Problema do Alinhamento de Sequencias de DNA

Uma abordagem sobre o desempenho de diferentes algoritmos

Aluno: Marcelo Cesário Miguel

imports

In [1]:
```
import subprocess
import time
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import random
```

In [2]:
```
def create_inputs(n_max,dimension,step):
    list_inputs = []
    for n in np.arange(4,n_max+1,step):
        if dimension == '2d':
            m = n
            seq=str(n)+' '+str(m)+" "+''.join(random.choices(['A','T','C'
            ''.join(random.choices(['A','T','C','G','-'],k=m))
            list_inputs.append(seq)
        else:
            for m in np.arange(4,n_max+1,step):
                # m = random.randint(n,n*5)
                # m = n # comentar caso tenha m diferente
                seq=str(n)+' '+str(m)+" "+''.join(random.choices(['A','T'
                ''.join(random.choices(['A','T','C','G','-'],k=m))
                list_inputs.append(seq)
    return list_inputs


def get_list_program(list_alg,list_inputs):
    list_execute=[]
    for i in list_alg:
        for input_ in list_inputs:
            n = input_.split()[0]
            m = input_.split()[1]
            start = time.perf_counter()
            proc = subprocess.run([i], input=input_, text=True, capture_o
            end = time.perf_counter()
            algoritmo = list_alg[i]
            dic = {
                'Tempo':float(end-start),
                'Saida':int(proc.stdout.split()[-1]),
                'n': int(n),
                'm': int(m),
                'Algoritmo': algoritmo
            }
            list_execute.append(dic)
    return list_execute
```

Análise com Smith Waterman e Random Local Search

- Smith Waterman
    - O Algoritmo de Smith-Waterman é um algoritmo de programação dinâmica que compara segmentos de todos os comprimentos possíveis e otimiza a medida de similaridade.
- Random Local Search
    - Busca local com métricas de aleatoriedade, no qual é criado uma subsequência sb de tamanho k para uma entrada e, feito isso, é feito várias subsequências sa da outra entrada com esse mesmo tamanho k, por fim é comparada cada uma das sa com a sb e retornado o par com maior similaridade.

In [51]:
```
list_alg = {'./main' : 'Smith Waterman','./random_local_search/main' : 'R
```

In [52]:
```
list_inputs = create_inputs(2000,'2d',10)
list_execute = get_list_program(list_alg,list_inputs)
```

In [53]:
```
df = pd.DataFrame(list_execute)
df= df.apply(pd.to_numeric,errors="ignore")
```

In [54]:
```
df
```

Out[54]:

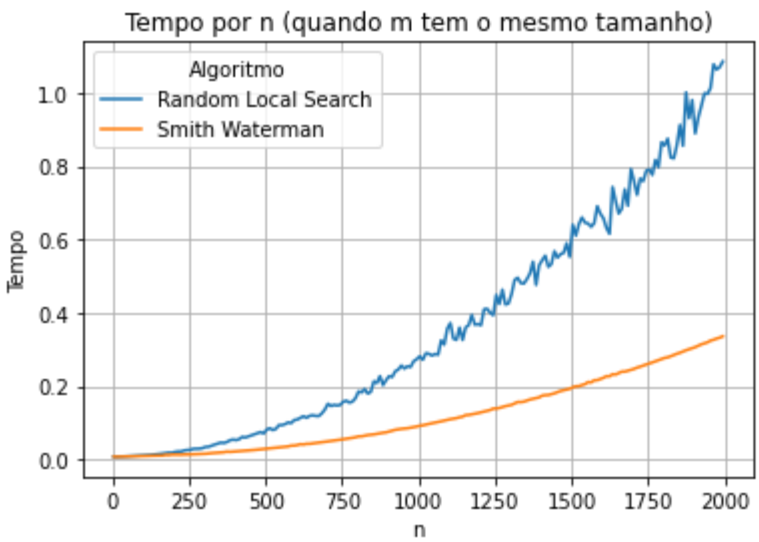
	Tempo	Saida	n	m	Algoritmo
0	0.008341	3	4	4	Smith Waterman
1	0.007763	6	14	14	Smith Waterman
2	0.008109	18	24	24	Smith Waterman
3	0.008527	16	34	34	Smith Waterman
4	0.008769	27	44	44	Smith Waterman
...	...	...	...	...	...
395	1.012811	9	1954	1954	Random Local Search
396	1.077691	-10	1964	1964	Random Local Search
397	1.063537	-7	1974	1974	Random Local Search
398	1.069931	12	1984	1984	Random Local Search
399	1.085322	2	1994	1994	Random Local Search

400 rows × 5 columns

Análise 2d

Análise bidimensional para entradas em que n e m tem o mesmo tamanho

In [55]:
```
df.pivot(index='n',columns='Algoritmo',values='Tempo').plot();
plt.title("Tempo por n (quando m tem o mesmo tamanho)");
plt.ylabel("Tempo");
plt.xlabel('n');
plt.grid();
```



Em uma primeira análise, percebe-se que a complexidade do Random Local Search está crescente. Isso ocorre pelo fato de que o K é aleatório, porém a repetição P é calculada a partir de todas as possibilidades de entradas com o mesmo tamanho K. Dessa forma, a complexidade acompanha o aumento de n e m.

