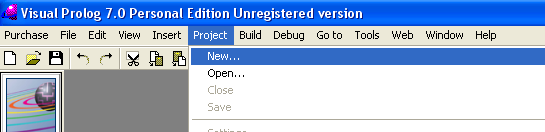
***СЪЗДАВАНЕ НА ИНТЕРАКТИВЕН ИНТЕРФЕЙС***

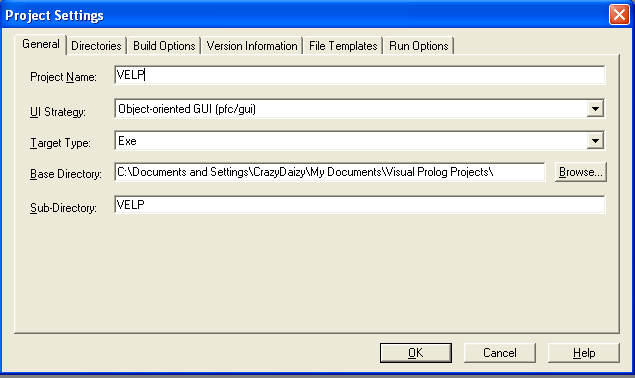
***ВЪВ VISUAL PROLOG***

***1.2. Създаване на проект във VISUAL PROLOG***

За да се създаде нов проект се избира от лентата с менюта опцията Project/New(фиг. 1.1). След това се попълва диалоговия прозорец Project Settings(фиг. 1.2 ).



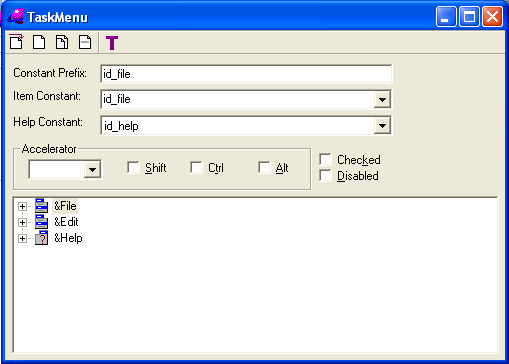
Фиг. 1.1: Лента с менюта: Нов Проект/New Project



Фиг 1.2. Project Settings

**1.2.1. Активиране на лентата с менюта на новият проект**

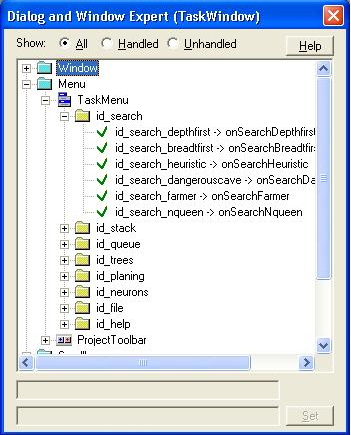
Избира се TaskMenu.mnu от дървовидната структура на създадения проект и се променя вида, по който изглежда лентата с менюта при стартиране на проекта. Това става чрез диалоговия прозорец, който се отваря при кликване два пъти с левия бутон на мишката върху TaskMenu.mnu - Task Menu (фиг.1.3).



фиг.1.3: Диалогов прозорец за оформяне на лентата с менюта на приложението

**1.2.2. Добавяне на код, чрез CodeExpert в дървовидната структура на проекта**

За добавяне на код във всички команди от лентата с менюта, които се използват, се клика с десен бутон на мишката върху файла TaskWindow.win от дървовидната структура на проекта. Избира се опцията Code Expert. Появава се диалоговия прозорец Dialog and Window Expert (фиг 1.4).



фиг.1.4: Dialog and Window Expert

В прозореца се избира папка Menu -> TaskMenu и се клика два пъти с левия бутон на мишката, например върху разклонението id\_search\_depthfirst->onSearchDepthfirst . Това отваря файла от проекта TaskWindow.pro, където се добавя кодът за съответната опция от менюто. Това става по следния начин:

На всяка опция от менюто отговаря предикат от файла TaskWindow.pro. Първоначално се създава автоматично стандартната декларация на този предикат. Това става чрез кликане два пъти с мишката върху id\_search\_depthfirst->onSearchDepthfirst или друга опция от менюто, за която искаме да добавим код и да активираме. Другият начин е чрез маркиране на опцията от менюто и натискане на бутона „Add” в долната дясна част на прозореца Dialod and Window Expert . Така стандартния код за предиката се добавя автоматично във файла TaskWindow.pro. След това, в зависимост от това, какво извършва опцията от менюто, се добавя необходимото в кода на предиката й.

Например в програмната среда PETA при избор от лентата с менюта на Search -> DepthFirst се появява диалоговия прозорец depth. Това става като към предиката onSearchDepthfirst в TaskWindow.pro добавяме следния код:

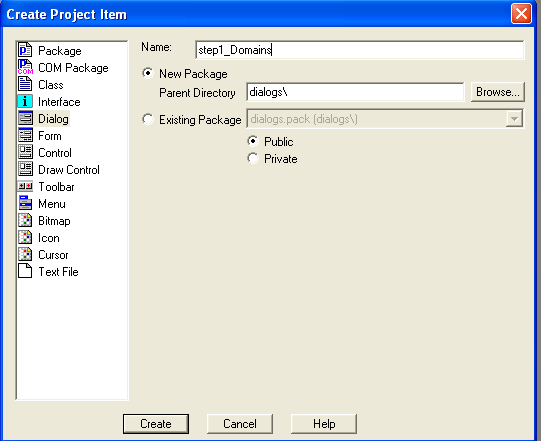
 onSearchDepthfirst(Source, \_MenuTag):- X=depth::new(Source),X:show().

**1.2.3. Създаване на диалогови прозорци във VISUAL PROLOG**

**1.2.3.1. Добавяне на нов диалогов прозорец**

Има две възможности за добавяне на нов диалогов прозорец към проекта: да се добави към нов пакет или към съществуващия основен пакет на проекта.

В зависимост от това се избира съответно: File/ New in New Package или File/New in Existing Package. След този избор се отваря диалоговия прозорец Create Project Item (фиг. 1.5). От левият панел на диалоговия прозорец се избира Dialog, а след това в полето Name се дава име на диалоговия прозорец. Ако е в нов пакет, името на пакета се въвежда в полето Parent Directory. Накрая се натиска бутона Create и диалоговия прозорец е създаден.

Фиг 1.5: Create Project Item

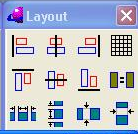
**1.2.3.2. Активиране на контролите на диалоговия прозорец**

След създаване на диалоговия прозорец се отваря редактор, който се състои от три диалогови прозореца: Controls, Layout, Properties.

В Controls (фиг.1.6) са контролите които могат да бъдат използвани в диалоговия прозорец. Layout (фиг.1.7) се използва за подравняване и позициониране на контролите в диалоговия прозорец. А прозорецът Properties (фиг. 1.8) съдържа основните характеристики на диалоговия прозорец (полето Properties), както и неговата активност(полето Events). Когато е селектирана конкретна контрола от прозореца, в прозореца Properties се показват нейните характеристики и нейната активност.



