Leonardo Pereira Medeiros

Caixeiro Viajante (Multi-Core)

Introdução

Este projeto consiste no estudo de técnicas mult-core na área de otimização discreta, que se aplica a problemas em que há uma sequência de escolhas, com a característica de que a solução ótima só pode ser encontrada ao enumerar todas as escolhas possíveis. Isto significa que todo algoritmo para sua solução é O(2n) ou pior.

Neste projeto serão utilizados quatro métodos de otimização discreta, com processamento mult-core estas são: enumeração exaustiva, local search, branch and bound e local search com branch and bound.

Enumeração exaustiva: Este método consiste em mapear todos as combinações existentes, e então definir entre estas a que possui o menor custo.

Branch and bound: Este método consiste em realizar uma enumeração exaustiva porém com condições de parada, ou seja, caso a sequência atual satisfaça uma condição a qual seja inviável atingir um custo melhor em comparação com o parâmetro de critério, este ramo é cessado.

Local search: Este método consiste em utilizar uma combinação inicial aleatória, e então aplicar melhorias a cada iteração para melhorar o parâmetro de critério. Embora este seja um método bastante rápido, seu valor final,em algumas execuções, será apenas próximo do valor ideal, pois esse método é afetado por mínimos locais, logo o processo de inicialização aleatória e melhoria e executado diversas vezes, para reduzir esse efeito colateral.

Local search com branch and bound: Este método consiste em dividir o funcionamento geral do algoritmo entre o local search, que faz um processamento rápido para encontrar um valor satisfatório para o parâmetro de critério, e o branch and bound, que em posse desse valor aplica condições de paradas mais restritivas realizando menos operações para encontrar a combinação ideal.

Ao final deste relatório estes métodos, assim como a forma sequencial serão comparados entre si, a fim de concretizar possíveis vantagens e desvantagens e definir qual método gera o melhor desempenho.

O Problema

Para aplicar os conceitos de otimização discreta foi implementado o problema do Caixeiro Viajante, que consiste em determinar a menor rota para percorrer uma série de cidades(sem repetições) e retornar a cidade de origem.

Este problema foi inspirado na necessidade dos vendedores em realizar entregas em diversos locais, percorrendo o menor caminho possível, e consequentemente reduzindo o tempo necessário para a viagem e os possíveis custos com transporte e combustível.

Estrutura Do projeto

O projeto está estruturado de forma que cada método implementado possui seu respectivo arquivo.cpp, esses arquivos possuem a função main, responsável por controlar todo o fluxo do código, lendo um arquivo de entrada com os conjuntos de pontos x e y, e lançando tasks para aplicar o método de otimização discreta de forma paralela sobre esse conjunto. Também há duas funções auxiliares: dist, que calcula a distância entre dois pontos, e path dist, que calcula a distância total de uma sequência de pontos.

Testes

Os inputs selecionados para testar o desempenho da aplicação de técnicas de otimização discreta foram formulados de forma a tentar levar ao máximo o processamento da cpu. Como o programa exige mais processamento quando há muitos pontos que o caixeiro deve percorrer, esse valor foi testado de forma crescente, e com grandes intervalos de iterações máximas, a fim de tentar tornar mais visível o ganho de desempenho ao aplicar essa técnica para diferentes demandas de processamento.

Os seguintes inputs foram testados:

- Input1: N = 11
- Input2: N = 12
- Input3: N = 13
- Input4: N = 14

Em que N representa o número de pontos no sistema.

Definido como variável de desempenho o tempo de execução do programa, foi utilizada a biblioteca Chrono para mensurar esta informação, biblioteca esta que fornece funções de alta resolução baseada em clock.

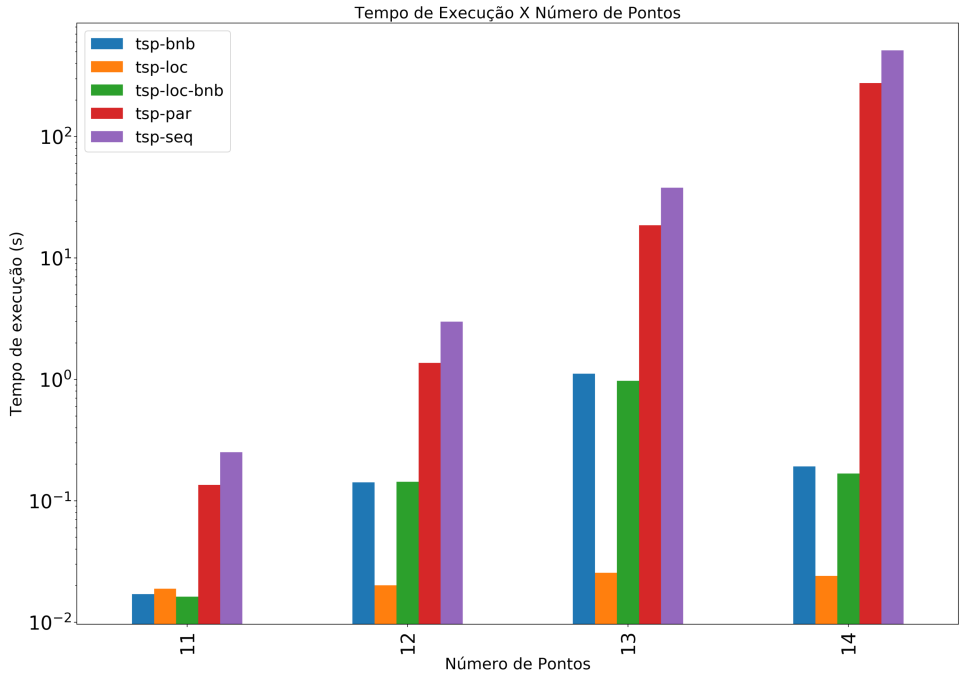
Como deseja-se medir apenas o desempenho do algoritmo do caixeiro viajante a medição não leva em conta o tempo gasto na leitura do arquivo de input e outros processos similares.

Resultados

```
##import dependences
import matplotlib.pyplot as plt
import pandas as pd
import subprocess
import sys
from IPython.display import display

##files
dir_name = "../build/"
files    = ["tsp-seq", "tsp-par", "tsp-loc", "tsp-bnb", "tsp-loc-bnb"]
inputs   = ["input1", "input2", "input3", "input4"]

n_sizes   = []
dic       = {}

## Run files and storage output
for input in inputs:
    dic[input] = {}

    for f in files:
        command = dir_name + f + " < " + input
        stout   = subprocess.check_output(command, stderr=subprocess.STDOUT, shell=True).decode(sys.stdout
.encoding)
        dic[input][f]  = float(stout.split("\n")[-2].split(":")[1])

    with open(input, 'r') as f:
        nrect = f.read().split("\n")[0]

    n_sizes.append(int(nrect))

## Generate Dataframe
s0 = [dic["input1"]["tsp-seq"], dic["input2"]["tsp-seq"], dic["input3"]["tsp-seq"], dic["input4"]["tsp-seq"
]]
s1 = [dic["input1"]["tsp-par"], dic["input2"]["tsp-par"], dic["input3"]["tsp-par"], dic["input4"]["tsp-par"
]]
s2 = [dic["input1"]["tsp-loc"], dic["input2"]["tsp-loc"], dic["input3"]["tsp-loc"], dic["input4"]["tsp-loc"
]]
s3 = [dic["input1"]["tsp-bnb"], dic["input2"]["tsp-bnb"], dic["input3"]["tsp-bnb"], dic["input4"]["tsp-bnb"
]]
s4 = [dic["input1"]["tsp-loc-bnb"], dic["input2"]["tsp-loc-bnb"], dic["input3"]["tsp-loc-bnb"], dic["input4
"]["tsp-loc-bnb"]]

df = pd.DataFrame({"tsp-seq":s0, "tsp-par":s1,
                   "tsp-loc":s2, "tsp-bnb":s3, "tsp-loc-bnb":s4}, index = n_sizes)

display(df)

df.plot.bar(figsize = (20,14), fontsize=24, logy = True)
plt.legend(prop={'size': 20})
plt.title("Tempo de Execução X Número de Pontos", fontsize = 20)
plt.xlabel("Número de Pontos", fontsize = 20)
plt.ylabel("Tempo de execução (s)", fontsize = 20)
```

	tsp-bnb	tsp-loc	tsp-loc-bnb	tsp-par	tsp-seq
11	0.017008	0.018890	0.016185	0.135461	0.251861
12	0.142284	0.020187	0.143236	1.369950	2.993210
13	1.111800	0.025502	0.975416	18.599600	37.956000
14	0.191836	0.024107	0.167342	276.442000	514.919000

```
Text(0, 0.5, 'Tempo de execução (s)')
```



Conclusão

Como pode-se observar para simulações com uma quantidade consideravelmente grande de pontos obtém-se a seguinte ordem de desempenho:

tsp-loc > tsp-loc-bnb > tsp-bnb > tsp-par > tsp-seq

A ordem de desempenho apresenta-se como o esperado, pode-se pensar nessa sequência como a inserção de uma nova otimização no método anterior, o método sequencial utiliza o algoritmo de enumeração exaustiva, o método paralelo adiciona a paralelização, o método de branch and bound adiciona uma condição de parada, o método combinado adiciona um processamento inicial muito rápido utilizando local search para encontrar um valor de condição mais restritivo. Já o local search não segue a estrutura dos métodos anteriores, ao invés de fazer enumerações este método realiza combinações iniciais aleatórias com melhorias iterativas, realizando poucas combinações, mas deve-se lembrar que este método é o menos eficiente para encontrar o valor ideal.

Embora a ordem faça sentido, esperava-se que o método combinado dos algoritmos local search e branch and bound tivesse um desempenho numérico muito melhor comparado com o método branch and bound, visto que no método combinado há um processamento inicial com o local-search, que rapidamente define um valor de critério inicial satisfatório, o processamento posterior do branch and bound para encontrar o valor ideal seria bem mais suave comparado com o método que usa apenas o branch and bound, ao verificar menos combinações devido a condição de parada mais restritiva. Talvez essa diferença numérica se torne mais evidente para entradas com mais números de pontos, em que o processamento inicial resulta-se em ganhos maiores.

Já para simulações com poucos pontos, em que não é necessário muito processamento o local search possui um desempenho ruim, isso é justificado pelo fato de que esse método itera sobre várias combinações aleatórias fazendo um overload de combinações para casos simples, já o método branch and bound apresenta o melhor desempenho ao fazer paradas na enumeração exaustiva.

Especificações da Máquina utilizada

Arquitetura: x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes: Little Endian
CPU(s): 4
Lista de CPU(s) on-line: 0-3

Thread(s) per núcleo: 2
Núcleo(s) por soquete: 2
Soquete(s): 1
Nó(s) de NUMA: 1
ID de fornecedor: GenuineIntel
Família da CPU: 6
Modelo: 61
Nome do modelo: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
Step: 4
CPU MHz: 2430.779
CPU MHz máx.: 3000,0000
CPU MHz mín.: 500,0000
BogoMIPS: 4788.89
Virtualização: VT-x
cache de L1d: 32K
cache de L1i: 32K
cache de L2: 256K
cache de L3: 4096K
CPU(s) de nó0 NUMA: 0-3

Published from doc.pmd using Pweave 0.30.3 on 02-10-2019.