Além disso, as curvas de tempo de cada um dos dois algoritmos podem estar com ruidos. Isso ocorre pelo fato de que o tempo está sendo calculado de uma maneira simples, apenas calculando a diferença de tempo antes e depois de rodar os algoritmos. Dessa forma, estão inclusos dentro desse cálculo outros processamentos do computador que influenciam no cálculo do tempo.

Dessa forma, caso rodado novamente as células acima que definem os valores do dataframe, o ruido dessa curva de tempo pode aumentar ou diminuir, mas a 'cara' da curva, que nesse caso seria exponencial para as duas continua a mesma

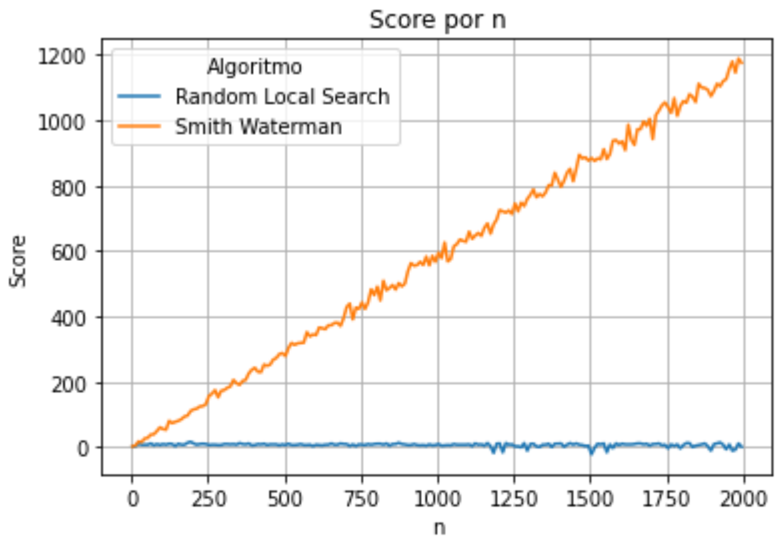
Alterando o tempo do Random Local Search

Como o algoritmo de busca local do código Random Local Search é repetido 100 vezes para cada 'n', será dividido em 100 o tempo do Random Local Search para calcularmos apenas o tempo do algoritmo de busca local

In [56]:
```
df.loc[df['Algoritmo'] == 'Random Local Search','Tempo'] /=100
```

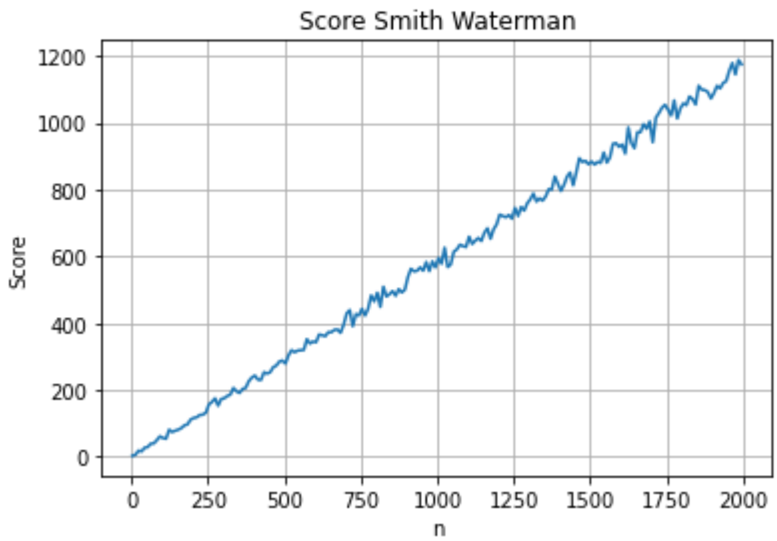
In [57]:
```
df.pivot(index='n',columns='Algoritmo',values='Tempo').plot()
plt.title("Tempo por n (quando m tem o mesmo tamanho)");
plt.ylabel("Tempo");
plt.grid();
plt.show();
```

In [58]:
```
df.pivot(index='n',columns='Algoritmo',values='Saida').plot()
plt.title("Score por n");
plt.grid()
plt.ylabel("Score");
```



In [59]:

```
df.pivot(index='n',columns='Algoritmo',values='Saida')['Smith Waterman'].
plt.title("Score Smith Waterman")
plt.grid()
plt.ylabel("Score");
plt.show()
df.pivot(index='n',columns='Algoritmo',values='Saida')['Random Local Sear
plt.title("Score Random Local Search")
plt.grid()
plt.ylabel("Score");
plt.show()
```



Análisando os Scores, mais especificamente do Random Local Search, percebe-se que seu Score não aumenta para sequências maiores, isso pode ser um indicativo de que repetir o código por uma constante (no caso 100 vezes) pode não ser o suficiente para garantir um bom desempenho do modelo. Dessa forma, essa constante pode ser baseada no tamanho da entrada de n ou m, por exemplo, para que a aleatoriedade acompanhe o tamanho de uma sequência.