Фиг. 1.7: Controls



Фиг.1.8: Layout



Фиг. 1.9: Properties

Активирането на контролите от диалоговия прозорец става чрез прозореца Properties. Първо се селектира контролата,на която искаме да добавим функционлност. Характеристиките й се появяват в полето Properties на прозореца Properties. Някои от по – важните от тях са:

- Name – име на контролата. С това име тя присъства и в последствие в програмния код.

- Text – това е текстът, който се изписва върху контролата.

- Width – широчина на контролата.

- Height – височина на контролата.

-X; -Y – координати на местоположението на контролата в диалоговия прозорец по X и Y.

Добавянето на функционалност на контролата става чрез полето Event на прозореца Properties.

Например в PETA, в диалоговия порзорец за търсене в дълбочина слагаме статичен текст, показващ предназначението на прозореца и бутон, който преименуваме „Start Depth-first”. За да бъде стартирано търсенето в дълбочина при натискане на бутона, правим следното: Избираме бутона „Start Depth-first” и в диалоговия прозорец Properties избраме полето Event. След това на първия ред ClickResponder избраме OnPushButtonClick и когато шракнем двукратно върху него се отвори файла depth.pro. В него поставяме необходимия код, реализиращ търсене в дълбочина , а към предиката OnPushButtonClick добавяме следния код:

  onPushButtonClick(\_Source) = button::defaultAction:- if dSearch([[r(0,"b")]],L) then prtSolution(L)  
              else stdio::write("No Solution!"), stdio::nl end if.

След това запаметявам файла depth.pro.

По сходен начин е реализирана и функционалността на останалите диалогови прозорци.

***2.3. Създаване на класове във VISUAL PROLOG 7.1***

Създаването на клас става отново от меню File. В зависимост от това дали искаме класа да е в нов, собствен пакет, или да е към вече съществуващ, се избира съответно File/New In New Package или File/ New In Existing Package.

Появява се диалоговия прозорец Create Project Item (вж. фиг. 1.5). От левия панел на прозореца този път се избира Class. Въвежда се име на класа и ако е в нов пакет се въвежда неговото име.

Когато се създава клас присъства и поле creates object. Това поле определя дали класа създава свои обекти или не. Ако полето е избрано в програмния код трябва да присъства предикат new(), който да създава обектите на класа. Това е всъщност конструктора на класа. Не е задължително той да е с име new(), може да се сложи и друго име. Но този предикат трябва задължително да е от тип procedure.

След като се изберат необходимите характеристики на класа, се натиска бутон create и той се създава. Класът съдържа три файла:

* <className>.cl – това е декларацията на класа.
* <className>.i – интерфейс на класа.
* <className>.pro – изпълнение на класа.

Интерфейса на класа съдържа декларациите, които са public (може да се извикват от други файлове) и са свързани с обектите, генерирани от този клас. Интерфейса на класа е всъщност описание на това, което външния свят може да види от обектите на класа. Може да се мисли за интерфейса и като за декларация на вида на обектите, които генерира този клас. Казва се, че интерфейса определя конкретен тип обект. Когато класа не може да създава обекти (това е решение на програмиста), файла с интерфейса на класа се пропуска, не е необходим и кода на класа се разпределя в два файла.

Файлът с декларацията на класа (.cl) съдържа дакларации на предикати, които също са public, но не са свързани с обектите, а с класа като цяло. Този файл може да съдържа дефиниции на константи и променливи и декларации на предикати. Другите части на програмата могат да видят и използват точно това, което е споменато в декларацията на класа. Казва се, че декларацията на класа определя публичната част на класа.

В третият файл (.pro) се съдържат дефиниции на предикати, които са всъщност клаузи на предикати и факти, които са декларирани в декларацията на класа или в интерфейса. Изпълнението на класа се използва, за да представи действието на предикатите и конструкторите, декларирани в декларацията на класа.

В зависимост от това, дали даден предикат определя обектите на класа, или се отнася за класа като цяло, се използват два вида предикати: „predicates” и „class predicates”. Подобен е принципа и при фактите. В зависимост от това, дали факта определя обектите на класа или класа като цяло, се използват: „facts” и „class facts”.

В програмната срада PETA са въведени класове за операциите със стек (stackprocs), операциите с опашка (classQ), обхождането на дървета (class Tree), за невронната мрежа (network) и др. (вж. Фиг. 1.10).

Класът за стекови операции „stackprocs” реализираме по следния начин: Маркираме папката TaskWindow (основната папка на проекта) и след това избраме от меню File -> New In Existing Package. Така класът става част от основната папка на проекта. След това в появилия се диалогов прозорец Create Project Item избраме Class, въвеждаме име на класа „stackprocs” и маркираме опцията create object. За да бъде създаден класа натискаме бутона Create.

По този начин се създава клас с три файла. Във файла „stackprocs.cl” добавяме следния код :

  predicates  
    push : (string Element) procedure (i).  
    peek : (string Element) determ (o).  
    pop : (string Element) determ (o).  
    dup : () determ.  
    swap : () determ.  
    showstack : () .

Това са декларациите на публичните предикати на класа. След въвеждането запаметяваме файла.

Във файла „stackprocs.pro” добавяме кода:

class facts  
    stacklist : stringlist :=["ddd", "ccc", "bbb", "aaa"].  
clauses  
    push(Element) :-  
        stacklist := [Element | stacklist].       
  
    peek(Element) :-  
        [Element | \_ ] = stacklist.        
  
    pop(Element) :-  
        [Element | Tail] = stacklist,      
        stacklist := Tail.             
   
    dup() :-  
        [Element | \_] = stacklist,             
        stacklist := [Element | stacklist].        
      
    swap() :-  
        [Element1, Element2 | Tail] = stacklist,       
        stacklist := [Element2, Element1 | Tail].       
      
    showstack() :-  
        stdIO::write("The stack contains: "),  
        stdIO::nl,  
        stdIO::write(stacklist).

След което запаметяваме файла и с това завършваме създаването на класа.

Другите използвани класове са създадени на същия принцип.

„Stackprocs” и „classQ” са в TaskWindow, а останалите са в отделни пакети от проекта. В зависимост от предназначението си някои класове имат интерфейс (създават обекти), а други – нямат. При класовете, които са в самостоятелни пакети, а не в основния пакет на проекта, трябва да се спазват правилата за публичните предикати. От останалите части на програмата могат да се използват само предикатите от декларацията на класа (предикатите, съдържащи се във файла с разширение .cl). Обектите могат да се използват чрез декларациите от интерфейса на класа (файлът с разширение .i).

При създаването на нов дилогов прозорец, активирането на контролите става чрез добавяне на код за предиката на контролата във файл. Този файл е с името на диалоговия прозорец и разширение „.pro”. Това е всъщност един от трите файла на класа, който се създава към диалоговия прозорец. При създаването на прозореца автоматично се създава и класа към него, който се състои от три файла. Така клаузите на предикатите, активиращи контролите, се намират във файла с разширение „.pro”. По този начин всеки диалогов прозорец има клас, който поддържа неговата функционалност.

