Vysoké učení technické v Brně – Fakulta informačních technologií

Tutorial ke knihovně GAlib pro C++

Luděk Žaloudek (xzalou04)

Spolupracovníci:
Radek Brich (xbrich02)
Tomáš Kavalek (xkaval01)
Jan Štefl (xstefl02)

1. Co je to GAlib

GAlib znamená „Genetic Algorithms library“. Jak již název vypovídá, jde o knihovnu pro C++ určenou k práci s genetickými algoritmy (GA). Za jejím vývojem stojí Matthew Wall z MIT. Zde je jeho stránka o GAlibu: http://lancet.mit.edu/ga/. Na stránce se můžeme dozvědět, že knihovna se dá použít v na jakékoli platformě a s jakoukoli reprezentací či operátory. Příklady obsahují dokonce ukázky paralelního zpracování. To je sice pěkné, ale dovedu si představit, že začátečníkovi z toho jde hlava kolem. Cílem tohoto tutoriálu je ukázka základních vlastností GAlibu a implementace jednoduchého problému s důrazem na srozumitelnost.

Při představě programování GA není třeba hned panikařit, GAlib nevyžaduje nějaké hluboké znalosti z oboru. Nicméně základy teorie o EA nikomu neuškodí. Na počátku GA je inicializována startovací populace, obvykle náhodně. Taková populace obsahuje určitý počet kandidátních řešení. Řešení jsou v každém cyklu algoritmu (tomuto cyklu se říká generace) vyhodnocena tzv. fitness funkcí, která určí jejich vyspělost. Poté nastává fáze reprodukce, kdy jsou lepší řešení vybírána s větší pravděpodobností než horší a jsou na ně aplikovány genetické operátory. Nejlepší jedinci (případně všichni) z těchto reprodukovaných jsou vybráni do další generace a celý cyklus se opakuje, dokud nedojdeme k ukončovacímu kritériu (obvykle požadovanému řešení).

2. Instalace

Instalace by mohla někomu způsobit problémy, proto ze zde o ní zmiňuji. Předem upozorňuji, že nejjednodušší je instalovat GAlib na Linux nebo Unix, protože podpora knihovny na MS Windows není zrovna kvalitní. Zde se vývoj zastavil někde u starších verzí kompilátorů od MS či Borlandu, takže je nutné ručně nastavit spoustu parametrů překladače a dělat vše z řádky. Mimo to jsou jisté problémy s striktností MS překladače, který dále vyžaduje udělat některé zásahy do kódu, aby šla knihovna vůbec přeložit. Pokud se však najde nějaký nadšenec s pevnými nervy, který si na instalaci ve Windows troufne, pak by se mohl podělit o své zkušenosti s ostatními.

Nejlepší způsob jak začít, je rozbalit archiv s GAlibem a překopírovat jeho obsah do složky *GAlib* v domovském adresáři. Tím se trochu zjednodušší makefiles, které budeme tvořit pro své programy. Čím hlubší cesta v adresáři, tím delší příkaz pro překlad a tím větší chaos. Po zkopírování můžeme změnit obsah souboru *makevars* podle svého překladače, OS a cesty pro instalaci a poté stačí napsat „*make install*“, případně „*make*“ či „*make test*“, podle toho, chcete-li kompletně nainstalovat knihovnu, pouze ji zkompilovat či otestovat příklady.

3. Základní prvky programu

Objekt genetického algoritmu

Ačkoli se může pojem objektu genetického algoritmu zdát poněkud abstraktní, není to tak docela pravda. Tento objekt je totiž zdaleka ten nejdůležitější v GAlibovém programu. Zahrnuje v sobě všechny kontrolní proměnné GA včetně pravděpodobností operátorů, přiřazení samotných funkcí reprezentujících operátory, délky trvání GA – tj. zastavovací kritérium, a samozřejmě je v něm obsažena definice typu samotného algoritmu. V GAlibu jsou čtyři základní typy algoritmu.