Da mesma forma, pode não fazer sentido repetir o código 100 vezes para uma sequência muito pequena, pelo fato de que esse excesso de tentativas pode aproximar muito a busca local da busca exausta, uma vez que a busca local acabe testando todas as possibilidades possíveis por conta da entrada ser pequena em relação a entrada

Análise em 3d

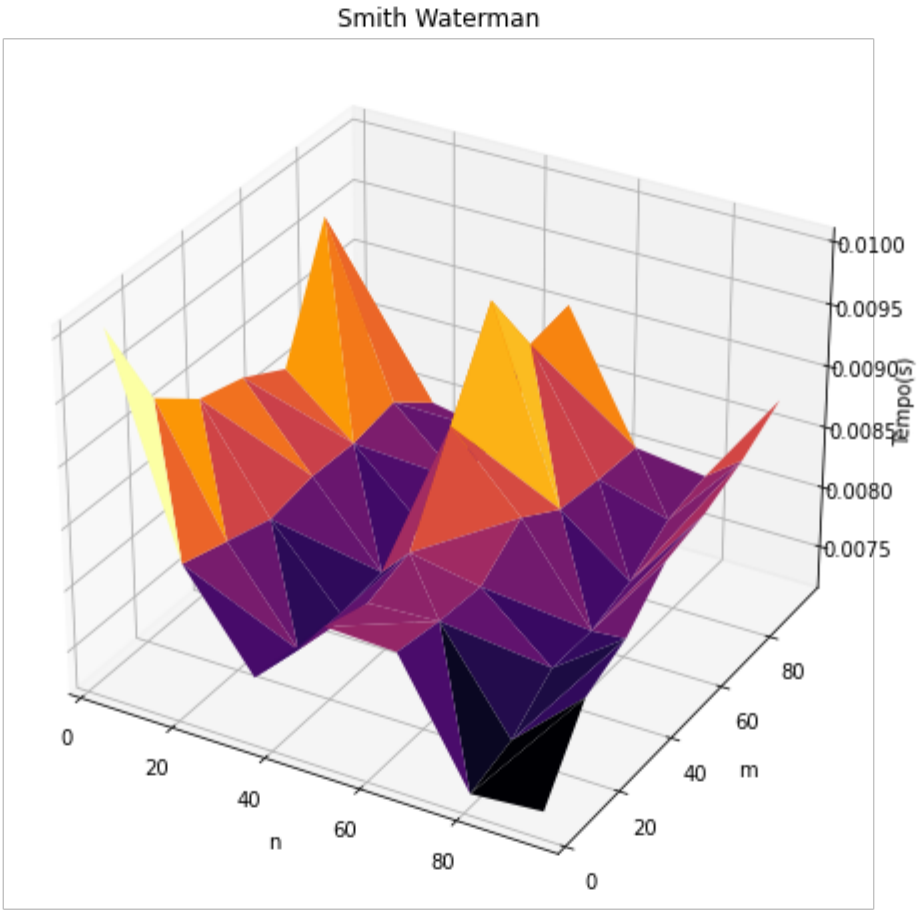
In [25]:
```
list_inputs = create_inputs(300,'3d',15)
list_execute = get_list_program(list_alg,list_inputs)
```

