

Programação Genética Fortemente Tipada com Condições Sintáticas

Rafael Castro

rafaelcgs10@gmail.com

Departamento de Ciência da Computação
Centro de Ciências e Tecnológicas
Universidade do Estado de Santa Catarina

4 de Dezembro de 2017

Método básico STGP

- Programação Genética Fortemente Tipada
- Utiliza tipos para reduzir o espaço de busca: em um caso, o espaço de busca são 10^{27} possíveis, mas somente 10^{12} são programas bem tipados.
- PolyGP utiliza Cálculo Lambda Tipado (sem abstração lambda). Recursão limitada apenas para o nome da função a ser evoluída.
- O polimorfismo paramétrico (Damas-Milner) tem papel fundamental: como funções podem ser aplicadas à argumentos de vários tipos, então há um grande reuso do material genético.

Objetivo: condições sintáticas

O objetivo é verificar o impacto do uso de condições sintáticas que evitam explorar programas que não são possíveis soluções.

Por exemplo, a função `if c e1 e2` pode ter a exata mesma sub-árvore para `e1` e `e2`, ou seja, o valor de retorno do `then` é o mesmo do `else`. Portanto, pode-se considerar a condição $e1 \neq e2$.

As condições podem ser custosas computacionalmente, pois podem necessitar comparar partes de programas e aplicar durante todo o processo evolucionário causaria um considerável aumento de tempo da evolução.

Representação: Cálculo Lambda

- Programas em Cálculo Lambda (sem abstração lambda).

```
data Expr = NamedFunction Id [Id] Expr Type |
            Lit Literal Type | Var Id Type |
            Primi Id Type | App Expr Expr Type |
            List [Expr] Type | ARG Type

data Type = Dummy TVar | Generic TVar |
            Temporary TVar | TArrow Type Type |
            TList Type | TConst TConst
```

Geração de programas

- O algoritmo de geração de programas funciona selecionando aleatoriamente funções primitivas (cujos os argumentos são temporariamente completados com ARGs) ou terminais, e aplicando as necessárias unificações e substituições de tipo.

- ① ARG::[G2].
- ② if::Bool->[G2]->[G2]->[G2].
- ③ (((if ARG::Bool) ARG::[G2]) ARG::[G2])::[G2].
- ④ (((if ARG::Bool) ARG::[G2]) []:[G2])::[G2].
- ⑤ (((if ARG::Bool) []:[G2]) []:[G2])::[G2].
- ⑥ (((if (null ARG::[T1]):Bool) []:[G2])
[]:[G2])::[G2].
- ⑦ (((if (null (tail ARG::[T1]):T1):Bool) []:[G2])
[]:[G2])::[G2].
- ⑧ (((if (null (tail l:[G1]):G1):Bool) []:[G2])
[]:[G2])::[G2].

Operador de *crossover*

- blah

Operadores de mutação

- *New-sub-tree*: troca uma sub-árvore de tipo τ por uma nova (gerada aleatoriamente) que não infrinja o tamanho máximo da árvore e que o tipo unifique com o tipo τ .
- *Swap-sub-tree*: Se uma expressão tem dois argumentos cujos tipos unifiquem, ex: $((\text{fun } e1) \ e2)$ tal que o tipo de $e1$ unifica com o tipo de $e2$, então permuta-se (*swap*) a posição das sub-árvores.
- Os operadores de mutação não inserem indivíduos repetidos na população.

Algoritmo Evolucionário

- População inicial pop de indivíduos únicos.
- Os operadores de mutação e *crossover* garantem a unicidade dos indivíduos durante toda a evolução.

Algoritmo evolucionário simplificado:

```
evolution pop 0 = return $ pop
evolution pop n = do
    evaluatedPop <- evalPopulation pop
    popInter <- tourment popFitness
    popCrossed <- crossPop popInter
    popMutated <- mutationPop popCrossed
    r <- evolution popMutated (n - 1)
    return $ r
```

Lidando com Computações Ruins

- Durante a computação dos programas, duas situações ruins podem acontecer: o programa não terminar dentro do limite de chamadas recursivas (provavelmente um *loop* sem fim) ou o programa realizar uma computação sem definição, como tirar a cabeça de uma lista vazia.
- Em ambos os cenários a computação é continuada para um valor arbitrário do tipo retorno esperado e é aplicado uma penalização no *fitness*.

	Int	Bool	Char	[Int]	[Char]
Valor	100	False	'z'	[100]	['z']

Tabela: Valores *default* para computações ruins.

