Pontifícia Universidade Católica
do Rio de Janeiro

INF1771 - Inteligência Artificial

CVRP - Problema de Roteamento de Veículos Capacitados

Autores:
Caio Valente
Gabrielle Brandemburg

Professora:
Renatha Capua

Data:
29/04/2019

Sumário

1. Introdução

Este trabalho foi proposto pela disciplina “INF1771 - Inteligência Artificial” da PUC-Rio em 2019.1, ministrada pela professora Renatha Capua, e realizado pelos alunos Caio Valente e Gabrielle Brandemburg.

O trabalho consiste em uma análise do problema de roteamento de veículos, ou VRP (Vehicle Routing Problem), que é o nome dado a uma classe de problemas onde um conjunto de rotas para uma frota de veículos baseados em um ou vários depósitos deve ser determinado para um número de cidades, clientes, ou “nós”, geograficamente dispersos. Mais especificamente, estaremos abordando a versão conhecida como CVRP (Capacitated VRP), que é uma das vertentes dessa classe de problemas em que os veículos têm uma capacidade de carga limitada.

Durante o estudo do CVRP, dois algoritmos de solução foram desenvolvidos: a busca local "Hill-climbing", utilizando a transformação de "relocate" e "2-opt" para geração de vizinhança; e Simulated Annealing, utilizando "relocate", "swap" ou "2-opt" para geração de vizinho aleatório. Esses algoritmos serão apresentados e analisados neste relatório, com diferentes tipos de abordagem. Os resultados também serão comparados aos resultados ótimos conhecidos para dadas situações.

2. Definição do Problema

O VRP é um problema combinatorial envolvendo cálculos com números naturais bem conhecido e que se enquadra na categoria de problemas NP Hard da computação, que significa que o esforço computacional necessário para resolver este problema aumenta exponencialmente com o volume dos dados de entrada. Para problemas dessa categoria, muitas vezes é preferível o uso de soluções de aproximação, que possam ser encontradas de forma rápida e sejam suficientemente precisas para o propósito.

O CVRP é a vertente do VRP no qual cada caminhão tem um limite de carga máximo suportado, idêntico para todos os caminhões. Ele é formalmente definido como um grafo completo direcional C = (V, E). V = $\{v_0, v_1, \ldots, v_n\}$ é um conjunto de vértices que representam os clientes ou cidades. Cada vértice possui uma demanda associada que é um valor escalar. E = $\{(v_i, v_j) \mid v_i, v_j \in V, i < j\}$ é o conjunto de arestas que representa a distância entre quaisquer pares de vértices. Nessa análise, cada aresta é calculada pela distância euclidiana entre dois vértices. Existe somente um único depósito para o problema abordado, representado pelo vértice v_0.

A solução para o CVRP consiste em uma partição $R_1, R_2, \ldots, R_m$ de V, que representa a rota de cada veículo. Cada rota R_i consiste em uma sequência de vértices $\{v_{i0}, v_{i1}, \ldots, v_{ik+1}\}$, em que o primeiro e o último elemento são o depósito. Para cada rota R, a soma das demandas dos vértices pertencentes a esta rota não podem ultrapassar a carga do caminhão. Neste caso, a rota é considerada inválida e não pode fazer parte de uma solução.

A **função de custo** do problema é a soma do comprimento de todas as rotas: $\sum_{i=0}^{m} Comprimento(R_i)$ e $Comprimento(R_i) = \sum_{j=0}^{k} c_{j,j+1}$, em que $c_{j,j+1}$ é a distância euclidiana entre v_{ij} e v_{ij+1}.

Os algoritmos propostos tem como objetivo buscar a solução que possui o menor valor para a função de custo descrita acima, ou seja, estamos buscando o conjunto de rotas que minimiza o total da distância percorrida, em que cada rota inicia e termina no depósito e cada cliente só é visitado uma única vez.

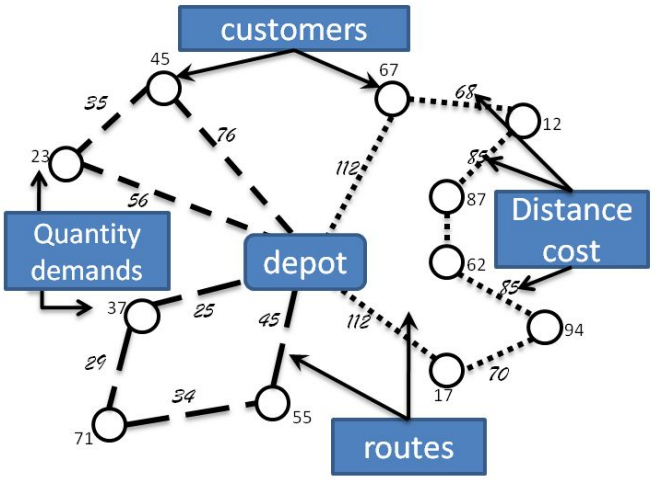


Figura 1: ilustração de uma possível solução para o CVRP [1]

A figura 1 ilustra um exemplo de solução para um problema de CVRP, que possui 3 rotas e o comprimento de cada rota é R1 = 167, R2 = 532, R3 = 133. A função de custo é de 832. O limite de carga para cada caminhão é de pelo menos 339, visto que a soma das demandas da maior rota (R2) possui esse valor.

3. Algoritmos implementados

Os algoritmos desenvolvidos tiveram por objetivo buscar, dentro do espaço de soluções, a solução cuja soma das distâncias é a menor possível, como foi proposto na seção de Definição do Problema. A maneira de se mover pelo espaço de soluções é através da geração de soluções vizinhas, a partir da solução corrente. Para cada um dos algoritmos propostos, foram utilizadas duas estruturas diferentes para representar a solução e vizinhanças diferentes.

3.1 Algoritmos geradores de soluções iniciais

Greedy (Algoritmo guloso)

A partir da posição do depósito, buscamos o nó mais próximo ao depósito e adicionamos na primeira rota, e partir desse nó, buscamos o nó mais próximo a esse e adicionamos na mesma rota e assim por diante, até que a rota atinja o limite de carga. Ao atingir o limite da rota, iniciamos uma nova rota a partir do depósito e aplicamos a mesma estratégia gulosa, até que todos os nós estejam incluídos em alguma rota.

Naive (Algoritmo randômico)

A rotas são construídas uma por uma, escolhendo sempre um nó aleatório que ainda não foi inserido em nenhuma rota. A rota é finalizada quando ela atinge o limite de carga disponível por caminhão.

O problema do CVRP proposto para o trabalho não restringe a quantidade de caminhões disponíveis. A solução implementada considera que o limite de caminhões é a quantidade de rotas geradas pela solução inicial. É possível que existam solucões com mais rotas mas que o custo seja menor da mesma forma, porém no geral quanto menor a quantidade de rotas, é provável que menor o custo aquela solução terá. Portanto, decidimos escolher essa abordagem por questões de praticidade.

3.2 Algoritmos CVRP

Local Search - Hill-climbing

A busca-local hill-climbing consiste em navegar pelo espaço de soluções através da geração de soluções vizinhas. A partir de uma dada solução inicial, é gerado uma série de vizinhos utilizando uma técnica de vizinhança de preferência. A solução se move somente para o vizinho que possui uma função de custo melhor do que a solução atual, em busca de atingir um mínimo local. As vizinhanças escolhidas para a implementação do hill-climbing nesse trabalho serão explicadas mais à frente.

I. Pseudo-código

```
busca_local():
      solucao_atual = gera_solucao_inicial()
      LOOP:
            vizinho1 = gera_vizinho_inter_rotas(solucao_atual)
            vizinho2 = gera_vizinho_intra_rotas(solucao_atual)

            vizinho = escolhe o melhor vizinho entre vizinho1 e vizinho2

            SE custo(vizinho) é pior ou igual que custo(solucao_atual):
                  PARA
            SE NÃO:
                  solucao_atual = vizinho
      RETORNA solucao_atual


gera_vizinho_inter_rotas(solucao_atual):
      PARA TODO par (rota1, rota2) na solução_atual:
            PARA TODO par de clientes (i, j):
                  novo_vizinho = remove cliente i da rota1 e coloca antes de cliente j
da rota2

                  SE custo(novo_vizinho) é melhor que cust(solucao_atual)
                        RETORNA novo_vizinho
      RETORNA solucao_atual

gera_vizinho_intra_rotas(solucao_atual):
      PARA CADA rota_corrente da solucao_atual:
            PARA TODO par de clientes (i, j) [tal que i < j] da rota_corrente:
                  novo_vizinho = inverte sequência que vai de i até j da rota_corrente

                  SE custo(novo_vizinho) é melhor que custo(solucao_atual)
                        RETORNA novo_vizinho
      RETORNA solucao_atual
```

II. Estrutura da solução

Para o Hill-climbing, a representação da solução foi armazenada em uma lista de listas. Cada sub-lista armazena uma sequência de índices. A informação sobre um determinado nó é acessada através desse índice em um vetor que armazena todas as informações dos nós extraídos do arquivo de entrada. O uso de listas separadas para representar cada rota não permitiu que houvesse uma grande variação de uma solução para uma solução vizinha, diferente do que acontece na estrutura usada no Simulated Annealing. Por outro lado, essa estrutura bem definida permitiu calcular o custo de cada rota separadamente. Desse modo, não foi necessário recalcular o custo da solução inteira, somente das rotas modificadas em cada transformação de vizinhança.

III. Vizinhanças

Foram escolhidas duas vizinhanças diferentes para a navegar pelo espaço de soluções na busca hill-climbing: uma vizinhança gerada por uma transformação inter-rota e uma pela transformação intra-rota. Para cada combinação diferente de rotas e clientes, é gerado um novo vizinho. Para a geração de toda a vizinhança, seria necessário processar todas as combinações possíveis de transformação. Como isso seria muito custoso, foi utilizado a abordagem *best-first* para escolha do vizinho, ou seja, o primeiro vizinho gerado que possui um custo melhor do que o custo da solução atual é o escolhido.

Inter-rota: relocate (1 elemento)

A primeira é uma transformação relocate inter-rota. Nessa vizinhança, é removido um nó de uma das rotas e esse nó é inserido em outra rota.

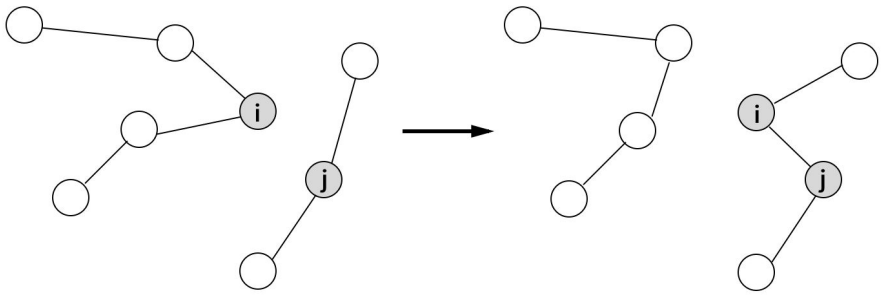


Figura2: relocate inter-rota de somente um elemento.

Intra-rota: 2-opt

Essa transformação inverte uma sub-sequência de uma determinada rota.

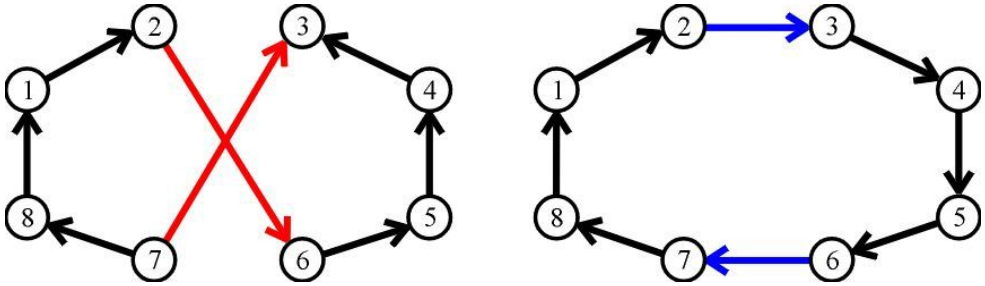


Figura 3: 2-opt - inversão da sequência 6-5-4-3 para 3-4-5-6

A cada iteração da busca local, as duas vizinhanças foram aplicadas de forma independente em cima da solução corrente. O novo vizinho que obtivesse o melhor resultado da função de custo é escolhido

como nova solução corrente. A condição de parada do algoritmo ocorre quando, a partir de uma determinada solução corrente, não for possível gerar um vizinho que seja melhor do que o custo da solução atual. Chegando assim, em um máximo local. Apesar de raro, essa condição nem sempre pode ser garantida pois a vizinhança pode ser muito grande. Portanto, foi adicionado um limite arbitrário de iterações para a busca local caso a condição do máximo local não seja satisfeita.

Simulated Annealing

Assim como o hill-climbing, o simulated annealing se move em busca de encontrar a solução ótima. Porém, para escapar de máximos locais indesejados, essa metaheurística é capaz de aceitar movimentos de piora de acordo com uma determinada probabilidade, modelada por variáveis de controle que mudam a cada iteração. A probabilidade de aceitação de movimentos de piora vai depender do quão ruim é o vizinho selecionado, em comparação com a solução corrente. Quanto mais discrepante for a diferença entre ambos, mais difícil será daquele movimento ser aceito. Além dessa diferença (que é medida pela variável deltaC no pseudo-código), essa probabilidade também vai depender de uma variável de controle T, que vai diminuindo a cada iteração. Para as primeiras iterações, o T é alto e mais movimentos de piora são aceitos, pois no início queremos realizar uma maior exploração do espaço de soluções. Ao longo das iterações, o T vai diminuindo e menos movimentos de piora são aceitos, a fim de concentrar a solução em um determinado local e convergir.

Na implementação do trabalho, a taxa de queda do T é constante e a quantidade de iterações (no pseudo-código é representado por DIM_VIZINHANCA) a cada valor de T depende da quantidade de nós do problema de entrada, que é multiplicado por um fator arbitrário.

O algoritmo de simulated annealing implementado neste trabalho é não-determinístico, pois suas três funções de transformação são aplicadas de forma aleatória sobre elementos aleatórios do vetor de soluções. Isso faz com que o algoritmo possa gerar diferentes soluções finais mesmo sendo aplicado sobre uma mesma solução inicial, com os mesmos valores em suas quatro variáveis de controle.

I. Pseudo-código

```
simulated_annealing():
      solucao_inicial = gera_solucao_inicial()
      solucao_corrente = solucao_opt = solucao_inicial

      ENQUANTO T > TEMP_FINAL:
            ENQUANTO i < VIZINHANCA:
                  novo_vizinho = gera_vizinho(solucao_corrente)

                  SE novo_vizinho não é válido:
                        pula iteração

                  deltaC = custo(novo_vizinho) - custo(solucao_corrente)

                  SE deltaC < 0:
                        solucao_corrente = novo_vizinho
                        SE novo_vizinho é melhor do que solucao_opt:
                              solucao_opt = novo_vizinho
                  SE NÃO:
                        aceita novo_vizinho como solucao_corrente com uma probabilidade
```

$e^{deltaC/T}$

```
                i = i + 1

            T = T*FATOR_T
      RETORNA solucao_opt

gera_vizinho(solucao):
      vizinho = aleatorio(swap(solucao), relocate(solucao), 2_opt(solucao))

swap(solucao_input):
      i,j = duas posições aleatórias de solucao_input
      novo_vizinho = troca nó da posição i com nó da posição j de solucao_input
      RETORNA novo_vizinho

relocate(solucao_input)
      i,j = duas posicoes aleatórias de solucao_input
      novo_vizinho = move nó da posicao i para a posicao j de solucao_input
    RETORNA novo_vizinho

reverse(solucao_input)
      i,j = duas posicoes aleatórias de solucao_input
      novo_vizinho = inverte a sequencia que vai de i até j de solucao input
      RETORNA novo_vizinho
```

II. Estrutura da solução

A estrutura utilizada para armazenar a solução no SA é diferente da estrutura utilizada no Hill-climbing. Dessa vez, optamos por armazenar toda a solução como uma lista de índices, no qual o índice 0 representa o depósito e está replicado ao longo do vetor, pois ele delimita a sequência de cada rota, como mostrado na figura a seguir:

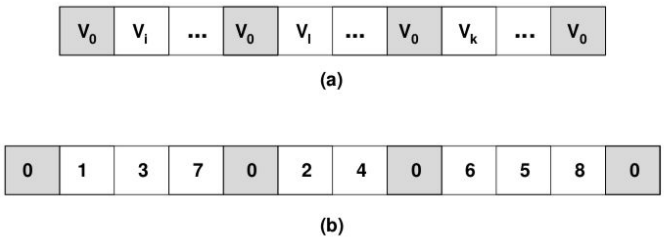


Figura 4: representação da solução: Os índices '0' delimitam as rotas. Na figura, a solução é formada pelas rotas 1->3->7; 2->4; e 6->5->8. A lista de solução não necessariamente precisa ter os indices '0' no início e no final. [5]

Essa representação possibilitou uma maior diversidade na geração de vizinhança utilizando heurísticas simples de swap, relocate e inversão de sequência (similar ao 2-opt, mas também gera transformações inter-rotas), pois dependendo dos índices escolhidos para geração do vizinho, ocorre uma transformação inter-rota ou intra-rota. Entretanto, o cálculo do custo se tornou mais custoso com essa abordagem, pois não é possível recalcular (de forma simples e direta) o custo somente para as rotas modificadas a cada transformação. Optamos por recalcular o custo para toda solução a cada novo vizinho gerado.

III. Vizinhanças

Swap (1 elemento):

Essa transformação troca dois elementos da solução de lugar. Dependendo dos elementos escolhidos, ela pode trocar dois elementos de uma mesma rota de lugar, ou dois elementos de rotas diferentes. Caso um dos elementos seja um depósito, ocorre a fusão de duas rotas e uma outra rota é dividida, como no exemplo a seguir:

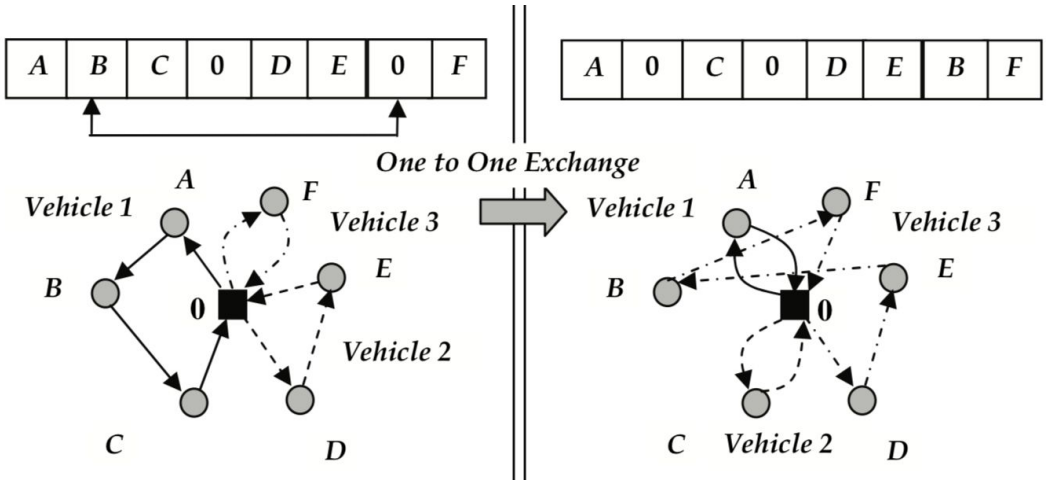


Figura 5: Swap envolvendo um elemento depósito e outro não depósito. As rotas DE e F se fundem com B e se tornam DEBF, enquanto que a rota ABC é divida em A e C. [6]

Relocate (1 elemento):

Assim como o relocate implementado para o Hill-climbing, essa transformação remove um elemento da sequência e insere em uma outra posição. Pela forma como a solução foi estruturada, a realocação pode acontecer dentro de uma mesma rota, de uma rota para outra rota diferente e, caso o elemento realocado seja do índice '0' (depósito), ocorre a fusão de duas rotas e a partição de outra rota em duas.

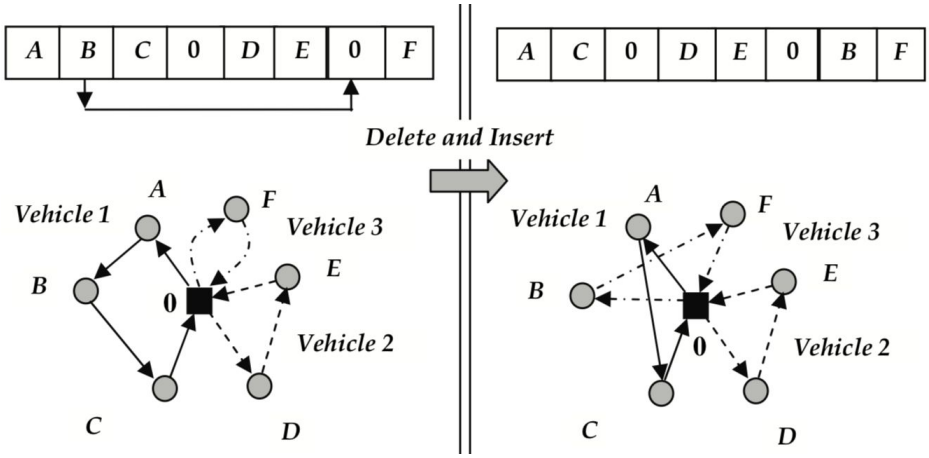


Figura 6: Exemplo de relocate de um elemento não-nulo. [5]

Inversão de sequência:

Essa transformação é similar ao 2-opt, em que dados duas posições (i,j), a sequência entre i e j é removida e inserida no mesmo local, porém invertida. Assim como as demais vizinhanças, ela também pode gerar uma transformação intra ou inter-rotas. Essa transformação pode gerar vizinhos com uma solução bem diferente da original, e também têm maior probabilidade de gerar soluções inválidas.

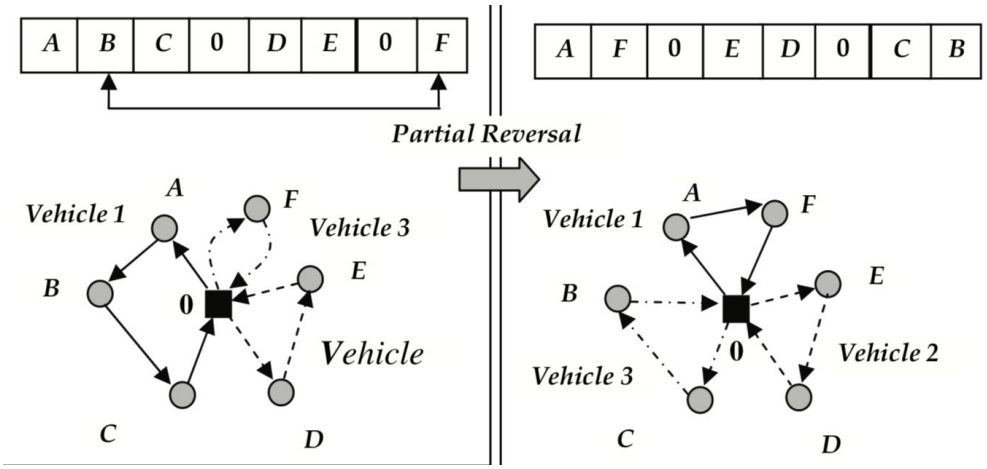


Figura 6: Inversão que afeta várias rotas ao mesmo tempo. [5]

4. O executável cvrp.py

4.1 Como instalar as dependências

O código foi escrito em python 3, que é uma dependência.

Dentro do projeto, existe um arquivo chamado install_deps.sh que instala as dependências de python necessárias para que nosso código funcione, em **Ubuntu**:

```
$ bash install_deps.sh
```

Se esse arquivo não funcionar ou se sua distribuição não for Ubuntu, a segunda opção é usar o pip3 para instalar manualmente as bibliotecas que estão listadas no arquivo requirements.txt.

Em casos de problema com a biblioteca matplotlib, que só é usada no fim do código para gerar um gráfico, ela será automaticamente desabilitada e o gráfico da melhor solução no fim do programa não será exibido.

4.2 Como executar o programa

```
$ python3 cvrp.py <algorithm name> <initial solution algorithm> <input file>
[-n <times to execute>] [--learn]
```

Onde:

Parâmetro	Opcões	Significado
`<algorithm name>`	‘annealing’ ou ‘local_search’	Algoritmo que será usado para abordar o problema.
`<initial solution algorithm>`	‘greedy’ ou ‘naive’	Algoritmo que será usado na geração de solução inicial.
`<input file>`	Path de algum arquivo em ./vrp/	Arquivo VRP contendo as posições e demandas dos nós do problema.
`[-n <times to execute>]`	Opcional; Inteiro maior que zero	Número de vezes em que o algoritmo escolhido será aplicado aos dados iniciais, buscando dados estatísticos. Default = 1
`[--learn]`	Opcional; Boolean	Se esse argumento estiver presente, o algoritmo vai usar a solução final de cada instância como solução inicial da próxima. Default = False

Exemplos de utilização

```
$ python3 cvrp.py annealing naive vrp/B-n31-k5.vrp -n 50 --learn
```

A linha acima executa o algoritmo annealing com uma solução inicial naive, nos nós descritos no arquivo vrp/B-n31-k5.vrp por 50 vezes, utilizando a solução final de cada instância como solução inicial da próxima

```
$ python3 cvrp.py local_search greedy vrp/A-n80-k10.vrp
```

A linha acima executa o algoritmo local_search com uma solução inicial do tipo greedy, nos nós descritos no arquivo vrp/A-n80-k10.vrp uma única vez.

4.3 Organização do código

- cvrp.py
  Ponto de entrada do programa, contém uma função main que agrupa todos os dados de entrada, executa o programa e exibe os dados de saída. Também é responsável por gerar estatísticas sobre a execução.

- constants.py
  Contém as constantes de ajuste dos dois algoritmos, que podem ser alteradas para estudar o comportamento destes.

- intial_solution_generator.py
  Módulo responsável por gerar soluções iniciais nos formatos esperados pelos algoritmos. Contém dois algoritmos de geração de solução inicial: greedy() e naive().

- parser.py
  Módulo responsável pela leitura e parsing dos argumentos da linha de comando e dos arquivos que são lidos pelo programa.

- plot.py
  Contém funções de plot, responsáveis pela geração de gráficos. É usado para exibir o gráfico da melhor solução encontrada, ao fim da execução do programa.

- annealing.py
  Esse módulo contém todas as funções relacionadas ao algoritmo annealing.

- local_search.py
  Esse módulo contém todas as funções relacionadas ao algoritmo de busca local.

- classes.py
  Contém as classes Route e Node, que definem as estruturas mais básicas utilizadas pelos algoritmos.

4.4 Abordagem estatística

O estudo estatístico consiste em aplicar um algoritmo de solução de CVRP diversas vezes em um mesmo estado inicial e analisar estatisticamente as variações em suas soluções. Nós estudamos, mais especificamente, o custo e tempo médios obtidos pelo algoritmo, assim como a melhor solução final encontrada após tantas execuções.

O estado inicial consiste em dois objetos: o espaço amostral de nós sobre os quais o algoritmo é aplicado, e a configuração das variáveis de controle do algoritmo. Um estudo estatístico de N instâncias leva em conta que todos esses valores são iguais em todas as N execuções do algoritmo.

Esse tipo de abordagem só faz sentido se as soluções dos algoritmos forem não-determinísticas, ou seja, se for possível ter soluções finais diferentes para duas instâncias de um mesmo algoritmo sendo aplicado num mesmo estado inicial. Para tal, é necessário que, pelo menos, uma parte do algoritmo completo seja não-determinístico, incluindo a geração de solução inicial.

O algoritmo annealing implementado nesse trabalho é não-determinístico por natureza, então podemos fazer um estudo estatístico do mesmo independentemente do algoritmo de geração de solução inicial utilizado. Já o algoritmo de busca local, não. Como esse algoritmo é determinístico, seu estudo estatístico só faz sentido quando combinado com um gerador de soluções iniciais que seja não-determinístico. Por esse motivo, o algoritmo local_search não pode ser estudado com estatisticamente utilizando nosso algoritmo gerador de soluções iniciais do tipo greedy - pois todos as instâncias teriam o mesmo resultado.

Essa abordagem é interessante principalmente por duas razões. A primeira, é que os dados estatísticos nos permitem prever com mais precisão o comportamento do algoritmo. Conseguimos determinar mais facilmente até que ponto o algoritmo é eficiente, e quando ele deixa de ser, tanto com relação ao valor de custo da melhor solução quanto ao tempo de processamento necessário. O segundo ponto interessante é que em alguns casos a aplicação de várias instâncias do algoritmo tem um tempo total que continua sendo relativamente viável para determinados fins, ou seja, o próprio método estatístico poderia vir a ser aplicado como abordagem para um problema de CVRP. Além do mais, nesse caso, as diversas instâncias do algoritmo poderiam ser executadas paralelamente, já que são independentes entre si, o que diminuiria notavelmente o tempo necessário para encontrar uma solução aproximada que é, em geral, melhor do que a solução de uma única instância.

O estudo estatístico pode ser ativado com o argumento **-n** no nosso executável, que recebe um inteiro como valor.

4.5 Abordagem de aprendizado de máquina

Assim como a abordagem estatística, a abordagem de aprendizado de máquina também se baseia em aplicar várias vezes um mesmo algoritmo em um problema, e, portanto, só faz sentido de ser aplicada em algoritmos não-determinísticos. No entanto, neste caso, o estado inicial muda de uma instância para a outra, pois a melhor solução encontrada por uma instância é utilizado como solução inicial da instância seguinte. Dessa forma, a instância seguinte vai ter sempre uma solução com custo menor ou igual ao da instância anterior.

Esse modo de execução na verdade se equipara à uma nova versão dos algoritmos, onde a solução “final” seria a solução final da main, saída da última instância do algoritmo, e não a solução saída de uma única instância do algoritmo. É uma abordagem do problema CVRP utilizando um mecanismo simples de aprendizado de máquina como catalisador para os dois algoritmos abordados neste trabalho.

Como no método estatístico, o método com aprendizado também multiplica o tempo de solução do problema. No entanto, esse método não é facilmente paralelizável, já que uma instância N depende da solução final da instância N-1. Outra desvantagem é que o algoritmo pode ficar preso em máximos locais mais facilmente do que em uma abordagem estatística com soluções iniciais que variem (no caso da busca local com gerador de soluções iniciais naive) ou com funções de transformação não-determinísticas (no caso do annealing).

As vantagens desse tipo de abordagem em relação à abordagem estatística são principalmente duas: a primeira é que o valor de custo das soluções converge mais rapidamente para um valor mais próximo do melhor valor possível. Desta forma, se o problema exigir um tempo de processamento menor, esse método pode resultar em uma aproximação aceitável em um tempo, na maioria das vezes, muito menor. A segunda vantagem é que, com esse método, a solução final do método melhora de forma proporcional ao número de instâncias (embora não linearmente). O que faz com que, para números de ciclo de execução altos, esse algoritmo garanta soluções melhores do que o estatístico.

A opção **--learn** ativa o método de execução dos algoritmos com aprendizado. Quando a opção está ativada, o looping principal do programa, que executa os algoritmos, vai injetar a saída de uma instância como entrada na instância seguinte. Ela deve ser usada em conjunto com o argumento **-n**, que representa o número de instâncias.

5. Resultados Gerais

Nas tabelas abaixo é feita uma comparação dos resultados entre o BKS, a busca local Hill-climbing e o Simulated Annealing utilizando o algoritmo guloso para a solução inicial, e a Hill-climbing e Simulated Annealing utilizando o algoritmo randômico como solução inicial. Como o simulated annealing é não determinístico, cada instância de entrada foi executada 100 vezes e o resultado das tabelas corresponde ao resultado médio.

Os parâmetros de controle utilizados para o SA foram T_INICIAL = 20, T_FINAL = 1, FATOR_T = 0.9, FATOR_N = 0.9. Esses parâmetros foram escolhidos empiricamente, ao testar a execução diversas vezes com valores diferentes. A medição de tempo corresponde ao tempo de CPU.

O resultado do Hill-climbing (Random) é a abordagem de Random-restart, no qual geramos 15 estados iniciais aleatórios e a buscamos pelo mínimo local a partir de todos esses estados iniciais. Pegamos o melhor valor obtido nas 15 buscas. Por isso, esse resultado foi o que levou o maior tempo, pois o tempo total não é a média das 15 buscas, e sim da soma de todas as buscas.

5.1 Tabelas

X-n101-k25	RESULTADO	ERRO RELATIVO [%] (BKS)	TEMPO (s)
BKS	27591	-	-
HILL-CLIMBING (GREEDY)	40346	31.61	5.723
SA (GREEDY)	32375	14.78	2.715
HILL-CLIMBING (RANDOM)	45744	39.68	429.835
SA (RANDOM)	35179	21.57	2.380

X-n110-k13	RESULTADO	ERRO RELATIVO [%] (BKS)	TEMPO (s)
BKS	14971	-	-
HILL-CLIMBING (GREEDY)	19139	21.78	4.275
SA (GREEDY)	17925	16.48	2.127
HILL-CLIMBING (RANDOM)	34845	57.04	1931.241
SA (RANDOM)	24531	38.97	1.862

X-n115-k10	RESULTADO	ERRO RELATIVO [%] (BKS)	TEMPO (s)
BKS	12747	-	-
HILL-CLIMBING (GREEDY)	17172	25.77	22.261
SA (GREEDY)	16663	23.50	1.764
HILL-CLIMBING (RANDOM)	27422	53.52	2272.781
SA (RANDOM)	23719	46.26	1.564

X-n204-k19	RESULTADO	ERRO RELATIVO [%] (BKS)	TEMPO (s)
BKS	19565	-	-
HILL-CLIMBING (GREEDY)	23551	16.92	83.844
SA (GREEDY)	23328	16.13	4.896
HILL-CLIMBING (RANDOM)	55532	64.77	927.222
SA (RANDOM)	38295	48.91	5.009

5.2 Análise

Como esperado, no geral o Simulated Annealing gerou um resultado melhor em comparação com a busca local hill-climbing. O SA também foi muito mais rápido que o HC, pois o HC depende de encontrar um mínimo local como condição de parada do algoritmo, e a condição de parada do SA depende de variáveis de controle arbitrárias. Além disso, pelo uso da estratégia "best-first" na escolha do vizinho, o HC se move mais "lentamente" no espaço de soluções, pois não escolhe o melhor vizinho e sim o primeiro que seja melhor que o estado atual, logo não ocorrem grandes saltos de um estado para o outro. Como o SA escolhe um vizinho aleatório e as funções de geração de vizinho utilizadas causam grandes modificações na solução, a movimentação de um estado para o outro representa um salto muito maior no espaço de soluções.

Obteve-se um resultado muito melhor usando a estratégia gulosa para a solução inicial em relação a solução inicial randômica, pois dessa forma já iniciamos a busca a partir de uma região propícia a ter o valor ótimo global, ou um ótimo local razoável.

Mesmo gerando 15 posições randômicas para a execução do hill-climbing, o resultado não foi satisfatório, pois o hill-climbing fica preso no ótimo local mais próximo e é improvável obter uma posição próxima de um bom ótimo local com apenas 15 tentativas. Não conseguimos ultrapassar o valor de 15 tentativas por conta do tempo de execução.

6. Conclusão

Os valores de custo obtidos nas buscas implementadas possuem uma diferença relativamente grande em relação a melhor solução conhecida (BKS). Após a análise, pensamos em algumas modificações que poderiam trazer resultados mais satisfatórios.

Para a busca hill-climbing, em vez de retornar o primeiro vizinho válido que fosse melhor do que a solução corrente, poderíamos escolher o melhor resultado entre os N primeiros vizinhos válidos que fossem melhores que a solução corrente. Poderíamos também utilizar transformações de vizinhança que gerassem vizinhos mais discrepantes em relação a solução atual, como por exemplo, em vez de aplicar um relocate de somente 1 elemento, poderíamos aplicar um relocate de mais elementos ao mesmo tempo. O mesmo vale para o 2-opt, pois utilizando a estratégia de best-first, a sub-sequência invertida normalmente possuía poucos elementos.

Para o Simulated Annealing, acreditamos que uma cooling schedule mais elaborada poderia ter enviesado a solução para um caminho mais inteligente. Poderíamos ter utilizado uma taxa de queda de temperatura dinâmica, que dependesse da qualidade da solução corrente. Quanto melhor a qualidade da solução, mais rápido a temperatura diminuiria. Aumentar a quantidade de iterações na vizinhança por temperatura também melhora a qualidade da solução ótima, mas aumenta muito o processamento. Além disso, armazenar a estrutura em listas de rotas separadamente deixaria o cálculo do custo mais eficiente, o que compensaria o processamento a mais feito no aumento de iterações por temperatura. O uso de listas separadas por rota também resultaria na geração de menos soluções inválidas.

7. Bibliografia

[1]
https://www.researchgate.net/figure/The-Capacitated-Vehicle-Routing-Problem-CVRP_fig1_319754352

[2] The VRP Web
http://www.bernabe.dorronsoro.es/vrp/index.html?VRP-Intro.html

[3] CVRP com Simulated Annealing
https://www.inf.ufrgs.br/~mrpritt/lib/exe/fetch.php?media=inf05010:cvrp_sa1.pdf

[4] Metaheuristic Algorithms For The Vehicle Routing Problem With Time Window And Skill Set Constraints
https://pdfs.semanticscholar.org/6227/de891f6e9b5b2b6e30e679ee82e04522a534.pdf

[5] A Simulated Annealing Algorithm for The Capacitated Vehicle Routing Problem
https://pdfs.semanticscholar.org/a2ab/0b7bc74a0291e8a8b97aac53360b239e93ca.pdf

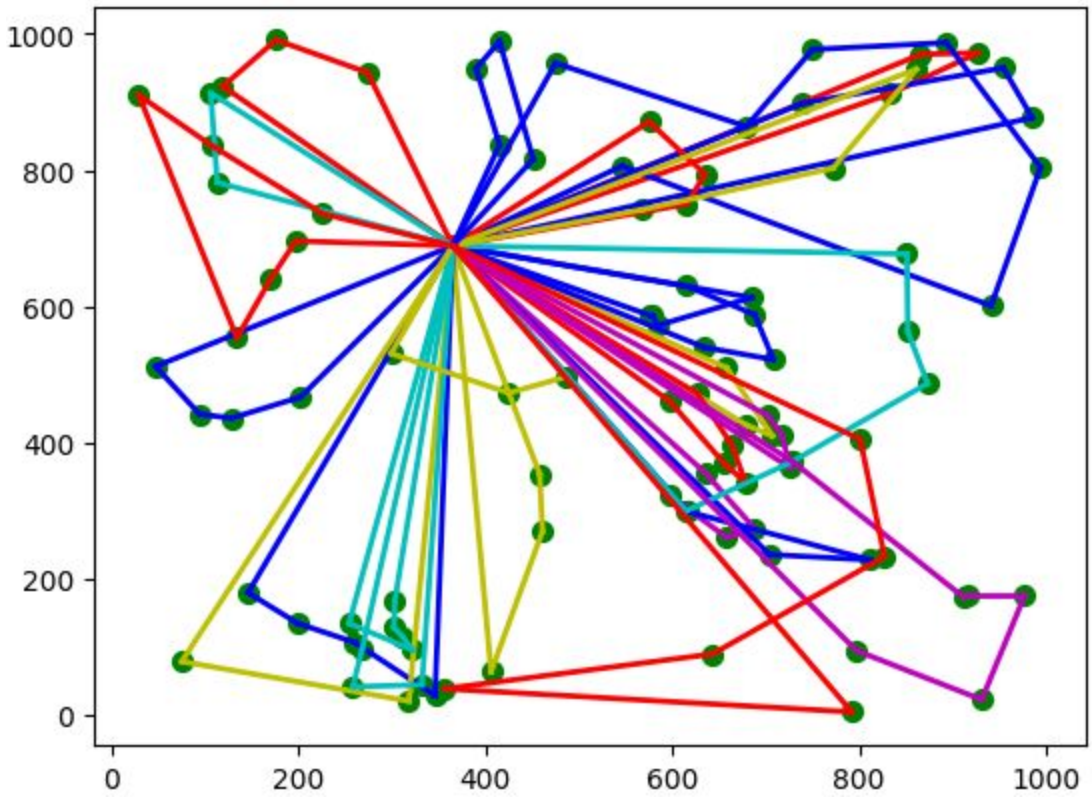
[6] Application of Simulated Annealing to Routing Problems in City Logistics
http://cdn.intechweb.org/pdfs/4627.pdf

8. APÊNDICE: resultados na íntegra

8.1 Annealing, Greedy, X-n101-k25, statistic

```
$ python3 cvrp.py annealing greedy vrp/X-n101-k25.vrp -n 100
```

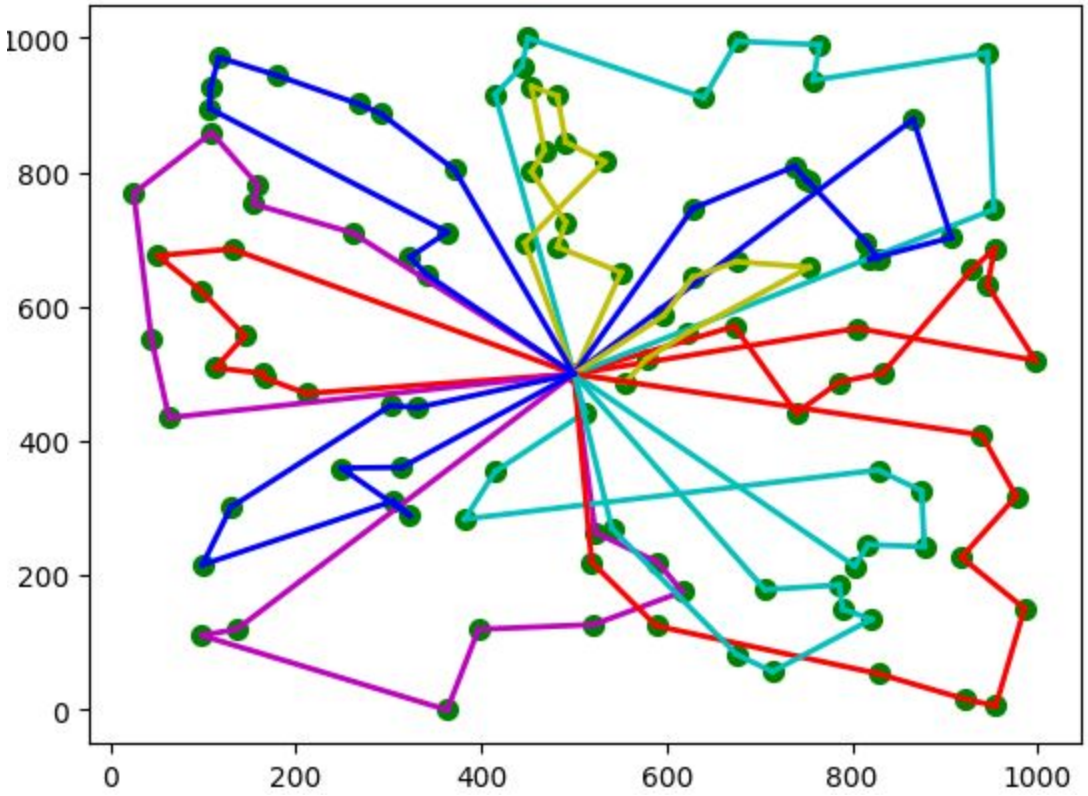
```
The algorithm annealing ran for 100 times !

Total execution time: 271.497 seconds
Average execution time: 2.715 seconds
Optimal solution cost: 27591
Initial solution cost: 41279

# AVERAGE SOLUTION ANALYSIS #

average cost: 32375
The average solution costs 14.78 percent MORE than the OPTIMAL solution
The average solution costs 21.57 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 33995
The worst solution costs 18.84 percent MORE than the OPTIMAL solution
The worst solution costs 17.65 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 30519
The best solution costs 9.59 percent MORE than the OPTIMAL solution
The best solution costs 26.07 percent LESS than the INITIAL solution

Best solution:: [0, 75, 93, 33, 0, 31, 53, 0, 63, 42, 65, 39, 0, 13, 4, 18, 0, 69,
54, 86, 1, 0, 20, 41, 22, 15, 0, 52, 62, 51, 0, 0, 0, 16, 70, 0, 0, 10, 6, 78, 0, 32,
95, 73, 35, 46, 24, 0, 55, 9, 92, 0, 0, 88, 67, 3, 0, 23, 47, 89, 71, 81, 97, 11, 0,
98, 99, 91, 0, 40, 77, 28, 14, 44, 0, 68, 76, 90, 84, 66, 0, 82, 60, 59, 0, 5, 12,
58, 0, 8, 94, 96, 64, 0, 56, 34, 17, 80, 0, 19, 61, 100, 0, 38, 26, 48, 57, 25, 0,
72, 36, 29, 29, 43, 45, 49, 0, 21, 27, 83, 0, 79, 30, 85, 50, 0, 0, 87, 37, 7, 74, 2,
0]
```


1000
800
600
400
200
0
0
200
400
600
800
1000

8.2 Annealing, Greedy, X-n110-k13, statistic

```
$ python3 cvrp.py annealing greedy vrp/X-n110-k13.vrp -n 100
```

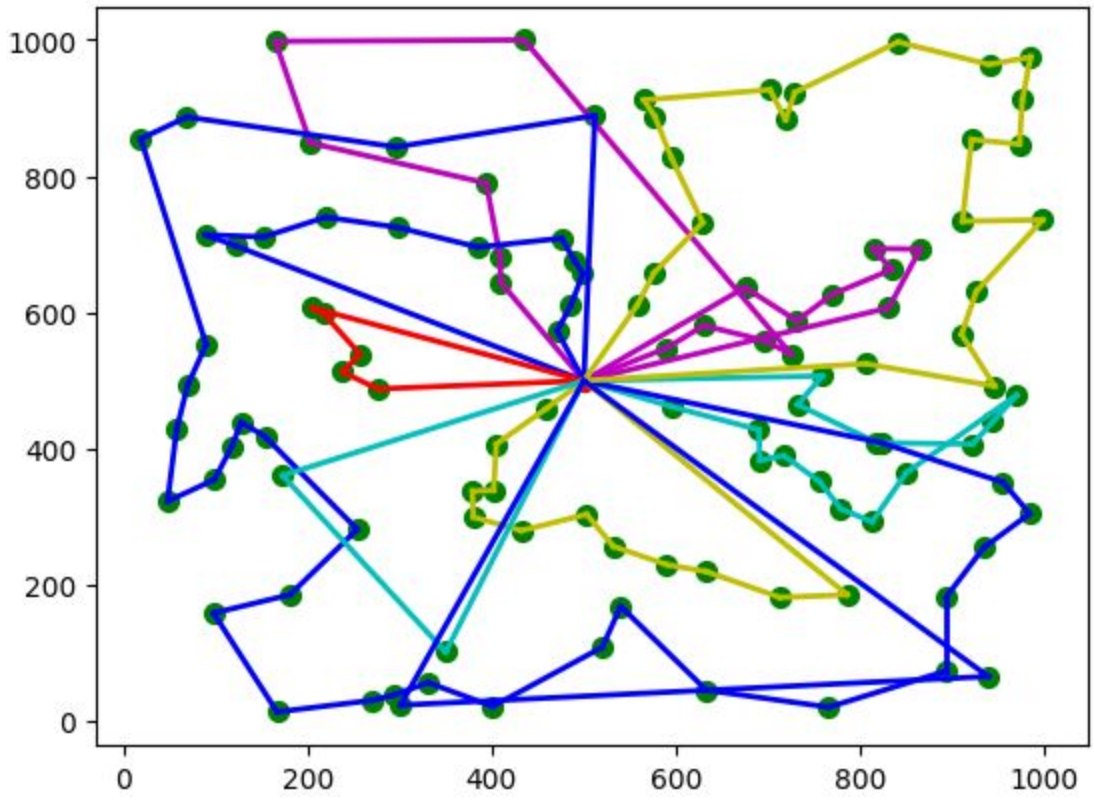
```
The algorithm annealing ran for 100 times !

Total execution time: 212.709 seconds
Average execution time: 2.127 seconds
Optimal solution cost: 14971
Initial solution cost: 19244

# AVERAGE SOLUTION ANALYSIS #

average cost: 17925
The average solution costs 16.48 percent MORE than the OPTIMAL solution
The average solution costs 6.85 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 17118
The best solution costs 12.54 percent MORE than the OPTIMAL solution
The best solution costs 11.05 percent LESS than the INITIAL solution
```



8.3 Annealing, Greedy, X-n115-k10, statistic

```
$ python3 cvrp.py annealing greedy vrp/X-n115-k10.vrp -n 100
```

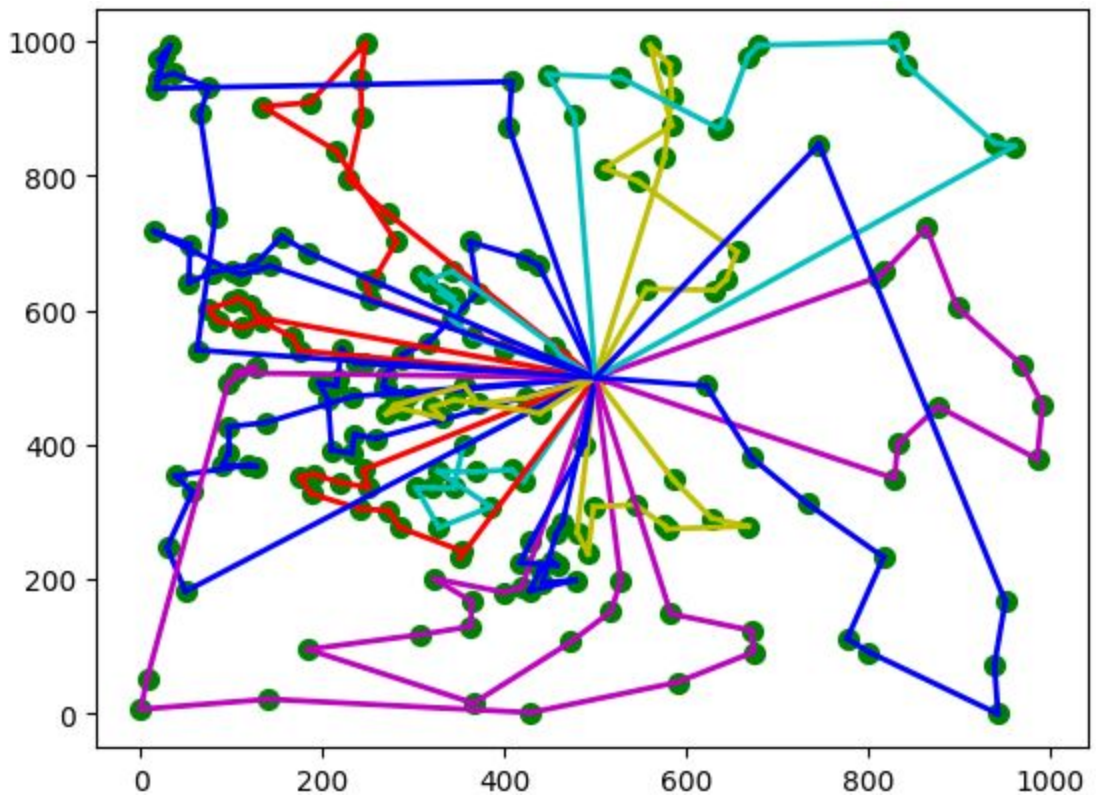
```
The algorithm annealing ran for 100 times !

Total execution time: 176.429 seconds
Average execution time: 1.764 seconds
Optimal solution cost: 12747
Initial solution cost: 17711

# AVERAGE SOLUTION ANALYSIS #

average cost: 16663
The average solution costs 23.50 percent MORE than the OPTIMAL solution
The average solution costs 5.92 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 15638
The best solution costs 18.49 percent MORE than the OPTIMAL solution
The best solution costs 11.70 percent LESS than the INITIAL solution
```



8.4 Annealing, Greedy, X-n204-k19, statistic

```
$ python3 cvrp.py annealing greedy vrp/X-n204-k19.vrp -n 100
```

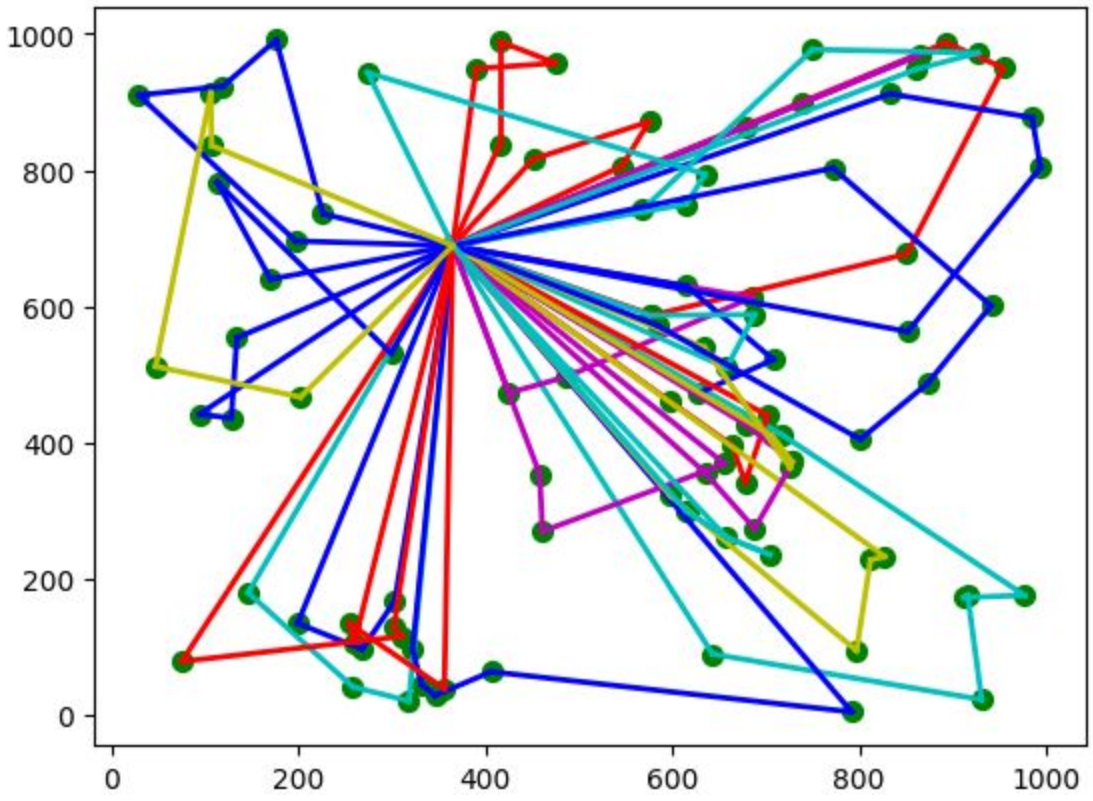
```
The algorithm annealing ran for 100 times !

Total execution time: 489.557 seconds
Average execution time: 4.896 seconds
Optimal solution cost: 19565
Initial solution cost: 23844

# AVERAGE SOLUTION ANALYSIS #

average cost: 23328
The average solution costs 16.13 percent MORE than the OPTIMAL solution
The average solution costs 2.16 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 22606
The best solution costs 13.45 percent MORE than the OPTIMAL solution
The best solution costs 5.19 percent LESS than the INITIAL solution
```



8.5 Annealing, Naive, X-n101-k25, statistic

```
$ python3 cvrp.py annealing naive vrp/X-n101-k25.vrp -n 100
```

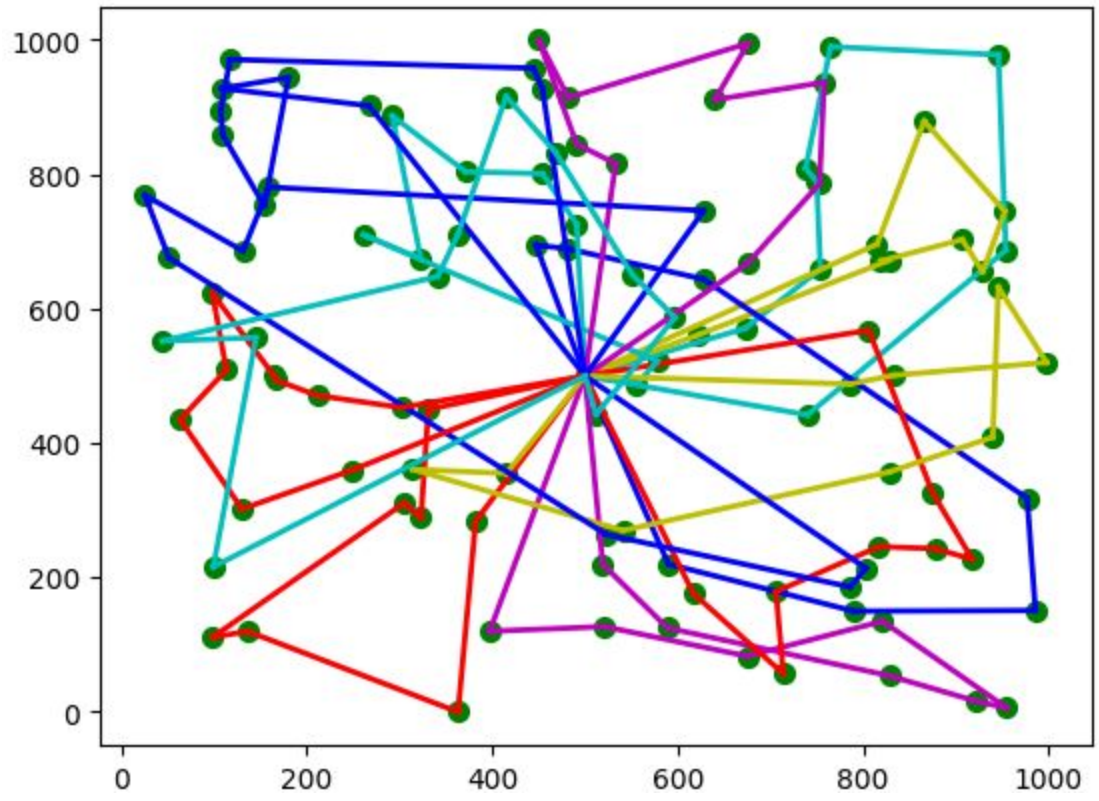
```
The algorithm annealing ran for 100 times !

Total execution time: 238.040 seconds
Average execution time: 2.380 seconds
Optimal solution cost: 27591
Initial solution cost: 60697

# AVERAGE SOLUTION ANALYSIS #

average cost: 35179
The average solution costs 21.57 percent MORE than the OPTIMAL solution
The average solution costs 42.04 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 37968
The worst solution costs 27.33 percent MORE than the OPTIMAL solution
The worst solution costs 37.45 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 31882
The best solution costs 13.46 percent MORE than the OPTIMAL solution
The best solution costs 47.47 percent LESS than the INITIAL solution

Best solution:: [0, 17, 38, 99, 71, 0, 60, 43, 36, 29, 45, 7, 7, 0, 56, 58, 12, 0, 0,
35, 22, 41, 0, 8, 96, 40, 0, 23, 19, 50, 0, 16, 9, 1, 0, 91, 51, 97, 0, 0, 0, 2, 13,
69, 55, 76, 0, 59, 28, 77, 44, 0, 3, 94, 80, 0, 18, 4, 14, 0, 39, 49, 6, 37, 0, 0,
52, 98, 89, 26, 0, 79, 85, 11, 30, 0, 83, 62, 81, 21, 0, 63, 42, 72, 72, 88, 0, 64,
64, 82, 57, 67, 0, 24, 73, 33, 33, 93, 32, 0, 86, 92, 66, 0, 84, 90, 70, 0, 100, 61,
75, 0, 0, 0, 27, 47, 48, 87, 34, 0, 5, 31, 46, 0, 74, 68, 54, 0, 65, 65, 78, 65, 10,
25, 0, 0, 95, 53, 20, 15, 0]
```


1000
800
600
400
200
0
0
200
400
600
800
1000

8.6 Annealing, Naive, X-n110-k13, statistic

```
$ python3 cvrp.py annealing naive vrp/X-n110-k13.vrp -n 100
```

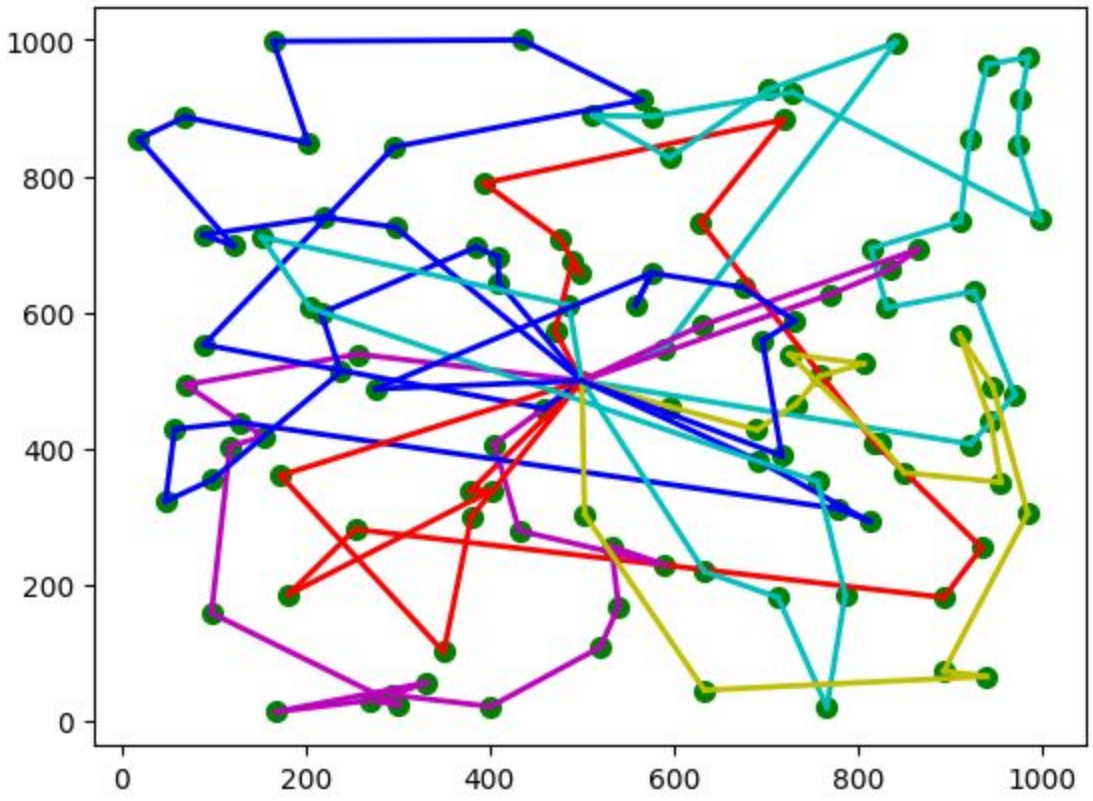
```
The algorithm annealing ran for 100 times !

Total execution time: 186.200 seconds
Average execution time: 1.862 seconds
Optimal solution cost: 14971
Initial solution cost: 58562

# AVERAGE SOLUTION ANALYSIS #

average cost: 24531
The average solution costs 38.97 percent MORE than the OPTIMAL solution
The average solution costs 58.11 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 21881
The best solution costs 31.58 percent MORE than the OPTIMAL solution
The best solution costs 62.64 percent LESS than the INITIAL solution
```



8.7 Annealing, Naive, X-n115-k10, statistic

```
$ python3 cvrp.py annealing naive vrp/X-n115-k10.vrp -n 100
```

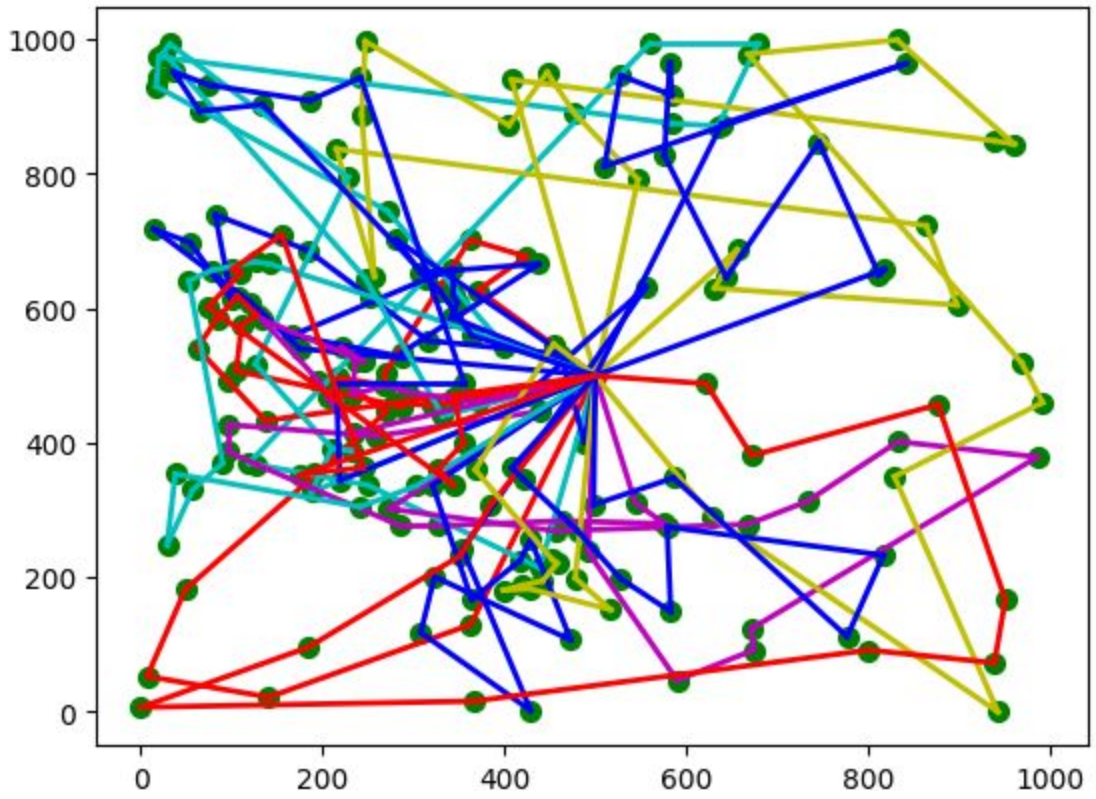
```
The algorithm annealing ran for 100 times !

Total execution time: 156.384 seconds
Average execution time: 1.564 seconds
Optimal solution cost: 12747
Initial solution cost: 62897

# AVERAGE SOLUTION ANALYSIS #

average cost: 23719
The average solution costs 46.26 percent MORE than the OPTIMAL solution
The average solution costs 62.29 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 20427
The best solution costs 37.60 percent MORE than the OPTIMAL solution
The best solution costs 67.52 percent LESS than the INITIAL solution
```



8.8 Annealing, Naive, X-n204-k19, statistic

```
$ python3 cvrp.py annealing naive vrp/X-n204-k19.vrp -n 100
```

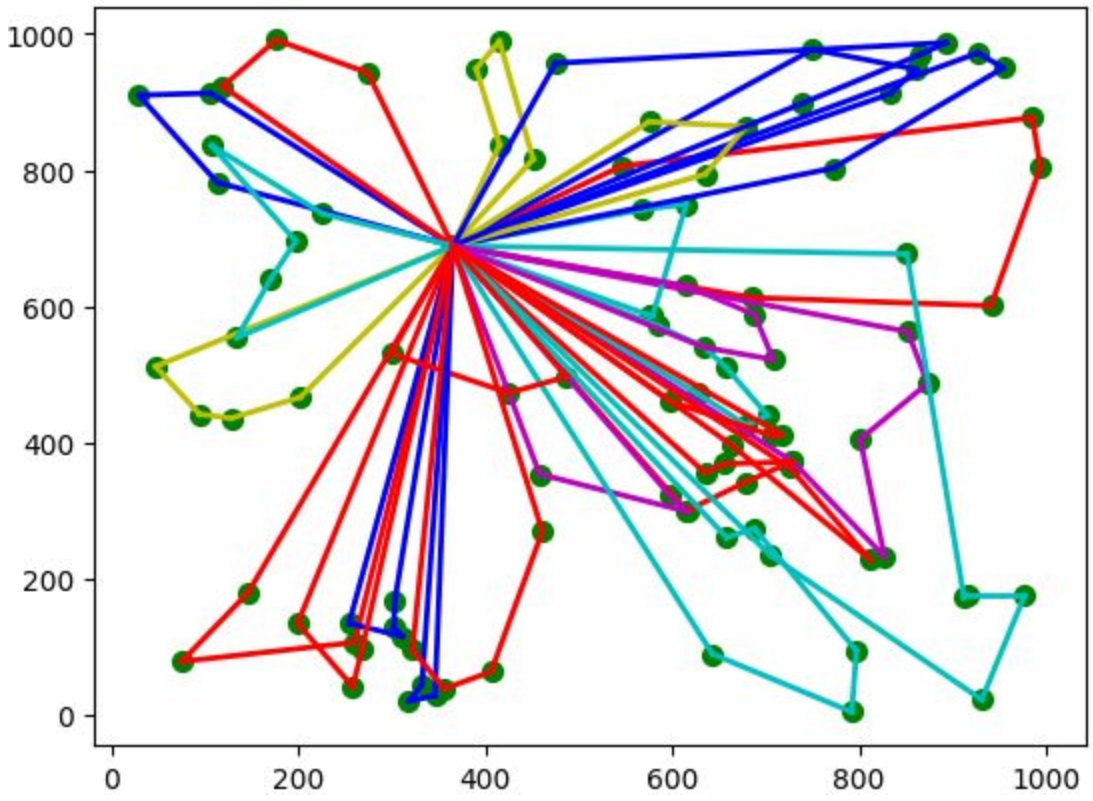
```
The algorithm annealing ran for 100 times !

Total execution time: 500.862 seconds
Average execution time: 5.009 seconds
Optimal solution cost: 19565
Initial solution cost: 92851

# AVERAGE SOLUTION ANALYSIS #

average cost: 38295
The average solution costs 48.91 percent MORE than the OPTIMAL solution
The average solution costs 58.76 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 35636
The best solution costs 45.10 percent MORE than the OPTIMAL solution
The best solution costs 61.62 percent LESS than the INITIAL solution
```



8.9 Annealing, Greedy, X-n101-k25, learning

```
$ python3 cvrp.py annealing greedy vrp/X-n101-k25.vrp -n 100 --learn
```

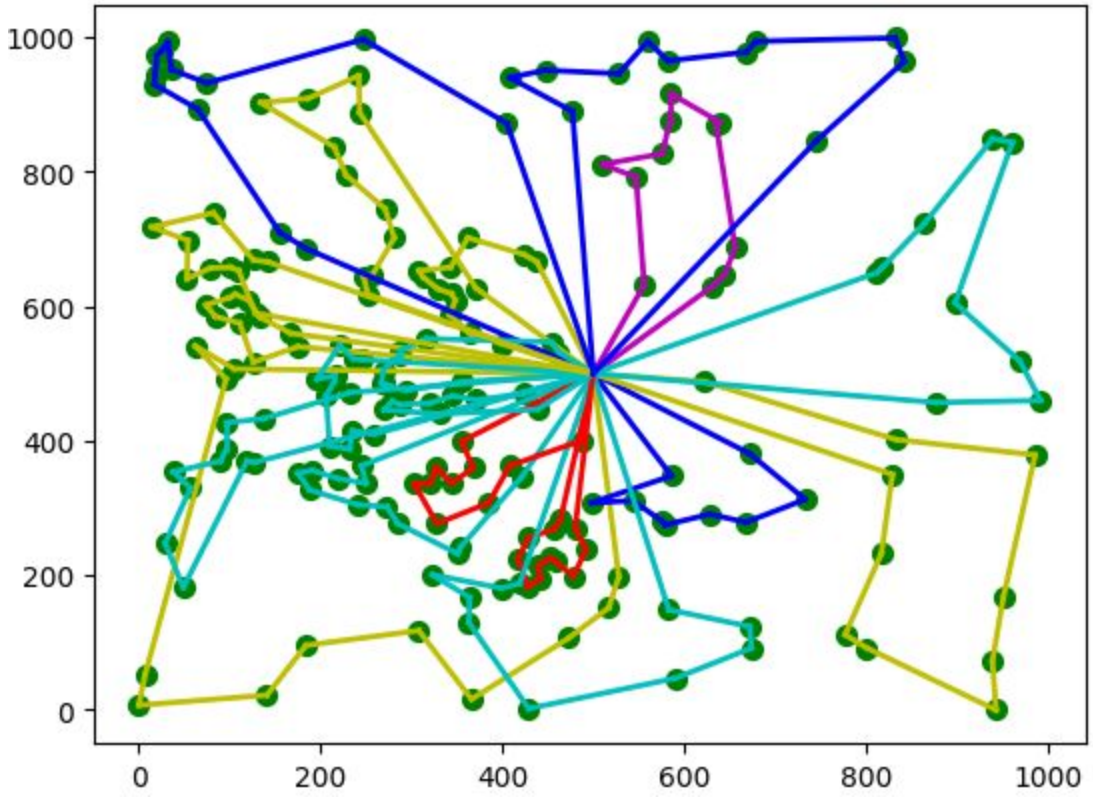
```
The algorithm annealing ran for 100 times !

Total execution time: 212.209 seconds
Average execution time: 2.122 seconds
Optimal solution cost: 27591
Initial solution cost: 41279

# AVERAGE SOLUTION ANALYSIS #

average cost: 29605
The average solution costs 6.80 percent MORE than the OPTIMAL solution
The average solution costs 28.28 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 32430
The worst solution costs 14.92 percent MORE than the OPTIMAL solution
The worst solution costs 21.44 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 29286
The best solution costs 5.79 percent MORE than the OPTIMAL solution
The best solution costs 29.05 percent LESS than the INITIAL solution
```


1000
800
600
400
200
0
0
200
400
600
800
1000

8.10 Annealing, Greedy, X-n204-k19, learning

```
$ python3 cvrp.py annealing greedy vrp/X-n204-k19.vrp -n 100 --learn
```

```
The algorithm annealing ran for 100 times !

Total execution time: 525.140 seconds
Average execution time: 5.251 seconds
Optimal solution cost: 19565
Initial solution cost: 23844

# AVERAGE SOLUTION ANALYSIS #

average cost: 21343
The average solution costs 8.33 percent MORE than the OPTIMAL solution
The average solution costs 10.49 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 23259
The worst solution costs 15.88 percent MORE than the OPTIMAL solution
The worst solution costs 2.45 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 21083
The best solution costs 7.20 percent MORE than the OPTIMAL solution
The best solution costs 11.58 percent LESS than the INITIAL solution
```

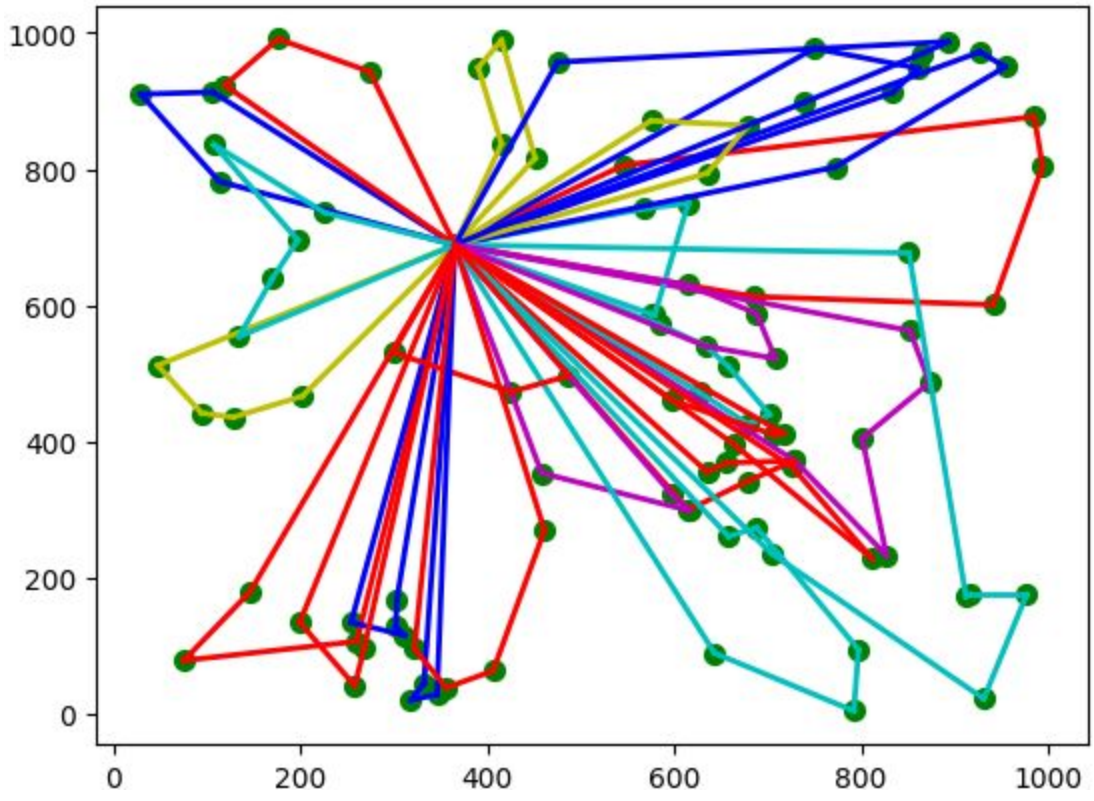

1000
800
600
400
200
0
0
200
400
600
800
1000

8.11 Local Search, Greedy, X-n101-k25

```
$ python3 cvrp.py local_search greedy vrp/X-n101-k25.vrp
```

```
The algorithm local_search ran for 1 times !

Total execution time: 5.723 seconds
Optimal solution cost: 27591
Initial solution cost: 41279

# AVERAGE SOLUTION ANALYSIS #

average cost: 40346
The average solution costs 31.61 percent MORE than the OPTIMAL solution
The average solution costs 2.26 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 40346
The worst solution costs 31.61 percent MORE than the OPTIMAL solution
The worst solution costs 2.26 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 40346
The best solution costs 31.61 percent MORE than the OPTIMAL solution
The best solution costs 2.26 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 2493 - path: [32, 95, 73, 24, 46, 35, 15, 64, 7],
 Route - cost: 659 - path: [50, 79, 30, 85],
 Route - cost: 602 - path: [5, 12, 58],
 Route - cost: 1520 - path: [21, 97, 61, 100, 34, 65],
 Route - cost: 2506 - path: [23, 19, 11, 81, 71, 87, 89],
 Route - cost: 1071 - path: [80, 17, 56, 57],
 Route - cost: 1576 - path: [8, 94, 96, 76],
 Route - cost: 1872 - path: [31, 53, 47],
 Route - cost: 2374 - path: [33, 93, 75, 43],
 Route - cost: 1590 - path: [44, 40, 3, 88, 29, 14],
 Route - cost: 1567 - path: [20, 41, 22, 72],
 Route - cost: 1430 - path: [18, 4, 39, 26],
 Route - cost: 1003 - path: [77, 67, 28],
 Route - cost: 985 - path: [59, 60, 82],
 Route - cost: 1353 - path: [27, 91, 62],
```

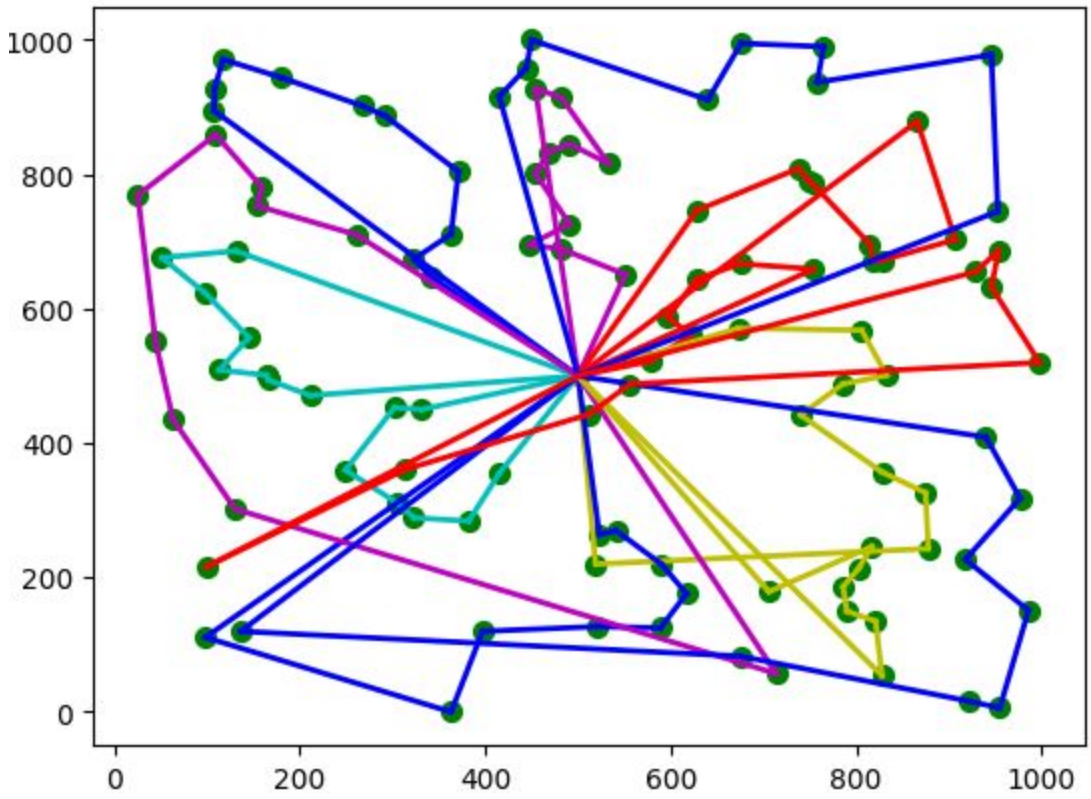
```
Route - cost: 1089 - path: [63, 25, 42],
Route - cost: 1790 - path: [10, 78, 6, 45],
Route - cost: 1584 - path: [38, 48, 37],
Route - cost: 1122 - path: [52, 83],
Route - cost: 1203 - path: [66, 84, 90, 68],
Route - cost: 1466 - path: [1, 86, 54, 69],
Route - cost: 2331 - path: [51, 99, 13],
Route - cost: 1372 - path: [92, 9, 55],
Route - cost: 2226 - path: [98, 36, 49],
Route - cost: 1361 - path: [74, 16],
Route - cost: 2201 - path: [70, 2]]
```



8.12 Local Search, Greedy, X-n110-k13

```
$ python3 cvrp.py local_search greedy vrp/X-n110-k13.vrp
```

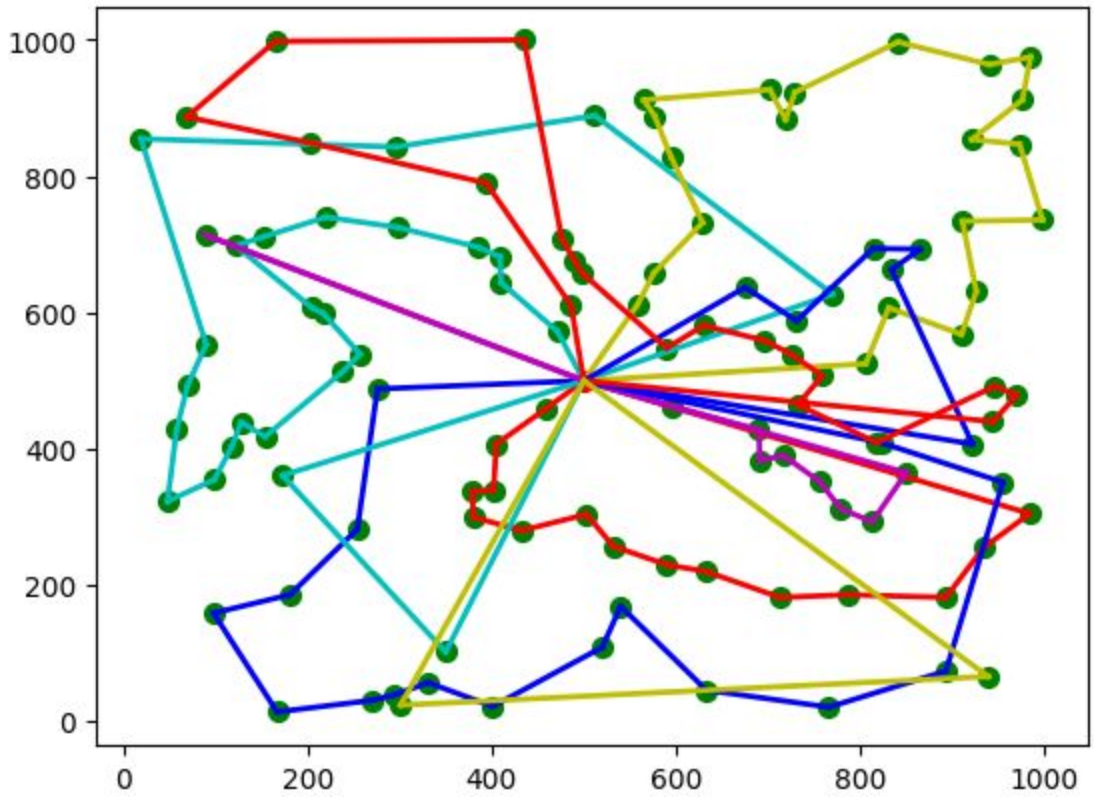
```
The algorithm local_search ran for 1 times !

Total execution time: 4.275 seconds
Optimal solution cost: 14971
Initial solution cost: 19244

# AVERAGE SOLUTION ANALYSIS #

average cost: 19139
The average solution costs 21.78 percent MORE than the OPTIMAL solution
The average solution costs 0.55 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 19139
The worst solution costs 21.78 percent MORE than the OPTIMAL solution
The worst solution costs 0.55 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 19139
The best solution costs 21.78 percent MORE than the OPTIMAL solution
The best solution costs 0.55 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 669 - path: [3, 108, 91, 5, 46, 89],
 Route - cost: 719 - path: [19, 58, 12, 41, 54, 50, 52],
 Route - cost: 1089 - path: [6, 109, 69, 18, 25, 103, 24, 84, 4, 101],
 Route - cost: 1405 - path: [38, 105, 32, 79, 1, 90, 16, 70, 100],
 Route - cost: 1335 - path: [71, 80, 30, 56, 55, 76, 83, 22, 61, 60],
 Route - cost: 1596 - path: [62, 8, 40, 47, 86, 10, 63, 72, 75],
 Route - cost: 1363 - path: [77, 11, 85, 51, 67, 45, 34, 106, 98],
 Route - cost: 1099 - path: [39, 28, 31, 17, 27, 65, 36, 99],
 Route - cost: 2283 - path: [2, 43, 93, 104, 64, 59, 95, 96, 35],
 Route - cost: 1280 - path: [48, 23, 13, 57, 82, 107, 7],
 Route - cost: 1899 - path: [102, 33, 9, 87, 26, 53, 15, 66, 42],
 Route - cost: 2259 - path: [78, 29, 44, 81, 94, 37, 73, 21],
 Route - cost: 2143 - path: [92, 97, 20, 74, 14, 49, 68, 88]]
```


1000
800
600
400
200
0
0
200
400
600
800
1000

8.13 Local Search, Greedy, X-n115-k10

```
$ python3 cvrp.py local_search greedy vrp/X-n115-k10.vrp
```

```
The algorithm local_search ran for 1 times !

Total execution time: 22.261 seconds
Optimal solution cost: 12747
Initial solution cost: 17711

# AVERAGE SOLUTION ANALYSIS #

average cost: 17172
The average solution costs 25.77 percent MORE than the OPTIMAL solution
The average solution costs 3.04 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 17172
The worst solution costs 25.77 percent MORE than the OPTIMAL solution
The worst solution costs 3.04 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 17172
The best solution costs 25.77 percent MORE than the OPTIMAL solution
The best solution costs 3.04 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 1493 - path: [15, 32, 89, 17, 66, 16, 104, 41, 91, 2, 23, 65, 113, 58,
77],
 Route - cost: 2765 - path: [50, 29, 53, 78, 28, 100, 57, 92, 107, 95, 59, 42, 93,
18, 109, 47, 79, 49, 99, 35, 101, 54, 30, 63],
 Route - cost: 866 - path: [13, 70, 36, 56, 31, 11, 81, 33],
 Route - cost: 2007 - path: [111, 110, 112, 105, 27, 44, 75, 84, 83, 8, 14, 68, 43,
40, 67, 39, 90, 37, 71, 98, 24],
 Route - cost: 1226 - path: [85, 34, 74, 1, 7, 97],
 Route - cost: 2444 - path: [10, 106, 73, 87, 86, 60, 48, 108, 19, 51, 25, 22, 96,
45, 38, 61],
 Route - cost: 2581 - path: [52, 9, 62, 20, 94, 114, 88, 82, 21, 46, 64, 26, 72, 102,
69, 103, 76, 55, 80],
 Route - cost: 1093 - path: [3, 5],
 Route - cost: 926 - path: [4],
 Route - cost: 1771 - path: [6, 12]]
```

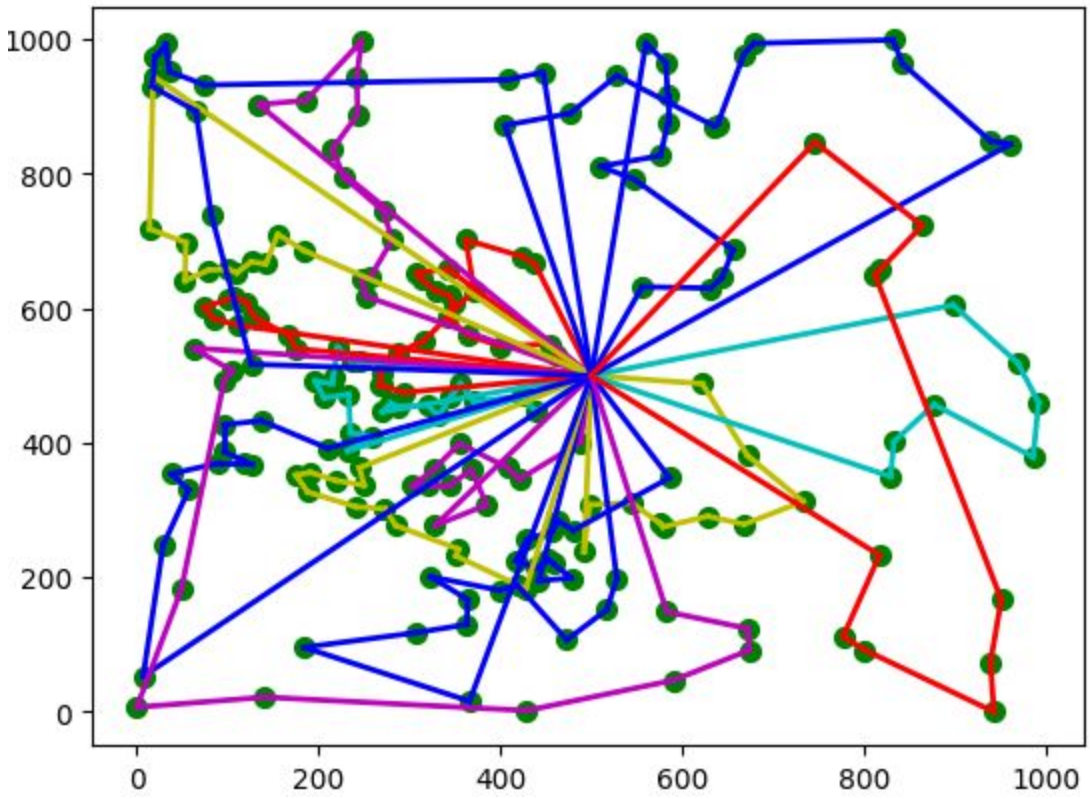

1000
800
600
400
200
0
0
200
400
600
800
1000

8.14 Local Search, Greedy, X-n204-k19

```
$ python3 cvrp.py local_search greedy vrp/X-n204-k19.vrp
```

```
The algorithm local_search ran for 1 times !

Total execution time: 83.844 seconds
Optimal solution cost: 19565
Initial solution cost: 23844

# AVERAGE SOLUTION ANALYSIS #

average cost: 23551
The average solution costs 16.92 percent MORE than the OPTIMAL solution
The average solution costs 1.23 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 23551
The worst solution costs 16.92 percent MORE than the OPTIMAL solution
The worst solution costs 1.23 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 23551
The best solution costs 16.92 percent MORE than the OPTIMAL solution
The best solution costs 1.23 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 532 - path: [145, 142, 22, 125, 117, 81, 44, 10, 102, 14, 147],
 Route - cost: 555 - path: [90, 194, 5, 151, 176, 109, 2, 124, 70, 62],
 Route - cost: 824 - path: [96, 173, 177, 170, 106, 155, 123, 131, 167, 65, 169],
 Route - cost: 923 - path: [58, 17, 43, 50, 45, 38, 189, 26, 183, 160],
 Route - cost: 1210 - path: [61, 42, 21, 73, 74, 51, 3, 87, 13, 25, 91],
 Route - cost: 829 - path: [63, 153, 77, 79, 186, 120, 105, 157, 122, 113, 162],
 Route - cost: 751 - path: [141, 140, 185, 201, 118, 98, 159, 193, 114, 190, 104],
 Route - cost: 820 - path: [188, 172, 1, 133, 127, 154, 195, 132, 18, 152, 182],
 Route - cost: 1393 - path: [76, 138, 191, 137, 55, 197, 139, 156, 110, 66, 89, 93],
 Route - cost: 1039 - path: [150, 181, 202, 12, 136, 135, 198, 16, 164, 143, 128,
71],
 Route - cost: 1579 - path: [83, 75, 49, 179, 99, 32, 72, 30, 174, 78, 52],
 Route - cost: 1577 - path: [200, 199, 203, 144, 148, 116, 119, 48, 36, 101, 34],
 Route - cost: 911 - path: [8, 166, 134, 68, 126, 192, 149, 47, 129, 121],
 Route - cost: 1283 - path: [29, 95, 67, 7, 84, 39, 33],
```

```
Route - cost: 2179 - path: [19, 20, 46, 80, 27, 100, 15, 53, 111, 187, 112],
Route - cost: 1532 - path: [115, 168, 97, 196, 108, 178, 107, 171, 161, 130, 184],
Route - cost: 1790 - path: [9, 175, 57, 163, 103, 146, 158, 180, 165, 40, 94],
Route - cost: 1729 - path: [56, 11, 54, 41, 31, 59, 4, 23, 24, 28, 6],
Route - cost: 2095 - path: [88, 82, 35, 37, 60, 64, 85, 92, 86, 69]]
```

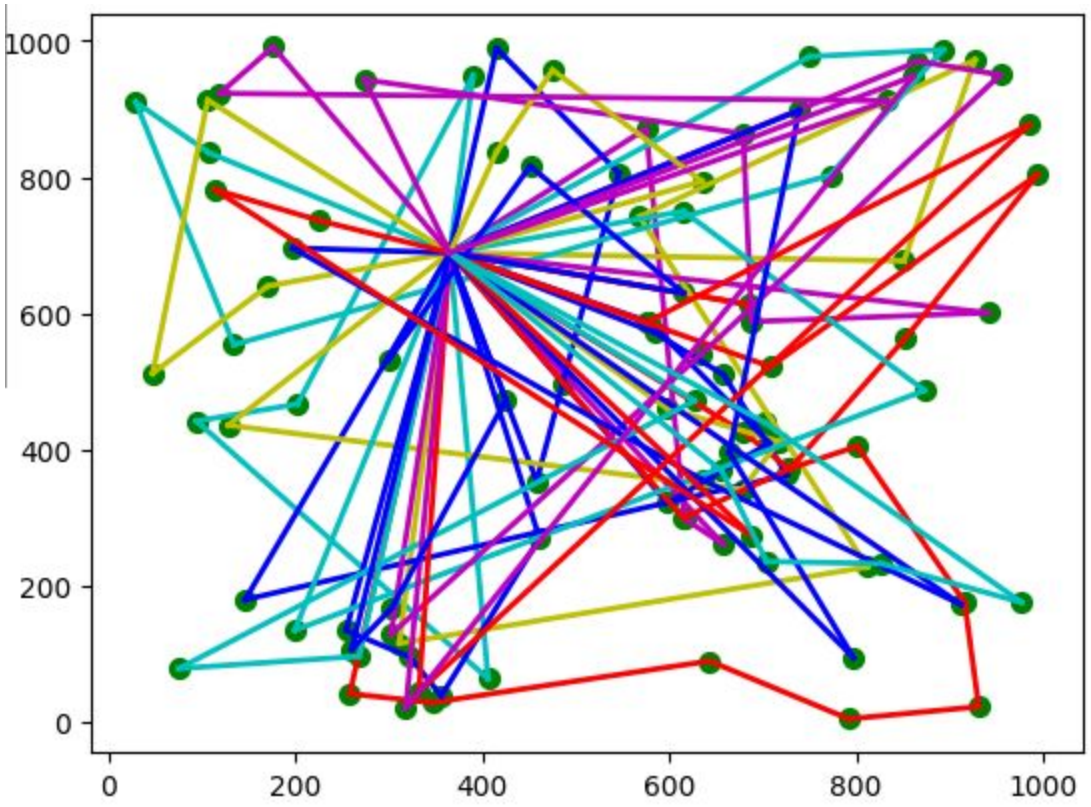


8.15 Local Search, Naive, X-n101-k25, statistic

```
$ python3 cvrp.py local_search naive vrp/X-n101-k25.vrp -n 15
```

```
The algorithm local_search ran for 15 times !

Total execution time: 429.835 seconds
Average execution time: 28.656 seconds
Optimal solution cost: 27591
Initial solution cost: 63974

# AVERAGE SOLUTION ANALYSIS #

average cost: 49053
The average solution costs 43.75 percent MORE than the OPTIMAL solution
The average solution costs 23.32 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 53497
The worst solution costs 48.43 percent MORE than the OPTIMAL solution
The worst solution costs 16.38 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 45744
The best solution costs 39.68 percent MORE than the OPTIMAL solution
The best solution costs 28.50 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 656 - path: [56],
 Route - cost: 2004 - path: [30, 15, 41, 13],
 Route - cost: 1424 - path: [19, 10, 65],
 Route - cost: 2872 - path: [90, 6, 21, 62, 38],
 Route - cost: 1336 - path: [18, 23, 85],
 Route - cost: 1667 - path: [1, 82, 3, 34],
 Route - cost: 2327 - path: [9, 69, 7, 2, 45, 29, 87, 57, 88],
 Route - cost: 2301 - path: [81, 71, 27, 35, 73, 95],
 Route - cost: 1792 - path: [75, 97, 94, 47],
 Route - cost: 1437 - path: [22, 28, 60, 44],
 Route - cost: 1029 - path: [17, 67, 39],
 Route - cost: 1391 - path: [4, 74, 76, 68],
 Route - cost: 1883 - path: [96, 89, 26, 72, 25, 58],
 Route - cost: 1963 - path: [86, 48, 100],
```

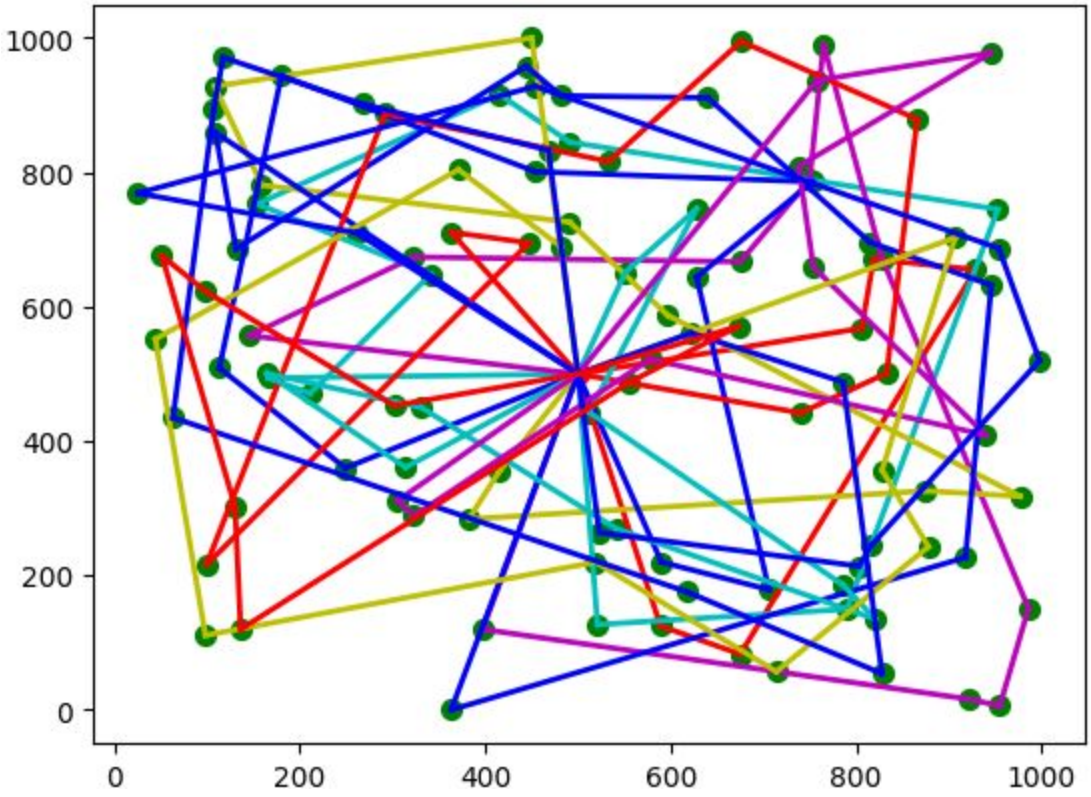
```
Route - cost: 2275 - path: [51, 99, 64, 84],
Route - cost: 1125 - path: [53, 20, 46],
Route - cost: 1804 - path: [36, 63, 24],
Route - cost: 1998 - path: [91, 77, 49],
Route - cost: 2436 - path: [55, 98, 80],
Route - cost: 1809 - path: [40, 70, 92],
Route - cost: 2302 - path: [83, 16],
Route - cost: 807 - path: [79, 11, 61],
Route - cost: 997 - path: [5, 50, 8],
Route - cost: 1217 - path: [54, 66, 12],
Route - cost: 1560 - path: [32, 31, 42],
Route - cost: 1656 - path: [14, 78, 37, 43, 59],
Route - cost: 1676 - path: [52, 33, 93]]
```



8.16 Local Search, Naive, X-n110-k13, statistic

```
$ python3 cvrp.py local_search naive vrp/X-n110-k13.vrp -n 15
```

```
The algorithm local_search ran for 15 times !

Total execution time: 1931.241 seconds
Average execution time: 128.749 seconds
Optimal solution cost: 14971
Initial solution cost: 59366

# AVERAGE SOLUTION ANALYSIS #

average cost: 36936
The average solution costs 59.47 percent MORE than the OPTIMAL solution
The average solution costs 37.78 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 38583
The worst solution costs 61.20 percent MORE than the OPTIMAL solution
The worst solution costs 35.01 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 34845
The best solution costs 57.04 percent MORE than the OPTIMAL solution
The best solution costs 41.30 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 1632 - path: [86, 73, 92, 45, 105],
 Route - cost: 2958 - path: [10, 82, 42, 24, 102, 43, 71, 39, 28],
 Route - cost: 3246 - path: [63, 37, 94, 81, 34, 53, 85, 46, 80, 27],
 Route - cost: 2823 - path: [19, 58, 16, 29, 91, 18, 93, 61, 9],
 Route - cost: 3278 - path: [72, 44, 20, 67, 87, 4, 33, 99, 104],
 Route - cost: 2613 - path: [40, 48, 5, 51, 25, 83, 17, 54],
 Route - cost: 3063 - path: [1, 32, 98, 26, 84, 55, 88, 69, 30],
 Route - cost: 2238 - path: [6, 77, 49, 57, 107, 8, 52, 31, 68],
 Route - cost: 2451 - path: [41, 12, 3, 78, 89, 11, 66, 15],
 Route - cost: 3071 - path: [109, 56, 59, 75, 100, 35, 70, 90, 106],
 Route - cost: 2854 - path: [103, 76, 22, 60, 95, 47, 7, 79, 108],
 Route - cost: 2628 - path: [62, 13, 23, 74, 97, 101, 64, 2],
 Route - cost: 1990 - path: [50, 65, 36, 96, 21, 14, 38]]
```


1000
800
600
400
200
0
0
200
400
600
800
1000

8.17 Local Search, Naive, X-n115-k10, statistic

```
$ python3 cvrp.py local_search naive vrp/X-n115-k10.vrp -n 15
```

```
The algorithm local_search ran for 15 times !

Total execution time: 2272.781 seconds
Average execution time: 151.519 seconds
Optimal solution cost: 12747
Initial solution cost: 56459

# AVERAGE SOLUTION ANALYSIS #

average cost: 28822
The average solution costs 55.77 percent MORE than the OPTIMAL solution
The average solution costs 48.95 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 30221
The worst solution costs 57.82 percent MORE than the OPTIMAL solution
The worst solution costs 46.47 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 27422
The best solution costs 53.52 percent MORE than the OPTIMAL solution
The best solution costs 51.43 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 4565 - path: [52, 110, 85, 98, 80, 97, 38, 65, 33, 61, 103, 56, 72,
26, 89, 48, 60, 86, 106, 93, 49, 59, 78, 62, 92, 101, 20],
 Route - cost: 5004 - path: [13, 41, 17, 108, 51, 22, 96, 113, 58, 76, 71, 90, 39,
67, 43, 75, 44, 57, 79, 18, 95, 28, 88, 21, 112, 111, 46, 34],
 Route - cost: 4125 - path: [50, 29, 9, 54, 100, 107, 99, 109, 87, 32, 15, 104, 25,
23, 45, 31, 37, 40, 68, 83, 84, 64],
 Route - cost: 2348 - path: [63, 7, 11, 66, 47, 19],
 Route - cost: 1668 - path: [73, 6, 5, 69, 24],
 Route - cost: 3253 - path: [82, 35, 42, 16, 2, 36, 70, 102, 74, 14, 105, 114, 27],
 Route - cost: 3244 - path: [53, 4, 94, 8, 91, 81, 55],
 Route - cost: 2057 - path: [30, 1, 77, 12],
 Route - cost: 710 - path: [3],
 Route - cost: 448 - path: [10]]
```

8.18 Local Search, Naive, X-n204-k19, statistic

```
$ python3 cvrp.py local_search naive vrp/X-n204-k19.vrp -n 15
```

```
The algorithm local_search ran for 15 times !

Total execution time: 13908.335 seconds
Average execution time: 927.222 seconds
Optimal solution cost: 19565
Initial solution cost: 96627

# AVERAGE SOLUTION ANALYSIS #

average cost: 55532
The average solution costs 64.77 percent MORE than the OPTIMAL solution
The average solution costs 42.53 percent LESS than the INITIAL solution

# WORST SOLUTION ANALYSIS #

worst cost: 58306
The worst solution costs 66.44 percent MORE than the OPTIMAL solution
The worst solution costs 39.66 percent LESS than the INITIAL solution

# BEST SOLUTION ANALYSIS #

best cost: 52935
The best solution costs 63.04 percent MORE than the OPTIMAL solution
The best solution costs 45.22 percent LESS than the INITIAL solution

Best solution::
[Route - cost: 1824 - path: [77, 113, 59, 25, 201, 81, 44],
 Route - cost: 3018 - path: [61, 26, 50, 88, 75, 99, 100, 123, 150, 133, 93],
 Route - cost: 2256 - path: [22, 140, 40, 197, 97, 149, 119, 135, 32, 52, 38],
 Route - cost: 4280 - path: [6, 73, 117, 158, 126, 112, 9, 48, 15, 179, 189, 35],
 Route - cost: 2887 - path: [69, 139, 55, 102, 121, 200, 167, 122, 105, 101, 136],
 Route - cost: 3147 - path: [90, 131, 12, 193, 57, 66, 29, 64, 82, 46, 19],
 Route - cost: 2599 - path: [42, 3, 13, 4, 165, 111, 76, 10, 176, 16, 164],
 Route - cost: 3170 - path: [191, 168, 161, 114, 62, 65, 49, 60, 21, 95],
 Route - cost: 2917 - path: [83, 71, 174, 2, 118, 127, 108, 94, 87, 86, 28],
 Route - cost: 3524 - path: [145, 143, 30, 53, 181, 166, 130, 184, 24, 85, 33],
 Route - cost: 3178 - path: [188, 107, 110, 56, 41, 92, 7, 17, 20, 27, 79, 173],
 Route - cost: 2290 - path: [142, 152, 116, 144, 192, 178, 89, 138, 125, 147, 31],
 Route - cost: 2584 - path: [194, 159, 1, 115, 196, 68, 199, 106, 34, 80, 72],
 Route - cost: 2313 - path: [146, 103, 175, 129, 154, 182, 202, 198, 162, 58, 63],
```

```
Route - cost: 2798 - path: [11, 163, 203, 195, 172, 104, 170, 43, 160, 169],
Route - cost: 2573 - path: [84, 45, 128, 155, 109, 18, 8, 98, 14, 91],
Route - cost: 3014 - path: [157, 39, 23, 54, 156, 180, 137, 151, 5, 124],
Route - cost: 2881 - path: [96, 183, 153, 37, 67, 74, 51, 141, 185, 148, 70],
Route - cost: 1682 - path: [177, 186, 120, 78, 36, 187, 47, 171, 134, 132, 190]]
```