Relatório intermediário

Supercomputação - INSPER - 7° Semestre

O problema

O problema `Maximin Share` é um problema *NP-difícil*. Sua resolução visa encontrar a distribuição mais igualitária possível de `N` objetos para `M` pessoas. É possível que determinadas pessoas acabem com mais objetos que outras ou também com maior valor acumulado que outras, uma vez que os objetos são indivisíveis. Esse problema, portanto, visa definir qual o *menor* valor que uma pessoa deveria aceitar. Basicamente, queremos maximizar o valor da pessoa que terá o menor valor. Esse é o chamado MMS.

Para exemplificar o problema, temos 6 objetos e 3 pessoas. Cada objeto possui um valor associado a ele: `{20, 11, 9, 13, 14, 37}`. Uma possível distribuição seria:

```
{37}
{20, 11}
{14, 13, 9}
```

Tendo em vista que cada linha representa os objetos que cada pessoa recebeu, o MMS é 31 (pessoa do meio). No entanto, é possível realizar divisões onde teremos um melhor MMS, como é o caso dessa distribuição:

```
{37}
{20, 14}
{13, 11, 9}
```

Aqui, temos MMS equivalente a 33 (última pessoa), que já é melhor que o cenário anterior.

Entradas e saídas

Por padrão, todas as entradas seguem o formato:

```
N M
v1 ... vN
```

Sendo `N` o número de objetos, `M` o número de pessoas e `vi` o valor do objeto `i`.

O formato da saída desse programa é como segue:

```
MMS
objetos da pessoa 1
...
objetos da M
```

Sendo `MMS` o valor do grupo menos valioso e `objetos da pessoa j` a lista de índices dos objetos que a pessoa `j` recebeu nessa distribuição.

As soluções

Para resolver esse problema, diferentes soluções foram implementadas:

- Heurística: cada pessoa deve receber ao menos `N/M` objetos (arredondado para baixo). Para a realização da divisão, ordena-se os itens por valor (decrescente) e atribua-os para cada pessoa sequencialmente. Ao chegar na última pessoa, deve-se voltar para a primeira e continuar a distribuição até acabar os objetos.
  - Para essa solução, foram criados structs de apoio e uma função responsável por ordenar os itens de forma a respeitar a heurística descrita:

```
typedef struct
{
  std::vector<int> itensIds;
  int idsIdx;
  int totalValue;
} Person;
typedef struct
{
  int id;
  int value;
} Item;

void mais_valioso(std::vector<Item> &itens)
{
  std::sort(itens.begin(), itens.end(), [](const auto &i, const auto &j) { return i.value >
j.value; });
}
```

  - Após a criação de vetores para os itens e para as pessoas, a distribuição dos itens pode ser realizada:

```
// ordenando do mais valioso para o menos valioso
mais_valioso(itens);

//faz a distribuição segundo a heurística
int actualPersonIdx = 0;
int idx;
for (int i = 0; i < N; i++)
{
  idx = person[actualPersonIdx].idsIdx;
  person[actualPersonIdx].itensIds[idx] = itens[i].id;
  person[actualPersonIdx].totalValue += itens[i].value;

  person[actualPersonIdx].idsIdx++;
  actualPersonIdx++;
  if (actualPersonIdx == M)
      actualPersonIdx = 0;
}
```

  - Para finalizar, basta identificar a pessoa de menor valor, que será a última da pessoa lista, assim obtem-se o MMS:

```
int MMS = person.back().totalValue;
```

- Busca Local:
  - Foi criado um struct para guardar a melhor resposta encontrada. `distribution` terá tamanho `N` (n° de objetos), e será o vetor que guarda o id da pessoa que recebeu o item na posição `i` do vetor. Exemplo: se tivermos `distribution = {0, 0, 1, 2}`, a pessoa 0 recebeu os itens 0 e 1, a pessoa 1 recebeu o item 2, e a pessoa 2 recebeu o item 3.

```
typedef struct
{
  int MMS;
  std::vector<int> distribution;
} Answer;
```

  - Os passos para a solução local estão descritos a seguir:

1. Atribuir os objetos para pessoas aleatorias;

```
std::default_random_engine generator;
generator.seed(SEED);
std::uniform_int_distribution<int> randomPersonSelector(0, M - 1);

// Passo 1: cada objeto é atribuído para uma pessoa aleatória
for (int i = 0; i < N; i++)
{
 personIdx = randomPersonSelector(generator);
 itens[i].owner = personIdx;
 person[personIdx].totalValue += itens[i].value;
}
```

2. Selecionar a pessoa (`P`) com menor valor. Passar por todos os outros objetos e verificar se pode ser doado para `P`; Se for possível, deve-se realizar a doação e calcular o novo MMS.

  - Vale ressaltar que no passo 2 um objeto pode ser doado se o valor total do doador removendo o item doado é maior que o valor total da pessoa `P`.
  - Nessa implementação, o cálculo do novo MMS é "feito" utilizando `min_value`, assim é possível obter o índice da pessoa `P` facilmente. Como temos o índice dessa pessoa, para obter o MMS basta acessar o vetor de valores de cada pessoa (`personValue`) no índice `P`.

```
// Passo 2:
for (int j = 0; j < N; j++)
{
  // encontra dono atual
  int currentOwner = itensOwners[j];

  bool canBeDonated = personValue[currentOwner] - itensValues[j] > personValue[P];

  // se pode ser doado
  if (canBeDonated)
  {
      donationHappened = true;

      //realiza doação
      personValue[currentOwner] -= itensValues[j];
      itensOwners[j] = P;
      personValue[P] += itensValues[j];

      // encontra nova pessoa P de menor valor (idx P, pessoa com menor valor)
      P = std::min_element(personValue.begin(), personValue.end()) - personValue.begin();
      if (personValue[P] > ans.MMS)
      {
          ans.MMS = personValue[P];
          for (int k = 0; k < N; k++)
              ans.distribution[k] = itensOwners[k];
      }
  }
}
```

1. Repetir a etapa 2 até não ser mais possível. Quando passar por todos os itens e nenhuma doação acontecer, temos uma indicação de que não são possíveis mais doações.

```
while (1)
 {
     donationHappened = false;

     [... PASSO 2 ...]

     if (!donationHappened)
            break;
 }
```

2. As etapas anteriores devem ser repetida algumas vezes para termos uma maior chance de encontrar um ponto de máximo local equivalente ao máximo global.