Problema map

Mapear uma função sobre uma lista:

```
map c2i ['a'..'g'] = [0..6]
```

```
T = {l::[a], []::[a], f::G1->G2}
```

```
F = {head::[a]->a, tail::[a]->[a],
      ()::a->[a]->[a], null::[a]->Bool,
      map::(G1->G2)->[G1]->[G2],
      if::Bool->a->a->a, f::G1->G2}
```

```
CS = {if c e1 e2 tal que e1 ≠ e2,
      head l tal que l ≠ [],
      tail l tal que l ≠ [] }
```

Função *fitness* do problema map

$$\begin{aligned}
 f(I, I') = & -2 \cdot |length(I) - length(I')| \\
 & + \sum_{e \in I} 10 \cdot (2^{dist(e, I')}) \\
 & - (10 + 2 \cdot length(I)) \cdot rtError \\
 & - (10 + 2 \cdot length(I)) \cdot reError,
 \end{aligned} \tag{1}$$

onde $dist(e, I')$ é ∞ quando $e \notin I'$, ou caso contrário é a distância de e até o elemento de índice e . $rtError$ e $reError$ representam, respectivamente, erros de computações indefinidas e computações que excederam o limite de chamadas recursivas, e tem valor 1 quando acontecem e 0 caso contrário.

Fitness normalizado!!!

Parâmetros de teste do problema map

- Populações de 250, 500 e 1000 indivíduos.
- Com e sem as condições.
- Taxa de mutação de 4%.
- Taxa de *crossover* de 100%.
- Número de gerações de 200.
- Limite de chamadas recursivas de 9.
- Altura máxima das árvores de 6.
- Geração da população pelo método *grow*.
- 50 execuções de teste.

Convergência do problema map

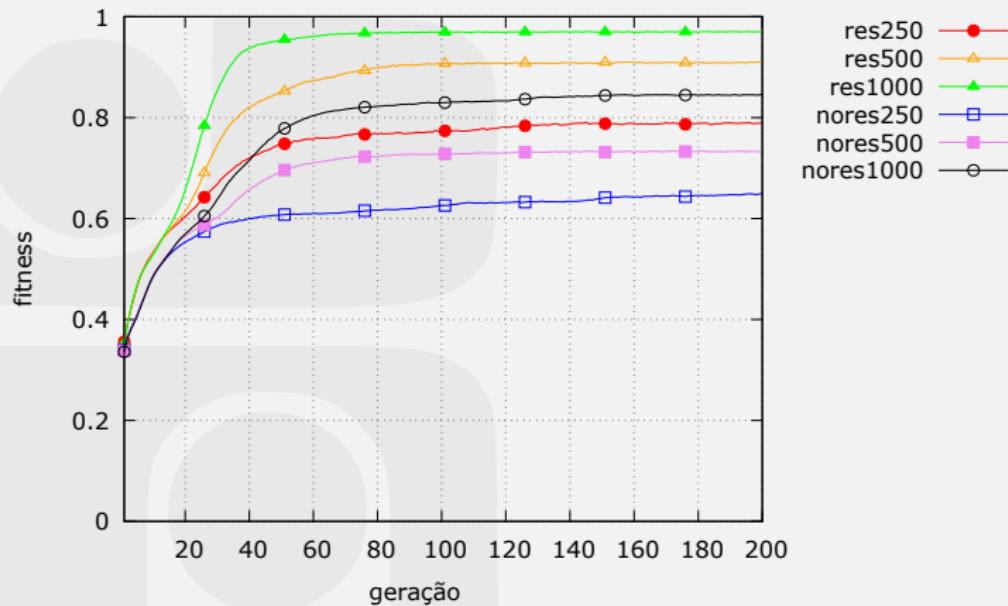


Figura: Média do fitness médio de 50 testes.

Desvio Padrão da convergência

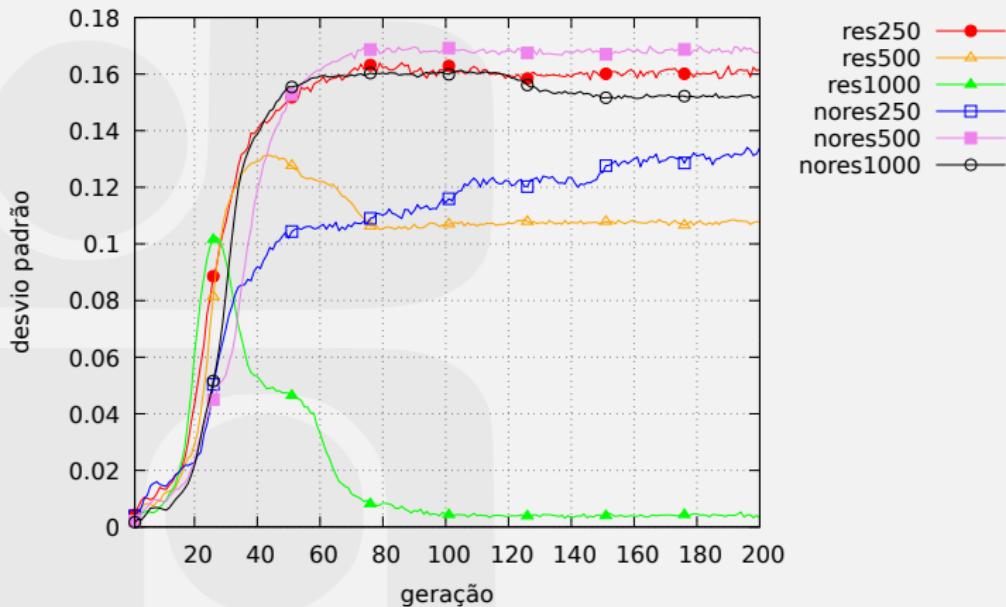


Figura: Desvio padrão do fitness médio de 50 testes.

Boxplot da convergência

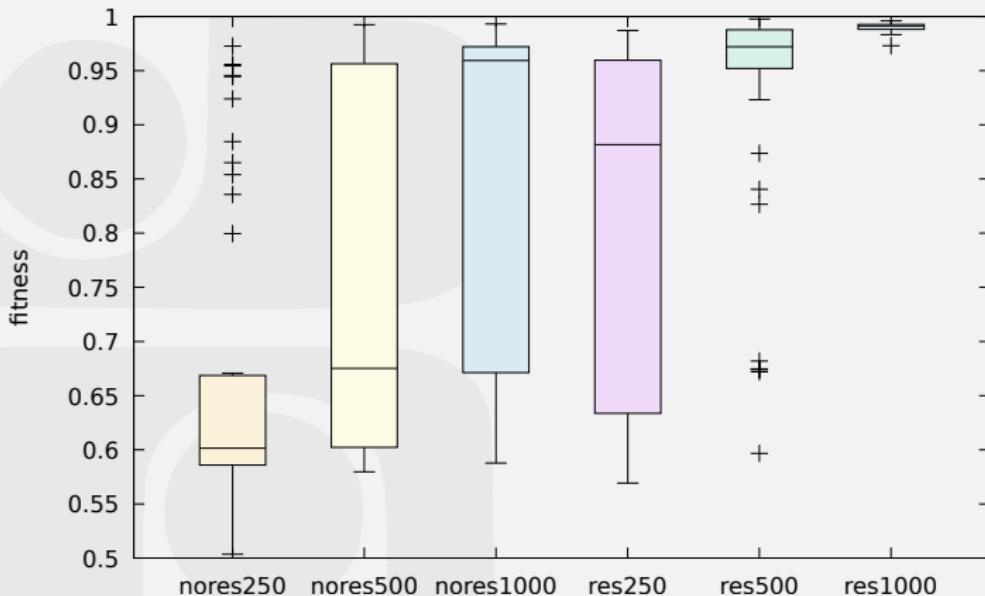


Figura: Boxplot do *fitness* da última geração.



Taxa de acerto

	nores250	nores500	nores100	res250	res500	res1000
Acerto	24%	42%	78%	78%	96%	100%

Tabela: Taxa de acerto da solução ótima.

Exemplo de resultado:

```
map f l = if null l then [] else f (head l) : map f (tail l)
```

Avaliações de *fitness* para encontrar o ótimo

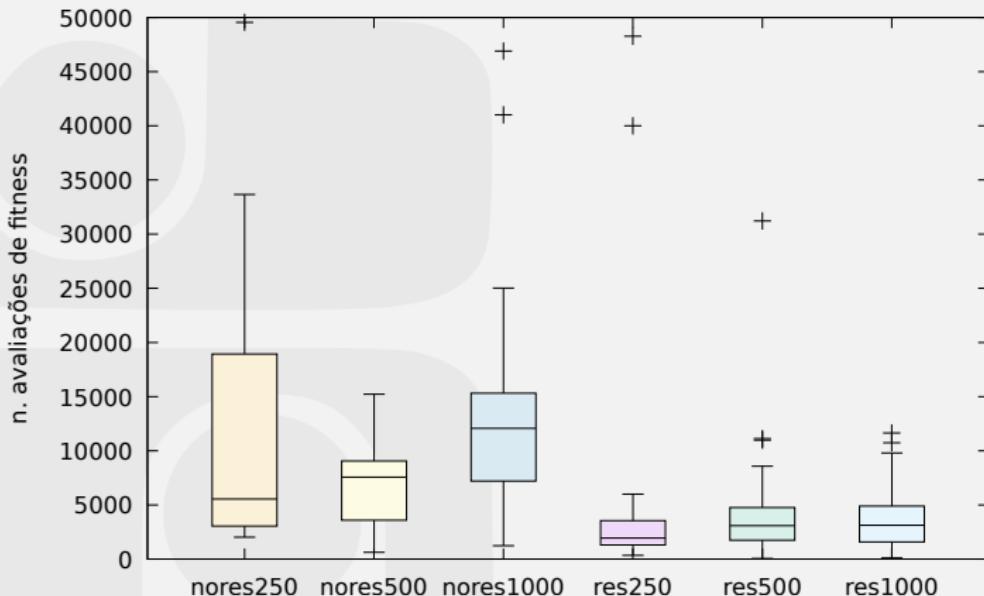


Figura: Boxplot do número de avaliações de *fitness* para encontrar a solução ótima.

Tempo médio de execução

	nores250	nores500	nores1000	res250	res500	res1000
Média	6.01s	13,74s	31.58s	8,28s	20,9s	51,38s

Tabela: Média do tempo de execução.

Considerações finais

- blah