Standardní *GASimpleGA* (povšimněte si, že každá třída v GAlibu začíná GA) je jednoduchý algoritmus, který v každé generaci používá úplně novou populaci, přičemž je možno zvolit elitismus – tedy že se nejlepší jedinec z poslední generace přenáší do generace následující. Abychom mohli tento typ GA použít, je nutné vepsat na začátek programu řádek:

```
#include <ga/GASimpleGA.h>
```

Dalším typem v řadě je *GASteadyStateGA*. U toho je možné pracovat s překrývajícími se populacemi. Lze rovněž určit, jaká část populace bude v každé generaci nahrazena. Pro tento typ je třeba vložit následující řádek:

```
#include <ga/GASStateGA.h>
```

Třetí je inkrementální GA s třídou jménem *GAIncrementalGA*. V tomto typu algoritmu je možné definovat způsob, jakým dojde k včleňování nové generace do stávající populace. Potomci mohou nahrazovat svoje rodiče, náhodné jedince či dokonce jedince sobě nejvíce podobného. Pro tento typ je třeba vložit následující řádek:

```
#include <ga/GAIncGA.h>
```

Posledním základním typem je *GADemeGA*. Zde je vyvíjeno několik paralelních populací s použitím steady-state algoritmu. V každé generaci je několik jedinců z jedné populace přesunuto do některé z dalších. Pro tento typ je třeba vložit následující řádek:

```
#include <ga/GADemeGA.h>
```

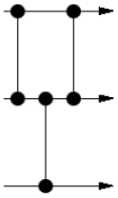
Existuje možnost vytvořit svůj vlastní typ GA. V příkladech přiložených ke knihovně by mělo být několik takových ukázek.

Další důležité prvky jsou různé statistiky a vlastnosti, které se dají dobře využívat k ladění algoritmu a zobrazování všech potřebných informací o tom, jak výpočet proběhl.

Jakmile jsme se rozhodli pro typ GA, je čas na další kroky. Nemá smysl popisovat zde výhody a nevýhody různých typů algoritmů, to by vydalo na další knihu. Začátečníkovi bude stačit jednoduchý GA. Můžeme tedy přejít k problému reprezentace dat.

Reprezentace

Aby bylo čtenáři hned jasné, o co vlastně jde, uvedu příklad: Chceme najít řadicí síť s určitým počtem vstupů, která má mít jistý maximální počet komparátorů. Řadicí síť se vlastně celá skládá z vodičů a dvouvstupových komparátorů, které na výstupech seřadí sestupně nebo vzestupně dodaná čísla. Dalo by se říci, že řadicí síť je vlastně něco jako n-vstupový komparátor. Máme tedy jakýchsi n úrovní vodičů, mezi které vkládáme komparátory. Reprezentovat to můžeme tak, že každému komparátoru přiřadíme dvojici čísel podle toho, které vstupní úrovně spojuje. Máme-li tedy sekvenci 0-1, 1-2, 0-1, mělo by nám být jasné, že reprezentuje tuto řadicí síť:



Stejně tak může např. binární strom reprezentovat rozmístění objektů v 2D prostoru nebo pole řetězců sadu pravidel, podle kterých máme sestavit elektrický obvod apod. Jde vlastně o jakési kódování kandidátních řešení. Odborně se těmto zakódovaným řešením říká genotypy a kódům namapováným do skutečnosti (tedy skutečným řešením) říkáme fenotypy. To však začátečníka nemusí příliš zajímat.

Každé takové struktuře obsahující jediné řešení říkáme **genom**. Je velmi důležité, aby struktura byla naprosto minimální, protože se tím snižuje celková doba výpočtu. Je však rovněž důležité, aby struktura reprezentovala všechny důležité aspekty problému. Byla by nám k ničemu reprezentace řadicí sítě, kdyby obsahovala pouze jedno číslo pro každý komparátor.

