

# Caixeiro viajante, numa abordagem HPC

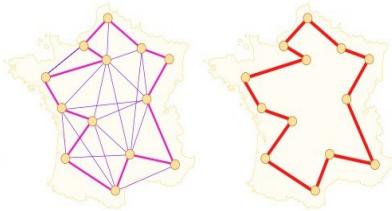
Humberto Vaz e João Dias

**Abstract**—Neste documento relatamos o trabalho realizado pelos autores com o intuito de atingir resultados relativos à performance de diferentes implementações para a resolução do problema do "Caixeiro Viajante". É de referir que os resultados descritos são avaliados em dois aspectos (i) tempo de execução e (ii) distância calculada. São então avaliadas 4 soluções para o problema em mãos (i) implementação Monte Carlo pura (MC), (ii) implementação que recorre à técnica meta-heurística probabilística Simulated Annealing (SA), (iii) algoritmo Greedy (GD) e (iv) o Genetic Algorithm (GA). As soluções serão implementadas em Python.

**Keywords** — Problema do Caixeiro Viajante, NP, MC, SA, GD, GA, Python, Monte Carlo, Genetic Algorithm, Simulated Annealing, Greedy

## 1. INTRODUÇÃO

O problema do caixeiro viajante é conhecido como um problema NP-difícil, ou seja, um problema polinomial não determinístico listado informalmente como "tão difícil quanto os problemas mais difíceis em NP". O problema consiste na procura do caminho mais curto que visite todas as cidades de um determinado conjunto, retornando, por fim, à cidade de origem. Um exemplo prático disso pode ser visto na seguinte imagem:



Nestas imagens temos um exemplo prático de um problema do Caixeiro Viajante, sendo que cada vértice representa uma cidade, as arestas de cor rosa são indicativas de todas as opções de trajeto. Na segunda imagem vemos então a solução, isto é, o caminho mais curto.

Este problema é ainda bastante relevante pela sua natureza permitindo servir de *benchmark* na comparação de algoritmos e bastante recorrente no campo da computação, sendo que, apesar da sua "idade avançada", ainda se discutem melhores implementações de solução.

## 2. CONSIDERAÇÕES INICIAIS

### 2.1. Caracterização do Hardware

Para este projeto recorremos a duas máquinas, ambos nodos do Cluster SeARCH **781** e **662**. Essas máquinas são devidamente caracterizadas na tabela abaixo:

	<b>662</b>	<b>781</b>
<b>Fabricante</b>	Intel	Intel
<b>Microarquitetura CPU</b>	Ivy Bridge	Broadwell
<b># Cores</b>	24 fis; 48 log	32 fis; 64 log
<b>Freq. Relogio</b>	2.4 Ghz	2.1 Ghz
<b>L1</b>	d:32K; i:32K	d:32K; i:32K
<b>L2</b>	256K	256K
<b>L3</b>	30720K	40960K
<b>RAM</b>	64G	256G

Estas escolhas são justificadas com o intuito de ter uma base de testes mais abrangente usando então duas microarquiteturas diferentes.

### 2.2. Estruturas de dados

Para a formulação do problema em mãos os dados de input serão:

- Conjunto de cidades
- Distância específica a cada par de cidades

É importante destacar que o problema está diretamente dependente da quantidade de cidades que disponibilizamos como variável.

O conjunto de cidades, isto é, o input, deste projeto é gerado aleatoriamente com as coordenadas que os definem ( $x, y$ ) limitadas por um quadrado de lado 10.

```
% generates the position of each town in a square of side 10...
x=10*rand(1,n); y=10*rand(1,n);
```

Fig. 1. Código disponibilizado no contexto do problema

tendo a sua tradução para Python o seguinte formato:

```
#Gerar coordenadas das cidades
def random_numbers(num, lower, upper):
    return [10 * random() for i in range(num)]
```

um exemplo da geração das cidades será então:

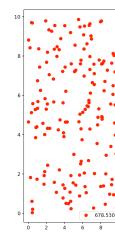


Fig. 2. Matriz representativa da geração de 160 cidades

Será a partir daqui então computada a distância para cada par de cidades:

```

for i=1:n
    for j=1:n
        D(i,j)=sqrt((x(i)-x(j))^2+(y(i)-y(j))^2);
    end
end

```

Fig. 3. Código disponibilizado no contexto do problema

posteriormente traduzido para:

```

D = [[0 for x in range(n)] for y in range(n)]
for i in range(0, n):
    for j in range(0, n):
        #and computes the distances between them
        D[i][j] = math.sqrt(math.pow(x[i] - x[j], 2) +
                            math.pow(y[i] - y[j], 2))

```

Como é perceptível para o leitor, será então a matriz **D** a estrutura que mapeia os valores das distâncias, sendo **n**,número de cidades, representativo também do lado da matriz.

Através desta análise inicial ao problema identificamos a ideologia de teste a realizar:

Dados que fiquem inseridos na :

- Cache L1;
- Cache L2;
- Cache L3;
- RAM;

Tendo estes requisitos em mente chegamos a uma formula para calcular a dimensão do problema de interesse:

$$n \leq \sqrt{\frac{\text{Tamanho de Cache}}{\text{sizeof(Double)}}}, n \in \mathbb{N}$$

Outra precaução que tivemos foi ainda a seleção de um **n** que ocupasse exatamente numa linha de cache, prevenindo assim custos desnecessários no carregamento de "lixo". Para tal, tendo em conta que nas máquinas utilizadas, uma linha de cache tem **64B** (8 Doubles) chegamos outra formula restringente para **n**:

$$\text{Linha de Cache} = \frac{n * n}{8}, \text{Linha de Cache} \in \mathbb{N}$$

Seguindo estas *guidelines* chegamos então às seguintes estruturas de dados:

<b>L1</b>	48
<b>L2</b>	160
<b>L3</b>	1600
<b>RAM</b>	3200

A escolha de *Double* como formato para armazenamento de dados deve-se ao facto de conseguirmos uma maior quantidade de números representativos, algo de interesse para o problema, é de denotar ainda que o tipo de dados *Float Python* são tratados como *Doubles* no que diz respeito à mantissa e o seu nível de precisão, sendo que, para o leitor este facto irá fazer diferença para uma análise mais detalhada.

### 3. SOLUÇÕES IMPLEMENTADAS

#### 3.1. MC - Monte Carlo puro

A implementação deste algoritmo é inicializada com a produção de uma solução (*pseudo-aleatória*, pois os algoritmos são na realidade deterministas e dependentes das condições iniciais, e em particular da semente (*seed*)). No seu "tempo de vida" o algoritmo vai então proceder à troca de ordem nas cidades visitadas por iteração, sendo que se a solução encontrada nessa iteração possuir uma distância inferior será esse resultado usado como base para a próxima iteração. Este processo é então repetido e termina quando se passarem um certo número de iterações sem um melhor resultado.

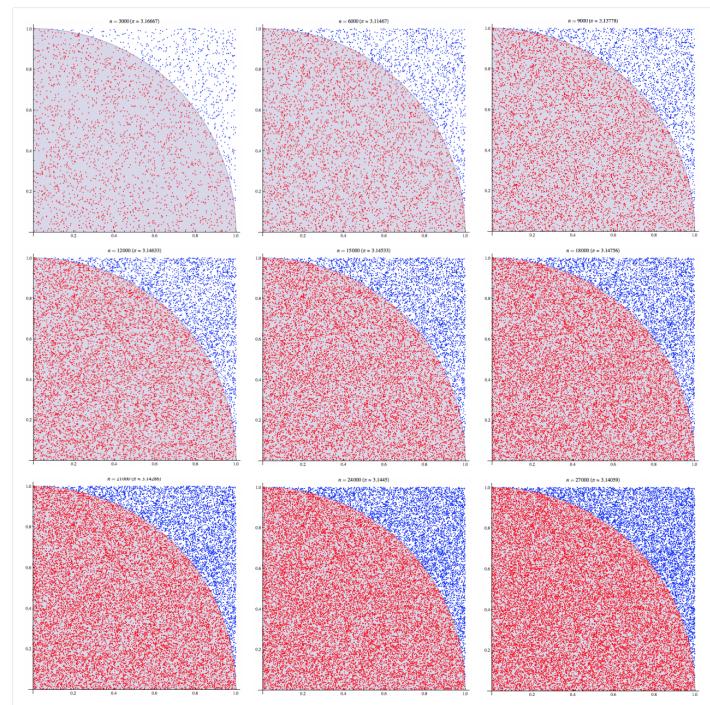
Consideremos o exemplo do cálculo de  $\pi$ , sabendo que obtemos a área de uma circunferência pela seguinte formula:

$$\pi = r^2$$

e que a área do quadrado é dada por:

$$Aq = l^2$$

se circunscrevermos uma circunferência de raio de  $r$  num quadrado de lado  $2r$ , pela relação entre ambas as fórmulas conseguimos extrapolar, através do algoritmo de *Monte Carlo* o valor de  $\pi$ .



Em suma, através da criação deste quadrado e círculo, e posterior geração de pontos internos ao quadrado somos capazes de realizar uma análise da distribuição (interna e externa ao círculo). Permitindo então calcular um valor para  $\pi$ , aumentando o valor de pontos gerados, teremos também uma aproximação superior à solução ótima.

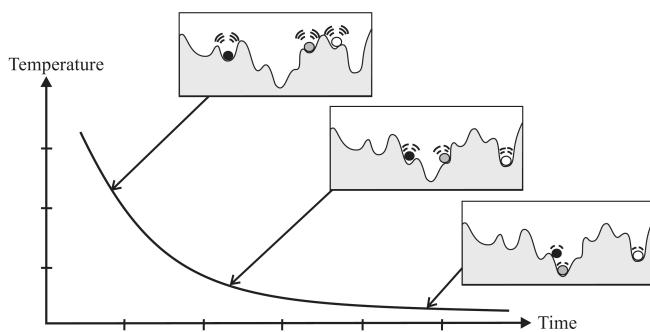
### 3.2. SA - Simulated Annealing

Simulated annealing (SA) é uma técnica probabilística meta-heurística que se baseia no *Monte Carlo*, este processo é vulgarmente associado ao processo metalúrgico de recozimento (annealing) que consiste em arrefecer e posteriormente aquecer o metal alterando assim as suas propriedades físicas. O algoritmo segue então esta ideologia, sendo representativo do metal o sistema que pretendemos arrefecer em ordem de obter o melhor resultado possível, ou até mesmo a solução ótima.

Com o objetivo de arrefecer o sistema iniciar a computação com uma variável representativa da temperatura, inicializamos essa variável com um valor alto

```
T = 1 # initial "temperature"
```

que ao longo das iterações decresce. Esta variável serve então como um regulador do problema visto que quanto maior for mais provável é para o sistema aceitar novas soluções que são "piores" (caminho mais longo) que a atual.



Como é possível observar na imagem superior o que ocorre é que aceitação de um resultado "pior" permite ao algoritmo não ficar estagnado num mínimo local, tendo isso em mente, esta abordagem da temperatura permite uma maior mobilidade na procura de melhores resultados globais pelas vizinhanças. À medida que as iterações se dão e o sistema "arrefece" a variável de temperatura diminui, a solução converge para o mínimo e torna-se cada vez mais complicado a aceitação de novas soluções porque o sistema se aproxima também da sua solução ótima. É nestas condições, à medida que nos aproximamos da solução ótima, não tendo novas soluções por um determinado conjunto de iterações ou a temperatura se aproxima demasiado de 0 que o algoritmo pára.

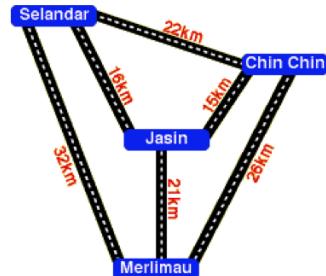
### 3.3. GR - Greedy

Este algoritmo é vulgarmente conhecido para resolver o problema do caixeiro viajante. A sua difusão está diretamente relacionada com a sua fácil compreensão e implementação.

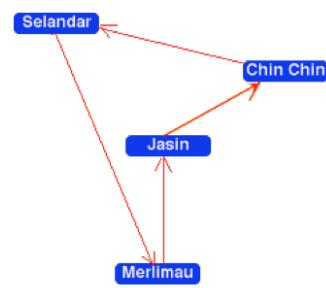
O objetivo fulcral deste algoritmo será então uma otimização local, isto é, o algoritmo escolhe o próximo nó que lhe parece ser a melhor solução estando (isto é, aquele com menor distância) no nó atual ignorando assim o panorama global de todo o problema. Daí surgir o nome "Greedy"

(ganancioso/guloso), porque tal como alguém pretensamente ganancioso escolherá sempre o caminho aparentemente melhor sem ter qualquer consideração nas possíveis implicações que isso terá no futuro.

Um caso prático deste algoritmo é o seguinte:

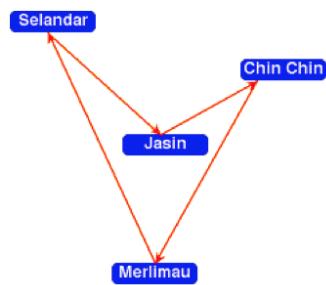


Iniciando o problema na cidade de Jasin obtemos a seguinte solução:



$$\text{distância total} = 15 + 22 + 32 + 26 = 95 \text{ Km}$$

Porém caso iniciemos o problema em Selendar obtemos:

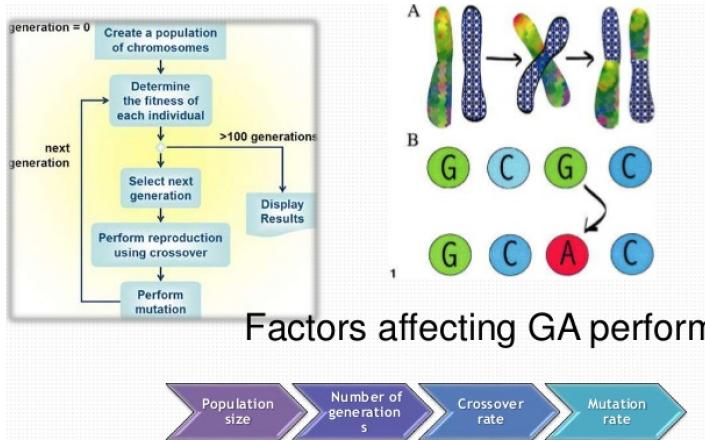


$$\text{distância total} = 16 + 15 + 26 + 32 = 89 \text{ Km}$$

Percebemos então que este algoritmo está dependente da cidade da qual iniciamos a viagem. Sendo este o objeto que devemos aleatoriamente alterar.

### 3.4. GA - Genetic Algorithm

O Genetic Algorithm (GA) é um algoritmo meta-heurístico que se inspira no processo de seleção natural e que pertence à classe dos algoritmos evolucionários (EA). Este algoritmo é usualmente utilizado para produzir soluções de elevada qualidade em problemas de procura apoianto-se em características biológicas tais como a *mutação*, *seleção* e *crossover*.



Como é possível perceber pela imagem acima, o algoritmo inicia-se com uma população de indivíduos (soluções). E por cada geração (passo) a solução é avaliada. As melhores soluções irão sofrer o dito *crossover* (recombinação) e posteriormente uma *mutação* voltando ao inicio do processo até que chegemos a um determinado nível de "gerações", isto é, passos. Sabemos ainda que os parâmetros que irão a performance do algoritmo serão o:

- tamanho da população;
- número de gerações (passos);
- taxa de crossover;
- taxa de mutação;

#### 4. ANÁLISE DE RESULTADOS

##### 4.1. Consistência das soluções

O que acontece nestes algoritmos é a paralelização que normalmente é iniciar sobre uma cidade aleatória e depois de um determinado número de passos fazer uma comparação das soluções entre todos os processos repetindo continuamente esta ação.

Para este tipo de algoritmos é de conhecimento geral uma certa dependência da solução da qual partimos, assim sendo um maior número de processos em paralelo terá por consequência uma maior probabilidade de chegar a uma solução melhor que apenas um a correr de forma sequencial.

Usando *Python*, mais concretamente o objeto *Parallel* :

```
from joblib import Parallel
```

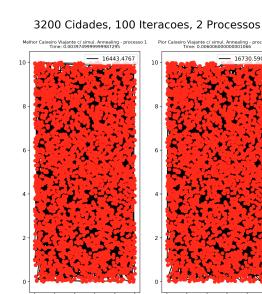
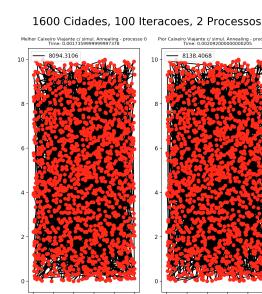
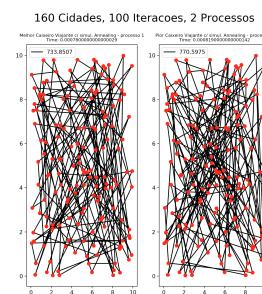
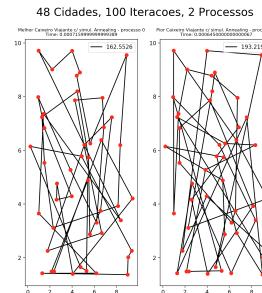
Iremos conseguir fazer uso do elevado número de cores das nossas máquinas. A nossa escolha do número de processos vai então variar conforme as máquinas usadas. Como utilizamos o nó **662** e o **781** temos para o primeiro caso 24 cores físicos e 48 lógicos (usando *hyperthreading*) e no **781** 32 físicos e 64 lógicos, iremos limitar as nossas escolhas pelo de menor capacidade, introduzindo variação pela escolha de número de processos:

- 2
- 4
- 8
- 16

- 32
- 48

Passamos então para a confirmação de que os algoritmos produzem resultados esperados, posteriormente numa segunda fase passaremos para a comparação das duas máquinas utilizadas.

Primeira fase:



Algumas dificuldades em produzir as imagens finais no *Cluster SeARCH* obrigaram-nos a produzir estas imagens num computador pessoal, algo que não afeta o compreensão e o propósito desta ação.

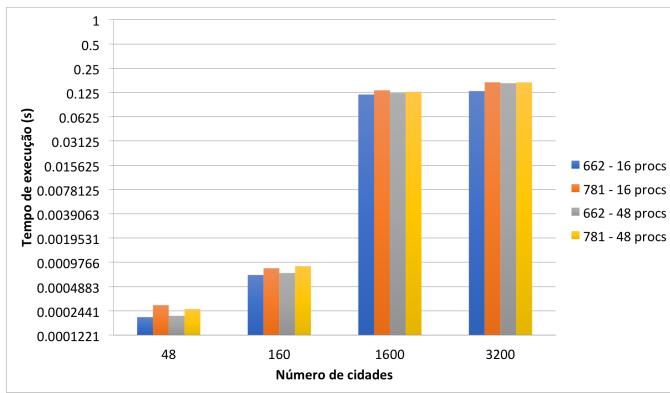
Em segundo lugar passamos à comparação dos resultados finais das máquinas.

Para tal, a abordagem ao problema é relativamente direta. Escolhemos um valor do leque dos resultados obtidos pelo algoritmo Greedy, a partir daí faz-se uma comparação com o tempo necessário por ambas as máquinas para atingir esse resultado. O mesmo pode aplicar-se aos outros algoritmos, escolhemos o Greedy porque é o que apresenta tempos superiores e ser dos processos com maior trabalho computacional, porém nada nos impedia de realizar o mesmo processo com qualquer um dos outros. O próximo gráfico relata os resultados obtidos do nó **662** vs **781**:

Fazemos uma comparação direta com 16 e 48 processos: para simular a utilização de

- apenas cores físicos;
- cores físicos e lógicos;

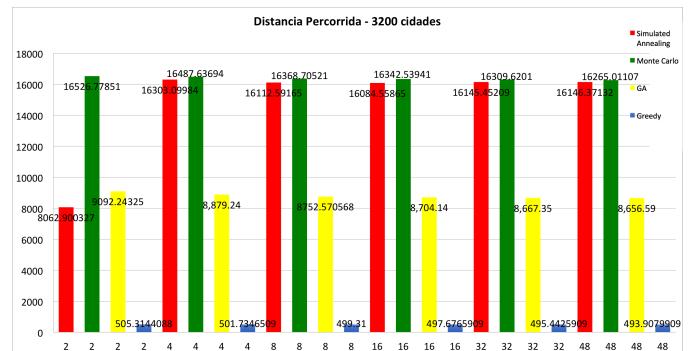
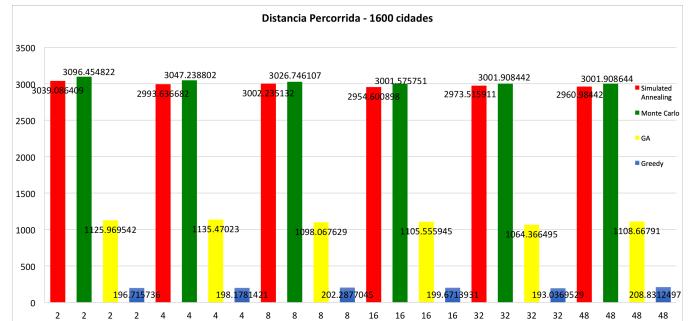
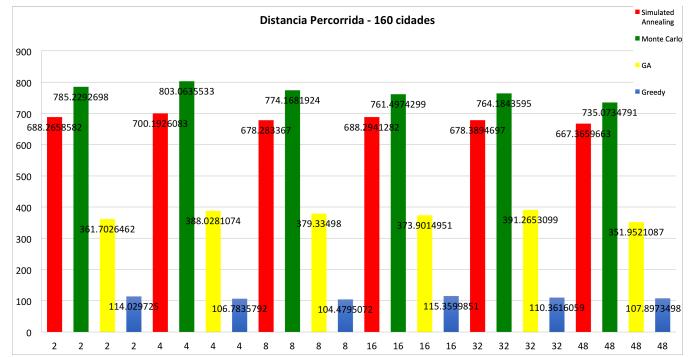
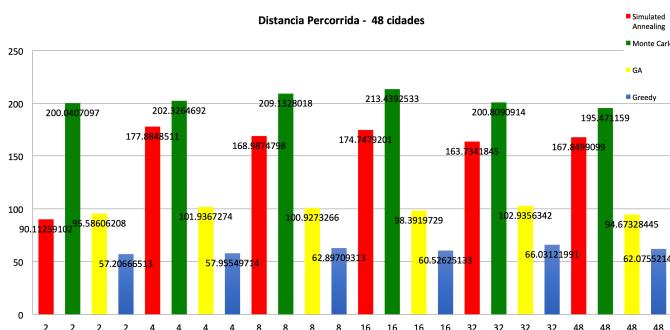
Pela natureza do problema só comparemos os tempos retirados das soluções.



Apesar de não ser escandaloso visionando estes resultados chegamos à conclusão que apesar de o nodo **781** apesar de ser mais recente não consegue superar o **662** que possui menos processadores mas mais potentes, assim sendo fará mais sentido proceder a mais medições utilizando o nodo **662**.

#### 4.2. Interpretação de resultados

Os próximos gráficos retratam resultados obtidos no nodo 662 para o número de passos estático (100):

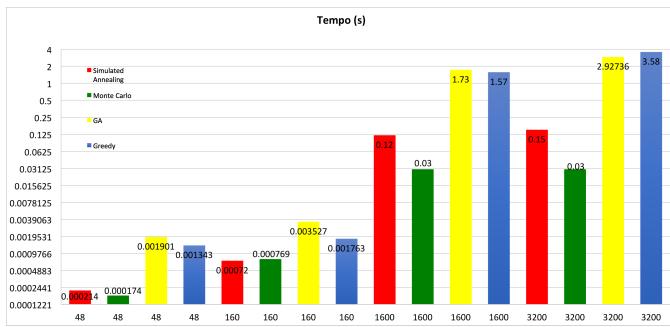


Uma análise inicial permite afirmar que o algoritmo Greedy produz melhores resultados, algo que seria de esperar tendo em conta o funcionamento do mesmo. Nos piores casos temos então o Monte Carlo (MC), o que também não é uma surpresa pois tanto o Simulated annealing como o Genetic Algorithm são expansões baseadas no MC, temos então uma disputa entre o SA e GA, ganha pelo Genetic Algorithm.

Estes resultados devem-se a diversos fatores, entre os quais destacamos no caso do Greedy que vai apresentar um tempo elevado comparado com os restantes pois se trata de um algoritmo computacionalmente extenuante pois, sendo a aleatoriedade apenas inserida na escolha da cidade inicial da viagem todas as restantes escolhas serão matemáticas, havendo a necessidade de comparar a distância entre a cidade presencial e a futura para todo o leque de opções (conjunto de cidades não visitadas), isto obviamente acarretará custos. Uma outra coisa que convém constatar é que este algoritmo não é propriamente conhecido por dar o resultado ótimo, como referido na introdução a esse algoritmo, a sua forma gananciosa de aproximação ao problema tem as suas consequências e à medida que o número de cidades aumenta, aumenta também

a propensão ao erro. No caso do Monte Carlo puro, a sua forma de utilização é praticamente aleatória nas suas escolhas, e será através de tentativa erro ou mesmo pura sorte que o melhor resultado será atingido. Já no caso do SA, o resultado anterior possui uma relação para com a solução e assim sucessivamente por passo. Possuindo ambos as mesmas raízes como é natural ambos os algoritmos conseguem esquivar-se de ficar estagnados num mínimo local, porém o algoritmo *Simulated Annealing* prova-se mais eficiente nesse sentido. Outro algoritmo também baseado no algoritmo *Monte Carlo* é o Genetic Algorithm,GA , que dos três (SA, MC e GA) possui os melhores resultados. Isto acontece também porque em vez de promover uma alteração, promove uma "população" delas, isto é, define vários pontos iniciais em simultâneo, o que, como é natural, com este método heurístico seja mais provável que uma dessas soluções seja melhor que uma opção que só tem um ponto inicial aleatório.

Contudo, estes melhores resultados apresentam um custo que se retrata nos tempos de execução, como é visível nos gráficos abaixo.



Como já referido anteriormente pela natureza do algoritmo *Greedy*, este vai causar que a intensidade computacional se traduza em tempos de execução mais elevados. Verificamos a oportunidade de paralelismo para iniciar o algoritmo em diferentes cidades, ou ainda dentro de um conjunto de cidades escolher algumas e correr separadamente o algoritmo para cada rota, terminando quando não sobrar mais nenhuma cidade e unindo as diferentes rotas criadas, esta opção já não é tão simples pois existem dependência de dados que serviriam como *bottleneck* para toda a operação. Já no caso do *Genetic Algorithm* o atraso é justificado pela necessidade do GA de tem de verificar a *fitness* de toda a população e posterior escolha do elemento inicial. Porém uma constante é que a utilização de um maior número de processos por norma implica uma maior probabilidade de melhores resultados, isto é subentendido como uma oportunidade para o paralelismo.

Podemos facilmente observar que o SA é mais lento que o MC de uma forma geral, o que é de esperar porque em cada iteração tenta encontrar um melhor resultado, isto pressupõe que no final SA e MC converjam para o mesmo resultado só que SA o faça em menos passos.

Também conseguimos entender o porquê do GA necessitar de uma menor quantidade de passos que o SA e que o MC, mas esta superioridade advém de custos, esses são retratados numa maior intensidade computacional (medição do fitness da população, crossover e mutação).

Uma das formas de reduzir este tempo de execução é fazendo "batota". Com isto queremos sugerir que uma diminuição da população sobre a qual por passo medimos o fitness, ou dentro deste grupo apenas selecionar uma aleatoriamente, porém, ambas as opções **trocaram a qualidade dos resultados pela velocidade**, algo que não consideramos ser benéfico para a resolução do problema.

Com a base de comparação dos algoritmos realizada passamos agora para a variação dos parâmetros heurísticos.

Dentro dos algoritmos apresentados escolhemos o SA para variar esses valores.

- Passos
- Processos
- Temperatura

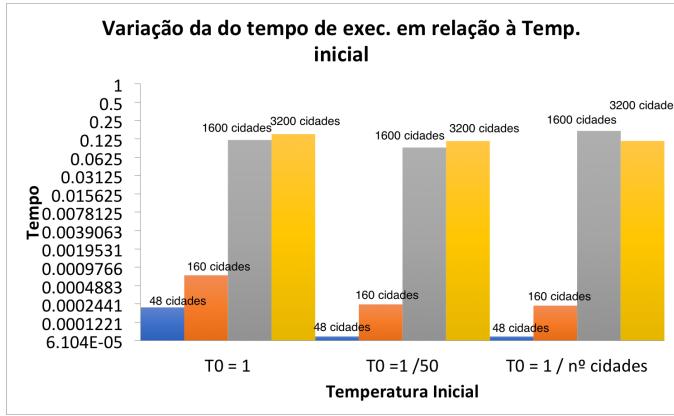
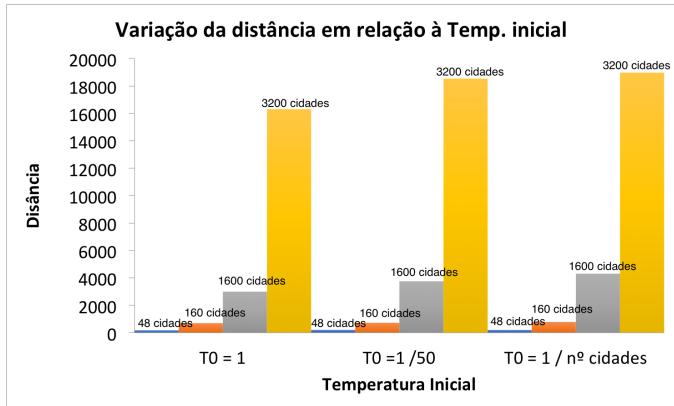
Como era de esperar um aumento no número de passos vai provocar um aumento do tempo de execução. Apesar de fazermos muitas mais iterações, notamos que não existe nenhum ganho especial nos inputs pequenos, talvez porque já exista uma convergência para estes casos considerando assim que o número de 100 passos é perfeitamente válido e que um número superior de passos só será significativo para um conjunto de cidades maiores. Assim sendo concluímos que este algoritmo está preparado para lidar com "poucos" passos e conseguir um resultado aceitável, sendo que um número alto de iterações vai pesar no tempo de execução.

Relativamente ao número de processos, analisando a tabela das distâncias, onde estão presentes todas as distâncias, notamos que existe um decréscimo parcial nos valores das distâncias, o que, como já previamente referido neste relatório é normal, tendo em conta que havendo um maior probabilidade em encontrar uma melhor solução, porém os resultados obtidos não são tão bons quanto gostaríamos.

Relembrando agora o algoritmo SA, podemos afirmar que a temperatura inicial tem um papel fulcral no algoritmo, ao aproximar-se muito de 0, termina. O que significa que apenas aumentar o número de iterações não chega. Isto também porque à medida que a temperatura diminui também o algoritmo tem maior dificuldade em aceitar novas soluções. Concluímos assim que fará sentido usar uma temperatura inicial maior para um conjunto de cidades superiores, dando assim aso a um maior banda de procura.

O gráfico abaixo refere-se à troca de parâmetros de temperatura inicial no algoritmo SA medindo ambos os valores de sucesso:

- tempo de execução;
- distância final;



Verificamos então que a inicialização de uma temperatura inicial alta vai provocar um maior tempo de execução mas também melhores resultados. Porém não consideramos que isso seja algo relevante visto que os restantes algoritmos que apresentam melhores resultados (Greedy e GA) apresentam também tempos de execução muito superiores, superiores ao ponto de podermos "ignorar" o overhead da temperatura alta considerando os benefícios que isso tem para o resultado final.

## 5. CONCLUSÃO E TRABALHO FUTURO

Em suma, concluímos que o SA (*Simulated Annealing*) será o algoritmo mais passível de utilizar numa aplicação real dos apresentados. Isto pois, apesar de não responder sempre da forma "mais correta" possível, em pequenos espaços de tempo é capaz de chegar a aproximações bastante razoáveis, especialmente com uma temperatura boa inicial. Quanto às outras implementações e os motivos pelos quais ficam em segundo plano, existem diversos motivos. Verificamos que o MC e os piores resultados, a nível de tempos o Greedy é aquele, devido à própria complexidade do algoritmo, que é mais lento porém é também o que apresenta melhores resultados. Havendo a necessidade de considerar ambos esses fatores vamos então deixá-lo de parte. Sobra então o GA que se encontra em situações muito similares ao Greedy, pois apesar de dar melhores resultados tem um *overhead* bastante considerável. Sendo assim tanto o GA como o Greedy devem ter preferência apenas quando existe uma necessidade por resultados de melhor qualidade. Ficaram contudo certas paraâmetros e métricas por analisar, tais como uma maior

comparação entre o nó 781 e o 662 em todos os algoritmos e não apenas no *Greedy* e um teste específico para a alteração dos parâmetros do algoritmo GA, achamos que especialmente esta última parte poderia ser interessante, tentando assim aproveitar os melhores resultados do GA e tentando diminuir o seu tempo de execução, caso atingíssemos esse objetivo a nossa resposta final e conclusão de projecto poderia também sofrer algumas alterações.