

Random and Local Search

November 5, 2024

1 Random and Local Search

1.1 Random Search

Before we can code, some imports: We are considering randomised algorithms, and we want to plot results. To run this notebook, you will also need matplotlib, numpy, and scipy installed.

```
[1]: import matplotlib.pyplot as plt
import random
```

We will start with the One-Max problem: The aim of this optimisation problem is to find a binary vector of length n such that all elements are 1. The solution is obvious to us, but not to a search algorithm. We first need to decide on a specific length of n for our experiments.

```
[2]: n = 20
```

In order to do any kind of random search, we need to be able to randomly sample solutions from the search space.

```
[3]: def get_random_solution():
    return [random.choice([0,1]) for _ in range(n)]
```

Now we can look at an example solution from our search space:

```
[4]: get_random_solution()
```

```
[4]: [0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0]
```

Every invocation will produce a different individual

```
[5]: get_random_solution()
```

```
[5]: [0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0]
```

To solve an optimisation problem, we also need an objective function, or *fitness function*, that quantifies how good a particular solution is with respect to the optimisation problem we are trying to solve. For our example one max problem, we can estimate how close we are to solving the problem by counting the number of 1s in our solution.

```
[6]: def get_fitness(solution):
    return sum(solution)
```

```
[7]: get_fitness([0,0,0,0,0])
```

```
[7]: 0
```

```
[8]: get_fitness([1,1,1,0,0])
```

```
[8]: 3
```

We will keep track of what the search does; for this I'm simply using a plain list that contains the best fitness value observed at each iteration of the search.

```
[9]: fitness_values = []
```

In some cases, we know when we have found an optimal solution; sometimes we don't. Sometimes it may also take very long to find the optimal solution, therefore we limit the search with some *stopping criterion*. We will start by running the search for a fixed number of steps.

```
[10]: max_steps = 100
```

The following implements our first metaheuristic: *Random Search*. It's not a particularly intelligent one, but it will be the baseline we are going to compare against from now on. If the fitness landscape is bad, this algorithm may actually be better than many other metaheuristics.

```
[11]: def randomsearch():  
  
    best = None  
    best_fitness = -1  
  
    for step in range(max_steps):  
        candidate = get_random_solution()  
        fitness = get_fitness(candidate)  
        if fitness > best_fitness:  
            best = candidate  
            best_fitness = fitness  
            print(f"Iteration {step}, fitness {fitness}: {candidate}")  
        fitness_values.append(best_fitness)  
  
    print(f"Solution fitness {best_fitness}: {best}")  
    return best
```

Before invoking it, we also have to reset our list of fitness values.

```
[12]: fitness_values = []  
       randomsearch()
```

```
Iteration 0, fitness 9: [0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1,  
0, 0]
```

```
Iteration 1, fitness 14: [1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1,  
1, 1]
```

```
Iteration 46, fitness 15: [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1,
```

```
0, 1]
```

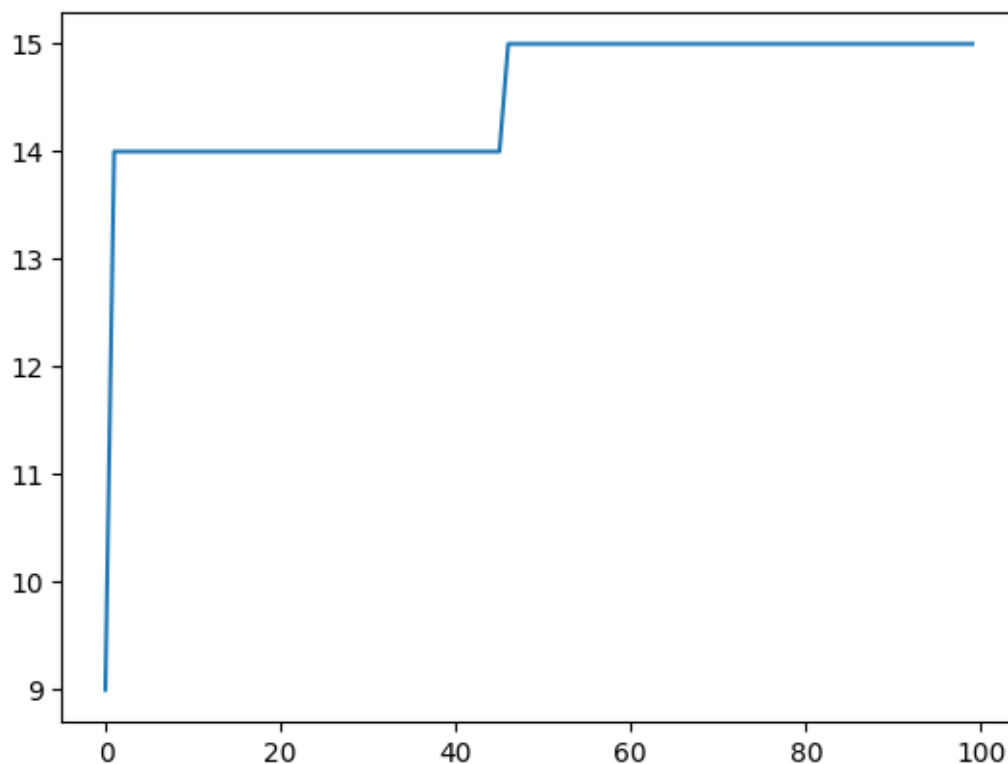
```
Solution fitness 15: [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

```
[12]: [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

We kept track of the best solution found throughout the search, and we will use this information to learn about the progress, and compare search algorithms. Therefore, we are going to visualise it.

```
[13]: import matplotlib.pyplot as plt
      plt.plot(fitness_values)
```

```
[13]: [<matplotlib.lines.Line2D at 0x10e1ab310>]
```



If you re-run the search multiple times you will get different results, but even for small values of n the random search will rarely find an optimal solution.

Random search is a *global* search algorithm – it considers the entire search space. An alternative strategy is to consider the local neighbourhood of candidate solutions, resulting in *local* search algorithms. As a naive baseline, we can change our random search to a local random search, also known as a *random walk*. A random walk consists of repeatedly picking a random neighbour for a given candidate solution, therefore we need to define a function that picks a random neighbour; that is, a solution which differs only in a single step (in our case, a bit-flip).

```
[14]: def get_neighbour(candidate):  
      pos = random.randint(0, len(candidate) - 1)  
      copy = candidate[:]  
      copy[pos] = 1 - copy[pos]  
      return copy
```

```
[15]: get_neighbour([0,0,0,0,0,1,0,0,0,0])
```

```
[15]: [0, 0, 0, 0, 0, 1, 0, 0, 1, 0]
```

```
[16]: get_neighbour([0,0,0,0,0,1,0,0,0,0])
```

```
[16]: [0, 1, 0, 0, 0, 1, 0, 0, 0, 0]
```

```
[17]: get_neighbour([0,0,0,0,0,1,0,0,0,0])
```

```
[17]: [0, 0, 0, 1, 0, 1, 0, 0, 0, 0]
```