Класовете, които са използвани за реализацията на програмната система PETA, са представени на следващите две диаграми. На първата диаграма (Фиг.1.10) са класовете за работа със стек, опашка, бинарни дървета и класът, реализиращ невронната мрежа. На втората диграма (Фиг.1.11) са класовете, участващи в генетичния алгоритъм.



Фиг 1.10: Класове, реализирани в програмната среда PETA



Фиг.1.11: Класове на генетичния алгоритъм

***Описание на работата с програмна среда за тестване на алгоритми PETA и резултати от експерименти, проведени с нея***

***2.1. Информация за PETA***

PETA (Programming Environment for Testing Algorithms) е програмна среда за тестване на алгоритми, която е създадена в помощ на хората, запознаващи се с алгоритмите и методите в Изкуствения интелект. Реализацията на алгоритмите е на езика за логическо програмиране – Visual Prolog.

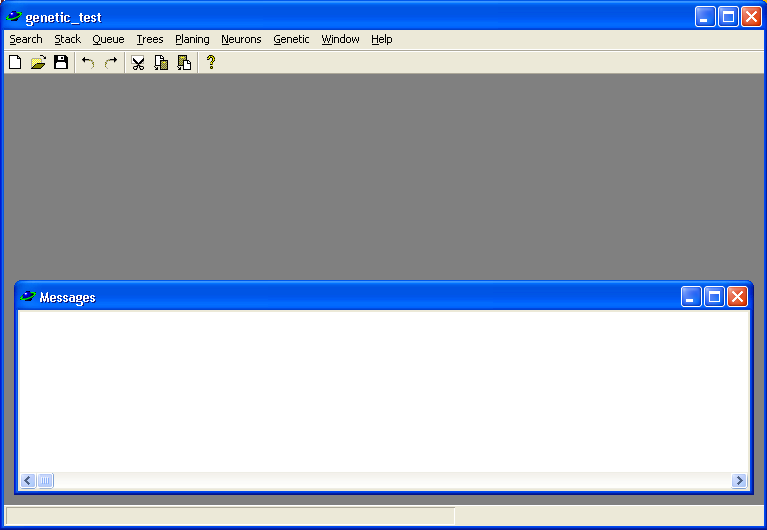
За създаването на програмната среда е използван изцяло Visual Prolog 7.1.

Средата PETA е съставена от основно меню, от което могат да се избират отделните алгоритми:

* Меню Search – В него са заложени алгоритмите, реализиращи различни видове търсене.
* Меню Stack – В него е реализирана структурата от данни стек, както и някои основни операции с нея.
* Меню Queue - В него е реализирана структурата от данни опашка и някои основни операции с нея.
* Меню Trees – Реализирани са различни видове обхождане на заложено в програмата дърво.
* Меню Planing – Съдържащо планиращ алгоритъм: Задачата „Ханойски кули”.
* Меню Neurons – Съдържащо реализация на невронна мрежа.
* Меню Genetic – Съдържащо реализация на генетичен алгоритъм.

***2.2. Стартиране на PETA***

Преди стартирането на PETA не е необходимо да има инсталирана версия на Visual Prolog. Достатъчно е да се стартира файла PETA.exe .

***3.3. Общи принципи за работа с PETA***

**3.3.1. Общ вид на екрана**

След стартиране на PETA се отваря прозорец, който е стандартен за Windows. Той изглежда по следния начин:

Фиг 3.12: Общ вида на диалоговия прозорец на PETA

В най-горния ред на екрана се намира заглавната лента, в лявата част на която е изписано името на стартираната програма. В дясната част на лентата има три бутона, чието предназначение е следното:

* Минимизира(скрива) прозореца на програмата – това може да се наложи, ако искате да отворите друга програма без да прекратявате работата с PETA. След тази операция PETA остава активна, като на най-долната екранна лента се появява нейния бутон. Когато решите да активирате програмата, просто трябва да кликнете върху него;
* Увеличава размера на прозореца на PETA, който при стартирането на програмата заема само част от екрана; при това се появява бутонът restore down, като неговото натискане води отново до минимизиране на прозореца;
* Затваря прозореца на екрана, като прекратява неговото действие;

**2.3.2. Диалогови прозорци**

Това са графични прозорци, в които се извършва конкретна дейност. Диалоговите прозорци, които са използвани в програмната среда PETA съдържат бутони, полета и статичен текст.

- Бутони (buttons) – върху бутона има надпис, определящ действието, което може да се извърши след натискането му. В програмната среда PETA основно диалоговите прозорци са с един бутон, стартиращ конкретния алгоритъм, който е избран от менюто. Посредством бутона close в горния десен ъгъл на всеки прозорец, се прекратява работа и прозорецът се затваря, без запомняне на промените.

- Поле (Edit field) – всяко поле се характеризира с име и съдържание. Това е място, в което се въвежда и съхранява информация. В тези полета се въвеждат стойности, необходими за работата на конкретния алгоритъм.

- Статичен текст – текст разположен в диалоговите прозорци, служещ за помощ, пояснение или насочване. Чрез статичен текст в PETA е описано действието и предназначението на всеки алгоритъм.

**2.3.3. Менюта**

Името на всяко меню представлява кратък текст, който обобщава набора от прилежащите му алгоритми. Тези текстове са разположени в лента в горния ляв край на основния програмен прозорец. При кликване върху дадено меню то „пада”, т.е. появяват се алгоритмите, които могат да бъдат избрани. Изборът на конкретен алгоритъм става като маркерът се позиционира върху него и се кликне еднократно.

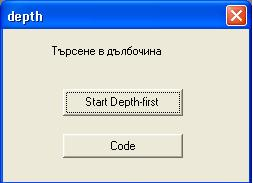
2.***3.4. Принципи за работа с алгоритмите на PETA***

**2.4.1. Работа с менюто „Search”**

Първото от менютата, включени в програмната срада за тестване на алгоритми, е менюто „Search”. То съдържа алгоритми и задачи, свързани с търсене в пространство на състоянията. Задачите за търсене са една основна част от задачите на Изкуствения интелект. Различни видове търсене (като алгоритми и методи) са познати в разработките по Изкуствен интелект. В програмната среда PETA са включени търсене в дълбочина, в широчина, евристично търсене, търсене при ограничения.

**2.4.1.1. „DepthFirst”**

Първата опция от меню „Search” – „DepthFirst” – е реализация на алгоритъм за търсене в дълбочина. При избор на този алгоритъм от менюто се появява следния диалогов прозорец:



Фиг. 2.13: depth – Диалогов прозорец за търсене в дълбочина

При натискане на бутона „Start Depth-first” алгоритъма се стартира. Резултата се извежда в полето „Messages” в долната част на програмния прозорец. При този алгоритъм не се въвеждат стойности от потребителя. Дървото, което се обхожда, е заложено в програмата и има следния вид:

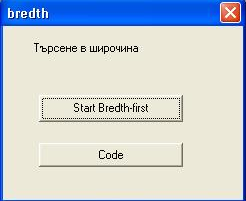
Фиг. 2.14: Дърво, което е заложено за обхождане в широчина и дълбочина

Пътят, който се намира като резултат е b – c – g – k. В резултата освен върха, който се посещава се извежда и пътят, по който е достигнато до него 0 – 1 – 5 – 9.

В дилаговия прозорец за търсене в дълбочина има и бутон „Code”. При натискане на този бутон потребителят има достъп до предиката в програмния код, който реализира търсенето. Предиката се изписва в полето „Messages”.

**2.4.1.2. „BreadtFirst”**

Втората опция от меню „Search” е реализация на алгоритъм за търсене в широчина. При избор на този алгоритъм от менюто, се появява следния диалогов прозорец:



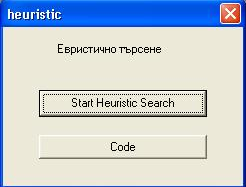
Фиг. 2.15: bredth – Диалогов прозорец за търсене в широчина

При натискане на бутона „Start Bredth-first” алгоритъма се стартира. Резултата отново се извежда в полето „Messages” в долната част на програмния прозорец. При този алгоритъм също не се въвеждат стойности от потребителя. Обхожда се дървото, заложено в програмата (фиг. 2.3).

При натискане на втория бутон в диалоговия прозорец „Code” в полето „Messages” се вижда програмния код на търсенето.

**2.4.1.3. „Heuristic”**

Тази опция от меню „Search” реализира евристично търсене. При нейния избор се появява следния диалогов прозорец:



Фиг. 2.16: heuristic – Диалогов прозорец за евристично търсене

При натискане на бутона „Start Heuristic Search” алгоритъма се стартира. Резултата отново е в полето „Messages” в долната част на програмния прозорец. При този алгоритъм също не се въвеждат стойности от потребителя.

При реализацията на търсене в широчина и дълбочина не се използват специални знания, които по някакъв начин да определят клон от дървото, по който търсенето да продължи. Тази слабост на двата алгоритъма се коригира в евристичната стратегия за търсене. При нея се оценяват „разходите” за преминаване от корена до целта, като се мине през даден връх N. Оценяването се извършва преди да сме посетили връх N.

Оценъчната функция, заложена в програмната среда PETA, се определя от следната формула:

*f*(*N*) = *g*(*N*) + *h*(*N*) (2.2)

Във Формула (2.1) g(N) е цената на прехода от корена до възела N, а функцията h(N) дава очакваните разходи за преминаване от възела N до целта. При евристичното търсене пътищата с по – малка стойност на f(N) се посещават първи.

При заложеното в програмата евристично търсене се намира път от A до S по следната карта:

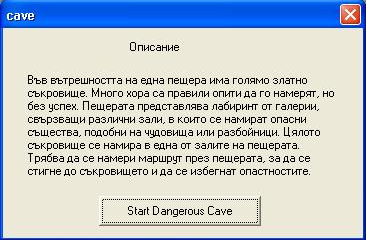
Фиг. 2.17: Карта за евристично търсене

Градовете са представени в програмния код със следните координати: xy("a", 2, 4). xy("b", 5, 6). Пътищата между градовете са описани по следния начин: op("a", "b" ). op("b", "m"). op("m", "f"). Всеки маршрут е представен като функтор от вида : t(real, real, it), където първия аргумент дава стойносттта на евристичната функция F, втория аргумент дава стойносттта на g(N). Сортиращ алгоритъм сравнява първите аргументи и насочва по-обещаващите „клони” от маршрута в началото. Така те се обхождат първи.

И при евристичното търсене потребителят има достъп до основния програмeн код, чрез който то е реализирано. Това става чрез натискане на бутона „Code”. Основния предиката за евристично търсене се извежда в полето „Messages”.

**2.4.1.4. „DangerousCave”**

Това е реализация на една от стандартните задачи за търсене – „Опасната пещера”. Типът на търсене в пространството на състоянията – в случая галериите на пещерата – е търсене при ограничения. При стартиране на задачата се появява диалогов прозорец, който изглежда по следния начин:

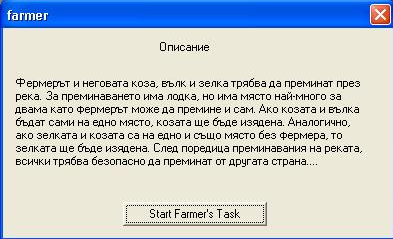


Фиг. 2.18: Cave – Диалогов прозорец на задачата за пещерата

Има описание на самата задача и на решението, до което трябва да бъде достигнато. Ограниченията, които се поставят са всъщност залите от пещерата, през които не трябва да се преминава. При натискане на бутона „Start Dangerous Cave” в полето „Messages” се извежда пътя през пещерата, който програмата е намерила.

**2.4.1.5. „Farmer”**

Тази опция от менюто реализира друга стандартна задача за търсене – „Задачата за фермера”. Отново типът на търсенето е търсене при ограничения, като в случая това са състоянията, в които не трябва да се попада. При стартиране на задачата се появява диалогов прозорец, който изглежда по следния начин:

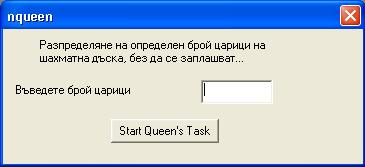


Фиг. 2.19: farmer – Диалогов прозорец на задачата за фермера

Отново е описана самата задача и решението, което се търси. При натискане на бутона „Start Farmer’s Task” се извежда поредицата преминавания на реката, които програмата е намерила като решение.

**2.4.1.6. „NQueen”**

Последната опция от менюто „Search” е реализация на евристично търсене. Това е стандартна задача, в която се търси подредба на царици на шахматна дъска, така че нито една от тях да не попада под обсега на другите. Това означава, че никои две царици не могат да бъдат поставени на един и същ ред, колона или диагонал. Диалоговия прозорец, който се появява след избор на „NQueen” изглежда по следния начин:



Фиг. 2.20: nqueen – Диалогов прозорец на задачата за цариците

В текстовото поле се въвежда броя на цариците, за който се търси решение. Това е и размерността на шахматната дъска. При задаване на 6 например имаме 6 царици на дъска с размерност 6X6. След това се натиска бутона „Start Queen’s Task” и решението се извежда в полето „Messages”. Извеждат се съответно редът и колоната, на които трябва да се постави всяка царица.

Алгоритъмът за работа със задачата за цариците е даден на Фиг.(2.30).

**2.4.2. Работа с менюто „Stack”**

Второто меню „Stack” реализира структурата от данни стек и основните операции с нея: извеждане на елементите на стека, вмъкване на нов елемент, достъп до елемента на върха на стека – това е последния вмъкнат елемент, изваждане на елемент от стека, дублиране на елемента на върха на стека, размяна на елемента на върха на стека с този пред него. При структурата от данни стек имаме достъп само до елемента на върха, затова операциите са свързани с него. В заложения в програмата стек има въведени четири елемента „aaa”, „bbb”,”ccc”,”ddd”. Те могат да бъдат изваждани, дублирани, разменяни. Това е стандартно, но потребителят може да въвежда свои елементи и да извършва операции с тях.