GAlib nám dává k dispozici pestrou škálu předdefinovaných struktur. Patří mezi ně např. *GA1DBinaryStringGenome*, *GA2DArrayGenome*< *T* > nebo *GATreeGenome* < *T* >. T v ostrých závorkách naznačuje, že jsou použity šablony, takže je možné používat různých typů. Chceme-li např. definovat genom pro uvedenou řadicí síť, vypadalo by nějak takto (k údajům v závorce konstruktoru se dostaneme později):

```
GA1DArrayGenome<int> genome(glength, ::fitness);
```

Samozřejmě musíme ještě ke každé datové struktuře vložit patřičný hlavičkový soubor:

```
#include <ga/GA1DArrayGenome.h>
```

Pokud někdo mluví o samotném **genu** (ne genomu), má na mysli jedinou část datové struktury. U příkladu výše by to bylo jediné číslo typu *int*.

Zde je nutno dodat jednu poznámku. Čím jsou datové typy a struktury genomu složitější, tím obezřetnější musíme být při definování genetických operátorů. Např. u křížení řadicí sítě by bylo nevhodné, kdyby se bod křížení mohl vybrat mezi dvěma čísly patřícími k jednomu komparátoru. Na podobné věci je třeba myslet. Tím se dostáváme k dalšímu kroku.

Definice genetických operátorů

Pro každý genom existují tři základní genetické operátory: inicializace, křížení a mutace. V GAlibu jsou tyto operátory předdefinovány, ale zcela zásadní je možnost si je upravit. Definice operátorů je další (téměř) nezbytný krok před samotným spuštěním výpočtu.

Inicializace je zvláštní operátor, který se volá jen na začátku GA. Nevytváří žádné nové genomy, ale implantuje genetický materiál do těch stávajících v počáteční populaci. Typicky jsou to náhodné hodnoty. Ale pozor, i zde je třeba mít na mysli smysluplnost. Bylo by hloupé v našem vzorovém problému řadicí sítě inicializovat v některém jedinci komparátory tak, aby byla možnost dvou stejných čísel u jednoho komparátoru.

Křížení je nejobvyklejší genetický operátor. Obvykle se jeho pravděpodobnost pohybuje okolo 70%. (To však neznamená, že u některých problémů se křížení zcela vynechává a dává se přednost mutaci, ale uvádím obvyklé postupy). Křížení je několik různých typů, ale zcela základní je křížení jednobodové. Funguje tak, že se náhodně vybere křížící bod u dvou tzv. rodičovských jedinců a druhá část genomu za bodem se mezi nimi prohodí. Mohlo by to vypadat například takto:

Rodič 1:	2 5 8 6 2 1 0 7 2 3
Rodič 2:	6 4 1 3 4 5 7 2 0 3
Potomek 1:	2 5 8 6 4 5 7 2 0 3
Potomek 2:	6 4 1 3 2 1 0 7 2 3

Mnohem zajímavějším operátorem je mutace. Hlavní důvode oné zajímavosti je fakt, že mutace může mít mnohem více různých obměn a je možné, aby nejen modifikovala existující genetický materiál, ale rovněž může vytvářet úplně nový. Mutace může být jeden ze způsobů, jak lze vyklouznout z lokálního minima, což je problém který často pronásleduje složitější úlohy v GA. Mutace může vypadat tak, že dochází k prohození dvou náhodných genů, výměně dvou podstromů ve stromových strukturách nebo k vygenerování nového kusu genomu. Mutace však může být i destruktivní – dojde např. k umazání určitého genu či podstromu apod. Pravděpodobnost mutace se obvykle nastavuje na jednotky procent, aby k ní nedocházelo příliš často. Jinak bychom totiž mohli dojít k obyčejnému náhodnému prohledávání stavového prostoru a to by byla jasná degradace GA.

Další ukázky různých verzí křížení a mutace můžete najít v úvodu dokumentace ke knihovně.

Na ukázku vkládám kód uživatelem definovaného inicializačního operátoru. Jak je vidět, není to nic těžkého:

```
void initializer(GAGenome &g)
{
      GA1DArrayGenome<bool> &genome = (GA1DArrayGenome<bool>&) g;

      // inicializace náhodnými bity
      for (int i = 0; i < glength; i++)
```

```
                genome.gene(i, GARandomBit());
                // lze také genome[i] = ...
        }
```

Výhodou GAlibu je rovněž to, že se používají ukazatele na funkce k přiřazování operátorů genomu. Proto je možné měnit operátory během algoritmu. Rovněž není nutné derivovat novou třídu kvůli změně chování vestavěných typů genomu. Nevýhodou je možnost chyby při práci s ukazateli. Tady je ukázka přiřazení operátoru výše ke genomu:

```
genome.initializer(::initializer);
```

Stanovení objektivní funkce

Objektivní funkce slouží k tomu, abychom věděli, jak dobře slouží daný jedinec zamýšlenému účelu. Fitness hodnota je potom výsledek objektivní funkce transformovaný tak, aby bylo možné určit vhodnost (fitness) daného jedince k páření. Někdy se rozdíl mezi oběmi hodnotami ztrácí a objektivní funkce je zvána fitness funkcí. Fitness je skvělá vlastnost všech EA, protože k dosažení cíle není potřeba nějakých zvláštních znalostí, ale stačí nám pouze jediné číslo. Dobře či špatně napsaná objektivní funkce má podstatný vliv na výsledek celého GA.

Shrnutí

Teď už známe všechny tři podstatné úkony, který musí být udělány, aby bylo možné vyřešit problém pomocí GA. Pro jistotu je ještě zopakuji:

- Definování reprezentace
- Definování genetických operátorů
- Definování objektivní (fitness) funkce

4. První program

Nyní zkusíme svůj první jednoduchý program. Proložím ho komentáři v normálním textu, aby bylo vše jasné. Program bude hledat jedince se všemi jedničkovými bity. Jako genom poslouží typ *GA1DBinaryString*, který má předdefinované funkce pro inicializaci, mutaci a křížení. Ukončovací kritérium je počet generací. Všimněte si, že při nižším počtu generací program nemusí nalézt řešení.

Nejprve je třeba vložit potřebné hlavičkové soubory a definovat konstanty, které budou použity jako parametry GA.

```
#include <iostream.h>
#include <ga/GASimpleGA.h>
#include <ga/GA1DBinStrGenome.h>

const int popsize = 10;       // velikost populace
const int ngens = 200;        // počet generací
const int glength = 20;       // délka genomu
const float pmut = 0.01;      // pravděpodobnost mutace ...
const float pcross = 0.7;     // ... a křížení
```

Teď je třeba přidat fitness funkci, která implicitně není definovaná. Funkce prochází celým polem binárních hodnot a pokud najde *true*, tak připočítá k fitness hodnotě 1. To znamená, že čím vyšší fitness, tím lepší jedinec.

```
float fitness(GAGenome &g)
{
    GA1DBinaryStringGenome &genome = (GA1DBinaryStringGenome&) g;
    float fit = 0.0;

    for (int i = 0; i < glength; i++)
      if (genome.gene(i)) fit++;

    return fit;
}
```

Dále následuje funkce main, kde jsou nastaveny parametry GA, je inicializován generátor náhodných čísel a je rozběhnut samotný výpočet genetického algoritmu.

```
int main(int argc, char **argv)
{
    GA1DBinaryStringGenome genome(glength, ::fitness);
    GARandomSeed();
    // vytvoření a nastavení parametrů jednoduchého GA
    GASimpleGA ga(genome);
    ga.populationSize(popsize);     // velikost populace
    ga.nGenerations(ngens);   // ukončení po ngens generacích
    ga.pCrossover(pcross);    // určení pravděpodobnosti křížení ...
    ga.pMutation(pmut);       // ... a mutace
    ga.evolve();              // spuštění evoluce

    // výpis výsledku
    cout << "Nalezene reseni:" << endl;
    cout << ga.statistics().bestIndividual() << endl;

    return 0;
}
```

To je celý první program v GAlibu.