To do a random walk, we start at a random solution, and repeatedly pick a random neighbour. Along the search, we keep track of the best solution found.

```
[18]: def randomwalk():  
      current = get_random_solution()  
      fitness = get_fitness(current)  
  
      best = current[:]  
      best_fitness = fitness  
      print(f"Starting at fitness {fitness}: {current}")  
  
      for step in range(max_steps):  
          current = get_neighbour(current)  
          fitness = get_fitness(current)  
  
          if fitness > best_fitness:  
              best = current  
              best_fitness = fitness  
              print(f"Iteration {step}, fitness {fitness}: {current}")  
              fitness_values.append(best_fitness)  
  
      print(f"Solution fitness {best_fitness}: {best}")  
      return best
```

```
[19]: fitness_values = []  
      randomwalk()
```

```
Starting at fitness 9: [0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0]
```

```
Iteration 20, fitness 10: [0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]
```

```

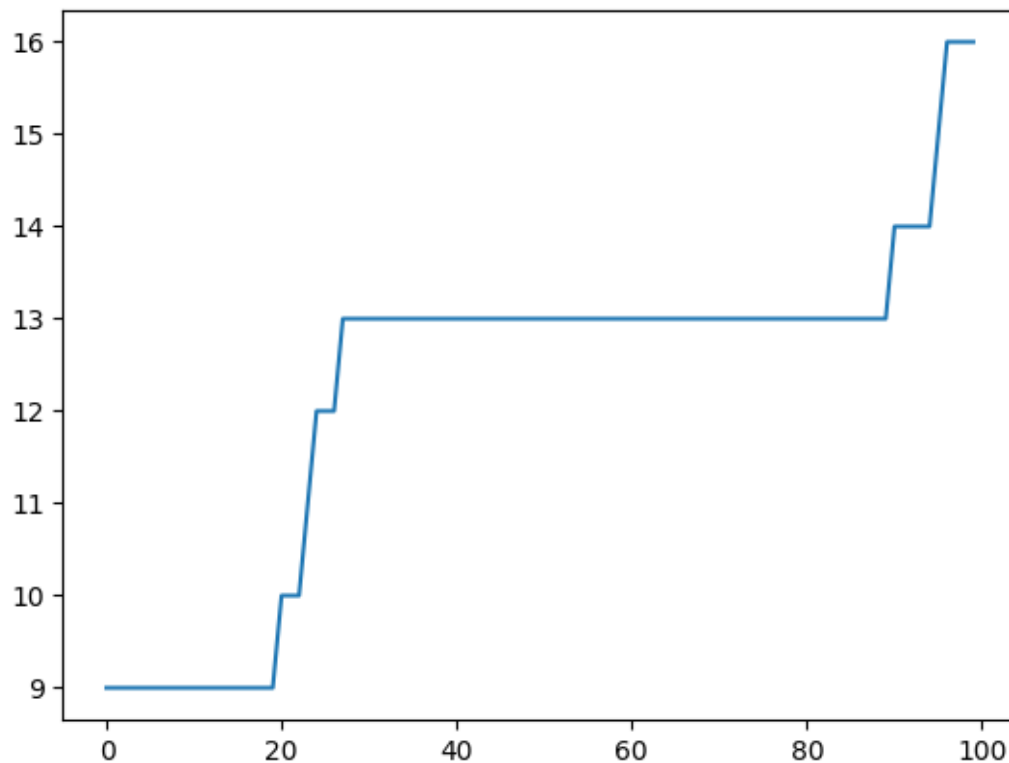
Iteration 23, fitness 11: [0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1,
1, 1]
Iteration 24, fitness 12: [0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,
1, 1]
Iteration 27, fitness 13: [0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1,
1, 1]
Iteration 90, fitness 14: [0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1,
1, 1]
Iteration 95, fitness 15: [0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,
1, 1]
Iteration 96, fitness 16: [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,
1, 1]
Solution fitness 16: [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,
1]

```

```
[19]: [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
```

```
[20]: plt.plot(fitness_values)
```

```
[20]: [<matplotlib.lines.Line2D at 0x10e2ce6b0>]
```



A random walk usually isn't well suited as an actual search algorithm, it tends to produce even worse solutions than the random search (at least on one max). However, we can now generalize the

idea of exploring the neighbourhood to construct some actual local search algorithms.

1.2 Local Search

A local search algorithm consists of (1) a definition of the neighbourhood for a candidate solution, and (2) an exploration operator that picks the next step in the neighbourhood. The neighbourhood is the set of solutions that one can reach from a candidate solution in one step – in our case, with a single bit flip:

```
[21]: def get_neighbours(candidate):
        neighbours = []
        for pos in range(len(candidate)):
            copy = candidate[:]
            copy[pos] = 1 - copy[pos]
            neighbours.append(copy)
        return neighbours
```

```
[22]: get_neighbours([0,0,0,0,0,1,0,0,0,0])
```

```
[22]: [[1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 1, 0, 0, 0, 1]]
```

1.2.1 Hillclimbing

The first local search algorithm we consider is very simple: The exploration operator always picks the best neighbour. The result is called *hillclimbing* because it climbs a hill in the fitness landscape (if we assume a maximisation problem, like in the case of one max). Starting from a random individual, we consider all neighbours, and pick the best one. Then, we repeat this process for this neighbour. The new point in the search space has to be strictly better.

```
[23]: def hillclimbing():

        current = get_random_solution()
        best = current[:]
        best_fitness = get_fitness(current)
        print(f"Starting at fitness {best_fitness}: {current}")

        for step in range(max_steps):

            best_neighbour = None
```

```

    neighbour_fitness = -1
    for neighbour in get_neighbours(current):
        fitness = get_fitness(neighbour)

        if fitness > neighbour_fitness:
            best_neighbour = neighbour
            neighbour_fitness = fitness

    if neighbour_fitness > best_fitness:
        print(f"Iteration {step}, fitness {best_fitness}: {best}")
        current = best_neighbour
        best = current[:]
        best_fitness = neighbour_fitness

    fitness_values.append(best_fitness)

print(f"Solution fitness {best_fitness}: {best}")
return best

```

When invoking the hillclimber, we need to reset the `fitness_values` list again so we can observe what is happening.

```

[24]: fitness_values = []
      hillclimbing()

```

```

Starting at fitness 10: [0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 0, fitness 10: [0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 1, fitness 11: [1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 2, fitness 12: [1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 3, fitness 13: [1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 4, fitness 14: [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 5, fitness 15: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 6, fitness 16: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0,
1, 1]
Iteration 7, fitness 17: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,
1, 1]
Iteration 8, fitness 18: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0,
1, 1]
Iteration 9, fitness 19: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,

```

```
1, 1]
```

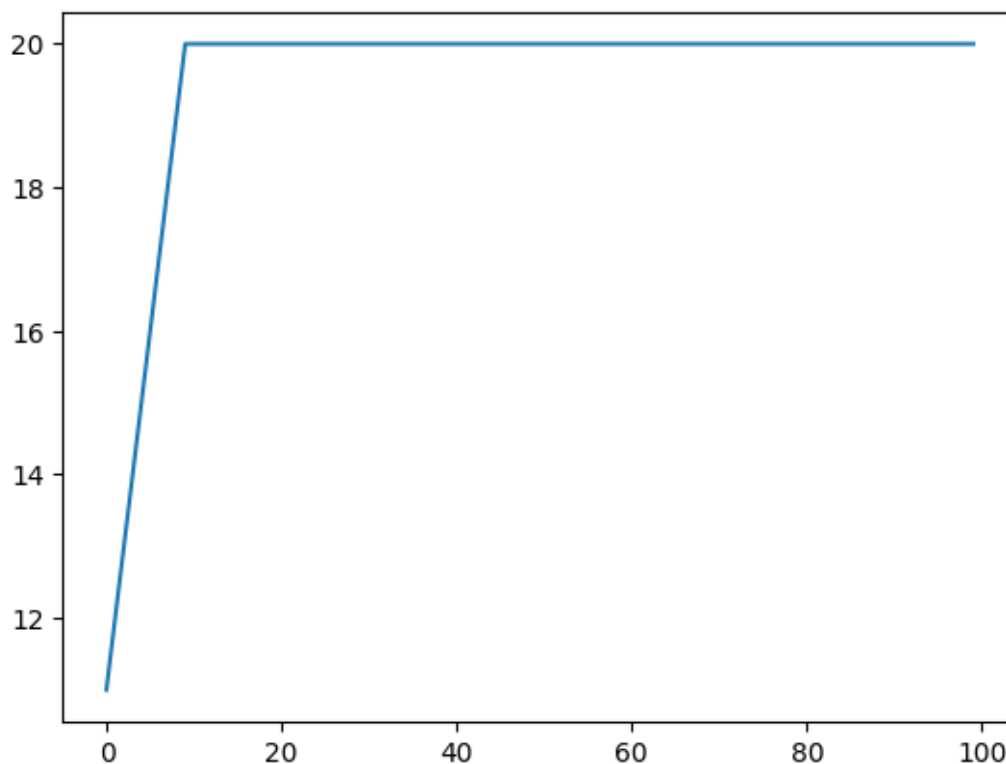
```
Solution fitness 20: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
[24]: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

The hillclimber will always find the solution very quickly, no matter how often you repeat the run, within at most `n` steps it will find an optimal solution. Plotting the `fitness_values` shows why the algorithm is called hillclimbing:

```
[25]: plt.plot(fitness_values)
```

```
[25]: [<matplotlib.lines.Line2D at 0x10e342fe0>]
```



1.2.2 The n-queens problem

The hillclimber seems perfect on the one max problem. However, not all problems are this easy. Let's consider an alternative problem: In the *n-queens* problem the aim is to place n queens on an $n \times n$ chessboard in such a way that they do not check each other.

As first step, we need to find a way to encode this problem such that we can apply the search on this encoding. A simple representation is a list of length n , where each element of the list is an integer up to n . That is, each element of the list maps to a column of the chess board, and the value of the element denotes the row in which a queen is located in this column.