- Busca exaustiva: utilizando recursão, é possível gerar todas as distribuições possíveis, dados N itens e M pessoas. Basta escolher a distribuição em que maximizamos o MMS.
  - Temos M^N distribuições possíveis.
  - Ao realizar todas as possíveis distribuições, basta encontrar aquela onde tem-se o máximo MMS.
  - Para 3 itens e 2 pessoas (representadas por 0 e 1) temos as seguintes possíveis distribuições:

Item 1	Item 2	Item 3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

- Para obter todas as distribuições possíveis e calcular o MMS na busca exaustiva, o struct e a função de apoio foram criadas:

```
typedef struct
  {
      int mms;
      std::vector<int> distribution;
  } Answer;

  int calculateMinValue(std::vector<int> indexes, std::vector<int> values, int N, int M)
  {
      std::vector<int> valuePerPerson(M, 0);
      for (int i = 0; i < M; i++)
          for (int j = 0; j < N; j++)
              if (indexes[j] == i)
                  valuePerPerson[i] += values[j];

      int min = *std::min_element(valuePerPerson.begin(), valuePerPerson.end());
      return min;
  }
```

  - A função recursiva responsável por gerar as distribuições é como segue:

```
void MMS(std::vector<int> &ownersIdx, std::vector<int> &V, int N, int M, int i, long
&analysedAnswers, Answer &best)
{
  if (i == N)
  {
      analysedAnswers++;

      int newMms = calculateMinValue(ownersIdx, V, N, M);

      if (best.mms < newMms)
      {
          best.distribution = ownersIdx;
          best.mms = newMms;
      }

      return;
  }

  for (int j = 0; j < M; j++)
  {
      ownersIdx[i] = j;
      MMS(ownersIdx, V, N, M, i + 1, analysedAnswers, best);
  }
}
```

Máquina utilizada para os testes

Processador: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz - 4 cores

RAM: 8Gb

GPU: GeForce 940MX

Como as soluções apresentadas até agora não são algoritmos que são executados em paralelo, as especificações da máquina não são tão relevantes para os resultados apresentados a seguir.

Testes e funções de apoio

Para avaliar o desempenho das soluções anteriores foram geradas entradas variando `N` (e `M` fixo) e outras com `M` variando (e `N` fixo). Nesse documento serão executados testes relacionados ao tempo de execução e qualidade da solução; a forma com que os itens foram distribuídos para cada pessoa não importa para sabermos a qualidade da solução, o MMS obtido na saída já é o suficiente para uma boa visualização e entendimento do problema. Também foram criadas funções para automatizar a execução das soluções anteriores com entradas de teste (geradas através de um script python, seguindo a formatação de uma entrada padrão).

In [1]:

```
%matplotlib inline
import time
import subprocess
import pandas as pd
import matplotlib.pyplot as plt
import glob, os
```

In [2]:

```
execs = [
    'heuristico/heuristico',
    'busca-local/local',
    'busca-global/global',
]

def getInputFiles(str_format, directory="testes_relatorio/"):
    ins = []
    os.chdir(directory)
    for file in glob.glob(str_format):
        ins += [directory + file]
    os.chdir('..')
    return ins
```

Aqui é interessante ressaltar a função `executeAndParseResults()` para a execução ("timeout"). Isso ocorre pois o tempo de computação para a busca exaustiva cresce exponencialmente, passando de meros minutos para inúmeras horas de execução. Para mitigar esses casos, foi decidido cortar a execução em uma menor quantidade de tempo. Assim, para a análise da solução exaustiva, foram utilizadas entradas adequadas para conseguir executar os testes em um tempo adequado.

In [3]:

```
def executeAndParseResults(executavel, arquivo_in):
    with open(f'{arquivo_in}') as f:
        entrada = f.read()
        entrada__ = entrada.split()
        entrada__r = {'N': int(entrada__[0]), 'M': int(entrada__[1])}
        timeout = 600 if (entrada__r['N'] <= 13 and entrada__r['M'] <= 6) else 3
        start = time.perf_counter()
        try:
            proc = subprocess.run([executavel], input=entrada, text=True, capture_output=True, timeout=timeout)
        except:
            return None
        end = time.perf_counter()

    entrada = entrada.split()
    entrada_r = {'N': entrada[0], 'M': entrada[1]}

    std = proc.stdout
    stdout = std.split()

    resultados = {'MMS': stdout[0]}

    execTime = end-start
    return (entrada_r, resultados, execTime)

def geraDF(executavel, ins):
    resultados = []
    for i in ins:
        res = executeAndParseResults(executavel, i)
        if res != None:
            resultados.append(res)

    df = pd.DataFrame()
    df['T'] = [i[2] for i in resultados]
    df['N'] = [int(i[0]['N']) for i in resultados]
    df['M'] = [int(i[0]['M']) for i in resultados]
    df['MMS'] = [int(i[1]['MMS']) for i in resultados]

    return df
```

Testes executados em todas as soluções, com M fixo:

In [4]:

```
insGlobal_M_fixo = getInputFiles("in*_M_global.txt")

resHeuristicoGlobal_M_fixo = geraDF(execs[0], insGlobal_M_fixo)
resLocalGlobal_M_fixo = geraDF(execs[1], insGlobal_M_fixo)
resGlobalGlobal_M_fixo = geraDF(execs[2], insGlobal_M_fixo)

dfGlobalM = pd.DataFrame()
dfGlobalM['N'] = resLocalGlobal_M_fixo['N']
dfGlobalM['M'] = resLocalGlobal_M_fixo['M']

dfGlobalM['MMS_Heuristico'] = resHeuristicoGlobal_M_fixo['MMS']
dfGlobalM['T_Heuristico'] = resHeuristicoGlobal_M_fixo['T']

dfGlobalM['MMS_Local'] = resLocalGlobal_M_fixo['MMS']
dfGlobalM['T_Local'] = resLocalGlobal_M_fixo['T']

dfGlobalM['MMS_Global'] = resGlobalGlobal_M_fixo['MMS']
dfGlobalM['T_Global'] = resGlobalGlobal_M_fixo['T']
```

In [5]:

```
dfGlobalM.sort_values('N')
```

Out[5]:

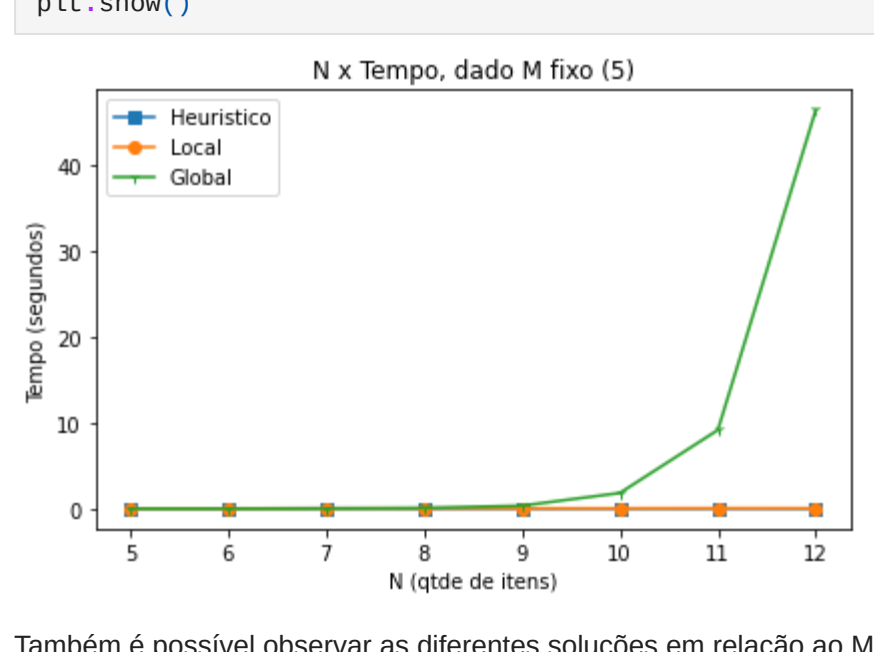
	N	M	MMS_Heuristico	T_Heuristico	MMS_Local	T_Local	MMS_Global	T_Global
7	5	5	7	0.009989	7	0.029565	7	0.005706
3	6	5	46	0.011840	51	0.031246	51	0.007481
0	7	5	32	0.044015	44	0.050509	44	0.017517
2	8	5	35	0.010114	66	0.042905	66	0.086378
1	9	5	43	0.012990	69	0.046039	69	0.351414
6	10	5	49	0.010701	77	0.049378	77	1.827927
4	11	5	64	0.010415	97	0.063842	97	9.168681
5	12	5	81	0.010539	109	0.060002	109	46.469911

Os resultados obtidos para um M fixo podem ser observados na tabela anterior. O que mais chama atenção nesse resultados é o tempo de execução da solução global, que pula de aproximadamente 9,36 segundos para 47,12 segundos.

A visualização do crescimento do tempo de execução dessa solução pode ser observada no gráfico abaixo. Nele, podemos ver que se comporta quase como uma exponencial. Caso fosse viável executar testes em maiores quantidades, poderíamos observar que a busca do melhor MMS entre M^N cenários (processados na busca exaustiva), passaria de minutos para horas, de horas para dias, e assim em diante.

In [6]:

```
dfGlobalM = dfGlobalM.sort_values('N')

plt.plot(dfGlobalM['N'], dfGlobalM['T_Heuristico'], label="Heuristico", marker="s")
plt.plot(dfGlobalM['N'], dfGlobalM['T_Local'], label="Local", marker="o")
plt.plot(dfGlobalM['N'], dfGlobalM['T_Global'], label="Global", marker="1")
plt.xlabel('N (qtde de itens)')
plt.ylabel('Tempo (segundos)')
plt.title("N x Tempo, dado M fixo (5)")
plt.legend()
plt.tight_layout()
plt.show()
```

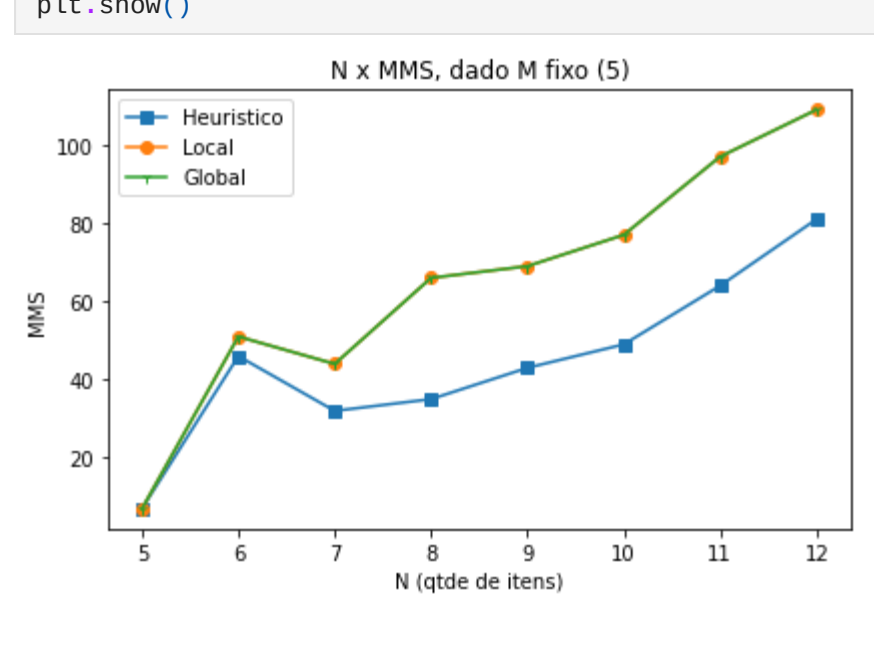


Também é possível observar as diferentes soluções em relação ao MMS da solução. A busca global possui MMS ótimo, como esperado. No entanto, a busca local não fica atrás dela.

Como a estratégia delimitada envolve repetir a busca local, a chance de obtermos um ponto de máximo local que equivale ao global é alto. Ao mesmo tempo que apresenta MMS ótimo, o tempo e os recursos consumidos pela local não chegam perto da global. Isso pode ser visto principalmente em entradas com muitos itens ou muitas pessoas, onde é viável executar a busca local mas não é viável executar a exaustiva.

