Busca Local Monótona para o Problema da Mochila

Fernando Concatto

Projeto

- Buscou-se implementar uma Busca Local Monótona com política de **Primeira Melhora** para o Problema da Mochila.
- A criação da solução inicial foi realizada aleatoriamente.
- Como possíveis movimentos, considerou-se inserção e troca de itens.
- Adotou-se uma representação binária da solução, onde 0 indica que o item não está presente e 1 determina que o item está na mochila.

Solução inicial

1. Crie uma solução vazia (nenhum item na mochila);
2. Adicione um item aleatório na mochila, desde que o mesmo já não esteja presente;
3. Se a capacidade atual estiver abaixo da máxima, retorne ao passo 2;
4. Caso contrário, desfaça a última inserção realizada;
5. Retorne a solução construída.

Movimentos: inserção e troca

função tentarInserir(solução):

1. **para cada** (índice, valor) **em** *escolherOrdem*(solução):
2. **se** valor = 0: # se o item não está presente
3. melhora = *Δqualidade*(solução, índice, 1)
4. **se** melhora > 0:
5. **retorne** *inserirItem*(solução, índice) # primeira melhora
6. **retorne** solução

Pseudocódigo da função de movimento de inserção

Observações sobre a inserção

- A função *escolherOrdem* define se a iteração será aleatória ou sequencial.
- A função *Δqualidade* calcula a melhora na avaliação da solução ao aplicar uma alteração em um índice (neste caso, uma inserção). Se o peso máximo for excedido, a melhora será negativa.
- A função *inserirItem* constrói e retorna uma nova solução com o item especificado presente na mochila.

```
função tentarTrocar(solução):
1.    para cada (índice, valor) em escolherOrdem(solução):
2.        se valor = 1: # se o item está presente
3.            clone = clonar(solução)
4.            removido = removerItem(clone, índice)
5.            vizinho = tentarInserir(removido)
6.            se avaliar(vizinho) > avaliar(solução):
7.                retorne vizinho
8.    retorne solução
```

Pseudocódigo da função de movimento de troca

Observações sobre a inserção

- A função *removerItem* é equivalente à *inserirItem*, porém realiza uma remoção ao invés de inserção.
- A função *avaliar* computa a qualidade total da solução, dada pelo somatório dos valores dos itens presentes na mochila. Caso o peso máximo seja excedido, a qualidade é dada pela diferença entre o peso total dos itens e a capacidade da mochila (um valor negativo).

Projeto da busca local

Projeto da busca local

1. Crie uma solução inicial;
2. Defina a solução atual como a solução inicial;
3. Tente inserir um item na solução atual utilizando a função *tentarInserir*;
4. Se houve melhora, retorne ao passo 2;
5. Tente trocar um item na solução atual utilizando a função *tentarTrocar;*
6. Se houve melhora, retorne ao passo 4;
7. Retorne a solução atual.

Considerações

- O passo 3 tem como objetivo preencher a mochila ao máximo, pois a solução inicial gerada aleatoriamente ainda pode possuir espaço para inserção de novos itens.
- Quando nenhuma troca oferecer melhora na qualidade, a busca pára, pois um ótimo local foi atingido.

Experimentos e resultados

Configuração dos experimentos

- A implementação foi realizada na linguagem Julia v0.6.3.
- O ambiente de testes foi um computador com sistema operacional Arch Linux, processador Intel i7-8700 @ 3.20GHz e 16 GB de memória RAM.
- Apenas uma *thread* foi utilizada para a coleta de tempo de execução.
- 30 replicações foram executadas para cada combinação de instância e ordem de iteração (sequencial/aleatória).

Complexidade amortizada

- A análise de complexidade amortizada foi realizada para estimar a dificuldade computacional da execução da busca em função do número de itens da instância.
- Realizaram-se regressões lineares utilizando a média de tempo de execução das 30 replicações, considerando tanto uma hipótese polinomial quanto exponencial. A plataforma R foi utilizada para efetuar a construção dos modelos lineares.

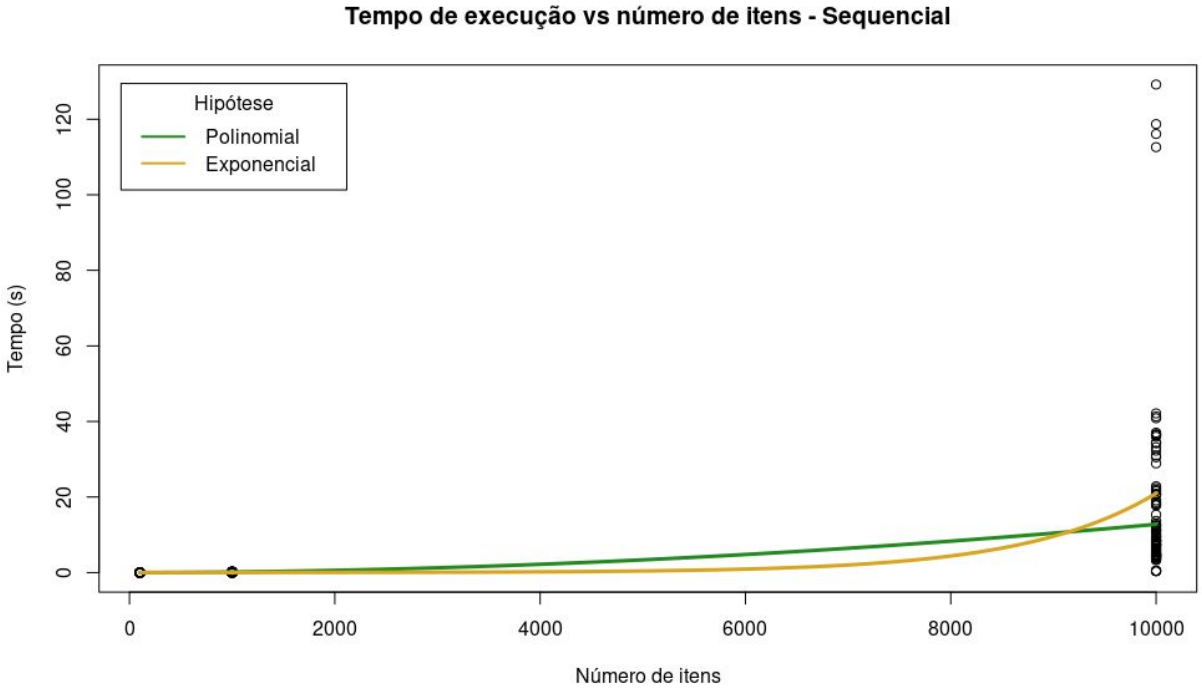
Hipóteses

- Polinomial:
  - `lm(log(tempo) ~ log(itens))`
- Exponencial:
  - `lm(log(tempo) ~ itens)`

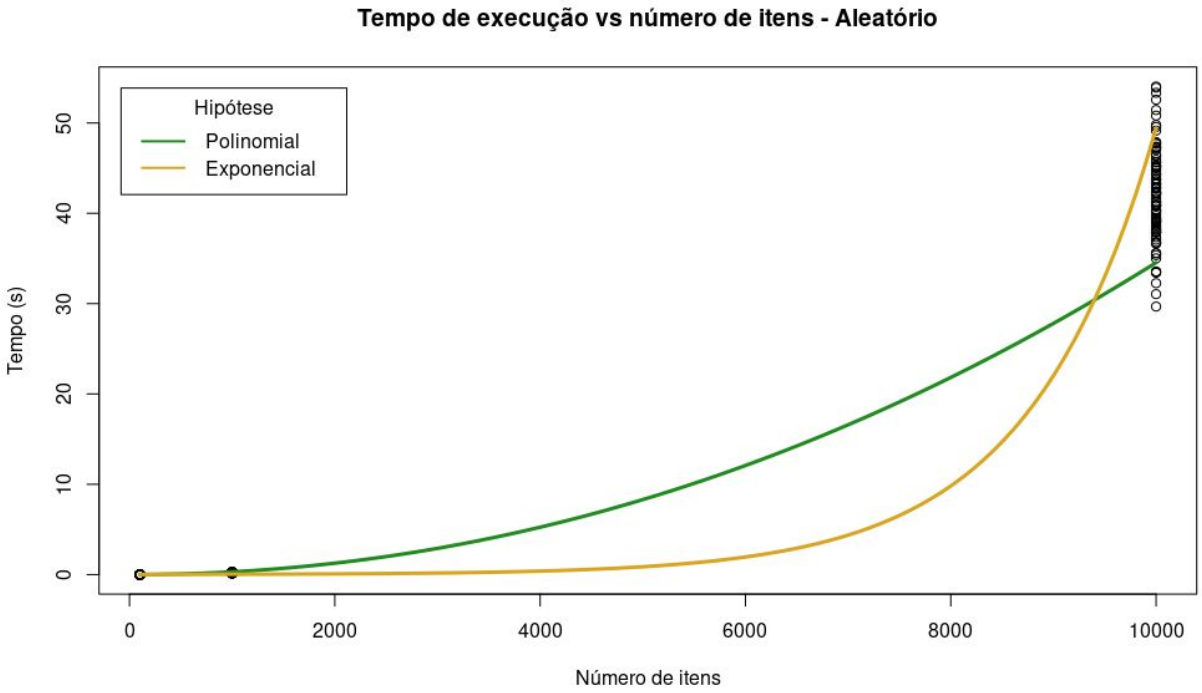
Cálculo da complexidade amortizada

Considerando um modelo linear na forma $f(x) = ax + b$:

- Complexidade amortizada polinomial
  - $f(x) = e^{b} \cdot x^{a}$
- Complexidade amortizada exponencial
  - $f(x) = e^{b} \cdot e^{ax}$



Resultado do cálculo da complexidade amortizada para a ordem de iteração sequencial



Resultado do cálculo da complexidade amortizada para a ordem de iteração aleatória

Iteração sequencial

Parâmetro	Polinomial	Exponencial
a *(slope)*	1.924	0.0007818
b *(intercept)*	-15.174	-4.7751435
p-value	0.09147	0.1895

Iteração aleatória

Parâmetro	Polinomial	Exponencial
a *(slope)*	2.057	0.0008087
b *(intercept)*	-15.404	-4.1866831
p-value	0.04542	0.2355

Valores dos coeficientes e *p-values* para ambas as formas de iteração e ambas as hipóteses.

Iteração sequencial

Instância	Qualidade	Tempo (s)
a100	1199.30	0.003
a1000	10895.87	0.127
a10000	104763.53	37.546
b100	1394.43	0.002
b1000	10059.10	0.031
b10000	101284.27	10.742
c100	1362.87	0.003
c1000	10615.50	0.060
c10000	99893.70	7.22

Iteração aleatória

Instância	Qualidade	Tempo (s)
a100	1740.17	0.003
a1000	17574.87	0.193
a10000	178772.03	41.541
b100	1847.37	0.003
b1000	17628.80	0.209
b10000	178397.40	41.640
c100	1844.40	0.003
c1000	17998.33	0.213
c10000	177726.20	42.812

Médias de qualidade e tempo de execução para as instâncias testadas, para ambos os métodos de iteração.

Considerações finais

Considerações finais

- A estratégia de primeira melhora foi adotada devido ao tempo de execução proibitivo da melhor melhora para instâncias grandes.
- Através dos gráficos, observa-se que ambas as hipóteses se encaixam relativamente bem com os dados coletados, porém a análise de *p-values* determina a aceitação hipótese polinomial.
- Verifica-se que método de iteração sequencial gera soluções de menor qualidade, porém demanda menos tempo de execução.

Considerações finais

- O parâmetro *slope* das regressões corrobora o menor tempo de execução da iteração sequencial.
- Percebe-se uma maior variação nos tempos de execução ao utilizar o método de iteração sequencial.
- A ausência de instâncias com 1000 a 10000 itens dificulta a distinção entre complexidade polinomial e exponencial.