OBJEKTOVÁ KNIHOVNA EVOLUČNÍCH ALGORITMŮ

David Bražina, Hashim Habiballa, Viktor Pavliska

Katedra informatiky a počítačů, PřF OU, 30. dubna 22, Ostrava 1

Ústav pro výzkum a aplikace fuzzy modelování, OU, 30. dubna 22, Ostrava 1

david.brazina@osu.cz, hashim.habiballa@osu.cz, viktor.pavliska@osu.cz*

Abstrakt

Příspěvek popisuje návrh knihovny pro implementaci evolučních algoritmů v optimalizačních problémech. Tato problematika je řešena v rámci grantu GAČR "Evoluční algoritmy se soutěžícími a spolupracujícími heuristikami". Článek postupně stručně popisuje principy evolučních algoritmů a zejména se zaměřuje na návrh objektové hierarchie pro řešení optimalizačních problémů s možností jejího rozšiřování uživatelem.

Abstract

The article describes the design of software library for evolutionary algorithms implementation in optimization problems. Creation of the library is related to the project GAČR "Evolutionary algorithms with competition and cooperation of heuristics". The paper briefly shows principles of evolutionary algorithms and it focuses mainly to the structure of the library for solutions of optimization problems with the possibility to extend it by its user.

Úvod

Úlohou nalezení globálního minima účelové funkce je nalezení bodu s nejnižší funkční hodnotou v prohledávaném prostoru. V řadě metod je potřeba nalézt globální minimum (nebo maximum) v souvislé oblasti a účelovou funkci umíme vyhodnotit s požadovanou přesností v každém bodu. Tuto úlohu lze formulovat následovně.

Mějme účelovou funkci

$$f : D \rightarrow R, D \subseteq R^d.$$

Pak $x^* = \arg\min_{x \in D} f(x)$ je globální minimum.

V takových úlohách však je možné užít stochastických algoritmů pro globální optimalizaci, zejména evolučních algoritmů [1], [2].

Z matematické analýzy známe postup, jak nalézt extrémy funkcí, u kterých existuje první a druhá derivace. Zdálo by se, že úloha nalezení globálního minima je velmi jednoduchá. Bohužel tomu tak není. Nalézt obecné řešení takto jednoduše formulovaného problému je obtížné, zvláště když účelová funkce je multimodální, není diferencovatelná, případně má

* Tato práce byla podporována z projektu GAČR: 201/05/0284 – „Evoluční algoritmy se soutěžícími a spolupracujícími heuristikami“.

další nepříjemné vlastnosti. Analýza problému globální optimalizace ukazuje, že neexistuje deterministický algoritmus řešící obecnou úlohu globální optimalizace (tj. nalezení dostatečně přesné aproximace x^*) v polynomiálním čase, tzn. problém globální optimalizace je NP-obtížný. Přitom globální optimalizace je úloha, kterou je nutno řešit v mnoha praktických problémech, mnohdy s velmi významným ekonomickým efektem, takže je nutné hledat algoritmy, které jsou pro řešení konkrétních problému použitelné [5].

Evoluční algoritmy jsou ve své podstatě modely Darwinovského vývoje populací. Charakteristické pro ně je to, že pracují s jednou či více populacemi jedinců (jedinec je většinou bod v prohledávaném prostoru a využívají tzv. evoluční operátory:

- selekce – nejsilnější jedinci z populace mají větší pravděpodobnost přežití a rozmnožení svých vlastností,
- křížení (rekombinace) – dva nebo více jedinců z populace si vymění informace a vzniknou tak noví jedinci kombinující vlastnosti rodičů,
- mutace – informace zakódovaná v jedinci muže být náhodně modifikována,
- migrace – u tzv. paralelních algoritmu, kdy existuje více populací vedle sebe, někteří jedinci migrují mezi populacemi.

V rámci řešení projektu GAČR je na katedře informatiky a počítačů Ostravské Univerzity vyvíjena objektová knihovna těchto algoritmů, která má poskytnout praktickou realizaci teoretických výsledků výzkumu. Knihovna umožní nejen použití teoreticky zkoumaných technik založených na principu evolučních algoritmů, ale i eventuální rozšiřování jejími uživateli.

Knihovna je programována v jazyce C++ a je založena na objektových principech. V tomto článku se věnujeme popisu práce, která byla provedena v prvních měsících práce na projektu, kdy jde především o specifikaci cílů a návrhu struktury knihovny. Tento návrh se opírá nejen o identifikaci a deklaraci abstraktních tříd, které vyplývají z logiky řešeného problému, ale i o konkrétní potomky těchto tříd pro jeden z nejjednodušších algoritmů – jednoduchou variantu řízeného náhodného prohledávání (CRS).

Objektová knihovna

V rámci návrhu vytvářené objektové knihovny byly specifikovány základní třídy objektů pro univerzální aplikaci evolučních algoritmů pomocí jazyka UML [3]. Jedná se o následující třídy:

– **`EAlgorithm`**: Jedná se o nejdůležitější třídu, která v sobě obsahuje jádro konkrétního algoritmu pro řešení optimalizačního problému. Potomci této abstraktní třídy musí implementovat minimálně dvě metody – `Compute`, která řídí vlastní proces výpočtu konkrétního algoritmu a `NewPopulation`, která specifikuje způsob tvorby nové populace.

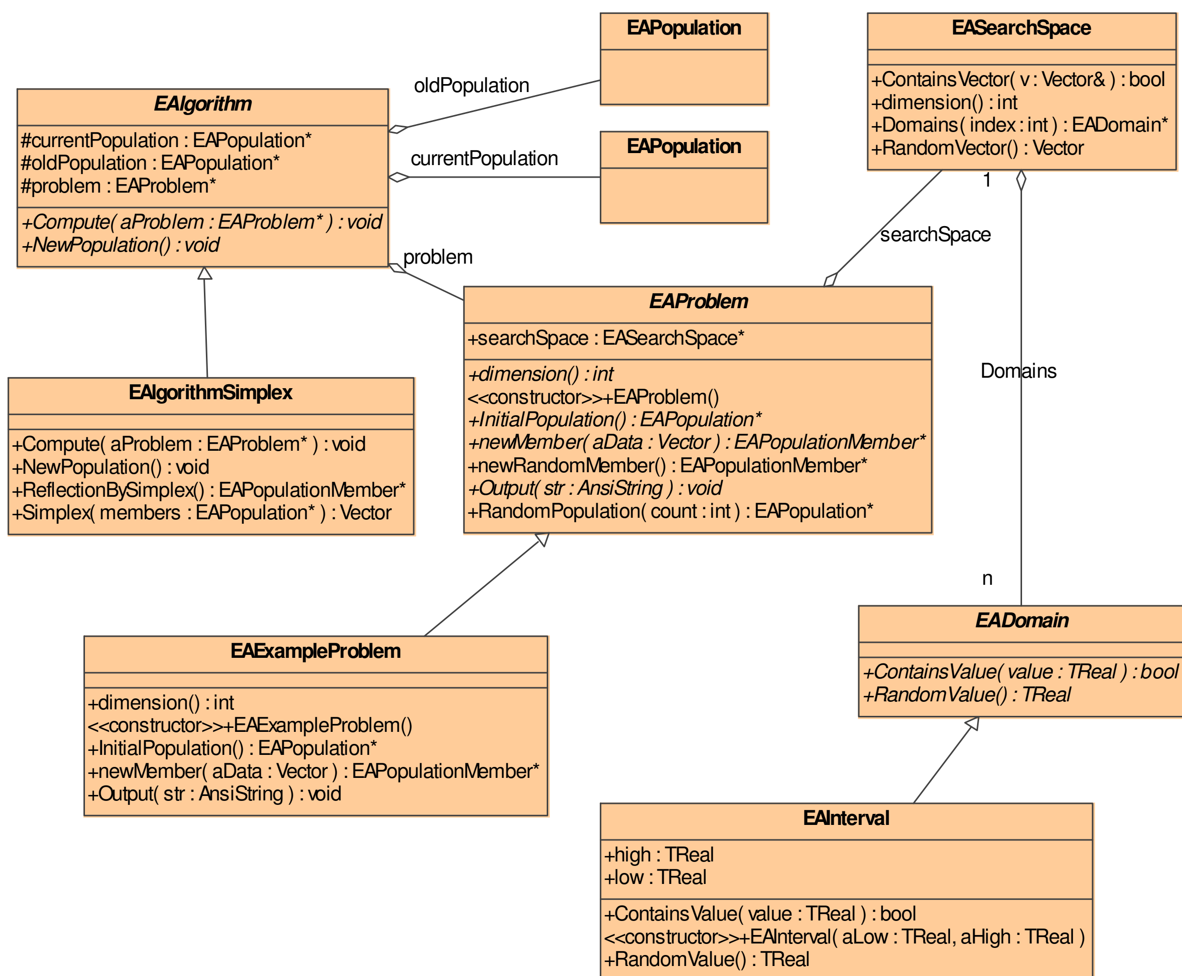
– **`EAProblem`**: Tato třída obsahuje zadání konkrétního řešeného problému. Instanci potomka této třídy musí mít k dispozici algoritmus, který má za úkol daný problém vyřešit. Potomci této třídy musí implementovat metodu `dimension`, která určuje dimenzi (rozměr) problému a dále metodu `InitialPopulation`, která vytvoří počáteční populaci, k čemuž může využít metody `RandomPopulation` (generuje náhodnou populaci) anebo může sama vygenerovat konkrétní populaci. Klíčovou metodou je `NewMember`, která je schopna z předložených dat vytvořit nového člena populace odpovídající typu řešené úlohy.

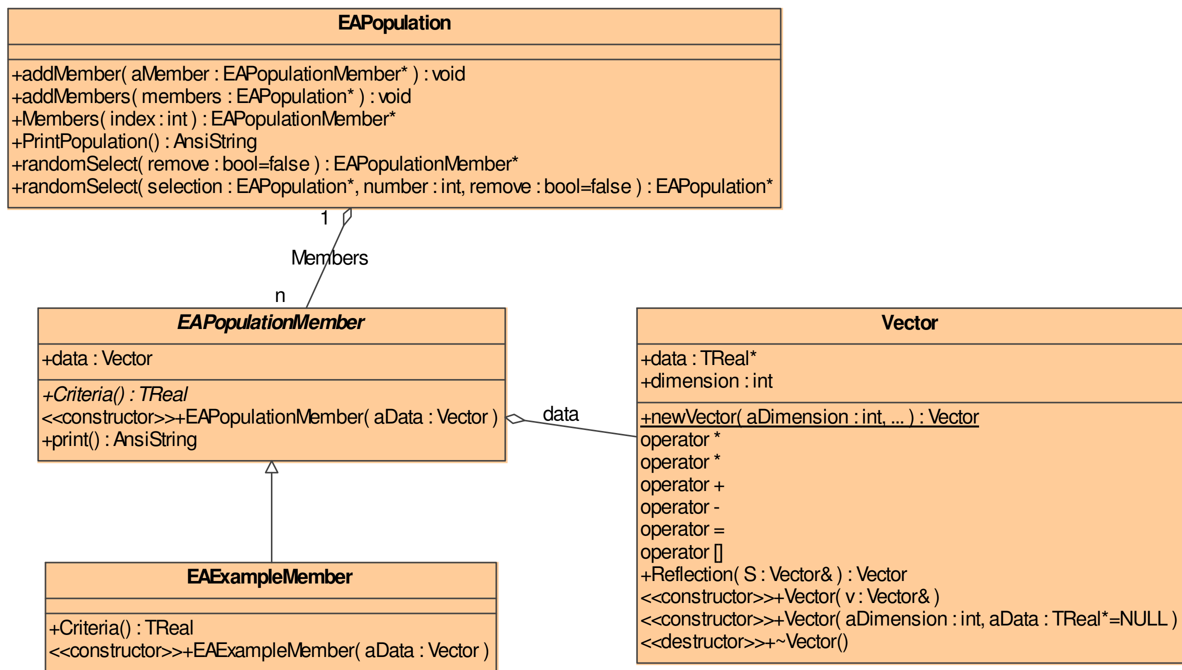
– **`EAPopulation`**: Implementuje seznam členů populace a umožňuje s nimi pracovat pomocí následujících metod: `addMember`, která přidá do populace nového člena a `randomSelect` umožňující náhodný výběr.

– **`EAPopulationMember`**: Zapouzdřuje datovou strukturu `Vector`, která slouží k uchovávání dat specifikujících člena v prohledávaném prostoru a dále důležitou metodu `Criteria`, která umí ohodnotit člena pomocí funkce popisující řešenou úlohu.

– **`EASearchSpace`**: Nese informace o prohledávaném prostoru a pomocí metody `ContainsVector` umožňuje určit příslušnost k němu a dále poskytuje metodu `RandomVector`, která vygeneruje náhodný vektor.

– **`EADomain`**: Abstraktní třída, jejíž potomci implementují strukturu jednoho rozměru prohledávaného prostoru. Nejpoužívanější implementací je jednoduchý interval `EAInterval`.





Způsob využití

Objektová knihovna poskytuje dva základní přístupy k jejímu využití:

1. Uživatelský: spočívá ve využití existujících potomků třídy `EAlgorithm` (uživatel si nevytváří vlastní algoritmy, ale pouze využívá již hotových). V tomto případě musí specifikovat pouze vlastní potomky tříd `EAProblem` a `EAPopulationMember`.
2. Programátorský: jde o náročnější úroveň použití, která spočívá ve vytváření vlastních potomků třídy EAlgorithm, čímž si může rozšířit knihovnu o implementaci vlastních evolučních algoritmů specifických pro jeho úlohy.

Pro oba přístupy je nutná znalost programovacího jazyka C++ včetně zásad objektově orientovaného programování. Dokumentace projektu bude tyto dva přístupy rozlišovat.

Řešený příklad

Ukázkou programátorského rozšíření knihovny je implementace algoritmu jednoduché varianty řízeného náhodného prohledávání (CRS) [4].

Řízené náhodné prohledávání (controlled random search, CRS) je příkladem velmi jednoduchého a přitom efektivního stochastického algoritmu pro hledání minima v souvislé oblasti. Algorimus CRS navrhl Price v sedmdesátých létech minulého století. Pracuje se s populací N bodů v prohledávaném prostoru D a z nich se heuristicky generuje nový bod $\mathbf{y}$, který může být zařazen do populace místo dosud nejhoršího bodu. Počet vygenerovaných bodů populace N je vetší než dimenze d prohledávaného prostoru D.

Jako heuristiku pro generování nového bodu $\mathbf{y}$ užíval Price reflexi simplexu, která je známa ze simplexové metody. Z populace se vybere náhodně $d + 1$ nekomplanárních bodů tvořících simplex. Pokud v novém bodu $\mathbf{y}$ je funkční hodnota $f(y)$ menší než je v nejhorším bodu populace, pak je tento nejhorší bod nahrazen bodem $\mathbf{y}$. Reflexi v simplexu můžeme vyjádřit vztahem

$$y = 2g - x,$$

kde x je bod simplexu s největší hodnotou účelové funkce a **g** je těžiště zbývajících *d* bodů simplexu. Nahrazením nejhoršího bodu populace novým bodem **y** dosahujeme toho, že populace se koncentruje v okolí dosud nalezeného bodu s nejmenší funkční hodnotou. Algoritmus můžeme jednoduše zapsat následujícím způsobem:

```
generuj populaci P, tj. N bodů náhodně v D
repeat
  najdi xworst z P takové, že f(xworst) >= f(x), x z P
  repeat
    y := reflexe simplexu, y z D
  until f(y) < f(xworst);
  xworst := y;
until podmínka ukončení;
```

V této ukázce je vytvořen potomek `EAlgorithmSimplex` odvozený z abstraktní třídy `EAlgorithm`.

```
class EAlgorithmSimplex : public EAlgorithm {
public:
  virtual void Compute(EAProblem *aProblem);
  virtual void NewPopulation();
  Vector Simplex(EAPopulation *members);
  EAPopulationMember* ReflectionBySimplex();
};

void EAlgorithmSimplex::Compute(EAProblem *aProblem) {
  problem = aProblem;

  oldPopulation = problem->InitialPopulation();

  problem->Output("Initial Population:");
  problem->Output(oldPopulation->PrintPopulation());

  for (int i = 0; i < 50; i++) {
    NewPopulation();

    problem->Output("New Population " + IntToStr(i+1) + ":");
    problem->Output(currentPopulation->PrintPopulation());

    oldPopulation = currentPopulation;
  }
}

Vector EAlgorithmSimplex::Simplex(EAPopulation *members) {
  // simplex se vytvori jako aritmeticky prumer z prvnich dim clenu
  Vector suma(problem->dimension());
  for(int i = 0; i < problem->dimension(); i++) {
    suma = suma + members->Members(i)->data;
  }
  return suma * (1.0 / problem->dimension());
}

EAPopulationMember* EAlgorithmSimplex::ReflectionBySimplex() {
  // novy clen vznikly reflexi podle simplexu
  EAPopulationMember *reflection = NULL;

  EAPopulation *selection = new EAPopulation;
```

```
  oldPopulation->randomSelect(selection,    problem->dimension()    +    1,
rsRemove);

  // vypocet simplexu z prvnich dim hodnot (0..dimension-1)
  Vector simplexVector = Simplex(selection);

  /* kontrola, zda se vektor nachazi v prohledavanem prostoru */
  if(problem->searchSpace->ContainsVector(simplexVector)) {
    // nejhorsi clen, ktery bude pouzit pro reflexi
    // clenove jsou setrizeni, takze posledni je nejhorsi
    EAPopulationMember *memberX = selection->Members(problem->dimension());
    // vypocet reflexe podle simplexu
    Vector reflectedVector = memberX->data.Reflection(simplexVector);
    // novy clen vznikly reflexi podle simplexu
    reflection = problem->newMember(reflectedVector);
  }

  // pridani vsech clenu pouzitych pro reflexi podle simplexu
  currentPopulation->addMembers(selection);

  delete selection;
  return reflection;
}

void EAlgorithmSimplex::NewPopulation() {
  currentPopulation = oldPopulation;
  // konrola min poctu clenu pro simplex
  if(oldPopulation->Count < problem->dimension() + 1) {
    return;
  }

  EAPopulationMember *newMember;
  bool populationChanged = false;

  while(!populationChanged) {
    newMember = ReflectionBySimplex();
    if(newMember == NULL) {
      continue;
    }
    // nejhorsi clen soucasne populace
    EAPopulationMember *worst = oldPopulation->worst();

    // kdyz je nove vytvoreny clen lepsi nez nejhorsi,
    // tak jej vytlaci z populace pryc
    if(newMember->Criteria() < worst->Criteria()) {
      currentPopulation->Remove(worst);
      delete worst;

      currentPopulation->addMember(newMember);
      populationChanged = true;
    }
    else {
      // kdyz je nove vytvoreny clen horsi nez puvodni, tak se zahodi
      delete newMember;
    }
  }
}

class EAExampleMember : public EAPopulationMember {
public:
  EAExampleMember(Vector aData);
  virtual TReal Criteria();
};
```

```
class EAExampleProblem : public EAProblem {
public:
  EAExampleProblem();
  virtual EAPopulationMember* newMember(Vector aData);
  virtual void Output(AnsiString str);
  virtual EAPopulation* InitialPopulation();
  virtual int dimension();
};

TReal EAExampleMember::Criteria() {
  return data*data;
}

EAExampleMember::EAExampleMember(Vector  aData)  :  EAPopulationMember(aData)
{
}

EAExampleProblem::EAExampleProblem() {
  searchSpace->Add(new EAInterval(-1, 1));
  searchSpace->Add(new EAInterval(-1, 1));
  searchSpace->Add(new EAInterval(-1, 1));
}

int EAExampleProblem::dimension() {
  return searchSpace->dimension();
}

EAPopulationMember* EAExampleProblem::newMember(Vector aData) {
  return new EAExampleMember(aData);
}

EAPopulation* EAExampleProblem::InitialPopulation() {
  return RandomPopulation(20);
}
```

Závěr

Prezentovaný projekt (resp. jeho implementační část) je prozatím v úvodní fázi. Další fáze bude spočívat v implementaci jednotlivých typů již známých algoritmů včetně jejich nově zkoumaných modifikací. Pro tento účel konkrétní specifikace abstraktních tříd vytvoří hierarchii algoritmů podle jejich typu. Projekt je plánován na dobu tří let a výsledná verze by měla být tedy k dispozici v roce 2007.

Literatura

[1] TVRDÍK, J. *Stochastické algoritmy pro globální optimalizaci a jejich aplikace ve výpočetní statistice*. Habilitační práce : Přírodovědecká fakulta OU, 2004. 135 s.

[2] KVASNIČKA, V., POSPÍCHAL, J., TIŇO, P.: Evolučné algoritmy. STU, Bratislava, 2000.

[3] MERUNKA, V. *Objektový přístup a UML v návrhu IS*. In Tvorba Softwaru 2002, Tanger 2002, Ostrava. s. 123-133.

[4] TVRDÍK, J., *Generalized controlled random search and competing heuristics.* In MENDEL 2004, 10th Internetional Conference on Soft Computing (Matoušek R. and Ošmera P. eds). University of Technology, Brno, 2004. s. 228-233.

[5] TVRDÍK, J. *Evoluční algoritmy*. Učební texty Ostravské Univerzity, Ostrava 2004, 73 s.