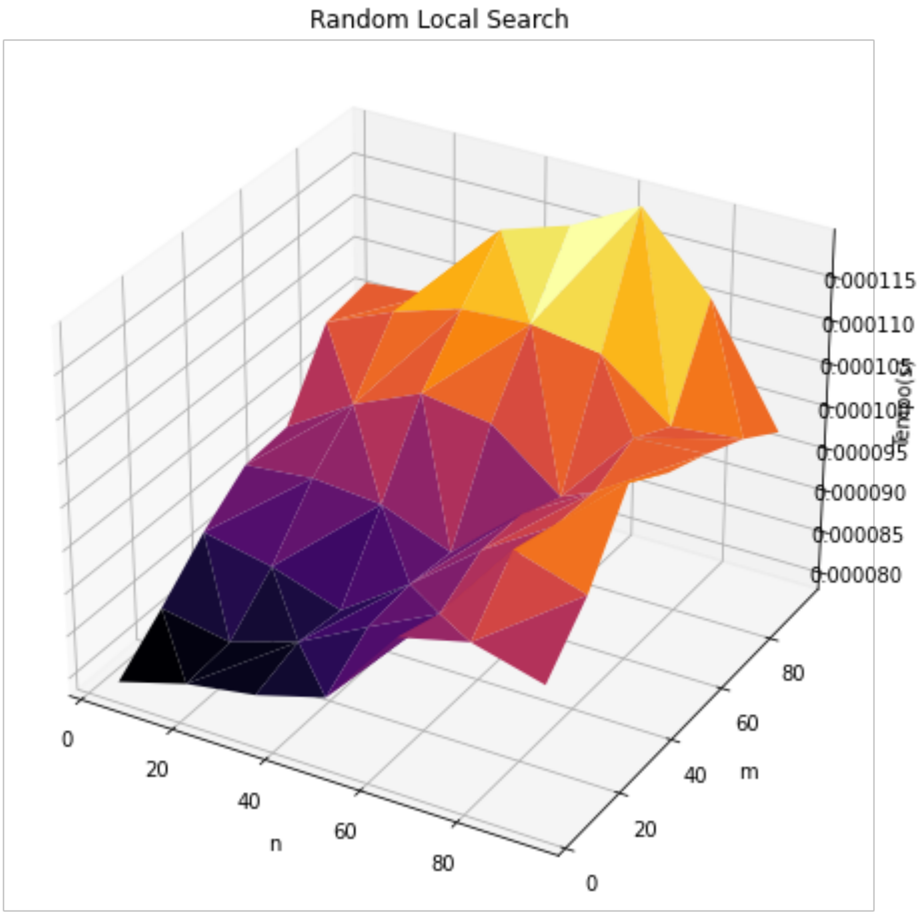
In [26]:
```
df = pd.DataFrame(list_execute)
df= df.apply(pd.to_numeric,errors="ignore")
```

In [27]:
```
df.loc[df['Algoritmo'] == 'Random Local Search','Tempo'] /=100
```

In [45]:
```
df_smith = df.loc[df['Algoritmo'] == 'Smith Waterman']
fig = plt.figure(figsize=(15,8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.plot_trisurf(df_smith['n'], df_smith['m'], df_smith['Tempo'],cmap
plt.title("Smith Waterman")
plt.xlabel('n')
plt.ylabel('m')
ax.set_zlabel('Tempo(s)')
plt.show()

df_random = df.loc[df['Algoritmo'] == 'Random Local Search']
fig = plt.figure(figsize=(15,8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.plot_trisurf(df_random['n'], df_random['m'], df_random['Tempo'],
plt.title("Random Local Search")
plt.xlabel('n')
plt.ylabel('m')
ax.set_zlabel('Tempo(s)');
```





Como o calculo do tempo no python pode gerar alguns ruídos, análisar em 3d pode nem sempre ser a melhor opção para verificar a curva do tempo em função do aumento das entradas n e m

Adicionando Exaustive Search

- Exaustive Search
  - Algoritmo que testa todas as substrings de uma entrada com todas as substrings da outra entrada, garantindo que sempre será retornado o melhor par de substrings possível

In [29]:
```
list_alg_exaust = {'./main' : 'Smith Waterman',
                   './random_local_search/main' : 'Random Local Search',
                   './exaustive_search/main':'Exaustive Search'}
```

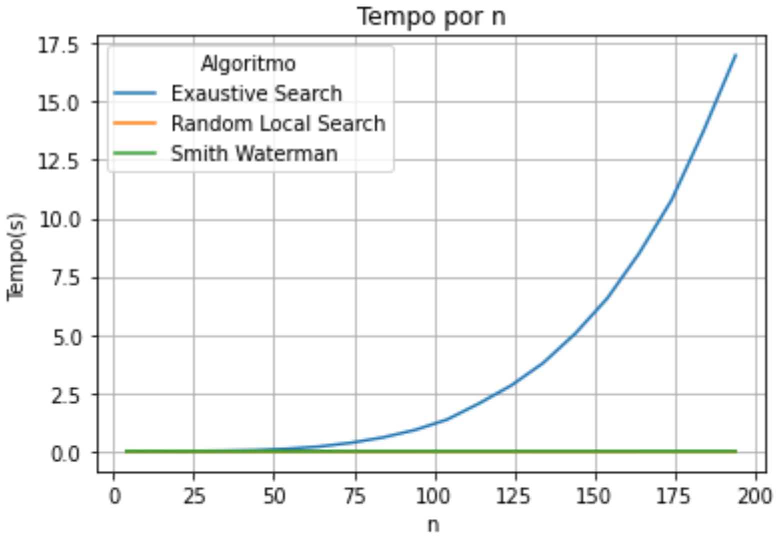
In [31]:
```
print("Esse código pode demorar alguns minutos")
list_inputs = create_inputs(200,'2d',10)
list_execute = get_list_program(list_alg_exaust,list_inputs)
```

```
Esse código pode demorar alguns minutos
```

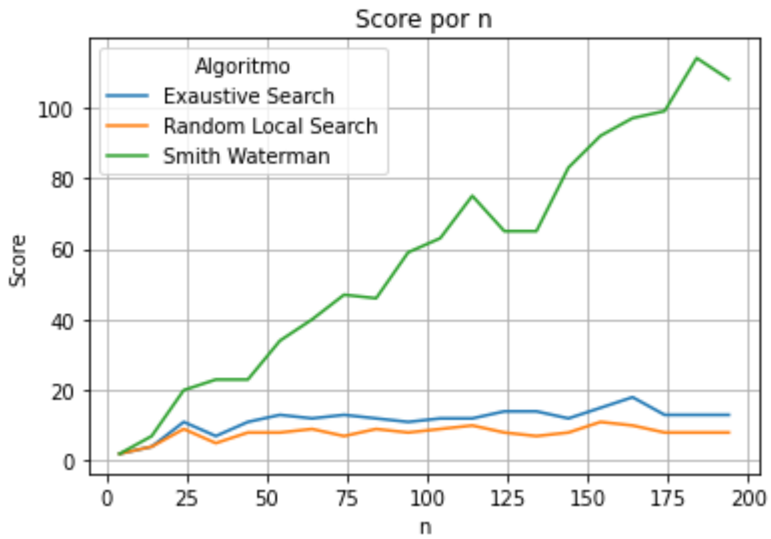
In [32]:
```
df = pd.DataFrame(list_execute)
df= df.apply(pd.to_numeric,errors="ignore")
```

In [33]:
```
df.loc[df['Algoritmo'] == 'Random Local Search','Tempo'] /=100
```

In [34]:
```
df.pivot(index='n',columns='Algoritmo',values='Tempo').plot()
plt.title("Tempo por n");
plt.ylabel("Tempo(s)");
plt.grid();
```



In [35]:
```
df.pivot(index='n',columns='Algoritmo',values='Saida').plot()
plt.title("Score por n");
plt.grid()
plt.ylabel("Score");
```



Como o algoritmo Smith Waterman pode realizar manipulaçoes nas entradas, seu score não pode ser comparado com os demais. Dessa forma, analisando o Exaustive Search com o Random Local Search, percebe-se que o Exaustive Search sempre deve ter um Score maior ou igual ao Random Local Search, já que ambos não realizam manipulações nas sequências e o Exaustive Search testa todas as substrings possíveis das duas entradas.

In [47]:
```
print("Esse código pode demorar alguns minutos")
list_inputs = create_inputs(200,'3d',30)
list_execute = get_list_program(list_alg_exaust,list_inputs)
```

```
Esse código pode demorar alguns minutos
```

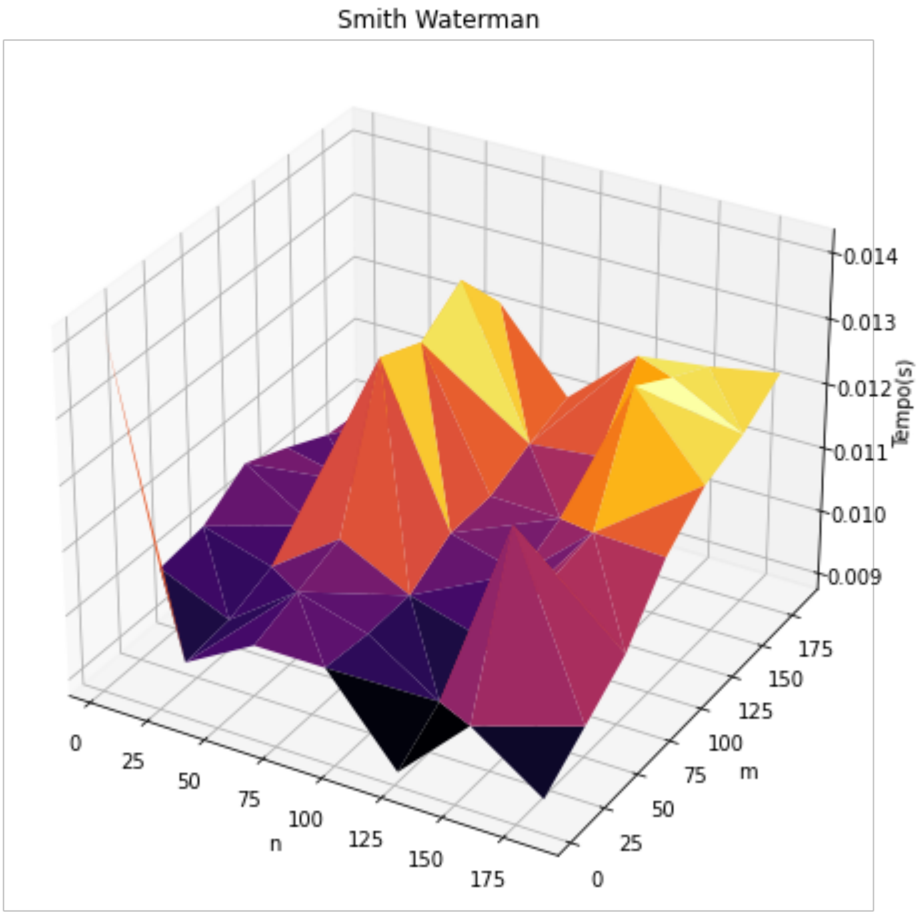
In [48]:
```
df = pd.DataFrame(list_execute)
df= df.apply(pd.to_numeric,errors="ignore")
```

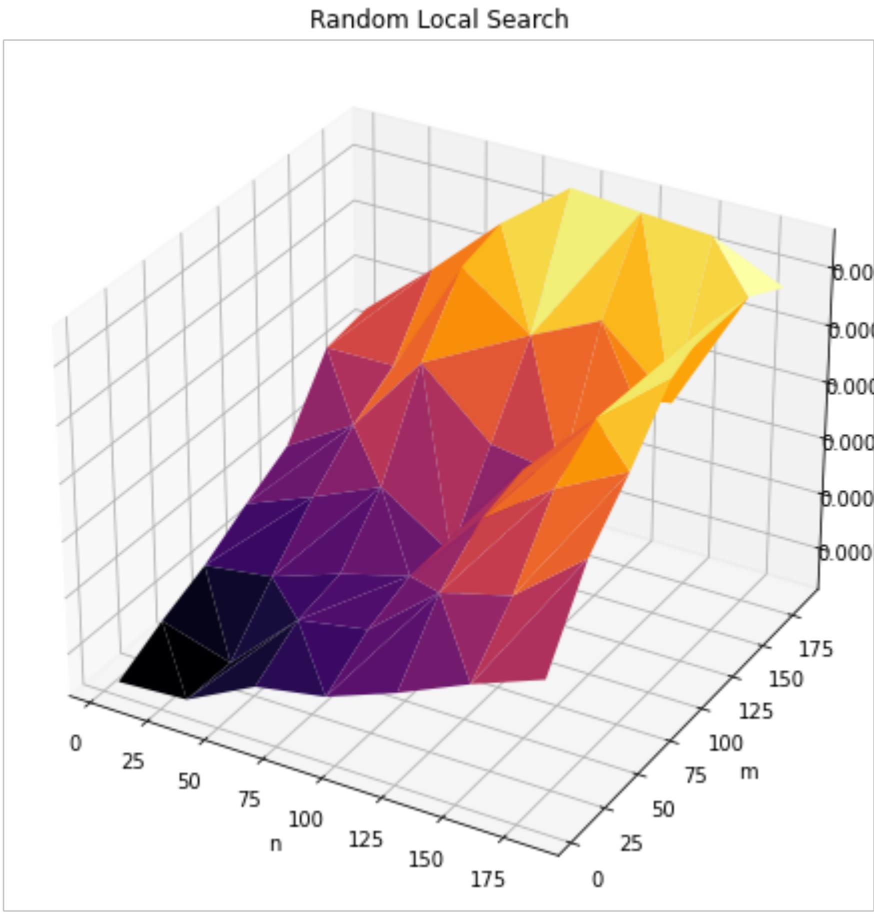
In [49]:
```
df.loc[df['Algoritmo'] == 'Random Local Search','Tempo'] /=100
```

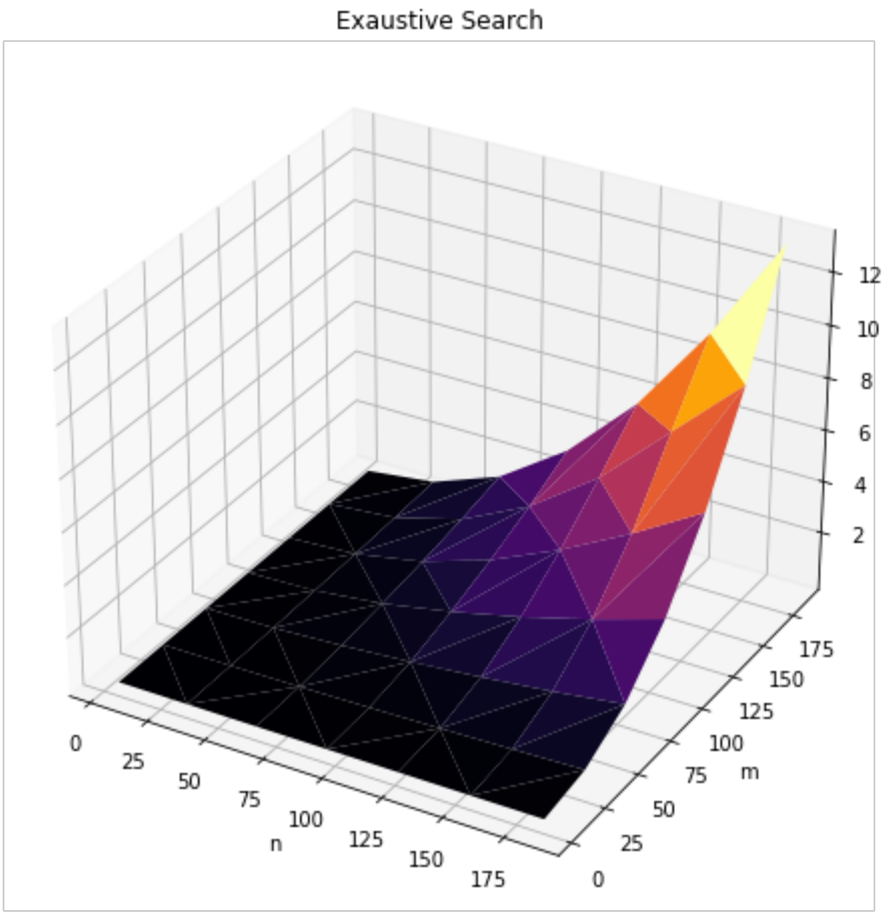
In [50]:
```
df_smith = df.loc[df['Algoritmo'] == 'Smith Waterman']
fig = plt.figure(figsize=(15,8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.plot_trisurf(df_smith['n'], df_smith['m'], df_smith['Tempo'],cmap
plt.title("Smith Waterman")
plt.xlabel('n')
plt.ylabel('m')
ax.set_zlabel('Tempo(s)')
plt.show()

df_random = df.loc[df['Algoritmo'] == 'Random Local Search']
fig = plt.figure(figsize=(15,8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.plot_trisurf(df_random['n'], df_random['m'], df_random['Tempo'],c
plt.title("Random Local Search")
plt.xlabel('n')
plt.ylabel('m')
ax.set_zlabel('Tempo(s)');

df_exaustive = df.loc[df['Algoritmo'] == 'Exaustive Search']
fig = plt.figure(figsize=(15,8))
ax = fig.add_subplot(111, projection='3d')
sc = ax.plot_trisurf(df_exaustive['n'], df_exaustive['m'], df_exaustive['
plt.title("Exaustive Search")
plt.xlabel('n')
plt.ylabel('m')
ax.set_zlabel('Tempo (s)');
```




Random Local Search
0.00020
0.00018
0.00016
0.00014
0.00012
0.00010
Tempo (s)
0
25
50
75
100
125
150
175
n
0
25
50
75
100
125
150
175
m


Exaustive Search
12
10
8
6
4
2
Tempo (s)
0
25
50
75
100
125
150
175
n
0
25
50
75
100
125
150
175
m

Profiling

Utilizando a mesma entrada para os diferentes algoritmos

- Smith Waterman

```
   519      for (int i = 1; i < n + 1; i++)
     .      {
 53,457         for (int j = 1; j < m + 1; j++)
     .          {
392,533             int diagonal = H[i - 1][j - 1].valor + w_score(a[i - 1], b[j - 1]);
159,135  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::vector<Alinhamento, std::allocator<Alinhamento> >, std::allocator<std::vector<Alinha
mento, std::allocator<Alinhamento> > > >::operator[](unsigned long) (10,609x)
127,308  => /usr/include/c++/9/bits/stl_vector.h:std::vector<Alinhamento, std::allocator<Alinhamento> >::operator[](unsigned long) (10,609x)
233,398  => /usr/include/c++/9/bits/stl_vector.h:std::vector<char, std::allocator<char> >::operator[](unsigned long) (21,218x)
173,420  => main.cpp:w_score(char, char) (10,609x)
159,135             int delecao = H[i - 1][j].valor - 1;
159,135  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::vector<Alinhamento, std::allocator<Alinhamento> >, std::allocator<std::vector<Alinha
mento, std::allocator<Alinhamento> > > >::operator[](unsigned long) (10,609x)
127,308  => /usr/include/c++/9/bits/stl_vector.h:std::vector<Alinhamento, std::allocator<Alinhamento> >::operator[](unsigned long) (10,609x)
159,135             int insercao = H[i][j - 1].valor - 1;
159,135  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::vector<Alinhamento, std::allocator<Alinhamento> >, std::allocator<std::vector<Alinha
mento, std::allocator<Alinhamento> > > >::operator[](unsigned long) (10,609x)
127,308  => /usr/include/c++/9/bits/stl_vector.h:std::vector<Alinhamento, std::allocator<Alinhamento> >::operator[](unsigned long) (10,609x)
339,488             H[i][j].valor = max({0, diagonal, delecao, insercao});
2,131,550  => /usr/include/c++/9/bits/stl_algo.h:int std::max<int>(std::initializer_list<int>) (10,609x)
159,135  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::vector<Alinhamento, std::allocator<Alinhamento> >, std::allocator<std::vector<Alinha
mento, std::allocator<Alinhamento> > > >::operator[](unsigned long) (10,609x)
127,308  => /usr/include/c++/9/bits/stl_vector.h:std::vector<Alinhamento, std::allocator<Alinhamento> >::operator[](unsigned long) (10,609x)
169,744             if (maior < H[i][j].valor)
159,135  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::vector<Alinhamento, std::allocator<Alinhamento> >, std::allocator<std::vector<Alinha
mento, std::allocator<Alinhamento> > > >::operator[](unsigned long) (10,609x)
127,308  => /usr/include/c++/9/bits/stl_vector.h:std::vector<Alinhamento, std::allocator<Alinhamento> >::operator[](unsigned long) (10,609x)
     .              {
   559                  maior = H[i][j].valor;
```

Pelo valgrind do Smith Waterman, a operação que demanda maior tempo é a de calcular o score dentro da matriz, que é calculado 10,609 vezes

- Random Local Search

```
  25,640         for (int i = 0; i<p;i++){
  30,288             SubstringValue sa_temp;
 434,652  => main.cpp:SubstringValue::~SubstringValue() (5,048x)
  95,912  => main.cpp:SubstringValue::SubstringValue() (5,048x)
  80,768             sa_temp.substring = a.substr(i,k);
  40,384  => ???:0x000000000010a300 (5,048x)
 683,562  => ???:0x000000000010a3a0 (5,048x)
 177,419  => ???:0x000000000010a3e0 (5,048x)
       .             // cout<<sa_temp.substring<<" ";
       .
       .             // int index_a = distribution_a(generator);
       .             // sa_temp.substring = create_substring(a,index_a,k);
  25,240             all_seqs.push_back(sa_temp.substring);
3,452,189  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<ch
llocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::push_back(std::__cxx11::basic_string<char,
its<char>, std::allocator<char> > const&) (5,048x)
 111,056             sa_temp.score=calculate_score(sa_temp.substring,sb);
12,891,279  => main.cpp:calculate_score(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::__cxx11::ba
ar, std::char_traits<char>, std::allocator<char> >) (5,048x)
1,878,619  => ???:0x000000000010a2b0 (10,096x)
 765,090  => ???:0x000000000010a300 (10,096x)
```

Pelo valgrind do Random Local Search, a operação que demanda maior tempo é a de calcular o score para todas as substrings aleatórias, sendo calculado 5,048 vezes (isso porque o código repete a operação de busca local 100 vezes)

- Exaustive Search

```
          .
     52,534      for (int i = 0; i < size_subs_a; i++)
          .      {
551,922,204          for (int j = 0; j < size_subs_b; j++)
          .          {
993,384,324              int size_a = substrings_a[i].size();
1,324,512,432  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, st
d::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::operator[](unsigned long) (110,376,036x)
551,880,180  => ???:0x000000000010a270 (110,376,036x)
993,384,324              int size_b = substrings_b[j].size();
1,324,512,432  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, st
d::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::operator[](unsigned long) (110,376,036x)
551,880,180  => ???:0x000000000010a270 (110,376,036x)
331,128,108              if (size_a == size_b)
          .              {
 48,460,676                  score = calculate_score(substrings_a[i], substrings_b[j]);
2,796,954,976  => main.cpp:calculate_score(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::__cxx11::basic_string
<char, std::char_traits<char>, std::allocator<char> >) (1,425,314x)
 34,207,536  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std:
:allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::operator[](unsigned long) (2,850,628x)
326,294,378  => ???:0x000000000010a220 (2,850,628x)
184,535,824  => ???:0x000000000010a260 (2,850,628x)
  4,275,942                  if (score > max_score)
          .                  {
         20                      max_score = score;
        110                      best_sub_a = substrings_a[i];
        120  => /usr/include/c++/9/bits/stl_vector.h:std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std:
:allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::operator[](unsigned long) (10x)
      1,888  => ???:0x000000000010a1f0 (10x)
```

Pelo valgrind do Exaustive Search, a operação que demanda maior tempo é a de calcular o match entre as duas entradas, sendo calculado 2,850,628 vezes.

Conclusão

Apesar de cada algoritmo calcular valores diferentes de Score demorando tempos diferentes, não se pode afirmar qual algoritmo seria o melhor. Isso porque, para definir o melhor algoritmo nesse caso, é necessário definir uma métrica. Por exemplo, se a métrica for puramente tempo, o busca local é o recomendado, já se a métrica for score, o Smith Waterman é o melhor, assim como se o tempo não for um problema, o Exaustive Search pode ser o mais recomendado.

Dessa forma, a decisão de utilizar um algoritmo depende de quais métricas estão sendo levadas em consideração. Entretanto, a análise de mesmas entradas para diferentes algoritmos pode ser interessante tambem para ser avaliado se, para um tamanho de entrada X, qual o algoritmo faz mais sentido ser utilizado. Por exemplo, se a entrada for pequena, o tempo de um Exaustive Search pode ter uma diferença mínima quando comparado a outros algoritmos, porém, para entradas grandes, sua lentidão se destaca mais. Dessa forma, o tamanho das sequências tambem influenciam na decisão de um algoritmo

In [ ]: