. Jeśli metoda lokalna jest jednopunktowa, nie dolosowuje żadnych punktów, jednak jeśli jest to np. sympleks Neldera-Meada, dolosowuje punkty wokół przekazanego.

Po zakończeniu swojego działania klasa **Algorithm** automatycznie zwalnia pamięć zawierającą te punkty, które dolosowała na swoje potrzeby wewnątrz metody *init_start_points()*.

UWAGA: przyjmuje się, że punkty startowe przekazane metodzie są wektorami $n+1$-elementowymi, z których ostatni element zawiera wartość funkcji celu w danym punkcie. Jest to konwencja ogólnie przyjęta w bibliotece GOOL/OM.

- **sprawdzenie kryteriów stopu wspólnych dla algorytmów**

Pozwala to na scentralizowane obsłużenie kryterium stopu związanego z przekroczeniem maksymalnej liczby iteracji. W obecnej wersji jest to jedyne tego typu kryterium.

- **obliczenie wartości funkcji celu**

Realizuje ją metoda *evaluate(goolVector)*, która wywołuje metodę *calculate_f(goolVector)* klasy **Task** na rzecz obiektu zadania, w obrębie którego działa dany algorytm.

- **rozpoczęcie procesu optymalizacji**

Optymalizację danego zadania przy użyciu aktywnej dla niego metody wywołuje się poprzez metodę *run()* obiektu klasy **Task**. Dokładna postać deklaracji tej metody to:

`int Task::run(goolVector*,unsigned int,Algorithm*)}, której`

Jako argument można podać wskaźnik do utworzonego algorytmu minimalizacji. Oczywiście algorytm można też zadać wcześniej (i odpowiednio ustawić wartości jego parametrów w kontekście zadania). Dodatkowo można wyspecyfikować jeden lub więcej punktów startowych. Po zakończeniu działania obiekt klasy **Task** kopiuje najlepsze rozwiązanie do przekazanej tablicy *goolVector** oraz w przypadku metod operujących na zbiorach punktów - $k-1$ najlepszych punktów ze zbioru z tego zbioru w momencie zakończenia optymalizacji. Wartość k jest parametrem zadania. Zgodnie z przyjętą konwencją w systemie GOOL, $n+1$-sza współrzędna każdego punktu zawiera w sobie wartość odpowiadającej mu funkcji celu. Dotyczy to zarówno punktów zwracanych, jak i punktów startowych, które opcjonalnie zadajemy podczas wywołania metody *run()*.

5.1.3 Generator

Jest to klasa bazowa dla wszystkich generatorów biblioteki GOOL/RG. Deklaruje abstrakcyjną metodę inicjalizacji generatora oraz wylosowania jednego punktu. Klasy potomne dostarczają ich definicji oraz rozszerzają interfejs o metody dla nich specyficzne.

5.1.4 GenManager

Klasa **GenManager** pełni funkcję menadżera generatorów losowych. To za jej pośrednictwem są tworzone oraz udostępniane generatory zaimplementowane w bibliotece GOOL/RG. Jest ona wykorzystywana zarówno podczas procesu optymalizacji zadania, jak i samoistnie, jeśli zachodzi potrzeba generacji zbioru punktów zgodnie z zadanym wymiarem, rozkładem oraz jego parametrami. Wprowadzenie klasy zarządzającej separuje bibliotekę GOOL/RG od reszty systemu, co znacznie ułatwia jej rozszerzanie o nowe rodzaje generatorów bez konieczności modyfikowania korzystającego z niej kodu.

5.1.5 AlgManager

Klasa **AlgManager** umożliwia zarządzanie zdefiniowanymi metodami optymalizacji w bibliotece GOOL/OM. Pozwala na tworzenie algorytmów na podstawie jego kodu numerycznego lub nazwy symbolicznej, co pozwala na konstruowanie odpowiednich obiektów na podstawie konfigurowalnych parametrów. Jest to istota obsługi algorytmów w systemach GOOL/CON oraz GOOL/GUI.

5.1.6 AlgorithmDirection

Jest to potomek klasy **Algorithm**, przeznaczony jako klasa bazowa dla metod minimalizacji w kierunku. Częściowo modyfikuje zachowanie klasy bazowej tak, by dostosować je do tego specyficznego problemu. Przed rozpoczęciem minimalizacji m.in. dokonuje oszacowania minimalnego i maksymalnego przesunięcia w wybranym kierunku oraz przesyła stosowną informację o rozpoczęciu działania. Dla wersji GOOL/GUI definiuje metody przeznaczone do komunikowania się z systemem okienkowym.

5.1.7 AlgorithmSet

Zamierzeniem tej klasy jest dostarczenie swoim potomkom łatwego zestawu metod ułatwiających operowanie na uporządkowanym zbiorze punktów roboczych algorytmu. Pozwala w łatwy sposób sortować punkty, zastępować punkt najgorszy nowym, usuwać punkty. Obiekty klas potomnych mają gwarancję, iż w pierwszym kroku optymalizacji ich zbiór roboczy będzie posortowany względem wartości funkcji celu punktów składowych (wstępne posortowanie można opcjonalnie wyłączyć). Potomkami tej klasy są m.in. **CRS23** i **CRS6**.

5.1.8 AlgUsingLocal

Klasa ta pozwala na uruchomienie metody optymalizacji lokalnej podczas działania danej metody globalnej. Funkcjonalność tę zapewnia metoda *begin_local_search()*, zwracająca kod kryterium stopu algorytmu lokalnego. Metoda lokalna działa na rzecz tego samego zadania, jednak jest „postrzegana" przez nie jako metoda pomocnicza. Wszystkie algorytmy, które wykorzystują ten sposób do rozwiązania zadania lokalnego podczas swojego kroku działania powinny być potomkami tej klasy. Przykładami z biblioteki GOOL/OM są metody: CRS3, SA, Törn.

5.1.9 AlgUsingDirection

Klasa ta pozwala na uruchomienie minimalizacji w kierunku przez daną metodę globalną lub lokalną. Metoda wywołująca specyfikuje punkt startowy oraz kierunek minimalizacji. Wszystkie algorytmy, które w czasie swego działania wykorzystują minimalizację kierunkową powinny być potomkami tej klasy. Przykładami z biblioteki GOOL/OM są tutaj: Powell oraz BFGS.

5.2 Hierarchia metod optymalizacji

Metody optymalizacji zaimplementowane w systemie GOOL tworzą spójną hierarchię klas. Schematyczny obraz tej dość rozbudowanej struktury zawierają diagramy klas 5.1 - 5.5, przedstawiające najważniejsze powiązania i właściwości pomiędzy algorytmami. Diagramy sporządzono zgodnie z obowiązującą notacją UML, wykorzystywaną do dokumentowania projektów obiektowych systemów informatycznych.

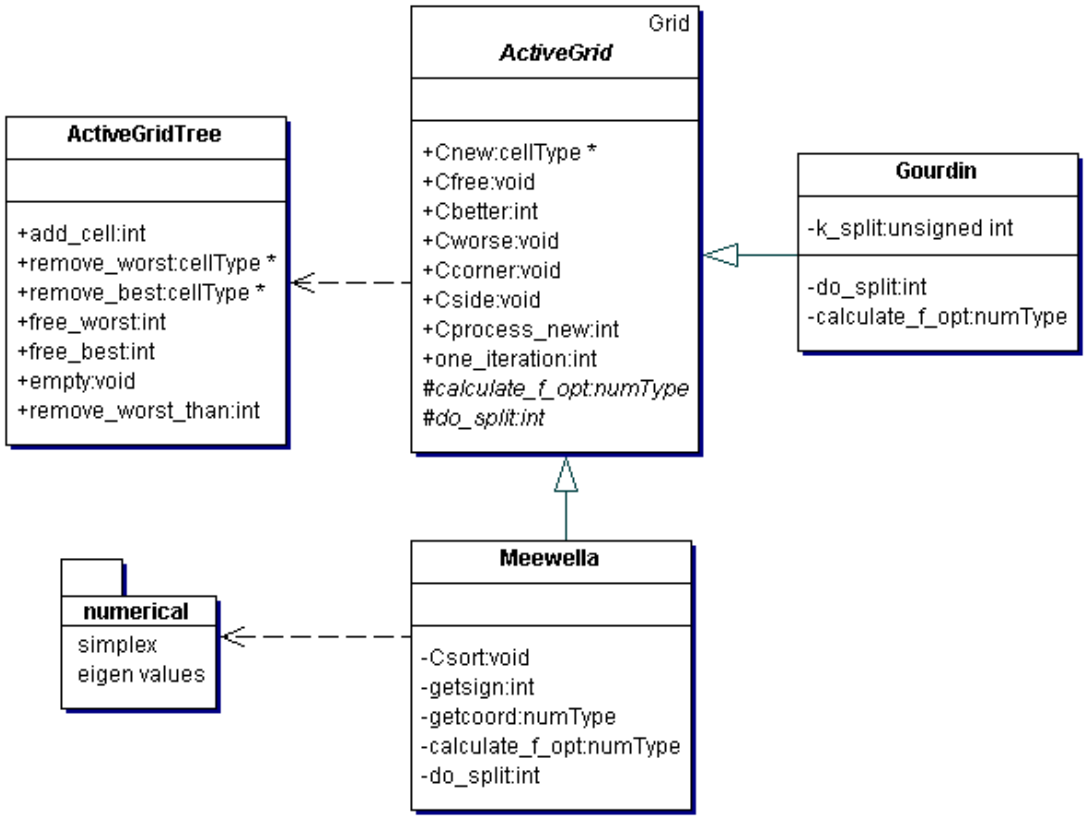
5.3 Komunikacja biblioteki GOOL → GOOL/GUI

5.3.1 Komunikaty ogólnego przeznaczenia

Implementacja tych metod znajduje się w klasie **Task**. Klasa ta niezależnie komunikuje się z systemem GOOL/GUI również w momencie wykrycia nowego najlepszego z dotychczas znalezionych rozwiązania. Wywołanie poniższych funkcji poza trybem edukacyjnym jest ignorowane.

Informacja o generowanych punktach

```
#ifdef __GOOLGUI_H
void Task::GUI_report(int kind, goolVector *points, int index,
```



Rysunek 5.1: Algorytmy siatki aktywnej

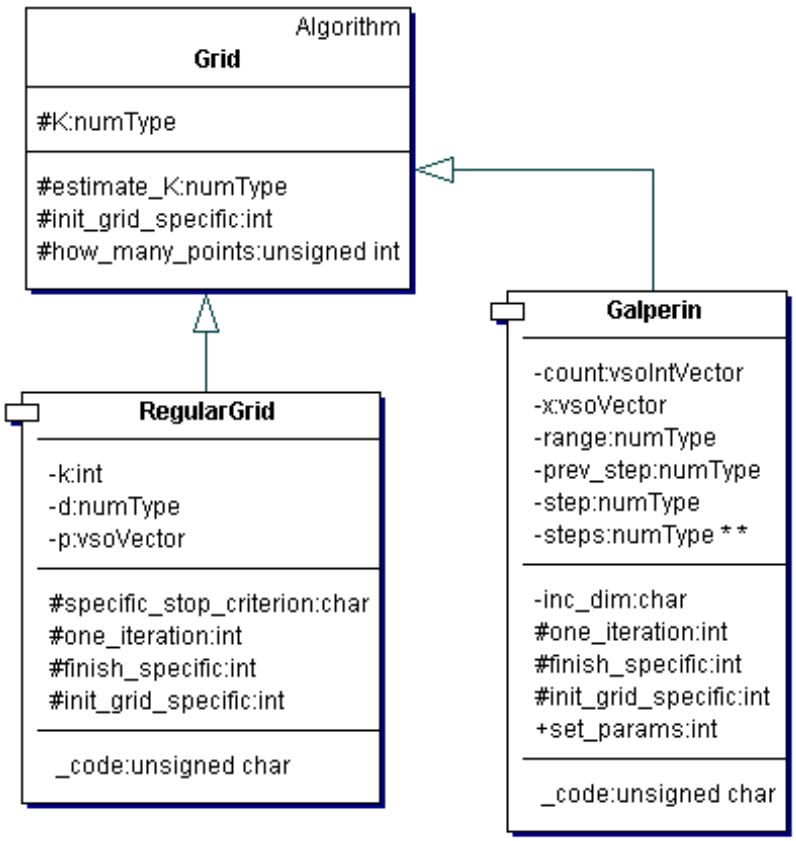
```
                unsigned int howMany,
                unsigned int aux = 0);
#endif
```

Interpretacja poszczególnych parametrów zależy zarówno od charakteru aktywnej metody optymalizacji, jak i wartości parametru *kind*, określającego typ przekazywanej informacji. Jego możliwymi wartościami są:

- **SET_COLOR** - podobnie jak INFO, jej zastosowanie jest czysto informacyjne; umożliwia zmianę koloru „bytu” w danym momencie iteracji, co obrazować ma jego aktywację bądź deaktywację; termin „byt” oznaczać może - w zależności od rodzaju aktywnej metody - jeden punkt (metoda jednopunktowa), grupę punktów lub komórkę siatki. Indeks koloru przekazywany jest w argumencie *aux* i musi być numerem któregoś ze zdefiniowanych zestawów (zestaw składa się z koloru obramowania oraz wypełnienia) kolorów. Kolejne zestawy mają przypisane etykiety symboliczne, zdefiniowane w pliku **gool_gui.h**.

  Byt definiowany jest przez współrzędne przesłanych *howMany* punktów lub podanie jego indeksu w zbiorze roboczym metody. Efektem działania jest wyróżnienie go kolorem aktywnym na tle wykresów poziomicowych.

  Opcja ta wykorzystywana jest m.in. do pokazywania bieżąco rozpatrywanej komórki siatki w metodzie Gourdina oraz Meewella-Mayne. Wykorzystuje ją również metoda CRS3 do wyodrębnienia innym kolorem grupy punktów podlegających procesowi optymalizacji lokalnej, za pomocą metody sympleks nieliniowy.



Rysunek 5.2: Algorytmy siatki regularnej (pasywnej)

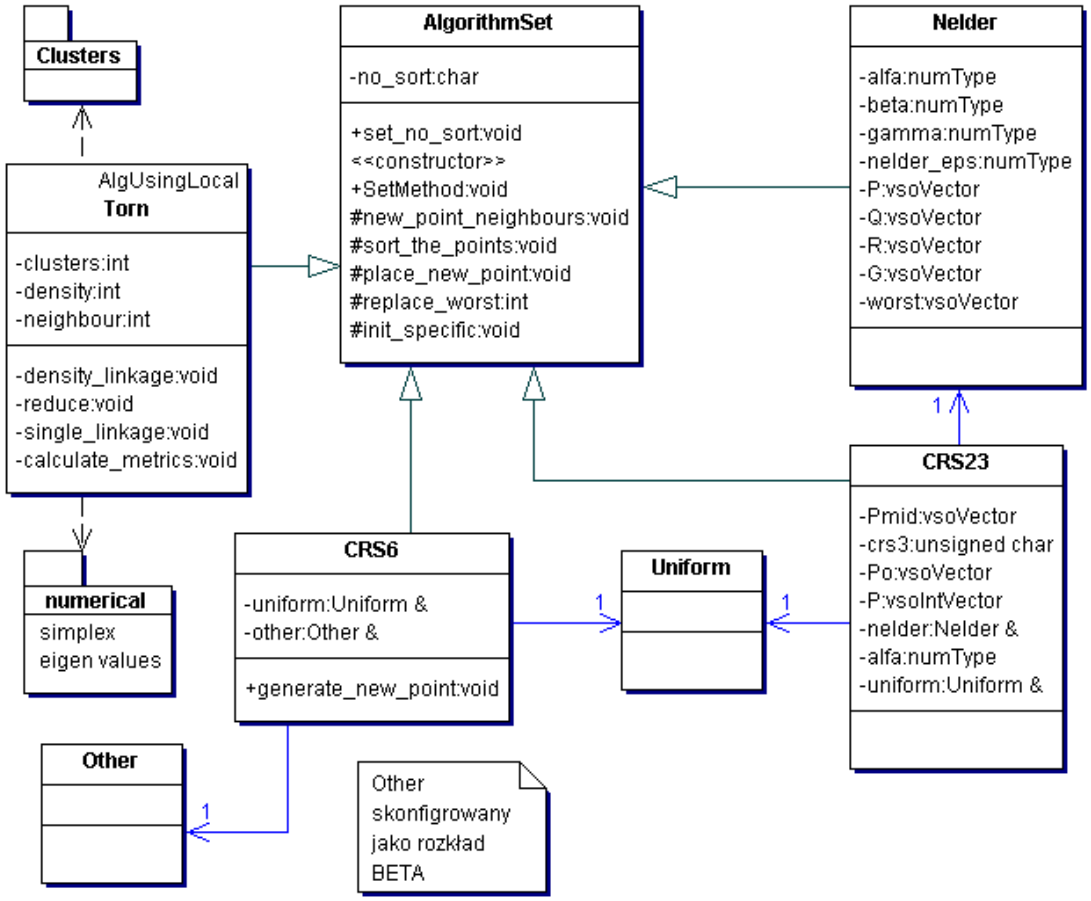
Ponieważ nowe punkty (opcja ADD) rysowane są kolorem domyślnym, istnieje również możliwość jego zmiany - należy wówczas indeks koloru podać w polu *auxe*, przy jednoczesnym ustawieniu pól *howMany* oraz *index* na 0.

Niekiedy potrzebna jest automatyczne przywrócenie wszystkim wyróżnionym bytom koloru początkowego - efekt ten uzyskujemy, podając jako wartość koloru stałą *NORMAL_COLOR*, pole *howMany* ustawiając na 0, zaś polu *index* przypisując wartość -1.

- **ADD** - umożliwia ona podanie całej grupy punktów (o liczebności *howMany*), które powinny zostać dodane (wstawione) do aktualnego zbioru punktów. Opcja stosowana jest w metodach określanych mianem „Population", tj. operujących na zbiorach punktów, jak np. CRS3, CRS6, algorytm genetyczny. Parametr *index* umożliwia podanie, w które miejsce zbioru (tablicy) aktualnych punktów powinny zostać wprowadzone punkty przesyłane. Kolejność przechowywanych punktów (bytów) jest istotna. Dzięki jej zachowaniu możliwe jest operowanie na zbiorze poprzez podawanie indeksów - bez konieczności każdorazowego przesyłania całego zbioru.

  Opcja ADD jest również wykorzystywana w metodach siatki.

- **DELETE** - używane do usuwania punktów. W tym przypadku istotnymi parametrami są *index* oraz *howMany*, specyfikujące ile punktów usuwamy oraz - poczynając od którego punktu. Podejście wykorzystujące indeksację punktów przydaje się w



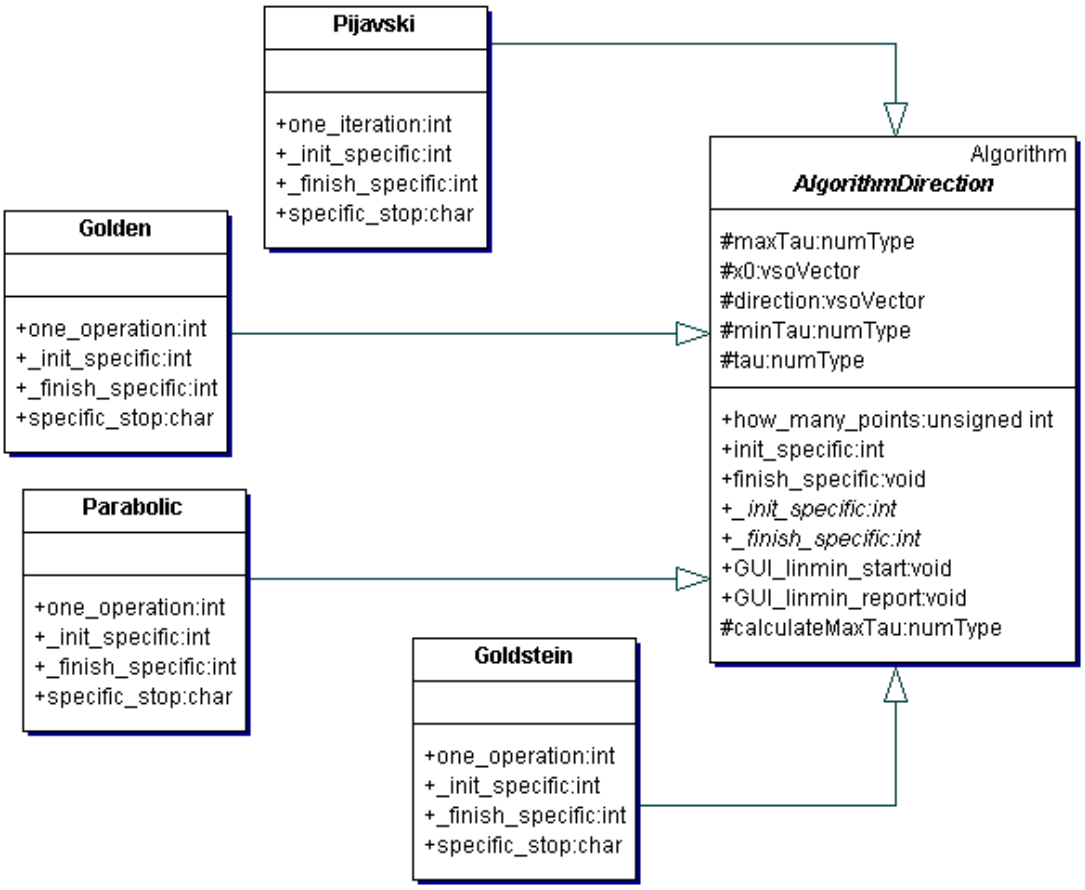
Rysunek 5.3: Algorytmy oparte na zbiorze uporządkowanym

metodach populacyjnych; w przypadku metody siatki wykorzystywane jest specyfikowanie usuwanych punktów poprzez podanie ich współrzędnych (parametr *points*), zaś polu *index* przypisywana jest wartość -1.

- **REPLACE** - opcja będąca połączeniem ADD oraz DELETE - umożliwia wymianę całej grupy punktów, przy czym punkty usuwane adresowane są tutaj TYLKO poprzez pole *index*. Opcja ta używana jest zarówno w metodach populacyjnych, jak i jednopunktowych - wówczas wymieniamy jeden punkt z aktualnego zbioru roboczego, który opcjonalnie może być nowym optimum.

- **OTHER** - opcja wykorzystywana do pozostałych, niestandardowych celów. Obecnie wykorzystywana jest przez algorytm genetyczny do przesyłania zbioru osobników wchodzących w skład populacji przejściowej w każdym kroku iteracji. Przesyłana informacja wykorzystywana jest do sporządzenia wykresu, obrazującego liczbę generowanych (poprzez operatory genetyczne) nowych osobników w każdym kroku z uwzględnieniem powrotu do osobników uzyskanych we wcześniejszych iteracjach. Wykres tego rodzaju ma na celu pokazanie stopnia eksploracji przestrzeni dopuszczalnej zadania.

Skonstruowanie takiej postaci funkcji raportującej ma w założeniu umożliwić prawidłową wizualizację przebiegu algorytmów o różnym charakterze, przenosząc całą, związaną z nią obsługę na skrypty stanowiące część środowiska GOOL/GUI.