**2.4.2.1. „Show”**

Това е опция, която показва в полето „Messages” елементите, които се съдържат в стека. Ако стека е празен се извежда „[]”. Ако потребителят не е въвел свои елементи се извеждат тези от заложения в програмата стек: ["ddd","ccc","bbb","aaa"].

**2.4.2.2. ”Push”**

С тази опция се вмъква нов елемент в стека. При нейният избор се появява следният диалогов прозорец:



Фиг. 2.21: Give a string – Диалогов прозорец за въвеждане на елемент

В текстовото поле се въвежда стринга, който ще се добави като нов елемент. След това се натиска бутона „OK”. Елементът се добавя на върха на стека. При натискане на бутон „Cancel” диалоговия прозорец се затваря без елементът да бъде добавен. При въвеждане на некоректни данни (напр. въвеждане на число) ,се извежда съобщение за грешка.

Последователността от действия за добавяне на елемент в стека е дадена на Фиг.(2.31). Това е Activity – диаграма, която отразява алгоритъма, който се следва при добавянето на нов елемент структурата от данни стек.

**2.4.2.3. „Peek”**

Тази опция осъществява достъп до елемента на върха на стека – последния добавен елемент. Той се извежда в полето „Messages”. Елементът просто се показва, без да се изважда от стека.

**2.4.2.4. „Pop”**

С тази опция се изважда от стека елемента, който сме добавили последно(т.е. елемента на върха на стека).

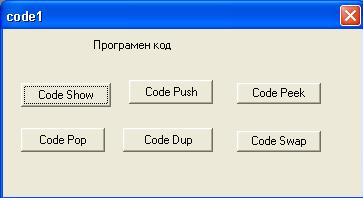
**2.4.2.5. „Dup”**

Тази опция дублира елемента на върха на стека. По този начин елементите в стека се увеличават с един, като последния и предпоследния са еднакви.

**2.4.2.6. „Swap”**

С тази опция се разменят елемента на върха на стека и този пред него. Извежда се и съобщение за извършеното действие в полето „Messages” .

**2.4.2.7.”Code”**

Тази опция дава достъп до програмния код за работа със стек. При избирането й се отваря следния диалогов прозорец:

Фиг.2.22: code1 – Диалогов прозорец за достъп до програмния код

В зависимост от това, за коя опция от менюто потребителят иска да види програмния код, се избира съответния бутон от диалоговия прозорец.

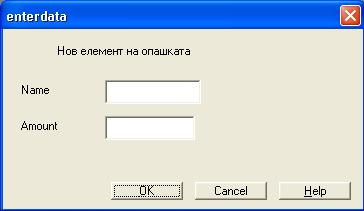
В резултат на това, в полето „Messages” се извежда предиката от програмния код, който реализира операцията.

**2.4.3. Работа с менюто „Queue”**

Това меню реализира структурата от данни опашка и основните операции с нея: добавяне на елемент към края на опашката, достъп до елемента в началото на опашката, изваждане на елемент от началото на опашката, дублиране на елемента в началото на опашката, разменяне на елемента в началото на опашката с този след него, показване на елементите на опашката. За разлика от структурата стек, тук няма стандартно заложени елементи в опашката. Тя е празна, докато потребителят не въведе елементи.

**2.4.3.1. „Push”**

Тази опция от менюто добавя елемент в края на опашката. Когато бъде избрана, се стартира следния диалогов прозорец:



Фиг. 2.23: enterdata – Диалогов прозорец за добавяне на елемент

За разлика от стека, опашката е реализирана с функтор и отделните й елементи са изградени от име (Name) – стринг и стойност (Amount) – цяло число. За да добави елемент, потребителя трябва да въведе стойности в двете полета : съответно в Name – стринг, а в Amount – цяло число. След това се натиска бутона OK и елементът се добавя в края на опашката. При натискане на бутона Cancel диалоговия прозорец се затваря и елементът не се добавя.

Алгоритъмът, който трябва да се следва, за да се добави елемент в опашката е представен на Фиг. (2.32) чрез Activity – диаграма.

**2.4.3.2. „Peek”**

С тази опция се осъществява достъп до елемента в началото на опашката, т.е. първия добавен елемент. Той се извежда в полето „Messages”. Ако опашката е празна се извежда съобщение: „Queue is empty”. Елемента не се изважда от опашката, а само се показва.

**2.4.3.3. „Pop”**

Тази опция изважда елемента, който е в началото на опашката. Ако опашката е празна се извежда съобщение.

**2.4.3.4. „Duplicate”**

Тази команда дублира елемента, който е в началото на опашката. Така елементите на опашката стават с един повече, като първия и втория елемент са еднакви. Ако опашката е празна се извежда съобщение: „Queue is empty, nothing to duplicate”.

**2.4.3.5.”Swap”**

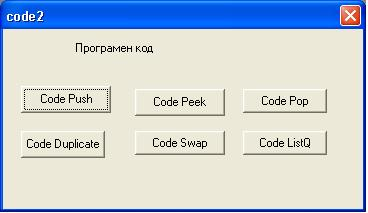
С тази опция се разменят местата на първия и втория елемент от опашката. Извежда се и съобщениe в полето „Messages”. Ако опашката е с по-малко от два елемента, съобщението е : „Queue contains less than two items. Nothing to swap” .

**2.4.3.6. „ListQ”**

Тази опция извежда елементите на опашката в лентата „Messages”. Ако тя е празна се извежда: „The queue contains: nothing” .

**2.4.3.7.”Code”**

Тази опция дава достъп до програмния код на операциите със структурата от данни опашка. При избор се появява следния диалогов прозорец:

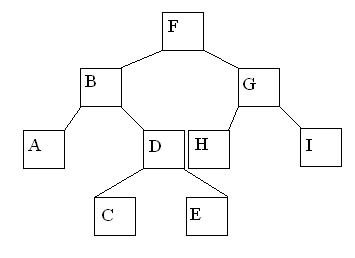


Фиг.2.24: code2 – Диалогов прозорец за достъп до програмния код

В зависимост от операцията, която го интересува, потребителят избира съответния бутон от диалоговия прозорец. Предиката, който реализира тази операция в структурата от данни опашка, се извежда в полето „Messages”.

**2.4.4. Работа с менюто „Trees”**

В това меню е раелизирана структурата от данни „дърво”, както и три различни възможности за нейното обхождане. Дървото е заложено стандартно в системата и не се въвежда от потребителя. Генерира се посредством списък от стойности. Програмата работи с двоично (бинарно) дърво. Двоично дърво е дървовидна структура от степен 2, т.е. всеки възел има не повече от два наследника - ляв и десен. При това стойностите на възлите в лявото поддърво са по-малки от стойността на корона, а стойностите на възлите в дясното поддърво са по-големи. В бинарните дървета търсенето е сравнително лесно и ефективно. Те се наричат още дървета за търсене. Дървото, заложено в програмата е следното:



Фиг. 2.25: Бинарното дърво заложено за обхождане в меню „Trees”

**2.4.4.1. „create”**

Първата опция от меню „Trees” създава бинарното дърво. Дървото се изгражда от списък с елементи, които ще бъдат заложени като възли. Посредством функцията в програмния код „add\_from\_list” елементите се добавят от списъка един по един, посредством функцията „treeinsert” те намират мястото си в дървото. Първия елемент от списъка става корен на бинарното дърво – F. Останалите елементи се сравняват с него. Ако са по-малки се записват в лявото поддърво, а ако са по-големи – в дясното. Това е стандартен алгоритъм за изграждане на бинарно дърво. След създаването на дълвото (Фиг.2.25) се извежда съобщение в полето „Messages”: „Tree has been created”.

Трите обхождания на бинарното дърво трябва да бъдат реализирани, след като дървото е създадено, чрез опцията „create”.

**2.4.4.2. „traverse PRE-order”**

Тази опция реализира обхождане на бинарното дърво в прав ред – PREORDER. Това е всъщност обхождане в следния ред: корен, ляво, дясно. Първо се посещава корена, след това лявото поддърво и накрая дясното поддърво. Резултатът от обхождането се извежда в полето „Messages”.

**2.4.4.3. „traverse IN-order”**

С тази опция бинарното дърво се обхожда симетрично или „във вътрешен ред”. Обхождането е в следната последователност: ляво, корен, дясно. Първо се обхожда лявото поддърво, след това корена и накрая дясното поддърво. Резултатът отново се извежда в полето „Messages”.

**2.4.4.4 .„traverse POST-order”**

Тази опция е обхождане на дървото в обратен ред. Последователността е следната: първо се обхожда лявото поддърво, след това дясното поддърво и накрая корена. Обхождането се извежда в полето „Messages”.

**2.4.4.5.”Code”**

Подобно на стека и опашката, и тук потребителят има достъп до програмния код. Диалоговия прозорец, който се визуализира при избор, е следния:



Фиг.2.26: code3 – Диалогов прозорец за достъп до програмния код

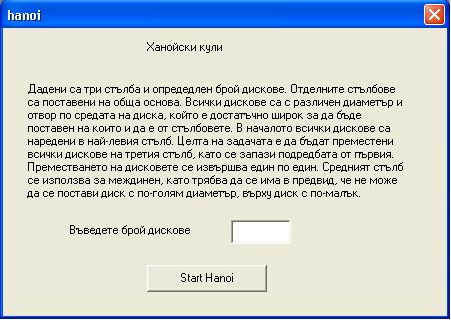
Избира се операцията, за която да се покаже програмния код и се натиска съответния бутон. Предиката, реализиращ операцията, се извежда в полето „Messages”.

**2.4.5. Работа с менюто „Planing”**

В това меню е представен планиращ алгоритъм. Планирането е много важна част от Изкуствения интелект. То се състои в намиране на последователност от действия за достигане на някакво множество от цели при дадени начални условия и временни ограничения.

**2.4.5.1. „Hanoi”**

В програмната среда PETA е реализиран един от стандартните алгоритми за планиране, решаващ задачата „Ханойски кули”. Когато се стартира опцията, се появява диалогов прозорец, който изглежда по следния начин:



Фиг.2.27: hanoi – Диалогов прозорец на задачата „Ханойски кули”

В началото е дадено обяснение на задачата и на решението, което се търси. Отдолу има поле, в което потребителят трябва да въведе броя на дисковете, с който иска да намери решение. След като е въведен брой, чрез бутона „Start Hanoi” се стартира задачата и решението се извежда в полето „Messages”. Решението е всъщност последователността от премествания на дискове, която трябва да се извърши, за да се реши задачата.

Ако не бъде въведена стойност от потребителя в полето за брой на дисковете и задачата се стартира се извежда съобщение за грешка. Така е и при останалите алгоритми, които изискват въвеждане на стойност.

Последователността от действия, която трябва да се реализира при използването на задачата „Ханойски кули”, е представена и на Фиг.(2.33) чрез Activity – диаграма.

**2.4.6. Работа с менюто „Neurons”**

В това меню е представена реализация на невронна мрежа - двуслоен персептрон.

Персептрона е една от първите реализации на невронни мрежи. При него има сумиране на всички постъпващи входни сигнали умножени по теглото (различно тегло за всеки входен сигнал). Ако резултатът е по-голям от даден фиксиран праг, неврона дава ненулев изход. Тази схема се подобрява от добавянето към сумата на сигмоидна функция и вземане на стойността, върната като изход от персептрона. Ако построите графиката на тази функция, тя прилича на гръцката буква сигма, откъдето идва и името й. Сигмоидната функция се дефинира по следния начин:

*&*(*x*) = (2.3)

На Пролог тази дефиниция изглежда така:

sigmoid(X)= 1.0/(1.0+math::exp(-X)).

Една интересна способност на Персептрона е способността му за обучение. Той може да бъде обучен да разпознава входния модел. Всъщност, давайки достатъчно примери, има алгоритъм, който модифицира теглата на Персептрона, така че той може да направи разлика между два или повече модела. За пример двуслойния персептрон, който е заложен и в програмната среда PETA, може да бъде обучен да определя кога изходът ще бъде 1 и кога ще бъде 0. Обучаващият алгоритъм на Персептрона трябва да има множество от обучаващи примери.

Теглата са заложени в програмния код по следния начин:

weight(0, 1.5). weight(1, 2.0). weight(2, 1.8). Това са инициализиращи стойности и те са фиксирани. Истинските стойности трябва да бъдат намерени с алгоритъма за обучение.

Това обаче е представяне чрез недетерминиран предикат, а ние имаме нужда от процeдура. Затова в програмния код е дефинирано следното:

getWgt(I, W) :- weight(I, W), !.

getWgt(\_I, 0.0).

Обучаващият алгоритъм е базиран на пресмятане и поправяне на грешката, направена от предвиждащата функция. Грешката при даден конкретен пример се пресмята от следния предикат:

evalError(Obj, E) :- nn:predicted\_out(Obj, VC),

nn:egratia(Obj, \_I1, \_I2, V),

E= (VC-V)\*(VC-V).

Сумата от грешките, допускани при всички примери се намира по следния начин:

errSum(Exs, \_Partial\_err, Total\_err) :- acc := 0.0,

foreach Eg= list::getMember\_nd(Exs) do

evalError(Eg, Err), acc := acc + Err

end foreach,

Total\_err= acc.

Тежестите се модифицират по такъв начин, че да се намали грешката до минимум. Модифицирането им става по метода „спускане по наклон” , по същия начин, както става редуцирането на грешката до минимум.

updateWeight(DX, LC, Exs) :- errSum(Exs, 0.0, Err0),

PN= [tuple(I, W) || nn:getWgt\_nd(I, W)],

nn:setWeight([ tuple(P, NV) ||

tuple(P, V) = list::getMember\_nd(PN),

V1= V+DX,

nn:assertWgt(P, V1),

errSum(Exs, 0.0, Nerr),

nn:popweight(P, V1),

NV= V+LC\*(Err0-Nerr)/DX]).

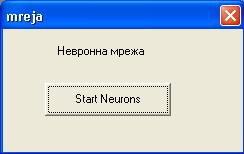
„Спускането по наклон” е модифициращ алгоритъм, който намира локален минимум на функцията. На всяка стъпка от метода „спускане по наклон” , тежестите на Персептрона се модифицират по следната формула:

*Wn*+1=*Wn - (3.4)*

В тази формула, ако е достатъчно малко число, error() < error(). Ако се започне с , то поредицата ще клони към минимум. Минимумът ще бъде достигнат , когато при спускането се достигне до нула(или число близко до нулата в практическите ситуации). Идеалната ситуация се случва, когато мрежата достигне до глобален минимум.

**2.4.6.1 . „Start”**

При избор на тази опция се появява следния диалогов прозорец:



Фиг. 2.28: mreja – Диалогов прозорец за стартиране на невронната мрежа

След натискане на бутона „Start Neurouns” невронната мрежа се стратира и резултатът се извежда в полето „Messages”.

Персептронът, който е заложен в програмната среда PETA e двуслоен и извършва пресмятане на логическите операции „XOR” – изключващо или и „&&” – логическо и. По дефиниция логическото и е истина тогава, когато поне едно от твърденията е истина. Изключващото или е истина тогава, когато едното твърдение е истина, а другото е лъжа.

При двуслойният персептрон има два слоя неврони, през които преминава информацията от входа до изхода на невронната мрежа. Този вид персептрон е една от първите реализации на невронни мрежи.

Първите два реда от резултата извеждат общата грешка: съответно при операцията „XOR” и „&&”. По надолу са конкретните стойности, които са получени при пресмятането от невронната мрежа. Те са реални числа, близки съответно до 0 или 1, в зависимост от това дали операцията дава резултат истина или лъжа. Това показва, че обучението на невронната мрежа се стреми да приближи алгоритъма до поведение, което е много близко до желаното поведение на операциите „XOR” и „&&”.

**2.4.7. Работа с менюто „Genetic”**

В последната опция от лентата с менюта на програмната среда PETA, е представена реализация на генетичен алгоритъм. Това е всъщност реализация на търсене чрез генетичен алгоритъм по модела на Голдберг.

В конкретния пример, който е представен, е реализирана популация с 10 индивида. Генетичните операции, които се използват са кръстосване (crossover) и мутация (mutation). Кросоувъра, който е представен, е с две точки на делене.

В алгоритъма е използвано бинарно(двоично) представяне на информацията. Така всеки индивид е представен с поседователност от 8 цифри: 0 или 1. При кросоувъра, при два индивида тази последователност се разделя на две места, и те разменят части от заложената в тях в информация. При мутацията имаме произволна промяна на последователността: т.е. промяна на някоя от нулите в единица или обратно. Така мутацията дава възможност за добиване на нова информация в поколението, която не е била присъща на старато (родителско) поколение.

Новото поколение се генерира чрез избор на най-добрите индивиди от старото. Това става чрез т.нар фитнес – функция. Чрез нея се оценява качеството на всеки индивид. Фитнес – фукцията, която е заложена при алгоритъма в програмната система PETA, е следната:

(2.4)

Във формула. (2.4) value e десеттичната стойност на двоичното (бинарно) представяне на информацията (индивидите в поколението). Например ако представянето на индивида е 01000010, то value = 66.

Генетичната операция мутация се реализира чрез следния предикат:

 mutateCellWithProba(Cell) :-  
        rnd() < mutationRate,  
        extract(Cell),   
        CellMutated = Cell:copy(), CellMutated:mutate(),  
        addCell(CellMutated ,1),  
    !.%  
    mutateCellWithProba(\_).

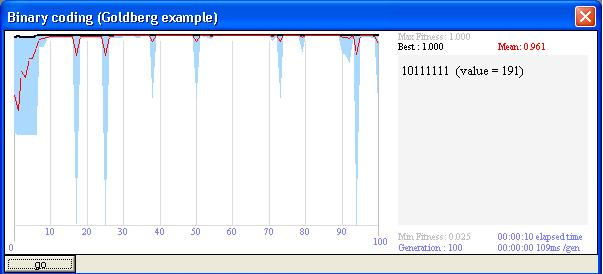
Кросоувъра (кръстсването на представителите в популацията) се реализира чрез предиката:

 performsCrossover() :-  
        std::repeat(),  
           C1= cellAtRandom(), extract(C1),  
           C2= cellAtRandom(), extract(C2),  
           C1:crossoverWith(C2,CC1,CC2), % two points cross over  
           newGen:addCell(CC1,1),  
           newGen:addCell(CC2,1),  
        newGen:isFull(),!  
    .%  
    performsCrossover().

**2.4.7.1. „Binary”**

Тази опция стартира генетичния алгоритъм. Търсенето, което се реализира, всъщност намира минимум на функция.

При избор на „Binary” се отваря бял диалогов прозорец с бутон „Go” в долната лява част. Когато се натисне бутона в диалоговия прозорец се появява графиката на генетичния алгоритъм (Фиг. 2.15).

Фиг. 2.29: Binary coding(Goldberg example) – Диалогов прозорец на генетичния алгоритъм

Алгоритъма действа, докато средната стойност на фитнес – функцията стане 1. Отстрани се извежда двоичното и десеттичното представяне (value) на решението, което е намерено.

Отдолу в полето „Messages” се извеждат представителите на първото поколение (Initial population) и на последното (Last population). Извежда се и най-добрият представител на последното поколение, който всъщност е и решението на алгоритъма за търсене.

***Приложения- Програмен код на алгоритмите в системата PETA***

**Приложение 1: Depth First**

class predicates

dSearch:(queue, path) determ (i,o).  
   nextLevel:(path, path) nondeterm (i,o).  
   solved:(path) determ.  
   prtSolution:(path).

clauses

nextLevel([r(Branch, N)|Path],[r(Op, Daughter),r(Branch, N)|Path ]):-  
    operator(Op, N, Daughter),  
    not(list::isMember(r(Op, Daughter),Path)).

 prtSolution(L):-  
    foreach P= list::getMember\_nd(list::reverse(L)) do  
     stdio::write(P), stdio::nl  
     end foreach.

dSearch([T|Queue],Solution):-if solved(T)   
then Solution=T else Extentions= [Daughter|| nextLevel(T, Daughter)],  
ExtendedQueue= list::append(Extentions, Queue),  
dSearch(ExtendedQueue, Solution) end if.

**Приложение 2: Breadth First**

class predicates  
   bSearch:(queue, path) determ (i,o).  
   nextLevel:(path, path) nondeterm (i,o).  
   solved:(path) determ.  
   prtSolution:(path).

   clauses

 bSearch([T|Queue],Solution) :- if solved(T)  
    then Solution=T else Extentions= [Daughter||nextLevel(T, Daughter )],  
     ExtendedQueue= list::append(Queue, Extentions),  
      bSearch(ExtendedQueue, Solution) end if.  
            
   nextLevel([r(Branch, N)|Path],[r(Op, Daughter),r(Branch, N)|Path ]):-  
    operator(Op, N, Daughter),  
    not(list::isMember(r(Op, Daughter),Path)).

 prtSolution(L):-  
    foreach P= list::getMember\_nd(list::reverse(L)) do  
     stdio::write(P), stdio::nl  
     end foreach.

**Приложение 3: Heuristic**

class predicates  
       
     getXY: (string, integer, integer) determ (i,o,o).  
     cost: (string, string) ->real.  
     hn: (string)-> real.  
     not\_in\_circle: (string, it ) determ (i,i).  
     theGoal: (string) procedure (o).  
     toDaughter: (node, node) nondeterm (i,o).  
     init: (string) procedure (o).  
     goalReached:(node) determ.  
     search: (tree, node) determ (i,o).  
     prtSolution: (node) procedure (i).  
     solve: () procedure.  
     cmp: (node, node) -> compareResult.

 clauses 

    search([T|Queue],S):-  
     if goalReached(T) then S=T  
      else Extension= [E||toDaughter(T,E)],  
      NewQueue= list::append(Queue, Extension),  
      BestFirst = list::sortBy(cmp, NewQueue),  
      search (BestFirst, S)  
      end if.  
        
      toDaughter(t(\_F,G,[r(B,N)|Path]),t(F1,G1, [r(Op, Child), r(B,N)|Path])):-  
             op(N, Child),  
             Op=string::format("%s to %s", N, Child),  
             not\_in\_circle(Child, Path),  
             G1= G+ cost(N, Child), F1=G1+hn(Child).  
               
       prtSolution(t(\_,\_,T)):-  
        foreach  X=list::getMember\_nd(list::reverse(T)) do  
         stdio::write(X), stdio::nl  
         end foreach.  
           
         solve():- if init (E), search([t(hn(E),0,[r("root", E)])],S)  
               then prtSolution(S)  
               else stdio::write("No Solution") end if.

**Приложение 4: Stack**

  predicates  
    push : (string Element) procedure (i).  
    peek : (string Element) determ (o).  
    pop : (string Element) determ (o).  
    dup : () determ.  
    swap : () determ.  
    showstack : () .

 clauses  
    push(Element) :-  
        stacklist := [Element | stacklist].       
  
    peek(Element) :-  
        [Element | \_ ] = stacklist.        
  
    pop(Element) :-  
        [Element | Tail] = stacklist,     
        stacklist := Tail.             
   
    dup() :-  
        [Element | \_] = stacklist,             
        stacklist := [Element | stacklist].        
      
    swap() :-  
        [Element1, Element2 | Tail] = stacklist,       
        stacklist := [Element2, Element1 | Tail].       
      
    showstack() :-  
        stdIO::write("The stack contains: "),  
        stdIO::nl,  
        stdIO::write(stacklist).

**Приложение 5: Queue**

predicates

push : (string Name, real Amount) procedure (i,i).  
peek : (string Name, real Amount) procedure (o,o).  
pop : (string Name, real Amount) procedure (o,o).  
listQ : ().  
duplicate : (string Name, real Amount) procedure (o,o).  
swap : (string Name1, real Amount1, string Name2, real Amount2) determ (o,o,o,o).

clauses

 push(Name, Amount) :-  
        assertz(q(Name, Amount)).  
  
    peek(Name, Amount) :-  
        q(Name, Amount), !.  
    peek("0",0).  
  
    pop(Name, Amount) :-  
        retract(q(Name, Amount)), !.  
    pop("0",0).  
  
class predicates  
    listAllInQueue : () procedure .  
clauses  
    listQ() :-  
        q(\_Name, \_Amount), !,                                         
        listAllInQueue().  
    listQ() :-                                                        
        stdIO::write("nothing"), stdIO::nl.  
  
    listAllInQueue() :-  
        foreach q(Name, Amount)  
        do stdIO::write(Name, " ", Amount),  
            stdIO::nl  
        end foreach.  
  
    duplicate(Name, Amount) :-  
        q(Name, Amount), !,  
        asserta(q(Name, Amount) ).  
    duplicate("0",0).  
      
    swap(Name1, Amount1, Name2, Amount2) :-  
        retract(q(Name1, Amount1)),                                             
        assert(qout(Name1, Amount1)),                                         
        retract(q(Name2, Amount2)), !,                                              
        asserta(q(Name1, Amount1)),                                          
        asserta(q(Name2, Amount2)),  
        retract(qout(Name1, Amount1)).                                         
    swap("0",0,"0",0) :-  
        retract(qout(Name1, Amount1)),                                        
        asserta(q(Name1, Amount1)).

**Приложение 6: Tree**

predicates  
    add\_from\_list : (stringlist) procedure (i).  
    treeinsert : (string NewItem, treetype OldTree, treetype Newtree) procedure (i,i,o).  
    traverse : (string TraverseOrder) procedure (i).  
    preOrderTraverse : (treetype Tree) procedure (i).  
    inOrderTraverse : ( treetype Tree) procedure (i).  
    postOrdertraverse : (treetype Tree) procedure (i).

clauses  
    add\_from\_list([]) :- !.  
    add\_from\_list([NewItem | Tail]) :-  
        treeinsert(NewItem, treefact, NewTree),  
        treefact := Newtree,  
        add\_from\_list(Tail).  
  
    treeinsert(NewItem, empty, tree(NewItem, empty, empty)) :- !.  
    treeinsert(NewItem, tree(Element, LEft, Right), tree(Element, Newleft, Right) ) :-  
        NewItem <= Element, !,  
        treeinsert(NewItem, Left, NewLeft).  
    treeinsert(NewItem, tree(Element, LEft, Right), tree(Element, Left, Newright) ) :-  
        treeinsert(NewItem, Right, NewRight).  
      
    traverse("preorder") :-  
        !, stdIO::nl, stdIO::write("the tree in PRE-order:"),  
        preOrderTraverse(treefact), !, stdIO::nl.  
    traverse("inorder") :-  
        !, stdIO::nl, stdIO::write("the tree in IN-order:"),  
        inOrderTraverse(treefact), !, stdIO::nl.  
    traverse("postorder") :-  
        !, stdIO::nl, stdIO::write("the tree in POST-order:"),   
        postOrderTraverse(treefact), !, stdIO::nl.  
    traverse(\_).  
      
    preOrderTraverse(empty) :- !.  
    preOrderTraverse(tree(Element, Left, Right)) :-  
        stdIO::write(Element, " "),  
        preOrderTraverse(Left),  
        preOrderTraverse(Right).  
      
    inOrderTraverse(empty) :- !.  
    inOrderTraverse(tree(Element, Left, Right)) :-  
        inOrderTraverse(Left),  
        stdIO::write(Element, " "),  
        inOrderTraverse(Right).  
  
    postOrderTraverse(empty) :- !.  
    postOrderTraverse(tree(Element, Left, Right)) :-  
        postOrderTraverse(Left),  
        postOrderTraverse(Right),  
        stdIO::write(Element, " ").