```
[26]: def get_random_solution():
        return [random.randint(0, n - 1) for _ in range(n)]
```

```
[27]: get_random_solution()
```

```
[27]: [3, 12, 3, 9, 8, 16, 2, 12, 8, 12, 7, 2, 8, 9, 10, 1, 8, 5, 8, 10]
```

In order to help us interpret such a vector, we can use a simple helper function that visualises the list as a matrix.

```
[28]: def print_solution(solution):
        n = len(solution)
        for y in range(n):
            for x in range(n):
                if solution[x] == y:
                    print(" Q ", end = '')
                else:
                    print(" . ", end = '')
            print()
```

```
[29]: n = 8
        x = get_random_solution()
        print(x)
        print_solution(x)
```

```
[5, 3, 5, 3, 7, 1, 3, 1]
```

```
. . . . . . . .
. . . . . Q . Q
. . . . . . . .
. Q . Q . . Q .
. . . . . . . .
Q . Q . . . . .
. . . . . . . .
. . . . . Q . .
```

Two queens check each other if they are in the same column, same row, or same diagonal. Since the goal is that no queens check each other, we can formulate an objective function based on how many queens check each other. To be consistent with our previous example, we will make this a *maximisation* problem, such that the objective function calculates how many queens *don't* check each other. Using our vector-encoding, there can only be one queen per column by construction. We thus only need to check if queens are in the same row or diagonal. For each pair of queens that does not check each other, we increase our fitness value by 1:

```
[30]: def get_fitness(solution):
        fitness = 0
        for i in range(len(solution) - 1):
            for j in range(i + 1, len(solution)):
                if solution[i] != solution[j] \
                    and solution[i] != solution[j] + (j - i) \
```

```

        and solution[j] != solution[i] + (j - i):
            fitness += 1

    return fitness

```

Here's a simple example of all queens in a row, such that they all check each other – the fitness is thus 0.

```

[31]: x = [0,0,0,0,0,0,0,0]
      print_solution(x)
      get_fitness(x)

```

```

Q  Q  Q  Q  Q  Q  Q  Q
.  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .

```

```

[31]: 0

```

Here is a solution where no queens check each other:

```

[32]: x = [4, 0, 3, 5, 7, 1, 6, 2]
      print_solution(x)
      get_fitness(x)

```

```

.  Q  .  .  .  .  .  .
.  .  .  .  .  Q  .  .
.  .  .  .  .  .  .  Q
.  .  Q  .  .  .  .  .
Q  .  .  .  .  .  .  .
.  .  .  Q  .  .  .  .
.  .  .  .  .  .  Q  .
.  .  .  .  Q  .  .  .

```

```

[32]: 28

```

The optimal fitness value is $n*(n-1)/2 = 28$ for $n = 8$.

Before we can apply search on this problem, we need to define the neighbourhood of our problem. In the simplest case, a neighbour moves a queen up or down by one, taking into account the boundaries of the chess board.

```

[33]: def get_neighbours(solution):
      neighbours = []
      for i in range(len(solution)):
          if solution[i] > 0:

```

```

        copy = solution[:]
        copy[i] = copy[i] - 1
        neighbours.append(copy)
    if solution[i] < n - 1:
        copy = solution[:]
        copy[i] = copy[i] + 1
        neighbours.append(copy)

    return neighbours

```

```
[34]: get_neighbours([4,4,4,4,4,4,4,4])
```

```

[34]: [[3, 4, 4, 4, 4, 4, 4, 4],
       [5, 4, 4, 4, 4, 4, 4, 4],
       [4, 3, 4, 4, 4, 4, 4, 4],
       [4, 5, 4, 4, 4, 4, 4, 4],
       [4, 4, 3, 4, 4, 4, 4, 4],
       [4, 4, 5, 4, 4, 4, 4, 4],
       [4, 4, 4, 3, 4, 4, 4, 4],
       [4, 4, 4, 5, 4, 4, 4, 4],
       [4, 4, 4, 4, 3, 4, 4, 4],
       [4, 4, 4, 4, 5, 4, 4, 4],
       [4, 4, 4, 4, 4, 3, 4, 4],
       [4, 4, 4, 4, 4, 5, 4, 4],
       [4, 4, 4, 4, 4, 4, 3, 4],
       [4, 4, 4, 4, 4, 4, 5, 4],
       [4, 4, 4, 4, 4, 4, 4, 3],
       [4, 4, 4, 4, 4, 4, 4, 5]]

```

Let's first try how well random search copes with this problem. Since the problem is quite a bit more difficult than one max, let's also increase the maximum number of steps.

```

[35]: max_steps = 1000
      fitness_values = []
      randomsearch()
      plt.plot(fitness_values)

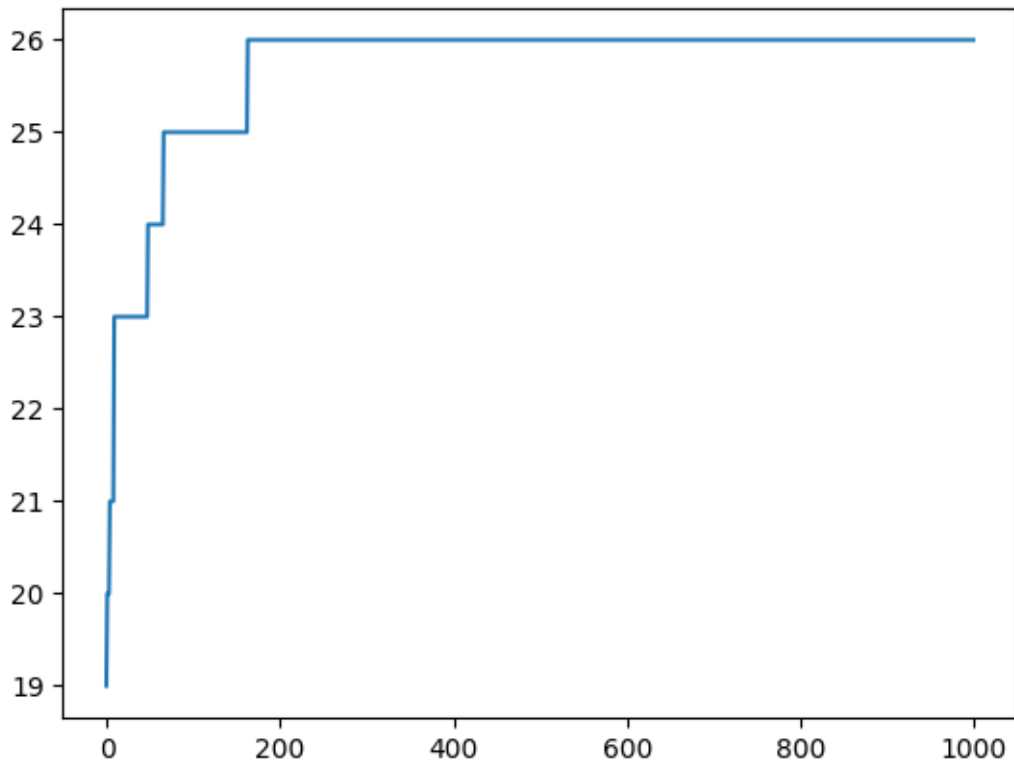
```

```

Iteration 0, fitness 19: [3, 5, 6, 5, 5, 5, 7, 4]
Iteration 1, fitness 20: [1, 4, 5, 1, 4, 0, 0, 0]
Iteration 4, fitness 21: [7, 4, 3, 5, 7, 2, 6, 7]
Iteration 9, fitness 23: [1, 3, 6, 3, 6, 0, 4, 0]
Iteration 48, fitness 24: [5, 3, 2, 0, 7, 6, 0, 6]
Iteration 66, fitness 25: [5, 0, 0, 6, 1, 7, 2, 7]
Iteration 163, fitness 26: [1, 6, 0, 3, 3, 7, 2, 4]
Solution fitness 26: [1, 6, 0, 3, 3, 7, 2, 4]

```