Rysunek 5.4: Algorytmy minimalizacji w kierunku

Przekazywanie informacji tekstowej

```
#ifdef __GOOLGUI_H
void Task::GUI_msg(char *msg);
#endif
```

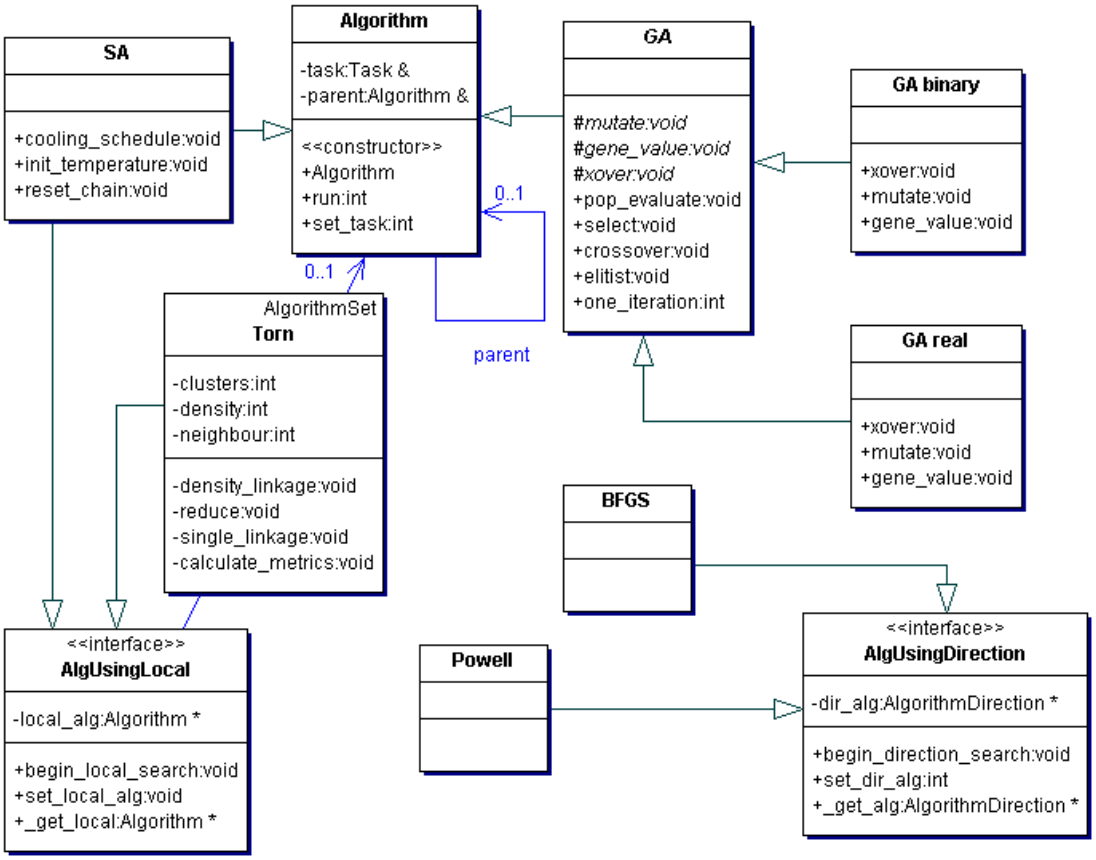
Służy do przekazywania informacji tekstowej o statusie algorytmu. W rezultacie pojawi się ona w polu **Status** okna dialogowego **„GOOL: Report”** (opisanego w rozdziale 4.5.4).

Przekazywanie informacji dodatkowych

```
void Task::GUI_report(numType a, numType b, numType c);
```

Metoda ta przesyła do środowiska GOOL aktualne wartości parametrów uznanych za kluczowe dla danej metody. Efektem jej wykorzystania jest okno raportu przebiegu metody, np. w przypadku algorytmu Griewanka obok standardowych wielkości opisujących działanie metody, dodatkowo prezentowane są wartość wektora prędkości oraz przyspieszenie poruszającej się cząstki. Funkcja pobiera maksymalnie do trzech argumentów rzeczywistych (reprentowanych typem bazowym systemu). Parametry przesyłane w ten sposób do środowiska GOOL są dla każdej metody wyspecyfikowane w pliku **methods.tcl** (patrz rozdział 6.4).

Wyniki przesyłane za pomocą tej metody są traktowane w sposób specjalny w przypadku metody symulowanego wyżarzania oraz algorytmu genetycznego. Oprócz wyświe-



Rysunek 5.5: Pozostałe algorytmy

tlania ich aktualnych wartości program zapamiętuje je w tablicach indeksowanych numerem iteracji po to, aby po zakończeniu działania utworzyć wykresy przedstawiające ich zmianę w czasie.

5.3.2 Metody związane z minimalizacją w kierunku

Ponieważ dotyczą one tylko podgrupy algorytmów, które są potomkami klasy **AlgorithmDirection**, ich implementacja znajduje się właśnie w tej klasie bazowej.

Inicjalizacja

```
void AlgorithmDirection::GUI_linmin_start(char *method,
                                          goolVector x,
                                          goolVector direction,
                                          numType tauMax, int dim);

    method    - stała znakowa identyfikująca dany algorytm;
                aktualnie istniejące stałe to: "GOLDEN", "PARAB_3P",
                "GOLDSTEIN", "PIJAVSKI"
    x         - punkt startowy minimalizacji
    direction - jej kierunek
    tauMax    - granica górna rozpatrywanego przedziału
    dim       - wymiar zadania
```

Metoda ta jest wołana automatycznie podczas rozpoczęcia optymalizacji, tj. klasa potomna nie musi jej jawnie wywoływać. Dostarcza ona informacje dotyczące wykonywanej minimalizacji w kierunku oraz wykreśla przekrój boczny funkcji w oparciu o punkt startowy, kierunek oraz granicę górną przedziału.

Informacja o stanie bieżącym

```
void AlgorithmDirection::GUI_linmin_report(char *str, int howMany, ...)

    str       - rodzaj przesyłanej informacji, którego
                znaczenie rozpatrywane jest w kontekście
                danej metody, przykładowo dla testu
                dwuskośnego Goldsteina dopuszczalne
                stałe to: "TESTS" - przesyłana
                informacja służy wykreśleniu prostych
                stanowiących test jedno- i dwuskośny
                oraz "PARABOLE" - do wyświetlenia
                aproksymacji kwadratowej w przypadku
                niespełnienia testów
    howMany   - liczba pozostałych argumentów
```

Wywołanie metody powoduje uaktualnienie wykresu-przekroju bocznego, z naniesieniem informacji graficznej przedstawiającej stan procesu oraz wyświetleniem wartości parametrów przekazanych jako argumenty dodatkowe (tj. następujące po *howMany*). Sposób prezentacji oraz doboru parametrów, których wartości są przesyłane, jest ściśle związany z daną metodą i jest ukryty w skryptach środowiska GOOL/GUI.

5.4 Komunikacja GOOL/GUI → biblioteka GOOL

5.4.1 Funkcje pomostowe

Komunikacja między programem VSO3/GUI, a bibliotekami VSO3/OM oraz VSO3/RG jest realizowana poprzez zbiór funkcji „pomostowych”, zaimplementowanych w C. Stanowią one rozszerzenie intepretera **wish** o nowe komendy, odpowiedzialne za komunikację z poszczególnymi częściami systemu VSO3. Implementacja tych funkcji znajduje się w podkatalogu **wish/** w gównym drzewie kodu źródłowego. Kod ten NIE JEST kompilowany podczas budowy programu VSO3/CON.

- **task_action args**

Jest to ogólna funkcja przeznaczona do dialogu z obiektami klasy **Task**, których identyfikacja odbywa się poprzez identyfikator numeryczny (zwracany podczas wywołania funkcji tworzącej nowe zadanie). Zdefiniowano następujące argumenty wywołania:

- NEW
- LOAD filepath
- PARAMS task_id params
  służy do ustawienia parametrów zadania (np. funkcja kary, dokładność obliczeń,

wymiar, ograniczenia kostkowe) lub pobrania ich wartości (dla params równego napisowi GET). Format łańcucha parametrów jest analogiczny do postaci łańcucha dla parametrów metod optymalizacji - jest to ciąg par *name=value* rozdzielonych średnikami,

- PENALTY task_id point
  otrzymanie wartości kary za naruszenie ograniczeń w danym punkcie,
- CONSTRAINT ADD task_id constr_name definition
  dodanie ograniczenia o podanej nazwie i definicji do zadania,
- CONSTRAINT TOGGLE task_id constr_name
  funkcja zmienia stan aktywności ograniczenia o podanej nazwie. Tylko ograniczenia aktywne partycypują w obliczaniu wartości funkcji celu. Na początku każde ograniczenie jest aktywne,
- CONSTRAINT ACTIVE task_id constr_name
  pobranie informacji o tym, czy ograniczenie o zadanej nazwie jest uzwględniane podczas obliczania wartości funkcji celu,
- CONSTRAINT LIST task_id constr_name
  pobranie listy symboli wszystkich ograniczeń zdefiniowanych dla zadania,
- CONSTRAINT DEFINITION task_id constr_name
  pobranie definicji ograniczenia w postaci napisu,
- SYMBOL ADD task_id name definition
  dodanie nowego symbolu do zadania,
- SYMBOL LIST task_id
  pobranie listy wszystkich symboli zdefiniowanych dla danego zadania,
- SYMBOL DEF task_id name point
  pobranie napisowej definicji symbolu o zadanej nazwie,
- SYMBOL VALUE task_id name point
  pobranie wartości symbolu dla przekazanego argumentu.
- RUN task_id start_point
  rozpoczęcie procesu optymalizacji zadania aktywnym dla niego algorytmem. Parametr *start_point* oznacza punkt startowy i jest opcjonalny.

- **task_eval task_id point**

Funkcja służy do obliczenia wartości funkcji celu dla podanego argumentu. Parametr *point* powinien być w postaci tekstowej reprezentacji wektora np:

```
set val [ task_eval $task_id { { 10, 20, 10 } } ]
```

Obliczenie uwzględnia człon kary za naruszone ograniczenia aktywne.

- **method_action args**

Funkcja ta umożliwia zarządzaniem metodami, tworzeniem ich (w sensie obiektów odpowiednich klas), konfigurowaniem ich parametrów oraz pobieraniem informacji o stanie. Tym samym kod Tcl systemu okienkowego zostaje w znacznym stopniu uproszczony, zajmując się teraz tylko pobieraniem odpowiednich danych i ich wyświetlaniem.

- PARAMS method_name param_string
  ustawienie parametrów danej metody.

- **generator_action args**

  Za pomocą tej funkcji uzyskujemy dostęp do generatorów biblioteki VSO3/RG.

  - ALLOCATE family which
    alokacja generatora danej rodziny oraz danego wariantu w obrębie tej rodziny. Późniejsze adresowanie generatora jest identyczne przez tę samą parę (family,name),
  - GENERATE family which dim params
    wygenerowanie punktu przez zadany generator o zadanej wymiarowości, z uwzględnieniem wartości parametrów charakterystycznych w kontekście wybranego generatora. W przypadku rozkładu jednostajnego i sekwencji losowych, podajemy tu wartość wartości kostkowe obszaru; w przypadku rozkładu normalnego zadajemy tu wektory wartości oczekiwanej oraz wariancji. Rozkład beta (jednowymiarowy) pobiera wartości swoich parametrów *a* i *b*.

5.5 Realizacja metod optymalizacji funkcji wielu zmiennych

Omówione metody optymalizacji zostały zrealizowane i mogą być uruchamiane w środowisku GOOL/GUI. Algorytmy mogą być też wykorzystane do obliczeń poza systemem GUI - dokładniejsze informacje na ten temat zawarte są w ostatniej części pracy.

Każda metoda optymalizacji zaimplementowana jest w oddzielnym pliku jako klasa potomna klasy bazowej **Algorithm** i skompilowana do postaci modułu relokowalnego (**.obj** w systemie Windows, bądź **.o** w systemie Linux). Niektóre metody ze względu na swoją specyfikę są potomkami pewnych klas pośrednich, które same dziedziczą z klasy bazowej **Algorithm**. Dokładniejsze informacje zawiera rozdział 5, w którym zawarto diagram klas metod biblioteki GOOL/OM.

Każda metoda działa w obrębie zadania (realizowanego przez klasę **Task**), które to narzuca następujące parametry (konfigurowalne na poziomie zadania):

- epsilon - dokładność wykonywanych obliczeń. Wartość ta jest używana podczas porównywania liczb zmiennoprzecinkowych (np. przekraczanie zakresu), testowania spełnienia kryteriów STOP-u itp.

Zaimplementowanie metod w postaci hierarchii klas oraz rozdzielenie funkcjonalności na klasy **Algorithm** (wraz z klasami pomocniczymi) oraz **Task** pozwoliło scentralizować pewne wspólne dla wszystkich metod operacje, oraz znacznie uprościło (w stosunku do poprzedniej wersji programu [8]) sposób dodawania nowych metod. W niniejszym rozdziale omówione są parametry charakterystyczne netod zaimplementowanych w bibliotece GOOL/OM oraz indywidualne kryteria stopu.

5.5.1 Metody optymalizacji lokalnej

Sympleks nieliniowy Neldera-Meada

Zaimplementowana wersja metody sympleks nieliniowy jest o tyle specyficzna, iż jej działanie jest różne w zależności od kontekstu, w jakim zostaje wywołana. Możemy wyróżnić tu dwa przypadki:

1. Metoda działa jako algorytm autonomiczny - zostaje jej wówczas przekazany jeden punkt startowy. Algorytm na samym początku dolosowuje *n* punktów sympleksu z obszaru dopuszczalnego oraz ustawia wartości parametrów indywidualnych zgodnie z przekazanymi parametrami wywołania,

2. metoda zostaje wywołana jako metoda lokalna algorytmu globalnego CRS2 (CRS3 to właśnie połączenie metody CRS2 z wywoływaniem sympleksu Neldera-Meada); wówczas nie losuje punktów sympleksu, ponieważ zostają jej one jawnie przekazane. Działanie różni się w stosunku do standardowego tym, że punkty otrzymane w wyniku odbicia, ekspansji i kontrakcji porównywane są nie z najgorszym punktem sympleksu, ale z najgorszym ze wszystkich lepszych od najgorszego. Dodatkowo nie następuje tu krok redukcji sympleksu - konieczność jego wykonania jest równoznaczna z zakończeniem działania metody.

UWAGA: ponieważ metoda może być wywołana jako podalgorytm, klasa **Algorithm** obsługuje wówczas sytuację, kiedy zostanie jej przekazany przez algortym nadrzędny tylko jeden punkt początkowy. Wówczas następuje dolosowanie pozostałych *n* punktów zgodnie z rozkładem normalnym (aktywnym dla zadania, na rzecz którego działa metoda) wokół przekazanego punktu. Jest to *de facto* jedyny algorytm w bibliotece GOOL/OM, który wykorzystuje tę właściwość, gdyż pozostałe algorytmy lokalne (wykorzystywane przez algorytmy nadrzędne) są jednopunktowe. Niemniej struktura ta umożliwia łatwe dodanie kolejnych metod lokalnych operujących na liczbie punktów > 1.

Parametry indywidualne:

- alfa - współczynnik odbicia nagorszego punktu względem centroidu,
- beta - współczynnik ekspansji punktu odbitego,
- gamma - współczynnik kontrakcji punktu odbitego,

Specyficzne kryteria STOPU (patrz [5, 12]):

- 1 - zmniejszenie odległości między środkiem symetrii sympleksu a jego wierzchołkami poniżej wartości granicznej,
- 2 - zmniejszenie odległości między wierzchołkami najlepszym i najgorszym poniżej wartości granicznej,
- 3 - punkt odbicia nie znalazł się wewnątrz dziedziny, jak również punkt kontrakcji,
- 4 - metoda została wywołana jako algorytm lokalny metody CRS3 i punkt kontrakcji nie jest lepszy od najgorszego ze wszystkich lepszych od najgorszego.

Raportowane parametry [3]: BRAK
Implementacja znajduje się w pliku **Nelder.cc**.
Metoda Powella

Metoda Powella została zaadoptowana z biblioteki Numerical Recipes [6] i dostosowana do wymagań programu GOOL/OM. Implementacja metody znajduje się w pliku głównym **Powell.c**, włączającym plik nagłówkowy **Powell.h**. Do minimalizacji w kierunku metoda korzysta z czterech opisanych algorytmów zaimplementowanych w bibliotece GOOL/OM.

Parametry indywidualne: BRAK
Dodatkowe kryteria STOPU:

- 0 - kryterium metody polegające na osiągnięciu minimalnej odległości między kolejnymi dwoma wyznaczonymi punktami.

Raportowane parametry: BRAK
Metoda BFGS

Implementacja metody znajduje się w pliku **BFGS.cc**; metoda wykorzystuje algorytmy minimalizacji w kierunku zaimplementowane przez autora.

Parametry indywidualne: BRAK
Dodatkowe kryteria STOPU:

- 0 - minimalne przesunięcie w stosunku do punktu z poprzedniej iteracji.

Raportowane parametry: BRAK

5.5.2 Metody siatki pasywnej

Metody siatki są potomkami klasy **Grid** i dziedziczą z niej następujące parametry ogólne:

- L - stała Lipschitza minimalizowanej funkcji. Wartość równa 0 (domyślna) oznacza, że algorytm na początku działania będzie próbował ją oszacować.

Raportowane parametry: L.
Metoda siatki regularnej

Implementacja znajduje się w pliku **RegularGrid.cc**, zaś deklarację zawiera plik **RegularGrid.h**.

Parametry indywidualne: BRAK
Dodatkowe kryteria STOPU: BRAK
Raportowane parametry: d (wyznaczona stała siatki).

[3] tzn. przesyłane przez metody w momencie działania w trybie ***Educational Mode***.

5.5.3 Metody siatki aktywnej

Algorytmy Galperina, Gourdina oraz Meewella-Mayne są potomkami klasy **ActiveGrid**, która posiada następującą charakterystykę ogólną:

Parametry indywidualne: BRAK
Dodatkowe kryteria STOPU: BRAK
Raportowane parametry: F_{opt}, f_{opt}.
Algorytm Galperina

Implementacja znajduje się w pliku **Galperin.cc**, który dołącza plik nagłówkowy **Galperin.h**.

Parametry indywidualne: BRAK
Dodatkowe kryteria STOPU: BRAK

Uwagi implementacyjne.

Ponieważ metoda operuje na komórkach będących hiperześcianami, daną komórkę definiuje punkt narożny oraz jej rozmiar.
Algorytm Gourdina

Implementacja znajduje się w pliku **Gourdin.cc**, który dołącza plik nagłówkowy **Gourdin.h**.

Parametry indywidualne:

- d - parametr podziału; specyfikuje, na ile komórek dzielić będą się komórki bazowe w trakcie działania algorytmu (a ściślej - ich najdłuższe boki).

Dodatkowe kryteria STOPU:

- 0 - minimalna różnica między oszacowaniem górnym (f_{opt}) i dolnym (F_{opt}).

Uwagi implementacyjne.

Ponieważ metoda operuje na komórkach będących hiperprostopadłościanami, daną komórkę definiują dwa punkty narożne. Tym samym komórki w tej metodzie zajmują więcej pamięci, niż w metodzie Galperina.
Algorytm Meewella-Mayne

Implementacja znajduje się w pliku: **Meewella.cc**, zaś jej deklaracja w pliku **Meewella.h**.

Parametry indywidualne: BRAK
Dodatkowe kryteria STOPU:

- 0 - minimalna różnica między oszacowaniem górnym (f_{opt}) i dolnym (F_{opt}).

Raportowane parametry: BRAK

Uwagi implementacyjne.

Ponieważ komórki dzielone są na podstawie rozwiązania zadania minimaksowego, które jednocześnie stanowi oszacowanie dolne komórki, do jej reprezentacji dołączany jest wyznaczony punkt podziału. Zadanie minimaksowe, rozwiązywane tu procedurą **simplx** z pakietu Numerical Recipes [6], polega na znalezieniu maksimum sztucznej zmiennej ograniczonej od góry hiperpłaszczyznami wychodzącymi z rogów komórki w kierunku jej środka. Nachylenie hiperpłaszczyzn co do wartości równe jest obowiązującej dla zadania stałej Lipschitza. Ponieważ jest to zadanie maksymalizacji, budujemy je transformując geometrię zadania oryginalnego poprzez symetrię wokół osi OX. Jednocześnie, aby zapewnić nieujemność maksymalizowanej zmiennej sztucznej, podnosimy hiperpłaszczyzny o wartość równą maksymalnej wartości (w zadaniu oryginalnym) funkcji celu w punktach narożnych komórki. Jeśli wartość maksymalna jest ujemna, przesunięcie to jest zbędne.

UWAGA: mimo że powyższe trzy metody operują na nieco innych strukturach reprezentujących komórki, ze względu na efektywność zrezygnowano z tworzenia odpowiadającego im drzewa klas. Algorytmy operują na strukturach, w których pola są ułożone w ten sposób, iż operacje możliwe są poprzez zwykłe rzutowanie ogólnego typu struktury do typu specyficznego dla danej metody. Oczywiście klasy potomne **ActiveGrid** muszą dostarczyć dodatkowych metod związanych z inicjalizacją i dealokacją komórek, uwzględniając pola dla nich specyficzne.

5.5.4 Metody trajektorii cząstki

Algorytm Griewanka

Implementacja metody znajduje się w pliku **Griewank.cc** [4]. Wartość funkcji celu oraz gradientu obliczane są przez procedury opisane w poprzednich punktach rozdziału.

Parametry indywidualne:

- M - współczynnik skalujący krok,

Dodatkowe kryteria STOPU: BRAK
Raportowane parametry: prędkość cząstki, przyspieszenie cząstki

5.5.5 Metody poszukiwania losowego

Algorytm symulowanego wyżarzania

Implementacja metody znajduje się w pliku **SA.cc**, który dołącza plik nagłówkowy **SA.h**.

Parametry indywidualne:

- p_loc - parametr decydujący o włączeniu metody lokalnej,
- n_step - liczba kroków metody lokalnej; gdy n_step=0, metoda lokalna zatrzymuje się po spełnieniu kryterium stopu.

[4] Wykorzystano implementację p. Roberta Śliwińskiego z pracy [12].

Dodatkowe kryteria STOPU:

- 0 - osiągnięcie przez temperaturę T wartości granicznej T_{min}.

Raportowane parametry: temperatura T

Po zakończeniu przebiegu w trybie edukacyjnym możliwe jest obejrzenie wykresu wartości temperatury w kolejno rozpatrywanych łańcuchach Markowa. Dodatkowo generowany jest wykres prawdopodobieństwa akceptacji punktu gorszego: największego, najmniejszego oraz średniego w każdym łańcuchu Markowa.

Algorytmy CRS2 i CRS3

Oba algorytmy realizuje funkcja CRS23, której jeden z parametrów decyduje o tym, czy (przy odpowiednich warunkach) włącza się optymalizacja lokalna (właściwa dla algorytmu CRS3). Implementacja znajduje się w pliku **CRS23.cc**, dołączającym plik nagłówkowy **CRS23.h**.

Parametry indywidualne:

- NP - wielkość, która pośrednio określa rozmiar tablicy roboczej algorytmu,
- crs3 - flaga, która przyjmując wartość jeden wymusza na algorytmie działanie zgodne z metodą CRS3.
- max_failure - maksymalna liczba iteracji, w których nowo wyznaczany punkt odbicia nie zawiera się w obszarze dopuszczalnym.

Dodatkowe kryteria STOPU:

- 1 - kolejny punkt wyznaczony jako odbicie punktu najgorszego przez max_failure iteracji nie zawierał się w dziedzinie.

Raportowane parametry: BRAK

Uwagi implementacyjne.

Ponieważ klasa **CRS23** jest potomkiem klasy **AlgorithmSet**. Tym samym operuje w sposób naturalny na posortowanym zbiorze punktów, posługując się odpowiednimi metodami tej klasy zapewniającymi porządek zbioru.

Algorytm CRS6

Implementacja znajduje się w pliku **CRS6.cc**, który dołącza plik nagłówkowy **CRS6.h**.

Parametry indywidualne:

- NP - wielkość, która pośrednio określa rozmiar tablicy roboczej algorytmu,
- M - liczba punktów generowanych rozkładem beta,
- max_failure - maksymalna liczba iteracji, w których nowy punkt wyznaczony jako minimum aproksymacji parabolicznej nie zawiera się w dziedzinie.

Dodatkowe kryteria STOPU:

- 1 - kolejny punkt aproksymacji parabolicznej przez max_failure iteracji nie zawierał się w dziedzinie.

Raportowane parametry: BRAK

Uwagi implementacyjne.

Klasa **CRS6** również jest potomkiem klasy **AlgorithmSet**. W szczególności ułatwia to realizację podkreślonego w opisie teoretycznym metody założenia, by do zbioru były dodawane tylko punkty inne niż już istniejące w tym zbiorze. Ze względu na uporządkowanie zbioru nie ma konieczności porównywania nowego punktu ze wszystkimi jego elementami - wystarczy porównać z sąsiadami, pomiędzy których punkt potencjalnie by trafił. Indeksy tych sąsiadów określa rzecz jasna klasa **AlgorithmSet**.

5.5.6 Algorytmy genetyczne

Zastosowano tu dwa algorytmy - jeden z kodowaniem binarnym, drugi z kodowaniem rzeczywistoliczbowym. Implementacja znajduje się w trzech plikach: **GA.cc**, definiującego klasę bazową, która określa ogólne zachowanie algorytmu, **GA_binary.cc**, dostarczającego operacji typowych dla kodowania binarnego oraz **GA_real.cc**, definiującego operacje dla kodowania rzeczywistego. Część z parametrów jest niezależna od rodzaju kodowania. Odpowiednie pliki nagłówkowe to **GA.h**, **GA_binary.h**, **GA_real.h**.

Parametry indywidualne (wspólne dla obu kodowań):

- popsize - wielkość populacji,
- pxover - prawdopodobieństwo krzyżowania,
- pmut - prawdopodobieństwo mutacji,
- elit_n - wprowadza sukcesję elitarną z parametrem równym wartości *elit_n* - liczba najlepszych osobników populacji bazowej branych pod uwagę przy określaniu puli wynikowej. Wartość zero równoznaczna jest z wyłączeniem sukcesji elitarnej,
- selection - rodzaj selekcji: 0 - proporcjonalna; 1 - rangowa; 2 - turniejowa.

  W przypadku selekcji rangowej przyjęto następujący wzór funkcji $F : r_i \rightarrow p$:

  $$p(i) = a \cdot (r_{max} - r_i)^b \tag{5.2}$$

  gdzie r_{max} jest liczbą o 1 większą od rangi maksymalnej, zaś b to parametr metody. Stała a musi być tak dobrana, by suma prawdopodobieństw wynosiła 1, a zatem równa jest odwrotności z sumy wyrażeń $(r_{max} - r_i)^b$
- par1 - w przypadku selekcji rangowej określa wartość współczynnika b funkcji F, zaś w przypadku selekcji turniejowej - wielkość zbioru turniejowego q,
- par2 - w przypadku selekcji rangowej określa parametr ε, rozpatrywany w procesie grupowania osobników w zbiory rangowe; w innych przypadkach jego wartość jest bez znaczenia.

Dodatkowe kryteria STOPU: BRAK

Raportowane parametry: f_{avg}, f_{min}, f_{avg} - czyli odpowiednio średnia, minimalna oraz maksymalna wartość funkcji przystosowania populacji w danej iteracji.

Parametry indywidualne (kodowanie rzeczywistoliczbowe):

- mutation - 0 (mutacja zgodna z rozkładem Gaussa) lub 1 (wykorzystanie rozkładu Cauchy'ego).

Po zakończeniu eksperymentu w trybie edukacyjnym możliwe jest obejrzenie wykresów wartości raportowanych parametrów w kolejnych iteracjach metody, oraz wykresu pokazującego jakość przeszukiwania przestrzeni dopuszczalnej wyrażoną przez liczbę nowych osobników generowanych w każdej iteracji (ze specjalnym uzwględnianiem osobników, które pojawiły się w iteracjach poprzednich).

5.5.7 Algorytm populacyjny Törna

W bibliotece GOOL/OM zastosowano trzy rodzaje grupowania:

- oryginalne, oparte na estymacie gęstości punktowej,
- single-linkage, wykorzystujące metrykę euklidesową,
- $k-tego$ sąsiada - jest to odmiana grupowania *single-linkage*, operująca na zmodyfikowanej metryce.

Opis teoretyczny powyższych grupowań zawiera praca [9].

Implementacja metody znajduje się w pliku **Torn.cc**, który dołącza plik nagłówkowy **Torn.h**. Do operacji na clustrach wykorzystany został moduł pomocniczy **clusters.c**

Parametry indywidualne:

- density - wartość binarna, określa czy grupujemy w oparciu o technikę gęstościową, czy też nie,
- neighbours - decyduje o tym, czy korzystamy z algorytmu grupowania $k-tego$ sąsiada. W tym wypadku $neighbours = k$. Wartość 0 oznacza, iż obowiązuje albo grupowanie gęstościowe, (dla density = 1), albo podstawowa metoda *single-linkage*.

Dodatkowe kryteria STOPU:

- 1 - liczba klastrów z poprzedniego kroku nie zmieniła się w stosunku do kroku bieżącego,
- 2 - w każdym klastrze pozostał tylko jeden punkt.

Raportowane parametry: liczba zidentyfikowanych klastrów, estymowana gęstość średnia obszaru.

Uwagi implementacyjne.

Podczas mechanizmów grupowania niezbędne jest określenie odległości pomiędzy każdą parą punktów zbioru, na którym operujemy. Stosujemy tu albo zwykłą metrykę euklidesową (grupowanie gęstościowe lub *single-linkage*, lub jej modyfikację ($k-tego$ sąsiada). W celu oszczędzenia pamięci odległości przechowywane są w tablicy trójkątnej, co powoduje, iż tablica taka zajmuje w pamięci L elementów wyrażonych wzorem:

$$L = \frac{n(n-1)}{2}$$

w porównaniu z n^2 elementów dla tablicy kwadratowej.

Dla każdego rodzaju grupowania punkty w klastrach dołączane są w taki sposób, aby pierwszy z nich był jednocześnie najlepszy. Zapewnia to zachowanie najlepszego punktu z każdego klastra podczas kroku redukcji.

Podczas grupowania gęstościowego dotychczas niepogrupowane punkty zostają posortowane względem odległości euklidesowej do punktu, wokół którego budujemy dane centrum.

Grupowanie niegęstościowe jest realizowane poprzez sukcesywne łączenie klastrów znajdujących się wobec siebie bliżej, niż ustalona odległość progowa. Klastry zorganizowane są w listę dwukierunkową i operują jedynie na indeksach punktów, które są do nich włączane. Ponieważ klasa **Torn** wykorzystuje klasę **AlgorithmSet**, punkty z rozpatrywanego zbioru przed rozpoczęciem grupowania są posortowane w kolejności od najgorszego do najlepszego. Łączenie klastrów w pary odbywa się w podwójnej pętli przeglądającej wszystkie klastry w kolejności ich występowania na liście. Wynikają z tego następujące fakty:

- na początku klastry są (powstają z pojedynczych punktów) uszeregowane od najgorszego do najlepszego,
- w momencie dołączania drugiego klastra do pierwszego, dołączamy klaster **lepszy** do **gorszego** - zatem konieczne jest dołączenie pierwszego punktu z drugiego klastra na początek klastra pierwszego, gdyż jest to nowy punkt najlepszy w klastrze wynikowym.

5.6 Realizacja metod poszukiwania minimum w kierunku

Wszystkie metody minimalizacji w kierunku są potomkami klasy abstrakcyjnej **AlgorithmDirection**, która definiuje pewne wspólne parametry:

1. maxTau - jest to wartość maksymalnego przesunięcia w zadanym kierunku. Dodatkowo dla każdej minimalizacji jest wyznaczana maksymalna wartość dopuszczalna tego przesunięcia, wynikająca z obowiązujących ograniczeń kostkowych dziedziny zadania. Zadane maxTau zostaje obcięte do tej wyznaczonej wartości (jeśli tę wartość przekracza).

5.6.1 Metoda złotego podziału

Implementacja metody znajduje się w pliku **direction/Golden.cc**, który dołącza plik nagłówkowy **direction/Golden.h**.

Raportowane wielkości: τ_a, τ_b, τ_c, τ_d, definiujące bieżący przedział w którym znajduje się poszukiwane minimum.

5.6.2 Test dwuskośny Goldsteina

Implementacja metody znajduje się w pliku **direction/Goldstein.cc**, który dołącza plik nagłówkowy **direction/Goldstein.h**.

1. *ro* - parametr decydujący o „obcinaniu" wartości skrajnych rozpatrywanego przedziału, wartości *ro* oraz *1 - ro* stanowią współczynniki kierunkowe prostych wyznaczających akceptowalny obszar parametru τ (dokładniejszy opis - patrz [3, 11]). Dopuszczalne wartości parametru to $(0, 0.5)$.

Raportowane wielkości to:

- w przypadku przesyłania danych o testach raportowane są wielkości: α_1, α_2 (krańce bieżąco rozpatrywanego przedziału), α (potencjalne rozwiązanie), $f(\alpha_1)$, $f(\alpha)$, $\nabla f(\alpha)$,
- w przypadku przesyłania danych o aktualnej aproksymacji parabolicznej, tworzonej w sytuacji niespełnienia któregoś z testów, przesyłane są wartości: *a*, *b*, *c* (współczynniki paraboloidy), α (powielone), $\hat{\alpha}$ (minimum aproksymacji kwadratowej).

5.6.3 Paraboliczna aproksymacja bezgradientowa

Implementacja metody znajduje się w pliku **direction/Parabolic.cc**, który dołącza plik nagłówkowy **direction/Parabolic.h**.

Parametry indywidualne:

1. *tau* - współczynnik początkowego przesunięcia (kroku) próbnego metody,
2. *ksi* - współczynnik wydłużenia kroku, typowe wielkości to *ksi = 2*,

Parametry *tau* oraz *epsilon* decydują o stopniu „skupienia" punktów stanowiących paraboliczną aproksymację oryginalnej funkcji. Szczegółowy opis algorytmu znajduje się w pracy [3].

Raportowanymi wielkościami są: τ_1, τ_2, τ_3, wyznaczające aktualne przybliżenie, *a*, *b*, *c* - współczynniki przybliżenia, f_{xx} - druga pochodna przybliżenia (informująca o tym, czy aproksymacja jest dla bieżących punktów wypukła, czy wklęsła), $\hat{\tau}$ - w przypadku wypukłości aproksymacji zawiera jej minimum (w przypadku wklęsłym wartość ta jest pomijana).

5.6.4 Metoda Pijavskiego-Shuberta

Implementacja metody znajduje się w pliku **direction/Pijavski.cc**, który dołącza plik nagłówkowy **direction/Pijavski.h**.

Parametry indywidualne:

1. *L* - stała Lipschitza funkcji, której sens omawiany był przy opisie aktywnych metod siatki.

Raportowane wielkości: oszacowanie górne f_{opt} i dolne F_{opt}. Dodatkowo metoda przesyła zbiór wartości współczynników τ, informujących o charakterze funkcji podpierającej $F_k\tau$.

Ponadto wszystkie metody (oprócz aproksymacji bezgradientowej) pozwalają wyspecyfikować opcjonalną wartość parametru τ_{max}, ograniczającego z góry rozpatrywany przedział minimalizacji. W przypadku pominięcia jego wartości, funkcja za każdym razem ustala jego wartość na maksymalną z wartości dopuszczalnych, zdefiniowaną jednoznacznie przez punkt początkowy minimalizacji, kierunek oraz istniejące ograniczenia kostkowe.

Rozdział 6

Rozbudowa biblioteki metod

Poniżej przedstawione są wszystkie niezbędne informacje potrzebne do rozszerzenia biblioteki GOOL/OM oraz zbudowania rozszerzonego interpretera Tcl/Tk przeznaczonego do uruchamiania środowiska GOOL/GUI.

6.1 Implementacja klasy realizującej nową metodę

Nowa metoda, którą chcemy dodać do biblioteki GOOL/OM powinna być zaimplementowana jako klasa pochodna od głównej klasy abstrakcyjnej bazowej **Algorithm**. Musi dostarczyć implementacji następujących metod wirtualnych, zadeklarowanych w klasie bazowej:

- get_code(void) - unikalny kod metody. Zalecane jest zdefiniowanie odpowiedniej stałej symbolicznej w pliku nagłówkowym **methods_codes.h** i posługiwanie się nią w każdym kontekście wymagającym podania kodu nowej metody,

- get_name(void) - nazwa metody w postaci łancucha znaków. Metoda ta jest wykorzystywana do alokacji algorytmu poprzez obiekt klasy menadżera algorytmów. Zarówno wersja CON jak i GUI używa interfejsu menadżera do pobierania i tworzenia obiektów metod, zatem metoda ta jest obowiązkowa. Wersja GOOL/CON dodatkowo korzysta z nazwy symbolicznej podczas generowania raportu z przebiegu optymalizacji,

- how_many_points(void) - zwraca liczbę punktów, na których operuje dana metoda. Przykładowo metody jednopunktowe (Powell, SA) zwracają 1, metody operujące na zbiorze - zazwyczaj wartość wynikającą z którejś z opcji metody (Torn, GA) lub wymiaru (Nelder),

- requires_grad(void) - informuje o tym, czy do działania metody niezbędne jest dysponowanie mechanizmem obliczającym gradient funkcji celu w danym punkcie,

- params_set(const char *string) - wykorzystywana do ustawienia opcji specyficznych dla tego algorytmu. Metoda powinna sekwencyjnie dla każdego pola prywatnego, przechowującego wartość danego parametru, wywołać metodę klasy bazowej params_parse_string(const char *s), przekazując jej argument wejściowy, nazwę symboliczną danej opcji, jej typ oraz adres w pamięci. Jako typu, należy użyć odpowiedniej stałej symbolicznej, zdefiniowanej w pliku **Algorithm.h**. Dodatkowo należy wywołać analogiczną metodę z klasy bazowej. Przykładowo klasa CRS23 definiuje tę metodę w sposób następujący:

```
int params_set(const char *string)
{
    params_parse_string("alfa", NUMTYPE_T, string, &alfa);
    params_parse_string("NP", UNSIGNED_INT_T, string, &NP);
    params_parse_string("crs3", UNSIGNED_CHAR_T, string, &crs3);
    crs3 -= '0';
    return Algorithm::params_set(string);
}
```

- one_iteration(void) - najważniejsza metoda. Powinna definiować JEDEN krok działania metody. Widzimy zatem, iż taka konstrukcja wymusza umieszczenie stanu przebiegu działania metody w polach prywatnych klasy,

- params_get(char *buffer) - metoda powinna do przekazanego bufora zwracać zakodowaną informację o wartości swoich parametrów. Na początku swojego działania powinna wywołać analogiczną metodę z klasy bazowej. Wartości te powinny tworzyć rekordy *name=value* oddzielone średnikami. Omawiania wyżej metoda CRS23 implementuje ją w sposób następujący:

  ```
  void params_get(char *buffer)
  {
      Algorithm::params_get(buffer);
      sprintf(GOOL_ENDP(buffer), "NP=%u;alfa=%f;crs3=%d", NP, alfa, crs3);
  }
  ```

  Dzięki tej metodzie m.in. środowisko GOOL/GUI nie musi już przechowywać wartości opcji metody w tablicach Tcl - każdorazowo zostają one pobierane z jądra systemu.

Wymagane jest również dostarczenie informacji odnośnie typu oraz wartości dopuszczalnych parametrów metody, jak i ich wartości domyślnych. W tym celu należy zdefiniować w pliku implementującym metody nowo tworzonej klasy statyczną tablicę, zawierającą rekordy typu *paramInfoType*, zdefiniowanego w pliku **Algorithm.h**. Następnie należy dokonać rejestracji i jednocześnie inicjalizacji parametrów poprzez sekwencyjne wywołanie metody *params_apply_default()* klasy bazowej. Przykład tablicy z informacją o parametrach oraz sposobu jej wykorzystania do inicjalizacji wartościami domyślnymi wygląda następująco:

```
//=====================================================================
// INFORMACJE O PARAMETRACH:
// nazwa, opis, typ, min, max, default.
//=====================================================================
static paramInfoType records[] = {

    // prawdopodobienstwo generowania punktu w oparciu o metode
    // lokalna
    { "ploc", "P(loc)", NUMTYPE_T, 0, 1, 0.25 },
    // liczba krokow procedury lokalnej
    { "loc_steps", "Local steps", UNSIGNED_INT_T, 0, 10, 3 },
    // poczatkowa dlugosc Lancucha Markowa
    { "L0", "Markow length", UNSIGNED_INT_T, 1, 15, 10 },
```

```
    // liczba losowanych punktow podczas inicjalizacji T
    { "M0", "Init M0", UNSIGNED_INT_T, 10, 100, 50 },
    // wsp. akceptacji prob przy inicjalizacji T
    { "A0", "Init A0", NUMTYPE_T, 0, 1, 0.9 },
    // ktory schemat wyzarzania?
    { "cool", "Cooling variant", CHAR_T, 1, 4, 1 },
    // parametr do wybranego schematu
    { "alpha", "Cooling param", NUMTYPE_T, 0, 100, SA_ALPHA },
    // minimalna T - jesli 0 (domyslnie) - kryterium jest nieaktywne
    { "T_min", "T_min", NUMTYPE_T, 0, 0, 0 }
};
static unsigned int rec_no = sizeof(records) / sizeof(paramInfoType);
//=================================================================
SA::SA(unsigned long max_iter) : Algorithm (max_iter),
                                 AlgUsingLocal(this)
{
    params_apply_default(records, rec_no, "L0", &L_0);
    params_apply_default(records, rec_no, "M0", &M_0);
    params_apply_default(records, rec_no, "A0", &A_0);
    params_apply_default(records, rec_no, "ploc", &p_loc);
    params_apply_default(records, rec_no, "loc_steps", &loc_steps);
    params_apply_default(records, rec_no, "cool", &cool);
    params_apply_default(records, rec_no, "alpha", &alpha);
    params_apply_default(records, rec_no, "T_min", &T_min);
}
```

Dodatkowo metoda może nadpisać następujące metody, zdefiniowane w klasie bazowej jako puste:

- init_specific(void) - inicjalizacja struktur metody tuż po zainicjalizowaniu punktów startowych, a tuż przed uruchomieniem jej przebiegu (metoda *run()*),
- finish_specific(void) - analogiczna metoda, zwalniająca wszystkie struktury specyficzne dla metody po zakończeniu działania,
- specific_stop_criterion(void) - metoda powinna zwrócić wartość dodatnią w przypadku uznania, że któreś ze specyficznych kryteriów stopu zostało spełnione. Wartość ta powinna jednoznacznie identyfikować specyficzne kryterium stopu dla tej metody. Alternatywnym sposobem zatrzymania optymalizacji jest wywołanie w trakcie jej trwania metody *stop_algorithm(char code)*, przekazując jej wartość odpowiadającą kryterium stopu.

Jeśli dodatkowo przewidujemy, że metoda ma komunikować się ze środowiskiem GOOL/GUI, należy wewnątrz jej definicji posługiwać się wywołaniami metod przeznaczonych do komunikacji z tym środowiskiem. Aby algorytm był widoczny przez środowisko okienkowe, należy dodać stosowny opis do pliku **methods.tcl**, opisanego w dalszej części pracy.

6.2 Dołączanie zaimplementowanej metody do interpretera

Dzięki zastosowaniu struktury obiektowej systemu, niepotrzebne są żadne dodatkowe modyfikacje w kodzie systemu GOOL, co jest znakomitym uproszczeniem w porównaniu z wersją VSO2.

6.3 Rekompilacja jądra systemu

Poniżej przedstawiona jest zawartość pliku **makefile**, przeznaczonego do kompilacji systemu GOOL kompilatorem GCC. Należy zwrócić uwagę na alternatywne opcje kompilacji wersji GOOL/CON i wersji GOOL/GUI - ta druga dołącza dodatkowe biblioteki oraz definiuje stałą *GOOL_GUI*.

```
INC   = -I./include/methods -I./include/generators  -I./include \
          -I./include/recipes -I./
LIBS_CON = -L/usr/lib -lm
LIBS_GUI = -L/usr/X11R6/lib -ltcl -ltk -lX11
DEBUG   =
CPPFLAGS = $(DEBUG) $(INC) -O2 -DUNIX -Wall
con custom : LIBS=$(LIBS_CON) -lefence
gui        : LIBS=$(LIBS_CON) $(LIBS_GUI)
gui        : CPPFLAGS=$(CPPFLAGS) -DGOOL_GUI

OBJ1 = Algorithms/Algorithm.o Algorithms/AlgManager.o \
       Algorithms/AlgUsingLocal.o Algorithms/AlgUsingDirection.o \
       Algorithms/AlgorithmDirection.o Algorithms/AlgorithmSet.o \
       Task.o
# obsluga wyrazen definiujacych zadanie
OBJ2 = Function.o expressions/diff.o \
       expressions/symbols.o expressions/trees.o \
       expressions/indexes.o expressions/indexed.o \
       expressions/iterator.o expressions/parser.o \
       expressions/indexer.o expressions/vector_params.o
# klasy pomocnicze zwiazane z algorytmami
OBJ3 = methods/Grid.o methods/clusters.o
OBJ4 = vector_matrix.o utils.o
# generatory:
OBJ5 = generators/Generator.o generators/GenManager.o \
       generators/Gaussian.o generators/Uniform.o \
       generators/Sequence.o generators/Other.o
# konkretne algorytmy:
OBJ6 =  methods/Galperin.o methods/RegularGrid.o \
        methods/ActiveGridTree.o methods/ActiveGrid.o \
        methods/Gourdin.o methods/Meewella.o \
        methods/CRS23.o methods/CRS6.o \
        methods/GA.o methods/GA_binary.o methods/GA_real.o \
        methods/SA.o methods/Torn.o methods/Nelder.o \
        methods/Powell.o \
```

```
        methods/direction/Parabolic.o
# moduly z recipes
OBJ7 = recipes/nrutil.o recipes/jacobi.o recipes/pythag.o recipes/tqli.o \
       recipes/tred2.o recipes/simplx.o recipes/simp1.o recipes/simp2.o \
       recipes/simp3.o
# ponizej funkcje pomostowe pomiedzy C, a C++.
OBJ9 = wish/cmd.o wish/task.o wish/utils.o \
       wish/generators.o
OBJS = $(OBJ1) $(OBJ2) $(OBJ3) $(OBJ4) $(OBJ5) $(OBJ6) $(OBJ7)

CC   = g++

con: $(OBJS) $(OBJ8) main/con_main.o
$(CC) -o gool_con $^ $(LIBS)

gui: $(OBJS) $(OBJ8) $(OBJ9) main/gui_main.o
$(CC) -o gui/gool_wish $^ $(LIBS)

custom : $(OBJS) main.o
$(CC) -o gool_custom $^ $(LIBS)

clean:
rm -f *.o methods/*.o generators/*.o program core program.core \
          recipes/*.o expressions/*.o main/*.o *.exe wish/*.o \
          Algorithms/*.o methods/direction/*.o
```

Zakłada się, iż kod źródłowy metod GOOL/OM znajduje się w katalogu **methods**, zaś pliki nagłówkowe - w katalogu **include/methods**. I tam właśnie należy umieścić stworzone pliki z implementacją oraz deklaracją nowej metody, a następnie dodać nazwę pliku **..cc** do sekcji OBJ6 pliku makefile.

Aby zbudować intepreter rozszerzony języka Tcl/Tk należy wowołać polecenie **make gui**. Utworzony interpreter o nazwie **gool_wish** zostanie umieszczony w katalogu **gui**, zawierającego kod systemu okien dialogowych.

6.4 Rejestracja metody w programie GOOL/GUI

Program GOOL/GUI pobiera informacje o metodach optymalizacji z pliku tekstowego **methods.tcl**. Jego fragment podany był już wcześniej przy opisie uruchamiania algorytmów, teraz przedstawimy dokładniej jego format.

```
Plik methods.tcl
Nazwa   "Opis" (G)lobal lub (L)ocal  (R)andom lub (D)eterministic
```

W pierwszej linii opisu metody podaje się kolejno:

1. nazwę symboliczną odpowiadającą danej metodzie - musi być taka sama, jak łańcuch zwracany przez metodę *get_name()* klasy implementującej tę metodę,
2. nazwę opisową metody - (dowolny ciąg znaków podany w cudzysłowach),

3. informację o tym, czy metoda jest lokalna, czy globalna (potrzebne do właściwego umieszczenia w hierarchii metod),

4. charakter losowości - czy jest deterministyczna, czy losowa,

5. lista parametrów przesyłanych do środowiska GOOL/GUI podczas działania w trybie edukacyjnym.

Poniżej znajduje się fragment pliku opisujący grupę metod dostępnych w bibliotece:

```
#-----------------------------------------------
BFGS     "BFGS"               L D
CRS23    "CRS 2,3"            G R
SA       "Simulated Annealing" G R "T"
GA_REAL  "Evolution Algorithm" G R "fit_min fit_avg fit_max"
GALPERIN "Galperin"           G D "K f_opt F_opt"
```

Format tego pliku uległ uproszczeniu w stosunku do wersji występującej w VSO2 dzięki opisywanemu przeniesieniu obsługi związanej z parametrami metod oraz ich charakterystyką (liczba punktów, na których operuje, wymagania, dotyczące gradientu) do kodu biblioteki w C++.

Ostatnim krokiem jest dopisanie wywołań odpowiednich metod komunikujących się z systemem GOOL/GUI zgodnie z charakterem algorytmu oraz wymaganiami związanymi z wizualizacją jej przebiegu. Opis sposobu komunikowania się z systemem okienkowym zawiera dodatek 5.3 pracy.

Rozdział 7

Wykorzystanie metod z biblioteki GOOL w innych programach

Jeśli użytkownikowi zależy jednocześnie na efektywności obliczeniowej oraz na maksymalnej elastyczności w definiowaniu zadania oraz specyfikowania szczegółowej hierarchii metod i ich parametrów do jego rozwiązania, należy zastosować wariant trzeci. Polega on na stworzeniu własnej funkcji **main** programu, w której za pomocą istniejących bibliotek GOOL/OM oraz GOOL/RG można utworzyć egzemplarze wszystkich pożądanych klas oraz odpowiednio je skonfigurować. Poniżej przedstawimy kilka możliwych scenariuszy wykorzystania bibliotek GOOL/OM oraz GOOL/RG.

7.1 Zadanie budowane ręcznie

```
#include <gool_defs.h>
#include "Task.h"
#include "AlgManager.h"
/*============================================================*/
int main(int argc, char **argv)
{
Task        task;
CRS6        crs6;
int         error = 0;
goolVector   result;
numType     val;

    error = task.def_symbol("cost", "[ 10, 20, 5 ]");
    task.set_objective("sum_i(1:10)(x_i - cost_i)");
    task.add_constraint("c1", "x1 - x2 - 2");
    task.add_constraint("c2", "x1 - x3 - 2");
    task.add_constraint("c3", "x4 * x5 - x6 < 7");
    task.set_method(&crs6);
    crs6.set_params("max_iter=100;max_failure=15;NP=15");
    result = newGoolVector(10);
    task.run(result, &fval);
    freeGoolVector(result);
    return 0;
}
```

7.2 Zadanie wczytane z pliku

```
Task        task;
CRS6        crs6;
int         error = 0;
goolVector   result;
numType     val;

    error = task.parse_file("data/task1.tsk");
    crs6.set_params("max_iter=100;max_failure=15;NP=15");
    result = newGoolVector(10);
    task.run(result, &fval);
    freeGoolVector(result);
    return 0;
```

Powyższe pliki należy skompilować i zlinkować, wykorzystując plik **makefile** przedstawiony w dodatku 6.3. Plikowi głównemu należy nadać nazwę **main.c**, oraz użyć polecenia **make custom** .

Dodatek A

Biblioteka obługi wyrażeń symbolicznych

W bibliotece GOOL przyjęto, iż jedną z postaci definiowania zadania optymalizacji jest wprowadzenie postaci analitycznej. Wymagało to stworzenia zbioru funkcji, pozwalających na łatwe definiowanie zarówno postaci funkcji celu, jak i związanych z zadaniem ograniczeń. Dla ułatwienia zarządzania i modyfikacji zadania konieczne stało się umożliwienie definiowania parametrów skalarnych, wektorowych oraz macierzowych zadania, zdefiniowanie zbioru pierwotnych funkcji i operatorów, pozwalających wprowadzanie skomplikowanych wyrażeń. W świetle rozpatrywanego w pracy praktycznego zadania optymalizacji niezbędne okazało się również umożliwienie definiowania funkcji iterowanych (wyrażeń typu „suma" oraz „iloczyn"), jak również zadawanie ograniczeń zadania jako wyrażeń indeksowanych.

A.1 Opis ogólny

Podsystem specyfikuje interfejs składający się ze zbioru następujących funkcji:

- **symbols_init(void)**
  inicjalizacja podsystemu, ta funkcja musi być wywołana przed pierwszym użyciem którejkolwiek z następujących dalej funkcji,
- **symbols_free(void)**
  końcowe czyszczenie struktur danych,
- **symbol_define(const char *, const char *, int *)**
  definicja nowego symbolu o zadanej nazwie,
- **symbol_redefine(const char *, const char *, int *)**
  redefinicja istniejącego symbolu lub jego części (np. składnik wyrażenia indeksowanego),
- **symbol_get_value(const char *, goolVector, int *)**
  pobranie wartości symbolu o podanej nazwie i zadanego argumentu. Nazwa w ogólnym przypadku może być wzorem całego wyrażenia - zostanie ono odpowiednio zinterpretowane i obliczone,
- **symbol_get_indexed_info(const char *, unsigned int *, unsigned int *)**
  za pomocą tej funkcji możemy pobrać informację o zakresach indeksów wyrażenia

indeksowanego. W obecnej wersji w przypadku wyrażenia indeksowanego możemy pobrać wartość tylko dla wybranego indeksu - tutaj otrzymamy ich zakres,

- **expr_node_get_value(exprNodeType *, goolVector, int *)**
  pobranie wartości poddrzewa o zadanym korzeniu (jest to metoda szybsza od obliczania wartości na podstawie nazwy),

Możliwości podsystemu najłatwiej prześledzić analizując przykłady jego użycia, nawiązując do rozważanego zadania optymalizacji cen. Model mieszany [1], którego postać została przedstawiona w pracy [9], jest reprezentowany następujący sposób (w przykładzie zastosowano model dla zadania o wymiarze $n = 4$):

```
int e = 0;
symbol_define("a", "[ 200, 180, 210, 215 ] ", &e);
symbol_define("c", " [ -2, 0.4, -3, 0.2 ]", &e)
symbol_define("cost", " [ 10,  8, 7, 11 ]", &e)
symbol_define("beta", " [ -0.05, 0.04, 0.03, 0.02;
                          0.02, -0.4, 0.09, 0.8 ;
                          0.07,  0.3, -0.54, 0.1 ;
                          0.08,  0.2, 0.09, -0.14 ] ", &e);
symbol_define("xsr", [ 50, 45, 57, 63 ]", &e )
symbol_define("xob", [ 45, 47, 81, 55 ]", &e )
symbol_define("S_i(1:4)", "a_i+alfa_i*mul_j(1:4)(pow(x_j,beta_i_j) +...", &e);
symbol_define("objective", "sum_i(1:4)(cost_i - x_i) * S_i", &e)
```

Jak widać, wszelkie wielkości skalarne i wektorowe zostały zdefiniowane parametrycznie, co umożliwia ich prostą modyfikację, bez naruszania struktury funkcji celu oraz ograniczeń. Wyrażenia wymagające sekwencyjnego sumowania lub przemnażania składników zostały w sposób zwarty zapisane za pomocą *iteratorów* , zaś rodziny ograniczeń – przy wykorzystaniu *wyrażeń indeksowanych* .

UWAGA: W kontekście systemu GOOL nie ma potrzeby wpisywania za każdym razem powyższych sekwencji w kodzie programu w celu zdefiniowania zadania. System umożliwia wygodny sposób definicji przy wykorzystaniu odpowiednio sformatowanych plików tekstowych (punkt 3.2).

Jeśli w danej chwili interesują nas wartości poszczególnych wyrażeń lub ich części, możemy je pobrać w następujący sposób:

```
int err = 0;
numType x[2] = {10, 20};
exprNodeType node = symbol_define("symbol", "x_1 + x_2", &err);
numType val = symbol_get_value("symbol", x, &err);
albo: (pobranie przez adres wezla - szybsze)
numType val = symbol_get_value(node, x, &err);
```

Ideę działania podsystemu opisuje następujący ciąg poleceń:

1. rozbicie wyrażenia na elementy składowe, uwzględnienie podczas analizy elementów podstawowych, indeksatorów, parametrów wektorowych jak i symboli zdefiniowanych już wcześniej,

[1] tj. wynikający z połączenia funkcji Cobba-Douglasa oraz Gutenberga

2. zbudowanie odpowiadających tokenom struktur danych i utworzenie drzewa binarnego, reprezentującego dane wyrażenie,

3. podpięcie drzewa do tablicy symboli pod przekazaną nazwę.

Obliczenie wartości danego wyrażenia polega na odpowiednim skojarzeniu z odpowiadającym mu drzewem wyrażenia (być może - zbudowanie takiego drzewa), a następnie wywołanie rekurencyjnej funkcji obliczającej wartość wyrażenia w korzeniu drzewa.

Należy zauważyć, że ze względu na możliwe wykorzystywanie elementów, bądź całych wyrażeń, przez inne wyrażenia, mamy tu do czynienia z wzajemnymi odwołaniami tworzonych struktur drzewiastych. Powoduje to konieczność uważnego zarządzania zwalnianiem z pamięci struktur tak, by mieć na uwadze potencjalne wskazania na zwalniane wyrażenie z innych, aktualnie zdefiniowanych symboli. W tym celu zastosowano mechanizm używany wewnątrz większości języków wyższego poziomu [2] związanego z odzyskiwaniem nieużywanej już pamięci [3], opartego na liczniku odwołań (referencji) do danej struktury. Ciekawostką może być fakt, iż systemy te **nie działają** dobrze w momencie obsługi dynamicznych struktur cyklicznych - cykliczność referencji powoduje, iż odpowiednie liczniki mogą nigdy nie osiągnąć zera. Jednak w przypadku, który tu rozważamy, problem ten praktycznie nie występuje.

A.2 Elementy składowe wyrażeń

Dopuszczalne elementy wyrażeń można podzielić na następujące grupy:

- elementy podstawowe
  Do tego zbioru należą stałe liczbowe, indeksy jawne zmiennej niezależnej, operatory: $($, $*$, $-$, $/$, $+$,$\hat{}$), funkcje: exp(x), sin(x), cos(x), pow(x,y), tg(x), ctg(x), sinh(x), cosh(x), min(x,y), max(x,y), sqrt(x). Przykładami indeksów jawnych zmiennej niezależnej są wyrażenia w postaci: x_1, x_10 (indeks jest zadany jako stała liczbowa). Stałe liczbowe to w ogólnym przypadku liczby zmiennoprzecinkowe ze znakiem, również w notacji 'e' (np. 10e+7, 10E-5). Elementy podstawowe to także symbole zdefiniowane w krokach wcześniejszych (tzw. aliasy). Przykłady wyrażeń opartych wyłącznie na elementach podstawowych:

  ```
  ALFA = 20 + tg(x_2)
  k = 10e-7 - 3.5456
  pow(sin(x_1) + cos(x_2), x_3) - 10 * ALFA
  x_2^x_1 + x_4 - max(x_3,10 * sinh(x_3-2))
  ```

- funkcje iterowane
  Zdefiniowano funkcję iterowanej sumy oraz iloczynu o następującej postaci ogólnej:

  ```
  mul_<idx1>(min1:max1)_<idx2><min2:max2>...(expr)
  sum_<idx1>(min1:max1)_<idx2><min2:max2>...(expr)
  ```

  idx1, ... są jednoliterowymi symbolami indeksów, z których można korzystać wewnątrz wyrażenia *expr*. Wartościami zakresów powinny być liczby całkowite ze znakiem. Przykłady poprawnych wyrażeń iterowanych:

[2] Jako przykłady wymienić tu można język Java oraz język skryptowy Perl
[3] ang. Garbage Collection

```
sum_i(1:10)(x_i + 5)
mul_k(10:200)(sin(x_k) * cos(x_k))
sum_k(1:10)(mul_j(2:5)(x_k + alfa_j))
```

Obecna wersja systemu pozwala korzystać tylko z iteratorów o jednym indeksie. Wyrażenia iterowane mogą być zagnieżdżane na dowolną głębokość.

- definicje parametrów wektorowych
  Są to wyrażenia pozwalające na definiowanie wektorów oraz macierzy. Elementami składowymi mogą być wyłącznie stałe liczbowe. W przypadku wektorów kolejne elementy powinny być oddzielone od siebie przecinkami. Dla macierzy, separatorami kolejnych wierszy jest znak średnika.

  ```
  cost = [ 10, 20, 5, 4 ]
  beta = [ 10, 20; -5.4, 10e-1 ]
  ```

  Definicja parametru wektorowego musi się pojawić po prawej stronie przypisania i nie może być częścią bardziej złożonego wyrażenia.

- wyrażenia indeksowane
  Wyrażenia indeksowane stanowią rozszerzenie parametrów wektorowych w tym sensie, że ich składnikami mogą być nie tylko stałe, ale dowolne poprawnie zbudowane wyrażenia. Składnia odpowiada tej, która jest stosowana dla iteratorów:

  ```
  symbol_<idx1>(min1:max1)_<idx2><min2:max2>... = expr
  ```

  Obecna wersja systemu pozwala korzystać tylko z wyrażeń indeksowanym operujących na jednym indeksie. Definicja wyrażenia indeksowanego może pojawić się tylko po lewej stronie przypisania, a zatem niemożliwe jest ich zagnieżdżanie. Można natomiast odwoływać się do elementów innych wyrażeń poprzez odpowiednie zastosowanie indeksatorów,

- indeksatory
  pozwalają na odwoływanie się do elementów składowych parametrów wektorowych, wyrażeń indeksowanych oraz zmiennej niezależnej *x*. Symbole indeksów analogicznie jak w przypadkach opisanych powyżej powinny być jednoliterowe. Dodatkowo możliwe jest nadawanie wartości ustalonych niektórym (lub wszystkim) indeksom dla danego indeksatora.

  Przyjęto konwencję, że minimalną wartością indeksu jest 1. W przypadku macierzy zastosowano adresowanie zgodne z przyjętym standardem (wiersz,kolumna):

  ```
  cost = [ 10, 20, 5, 4 ]
  beta = [ 10, 20, 5; -5.4, 10e-1, 19 ]
  cost_1 = 10
  cost_2 = 20
  beta_2_1 = -5.4
  beta_1_3 = 5
  beta_1_4 = ERROR
  ```

  Indeksatory mogą odnosić się tylko do zmiennej niezależnej, wyrażeń indeksowanych oraz parametrów wektorowych.

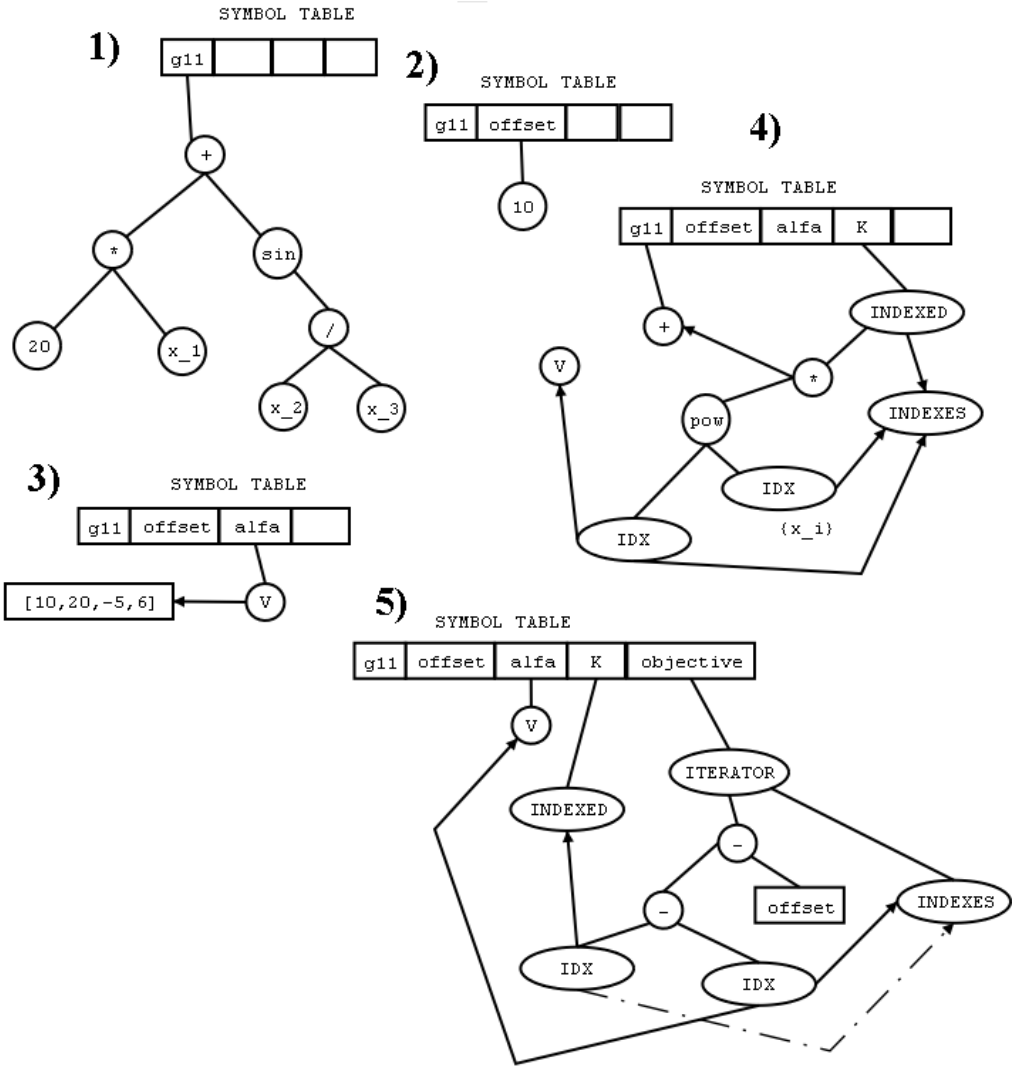
  Przykłady użycia indeksatorów:

```
beta = [ 10, 20, 5, 7, 8; -5.4, 10e-1, 19, 87, 34 ]
# indeksator zmiennej niezaleznej oraz parametru wektorowego:
expr1_i(1:5) = x_i * beta_2_i
# indeksator wyrazenia indeksowanego:
expr2 = x_i + expr1_2
```

UWAGA: indeksatory zmiennej niezależnej o ustalonej wartości indeksu (zmienna niezależna jest jednowymiarowa) są reprezentowane w podsystemie jako elementy podstawowe.

A.3 Przykłady tworzonych struktur

Aby dokładnie zrozumieć specyfikę działania systemu wraz ze wszelkimi ograniczeniami z niej wynikających, prześledźmy, w jaki sposób system obsługuje tworzenie przykładowych wyrażeń.



Rysunek A.1: Kolejne kroki tworzenia opisywanych wyrażeń.

```
int err = 0;
```

```
exprNodeType node;

node = symbol_define("g11", "20 * x_1 + sin(x_2 / x_3)", &err);  /* 1 */
node = symbol_define("offset", "10", &err);                      /* 2 */
node = symbol_define("alfa", "[ 10, 20, -5, 6 ], &err");         /* 3 */
node = symbol_define("K_i(1:4)", "pow(alfa_i,x_i) * g11,
                      &err");                                    /* 4 */
node = symbol_define("objective",
                   "sum_j(1:4)(K_j - alfa_j - offset)", &err);  /* 5 */
```

Na diagramach kółkami oznaczono struktury typu *exprNodeType*. W kolejnych krokach dla poprawienia czytelności usunięto te symbole, do których nie ma bezpośrednich odwołań nowo tworzonych tokenów.

W ten sposób możemy nawet bardzo skomplikowane wyrażenia budować sukcesywnie ze składowych podwyrażeń, jednocześnie mając możliwość wielokrotnego wykorzystywania symboli już zdefiniowanych. Zapobiega to zbędnemu rozrostowi struktur danych w pamięci, co ma duże znaczenie w przypadku wykorzystania systemu w programie GOOL.

A.4 Najistotniejsze problemy

Głównym problemem, jaki musiano rozwiązać, była odpowiednia obsługa indeksów definiowanych przez funkcje iterowane oraz wyrażenia indeksowane i odpowiednie ich powiązanie z wykorzystującymi je indeksatorami. Trudności, które się pojawiały, można przedstawić na przykładzie następującej sekwencji wyrażeń:

$$\begin{aligned} s_{i,k} &= \alpha_k \cdot \prod_{j=1}^{5} \beta_{k,j,i} \\ w_i &= s_{2,i} \end{aligned}$$

Zauważmy, iż wartość wyrażenia indeksowanego reprezentowanego przez token $s_{i,j}$ (w ogólnym przypadku - nie możemy kojarzyć na sztywno symboli *i* oraz *k* z tokenem *s*) zależy tu od kontekstu, w jakim został użyty. Indeksator w_i narzuca wartość ustaloną (równą 2) pierwszemu indeksowi wyrażenia $s_{i,j}$ i z tej wartości powinny poprawnie korzystać indeksatory definiujące wyrażenia na głębszym poziomie (czyli w tym przypadku indeksator symbolu β. Jednocześnie widzimy, iż ani wyrażenia indeksowane, ani indeksatory nie mogą operować na symbolach indeksów (w sensie kodów ASCII) w celu pobierania ich aktualnych wartości, bowiem te mogą się zmieniać w zależności od kontekstu, w jakim są używane.

Aby rozwiązać te problemy, zastosowano powiązanie pomiędzy wskaźnikami indeksów definiowanych przez wyrażenia indeksowane oraz iteratory (są to jedyne tokeny definiujące symbole lub nadpisujące - tylko iteratory - symbole już istniejące w ich kontekście), a korzystającymi z nich indeksatorami. Każdy indeksator posiada tablicę wskaźników „wejściowych”, wskazujących na kolejne adresy indeksów odpowiadających symbolom w momencie parsowania wyrażenia. Powiązanie symboli znakowych indeksów z adresami zapewniają wspomniane wyżej tokeny definiujące symbole w bieżącym kontekście. Dla wartości ustalonych odpowiednie wskaźniki są ustawiane na wartość NULL. Ponadto, aby poprawnie aktualizować wartości indeksów, z których korzystają podwyrażenia indeksowane przez dany indeksator, posiada on również tablicę wskaźników „wyjściowych”, w

które wpisuje wartości na podstawie tablicy „wejściowej” oraz własnej informacji o wartościach ustalonych. W ten sposób zapewniamy prawidłowe dowiązanie indeksów indeksatorów wraz z poprawną obsługą przesłaniania odpowiednich indeksów przez wartości ustalone i ich uwzględnianie w podwyrażeniach.

A.5 Obliczanie pochodnych wyrażeń

Oprócz definiowania, redefiniowania, usuwania i obliczania wartości odpowiednich wyrażeń, podsystem pozwala również na automatyczne różniczkowanie funkcji celu. Jednak ze względu na trudności, które pojawiają się dla tej operacji w przypadku stosowania funkcji iterowanych oraz wyrażeń indeksowanych, biblioteka nie umożliwia obliczania gradientu funkcji, w których pojawiają się ww. składniki.

A.6 Rozbudowa podsystemu

Oczywiście możliwe jest definiowanie nowych tokenów (w szczególności - brakujących funkcji). W tym celu należy dodać odpowiednie wpisy symboli do tablicy statycznej zdefiniowanej w pliku **symbols.c**, zdefiniować odpowiadające im funkcje dwuargumentowe oraz umieścić w tablicy wskaźniki do nich. W przypadku, gdy jakaś funkcja jest już zaimplementowana bibliotece standardowej (tak, jak ma to miejsce w przypadku funkcji *pow*), wystarczy w tablicy umieścić sam wskaźnik do niej, bez dodatkowej definicji.

Rozszerzenie funkcjonalności na iteratory oraz wyrażenia indeksowane operujące na większej liczbie indeksów wiązałoby się z modyfikacją sposobu reprezentacji struktur reprezentujących te tokeny, jednak sam mechanizm tworzenia struktur oraz obliczania ich wartości nie wymagałby zmian.

Literatura

[1] Arabas J., *Wykłady z algorytmów ewolucyjnych.* WNT, Warszawa. 2001.

[2] Aramini M. J., *Implementation of an improved contour plotting algorithm.* Stevens Institute of Technology, 1980.

[3] Findeisen, W., J. Szymanowski, A. Wierzbicki, *Teoria i metody obliczeniowe optymalizacji.* PWN, Warszawa. 1980.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 2001.

[5] Nelder, J. A., R. Mead, *A simplex method for function minimization.* Computer Press. 1992.

[6] Press, W. H. , S. A. Tukolsky, W. T. Vetterling, B. P. Flannery, *Numerical recipes in C. The Art of Scientific Computing.* Cambridge University Press. 1992.

[7] Publicewicz, M., Śliwiński, R., Niewiadomska-Szynkiewicz, E.,Bolek, P., *VSO - Środowisko graficzne do badania algorytmów optymalizacji. Wersja 2.0*, Raport IAiIS, No. 00-13, 2000.

[8] Publicewicz, M., Niewiadomska-Szynkiewicz, E., *Metody optymalizacji globalnej w środowisku graficznym - Praca dyplomowa*, Instytut IAiIS PW, Warszawa marzec, 2001.

[9] Publicewicz, M., Niewiadomska-Szynkiewicz, E., *GOOL - biblioteka metod optymalizacji globalnej - Praca magisterska*, Instytut IAiIS PW, Warszawa marzec, 2003.

[10] Snyder, W. V., *Algorithm 531, Contour plotting*, ACM Transactions on Mathematical Software **4, 3** s. 290-294 (1978).

[11] Stachurski, A., Wierzbicki, A., *Podstawy optymalizacji*, Oficyna Wydawnicza PW, Warszawa, 1999.

[12] Śliwiński, R., *Środowisko graficzne do badania algorytmów optymalizacji - Praca magisterska*, Instytut IAiIS PW, Warszawa styczeń, 1999.

[13] Törn, A., Zilinskas, A., *Global Optimization*, Springer, 1989.

[14] Welch, B., *Practical programming in Tcl and Tk. Updated for Tcl 7.4 and Tk 4.0.* Prentice Hall.DRAFT January 13, 1995.