Makefile

Při překladu příkladů v tomto tutoriálu je vhodné využít Makefile, který zde přikládám. GAlib si potrpí na několik speciálních parametrů, které jsou v Makefile zahrnuty. Je třeba, aby byla knihovna uložena v adresáři /~/GAlib. Při překladu bez Makefile je potom nutné uvést tyto parametry překladače:

-I../GAlib - pro informaci o hlavičkových souborech
-L../GAlib/ga - adresář s knihovnou (soubor libga.a)
-lga - přilinkování GAlibu k programu

Kompilovaný program musí byt umístěn v nějakém adresáři domovského adresáře. Pokud by byl hlouběji, např. ~/EVA/Projekt, musí se u parametru překladače zohlednit druhá úroveň zanoření ../../GAlib.

Samotný Makefile tedy bude vypadat takto (nezapomeňte změnit případná jména programů):

```
all:  Priklad1.cc Priklad2.cc
      g++ -o Priklad1 Priklad1.cc -Wno-deprecated -lcurses -I../GAlib -
L../GAlib/ga -lga
      g++ -o Priklad2 Priklad2.cc -Wno-deprecated -lcurses -I../GAlib -
L../GAlib/ga -lga
```

```
clean:
	rm -f Priklad1 Priklad2
```

Než přistoupíme k druhé, trošku složitější ukázce, vysvětlím ještě několik dalších důležitých pojmů, bez kterých by se programátor GAlibu neměl obejít.

5. Další důležité vlastnosti a třídy

GAStatistics

V případě, že ladíte svůj program, je dobré znát, co se děje během algoritmu. K tomu slouží právě třída *GAStatistics*. Velmi užitečná je nejen na konci každého algoritmu, ale během jednotlivých kroků. To je mimochodem další zajímavá věc. GA můžete spustit nejen jako např. `ga.evolve();`, ale je možné postupovat po krocích. To se může hodit jak k zjišťování ukončovacího kritéria, tak např. při implementaci *koevoluce*, kdy dojde ke kroku v několika GA současně a mezi tím se předají různé hodnoty potřebné pro výpočty fitness hodnot koevolvujících jedinců. V jednoduchém případě (koevoluci hned raději nezkoušejte :-) to může vypadat nějak takto:

```
while (ga.statistics().maxEver() != glength)
{
	ga.step();  // krok GA
}
```

Užitečnými funkcemi mohou být zejména: *bestEver*, *bestIndividual*, *generation*

Při ukončení běhu programu je možné použít následující řádek zobrazující nejdůležitější statistiky:

```
cout << ga.statistics() << endl;
```

GAPopulation

Tento objekt vlastně funguje jako kontejner na genomy. Má vlastní funkce na inicializaci a vyhodnocování (které obvykle volají podobné funkce u každého jednotlivého genomu) a také udržuje další statistické informace o své populaci. To může být průměr, odchylky atd.

V GAPopulation je rovněž definována metoda výběru (souvisí s reprodukcí) a také obsahuje škálovací schéma pro konverzi objektivní hodnoty na fitness.

Předchozí vlastnosti obvykle nemusí člověka zajímat, ale důležité je, že v každém kroku lze z GA „vytáhnout“ aktuální populaci a dělat s ní různé věci, třeba zpracovávat vlastní statistiky nebo ji použít třeba při koevoluci.

6. Druhý, složitější program

Teď se dostáváme k ukázce druhého programu, který obsahuje definice vlastních funkcí, jiný typ ukončovacího kritéria a pár dalších maličkostí. Zařadil jsem ho hlavně proto, aby si někdo nestěžoval, že první program je příliš jednoduchý (ten jsem ale radši zařadil, aby si někdo nestěžoval, že je ten druhý příliš složitý :-). Pokud plně porozumíte tomuto programu, jste na nejlepší cestě začít psát vlastní skutečné programy s GAlibem.

V tomto programu jde vlastně o to samé, co v tom prvním, ale chromozom je reprezentován typem *GA1DArrayGenome<bool>*. Příklad slouží především jako ukázka uživatelsky definovaných operátorů genetického algoritmu, stejným způsobem by bylo možné nadefinovat například uniformní křížení nebo

pokročilou mutaci pro složitější problémy. U definovaných funkcí musí byt dodrženo předepsané pořadí a typ parametru. Ukončující podmínkou GA je dosažení maximální fitness hodnoty u některého jedince - hledané řešení. Není omezeno počtem generací, GA skončí, jakmile se zrodí jedinec s jedničkovými bity.

Opět musíme vložit hlavičkové soubory a určit konstanty.

```
#include <iostream.h>
#include <ga/GASimpleGA.h>
#include <ga/GA1DArrayGenome.h>

const int popsize = 10;             // velikost populace
const int glength = 20;             // délka genomu
const float pmut = 0.01;            // pravděpodobnost mutace ...
const float pcross = 0.7;           // ... a křížení
```

Následuje výpočet fitness, téměř stejný jako u předchozího příkladu, jen s jiným datovým typem.

```
float fitness(GAGenome &g)
{
    GA1DArrayGenome<bool> &genome = (GA1DArrayGenome<bool>&) g;
    float fit = 0.0;

    for (int i = 0; i < glength; i++)
      if (genome.gene(i)) fit++;

    return fit;
}
```

Další na řadě je definice funkce operátoru mutace. Náhodný bit (gen) je změněn na opačnou hodnotu. Povšimněte si náhodné funkce *GAFlipCoin*, která představuje hod mincí s předdefinovanou pravděpodobností pádu na jednu stranu. Náhodné generátory a související funkce mohou být užitečnými pomocníky pro programování evoluce. Doporučuji nahlédnout do dokumentace GAlibu. Návratová hodnota této funkce narozdíl od fitness slouží pouze k sledování statistiky počtu mutací.

```
int mutator(GAGenome &g, float pmut)
{
    GA1DArrayGenome<bool> &genome = (GA1DArrayGenome<bool>&) g;
    int result = 0;     // pro statistické účely, není nezbytně nutné

    for (int i = 0; i < glength; i++)
    {
      if (GAFlipCoin(pmut))   // náhodný bit s p-sti pmut, ze padne "1"
      {
          genome.gene(i, !genome.gene(i));
          result++;
      }
    }

    return result;
}
```

Jednobodové křížení. V případě, že by šlo např. o řadicí síť, musela by následovat kontrola, zda nedošlo ke křížení uprostřed komparátoru a nedošlo třeba k duplikování jeho vstupu. Vždy je dobré se zamyslet, jestli vaše gen. operátory neprodukují nesmysly, naštěstí v případě hledání jedniček to není nutné. Všimněte si další náhodné funkce *GARandomInt*.

```
int crossover(const GAGenome &p1, const GAGenome &p2,GAGenome *o1, GAGenome
*o2)
{
    GA1DArrayGenome<bool> &parent1 = (GA1DArrayGenome<bool>&) p1;
    GA1DArrayGenome<bool> &parent2 = (GA1DArrayGenome<bool>&) p2;
    GA1DArrayGenome<bool> &offspring1 = (GA1DArrayGenome<bool>&) *o1;
    GA1DArrayGenome<bool> &offspring2 = (GA1DArrayGenome<bool>&) *o2;

    int cpoint = GARandomInt(1, glength - 1);  // bod křížení (v mezích od,
do)
    for (int i = 0; i < glength; i++)
    {
      if (i < cpoint)
      {
          offspring1.gene(i, parent1.gene(i));
          offspring2.gene(i, parent2.gene(i));
      }
      else
      {
          offspring1.gene(i, parent2.gene(i));
          offspring2.gene(i, parent1.gene(i));
      }
    }

    return 1;     // opět pouze pro statistiku
}
```

Inicializace je téměř stejná, jako v předchozím příkladě, jenom s jiným datovým typem.

```
void initializer(GAGenome &g)
{
    GA1DArrayGenome<bool> &genome = (GA1DArrayGenome<bool>&) g;

    // inicializace náhodnými bity
    for (int i = 0; i < glength; i++)
      genome.gene(i, GARandomBit());
      // lze také genome[i] = ...
}
```

Nyní následuje trochu větší funkce main, než v prvním příkladě.

```
int main(int argc, char **argv)
{
    // vytvoření chromozomu ve zvolene reprezentaci
    GA1DArrayGenome<bool> genome(glength, ::fitness);
    GARandomSeed();     // inicializace generátoru náhodných čísel
    genome.initializer(::initializer);   // inic. funkce pro chromozom
    genome.mutator(::mutator);      // nastaveni funkce operátoru mutace
    // vytvoření jednoduchého GA a nastavení parametrů
    GASimpleGA ga(genome);
    ga.crossover(::crossover);      // operátor křížení
    ga.populationSize(popsize);     // velikost populace
    ga.pCrossover(pcross);          // p-st křížení ...
    ga.pMutation(pmut);             // ... a mutace
    ga.initialize(0);
```

```
    // v případě nalezení jedince s jedničkovými bity odpovídá maximální
    // hodnota fitness funkce délce chromozomu -> ukončující podmínka evol.
    while (ga.statistics().maxEver() != glength)
    {
      ga.step();  // krok GA
      // zde mohou být případně další činnosti
      // ....
    }
    ga.flushScores();   // nutné při "ručním" ukončení GA

    cout << "Reseni nalezeno v " << ga.generation() << ". generaci" <<
endl;
    cout << ga.statistics().bestIndividual() << endl;

    return 0;
}
```

7. Závěr

GAlib obecně umožňuje velmi mnoho předělávek již definovaných věcí a téměř se nenajde parametr, který by nešel změnit. Já sám jsem již v GAlibu řešil několik poměrně složitých problémů včetně vyrovnání se již nejlepším dosaženým řadicím sítím a koevoluce. Ne všechny se mi však podařilo vyřešit. Je třeba mít na paměti, že GA nejsou dokonalé a často se stává, že řešení uvázne na nějakém lokálním optimu a může trvat neúnosně dlouho, než přijdeme na nějaké řešení, jak z něho vyklouznout. Proto od této knihovny neočekávejte zázraky. Dalším problémem může být čas. Evoluce složitějších problémů často trvá celé hodiny nebo dokonce dny a využívá velkou porci CPU. Proto prosím nepouštějte takové problematické programy na školních serverech, které k tomu nejsou určeny.

Vzhledem k dalším hlavním směrům EA, jako jsou genetické programování, evoluční strategie a evoluční programování, je asi nejen z mé strany použití GAlibu neprobádanou končinou. Je však více než pravděpodobné, že pro to existují vhodnější nástroje.

Než se pustíte do skutečného programování s GAlibem, doporučuji se alespoň zběžně podívat na jeho domovskou stránku a do dokumentace. Najdete tam více příkladů a informací, které se do tohoto krátkého tutoriálu nevešly.

Evoluci zdar!

8. Zdroje

1. Žaloudek Luděk, Koevoluce řadicích sítí a trénovacích vektorů, bakalářská práce, FIT VUT Brno, 2004
2. Bidlo Michal, Evoluční návrh řadicího algoritmu, diplomový projekt, FIT VUT Brno, 2004
3. GAlib: A C++ Library of Genetic Algorithm Components, version 2.4, Documentation Revision B, August 1996
4. http://lancet.mit.edu/ga/
5. Ukázkové příklady zaslané M. Bidlem