```
[35]: [<matplotlib.lines.Line2D at 0x10e3bb190>]
```

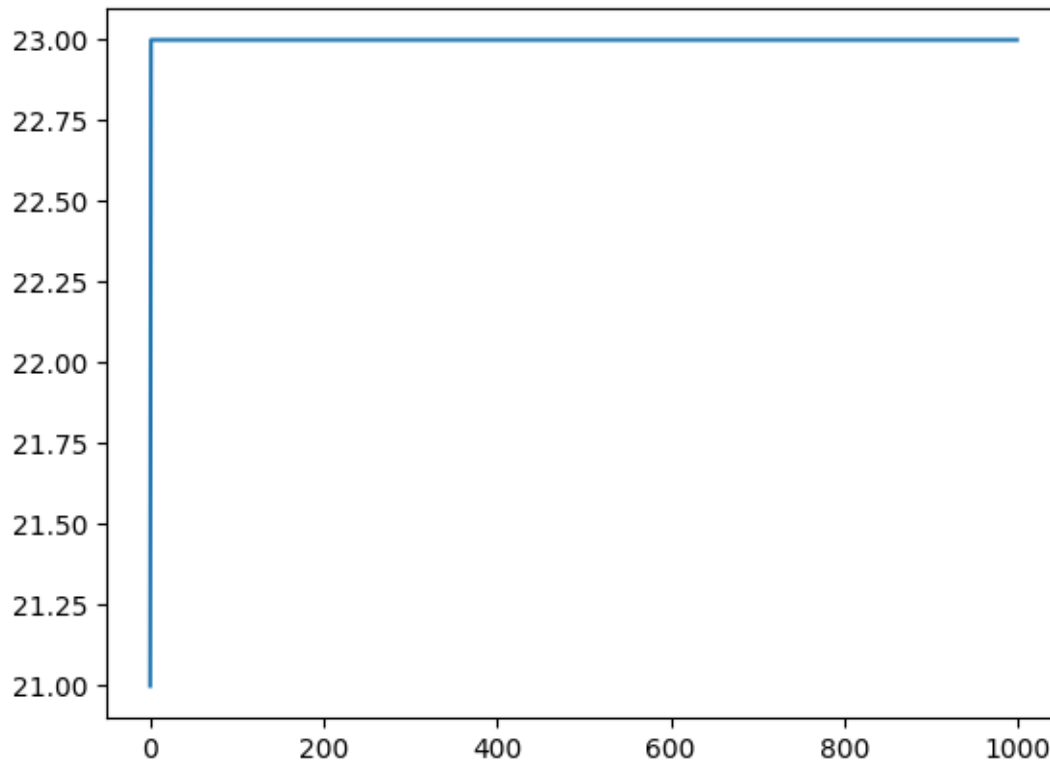


Now let's compare this to our local search algorithm, hillclimbing:

```
[36]: fitness_values = []  
      hillclimbing()  
      plt.plot(fitness_values)
```

```
Starting at fitness 17: [3, 0, 5, 0, 2, 3, 3, 0]  
Iteration 0, fitness 17: [3, 0, 5, 0, 2, 3, 3, 0]  
Iteration 1, fitness 21: [4, 0, 5, 0, 2, 3, 3, 0]  
Solution fitness 23: [4, 1, 5, 0, 2, 3, 3, 0]
```

```
[36]: [<matplotlib.lines.Line2D at 0x10e42aa70>]
```



Is hillclimbing still better than the random search? In many cases, it isn't. The reason is that, unlike for one max, there are *local optima* in the search space. Our hillclimber is forced to consider only strictly better neighbours, hence it is stuck once it reaches a local optimum. (If we would allow it to move across fitness-plateaux, i.e., neighbours of equal fitness, it might get stuck in an infinite loop.)

As an example, consider the following solution:

```
[37]: x = [7, 1, 2, 0, 5, 3, 0, 4]
      print_solution(x)
      get_fitness(x)
```

```
. . . Q . . Q .
. Q . . . . .
. . Q . . . .
. . . . . Q .
. . . . . . Q
. . . . Q . .
. . . . . . .
Q . . . . . .
```

```
[37]: 25
```

Let's consider the fitness values of all the neighbours of this solution:

```
[38]: for neighbour in get_neighbours([7, 1, 2, 0, 5, 3, 0, 4]):  
       print(f"{neighbour} -> {get_fitness(neighbour)}")
```

```
[6, 1, 2, 0, 5, 3, 0, 4] -> 24  
[7, 0, 2, 0, 5, 3, 0, 4] -> 24  
[7, 2, 2, 0, 5, 3, 0, 4] -> 23  
[7, 1, 1, 0, 5, 3, 0, 4] -> 24  
[7, 1, 3, 0, 5, 3, 0, 4] -> 24  
[7, 1, 2, 1, 5, 3, 0, 4] -> 24  
[7, 1, 2, 0, 4, 3, 0, 4] -> 21  
[7, 1, 2, 0, 6, 3, 0, 4] -> 25  
[7, 1, 2, 0, 5, 2, 0, 4] -> 21  
[7, 1, 2, 0, 5, 4, 0, 4] -> 23  
[7, 1, 2, 0, 5, 3, 1, 4] -> 24  
[7, 1, 2, 0, 5, 3, 0, 3] -> 25  
[7, 1, 2, 0, 5, 3, 0, 5] -> 24
```

As you can see, none of the neighbours of x has a better fitness value than x itself. The hillclimber has nowhere to go, and is stuck.

A simple solution to this problem is to use *random restarts*: If the hillclimber is stuck in an optimum, we restart the search from a new random point. With some luck, the next attempt will climb a different optimum.

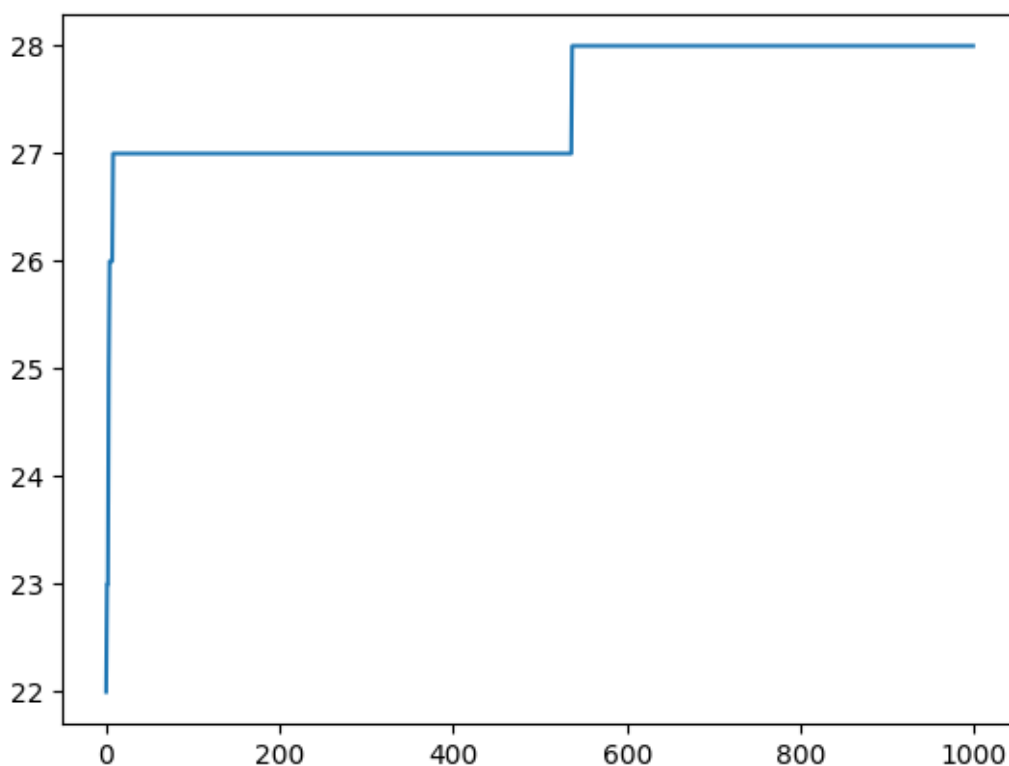
```
[39]: def hillclimbing_restart():  
  
       current = get_random_solution()  
       best = current[:]  
       best_fitness = get_fitness(current)  
  
       for step in range(max_steps):  
  
           best_neighbour = None  
           neighbour_fitness = -1  
           for neighbour in get_neighbours(current):  
               fitness = get_fitness(neighbour)  
  
               if fitness > neighbour_fitness:  
                   best_neighbour = neighbour  
                   neighbour_fitness = fitness  
  
           # Random restart if no neighbour is better  
           if neighbour_fitness <= get_fitness(current):  
               current = get_random_solution()  
               neighbour_fitness = get_fitness(current)  
           else:  
               current = best_neighbour  
  
           if neighbour_fitness > best_fitness:
```

```
        best = current[:]
        best_fitness = neighbour_fitness
        fitness_values.append(best_fitness)

    return best
```

```
[40]: fitness_values = []
      hillclimbing_restart()
      plt.plot(fitness_values)
```

```
[40]: [<matplotlib.lines.Line2D at 0x10e48afb0>]
```



Using the random restarts, the hillclimber tends to be better than before, and better than random search – even if it still doesn’t always find the best solution, depending on what value you choose for `max_steps`.

The version we considered is called *steepest ascent* hillclimbing, because it will always explore all neighbours, and then pick the best one. An alternative approach is *first ascent*, which will pick the first neighbour that has a better fitness value.

```
[41]: def hillclimbing_first_ascent():
```

```

current = get_random_solution()
fitness = get_fitness(current)
best = current[:]
best_fitness = fitness

for step in range(max_steps):

    replaced = False
    for neighbour in get_neighbours(current):
        neighbour_fitness = get_fitness(neighbour)

        if neighbour_fitness > fitness:
            current = neighbour
            fitness = neighbour_fitness
            replaced = True
            break

    # Random restart if no neighbour is better
    if not replaced:
        current = get_random_solution()
        fitness = get_fitness(current)

    if fitness > best_fitness:
        best = current[:]
        best_fitness = fitness
    fitness_values.append(best_fitness)

return best

```

For some problems, a first-ascent strategy may be less likely to run into local optima, because it is not so “eager” to climb the hill. However, this depends on the problem at hand, and luck.

```

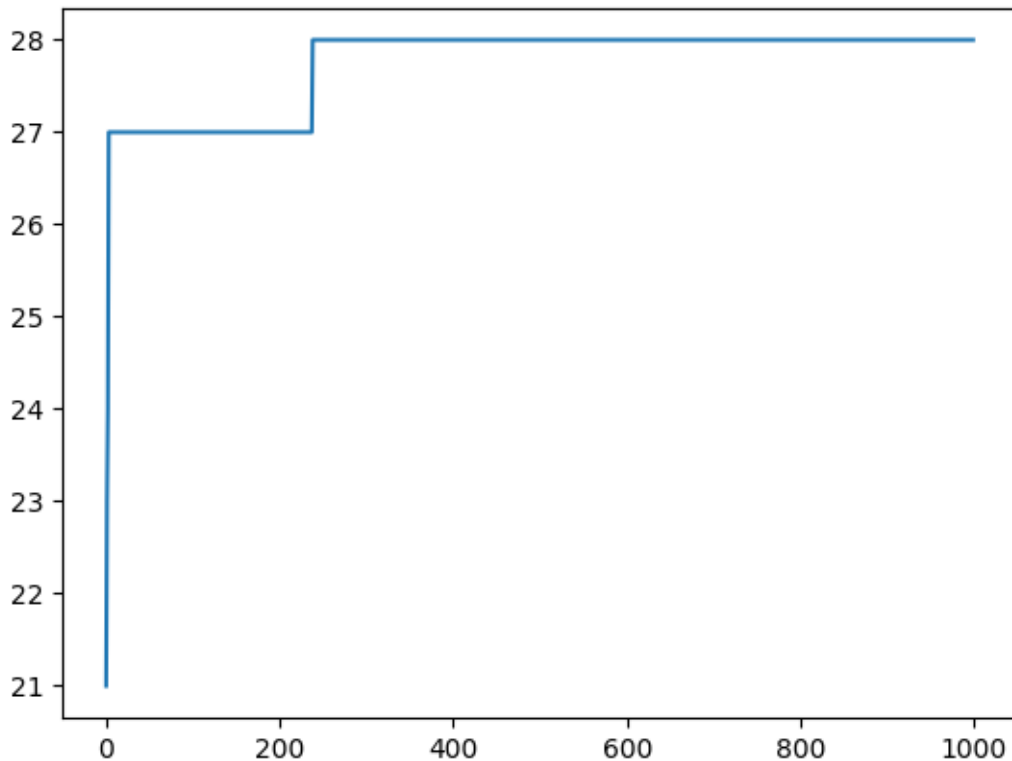
[42]: fitness_values = []
      hillclimbing_first_ascent()
      plt.plot(fitness_values)

```

```

[42]: [<matplotlib.lines.Line2D at 0x10e4e6530>]

```

For our n-queens problem, is steepest ascent or first ascent better? And is either of them better than random search? To find out, we can run all three algorithms and compare the results.

```
[43]: fitness_values = []
      randomsearch()
      f_random = fitness_values[:]

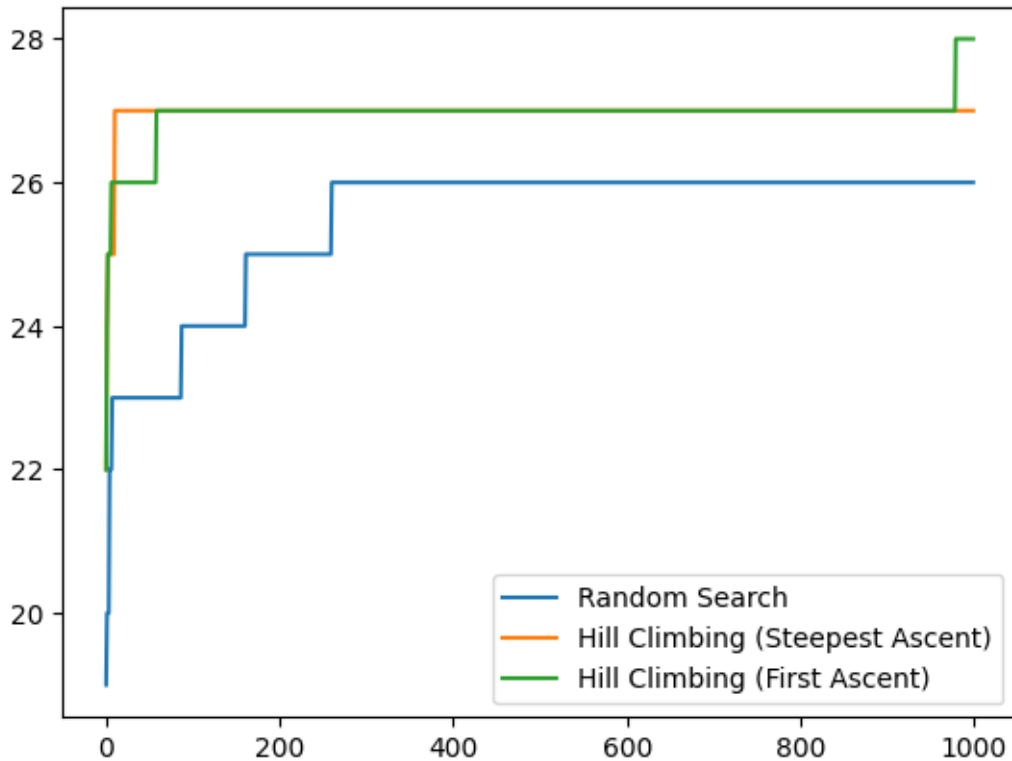
      fitness_values = []
      hillclimbing_restart()
      f_hill = fitness_values[:]

      fitness_values = []
      hillclimbing_first_ascent()
      f_first = fitness_values[:]
```

```
Iteration 0, fitness 19: [2, 4, 4, 1, 4, 1, 1, 0]
Iteration 1, fitness 20: [3, 0, 0, 7, 3, 5, 0, 6]
Iteration 4, fitness 22: [0, 5, 0, 4, 7, 5, 5, 2]
Iteration 7, fitness 23: [0, 7, 4, 0, 5, 0, 2, 5]
Iteration 87, fitness 24: [1, 2, 7, 1, 7, 2, 0, 6]
Iteration 161, fitness 25: [2, 6, 1, 3, 5, 0, 1, 4]
Iteration 260, fitness 26: [1, 7, 3, 6, 4, 2, 0, 5]
Solution fitness 26: [1, 7, 3, 6, 4, 2, 0, 5]
```

```
[44]: plt.plot(f_random, label="Random Search")
plt.plot(f_hill, label="Hill Climbing (Steepest Ascent)")
plt.plot(f_first, label="Hill Climbing (First Ascent)")
plt.legend()
```

[44]: <matplotlib.legend.Legend at 0x10e5565c0>



There are two problems with this comparison:

1. Every time you run the algorithm, you will get a different result. Sometimes one algorithm performs better than another, sometimes it doesn't. There may be trends, though.
2. The comparison actually isn't fair. In each iteration, random search considers exactly one candidate solution, steepest ascent considers up to 2^n solutions, and first ascent is somewhere in between.

To address the first problem, we need to compare multiple runs, not just individual runs. We will get back to this problem in a bit.

To make the comparison fair, we need to use a stopping condition that amounts to the same computation time for each algorithm. In practice, one often uses *time* (because users may not understand the algorithms applied but do have an opinion on how long they are prepared to wait). Another metric for fair comparisons, which we will use, is the number of *fitness evaluations*. This is already what our random search does, but we have to adapt our hillclimbers slightly.

```

[45]: def hillclimbing_restart():
    current = get_random_solution()
    best, best_fitness = current[:], get_fitness(current)
    step = 1
    while step < max_steps:
        best_neighbour, neighbour_fitness = None, -1
        for neighbour in get_neighbours(current):
            fitness = get_fitness(neighbour)
            step += 1

            if fitness > neighbour_fitness:
                best_neighbour = neighbour
                neighbour_fitness = fitness
                if fitness > best_fitness:
                    best = current[:]
                    best_fitness = fitness
            fitness_values.append(best_fitness)

        # Random restart if no neighbour is better
        if neighbour_fitness <= get_fitness(current):
            current = get_random_solution()
            neighbour_fitness = get_fitness(current)
            step += 1
            if fitness > best_fitness:
                best = current[:]
                best_fitness = fitness
            fitness_values.append(best_fitness)
        else:
            current = best_neighbour

    return best

```

```

[46]: def hillclimbing_first_ascent():
    current = get_random_solution()
    fitness = get_fitness(current)
    best, best_fitness = current[:], fitness
    step = 1
    while step < max_steps:
        replaced = False
        for neighbour in get_neighbours(current):
            neighbour_fitness = get_fitness(neighbour)
            step += 1
            if neighbour_fitness > fitness:
                current, fitness = neighbour, neighbour_fitness
                replaced = True
            if fitness > best_fitness:
                best = current[:]

```

```

        best_fitness = fitness
        fitness_values.append(best_fitness)
        break
    else:
        fitness_values.append(best_fitness)

    # Random restart if no neighbour is better
    if not replaced:
        current = get_random_solution()
        fitness = get_fitness(current)
        step += 1
        if fitness > best_fitness:
            best, best_fitness = current[:], fitness
            fitness_values.append(best_fitness)

    return best

```

```

[47]: fitness_values = []
      randomsearch()
      f_random = fitness_values[:]

      fitness_values = []
      hillclimbing_restart()
      f_hill = fitness_values[:]

      fitness_values = []
      hillclimbing_first_ascent()
      f_first = fitness_values[:]

```

```

Iteration 0, fitness 21: [6, 0, 4, 5, 2, 7, 3, 7]
Iteration 2, fitness 23: [7, 0, 6, 3, 2, 7, 4, 6]
Iteration 82, fitness 24: [4, 6, 1, 0, 3, 7, 2, 5]
Iteration 137, fitness 25: [7, 0, 5, 3, 6, 4, 2, 4]
Iteration 320, fitness 27: [5, 3, 6, 0, 7, 1, 4, 0]
Solution fitness 27: [5, 3, 6, 0, 7, 1, 4, 0]

```

```

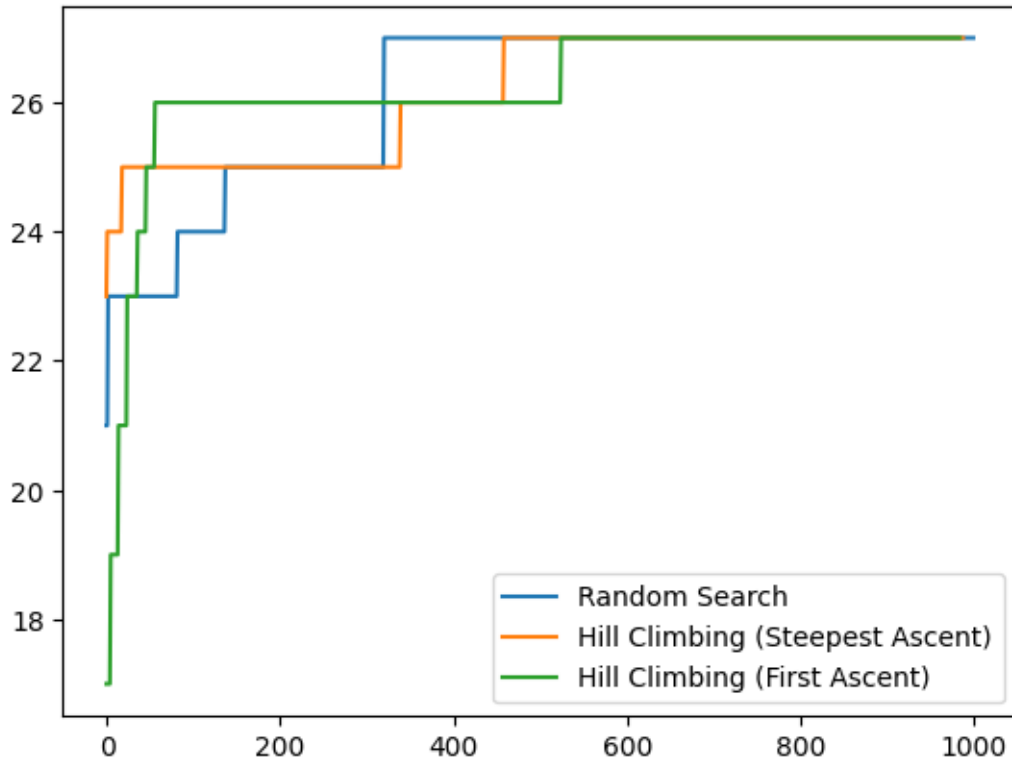
[48]: plt.plot(f_random, label="Random Search")
      plt.plot(f_hill, label="Hill Climbing (Steepest Ascent)")
      plt.plot(f_first, label="Hill Climbing (First Ascent)")
      plt.legend()

```

```

[48]: <matplotlib.legend.Legend at 0x10e59dcc0>

```



There probably isn't a clear winner yet. We will first introduce an algorithm that can cope better with local optima before turning to the question how to compare algorithms.

1.3 Tabu Search

In tabu search, a tabu list prevents re-exploration of already visited parts of search space, but therefore the search is allowed to make moves in the search space that lead to worse fitness (whereas hillclimbing only allows moves that improve fitness).

The tabu list is often implemented as a circular buffer, where solutions are added as their fitness is evaluated, and they are removed once they have been in the buffer for a certain number of steps. Let's introduce a parameter to denote the size of our buffer.

```
[49]: tabu_size = 500
```

The algorithm itself is very similar to hillclimbing, except for the following changes: - Only neighbours that are not in the tabu list are considered - The search moves to the best neighbour, even if that neighbour has worse fitness - The tabu list is updated at each step

```
[50]: def tabusearch():
    current = get_random_solution()
    fitness = get_fitness(current)
    best_solution, best_fitness = current, fitness
```

```

tabu = [current]
step = 1
while step < max_steps:

    best_neighbour = None
    neighbour_fitness = -1
    for neighbour in get_neighbours(current):
        if neighbour not in tabu:
            new_fitness = get_fitness(neighbour)
            step += 1
            if new_fitness > neighbour_fitness:
                best_neighbour, neighbour_fitness = neighbour, new_fitness
                if neighbour_fitness > best_fitness:
                    best_solution, best_fitness = best_neighbour, neighbour_fitness
            ↪neighbour_fitness
            fitness_values.append(best_fitness)

    # Append at the end of the tabu list
    tabu.append(best_neighbour)
    current = best_neighbour

    # Remove elements from the front of the tabu list
    if len(tabu) > tabu_size:
        tabu.pop(0)

    return best_solution

```

```

[51]: max_steps = 5000

fitness_values = []
randomsearch()
f_random = fitness_values[:]

fitness_values = []
tabusearch()
f_tabu = fitness_values[:]

fitness_values = []
hillclimbing_first_ascent()
f_first = fitness_values[:]

```

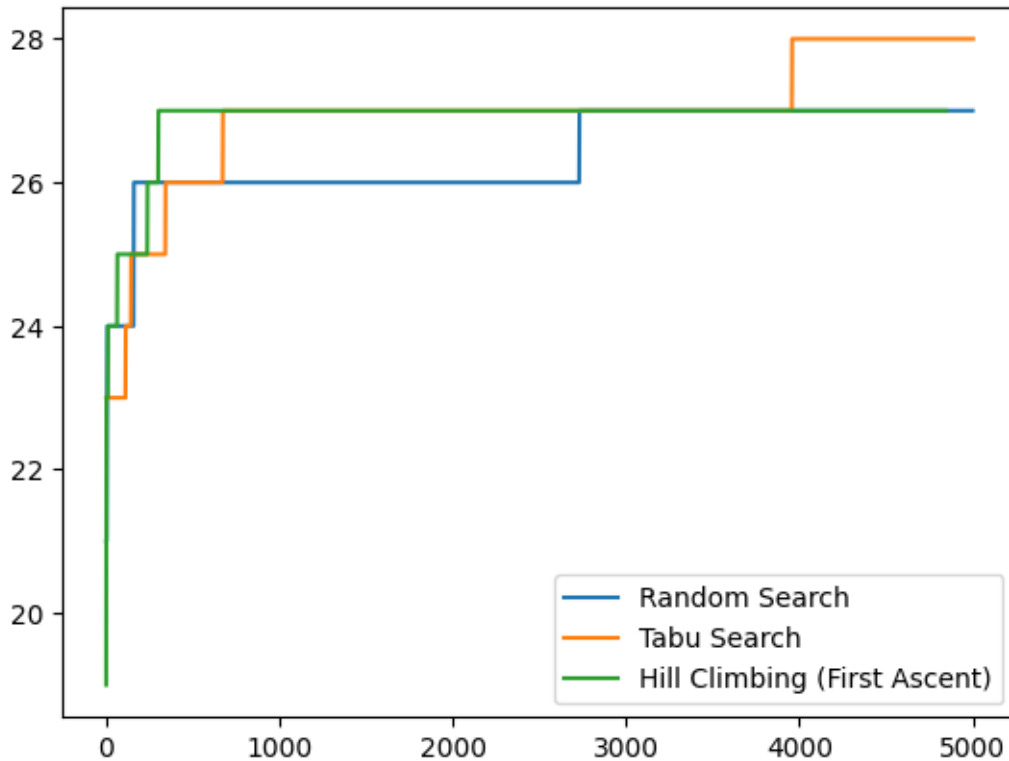
```

Iteration 0, fitness 21: [6, 3, 5, 7, 2, 2, 3, 4]
Iteration 4, fitness 22: [0, 0, 7, 5, 7, 2, 7, 2]
Iteration 6, fitness 24: [0, 7, 1, 5, 6, 2, 0, 5]
Iteration 159, fitness 26: [6, 3, 1, 7, 4, 2, 1, 5]
Iteration 2730, fitness 27: [6, 1, 3, 7, 0, 4, 1, 5]
Solution fitness 27: [6, 1, 3, 7, 0, 4, 1, 5]

```

```
[52]: plt.plot(f_random, label="Random Search")
plt.plot(f_tabu, label="Tabu Search")
plt.plot(f_first, label="Hill Climbing (First Ascent)")
plt.legend()
```

```
[52]: <matplotlib.legend.Legend at 0x10e77ffa0>
```



1.4 Comparing Search Algorithms

In order to draw a conclusion whether one randomised algorithm is better than another one, we cannot use individual runs, but need to run each algorithm multiple times, and then statistically check whether the observations result from different distributions.

As a simple example, we will compare the algorithms with respect to the fitness values achieved. Does tabu search lead to better fitness values than hillclimbing? For this, we need to run an experiment in which we run each algorithm multiple times, and collect the fitness values we want to compare.

```
[53]: from IPython.utils import io

max_steps = 1000
repetitions = 30
```

```

values_random = []
values_hill = []
values_tabu = []

for i in range(repetitions):

    print(f"Repetition {i}")
    with io.capture_output() as captured:
        fitness_values = []
        randomsearch()
        values_random.append(fitness_values[-1])

        fitness_values = []
        hillclimbing_first_ascent()
        values_hill.append(fitness_values[-1])

        fitness_values = []
        tabusearch()
        values_tabu.append(fitness_values[-1])

```

```

Repetition 0
Repetition 1
Repetition 2
Repetition 3
Repetition 4
Repetition 5
Repetition 6
Repetition 7
Repetition 8
Repetition 9
Repetition 10
Repetition 11
Repetition 12
Repetition 13
Repetition 14
Repetition 15
Repetition 16
Repetition 17
Repetition 18
Repetition 19
Repetition 20
Repetition 21
Repetition 22
Repetition 23
Repetition 24
Repetition 25
Repetition 26
Repetition 27

```


Repetition 28

Repetition 29

```
[54]: print(values_random)
      print(values_hill)
      print(values_tabu)
```

```
[26, 26, 26, 26, 26, 27, 26, 26, 26, 27, 26, 26, 26, 27, 26, 27, 26, 26, 26, 25,
26, 26, 26, 26, 26, 26, 26, 26, 26]
[27, 27, 27, 26, 26, 26, 27, 26, 26, 27, 27, 28, 27, 27, 27, 27, 28, 27, 27, 27,
26, 26, 27, 26, 27, 26, 27, 27, 28, 27]
[27, 27, 28, 28, 28, 28, 28, 27, 27, 27, 28, 27, 27, 27, 28, 28, 28, 27, 27, 28,
27, 27, 27, 28, 28, 27, 28, 27, 27, 28]
```

As a result of this experiment, we now have three lists of fitness values, one for the results achieved by random search, one for those of our hillclimber, and one for tabu search.

Is our hillclimber any better than random search? To answer this question we compare the two corresponding datasets. First, we'll just look at the means:

```
[55]: from numpy import mean
      mean(values_random)
```

[55]: 26.1

```
[56]: mean(values_hill)
```

[56]: 26.8

There may be some differences in the means, and by chance either of the two algorithms can have a higher mean. To really find out if there is a difference, we apply a statistical test. We will not make any assumptions on normality of data, and therefore apply a non-parametric test.

```
[57]: from scipy.stats import mannwhitneyu
      result = mannwhitneyu(values_random, values_hill)
      result[1]
```

[57]: 7.886512477242196e-06

By convention, a p-value lower than 0.05 is usually considered statistically significant (though this is debated). If the p-value is larger, then either there really is no difference, or the difference is so small that more observations (i.e., more repetitions in our experiment) are required in order to detect it.

As a further way to quantify the difference, one typically uses effect size measurements. Again there are different statistical measures, in particular differing whether they are standardised (i.e., independent of the measurement unit) or non-parametric (i.e., make no assumptions on the underlying distribution). An example non-parametric effect size is the Vargha-Delaney A12 effect size measurement: Given a performance measure M seen in m measures of X and n measures of Y , the A12 statistics measures the probability that running algorithm X yields higher M values than running another algorithm Y .

```
[58]: # https://gist.github.com/timm/5630491
def a12(lst1, lst2, rev=True):
    "how often is x in lst1 more than y in lst2?"
    more = same = 0.0
    for x in lst1:
        for y in lst2:
            if x==y : same += 1
            elif rev and x > y : more += 1
            elif not rev and x < y : more += 1
    return (more + 0.5*same) / (len(lst1)*len(lst2))
```

```
[59]: a12(values_random, values_hill)
```

```
[59]: 0.205
```

For the comparison (random search, hillclimbing) we most likely see an A12 value < 0.5 . The interpretation is that, if you run random search on our n-queens problem, with a probability of A12 it will produce a higher fitness value. If $A12 < 0.5$, then that means the hill climber is better. If $A12 = 0$, then random would *always* produce worse results, and if $A12 = 1$, then hillclimbing would *always* produce worse (lower) results; an A12 of 0.5 would mean both algorithms perform equally well. The value of A12 gives us an intuition of the magnitude, and a common interpretation is that a big effect is with A12 over 0.71, medium over 0.64, and small over 0.56 (and the same mirrored at 0.5).

We can now also check if tabu search is better than hillclimbing:

```
[60]: result = mannwhitneyu(values_hill, values_tabu)
result[1]
```

```
[60]: 8.54491638698574e-05
```

```
[61]: mean(values_hill)
```

```
[61]: 26.8
```

```
[62]: mean(values_tabu)
```

```
[62]: 27.466666666666665
```

```
[63]: a12(values_hill, values_tabu)
```

```
[63]: 0.23666666666666666
```

The difference between hillclimbing and tabu search on our n-queens problems tends to be much smaller than the difference between these algorithms and random search, which tabu search usually performing *slightly* better.

1.5 Simulated Annealing

The last local search algorithm we will consider is simulated annealing, which is another variant of hillclimbing with a twist in order to escape local optima. The algorithm simulates physical processes from metallurgy, where the atoms in heated metal undergo disordered movements of large amplitude, while cooling the metal down progressively reduces movement and stabilises atoms around fixed positions in a regular crystal structure with minimal energy.

In order to simulate these effects, we will require some helper maths functions.

```
[64]: import math
```

The central aspect of simulated annealing is that the local search accepts movements to neighbours with worse fitness values with a probability that is proportional to the current temperature (just like atoms move more freely in heated metal). To calculate the acceptance probability, we need to compare the current individual of the search and a neighbour. If the neighbour is better, the probability of accepting it is always 1. If it is worse, then the probability is calculated as follows:

```
[65]: def acceptance_probability(fitness, new_fitness, temperature):
        if new_fitness > fitness:
            return 1
        else:
            p = math.exp( (new_fitness - fitness) / temperature)
            return p
```

The temperature is varied along the search, starting at a high value, and gradually cooling down. It is possible to use different cooling schedules; we will use a simple schedule where the temperature is calculated based on the fraction of the overall search budget (i.e., number of steps in the search) used.

```
[66]: def temperature(fraction):
        return max(0.01, min(1, 1 - fraction))
```

Unlike hillclimbing and tabu search, in each step we only consider one random neighbour. The acceptance probability then decides whether the search moves to this individual, or remains at the current position.

```
[67]: def simulatedannealing():
        current = get_random_solution()
        fitness = get_fitness(current)

        best_solution = current
        best_fitness = fitness

        step = 1

        while step < max_steps:
            fraction = step / float(max_steps)
            T = temperature(fraction)
```

```

    neighbour = random.choice(get_neighbours(current))
    neighbour_fitness = get_fitness(neighbour)
    step += 1

    if acceptance_probability(fitness, neighbour_fitness, T) > random.
↪random():
        current, fitness = neighbour, neighbour_fitness

        if fitness > best_fitness:
            best_fitness = fitness
            best_solution = current
        fitness_values.append(best_fitness)

    return best_solution

```

In order to evaluate simulated annealing, let's first have a look at an individual run of the algorithm compared to individual runs of other algorithms.

```

[68]: max_steps = 5000

fitness_values = []
randomsearch()
f_random = fitness_values[:]

fitness_values = []
tabusearch()
f_tabu = fitness_values[:]

fitness_values = []
hillclimbing_first_ascent()
f_first = fitness_values[:]

fitness_values = []
simulatedannealing()
f_anneal = fitness_values[:]

```

```

Iteration 0, fitness 16: [0, 5, 4, 5, 4, 6, 5, 5]
Iteration 1, fitness 23: [1, 5, 2, 6, 4, 3, 6, 0]
Iteration 19, fitness 24: [3, 5, 2, 0, 4, 0, 7, 6]
Iteration 29, fitness 25: [2, 2, 5, 7, 1, 3, 0, 3]
Iteration 47, fitness 26: [1, 6, 2, 2, 7, 4, 0, 3]
Solution fitness 26: [1, 6, 2, 2, 7, 4, 0, 3]

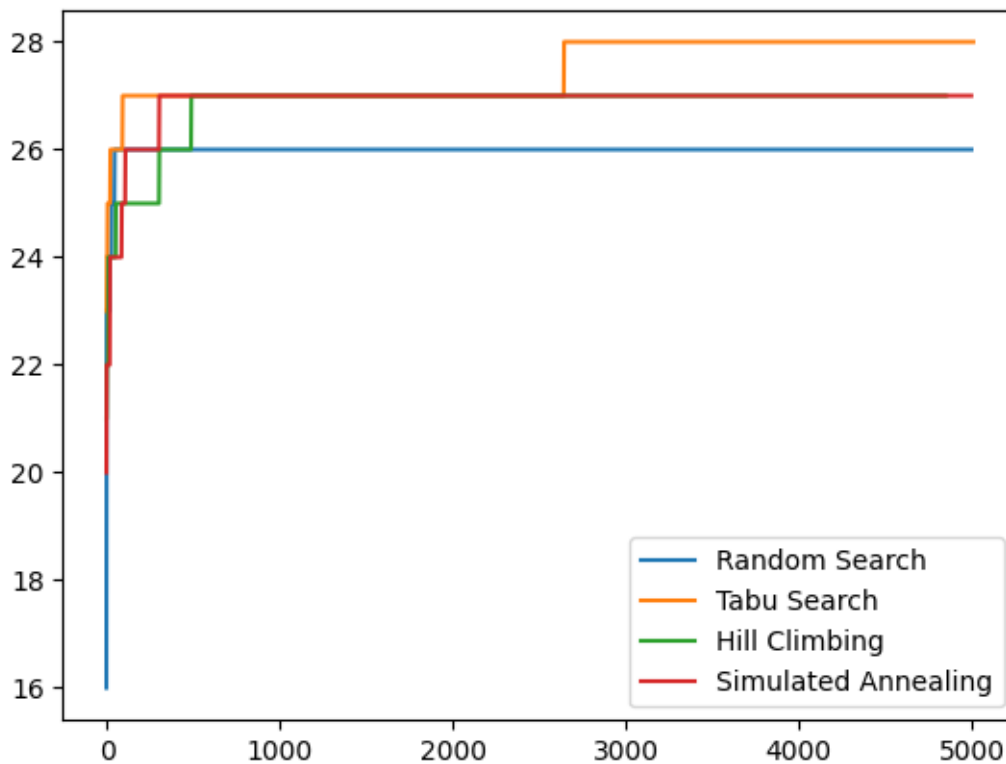
```

```

[69]: plt.plot(f_random, label="Random Search")
plt.plot(f_tabu, label="Tabu Search")
plt.plot(f_first, label="Hill Climbing")
plt.plot(f_anneal, label="Simulated Annealing")
plt.legend()

```

[69]: <matplotlib.legend.Legend at 0x16b66ab30>



To statistically compare the algorithms with each other, we need to run an experiment again for fair comparison.

```
[70]: from IPython.utils import io

max_steps = 10000
repetitions = 30

values_random = []
values_hill = []
values_tabu = []
values_anneal = []

for i in range(repetitions):

    print(f"Repetition {i}")
    with io.capture_output() as captured:
        fitness_values = []
        randomsearch()
        values_random.append(fitness_values[-1])
```

```

fitness_values = []
hillclimbing_first_ascent()
values_hill.append(fitness_values[-1])

fitness_values = []
tabusearch()
values_tabu.append(fitness_values[-1])

fitness_values = []
simulatedannealing()
values_anneal.append(fitness_values[-1])

```

```

Repetition 0
Repetition 1
Repetition 2
Repetition 3
Repetition 4
Repetition 5
Repetition 6
Repetition 7
Repetition 8
Repetition 9
Repetition 10
Repetition 11
Repetition 12
Repetition 13
Repetition 14
Repetition 15
Repetition 16
Repetition 17
Repetition 18
Repetition 19
Repetition 20
Repetition 21
Repetition 22
Repetition 23
Repetition 24
Repetition 25
Repetition 26
Repetition 27
Repetition 28
Repetition 29

```

```

[71]: print(values_random)
      print(values_hill)
      print(values_tabu)

```

```
print(values_anneal)
```

```
[26, 27, 26, 27, 27, 27, 27, 27, 27, 27, 27, 27, 26, 27, 26, 27, 27, 27, 27, 27, 27, 27, 27, 26, 27, 27, 27, 27, 27, 27, 27, 26, 27]
[27, 28, 28, 28, 28, 27, 28, 28, 28, 28, 27, 28, 28, 27, 28, 27, 27, 28, 28, 27, 27, 27, 28, 27, 28, 28, 28, 28]
[28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28]
[28, 28, 27, 27, 28, 28, 27, 28, 28, 28, 28, 27, 28, 28, 28, 27, 28, 28, 28, 28, 27, 28, 28, 28, 28, 28, 27, 28, 28, 28, 28, 28, 28, 28, 28, 28, 27, 27]
```

Is simulated annealing better than random search?

```
[72]: result = mannwhitneyu(values_random, values_anneal)
p = result[1]
a = a12(values_random, values_anneal)
print("Random search vs. Simulated Annealing: p={:f}, A12={:f}".format(p, a))
```

Random search vs. Simulated Annealing: p=0.000000, A12=0.106667

Is simulated annealing better than hillclimbing?

```
[73]: result = mannwhitneyu(values_hill, values_anneal)
p = result[1]
a = a12(values_hill, values_anneal)
print("Hill climbing vs. Simulated Annealing: p={:f}, A12={:f}".format(p, a))
```

Hill climbing vs. Simulated Annealing: p=0.582725, A12=0.466667

Is simulated annealing better than tabu search?

```
[74]: result = mannwhitneyu(values_tabu, values_anneal)
p = result[1]
a = a12(values_tabu, values_anneal)
print("Tabu search vs. Simulated Annealing: p={:f}, A12={:f}".format(p, a))
```

Tabu search vs. Simulated Annealing: p=0.002698, A12=0.633333

Most likely, simulated annealing is better than random search and hillclimbing, but there probably is no significant difference between tabu search and simulated annealing, on the n-queens problem. However, it generally is not known for a new problem which algorithm will work best.