In [7]:

```
plt.plot(dfGlobalM['N'], dfGlobalM['MMS_Heuristico'], label="Heuristico", marker="s")
plt.plot(dfGlobalM['N'], dfGlobalM['MMS_Local'], label="Local", marker="o")
plt.plot(dfGlobalM['N'], dfGlobalM['MMS_Global'], label="Global", marker="1")
plt.xlabel('N (qtde de itens)')
plt.ylabel('MMS')
plt.title("N x MMS, dado M fixo (5)")
plt.legend()
plt.tight_layout()
plt.show()
```



Testes executados em todas as soluções, com N fixo:

In [8]:

```
insGlobal_N_fixo = getInputFiles("in*_N_global.txt")

resHeuristicoGlobal_N_fixo = geraDF(execs[0], insGlobal_N_fixo)
resLocalGlobal_N_fixo = geraDF(execs[1], insGlobal_N_fixo)
resGlobalGlobal_N_fixo = geraDF(execs[2], insGlobal_N_fixo)

dfGlobalN = pd.DataFrame()
dfGlobalN['N'] = resLocalGlobal_N_fixo['N']
dfGlobalN['M'] = resLocalGlobal_N_fixo['M']

dfGlobalN['MMS_Heuristico'] = resHeuristicoGlobal_N_fixo['MMS']
dfGlobalN['T_Heuristico'] = resHeuristicoGlobal_N_fixo['T']

dfGlobalN['MMS_Local'] = resLocalGlobal_N_fixo['MMS']
dfGlobalN['T_Local'] = resLocalGlobal_N_fixo['T']

dfGlobalN['MMS_Global'] = resGlobalGlobal_N_fixo['MMS']
dfGlobalN['T_Global'] = resGlobalGlobal_N_fixo['T']
```

In [9]:

```
dfGlobalN.sort_values('M')
```

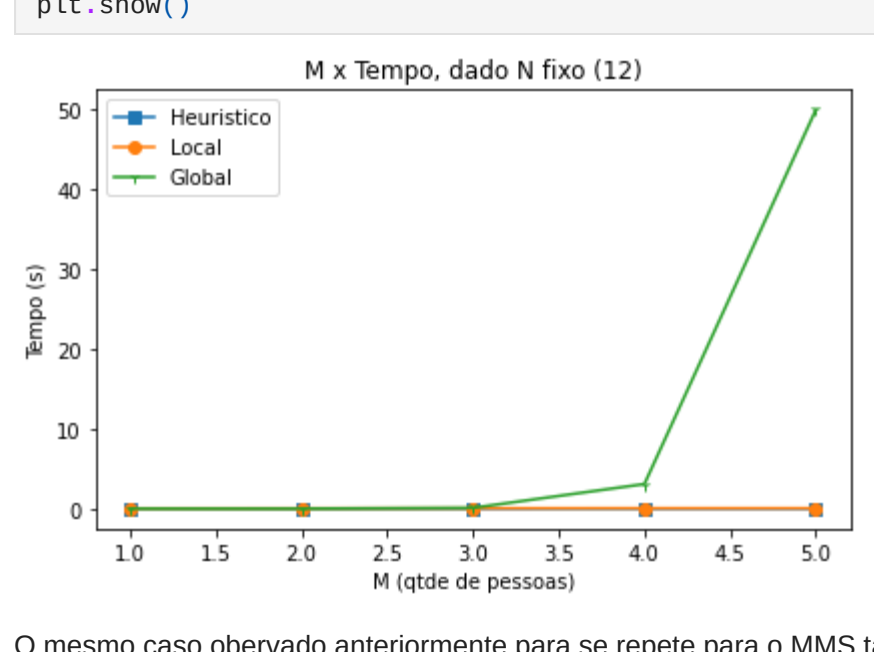
Out[9]:

	N	M	MMS_Heuristico	T_Heuristico	MMS_Local	T_Local	MMS_Global	T_Global
4	12	1	505	0.005259	505	0.036809	505	0.005467
3	12	2	271	0.005940	290	0.045980	290	0.006240
0	12	3	178	0.009029	201	0.055236	201	0.104710
1	12	4	123	0.007998	171	0.057449	171	3.085236
2	12	5	77	0.006530	118	0.059154	118	49.945414

Variando o M, a mesma constatação pode ser feita; o tempo de execução da busca exaustiva aumenta exponencialmente de um passo para outro. A mesma coisa pode ser observada no gráfico seguinte.

In [10]:

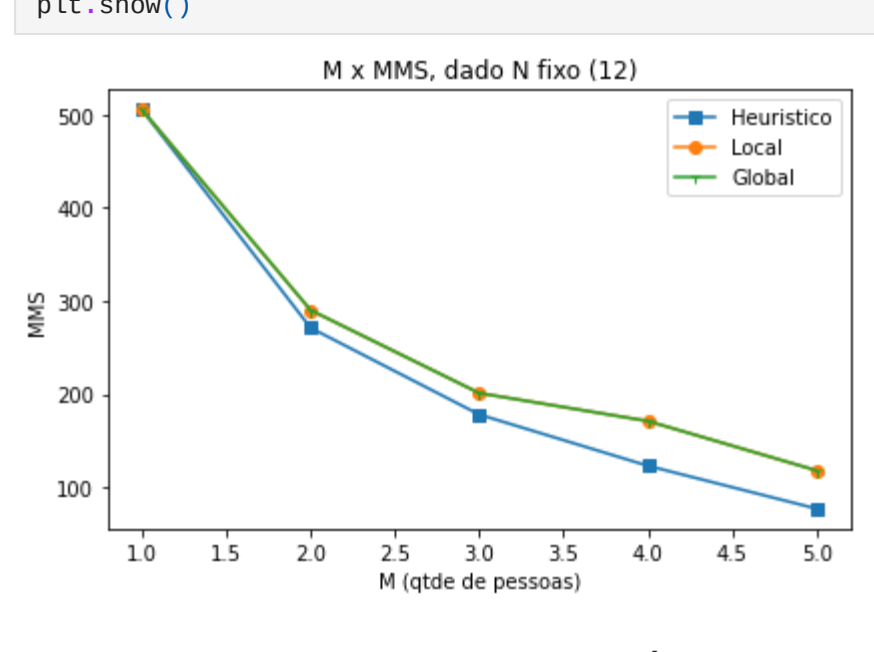
```
dfGlobalN = dfGlobalN.sort_values('M')

plt.plot(dfGlobalN['M'], dfGlobalN['T_Heuristico'], label="Heuristico", marker="s")
plt.plot(dfGlobalN['M'], dfGlobalN['T_Local'], label="Local", marker="o")
plt.plot(dfGlobalN['M'], dfGlobalN['T_Global'], label="Global", marker="1")
plt.xlabel('M (qtde de pessoas)')
plt.ylabel('Tempo (s)')
plt.title("M x Tempo, dado N fixo (12)")
plt.legend()
plt.tight_layout()
plt.show()
```



O mesmo caso obervado anteriormente para se repete para o MMS também.

In [11]:

```
plt.plot(dfGlobalN['M'], dfGlobalN['MMS_Heuristico'], label="Heuristico", marker="s")
plt.plot(dfGlobalN['M'], dfGlobalN['MMS_Local'], label="Local", marker="o")
plt.plot(dfGlobalN['M'], dfGlobalN['MMS_Global'], label="Global", marker="1")
plt.xlabel('M (qtde de pessoas)')
plt.ylabel('MMS')
plt.title("M x MMS, dado N fixo (12)")
plt.legend()
plt.tight_layout()
plt.show()
```



Testes executados na solução heurística e local, com M fixo:

In [12]:

```
ins_M_fixo = getInputFiles("in*_M.txt")

resHeuristico_M_fixo = geraDF(execs[0], ins_M_fixo)
resLocal_M_fixo = geraDF(execs[1], ins_M_fixo)

dfM = pd.DataFrame()
dfM['N'] = resLocal_M_fixo['N']
dfM['M'] = resLocal_M_fixo['M']

dfM['MMS_Heuristico'] = resHeuristico_M_fixo['MMS']
dfM['T_Heuristico'] = resHeuristico_M_fixo['T']

dfM['MMS_Local'] = resLocal_M_fixo['MMS']
dfM['T_Local'] = resLocal_M_fixo['T']
```

In [13]:

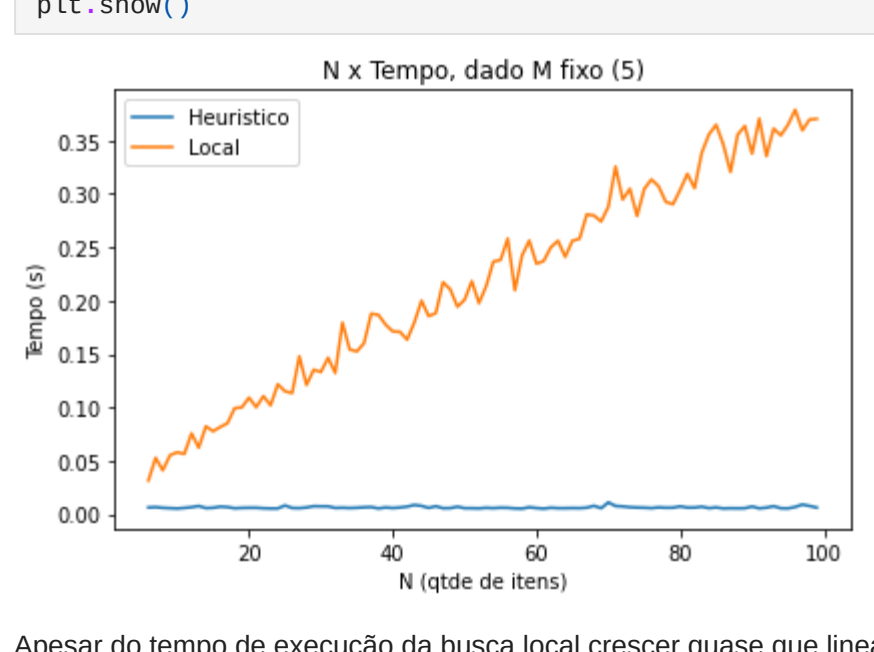
```
dfM.sort_values('N').head(5)
```

Out[13]:

	N	M	MMS_Heuristico	T_Heuristico	MMS_Local	T_Local
89	6	5	11	0.006298	20	0.031531
3	7	5	58	0.006611	68	0.052784
93	8	5	61	0.005949	93	0.041092
27	9	5	28	0.005636	63	0.055186
4	10	5	93	0.005338	104	0.057990

In [14]:

```
dfM = dfM.sort_values('N')

plt.plot(dfM['N'], dfM['T_Heuristico'], label="Heuristico")
plt.plot(dfM['N'], dfM['T_Local'], label="Local")
plt.xlabel('N (qtde de itens)')
plt.ylabel('Tempo (s)')
plt.title("N x Tempo, dado M fixo (5)")
plt.legend()
plt.tight_layout()
plt.show()
```

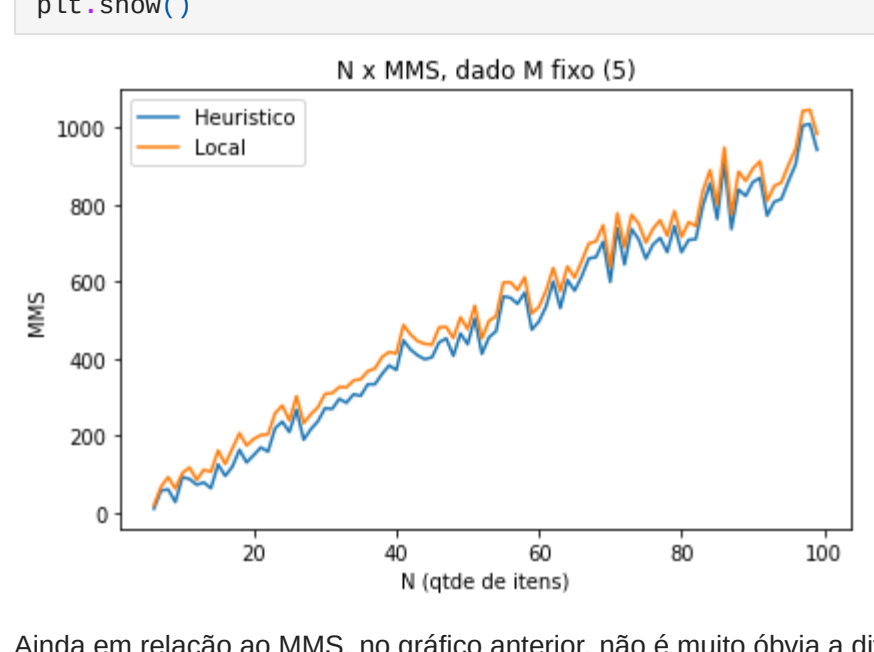


Apesar do tempo de execução da busca local crescer quase que linearmente, enquanto o tempo da heurística permanece praticamente constante, como mostra o gráfico acima, temos que o MMS local é melhor que o heurístico.

Retomando os cenários da busca global, demorariam milhares de horas para executar um solução para 5 pessoas e 90 itens, por exemplo. A busca local "brilha" nesse aspecto; temos uma solução ótima (ou pelo mesnos bem próximas da ótima), mesmo utilizando a busca local, mas que funciona para entradas grandes e é executada em um tempo muito menor que o global.

In [15]:

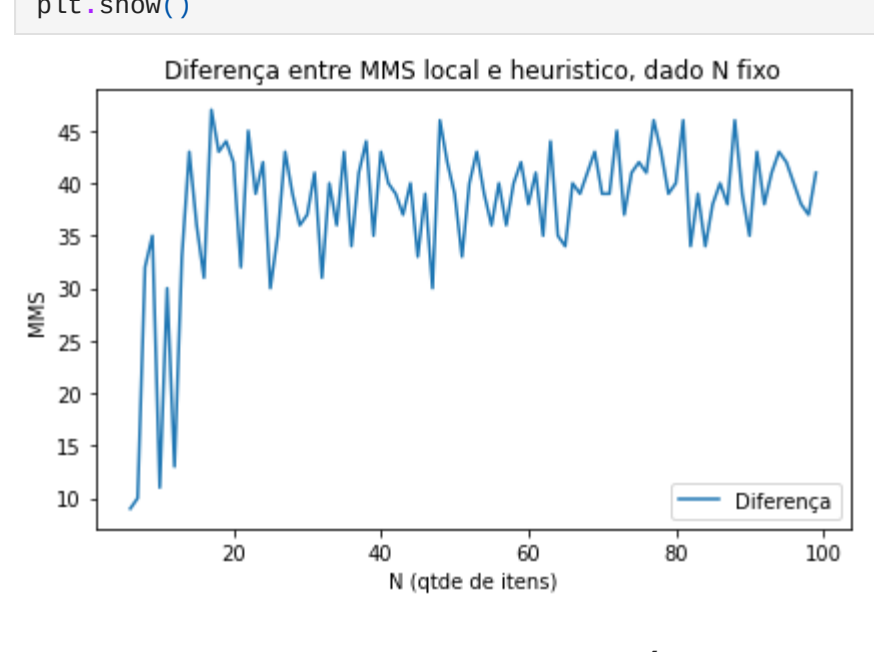
```
plt.plot(dfM['N'], dfM['MMS_Heuristico'], label="Heuristico")
plt.plot(dfM['N'], dfM['MMS_Local'], label="Local")
plt.xlabel('N (qtde de itens)')
plt.ylabel('MMS')
plt.title("N x MMS, dado M fixo (5)")
plt.legend()
plt.tight_layout()
plt.show()
```



Ainda em relação ao MMS, no gráfico anterior, não é muito óbvia a diferençaentre as duas soluções, dada a escala utilizada. No entanto, podemos observar essa variação no próximo gráfico. A diferença no MMS obtido pela local é significante.

In [16]:

```
plt.plot(dfM['N'], dfM['MMS_Local'] - dfM['MMS_Heuristico'], label="Diferença")
plt.xlabel('N (qtde de itens)')
plt.ylabel('MMS')
plt.title("Diferença entre MMS local e heuristico, dado N fixo")
plt.legend()
plt.tight_layout()
plt.show()
```



Testes executados na solução heurística e local, com N fixo:

In [17]:

```
ins_N_fixo = getInputFiles("in*_N.txt")

resHeuristico_N_fixo = geraDF(execs[0], ins_N_fixo)
resLocal_N_fixo = geraDF(execs[1], ins_N_fixo)

dfN = pd.DataFrame()
dfN['N'] = resLocal_N_fixo['N']
dfN['M'] = resLocal_N_fixo['M']

dfN['MMS_Heuristico'] = resHeuristico_N_fixo['MMS']
dfN['T_Heuristico'] = resHeuristico_N_fixo['T']

dfN['MMS_Local'] = resLocal_N_fixo['MMS']
dfN['T_Local'] = resLocal_N_fixo['T']
```

In [18]:

```
dfN.sort_values('M').head(5)
```

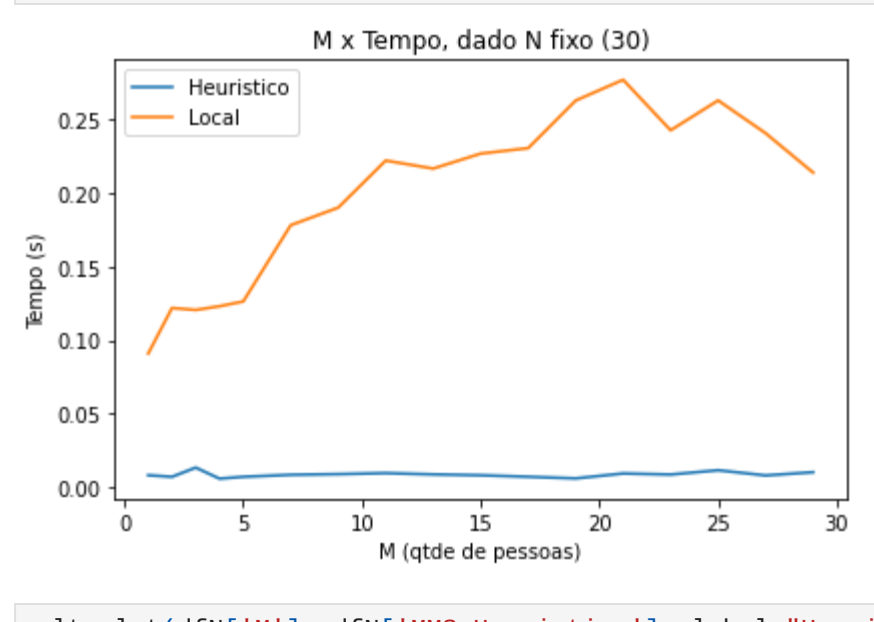
Out[18]:

	N	M	MMS_Heuristico	T_Heuristico	MMS_Local	T_Local
13	30	1	1711	0.008048	1711	0.090865
9	30	2	691	0.006972	714	0.121841
14	30	3	398	0.013245	433	0.120498
4	30	4	400	0.005724	434	0.123023
5	30	5	285	0.006978	332	0.126278

Como visto para os testes com M fixo, o tempo que a busca local leva para ser executada cresce com o tamanho das entradas.

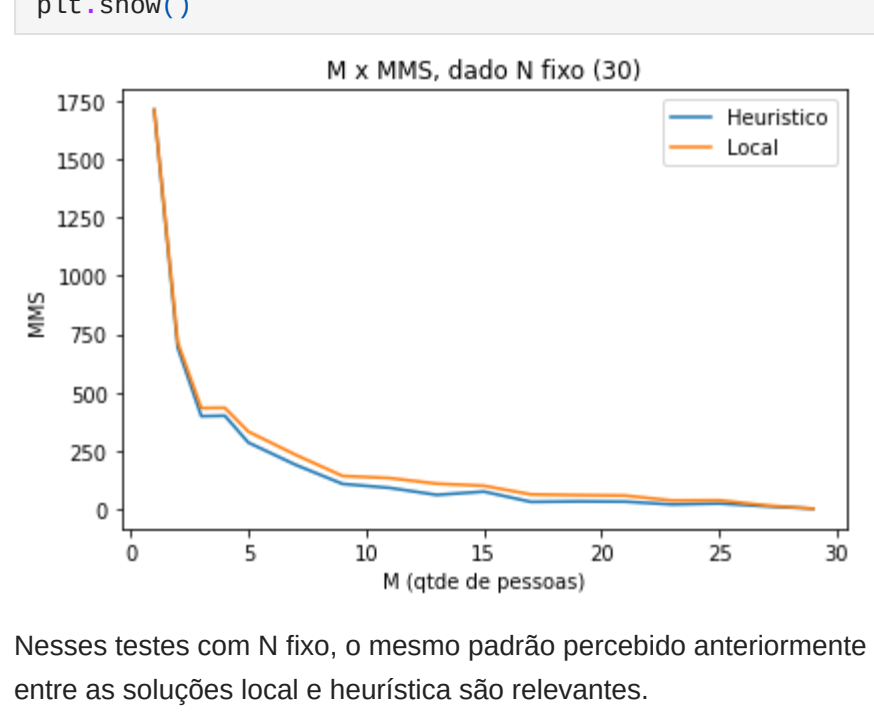
In [19]:

```
dfN = dfN.sort_values('M')

plt.plot(dfN['M'], dfN['T_Heuristico'], label="Heuristico")
plt.plot(dfN['M'], dfN['T_Local'], label="Local")
plt.xlabel('M (qtde de pessoas)')
plt.ylabel('Tempo (s)')
plt.title("M x Tempo, dado N fixo (30)")
plt.legend()
plt.tight_layout()
plt.show()
```



In [20]:

```
plt.plot(dfN['M'], dfN['MMS_Heuristico'], label="Heuristico")
plt.plot(dfN['M'], dfN['MMS_Local'], label="Local")
plt.xlabel('M (qtde de pessoas)')
plt.ylabel('MMS')
plt.title("M x MMS, dado N fixo (30)")
plt.legend()
plt.tight_layout()
plt.show()
```

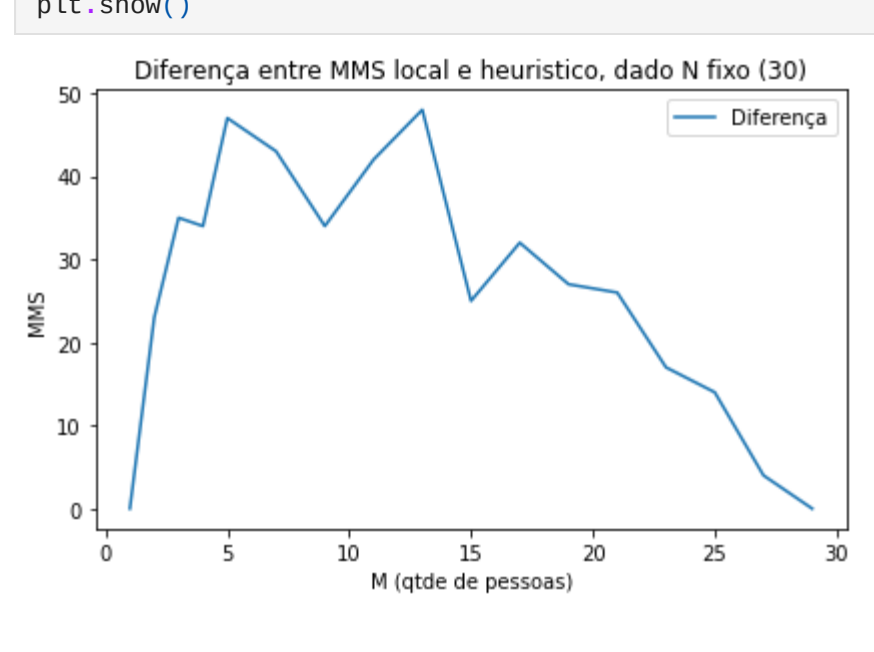


Nesses testes com N fixo, o mesmo padrão percebido anteriormente se repete. Temos que na maioria dos casos a variação de MMS entre as soluções local e heurística são relevantes.

Vale ressaltar que a busca local encontra uma distribuição de "máximo local", por causa da distribuição aleatória dos itens para as pessoas. Novamente, dada a estratégia adotada, é provável encontrarmos uma resposta ótima para uma distribuição. No entanto ainda é possível que o ponto ótimo não seja encontrado. No caso médio, no entanto, a busca local acaba sendo a melhor solução, quando consideramos tanto o seu tempo (se comparada a global) e a qualidade do resultado (se comparada a heurística).

In [21]:

```
plt.plot(dfN['M'], dfN['MMS_Local'] - dfN['MMS_Heuristico'], label="Diferença")
plt.xlabel('M (qtde de pessoas)')
plt.ylabel('MMS')
plt.title("Diferença entre MMS local e heuristico, dado N fixo (30)")
plt.legend()
plt.tight_layout()
plt.show()
```



Testes executados na solução heurística e local, com itens com valores semelhantes:

Por causa do resultado anterior, temos uma nova análise possível.

A busca heurística não é ruim, como aparenta ser dados os testes anteriores. Para casos em que temos itens com valores muito parecidos, o resultado da heurística é equiparável tanto ao da local quanto da global.

Nos testes seguintes, foram variados tanto N e M, mas mantendo os itens com valores próximos.

Nos gráficos, um dos eixos foi fixado como o número de itens por simplicidade, visto que o intuito dessa análise é a qualidade das diferentes soluções.

In [22]:

```
insValor = getInputFiles("in*_valor.txt")

resHeuristicoValor = geraDF(execs[0], insValor)
resLocalValor = geraDF(execs[1], insValor)
resGlobalValor = geraDF(execs[2], insValor)

dfValor = pd.DataFrame()
dfValor['N'] = resLocalValor['N']
dfValor['M'] = resLocalValor['M']

dfValor['MMS_Heuristico'] = resHeuristicoValor['MMS']
dfValor['T_Heuristico'] = resHeuristicoValor['T']

dfValor['MMS_Local'] = resLocalValor['MMS']
dfValor['T_Local'] = resLocalValor['T']

dfValor['MMS_Global'] = resGlobalValor['MMS']
dfValor['T_Global'] = resGlobalValor['T']
```

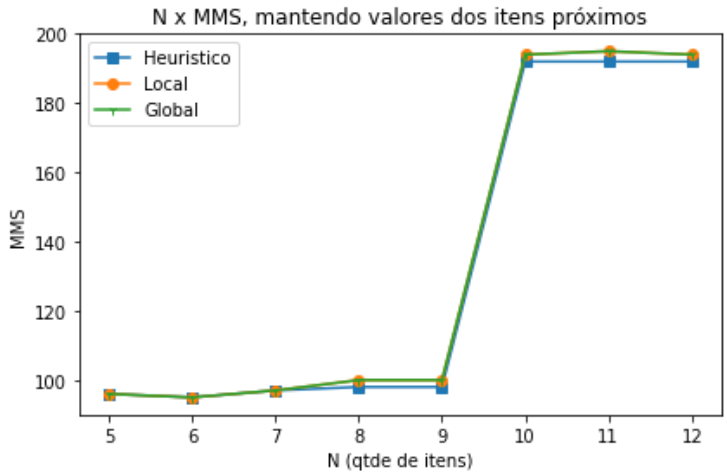
In [23]:

```
dfValor = dfValor.sort_values('N')
dfValor.head(5)
```

Out[23]:

	N	M	MMS_Heuristico	T_Heuristico	MMS_Local	T_Local	MMS_Global	T_Global
2	5	5	96	0.010800	96	0.027947	96	0.006149
0	6	5	95	0.013578	95	0.030911	95	0.009745
6	7	5	97	0.006095	97	0.043397	97	0.026899
7	8	5	98	0.006008	100	0.043790	100	0.100284
4	9	5	98	0.007572	100	0.051532	100	0.378643

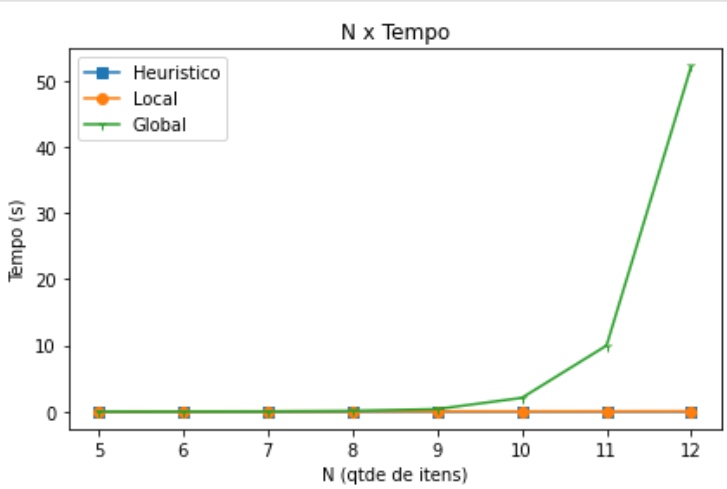
In [24]:
```
plt.plot(dfValor['N'], dfValor['MMS_Heuristico'], label="Heuristico", marker="s")
plt.plot(dfValor['N'], dfValor['MMS_Local'], label="Local", marker="o")
plt.plot(dfValor['N'], dfValor['MMS_Global'], label="Global", marker="1")
plt.xlabel('N (qtde de itens)');
plt.ylabel('MMS')
plt.title("N x MMS, mantendo valores dos itens próximos")
plt.legend()
plt.tight_layout()
plt.show()
```

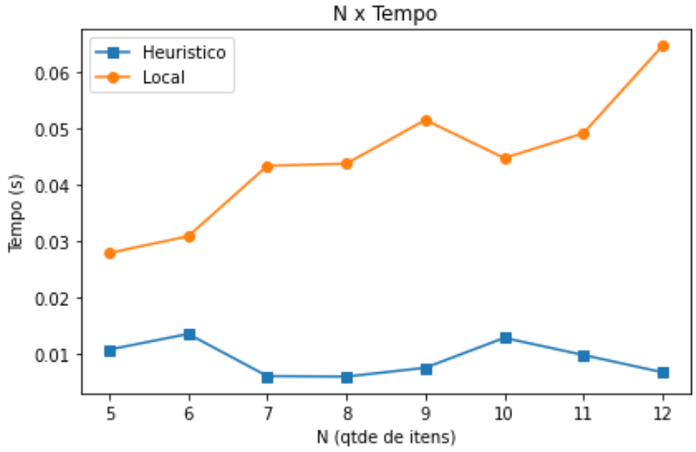


Nesse caso, temos um pulo no valor do MMS simplesmente porque cada pessoa passou a receber pelo menos dois 2 itens. Mas o que fica claro é o valor bem próximo, se não igual, entre o MMS das três soluções.

Como esperado, o tempo da heurística e da local continuam inferiores ao da global (1° gráfico abaixo), e o tempo da local é marginalmente superior ao da heurística (2° gráfico abaixo).

In [25]:
```
plt.plot(dfValor['N'], dfValor['T_Heuristico'], label="Heuristico", marker="s")
plt.plot(dfValor['N'], dfValor['T_Local'], label="Local", marker="o")
plt.plot(dfValor['N'], dfValor['T_Global'], label="Global", marker="1")
plt.xlabel('N (qtde de itens)')
plt.ylabel('Tempo (s)')
plt.title("N x Tempo")
plt.legend()
plt.tight_layout()
plt.show()

plt.plot(dfValor['N'], dfValor['T_Heuristico'], label="Heuristico", marker="s")
plt.plot(dfValor['N'], dfValor['T_Local'], label="Local", marker="o")
plt.xlabel('N (qtde de itens)')
plt.ylabel('Tempo (s)')
plt.title("N x Tempo")
plt.legend()
plt.tight_layout()
plt.show()
```





Conclusão

Tendo os testes e resultados acima como base, a conclusão que pode ser tirada é que nenhuma solução é melhor que a outra. Tudo depende do cenário (no caso, as entradas, uma vez que CPU e GPU não são relevantes nesses algortimos). Para os cenários mais diversos, sejam eles cenários com entradas grandes, que dificilmente seriam executados na global, ou cenários com objetos de valores muito diferentes, que remetem a "fraqueza" da heurística, temos a busca local como a campeã entre as soluções apresentadas.

In [ ]: