Solving problems by Searching

This notebook serves as supporting material for topics covered in **Chapter 3 - Solving Problems by Searching** and **Chapter 4 - Beyond Classical Search** from the book *Artificial Intelligence: A Modern Approach*. This notebook uses implementations from search.py module. Let's start by importing everything from search module.

In [1]:
```
# Add --break-system-packages at the end of the pip install
# for local installs on OS's like Ubuntu
#!pip install -r requirements.txt
```

In [2]:
```
from search import *
from notebook import psource, heatmap, gaussian_kernel, show_map, final_path

# Needed to hide warnings in the matplotlib sections
import warnings
warnings.filterwarnings("ignore")
```

CONTENTS

OVERVIEW

Here, we learn about a specific kind of problem solving - building goal-based agents that can plan ahead to solve problems. In particular, we examine navigation problem/route finding problem. We must begin by precisely defining **problems** and their **solutions**. We will look at several general-purpose search algorithms.

Search algorithms can be classified into two types:

- **Uninformed search algorithms**: Search algorithms which explore the search space without having any information about the problem other than its definition.
    - Examples:
        1. Breadth First Search
        2. Depth First Search
        3. Depth Limited Search
        4. Iterative Deepening Search
- **Informed search algorithms**: These type of algorithms leverage any information (heuristics, path cost) on the problem to search through the search space to find the solution efficiently.
    - Examples:
        1. Best First Search
        2. Uniform Cost Search
        3. A* Search
        4. Recursive Best First Search

Don't miss the visualisations of these algorithms solving the route-finding problem defined on Romania map at the end of this notebook.

For visualisations, we use networkx and matplotlib to show the map in the notebook and we use ipywidgets to interact with the map to see how the searching algorithm works. These are imported as required in `notebook.py`.

In [3]:
```
%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import lines

from ipywidgets import interact
import ipywidgets as widgets
from IPython.display import display
import time
```

PROBLEM

Let's see how we define a Problem. Run the next cell to see how abstract class `Problem` is defined in the search module.

In [4]:
```
psource(Problem)
```

```python
class Problem:
    """The abstract class for a formal problem. You should subclass
    this and implement the methods actions and result, and possibly
    __init__, goal_test, and path_cost. Then you will create instances
    of your subclass and solve them with the various search functions."""

    def __init__(self, initial, goal=None):
        """The constructor specifies the initial state, and possibly a goal
        state, if there is a unique goal. Your subclass's constructor can add
        other arguments."""
        self.initial = initial
        self.goal = goal

    def actions(self, state):
        """Return the actions that can be executed in the given
        state. The result would typically be a list, but if there are
        many actions, consider yielding them one at a time in an
        iterator, rather than building them all at once."""
        raise NotImplementedError

    def result(self, state, action):
        """Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state)."""
        raise NotImplementedError

    def goal_test(self, state):
        """Return True if the state is a goal. The default method compares the
        state to self.goal or checks for state in self.goal if it is a
        list, as specified in the constructor. Override this method if
        checking against a single self.goal is not enough."""
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal

    def path_cost(self, c, state1, action, state2):
        """Return the cost of a solution path that arrives at state2 from
        state1 via action, assuming cost c to get up to state1. If the problem
        is such that the path doesn't matter, this function will only look at
        state2. If the path does matter, it will consider c and maybe state1
        and action. The default method costs 1 for every step in the path."""
        return c + 1

    def value(self, state):
        """For optimization problems, each state has a value. Hill Climbing
        and related algorithms try to maximize this value."""
        raise NotImplementedError
```

The `Problem` class has six methods.

- `__init__(self, initial, goal)` : This is what is called a `constructor`. It is the first method called when you create an instance of the class as `Problem(initial, goal)`. The variable `initial` specifies the initial state s_0 of the search problem. It represents the beginning state. From here, our agent begins its task of exploration to find the goal state(s) which is given in the `goal` parameter.
- `actions(self, state)` : This method returns all the possible actions agent can execute in the given state `state`.
- `result(self, state, action)` : This returns the resulting state if action `action` is taken in the state `state`. This `Problem` class only deals with deterministic outcomes. So we know for sure what every action in a state would result to.
- `goal_test(self, state)` : Return a boolean for a given state - `True` if it is a goal state, else `False`.
- `path_cost(self, c, state1, action, state2)` : Return the cost of the path that arrives at `state2` as a result of taking `action` from `state1`, assuming total cost of `c` to get up to `state1`.
- `value(self, state)` : This acts as a bit of extra information in problems where we try to optimise a value when we cannot do a goal test.

NODE

Let's see how we define a Node. Run the next cell to see how abstract class `Node` is defined in the search module.

```
In [5]: psource(Node)
```

```python
class Node:
    """A node in a search tree. Contains a pointer to the parent (the node
    that this is a successor of) and to the actual state for this node. Note
    that if a state is arrived at by two paths, then there are two nodes with
    the same state. Also includes the action that got us to this state, and
    the total path_cost (also known as g) to reach the node. Other functions
    may add an f and h value; see best_first_graph_search and astar_search for
    an explanation of how the f and h values are handled. You will not need to
    subclass this class."""

    def __init__(self, state, parent=None, action=None, path_cost=0):
        """Create a search tree Node, derived from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

    def __repr__(self):
        return "<Node {}>".format(self.state)

    def __lt__(self, node):
        return self.state < node.state

    def expand(self, problem):
        """List the nodes reachable in one step from this node."""
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]

    def child_node(self, problem, action):
        """[Figure 3.10]"""
        next_state = problem.result(self.state, action)
        next_node = Node(next_state, self, action, problem.path_cost(self.path_cost, self.stat
e, action, next_state))
        return next_node

    def solution(self):
        """Return the sequence of actions to go from the root to this node."""
        return [node.action for node in self.path()[1:]]
```

```
def path(self):
    """Return a list of nodes forming the path from the root to this node."""
    node, path_back = self, []
    while node:
        path_back.append(node)
        node = node.parent
    return list(reversed(path_back))

# We want for a queue of nodes in breadth_first_graph_search or
# astar_search to have no duplicated states, so we treat nodes
# with the same state as equal. [Problem: this may not be what you
# want in other contexts.]

def __eq__(self, other):
    return isinstance(other, Node) and self.state == other.state

def __hash__(self):
    # We use the hash value of the state
    # stored in the node instead of the node
    # object itself to quickly search a node
    # with the same state in a Hash Table
    return hash(self.state)
```

The `Node` class has nine methods. The first is the `__init__` method.

- `__init__(self, state, parent, action, path_cost)` : This method creates a node. `parent` represents the node that this is a successor of and `action` is the action required to get from the parent node to this node. `path_cost` is the cost to reach current node from parent node.

The next 4 methods are specific `Node` -related functions.

- `expand(self, problem)` : This method lists all the neighbouring(reachable in one step) nodes of current node.

- `child_node(self, problem, action)` : Given an `action`, this method returns the immediate neighbour that can be reached with that `action`.

- `solution(self)` : This returns the sequence of actions required to reach this node from the root node.

- `path(self)` : This returns a list of all the nodes that lies in the path from the root to this node.

The remaining 4 methods override standards Python functionality for representing an object as a string, the less-than ($<$) operator, the equal-to ($=$) operator, and the `hash` function.

- `__repr__(self)` : This returns the state of this node.
- `__lt__(self, node)` : Given a `node`, this method returns `True` if the state of current node is less than the state of the `node`. Otherwise it returns `False`.
- `__eq__(self, other)` : This method returns `True` if the state of current node is equal to the other node. Else it returns `False`.
- `__hash__(self)` : This returns the hash of the state of current node.

We will use the abstract class `Problem` to define our real **problem** named `GraphProblem`. You can see how we define `GraphProblem` by running the next cell.

```
In [6]: psource(GraphProblem)
```

```python
class GraphProblem(Problem):
    """The problem of searching a graph from one node to another."""

    def __init__(self, initial, goal, graph):
        super().__init__(initial, goal)
        self.graph = graph

    def actions(self, A):
        """The actions at a graph node are just its neighbors."""
        return list(self.graph.get(A).keys())

    def result(self, state, action):
        """The result of going to a neighbor is just that neighbor."""
        return action

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or np.inf)

    def find_min_edge(self):
        """Find minimum value of edges."""
        m = np.inf
        for d in self.graph.graph_dict.values():
            local_min = min(d.values())
            m = min(m, local_min)

        return m

    def h(self, node):
        """h function is straight-line distance from a node's state to goal."""
        locs = getattr(self.graph, 'locations', None)
        if locs:
            if type(node) is str:
                return int(distance(locs[node], locs[self.goal]))

            return int(distance(locs[node.state], locs[self.goal]))
        else:
            return np.inf
```

Have a look at our romania_map, which is an Undirected Graph containing a dict of nodes as keys and neighbours as values.

In [7]:
```
romania_map = UndirectedGraph(dict(
    Arad=dict(Zerind=75, Sibiu=140, Timisoara=118),
    Bucharest=dict(Urziceni=85, Pitesti=101, Giurgiu=90, Fagaras=211),
    Craiova=dict(Drobeta=120, Rimnicu=146, Pitesti=138),
    Drobeta=dict(Mehadia=75),
    Eforie=dict(Hirsova=86),
    Fagaras=dict(Sibiu=99),
    Hirsova=dict(Urziceni=98),
    Iasi=dict(Vaslui=92, Neamt=87),
    Lugoj=dict(Timisoara=111, Mehadia=70),
    Oradea=dict(Zerind=71, Sibiu=151),
    Pitesti=dict(Rimnicu=97),
    Rimnicu=dict(Sibiu=80),
    Urziceni=dict(Vaslui=142)))

romania_map.locations = dict(
    Arad=(91, 492), Bucharest=(400, 327), Craiova=(253, 288),
    Drobeta=(165, 299), Eforie=(562, 293), Fagaras=(305, 449),
    Giurgiu=(375, 270), Hirsova=(534, 350), Iasi=(473, 506),
    Lugoj=(165, 379), Mehadia=(168, 339), Neamt=(406, 537),
    Oradea=(131, 571), Pitesti=(320, 368), Rimnicu=(233, 410),
    Sibiu=(207, 457), Timisoara=(94, 410), Urziceni=(456, 350),
    Vaslui=(509, 444), Zerind=(108, 531))
```

It is pretty straightforward to understand this `romania_map`. The first node **Arad** has three neighbours named **Zerind**, **Sibiu**, **Timisoara**. Each of these nodes are 75, 140, 118 units apart from **Arad** respectively. And the same goes with other nodes.

And `romania_map.locations` contains the positions of each of the nodes. We will use the straight line distance (which is different from the one provided in `romania_map`) between two cities in algorithms like A*-search and Recursive Best First Search.

Define a problem: Now it's time to define our problem. We will define it by passing `initial`, `goal`, `graph` to `GraphProblem`. So, our problem is to find the goal state starting from the given initial state on the provided graph.

Say we want to start exploring from **Arad** and try to find **Bucharest** in our romania_map. So, this is how we do it.

In [8]:
```
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
```

Romania Map Visualisation

Let's have a visualisation of Romania map [Figure 3.2] from the book and see how different searching algorithms perform / how frontier expands in each search algorithm for a simple problem named `romania_problem`.

Have a look at `romania_locations`. It is a dictionary defined in search module. We will use these location values to draw the romania graph using **networkx**.

```
In [9]: romania_locations = romania_map.locations
        print(romania_locations)
```

```
{'Arad': (91, 492), 'Bucharest': (400, 327), 'Craiova': (253, 288), 'Drobet
a': (165, 299), 'Eforie': (562, 293), 'Fagaras': (305, 449), 'Giurgiu': (37
5, 270), 'Hirsova': (534, 350), 'Iasi': (473, 506), 'Lugoj': (165, 379), 'Me
hadia': (168, 339), 'Neamt': (406, 537), 'Oradea': (131, 571), 'Pitesti': (3
20, 368), 'Rimnicu': (233, 410), 'Sibiu': (207, 457), 'Timisoara': (94, 41
0), 'Urziceni': (456, 350), 'Vaslui': (509, 444), 'Zerind': (108, 531)}
```

Let's get started by initializing an empty graph. We will add nodes, place the nodes in their location as shown in the book, add edges to the graph.

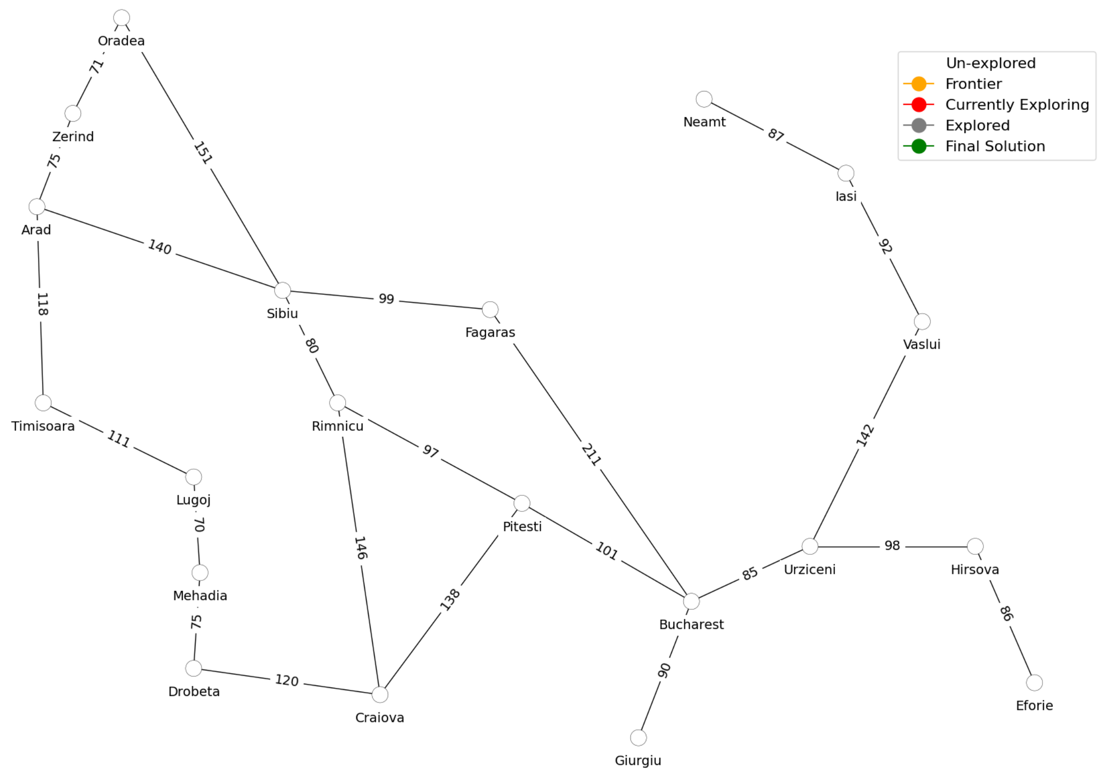
```
In [10]: # node colors, node positions and node label positions
         node_colors = {node: 'white' for node in romania_map.locations.keys()}
         node_positions = romania_map.locations
         node_label_pos = { k:[v[0],v[1]-10]  for k,v in romania_map.locations.items(
         edge_weights = {(k, k2) : v2 for k, v in romania_map.graph_dict.items() for

         romania_graph_data = {  'graph_dict' : romania_map.graph_dict,
                                 'node_colors': node_colors,
                                 'node_positions': node_positions,
                                 'node_label_positions': node_label_pos,
                                  'edge_weights': edge_weights
                              }
```

We have completed building our graph based on romania_map and its locations. It's time to display it here in the notebook. This function `show_map(node_colors)` helps us do that. We will be calling this function later on to display the map at each and every interval step while searching, using variety of algorithms from the book.

We can simply call the function with node_colors dictionary object to display it.

```
In [11]: show_map(romania_graph_data)
```



Voila! You see, the romania map as shown in the Figure[3.2] in the book. Now, see how different searching algorithms perform with our problem statements.

SIMPLE PROBLEM SOLVING AGENT PROGRAM

Let us now define a Simple Problem Solving Agent Program. Run the next cell to see how the abstract class `SimpleProblemSolvingAgentProgram` is defined in the search module.

```
In [12]: psource(SimpleProblemSolvingAgentProgram)
```

```python
class SimpleProblemSolvingAgentProgram:
    """
    [Figure 3.1]
    Abstract framework for a problem-solving agent.
    """

    def __init__(self, initial_state=None):
        """State is an abstract representation of the state
        of the world, and seq is the list of actions required
        to get to a particular state from the initial state(root)."""
        self.state = initial_state
        self.seq = []

    def __call__(self, percept):
        """[Figure 3.1] Formulate a goal and problem, then
        search for a sequence of actions to solve it."""
        self.state = self.update_state(self.state, percept)
        if not self.seq:
            goal = self.formulate_goal(self.state)
            problem = self.formulate_problem(self.state, goal)
            self.seq = self.search(problem)
            if not self.seq:
                return None
        return self.seq.pop(0)

    def update_state(self, state, percept):
        raise NotImplementedError

    def formulate_goal(self, state):
        raise NotImplementedError

    def formulate_problem(self, state, goal):
        raise NotImplementedError

    def search(self, problem):
        raise NotImplementedError
```

The SimpleProblemSolvingAgentProgram class has six methods:

- `__init__(self, intial_state=None)` : This is the `contructor` of the class and is the first method to be called when the class is instantiated. It takes in a keyword argument, `initial_state` which is initially `None`. The argument `initial_state` represents the state from which the agent starts.

- `__call__(self, percept)`: This method updates the `state` of the agent based on its `percept` using the `update_state` method. It then formulates a `goal` with the help of `formulate_goal` method and a `problem` using the `formulate_problem` method and returns a sequence of actions to solve it (using the `search` method).

- `update_state(self, percept)`: This method updates the `state` of the agent based on its `percept`.

- `formulate_goal(self, state)`: Given a `state` of the agent, this method formulates the `goal` for it.

- `formulate_problem(self, state, goal)`: It is used in problem formulation given a `state` and a `goal` for the `agent`.

- `search(self, problem)`: This method is used to search a sequence of `actions` to solve a `problem`.

Let us now define a Simple Problem Solving Agent Program. We will create a simple `vacuumAgent` class which will inherit from the abstract class `SimpleProblemSolvingAgentProgram` and overrides its methods. We will create a simple intelligent vacuum agent which can be in any one of the following states. It will move to any other state depending upon the current state as shown in the picture by arrows:

simple problem solving agent

In [13]:
```
class vacuumAgent(SimpleProblemSolvingAgentProgram):
        def update_state(self, state, percept):
            return percept

        def formulate_goal(self, state):
            goal = [state7, state8]
            return goal

        def formulate_problem(self, state, goal):
            problem = state
            return problem

        def search(self, problem):
            if problem == state1:
                seq = ["Suck", "Right", "Suck"]
            elif problem == state2:
                seq = ["Suck", "Left", "Suck"]
            elif problem == state3:
                seq = ["Right", "Suck"]
            elif problem == state4:
                seq = ["Suck"]
            elif problem == state5:
                seq = ["Suck"]
            elif problem == state6:
                seq = ["Left", "Suck"]
            return seq
```

Now, we will define all the 8 states and create an object of the above class. Then, we will pass it different states and check the output:

In [14]:
```
state1 = [(0, 0), [(0, 0), "Dirty"], [(1, 0), ["Dirty"]]]
state2 = [(1, 0), [(0, 0), "Dirty"], [(1, 0), ["Dirty"]]]
state3 = [(0, 0), [(0, 0), "Clean"], [(1, 0), ["Dirty"]]]
state4 = [(1, 0), [(0, 0), "Clean"], [(1, 0), ["Dirty"]]]
state5 = [(0, 0), [(0, 0), "Dirty"], [(1, 0), ["Clean"]]]
state6 = [(1, 0), [(0, 0), "Dirty"], [(1, 0), ["Clean"]]]
state7 = [(0, 0), [(0, 0), "Clean"], [(1, 0), ["Clean"]]]
state8 = [(1, 0), [(0, 0), "Clean"], [(1, 0), ["Clean"]]]

a = vacuumAgent(state1)

print(a(state6))
print(a(state1))
print(a(state3))
```

```
Left
Suck
Right
```

SEARCHING ALGORITHMS VISUALIZATION

In this section, we have visualizations of the following searching algorithms:

1. Breadth First Tree Search
2. Depth First Tree Search
3. Breadth First Search
4. Depth First Graph Search
5. Best First Graph Search
6. Uniform Cost Search
7. Depth Limited Search
8. Iterative Deepening Search
9. Greedy Best First Search
10. A*-Search
11. Recursive Best First Search

We add the colors to the nodes to have a nice visualisation when displaying. So, these are the different colors we are using in these visuals:

- Un-explored nodes - white
- Frontier nodes - orange
- Currently exploring node - red

- Already explored nodes - gray

1. BREADTH-FIRST TREE SEARCH

We have a working implementation in search module. But as we want to interact with the graph while it is searching, we need to modify the implementation. Here's the modified breadth first tree search.

In [15]:
```python
def tree_breadth_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    #Adding first node to the queue
    frontier = deque([Node(problem.initial)])

    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        #Popping first node of queue
        node = frontier.popleft()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        frontier.extend(node.expand(problem))

        for n in node.expand(problem):
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
    return None

def breadth_first_tree_search_visual(problem):
    "Search the shallowest nodes in the search tree first."
    iterations, all_node_colors, node = tree_breadth_search_for_vis(problem)
    return(iterations, all_node_colors, node)
```

Now, we use `ipywidgets` to display a slider, a button and our romania map. By sliding the slider we can have a look at all the intermediate steps of a particular search algorithm. By pressing the button **Visualize**, you can see all the steps without interacting with the slider. These two helper functions are the callback functions which are called when we interact with the slider and the button.

In [16]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
a, b, c = breadth_first_tree_search_visual(romania_problem)
display_visual(romania_graph_data, user_input=False,
               algorithm=breadth_first_tree_search_visual,
               problem=romania_problem)
```

iteration 0

visualize

2. DEPTH-FIRST TREE SEARCH

Now let's discuss another searching algorithm, Depth-First Tree Search.

In [17]:
```
def tree_depth_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    #Adding first node to the stack
    frontier = [Node(problem.initial)]

    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        #Popping first node of stack
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        frontier.extend(node.expand(problem))

        for n in node.expand(problem):
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return None

def depth_first_tree_search_visual(problem):
    "Search the deepest nodes in the search tree first."
    iterations, all_node_colors, node = tree_depth_search_for_vis(problem)
    return(iterations, all_node_colors, node)
```

In [18]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_first_tree_search_visual,
               problem=romania_problem)
```

iteration 0

visualize

3. BREADTH-FIRST GRAPH SEARCH

Let's change all the `node_colors` to starting position and define a different problem statement.

In [19]:
```
def breadth_first_search_graph(problem):
    "[Figure 3.11]"

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    node = Node(problem.initial)
```

```python
    node_colors[node.state] = "red"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    if problem.goal_test(node.state):
        node_colors[node.state] = "green"
        iterations += 1
        all_node_colors.append(dict(node_colors))
        return(iterations, all_node_colors, node)

    frontier = deque([node])

    # modify the color of frontier nodes to blue
    node_colors[node.state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    explored = set()
    while frontier:
        node = frontier.popleft()
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        explored.add(node.state)

        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    node_colors[child.state] = "green"
                    iterations += 1
                    all_node_colors.append(dict(node_colors))
                    return(iterations, all_node_colors, child)
                frontier.append(child)

                node_colors[child.state] = "orange"
                iterations += 1
                all_node_colors.append(dict(node_colors))

        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
    return None
```

In [20]:
```python
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=breadth_first_search_graph,
               problem=romania_problem)
```

iteration 0

visualize

4. DEPTH-FIRST GRAPH SEARCH

Although we have a working implementation in search module, we have to make a few changes in the algorithm to make it suitable for visualization.

In [21]:
```python
def graph_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Figure 3.7]"""
    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    frontier = [(Node(problem.initial))]
    explored = set()

    # modify the color of frontier nodes to orange
    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        # Popping first node of stack
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)

        for n in frontier:
            # modify the color of frontier nodes to orange
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
    return None


def depth_first_graph_search(problem):
    """Search the deepest nodes in the search tree first."""
    iterations, all_node_colors, node = graph_search_for_vis(problem)
    return(iterations, all_node_colors, node)
```

In [22]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_first_graph_search,
               problem=romania_problem)
```

iteration 0

visualize

5. BEST FIRST SEARCH

Let's change all the `node_colors` to starting position and define a different problem statement.

In [23]:
```
def best_first_graph_search_for_vis(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned.""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    f = memoize(f, 'f')
    node = Node(problem.initial)

    node_colors[node.state] = "red"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    if problem.goal_test(node.state):
        node_colors[node.state] = "green"
        iterations += 1
        all_node_colors.append(dict(node_colors))
        return(iterations, all_node_colors, node)

    frontier = PriorityQueue('min', f)
    frontier.append(node)
    node_colors[node.state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    explored = set()
    while frontier:
        node = frontier.pop()

        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
                node_colors[child.state] = "orange"
                iterations += 1
                all_node_colors.append(dict(node_colors))
            elif child in frontier:
                incumbent = frontier[child]
                if f(child) < incumbent:
                    del frontier[child]
                    frontier.append(child)
                    node_colors[child.state] = "orange"
                    iterations += 1
                    all_node_colors.append(dict(node_colors))

        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
    return None
```

6. UNIFORM COST SEARCH

Let's change all the `node_colors` to starting position and define a different problem statement.

In [24]:
```
def uniform_cost_search_graph(problem):
    "[Figure 3.14]"
    #Uniform Cost Search uses Best First Search algorithm with f(n) = g(n)
    iterations, all_node_colors, node = best_first_graph_search_for_vis(prob
    return(iterations, all_node_colors, node)
```

In [25]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
```

```
display_visual(romania_graph_data, user_input=False,
               algorithm=uniform_cost_search_graph,
               problem=romania_problem)
```

iteration 0

visualize

7. DEPTH LIMITED SEARCH

Let's change all the 'node_colors' to starting position and define a different problem statement.
Although we have a working implementation, but we need to make changes.

In [26]:
```
def depth_limited_search_graph(problem, limit = -1):
    '''
    Perform depth first search of graph g.
    if limit >= 0, that is the maximum depth of the search.
    '''
    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    frontier = [Node(problem.initial)]
    explored = set()

    cutoff_occurred = False
    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        # Popping first node of queue
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        elif limit >= 0:
            cutoff_occurred = True
            limit += 1
            all_node_colors.pop()
            iterations -= 1
            node_colors[node.state] = "gray"


        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)

        for n in frontier:
            limit -= 1
            # modify the color of frontier nodes to orange
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return 'cutoff' if cutoff_occurred else None


def depth_limited_search_for_vis(problem):
    """Search the deepest nodes in the search tree first."""
    iterations, all_node_colors, node = depth_limited_search_graph(problem)
    return(iterations, all_node_colors, node)
```

In [27]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_limited_search_for_vis,
               problem=romania_problem)
```

iteration 0

visualize

8. ITERATIVE DEEPENING SEARCH

Let's change all the 'node_colors' to starting position and define a different problem statement.

In [28]:
```
def iterative_deepening_search_for_vis(problem):
    for depth in range(sys.maxsize):
        iterations, all_node_colors, node=depth_limited_search_for_vis(probl
        if iterations:
            return (iterations, all_node_colors, node)
```

In [29]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=iterative_deepening_search_for_vis,
               problem=romania_problem)
```

iteration 0

visualize

9. GREEDY BEST FIRST SEARCH

Let's change all the node_colors to starting position and define a different problem statement.

In [30]:
```
def greedy_best_first_search(problem, h=None):
    """Greedy Best-first graph search is an informative searching algorithm
    You need to specify the h function when you call best_first_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    iterations, all_node_colors, node = best_first_graph_search_for_vis(prob
    return(iterations, all_node_colors, node)
```

In [31]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=greedy_best_first_search,
               problem=romania_problem)
```

iteration 0

visualize

10. A* SEARCH

Let's change all the `node_colors` to starting position and define a different problem statement.

In [32]:
```
def astar_search_graph(problem, h=None):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    iterations, all_node_colors, node = best_first_graph_search_for_vis(prob
                                                            lambda n: n.
    return(iterations, all_node_colors, node)
```

In [33]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=astar_search_graph,
               problem=romania_problem)
```

iteration 0

visualize

11. RECURSIVE BEST FIRST SEARCH

Let's change all the `node_colors` to starting position and define a different problem statement.

In [34]:
```
def recursive_best_first_search_for_vis(problem, h=None):
    """[Figure 3.26] Recursive best-first search"""
    # we use these two variables at the time of visualizations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    h = memoize(h or problem.h, 'h')

    def RBFS(problem, node, flimit):
        nonlocal iterations
        def color_city_and_update_map(node, color):
            node_colors[node.state] = color
            nonlocal iterations
            iterations += 1
            all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            color_city_and_update_map(node, 'green')
            return (iterations, all_node_colors, node), 0  # the second valu

        successors = node.expand(problem)
        if len(successors) == 0:
            color_city_and_update_map(node, 'gray')
            return (iterations, all_node_colors, None), infinity

        for s in successors:
            color_city_and_update_map(s, 'orange')
            s.f = max(s.path_cost + h(s), node.f)

        while True:
            # Order by lowest f value
            successors.sort(key=lambda x: x.f)
            best = successors[0]
            if best.f > flimit:
                color_city_and_update_map(node, 'gray')
                return (iterations, all_node_colors, None), best.f

            if len(successors) > 1:
                alternative = successors[1].f
            else:
                alternative = infinity

            node_colors[node.state] = 'gray'
```

```
                node_colors[best.state] = 'red'
                iterations += 1
                all_node_colors.append(dict(node_colors))
                result, best.f = RBFS(problem, best, min(flimit, alternative))
                if result[2] is not None:
                    color_city_and_update_map(node, 'green')
                    return result, best.f
                else:
                    color_city_and_update_map(node, 'red')

        node = Node(problem.initial)
        node.f = h(node)

        node_colors[node.state] = 'red'
        iterations += 1
        all_node_colors.append(dict(node_colors))
        result, bestf = RBFS(problem, node, float('inf'))
        return result
```

In [35]:
```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=recursive_best_first_search_for_vis,
               problem=romania_problem)
```

iteration 0

visualize

In [36]:
```
all_node_colors = []
# display_visual(romania_graph_data, user_input=True, algorithm=breadth_firs
algorithms = {  "Breadth First Tree Search": tree_breadth_search_for_vis,
                "Depth First Tree Search": tree_depth_search_for_vis,
                "Breadth First Search": breadth_first_search_graph,
                "Depth First Graph Search": graph_search_for_vis,
                "Best First Graph Search": best_first_graph_search_for_vis,
                "Uniform Cost Search": uniform_cost_search_graph,
                "Depth Limited Search": depth_limited_search_for_vis,
                "Iterative Deepening Search": iterative_deepening_search_for
                "Greedy Best First Search": greedy_best_first_search,
                "A-star Search": astar_search_graph,
                "Recursive Best First Search": recursive_best_first_search_f
display_visual(romania_graph_data, algorithm=algorithms, user_input=True)
```

Search algor… Breadth First Tree Search

Start city: Arad

Goal city: Fagaras

visualize

iteration 0

RECURSIVE BEST-FIRST SEARCH

Recursive best-first search is a simple recursive algorithm that improves upon heuristic search by reducing the memory requirement. RBFS uses only linear space and it attempts to mimic the operation of standard best-first search. Its structure is similar to recursive depth-first search but it doesn't continue indefinitely down the current path, the `f_limit` variable is used to keep track of the f-value of the best *alternative* path available from any ancestor of the current node. RBFS remembers the f-value of the best leaf in the forgotten subtree and can decide whether it is worth re-expanding the tree later.
However, RBFS still suffers from excessive node regeneration.
Let's have a look at the implementation.

In [37]:
```
psource(recursive_best_first_search)
```

```
def recursive_best_first_search(problem, h=None):
    """[Figure 3.26]"""
    h = memoize(h or problem.h, 'h')

    def RBFS(problem, node, flimit):
        if problem.goal_test(node.state):
            return node, 0  # (The second value is immaterial)
        successors = node.expand(problem)
        if len(successors) == 0:
            return None, np.inf
        for s in successors:
            s.f = max(s.path_cost + h(s), node.f)
        while True:
            # Order by lowest f value
            successors.sort(key=lambda x: x.f)
            best = successors[0]
            if best.f > flimit:
                return None, best.f
            if len(successors) > 1:
                alternative = successors[1].f
            else:
                alternative = np.inf
            result, best.f = RBFS(problem, best, min(flimit, alternative))
            if result is not None:
                return result, best.f

    node = Node(problem.initial)
    node.f = h(node)
    result, bestf = RBFS(problem, node, np.inf)
    return result
```

This is how `recursive_best_first_search` can solve the `romania_problem`

In [38]:
```
recursive_best_first_search(romania_problem).solution()
```

Out[38]:
```
['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest']
```

`recursive_best_first_search` can be used to solve the 8 puzzle problem too, as discussed later.

In [39]:
```
puzzle = EightPuzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
assert puzzle.check_solvability((2, 4, 3, 1, 5, 6, 7, 8, 0))
recursive_best_first_search(puzzle).solution()
```

```
Out[39]: ['UP', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'DOWN']
```

A* HEURISTICS

Different heuristics provide different efficiency in solving A* problems which are generally defined by the number of explored nodes as well as the branching factor. With the classic 8 puzzle we can show the efficiency of different heuristics through the number of explored nodes.

8 Puzzle Problem

The *8 Puzzle Problem* consists of a 3x3 tray in which the goal is to get the initial configuration to the goal state by shifting the numbered tiles into the blank space.

example:-

```
         Initial State                                  Goal State
         | 7 | 2 | 4 |                                 | 1 | 2 | 3 |
         | 5 | 0 | 6 |                                 | 4 | 5 | 6 |
         | 8 | 3 | 1 |                                 | 7 | 8 | 0 |
```

We have a total of 9 blank tiles giving us a total of 9! initial configuration but not all of these are solvable. The solvability of a configuration can be checked by calculating the Inversion Permutation. If the total Inversion Permutation is even then the initial configuration is solvable else the initial configuration is not solvable which means that only 9!/2 initial states lead to a solution.
Let's define our goal state.

```
In [40]: goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

Heuristics :-

1. Manhattan Distance:- For the 8 puzzle problem Manhattan distance is defined as the distance of a tile from its goal state(for the tile numbered '1' in the initial configuration Manhattan distance is 4 "2 for left and 2 for upward displacement").
2. No. of Misplaced Tiles:- The heuristic calculates the number of misplaced tiles between the current state and goal state.
3. Sqrt of Manhattan Distance:- It calculates the square root of Manhattan distance.
4. Max Heuristic:- It assign the score as the maximum between "Manhattan Distance" and "No. of Misplaced Tiles".

In [41]:
```
# Heuristics for 8 Puzzle Problem
import math

def linear(node):
    return sum([1 if node.state[i] != goal[i] else 0 for i in range(8)])

def manhattan(node):
    state = node.state
    index_goal = {0:[2,2], 1:[0,0], 2:[0,1], 3:[0,2], 4:[1,0], 5:[1,1], 6:[1
    index_state = {}
    index = [[0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2]]
    x, y = 0, 0

    for i in range(len(state)):
        index_state[state[i]] = index[i]

    mhd = 0

    for i in range(8):
        for j in range(2):
            mhd = abs(index_goal[i][j] - index_state[i][j]) + mhd

    return mhd

def sqrt_manhattan(node):
    state = node.state
    index_goal = {0:[2,2], 1:[0,0], 2:[0,1], 3:[0,2], 4:[1,0], 5:[1,1], 6:[1
    index_state = {}
    index = [[0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2]]
    x, y = 0, 0

    for i in range(len(state)):
        index_state[state[i]] = index[i]

    mhd = 0

    for i in range(8):
        for j in range(2):
            mhd = (index_goal[i][j] - index_state[i][j])**2 + mhd

    return math.sqrt(mhd)

def max_heuristic(node):
    score1 = manhattan(node)
    score2 = linear(node)
    return max(score1, score2)
```

We can solve the puzzle using the `astar_search` method.

In [42]:
```
# Solving the puzzle
puzzle = EightPuzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
puzzle.check_solvability((2, 4, 3, 1, 5, 6, 7, 8, 0)) # checks whether the i
```

Out[42]:
```
True
```

This case is solvable, let's proceed.
The default heuristic function returns the number of misplaced tiles.

```
In [43]: astar_search(puzzle).solution()
```

```
Out[43]: ['UP', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'DOWN']
```

In the following cells, we use different heuristic functions.

```
In [44]: astar_search(puzzle, linear).solution()
```

```
Out[44]: ['UP', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'DOWN']
```

```
In [45]: astar_search(puzzle, manhattan).solution()
```

```
Out[45]: ['LEFT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'DOWN', 'RIGHT']
```

```
In [46]: astar_search(puzzle, sqrt_manhattan).solution()
```

```
Out[46]: ['LEFT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'DOWN', 'RIGHT']
```

```
In [47]: astar_search(puzzle, max_heuristic).solution()
```

```
Out[47]: ['LEFT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'DOWN', 'RIGHT']
```

And here's how `recursive_best_first_search` can be used to solve this problem too.

```
In [48]: recursive_best_first_search(puzzle, manhattan).solution()
```

```
Out[48]: ['LEFT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'DOWN', 'UP', 'DOWN', 'RIGH
         T']
```

Even though all the heuristic functions give the same solution, the difference lies in the computation time.
This might make all the difference in a scenario where high computational efficiency is required.
Let's define a few puzzle states and time `astar_search` for every heuristic function.
We will use the %%timeit magic for this.

```
In [49]: puzzle_1 = EightPuzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
         puzzle_2 = EightPuzzle((1, 2, 3, 4, 5, 6, 0, 7, 8))
         puzzle_3 = EightPuzzle((1, 2, 3, 4, 5, 7, 8, 6, 0))
```

The default heuristic function is the same as the `linear` heuristic function, but we'll still check both.

```
In [50]: %%timeit
         astar_search(puzzle_1)
astar_search(puzzle_2)
astar_search(puzzle_3)
```

```
862 µs ± 3.66 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [51]:
```
%%timeit
astar_search(puzzle_1, linear)
astar_search(puzzle_2, linear)
astar_search(puzzle_3, linear)
```

```
832 µs ± 19.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [52]:
```
%%timeit
astar_search(puzzle_1, manhattan)
astar_search(puzzle_2, manhattan)
astar_search(puzzle_3, manhattan)
```

```
731 µs ± 2.57 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [53]:
```
%%timeit
astar_search(puzzle_1, sqrt_manhattan)
astar_search(puzzle_2, sqrt_manhattan)
astar_search(puzzle_3, sqrt_manhattan)
```

```
8.63 ms ± 82.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [54]:
```
%%timeit
astar_search(puzzle_1, max_heuristic)
astar_search(puzzle_2, max_heuristic)
astar_search(puzzle_3, max_heuristic)
```

```
809 µs ± 9.25 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

We can infer that the `manhattan` heuristic function works the fastest.
`sqrt_manhattan` has an extra `sqrt` operation which makes it quite a lot slower than the others.
`max_heuristic` should have been a bit slower as it calls two functions, but in this case, those values were already calculated which saved some time. Feel free to play around with these functions.

For comparison, this is how RBFS performs on this problem.

In [55]:
```
%%timeit
recursive_best_first_search(puzzle_1, linear)
recursive_best_first_search(puzzle_2, linear)
recursive_best_first_search(puzzle_3, linear)
```

```
29.3 ms ± 666 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

It is quite a lot slower than `astar_search` as we can see.

HILL CLIMBING

Hill Climbing is a heuristic search used for optimization problems. Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may or may not be the global optimum. The algorithm is a variant of generate and test algorithm.
As a whole, the algorithm works as follows:

- Evaluate the initial state.
- If it is equal to the goal state, return.
- Find a neighboring state (one which is heuristically similar to the current state)
- Evaluate this state. If it is closer to the goal state than before, replace the initial state with this state and repeat these steps.

In [56]:
```
psource(hill_climbing)
```

```
def hill_climbing(problem):
    """
    [Figure 4.2]
    From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better.
    """
    current = Node(problem.initial)
    while True:
        neighbors = current.expand(problem)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors, key=lambda node: problem.value(node.st
ate))
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current = neighbor
    return current.state
```

We will find an approximate solution to the traveling salespersons problem using this algorithm.
We need to define a class for this problem.
`Problem` will be used as a base class.

In [57]:
```
class TSP_problem(Problem):

    """ subclass of Problem to define various functions """

    def two_opt(self, state):
        """ Neighbour generating function for Traveling Salesman Problem """
        neighbour_state = state[:]
        left = random.randint(0, len(neighbour_state) - 1)
        right = random.randint(0, len(neighbour_state) - 1)
        if left > right:
            left, right = right, left
        neighbour_state[left: right + 1] = reversed(neighbour_state[left: ri
        return neighbour_state

    def actions(self, state):
        """ action that can be excuted in given state """
        return [self.two_opt]

    def result(self, state, action):
        """  result after applying the given action on the given state """
        return action(state)

    def path_cost(self, c, state1, action, state2):
        """ total distance for the Traveling Salesman to be covered if in st
        cost = 0
        for i in range(len(state2) - 1):
            cost += distances[state2[i]][state2[i + 1]]
        cost += distances[state2[0]][state2[-1]]
        return cost

    def value(self, state):
        """ value of path cost given negative for the given state """
        return -1 * self.path_cost(None, None, None, state)
```

We will use cities from the Romania map as our cities for this problem.
A list of all cities and a dictionary storing distances between them will be populated.

In [58]:
```
distances = {}
all_cities = []

for city in romania_map.locations.keys():
    distances[city] = {}
    all_cities.append(city)

all_cities.sort()
print(all_cities)
```

```
['Arad', 'Bucharest', 'Craiova', 'Drobeta', 'Eforie', 'Fagaras', 'Giurgiu',
'Hirsova', 'Iasi', 'Lugoj', 'Mehadia', 'Neamt', 'Oradea', 'Pitesti', 'Rimnic
u', 'Sibiu', 'Timisoara', 'Urziceni', 'Vaslui', 'Zerind']
```

Next, we need to populate the individual lists inside the dictionary with the manhattan distance between the cities.

In [59]:
```
import numpy as np
for name_1, coordinates_1 in romania_map.locations.items():
        for name_2, coordinates_2 in romania_map.locations.items():
            distances[name_1][name_2] = np.linalg.norm(
                [coordinates_1[0] - coordinates_2[0], coordinates_1[1] - coo
            distances[name_2][name_1] = np.linalg.norm(
                [coordinates_1[0] - coordinates_2[0], coordinates_1[1] - coo
```

The way neighbours are chosen currently isn't suitable for the travelling salespersons problem. We need a neighboring state that is similar in total path distance to the current state.
We need to change the function that finds neighbors.

In [60]:
```python
def tsp_hill_climbing(problem):

    """From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better. [Figure 4.2]"""

    def find_neighbors(state, number_of_neighbors=100):
        """ finds neighbors using two_opt method """

        neighbors = []

        for i in range(number_of_neighbors):
            new_state = problem.two_opt(state)
            neighbors.append(Node(new_state))
            state = new_state

        return neighbors

    # as this is a stochastic algorithm, we will set a cap on the number of
    iterations = 10000
    current = Node(problem.initial)

    while iterations:
        neighbors = find_neighbors(current.state)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors,
                                     key=lambda node: -problem.value(node.st
        if problem.value(neighbor.state) <= problem.value(current.state):
            # Note that it is based on negative path cost method
            current.state = neighbor.state
        else:
            break
        iterations -= 1

    return current.state
```

An instance of the TSP_problem class will be created.

In [61]:
```python
tsp = TSP_problem(all_cities)
```

We can now generate an approximate solution to the problem by calling `hill_climbing`. The results will vary a bit each time you run it.

In [62]:
```python
tsp_hill_climbing(tsp)
```

```
Out[62]: ['Urziceni',
          'Craiova',
          'Arad',
          'Eforie',
          'Lugoj',
          'Hirsova',
          'Timisoara',
          'Pitesti',
          'Rimnicu',
          'Drobeta',
          'Sibiu',
          'Vaslui',
          'Fagaras',
          'Mehadia',
          'Iasi',
          'Neamt',
          'Giurgiu',
          'Oradea',
          'Bucharest',
          'Zerind']
```

The solution looks like this. It is not difficult to see why this might be a good solution.
title

SIMULATED ANNEALING

The intuition behind Hill Climbing was developed from the metaphor of climbing up the graph of a function to find its peak. There is a fundamental problem in the implementation of the algorithm however. To find the highest hill, we take one step at a time, always uphill, hoping to find the highest point, but if we are unlucky to start from the shoulder of the second-highest hill, there is no way we can find the highest one. The algorithm will always converge to the local optimum. Hill Climbing is also bad at dealing with functions that flatline in certain regions. If all neighboring states have the same value, we cannot find the global optimum using this algorithm.

Let's now look at an algorithm that can deal with these situations.
Simulated Annealing is quite similar to Hill Climbing, but instead of picking the *best* move every iteration, it picks a *random* move. If this random move brings us closer to the global optimum, it will be accepted, but if it doesn't, the algorithm may accept or reject the move based on a probability dictated by the *temperature*. When the `temperature` is high, the algorithm is more likely to accept a random move even if it is bad. At low temperatures, only good moves are accepted, with the occasional exception. This allows exploration of the state space and prevents the algorithm from getting stuck at the local optimum.

```
In [63]: psource(simulated_annealing)
```

```
def simulated_annealing(problem, schedule=exp_schedule()):
    """[Figure 4.5] CAUTION: This differs from the pseudocode as it
    returns a state instead of a Node."""
    current = Node(problem.initial)
    for t in range(sys.maxsize):
        T = schedule(t)
        if T == 0:
            return current.state
        neighbors = current.expand(problem)
        if not neighbors:
            return current.state
        next_choice = random.choice(neighbors)
        delta_e = problem.value(next_choice.state) - problem.value(current.state)
        if delta_e > 0 or probability(np.exp(delta_e / T)):
            current = next_choice
```

The temperature is gradually decreased over the course of the iteration. This is done by a scheduling routine. The current implementation uses exponential decay of temperature, but we can use a different scheduling routine instead.

In [64]:
```
psource(exp_schedule)
```

```
def exp_schedule(k=20, lam=0.005, limit=100):
    """One possible schedule function for simulated annealing"""
    return lambda t: (k * np.exp(-lam * t) if t < limit else 0)
```

Next, we'll define a peak-finding problem and try to solve it using Simulated Annealing. Let's define the grid and the initial state first.

In [65]:
```
initial = (0, 0)
grid = [[3, 7, 2, 8], [5, 2, 9, 1], [5, 3, 3, 1]]
```

We want to allow only four directions, namely `N`, `S`, `E` and `W`. Let's use the predefined `directions4` dictionary.

In [66]:
```
directions4
```

Out[66]:
```
{'W': (-1, 0), 'N': (0, 1), 'E': (1, 0), 'S': (0, -1)}
```

Define a problem with these parameters.

In [67]:
```
problem = PeakFindingProblem(initial, grid, directions4)
```

We'll run `simulated_annealing` a few times and store the solutions in a set.

In [68]:
```
solutions = {problem.value(simulated_annealing(problem)) for i in range(100)
```

In [69]:
```
max(solutions)
```

Out[69]:
```
9
```

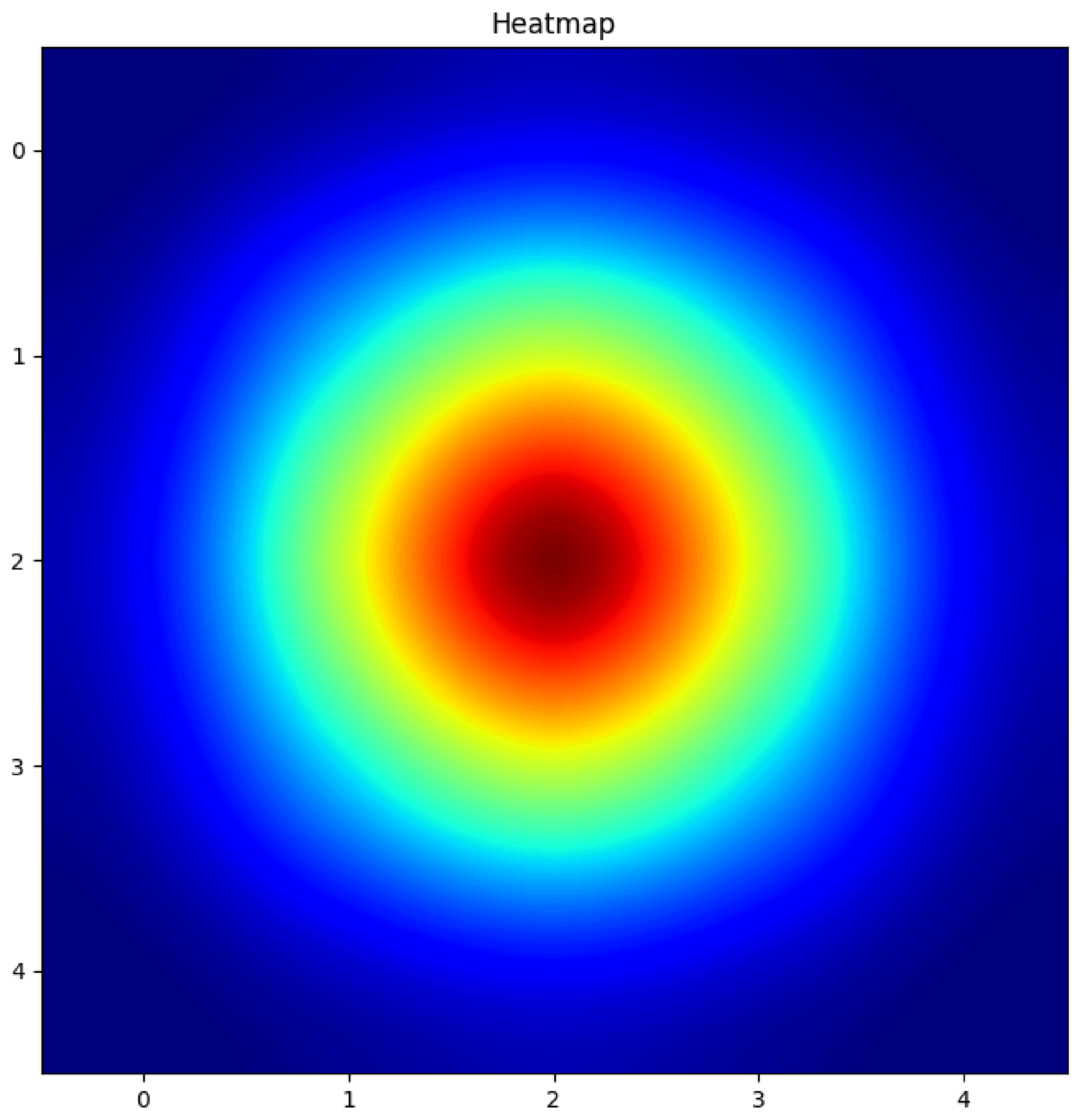
Hence, the maximum value is 9.

Let's find the peak of a two-dimensional gaussian distribution. We'll use the `gaussian_kernel` function from notebook.py to get the distribution.

In [70]:
```
grid = gaussian_kernel()
```

Let's use the `heatmap` function from notebook.py to plot this.

In [71]:
```
heatmap(grid, cmap='jet', interpolation='spline16')
```



Let's define the problem. This time, we will allow movement in eight directions as defined in `directions8`.

```
In [72]: directions8
```

```
Out[72]: {'W': (-1, 0),
          'N': (0, 1),
          'E': (1, 0),
          'S': (0, -1),
          'NW': (-1, 1),
          'NE': (1, 1),
          'SE': (1, -1),
          'SW': (-1, -1)}
```

We'll solve the problem just like we did last time.
Let's also time it.

```
In [73]: problem = PeakFindingProblem(initial, grid, directions8)
```

```
In [74]: %%timeit
         solutions = {problem.value(simulated_annealing(problem)) for i in range(100)
```

```
56.6 ms ± 285 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [75]: max(solutions)
```

```
Out[75]: 9
```

The peak is at 1.0 which is how gaussian distributions are defined.
This could also be solved by Hill Climbing as follows.

```
In [76]: %%timeit
         solution = problem.value(hill_climbing(problem))
```

```
21.9 µs ± 91.9 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
In [77]: solution = problem.value(hill_climbing(problem))
         solution
```

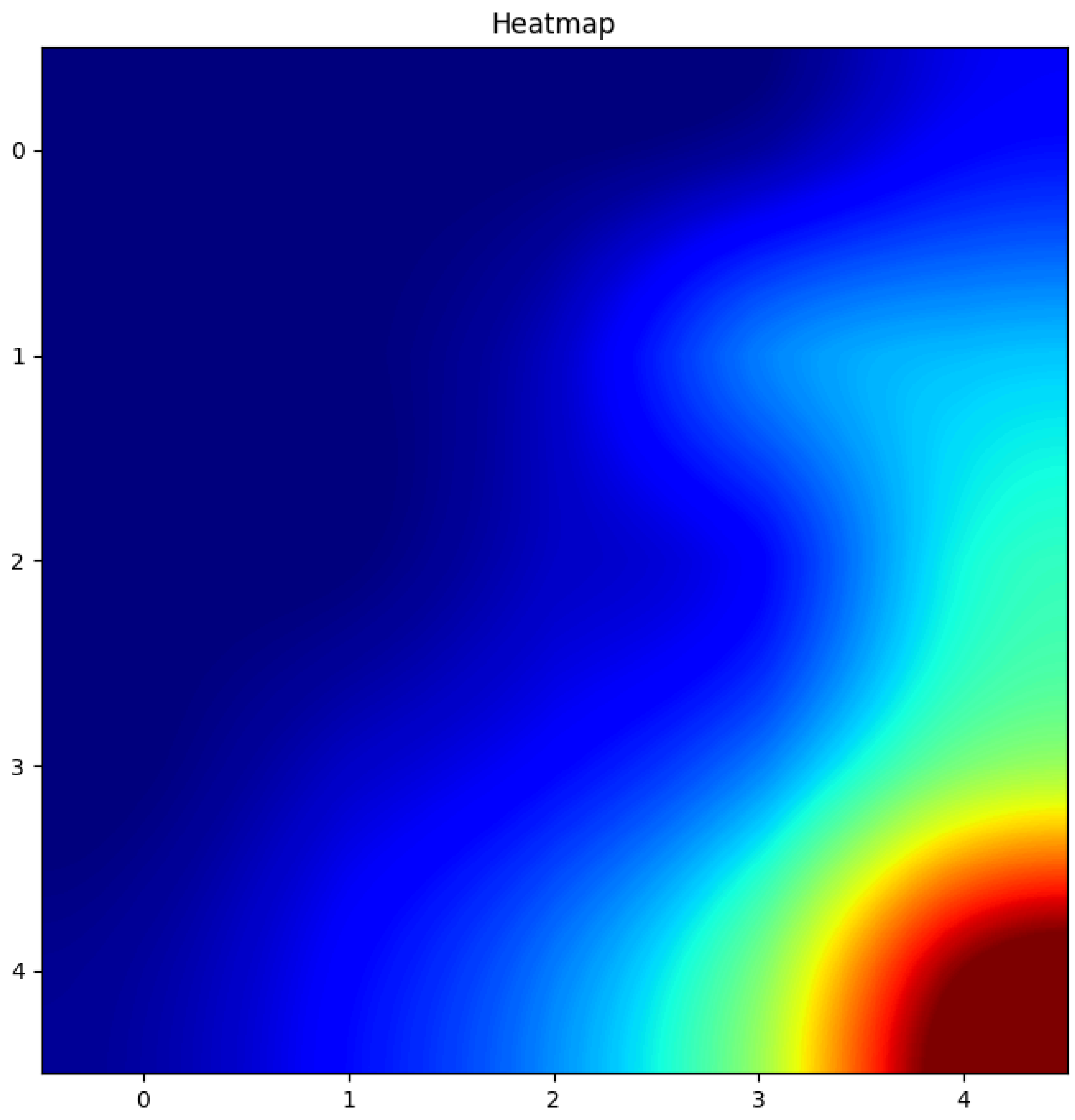
```
Out[77]: np.float64(1.0)
```

As you can see, Hill-Climbing is about 24 times faster than Simulated Annealing. (Notice that we ran Simulated Annealing for 100 iterations whereas we ran Hill Climbing only once.)
Simulated Annealing makes up for its tardiness by its ability to be applicable in a larger number of scenarios than Hill Climbing as illustrated by the example below.

Let's define a 2D surface as a matrix.

```
In [78]: grid = [[0, 0, 0, 1, 4],
                 [0, 0, 2, 8, 10],
                 [0, 0, 2, 4, 12],
                 [0, 2, 4, 8, 16],
                 [1, 4, 8, 16, 32]]
```

```
In [79]: heatmap(grid, cmap='jet', interpolation='spline16')
```



The peak value is 32 at the lower right corner.
The region at the upper left corner is planar.

Let's instantiate `PeakFindingProblem` one last time.

```
In [80]: problem = PeakFindingProblem(initial, grid, directions8)
```

Solution by Hill Climbing

```
In [81]: solution = problem.value(hill_climbing(problem))
```

```
In [82]: solution
```

```
Out[82]: 0
```

Solution by Simulated Annealing

In [83]:
```
solutions = {problem.value(simulated_annealing(problem)) for i in range(100)
max(solutions)
```

Out[83]:
```
32
```

Notice that even though both algorithms started at the same initial state, Hill Climbing could never escape from the planar region and gave a locally optimum solution of **0**, whereas Simulated Annealing could reach the peak at **32**.
A very similar situation arises when there are two peaks of different heights. One should carefully consider the possible search space before choosing the algorithm for the task.

GENETIC ALGORITHM

Genetic algorithms (or GA) are inspired by natural evolution and are particularly useful in optimization and search problems with large state spaces.

Given a problem, algorithms in the domain make use of a *population* of solutions (also called *states*), where each solution/state represents a feasible solution. At each iteration (often called *generation*), the population gets updated using methods inspired by biology and evolution, like *crossover*, *mutation* and *natural selection*.

Overview

A genetic algorithm works in the following way:

1. Initialize random population.
2. Calculate population fitness.
3. Select individuals for mating.
4. Mate selected individuals to produce new population.
    - Random chance to mutate individuals.
5. Repeat from step 2) until an individual is fit enough or the maximum number of iterations is reached.

Glossary

Before we continue, we will lay the basic terminology of the algorithm.

- Individual/State: A list of elements (called *genes*) that represent possible solutions.
- Population: The list of all the individuals/states.

- Gene pool: The alphabet of possible values for an individual's genes.
- Generation/Iteration: The number of times the population will be updated.
- Fitness: An individual's score, calculated by a function specific to the problem.

Crossover

Two individuals/states can "mate" and produce one child. This offspring bears characteristics from both of its parents. There are many ways we can implement this crossover. Here we will take a look at the most common ones. Most other methods are variations of those below.

- Point Crossover: The crossover occurs around one (or more) point. The parents get "split" at the chosen point or points and then get merged. In the example below we see two parents get split and merged at the 3rd digit, producing the following offspring after the crossover.

point crossover

- Uniform Crossover: This type of crossover chooses randomly the genes to get merged. Here the genes 1, 2 and 5 were chosen from the first parent, so the genes 3, 4 were added by the second parent.

uniform crossover

Mutation

When an offspring is produced, there is a chance it will mutate, having one (or more, depending on the implementation) of its genes altered.

For example, let's say the new individual to undergo mutation is "abcde". Randomly we pick to change its third gene to 'z'. The individual now becomes "abzde" and is added to the population.

Selection

At each iteration, the fittest individuals are picked randomly to mate and produce offsprings. We measure an individual's fitness with a *fitness function*. That function depends on the given problem and it is used to score an individual. Usually the higher the better.

The selection process is this:

1. Individuals are scored by the fitness function.

2. Individuals are picked randomly, according to their score (higher score means higher chance to get picked). Usually the formula to calculate the chance to pick an individual is the following (for population *P* and individual *i*):

$$chance(i) = \frac{fitness(i)}{\sum_{k \, in \, P} fitness(k)}$$

Implementation

Below we look over the implementation of the algorithm in the `search` module.

First the implementation of the main core of the algorithm:

```
In [84]: psource(genetic_algorithm)
```

```
def genetic_algorithm(population, fitness_fn, gene_pool=[0, 1], f_thres=None, ngen=100
0, pmut=0.1):
    """[Figure 4.8]"""
    for i in range(ngen):
        population = [mutate(recombine(*select(2, population, fitness_fn)), gene_pool, pmut)
                      for i in range(len(population))]

        fittest_individual = fitness_threshold(fitness_fn, f_thres, population)
        if fittest_individual:
            return fittest_individual

    return max(population, key=fitness_fn)
```

The algorithm takes the following input:

- `population` : The initial population.
- `fitness_fn` : The problem's fitness function.
- `gene_pool` : The gene pool of the states/individuals. By default 0 and 1.
- `f_thres` : The fitness threshold. If an individual reaches that score, iteration stops. By default 'None', which means the algorithm will not halt until the generations are ran.
- `ngen` : The number of iterations/generations.
- `pmut` : The probability of mutation.

The algorithm gives as output the state with the largest score.

For each generation, the algorithm updates the population. First it calculates the fitnesses of the individuals, then it selects the most fit ones and finally crosses them over to produce offsprings. There is a chance that the offspring will be mutated, given by `pmut`. If at the end of the generation an individual meets the fitness threshold, the algorithm halts and returns that individual.

The function of mating is accomplished by the method `recombine`:

```
In [85]: psource(recombine)
```

```
def recombine(x, y):
    n = len(x)
    c = random.randrange(0, n)
    return x[:c] + y[c:]
```

The method picks at random a point and merges the parents (`x` and `y`) around it.

The mutation is done in the method `mutate`:

```
In [86]: psource(mutate)
```

```
def mutate(x, gene_pool, pmut):
    if random.uniform(0, 1) >= pmut:
        return x

    n = len(x)
    g = len(gene_pool)
    c = random.randrange(0, n)
    r = random.randrange(0, g)

    new_gene = gene_pool[r]
    return x[:c] + [new_gene] + x[c + 1:]
```

We pick a gene in `x` to mutate and a gene from the gene pool to replace it with.

To help initializing the population we have the helper function `init_population`:

```
In [87]: psource(init_population)
```

```
def init_population(pop_number, gene_pool, state_length):
    """Initializes population for genetic algorithm
    pop_number  :  Number of individuals in population
    gene_pool   :  List of possible values for individuals
    state_length:  The length of each individual"""
    g = len(gene_pool)
    population = []
    for i in range(pop_number):
        new_individual = [gene_pool[random.randrange(0, g)] for j in range(state_length)]
        population.append(new_individual)

    return population
```

The function takes as input the number of individuals in the population, the gene pool and the length of each individual/state. It creates individuals with random genes and returns the population when done.

Explanation

Before we solve problems using the genetic algorithm, we will explain how to intuitively understand the algorithm using a trivial example.

Generating Phrases

In this problem, we use a genetic algorithm to generate a particular target phrase from a population of random strings. This is a classic example that helps build intuition about how to use this algorithm in other problems as well. Before we break the problem down, let us try to brute force the solution. Let us say that we want to generate the phrase "genetic algorithm". The phrase is 17 characters long. We can use any character from the 26 lowercase characters and the space character. To generate a random phrase of length 17, each space can be filled in 27 ways. So the total number of possible phrases is

$$27^{17} = 2153693963075557766310747$$

which is a massive number. If we wanted to generate the phrase "Genetic Algorithm", we would also have to include all the 26 uppercase characters into consideration thereby increasing the sample space from 27 characters to 53 characters and the total number of possible phrases then would be

$$53^{17} = 205442259656281392806087233013$$

If we wanted to include punctuations and numerals into the sample space, we would have further complicated an already impossible problem. Hence, brute forcing is not an option. Now we'll apply the genetic algorithm and see how it significantly reduces the search space. We essentially want to *evolve* our population of random strings so that they better approximate the target phrase as the number of generations increase. Genetic algorithms work on the principle of Darwinian Natural Selection according to which, there are three key concepts that need to be in place for evolution to happen. They are:

- **Heredity**: There must be a process in place by which children receive the properties of their parents.
  For this particular problem, two strings from the population will be chosen as parents and will be split at a random index and recombined as described in the `recombine` function to create a child. This child string will then be added to the new generation.

- **Variation**: There must be a variety of traits present in the population or a means with which to introduce variation.
  If there is no variation in the sample space, we might never reach the global optimum. To ensure that there is enough variation, we can initialize a large population, but this gets computationally expensive as the population gets larger. Hence, we often use another method called mutation. In this method, we randomly change one or more characters of some strings in the population based on a predefined probability value called the mutation rate or mutation probability as described in the `mutate` function. The mutation rate is usually kept quite low. A mutation rate of zero fails to introduce variation in the population and a high mutation rate (say 50%) is as good as a coin flip and the population fails to benefit from the previous recombinations. An optimum balance has to be maintained between population size and mutation rate so as to reduce the computational cost as well as have sufficient variation in the population.

- **Selection**: There must be some mechanism by which some members of the population have the opportunity to be parents and pass down their genetic information and some do not. This is typically referred to as "survival of the fittest".
  There has to be some way of determining which phrases in our population have a better chance of eventually evolving into the target phrase. This is done by introducing a fitness function that calculates how close the generated phrase is to the target phrase. The function will simply return a scalar value corresponding to the number of matching characters between the generated phrase and the target phrase.

Before solving the problem, we first need to define our target phrase.

In [88]:
```
target = 'Genetic Algorithm'
```

We then need to define our gene pool, i.e the elements which an individual from the population might comprise of. Here, the gene pool contains all uppercase and lowercase letters of the English alphabet and the space character.

In [89]:
```
# The ASCII values of uppercase characters ranges from 65 to 91
u_case = [chr(x) for x in range(65, 91)]
# The ASCII values of lowercase characters ranges from 97 to 123
l_case = [chr(x) for x in range(97, 123)]

gene_pool = []
gene_pool.extend(u_case) # adds the uppercase list to the gene pool
gene_pool.extend(l_case) # adds the lowercase list to the gene pool
gene_pool.append(' ')    # adds the space character to the gene pool
```

We now need to define the maximum size of each population. Larger populations have more variation but are computationally more expensive to run algorithms on.

In [90]:
```
max_population = 100
```

As our population is not very large, we can afford to keep a relatively large mutation rate.

In [91]:
```
mutation_rate = 0.07 # 7%
```

Great! Now, we need to define the most important metric for the genetic algorithm, i.e the fitness function. This will simply return the number of matching characters between the generated sample and the target phrase.

In [92]:
```
def fitness_fn(sample):
    # initialize fitness to 0
    fitness = 0
    for i in range(len(sample)):
        # increment fitness by 1 for every matching character
        if sample[i] == target[i]:
            fitness += 1
    return fitness
```

Before we run our genetic algorithm, we need to initialize a random population. We will use the `init_population` function to do this. We need to pass in the maximum population size, the gene pool and the length of each individual, which in this case will be the same as the length of the target phrase.

In [93]:
```
population = init_population(max_population, gene_pool, len(target))
```

We will now define how the individuals in the population should change as the number of generations increases. First, the `select` function will be run on the population to select *two* individuals with high fitness values. These will be the parents which will then be recombined using the `recombine` function to generate the child.

In [94]:
```
parents = select(2, population, fitness_fn)
```

In [95]:
```
# The recombine function takes two parents as arguments, so we need to unpac
child = recombine(*parents)
```

Next, we need to apply a mutation according to the mutation rate. We call the `mutate` function on the child with the gene pool and mutation rate as the additional arguments.

In [96]:
```
child = mutate(child, gene_pool, mutation_rate)
```

The above lines can be condensed into

`child = mutate(recombine(*select(2, population, fitness_fn)), gene_pool, mutation_rate)`

And, we need to do this `for` every individual in the current population to generate the new population.

In [97]:
```
population = [mutate(recombine(*select(2, population, fitness_fn)), gene_poo
```

The individual with the highest fitness can then be found using the `max` function.

In [98]:
```
current_best = max(population, key=fitness_fn)
```

Let's print this out

In [99]:
```
print(current_best)
```

```
['N', 'D', 'K', 'B', 'P', 'y', 'c', 'k', 'R', 'l', 'A', 'M', 'Q', 'D', 'w',
'h', 'z']
```

We see that this is a list of characters. This can be converted to a string using the join function

In [100…
```
current_best_string = ''.join(current_best)
print(current_best_string)
```

```
NDKBPyckRlAMQDwhz
```

We now need to define the conditions to terminate the algorithm. This can happen in two ways

1. Termination after a predefined number of generations
2. Termination when the fitness of the best individual of the current generation reaches a predefined threshold value.

We define these variables below

In [101…
```
ngen = 1200 # maximum number of generations
# we set the threshold fitness equal to the length of the target phrase
# i.e the algorithm only terminates whne it has got all the characters corre
# or it has completed 'ngen' number of generations
f_thres = len(target)
```

To generate `ngen` number of generations, we run a `for` loop `ngen` number of times. After each generation, we calculate the fitness of the best individual of the generation and compare it to the value of `f_thres` using the `fitness_threshold` function. After every generation, we print out the best individual of the generation and the corresponding fitness value. Lets now write a function to do this.

In [102…
```
def genetic_algorithm_stepwise(population, fitness_fn, gene_pool=[0, 1], f_t
    for generation in range(ngen):
        population = [mutate(recombine(*select(2, population, fitness_fn)),
        # stores the individual genome with the highest fitness in the curre
        current_best = ''.join(max(population, key=fitness_fn))
        print(f'Current best: {current_best}\t\tGeneration: {str(generation)

        # compare the fitness of the current best individual to f_thres
        fittest_individual = fitness_threshold(fitness_fn, f_thres, populati

        # if fitness is greater than or equal to f_thres, we terminate the a
        if fittest_individual:
            return fittest_individual, generation
    return max(population, key=fitness_fn) , generation
```

The function defined above is essentially the same as the one defined in `search.py` with the added functionality of printing out the data of each generation.

In [103…
```
psource(genetic_algorithm)
```

```
def genetic_algorithm(population, fitness_fn, gene_pool=[0, 1], f_thres=None, ngen=100
0, pmut=0.1):
    """[Figure 4.8]"""
    for i in range(ngen):
        population = [mutate(recombine(*select(2, population, fitness_fn)), gene_pool, pmut)
                      for i in range(len(population))]

        fittest_individual = fitness_threshold(fitness_fn, f_thres, population)
        if fittest_individual:
            return fittest_individual

    return max(population, key=fitness_fn)
```

We have defined all the required functions and variables. Let's now create a new population and test the function we wrote above.

In [104…
```
population = init_population(max_population, gene_pool, len(target))
solution, generations = genetic_algorithm_stepwise(population, fitness_fn, g
Current best: vjNefMPhRATNZYYhm         Generation: 0          Fitness: 3
Current best: LenISieEFoYqkuqhm         Generation: 1          Fitness: 5
Current best: LenISieEFoNoJkBZm         Generation: 2          Fitness: 5
Current best: LeneeNUIAMhNZYYhm         Generation: 3          Fitness: 6
Current best: LenefMihXReorkbhm         Generation: 4          Fitness: 7
Current best: LenefMihRATNrkbhm         Generation: 5          Fitness: 6
Current best: LeneeNUwXKeorkYhm         Generation: 6          Fitness: 7
Current best: LeneeNUwXKeorkYhm         Generation: 7          Fitness: 7
Current best: LenOuMCIAReorkbhm         Generation: 8          Fitness: 7
Current best: LeneuMXIAReorkbhm         Generation: 9          Fitness: 8
Current best: LeneuiGIxuNorkbhm         Generation: 10         Fitness: 8
Current best: LeneuiihXReorkbhm         Generation: 11         Fitness: 8
Current best: LeneuiGIxuNorkbhm         Generation: 12         Fitness: 8
Current best: LeneuiGIxuNorkYhm         Generation: 13         Fitness: 8
Current best: LenefiUwXuNorkbhm         Generation: 14         Fitness: 8
Current best: LenefiGIxuNorkbhm         Generation: 15         Fitness: 8
Current best: LenefiG XRNorkbhm         Generation: 16         Fitness: 9
Current best: Lene iN XReorkYhm         Generation: 17         Fitness: 9
Current best: Lene iN XReorkYhm         Generation: 18         Fitness: 9
Current best: Lene iN XRNorkbhm         Generation: 19         Fitness: 9
Current best: Lene iN XRNorkbhm         Generation: 20         Fitness: 9
Current best: LeneuiN iRNorkbhm         Generation: 21         Fitness: 9
Current best: LenefiU XuNorkbhm         Generation: 22         Fitness: 9
Current best: Lene iN XRNorkbhm         Generation: 23         Fitness: 9
Current best: Lene iN XReorkbhm         Generation: 24         Fitness: 9
Current best: Lene iN XReorkbhm         Generation: 25         Fitness: 9
Current best: LenefiG XReorkYhm         Generation: 26         Fitness: 9
Current best: Lene iN XReorkbhm         Generation: 27         Fitness: 9
Current best: LeneuiN xReorkYhm         Generation: 28         Fitness: 9
Current best: Lene iN XyeorkYhm         Generation: 29         Fitness: 9
Current best: Lene iN XRNorkbhm         Generation: 30         Fitness: 9
Current best: LeneuiG XRNorkbhm         Generation: 31         Fitness: 9
Current best: LenefiG XReorkYhm         Generation: 32         Fitness: 9
Current best: Lene iN XKeorkbhm         Generation: 33         Fitness: 9
Current best: LenefiG xieorkYhm         Generation: 34         Fitness: 9
Current best: Lene iF XKeorUbhm         Generation: 35         Fitness: 9
Current best: LeneuiN XKeorkbhm         Generation: 36         Fitness: 9
Current best: LenefiF XKeorUbhm         Generation: 37         Fitness: 9
Current best: LeneuiG xKeorkYhm         Generation: 38         Fitness: 9
Current best: LeneuiN XReorYYhm         Generation: 39         Fitness: 9
Current best: LenefiG xueorpbhm         Generation: 40         Fitness: 9
Current best: LeneuiN xueorkYhm         Generation: 41         Fitness: 9
Current best: LenefiG xRNorkYhm         Generation: 42         Fitness: 9
Current best: LenefiG xReorkbhm         Generation: 43         Fitness: 9
Current best: LeneuiN xRNorkYhm         Generation: 44         Fitness: 9
Current best: LeneuiN XuNorkYhm         Generation: 45         Fitness: 9
Current best: LenefiN xRNorkYhm         Generation: 46         Fitness: 9
Current best: Lene in xyeorkYhm         Generation: 47         Fitness: 9
Current best: LenefiG xyGorkYhm         Generation: 48         Fitness: 9
Current best: LeneziG xReorkYhm         Generation: 49         Fitness: 9
Current best: Lene in xRNorkYhm         Generation: 50         Fitness: 9
Current best: LeneuiG XRNorkYhm         Generation: 51         Fitness: 9
Current best: LeneuiN XRNorkYhm         Generation: 52         Fitness: 9
Current best: LeneuiN xReorkYhm         Generation: 53         Fitness: 9
Current best: LeneuiN xneorkYhm        Generation: 54          Fitness: 9
Current best: LenefiM XuNorkYhm        Generation: 55          Fitness: 9
Current best: Leneuin xReorkbhm        Generation: 56          Fitness: 9
Current best: Leneuic xReorkLhm        Generation: 57          Fitness: 10
Current best: LeneziN xReorkYhm        Generation: 58          Fitness: 9
Current best: LeneLin xRNorkYhm        Generation: 59          Fitness: 9
Current best: Lenefia xyeorkYhm        Generation: 60          Fitness: 9
Current best: LeneLiF xReorkYhm        Generation: 61          Fitness: 9
Current best: LeneziF xleorkYhm        Generation: 62          Fitness: 10
Current best: LenefiN xleorkYhm        Generation: 63          Fitness: 10
Current best: LeneziF xleorkYhm        Generation: 64          Fitness: 10
Current best: LenefiN xleorkYhm        Generation: 65          Fitness: 10
Current best: LeneoiN xlNorkbhm        Generation: 66          Fitness: 10
Current best: LeneziF xleorkYhm        Generation: 67          Fitness: 10
Current best: LeneMiF xleorkYhm        Generation: 68          Fitness: 10
Current best: LeneziF xlNorkbhm        Generation: 69          Fitness: 10
Current best: LeneziF xlNorkbhm        Generation: 70          Fitness: 10
Current best: LeneziF xlNorkbhm        Generation: 71          Fitness: 10
Current best: LeneziM xKNorkYhm        Generation: 72          Fitness: 9
Current best: LeneziM xReorkYhm        Generation: 73          Fitness: 9
Current best: Lenezil xyeorkYhm        Generation: 74          Fitness: 9
Current best: LenefiN xRGorkbhm        Generation: 75          Fitness: 9
Current best: Lenezil xKeorkYhm        Generation: 76          Fitness: 9
Current best: LeneRiN xyeorkbhm        Generation: 77          Fitness: 9
Current best: YenefiN QReorkbhm        Generation: 78          Fitness: 9
Current best: LenefiN xRGorkbhm        Generation: 79          Fitness: 9
Current best: LeneRiN xyeorkbhm        Generation: 80          Fitness: 9
Current best: LeneuiG xReoribhm        Generation: 81          Fitness: 10
Current best: LeneRiN xRGoribhm        Generation: 82          Fitness: 10
Current best: LeneRiN xRGoribhm        Generation: 83          Fitness: 10
Current best: meneRiN xRGoribhm        Generation: 84          Fitness: 10
Current best: LeneRiN xRGoribhm        Generation: 85          Fitness: 10
Current best: LeneRiN xRGoribhm        Generation: 86          Fitness: 10
Current best: LeneRiN xRGoribhm        Generation: 87          Fitness: 10
Current best: LeneRiN xreoribhm        Generation: 88          Fitness: 10
Current best: meneziN xkeoriYhm        Generation: 89          Fitness: 10
Current best: LenefiN xBeoribhm        Generation: 90          Fitness: 10
Current best: meneziN xkeorkbhm        Generation: 91          Fitness: 9
Current best: LenefiF xkGorkbhm        Generation: 92          Fitness: 9
Current best: meneziG xreorkbhm        Generation: 93          Fitness: 9
Current best: LeneziN xreorkbhm        Generation: 94          Fitness: 9
Current best: LeneziN xReorkbhm        Generation: 95          Fitness: 9
Current best: LeneziG xreorkbhm        Generation: 96          Fitness: 9
Current best: beneziG xReorkXhm        Generation: 97          Fitness: 9
Current best: beneziG xreorkbhm        Generation: 98          Fitness: 9
Current best: beneziG xreorkbhm        Generation: 99          Fitness: 9
Current best: KeneziN xreorkbhm        Generation: 100         Fitness: 9
Current best: KeneyiG xkeorUYhm        Generation: 101         Fitness: 9
Current best: LeneziG xReorkAhm        Generation: 102         Fitness: 9
Current best: LeneziN xReorkXhm        Generation: 103        Fitness: 9
Current best: KeneziF xReorkbhm        Generation: 104        Fitness: 9
Current best: meneziN xkeorkAhm        Generation: 105        Fitness: 9
Current best: meneRiG xreorkbhm        Generation: 106        Fitness: 9
Current best: meneRiG xreorkXhm        Generation: 107        Fitness: 9
Current best: LeneziG xreorkYhm        Generation: 108        Fitness: 9
Current best: meneniN xMeorEXhm        Generation: 109        Fitness: 9
Current best: meneziG xreorkbhm        Generation: 110        Fitness: 9
Current best: meneRiG xreorkbhm        Generation: 111        Fitness: 9
Current best: meneRiG xReorkXhm        Generation: 112        Fitness: 9
Current best: meneniN xreorkXhm        Generation: 113        Fitness: 9
Current best: LeneziG xreorkXhm        Generation: 114        Fitness: 9
Current best: meneziN xreorEAhm        Generation: 115        Fitness: 9
Current best: meneziN Jreorkbhm        Generation: 116        Fitness: 9
Current best: ieneniG xMeorEXhm        Generation: 117        Fitness: 9
Current best: meneRiG xreorkXhm        Generation: 118        Fitness: 9
Current best: meneniF xReorkbhm        Generation: 119        Fitness: 9
Current best: meneziF xreorkXhm        Generation: 120        Fitness: 9
Current best: meneziG xReorEbhm        Generation: 121        Fitness: 9
Current best: meneRiG xreorkAhm        Generation: 122        Fitness: 9
Current best: jeneziF xrWorEXhm        Generation: 123        Fitness: 9
Current best: meneziN xreorkbhm        Generation: 124        Fitness: 9
Current best: meneziF xKeorkbhm        Generation: 125        Fitness: 9
Current best: meneziN xveorEXhm        Generation: 126        Fitness: 9
Current best: meneziF xReorkbhm        Generation: 127        Fitness: 9
Current best: meneRiN xkgorkbhm        Generation: 128        Fitness: 10
Current best: meneRiN xkgorkbhm        Generation: 129        Fitness: 10
Current best: meneRiN xkgorkXhm        Generation: 130        Fitness: 10
Current best: jeneziG xkgorEXhm        Generation: 131        Fitness: 10
Current best: meneziG xkgorkbhm        Generation: 132        Fitness: 10
Current best: ueneziG xkgorkXhm        Generation: 133        Fitness: 10
Current best: ueneziG xkgorkXhm        Generation: 134        Fitness: 10
Current best: jeneziG xkgorkthm        Generation: 135        Fitness: 11
Current best: meneziN xKeorkthm        Generation: 136        Fitness: 10
Current best: KeneziN xKeorkthm        Generation: 137        Fitness: 10
Current best: KeneziG xreorkthm        Generation: 138        Fitness: 10
Current best: meneziN xreorkthm        Generation: 139        Fitness: 10
Current best: meneziN xreorkthm        Generation: 140        Fitness: 10
Current best: TeneziG xreorkthm        Generation: 141        Fitness: 10
Current best: meneziG xkLorkthm        Generation: 142        Fitness: 10
Current best: meneziG xreorkthm        Generation: 143        Fitness: 10
Current best: meneziN xreorkthm        Generation: 144        Fitness: 10
Current best: TeneziN xreorkthm        Generation: 145        Fitness: 10
Current best: meneziG xFeorkthm        Generation: 146        Fitness: 10
Current best: meneziN xkeorvthm        Generation: 147        Fitness: 10
Current best: meneziG xFeorkthm        Generation: 148        Fitness: 10
Current best: meneziN xreorkthm        Generation: 149        Fitness: 10
Current best: meneziG xFeorkthm        Generation: 150        Fitness: 10
Current best: XeneziG xleorXXhm        Generation: 151        Fitness: 10
Current best: meneziN xreorkthm        Generation: 152        Fitness: 10
Current best: meneziG xleorEYhm        Generation: 153        Fitness: 10
Current best: meneziG rleorEYhm        Generation: 154        Fitness: 10
Current best: XeneziG xleorEbhm        Generation: 155        Fitness: 10
Current best: XeneziF rleorEYhm        Generation: 156        Fitness: 10
Current best: meneziN xreorGthm        Generation: 157        Fitness: 10
Current best: meneziN xreorGthm        Generation: 158        Fitness: 10
Current best: meneziN xreorGthm        Generation: 159        Fitness: 10
Current best: ceneziF rleorEYhm        Generation: 160        Fitness: 10
Current best: XeneziG rleorkXhm        Generation: 161        Fitness: 10
Current best: meneziN xreorkthm        Generation: 162        Fitness: 10
Current best: XeneziN xreorEthm        Generation: 163        Fitness: 10
Current best: XeneziF xreorkthm        Generation: 164        Fitness: 10
Current best: geneziG xreorkthm        Generation: 165        Fitness: 10
Current best: meneziN rleorkthm        Generation: 166        Fitness: 11
Current best: XeneziN xreorkthm        Generation: 167        Fitness: 10
Current best: geneziN xreorGthm        Generation: 168        Fitness: 10
Current best: XeneziF xreorGthm        Generation: 169        Fitness: 10
Current best: meneziN xreorGthm        Generation: 170        Fitness: 10
Current best: meneziG wHeorEthm        Generation: 171        Fitness: 10
Current best: meneziN xreorkthm        Generation: 172        Fitness: 10
Current best: meneziN xHeorEthm        Generation: 173        Fitness: 10
Current best: XeneziN xreorkthm        Generation: 174        Fitness: 10
Current best: meneziG xreorEthm        Generation: 175        Fitness: 10
Current best: meneziN xreorEthm        Generation: 176        Fitness: 10
Current best: feneziN xreorEthm        Generation: 177        Fitness: 10
Current best: menemiN PreorGthm        Generation: 178        Fitness: 10
Current best: meneziN xreoruthm        Generation: 179        Fitness: 10
Current best: XeneziN xreorEthm        Generation: 180        Fitness: 10
Current best: XeneziN xreorEthm        Generation: 181        Fitness: 10
Current best: geneziN xmeorkthm        Generation: 182        Fitness: 10
Current best: geneziN xreorEthm        Generation: 183        Fitness: 10
Current best: XeneziN xHeorGthm        Generation: 184        Fitness: 10
Current best: meneziN xreorkthm        Generation: 185        Fitness: 10
Current best: ceneziF xrhorEthm        Generation: 186        Fitness: 10
Current best: XeneziN xreorkthm        Generation: 187        Fitness: 10
Current best: meneziN xreorkthm        Generation: 188        Fitness: 10
Current best: geneziN xreorkthm        Generation: 189        Fitness: 10
Current best: XeneziN xreorkthm        Generation: 190        Fitness: 10
Current best: geneziN xreorkthm        Generation: 191        Fitness: 10
Current best: ceneziN xreorkthm        Generation: 192        Fitness: 10
Current best: XenegiN xmeorkthm        Generation: 193        Fitness: 10
Current best: geneziN xHeorkthm        Generation: 194        Fitness: 10
Current best: genegiN xreorkthm        Generation: 195         Fitness: 10
Current best: XenemiN xreorkthm        Generation: 196         Fitness: 10
Current best: XenegiN xreorsthm        Generation: 197         Fitness: 10
Current best: meneziN xWeorkthm        Generation: 198         Fitness: 10
Current best: geneziN wrgorkthm        Generation: 199         Fitness: 11
Current best: XeneziN wreorEthm        Generation: 200         Fitness: 10
Current best: meneziN wreorEthm        Generation: 201         Fitness: 10
Current best: meneziF xHeorkthm        Generation: 202         Fitness: 10
Current best: XeneziF xmeorEthm        Generation: 203         Fitness: 10
Current best: meneziN xreorkthm        Generation: 204         Fitness: 10
Current best: XeneziN xHeorkthm        Generation: 205         Fitness: 10
Current best: meneziN xreorkthm        Generation: 206         Fitness: 10
Current best: meneziN xmeorkthm        Generation: 207         Fitness: 10
Current best: XeneziN xreorkthm        Generation: 208         Fitness: 10
Current best: XeneziF kreorkthm        Generation: 209         Fitness: 10
Current best: meneziN xreorkthm        Generation: 210         Fitness: 10
Current best: yeneziF xreorkthm        Generation: 211         Fitness: 10
Current best: meneziN xmeorkthm        Generation: 212         Fitness: 10
Current best: XenexiQ xmeorkthm        Generation: 213         Fitness: 10
Current best: meneziF xreorkthm        Generation: 214         Fitness: 10
Current best: XeneziF xmeorkthm        Generation: 215         Fitness: 10
Current best: menexiN xmgorwthm        Generation: 216         Fitness: 11
Current best: menexiN xmgorwthm        Generation: 217         Fitness: 11
Current best: menexiN xmgorwthm        Generation: 218         Fitness: 11
Current best: menexiN xHeorkthm        Generation: 219         Fitness: 10
Current best: meneziN xreorkthm        Generation: 220         Fitness: 10
Current best: meneziN xHeorEthm        Generation: 221         Fitness: 10
Current best: XeneziN xHeorkthm        Generation: 222         Fitness: 10
Current best: meneziN xreorEthm        Generation: 223         Fitness: 10
Current best: meneziN xWeorYthm        Generation: 224         Fitness: 10
Current best: menexiN xreorEthm        Generation: 225         Fitness: 10
Current best: XeneziF xHeorkthm        Generation: 226         Fitness: 10
Current best: XeneziN x eorEthm        Generation: 227         Fitness: 10
Current best: XeneziF xyeorkthm        Generation: 228         Fitness: 10
Current best: meneziN x eorEthm        Generation: 229         Fitness: 10
Current best: meneziN XHeorEthm        Generation: 230         Fitness: 10
Current best: XeneziN xHeorkthm        Generation: 231         Fitness: 10
Current best: meneziN xyeorkthm        Generation: 232         Fitness: 10
Current best: meneziN xHeorkthm        Generation: 233         Fitness: 10
Current best: meneziF xHeorathm        Generation: 234         Fitness: 10
Current best: XeneziN X jorkthm        Generation: 235         Fitness: 10
Current best: meneziN xyeorkthm        Generation: 236         Fitness: 10
Current best: meneziN xWeorEthm        Generation: 237         Fitness: 10
Current best: meneziN xleorkthm        Generation: 238         Fitness: 11
Current best: XeneziN XmeorEthm        Generation: 239         Fitness: 10
Current best: meneziN XHeorkthm        Generation: 240         Fitness: 10
Current best: meneziN XWjorkthm        Generation: 241         Fitness: 10
Current best: meneziN xWeorkthm         Generation: 242         Fitness: 10
Current best: XeneziF XWeorEthm         Generation: 243         Fitness: 10
Current best: XeneziN xCeorkthm         Generation: 244         Fitness: 10
Current best: meneziN XWeorkthm         Generation: 245         Fitness: 10
Current best: meneziN xmeorethm         Generation: 246         Fitness: 10
Current best: GeneziN xmeorEthm         Generation: 247         Fitness: 11
Current best: GeneziN x eorEthm         Generation: 248         Fitness: 11
Current best: GeneziW XWeorEthm         Generation: 249         Fitness: 11
Current best: GeneziF xmeorEthm         Generation: 250         Fitness: 11
Current best: GeneziN x eorkthm         Generation: 251         Fitness: 11
Current best: GeneziN x eorEthm         Generation: 252         Fitness: 11
Current best: GeneziN x eorEthm         Generation: 253         Fitness: 11
Current best: GeneziN XWeorEthm         Generation: 254         Fitness: 11
Current best: GeneziN xmeorkthm         Generation: 255         Fitness: 11
Current best: GeneziK S eorkthm         Generation: 256         Fitness: 11
Current best: GeneziW XWeorEthm         Generation: 257         Fitness: 11
Current best: GeneziF xmeorEthm         Generation: 258         Fitness: 11
Current best: GeneziW XmeorEthm         Generation: 259         Fitness: 11
Current best: GeneziN x eorEthm         Generation: 260         Fitness: 11
Current best: GeneziN x eorkthm         Generation: 261         Fitness: 11
Current best: GeneziW ymeorEthm         Generation: 262         Fitness: 11
Current best: GenetiN pmeorkthm         Generation: 263         Fitness: 12
Current best: GenetiN xmeorEthm         Generation: 264         Fitness: 12
Current best: GenetiN xmeorEthm         Generation: 265         Fitness: 12
Current best: GenetiN xmeorEthm         Generation: 266         Fitness: 12
Current best: GenetiN x forkthm         Generation: 267         Fitness: 12
Current best: GenetiN SmeorEthm         Generation: 268         Fitness: 12
Current best: GenetiN SmeorEthm         Generation: 269         Fitness: 12
Current best: GenetiN S eorEthm         Generation: 270         Fitness: 12
Current best: GenetiN SmeorEthm         Generation: 271         Fitness: 12
Current best: GenetiN SmeorEthm         Generation: 272         Fitness: 12
Current best: GeneziN pmeorkthm         Generation: 273         Fitness: 11
Current best: GeneziN Smeorkthm         Generation: 274         Fitness: 11
Current best: menetiW pmeorkthm         Generation: 275         Fitness: 11
Current best: GeneziN Umeorkthm         Generation: 276         Fitness: 11
Current best: GeneziN x eorEthm         Generation: 277         Fitness: 11
Current best: GenetiW U forkthm         Generation: 278         Fitness: 12
Current best: GeneziN XWeorEthm         Generation: 279         Fitness: 11
Current best: GeneziN x eorEthm         Generation: 280         Fitness: 11
Current best: GenutiW xmeorkthm         Generation: 281         Fitness: 11
Current best: menetiW xmeorkthm         Generation: 282         Fitness: 11
Current best: GenetiW U eorEthm         Generation: 283         Fitness: 12
Current best: GenetiW U eorEthm         Generation: 284         Fitness: 12
Current best: GenetiW r eorEthm         Generation: 285         Fitness: 12
Current best: GenetiW p forEthm         Generation: 286         Fitness: 12
Current best: GenetiW p forEthm         Generation: 287         Fitness: 12
Current best: GenetiW x eorkthm         Generation: 288         Fitness: 12
Current best: GenetiW x forzthm         Generation: 289         Fitness: 12
Current best: GenetiN xmeorkthm         Generation: 290         Fitness: 12
Current best: GenetiW xzeorEthm         Generation: 291         Fitness: 12
Current best: Genetil xmeorkthm         Generation: 292         Fitness: 12
Current best: GenetiN CmXorEthm         Generation: 293         Fitness: 12
Current best: GenetiW pzeorEthm         Generation: 294         Fitness: 12
Current best: GenetiW x XorEthm         Generation: 295         Fitness: 12
Current best: GenetiN S forEthm         Generation: 296         Fitness: 12
Current best: GenetiN S eorkthm         Generation: 297         Fitness: 12
Current best: GenetiN S forEthm         Generation: 298         Fitness: 12
Current best: GenetiN p eorEthm         Generation: 299         Fitness: 12
Current best: GenetiN S forkthm         Generation: 300         Fitness: 12
Current best: GenetiN p forEthm         Generation: 301         Fitness: 12
Current best: GenetiW xmeorEthm         Generation: 302         Fitness: 12
Current best: GenetiN p LorEthm         Generation: 303         Fitness: 12
Current best: Genetik p Lorkthm         Generation: 304         Fitness: 12
Current best: GenetiN p eortthm         Generation: 305         Fitness: 12
Current best: Genetik p LorEthm         Generation: 306         Fitness: 12
Current best: GenetiN p Lortthm         Generation: 307         Fitness: 12
Current best: Genetik p forEthm         Generation: 308         Fitness: 12
Current best: GenetiN S CorEthm         Generation: 309         Fitness: 12
Current best: Genetic p eorEthm         Generation: 310         Fitness: 13
Current best: GenetiW S CorEthm         Generation: 311         Fitness: 12
Current best: Genetik p Lorkthm         Generation: 312         Fitness: 12
Current best: GenetiN xzLorkthm         Generation: 313         Fitness: 12
Current best: GenetiN p forkthm         Generation: 314         Fitness: 12
Current best: Genetik pzeorkthm         Generation: 315         Fitness: 12
Current best: GenetiW s eorEthm         Generation: 316         Fitness: 12
Current best: GenetiN X eorOthm         Generation: 317         Fitness: 12
Current best: GenetiN S eorEthm         Generation: 318         Fitness: 12
Current best: GenetiN p eorEthm         Generation: 319         Fitness: 12
Current best: GenetiN p eorEthm         Generation: 320         Fitness: 12
Current best: GenetiN p eorEthm         Generation: 321         Fitness: 12
Current best: GenetiN p norOthm         Generation: 322         Fitness: 12
Current best: GenetiN p forkthm         Generation: 323         Fitness: 12
Current best: Genetik X eorOthm         Generation: 324         Fitness: 12
Current best: Genetik X eorOthm         Generation: 325         Fitness: 12
Current best: GenetiN S eorkthm         Generation: 326         Fitness: 12
Current best: GenetiI p gorkthm         Generation: 327         Fitness: 13
Current best: GenetiI p gorkthm         Generation: 328         Fitness: 13
Current best: GenetiI p eorSthm         Generation: 329         Fitness: 12
Current best: GenetiI p eorOthm         Generation: 330         Fitness: 12
Current best: GenetiN p eorOthm         Generation: 331         Fitness: 12
Current best: GenetiN p forkthm         Generation: 332         Fitness: 12
Current best: GenetiI H eorOthm         Generation: 333         Fitness: 12
Current best: GenetiN p forEthm         Generation: 334         Fitness: 12
Current best: Genetik p eorEthm         Generation: 335         Fitness: 12
Current best: GenetiI p LorEthm         Generation: 336         Fitness: 12
Current best: GenetiW SlforSthm         Generation: 337         Fitness: 13
Current best: GenetiW SlforSthm         Generation: 338         Fitness: 13
Current best: Genetik p eorSthm         Generation: 339         Fitness: 12
Current best: GenetiN p LorOthm         Generation: 340         Fitness: 12
Current best: GenetiN p eorSthm         Generation: 341         Fitness: 12
Current best: GenetiN p eorSthm         Generation: 342         Fitness: 12
Current best: Genetik p forkthm         Generation: 343         Fitness: 12
Current best: GenetiW S forOthm         Generation: 344         Fitness: 12
Current best: GenetiW p forEthm         Generation: 345         Fitness: 12
Current best: GenetiN p LorEthm         Generation: 346         Fitness: 12
Current best: GenetiI p eorOthm         Generation: 347         Fitness: 12
Current best: GenetiW S LorEthm         Generation: 348         Fitness: 12
Current best: GenetiW p eor thm         Generation: 349         Fitness: 12
Current best: GenetiN p LorEthm         Generation: 350         Fitness: 12
Current best: GenetiI p LorOthm         Generation: 351         Fitness: 12
Current best: GenetiN p forEthm         Generation: 352         Fitness: 12
Current best: GenetiW p forDthm         Generation: 353         Fitness: 12
Current best: GenetiW p eorEthm         Generation: 354         Fitness: 12
Current best: Genetik p eorEthm         Generation: 355         Fitness: 12
Current best: GenetiN p eorEthm         Generation: 356         Fitness: 12
Current best: GenetiI p LorEthm         Generation: 357         Fitness: 12
Current best: GenetiI p porSthm         Generation: 358         Fitness: 12
Current best: GenetiW p LorEthm         Generation: 359         Fitness: 12
Current best: Genetik pzLorEthm         Generation: 360         Fitness: 12
Current best: GenetiW p LorEthm         Generation: 361         Fitness: 12
Current best: GenetiI p eorSthm         Generation: 362         Fitness: 12
Current best: GenetiI p eorSthm         Generation: 363         Fitness: 12
Current best: GenetiN p LorEthm         Generation: 364         Fitness: 12
Current best: GenetiN pgeorIthm         Generation: 365         Fitness: 12
Current best: GenetiN b LorEthm         Generation: 366         Fitness: 12
Current best: GenetiW p LorEthm         Generation: 367         Fitness: 12
Current best: Genetil p LorEthm         Generation: 368         Fitness: 12
Current best: GenetiN p eorIthm         Generation: 369         Fitness: 12
Current best: GenetiI p eorIthm         Generation: 370         Fitness: 12
Current best: GenetiW pbeorEthm         Generation: 371         Fitness: 12
Current best: GenetiN p eorIthm         Generation: 372         Fitness: 12
Current best: GenetiW pbeoLithm         Generation: 373         Fitness: 12
Current best: GenetiN y forEthm         Generation: 374         Fitness: 12
Current best: GenetiW p eorEthm         Generation: 375         Fitness: 12
Current best: Genetik p eorIthm         Generation: 376         Fitness: 12
Current best: GenetiW p eorSthm         Generation: 377         Fitness: 12
Current best: GenetiN pALorEthm         Generation: 378         Fitness: 12
Current best: Genetib p LorEthm         Generation: 379         Fitness: 12
Current best: GenetiW p LorEthm         Generation: 380         Fitness: 12
Current best: GenetiN p eorEthm         Generation: 381         Fitness: 12
Current best: GenetiN p eorIthm         Generation: 382         Fitness: 12
Current best: GenetiN b LorIthm         Generation: 383         Fitness: 12
Current best: GenetiN p LorIthm         Generation: 384         Fitness: 12
Current best: GenetiN p eorIthm         Generation: 385         Fitness: 12
Current best: GenetiW b LorIthm         Generation: 386         Fitness: 12
Current best: GenetiN p LorSthm         Generation: 387         Fitness: 12
Current best: GenetiN p eorIthm         Generation: 388         Fitness: 12
Current best: GenetiW p eorSthm         Generation: 389         Fitness: 12
Current best: GenetiW p LorEthm         Generation: 390         Fitness: 12
Current best: GenetiN p Lorpthm         Generation: 391         Fitness: 12
Current best: GenetiW p eorSthm         Generation: 392         Fitness: 12
Current best: GenetiW p LorSthm         Generation: 393         Fitness: 12
Current best: GenetiW p eorEthm         Generation: 394         Fitness: 12
Current best: GenetiN O eorSthm         Generation: 395         Fitness: 12
Current best: GenetiW p Lorpthm         Generation: 396         Fitness: 12
Current best: GenetiW pbLorSthm         Generation: 397         Fitness: 12
Current best: GenetiW p eorSthm         Generation: 398         Fitness: 12
Current best: GenetiN pbLorSthm         Generation: 399         Fitness: 12
Current best: GenetiN OvLorSthm         Generation: 400         Fitness: 12
Current best: GenetiW p LorEthm         Generation: 401         Fitness: 12
Current best: GenetiW MbeorSthm         Generation: 402         Fitness: 12
Current best: GenetiW pUJorSthm         Generation: 403         Fitness: 12
Current best: GenetiN pbLorSthm         Generation: 404         Fitness: 12
Current best: GenetiN pbLorSthm         Generation: 405         Fitness: 12
Current best: GenetiW OUJorAthm         Generation: 406         Fitness: 12
Current best: GenetiN pbLorEthm         Generation: 407         Fitness: 12
Current best: GenetiD p LorSthm         Generation: 408         Fitness: 12
Current best: GenetiN pbeorEthm         Generation: 409         Fitness: 12
Current best: GenetiW pbporSthm         Generation: 410         Fitness: 12
Current best: GenetiW O LorSthm         Generation: 411         Fitness: 12
Current best: GenetiW pUJorSthm         Generation: 412         Fitness: 12
Current best: GenetiW p LorEthm         Generation: 413         Fitness: 12
Current best: GenetiW pbJorEthm         Generation: 414         Fitness: 12
Current best: GenetiW p eorSthm         Generation: 415         Fitness: 12
Current best: GenetiD pbLorSthm         Generation: 416         Fitness: 12
Current best: GenetiD OUJorEthm         Generation: 417         Fitness: 12
Current best: GenetiW OvLorSthm         Generation: 418         Fitness: 12
Current best: GenetiW pbJorSthm         Generation: 419         Fitness: 12
Current best: GenetiN pbLorSthm         Generation: 420         Fitness: 12
Current best: GenetiW pUvorSthm         Generation: 421         Fitness: 12
Current best: GenetiW p JorEthm         Generation: 422         Fitness: 12
Current best: GenetiN jbeorSthm         Generation: 423         Fitness: 12
Current best: GenetiW ppJorEthm         Generation: 424         Fitness: 12
Current best: GenetiN pUJorSthm         Generation: 425         Fitness: 12
Current best: GenetiN pULorSthm        Generation: 426         Fitness: 12
Current best: GenetiD jbLorSthm        Generation: 427         Fitness: 12
Current best: GenetiD jbLorSthm        Generation: 428         Fitness: 12
Current best: GenetiN p LorSthm        Generation: 429         Fitness: 12
Current best: GenetiD jbLorSthm        Generation: 430         Fitness: 12
Current best: GenetiD pbLorSthm        Generation: 431         Fitness: 12
Current best: GenetiD jbLorSthm        Generation: 432         Fitness: 12
Current best: GenetiN pbeorEthm        Generation: 433         Fitness: 12
Current best: GenetiD p LorSthm        Generation: 434         Fitness: 12
Current best: GenetiW EbeorSthm        Generation: 435         Fitness: 12
Current best: GenetiD jbLorSthm        Generation: 436         Fitness: 12
Current best: GenetiD p LorSthm        Generation: 437         Fitness: 12
Current best: GenetiD p LorSthm        Generation: 438         Fitness: 12
Current best: GenetiN jbLorEthm        Generation: 439         Fitness: 12
Current best: GenetiW pbJorSthm        Generation: 440         Fitness: 12
Current best: GenetiN p Lorithm        Generation: 441         Fitness: 13
Current best: GenetiN p Lorithm        Generation: 442         Fitness: 13
Current best: GenetiN p Lorithm        Generation: 443         Fitness: 13
Current best: GenetiW p Lorithm        Generation: 444         Fitness: 13
Current best: GenetiN j Lorithm        Generation: 445         Fitness: 13
Current best: GenetiN pbLorithm        Generation: 446         Fitness: 13
Current best: GenetiW p Lorithm        Generation: 447         Fitness: 13
Current best: GenetiW p Lorithm        Generation: 448         Fitness: 13
Current best: GenetiW p Lorithm        Generation: 449         Fitness: 13
Current best: GenetiW pULorithm        Generation: 450         Fitness: 13
Current best: GenetiW pbLorithm        Generation: 451         Fitness: 13
Current best: GenetiW ebLorithm        Generation: 452         Fitness: 13
Current best: GenetiD p Lorithm        Generation: 453         Fitness: 13
Current best: GenetiD p Lorithm        Generation: 454         Fitness: 13
Current best: GenetiW pUBorithm        Generation: 455         Fitness: 13
Current best: GenetiD pQLorithm        Generation: 456         Fitness: 13
Current best: GenetiW pUBorithm        Generation: 457         Fitness: 13
Current best: GenetiW pbLorithm        Generation: 458         Fitness: 13
Current best: GenetiW p Lorithm        Generation: 459         Fitness: 13
Current best: GenetiD pbLorithm        Generation: 460         Fitness: 13
Current best: GenetiW pbLorithm        Generation: 461         Fitness: 13
Current best: GenetiW p xorithm        Generation: 462         Fitness: 13
Current best: Gynetic pbLorithm        Generation: 463         Fitness: 13
Current best: Genetic pbLorithm        Generation: 464         Fitness: 14
Current best: GenetiW pbJorithm        Generation: 465         Fitness: 13
Current best: GenetiD pbLorithm        Generation: 466         Fitness: 13
Current best: GenetiW pbLorithm        Generation: 467         Fitness: 13
Current best: GenetiD p Lorithm        Generation: 468         Fitness: 13
Current best: GenetiW TbLorithm        Generation: 469         Fitness: 13
Current best: GenetiW pbLorithm         Generation: 470         Fitness: 13
Current best: GenetiW pbLorithm         Generation: 471         Fitness: 13
Current best: GenetiW ebLorithm         Generation: 472         Fitness: 13
Current best: GenetiW ebLorithm         Generation: 473         Fitness: 13
Current best: GenetiW p Jorithm         Generation: 474         Fitness: 13
Current best: GenetiW xULorithm         Generation: 475         Fitness: 13
Current best: GenetiD ebLorithm         Generation: 476         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 477         Fitness: 13
Current best: GenetiW pbLorithm         Generation: 478         Fitness: 13
Current best: GenetiW xULorithm         Generation: 479         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 480         Fitness: 13
Current best: GenetiD PbLorithm         Generation: 481         Fitness: 13
Current best: GenetiD rbLorithm         Generation: 482         Fitness: 13
Current best: GenetiW pbLorithm         Generation: 483         Fitness: 13
Current best: GenetiD rUJorithm         Generation: 484         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 485         Fitness: 13
Current best: GenetiW pULorithm         Generation: 486         Fitness: 13
Current best: GenetiD pwLorithm         Generation: 487         Fitness: 13
Current best: GenetiW pULorithm         Generation: 488         Fitness: 13
Current best: GenetiD xULorithm         Generation: 489         Fitness: 13
Current best: GenetiD xULorithm         Generation: 490         Fitness: 13
Current best: GenetiW pULorithm         Generation: 491         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 492         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 493         Fitness: 13
Current best: Genetiw xULorithm         Generation: 494         Fitness: 13
Current best: GenetiW pULorithm         Generation: 495         Fitness: 13
Current best: GenetiD eULorithm         Generation: 496         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 497         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 498         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 499         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 500         Fitness: 13
Current best: Genetiw xVLorithm         Generation: 501         Fitness: 13
Current best: GenetiW pbLorithm         Generation: 502         Fitness: 13
Current best: Genetiw pbLorithm         Generation: 503         Fitness: 13
Current best: Genetiw pbLorithm         Generation: 504         Fitness: 13
Current best: GenetiW pbsorithm         Generation: 505         Fitness: 13
Current best: GenetiW pUCorithm         Generation: 506         Fitness: 13
Current best: GenetiD xULorithm         Generation: 507         Fitness: 13
Current best: GenetiD eULorithm         Generation: 508         Fitness: 13
Current best: GenetiD eULorithm         Generation: 509         Fitness: 13
Current best: GenetiW pbLorithm         Generation: 510         Fitness: 13
Current best: GenetiD pbLorithm         Generation: 511         Fitness: 13
Current best: GenetiW AbLorithm         Generation: 512         Fitness: 14
Current best: GenetiW AbLorithm         Generation: 513         Fitness: 14
Current best: GenetiW Absorithm         Generation: 514         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 515         Fitness: 14
Current best: GenetiD AbLorithm        Generation: 516         Fitness: 14
Current best: GenetiD AbLorithm        Generation: 517         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 518         Fitness: 14
Current best: GenetiD AbLorithm        Generation: 519         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 520         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 521         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 522         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 523         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 524         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 525         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 526         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 527         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 528         Fitness: 14
Current best: GenetiW Absorithm        Generation: 529         Fitness: 14
Current best: GenetiW Absorithm        Generation: 530         Fitness: 14
Current best: GenetiD AbLorithm        Generation: 531         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 532         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 533         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 534         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 535         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 536         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 537         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 538         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 539         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 540         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 541         Fitness: 14
Current best: GenetiA AbLorithm        Generation: 542         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 543         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 544         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 545         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 546         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 547         Fitness: 14
Current best: GenetiW AbLorithm        Generation: 548         Fitness: 14
Current best: GenetiW pbLorithm        Generation: 549         Fitness: 13
Current best: GenetiD qbsorithm        Generation: 550         Fitness: 13
Current best: GenetiD pbLorithm        Generation: 551         Fitness: 13
Current best: GenetiW pbsorithm        Generation: 552         Fitness: 13
Current best: GenetiD pbsorithm        Generation: 553         Fitness: 13
Current best: GenetiW pbsorithm        Generation: 554         Fitness: 13
Current best: GenetiW pbLorithm        Generation: 555         Fitness: 13
Current best: GenetiM pbsorithm        Generation: 556         Fitness: 13
Current best: Genetic tbLorithm        Generation: 557         Fitness: 14
Current best: Genetic tbLorithm        Generation: 558         Fitness: 14
Current best: Genetic tbLorithm        Generation: 559         Fitness: 14
Current best: Genetic tbLorithm        Generation: 560        Fitness: 14
Current best: Genetic tiLorithm        Generation: 561        Fitness: 14
Current best: GenetiW tbLorithm        Generation: 562        Fitness: 13
Current best: GenetiL tbLorithm        Generation: 563        Fitness: 13
Current best: Genetic tbLorithm        Generation: 564        Fitness: 14
Current best: GenetiC tbLorithm        Generation: 565        Fitness: 13
Current best: Genetic tbLorithm        Generation: 566        Fitness: 14
Current best: Geeetic pbLorithm        Generation: 567        Fitness: 13
Current best: GenetiW pbLorithm        Generation: 568        Fitness: 13
Current best: GenetiW ebLorithm        Generation: 569        Fitness: 13
Current best: Genetic pbsorithm        Generation: 570        Fitness: 14
Current best: Genetic pbsorithm        Generation: 571        Fitness: 14
Current best: Genetic pbsorithm        Generation: 572        Fitness: 14
Current best: GenXtid pbgorithm        Generation: 573        Fitness: 13
Current best: Genetid pbgorithm        Generation: 574        Fitness: 14
Current best: Genetid pbgorithm        Generation: 575        Fitness: 14
Current best: GenetiC pbgorithm        Generation: 576        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 577        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 578        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 579        Fitness: 14
Current best: GenetiC pbgorithm        Generation: 580        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 581        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 582        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 583        Fitness: 14
Current best: GenetiC pbgorithm        Generation: 584        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 585        Fitness: 14
Current best: GenetiC pbgorithm        Generation: 586        Fitness: 14
Current best: GenetiC pbgorithm        Generation: 587        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 588        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 589        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 590        Fitness: 14
Current best: GenetiC pbgorithm        Generation: 591        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 592        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 593        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 594        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 595        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 596        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 597        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 598        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 599        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 600        Fitness: 14
Current best: GenetiW pbForithm        Generation: 601        Fitness: 13
Current best: GenetiW pbgorithm        Generation: 602        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 603        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 604         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 605         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 606         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 607         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 608         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 609         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 610         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 611         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 612         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 613         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 614         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 615         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 616         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 617         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 618         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 619         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 620         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 621         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 622         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 623         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 624         Fitness: 14
Current best: Genetik pbgorithm        Generation: 625         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 626         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 627         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 628         Fitness: 14
Current best: GenetiW vbgorithm        Generation: 629         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 630         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 631         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 632         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 633         Fitness: 14
Current best: GenetiX pbgorithm        Generation: 634         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 635         Fitness: 14
Current best: GenetiC pbgorithm        Generation: 636         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 637         Fitness: 14
Current best: GenetiW vbgorithm        Generation: 638         Fitness: 14
Current best: GenetiC vbgorithm        Generation: 639         Fitness: 14
Current best: GenetiX pbgorithm        Generation: 640         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 641         Fitness: 14
Current best: GenetiW vbgorithm        Generation: 642         Fitness: 14
Current best: GenetiW vbgorithm        Generation: 643         Fitness: 14
Current best: GenetiW vbgorithm        Generation: 644         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 645         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 646         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 647         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 648         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 649         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 650         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 651         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 652         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 653         Fitness: 14
Current best: GenetiW vbgorithm         Generation: 654         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 655         Fitness: 14
Current best: Genetia pbgorithm         Generation: 656         Fitness: 14
Current best: Genetia pbgorithm         Generation: 657         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 658         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 659         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 660         Fitness: 14
Current best: GenetiC Cbgorithm         Generation: 661         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 662         Fitness: 14
Current best: GenetiC jbgorithm         Generation: 663         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 664         Fitness: 14
Current best: GenetiW vbgorithm         Generation: 665         Fitness: 14
Current best: GenetiW vbgorithm         Generation: 666         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 667         Fitness: 14
Current best: Genetia pbgorithm         Generation: 668         Fitness: 14
Current best: GenetiE pWgorithm         Generation: 669         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 670         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 671         Fitness: 14
Current best: GenetiA pbgorithm         Generation: 672         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 673         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 674         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 675         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 676         Fitness: 14
Current best: GenetiW Sbgorithm         Generation: 677         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 678         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 679         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 680         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 681         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 682         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 683         Fitness: 14
Current best: GenetiW Ibgorithm         Generation: 684         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 685         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 686         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 687         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 688         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 689         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 690         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 691         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 692         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 693         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 694         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 695         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 696         Fitness: 14
Current best: GenetiQ pbgorithm         Generation: 697         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 698         Fitness: 14
Current best: GenetiW pUgorithm         Generation: 699         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 700         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 701         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 702         Fitness: 14
Current best: GenetiC pUgorithm         Generation: 703         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 704         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 705         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 706         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 707         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 708         Fitness: 14
Current best: GenetiA pbgorithm         Generation: 709         Fitness: 14
Current best: GenetiC pUgorithm         Generation: 710         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 711         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 712         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 713         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 714         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 715         Fitness: 14
Current best: GenetiW pUgorithm         Generation: 716         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 717         Fitness: 14
Current best: GenetiA pbgorithm         Generation: 718         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 719         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 720         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 721         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 722         Fitness: 14
Current best: GenetiA pbgorithm         Generation: 723         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 724         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 725         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 726         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 727         Fitness: 14
Current best: GenetiA pbgorithm         Generation: 728         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 729         Fitness: 14
Current best: GenetiC pbgorithm         Generation: 730         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 731         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 732         Fitness: 14
Current best: GenetiW tmgorithm         Generation: 733         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 734         Fitness: 14
Current best: GenetiW pbgorithm         Generation: 735         Fitness: 14
Current best: GenetiW pegorithm         Generation: 736         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 737         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 738         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 739         Fitness: 14
Current best: GenetiW pDgorithm        Generation: 740         Fitness: 14
Current best: GenetiW plgorithm        Generation: 741         Fitness: 15
Current best: GenetiW plgorithm        Generation: 742         Fitness: 15
Current best: GenetiW plgorithm        Generation: 743         Fitness: 15
Current best: GenetiW plgorithm        Generation: 744         Fitness: 15
Current best: GenetiW plgorithm        Generation: 745         Fitness: 15
Current best: GenetiW pDgorithm        Generation: 746         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 747         Fitness: 14
Current best: GenetiW pegorithm        Generation: 748         Fitness: 14
Current best: GenetiW pDgorithm        Generation: 749         Fitness: 14
Current best: GenetiW pegorithm        Generation: 750         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 751         Fitness: 14
Current best: GenetiK pbgorithm        Generation: 752         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 753         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 754         Fitness: 14
Current best: GenetiW plgorithm        Generation: 755         Fitness: 15
Current best: GenetiW plgorithm        Generation: 756         Fitness: 15
Current best: GenetiW plgorithm        Generation: 757         Fitness: 15
Current best: GenetiW plgorithm        Generation: 758         Fitness: 15
Current best: GenetiW plgorithm        Generation: 759         Fitness: 15
Current best: GenetiW plgorithm        Generation: 760         Fitness: 15
Current best: GenetiW p gorithm        Generation: 761         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 762         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 763         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 764         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 765         Fitness: 14
Current best: GenetiW pDgorithm        Generation: 766         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 767         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 768         Fitness: 14
Current best: GenetiW pegorithm        Generation: 769         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 770         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 771         Fitness: 14
Current best: GenetiW pegorithm        Generation: 772         Fitness: 14
Current best: GenetiW pDgorithm        Generation: 773         Fitness: 14
Current best: GenetiK pbgorithm        Generation: 774         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 775         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 776         Fitness: 14
Current best: GenetiW pDgorithm        Generation: 777         Fitness: 14
Current best: GenetiW  egorithm        Generation: 778         Fitness: 14
Current best: GenetiK pbgorithm        Generation: 779         Fitness: 14
Current best: GenetiW pbgorithm        Generation: 780         Fitness: 14
Current best: GenetiW pegorithm        Generation: 781        Fitness: 14
Current best: GenetiW Bbgorithm        Generation: 782        Fitness: 14
Current best: GenetiK pbgorithm        Generation: 783        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 784        Fitness: 14
Current best: GenetiK pegorithm        Generation: 785        Fitness: 14
Current best: GenetiW pegorithm        Generation: 786        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 787        Fitness: 14
Current best: GenetiK pegorithm        Generation: 788        Fitness: 14
Current best: GenetiW pegorithm        Generation: 789        Fitness: 14
Current best: GenetiK pegorithm        Generation: 790        Fitness: 14
Current best: GenetiK pegorithm        Generation: 791        Fitness: 14
Current best: GenetiW pegorithm        Generation: 792        Fitness: 14
Current best: GenetiK pegorithm        Generation: 793        Fitness: 14
Current best: GenetiK pegorithm        Generation: 794        Fitness: 14
Current best: GenetiK pbgorithm        Generation: 795        Fitness: 14
Current best: GenetiL pbgorithm        Generation: 796        Fitness: 14
Current best: GenetiK pegorithm        Generation: 797        Fitness: 14
Current best: GenetiL pbgorithm        Generation: 798        Fitness: 14
Current best: GenetiW pegorithm        Generation: 799        Fitness: 14
Current best: GenetiK pegorithm        Generation: 800        Fitness: 14
Current best: GenetiK pbgorithm        Generation: 801        Fitness: 14
Current best: GenetiK pegorithm        Generation: 802        Fitness: 14
Current best: GenetiK pegorithm        Generation: 803        Fitness: 14
Current best: GenetiK pegorithm        Generation: 804        Fitness: 14
Current best: GenetiK pegorithm        Generation: 805        Fitness: 14
Current best: GenetiL pegorithm        Generation: 806        Fitness: 14
Current best: Genetiz pbgorithm        Generation: 807        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 808        Fitness: 14
Current best: GenetiW pegorithm        Generation: 809        Fitness: 14
Current best: GenetiK pegorithm        Generation: 810        Fitness: 14
Current best: GenetiK pegorithm        Generation: 811        Fitness: 14
Current best: GenetiW pbgorithm        Generation: 812        Fitness: 14
Current best: GenetiL pegorithm        Generation: 813        Fitness: 14
Current best: Genetic pbgorithm        Generation: 814        Fitness: 15
Current best: Genetiz pegorithm        Generation: 815        Fitness: 14
Current best: GenetiW pegorithm        Generation: 816        Fitness: 14
Current best: GenetiK pegorithm        Generation: 817        Fitness: 14
Current best: GenetiW pwgorithm        Generation: 818        Fitness: 14
Current best: Genetic pegorithm        Generation: 819        Fitness: 15
Current best: Genetic pegorithm        Generation: 820        Fitness: 15
Current best: Genetic pwgorithm        Generation: 821        Fitness: 15
Current best: Genetic pbgorithm        Generation: 822        Fitness: 15
Current best: Genetic pbgorithm        Generation: 823        Fitness: 15
Current best: Genetic pbgorithm        Generation: 824        Fitness: 15
Current best: Genetic pbgorithm         Generation: 825         Fitness: 15
Current best: Genetic pbgorithm         Generation: 826         Fitness: 15
Current best: Genetic pbgorithm         Generation: 827         Fitness: 15
Current best: Genetic pegorithm         Generation: 828         Fitness: 15
Current best: Genetic pegorithm         Generation: 829         Fitness: 15
Current best: Genetic pegorithm         Generation: 830         Fitness: 15
Current best: Genetic pbgorithm         Generation: 831         Fitness: 15
Current best: Genetic pegorithm         Generation: 832         Fitness: 15
Current best: Genetic awgorithm         Generation: 833         Fitness: 15
Current best: Genetic pegorithm         Generation: 834         Fitness: 15
Current best: Genetic pDgorithm         Generation: 835         Fitness: 15
Current best: Genetic pbgorithm         Generation: 836         Fitness: 15
Current best: Genetic pbgorithm         Generation: 837         Fitness: 15
Current best: Genetic pegorithm         Generation: 838         Fitness: 15
Current best: Genetic pegorithm         Generation: 839         Fitness: 15
Current best: Genetic pbgorithm         Generation: 840         Fitness: 15
Current best: Genetic pegorithm         Generation: 841         Fitness: 15
Current best: Genetic pbgorithm         Generation: 842         Fitness: 15
Current best: Genetic pbgorithm         Generation: 843         Fitness: 15
Current best: Genetic pbgorithm         Generation: 844         Fitness: 15
Current best: Genetic pbgorithm         Generation: 845         Fitness: 15
Current best: Genetic pbgorithm         Generation: 846         Fitness: 15
Current best: GenetiK pbgorithm         Generation: 847         Fitness: 14
Current best: GenetiW abgorithm         Generation: 848         Fitness: 14
Current best: Genetic pbgorithm         Generation: 849         Fitness: 15
Current best: Genetic pbgorithm         Generation: 850         Fitness: 15
Current best: GenetiW ibgorithm         Generation: 851         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 852         Fitness: 14
Current best: GenetiW ibgorithm         Generation: 853         Fitness: 14
Current best: GenetiW ibgorithm         Generation: 854         Fitness: 14
Current best: GenetiW ibgorithm         Generation: 855         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 856         Fitness: 14
Current best: GenetiW ibgorithm         Generation: 857         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 858         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 859         Fitness: 14
Current best: GenetiW iDgorithm         Generation: 860         Fitness: 14
Current best: GenetiW ibgorithm         Generation: 861         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 862         Fitness: 14
Current best: GenetiW iDgorithm         Generation: 863         Fitness: 14
Current best: GenetiA abgorithm         Generation: 864         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 865         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 866         Fitness: 14
Current best: GenetiW pegorithm         Generation: 867         Fitness: 14
Current best: GenetiW iDgorithm         Generation: 868         Fitness: 14
Current best: GenetiW pegorithm         Generation: 869         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 870         Fitness: 14
Current best: GenetiW iDgorithm         Generation: 871         Fitness: 14
Current best: GenetiW iDgorithm         Generation: 872         Fitness: 14
Current best: GenetiW pDgorithm         Generation: 873         Fitness: 14
Current best: GenetiW pDgorithm         Generation: 874         Fitness: 14
Current best: GenetiK iDgorithm         Generation: 875         Fitness: 14
Current best: Genetib ibgorithm         Generation: 876         Fitness: 14
Current best: GenetiW qDgorithm         Generation: 877         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 878         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 879         Fitness: 14
Current best: GenetiA aDgorithm         Generation: 880         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 881         Fitness: 14
Current best: GenetiK obgorithm         Generation: 882         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 883         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 884         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 885         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 886         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 887         Fitness: 14
Current best: GenetiW pLgorithm         Generation: 888         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 889         Fitness: 14
Current best: GenetiW pLgorithm         Generation: 890         Fitness: 14
Current best: GenetiK iDgorithm         Generation: 891         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 892         Fitness: 14
Current best: GenetiW iDgorithm         Generation: 893         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 894         Fitness: 14
Current best: GenetiW awgorithm         Generation: 895         Fitness: 14
Current best: GenetiK iDgorithm         Generation: 896         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 897         Fitness: 14
Current best: GenetiK iwgorithm         Generation: 898         Fitness: 14
Current best: GenetiW iDgorithm         Generation: 899         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 900         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 901         Fitness: 14
Current best: GenetiW ipgorithm         Generation: 902         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 903         Fitness: 14
Current best: GenetiW aLgorithm         Generation: 904         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 905         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 906         Fitness: 14
Current best: GenetiK iDgorithm         Generation: 907         Fitness: 14
Current best: GenetiW aLgorithm         Generation: 908         Fitness: 14
Current best: GenetiW pDgorithm         Generation: 909         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 910         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 911         Fitness: 14
Current best: GenetiW aDgorithm        Generation: 912        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 913        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 914        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 915        Fitness: 14
Current best: GenetiK iDgorithm        Generation: 916        Fitness: 14
Current best: GenetiW aLgorithm        Generation: 917        Fitness: 14
Current best: GenetiK iDgorithm        Generation: 918        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 919        Fitness: 14
Current best: GenetiK iDgorithm        Generation: 920        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 921        Fitness: 14
Current best: GenetiK aLgorithm        Generation: 922        Fitness: 14
Current best: GenetiW aLgorithm        Generation: 923        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 924        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 925        Fitness: 14
Current best: GenetiY iDgorithm        Generation: 926        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 927        Fitness: 14
Current best: GenetiW iLgorithm        Generation: 928        Fitness: 14
Current best: GenetiW iLgorithm        Generation: 929        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 930        Fitness: 14
Current best: GenetiW kDgorithm        Generation: 931        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 932        Fitness: 14
Current best: GenetiW iDgorithm        Generation: 933        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 934        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 935        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 936        Fitness: 14
Current best: GenetiK pDgorithm        Generation: 937        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 938        Fitness: 14
Current best: GenetiK aLgorithm        Generation: 939        Fitness: 14
Current best: GenetiK aLgorithm        Generation: 940        Fitness: 14
Current best: GenetiK pDgorithm        Generation: 941        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 942        Fitness: 14
Current best: GenetiK pDgorithm        Generation: 943        Fitness: 14
Current best: GenetiO aDgorithm        Generation: 944        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 945        Fitness: 14
Current best: GenetiK aLgorithm        Generation: 946        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 947        Fitness: 14
Current best: GenetiK aDgorithm        Generation: 948        Fitness: 14
Current best: GenetiK iDgorithm        Generation: 949        Fitness: 14
Current best: GenetiK kDgorithm        Generation: 950        Fitness: 14
Current best: GenetiW aDgorithm        Generation: 951        Fitness: 14
Current best: GenetiK kDgorithm        Generation: 952        Fitness: 14
Current best: Genetib kDgorithm        Generation: 953        Fitness: 14
Current best: GenetiK iDgorithm        Generation: 954        Fitness: 14
Current best: GenetiO aDgorithm        Generation: 955        Fitness: 14
Current best: GenetiK kDgorithm         Generation: 956         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 957         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 958         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 959         Fitness: 14
Current best: GenetiO kDgorithm         Generation: 960         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 961         Fitness: 14
Current best: GenetiK NDgorithm         Generation: 962         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 963         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 964         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 965         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 966         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 967         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 968         Fitness: 14
Current best: Genetic pDgorithm         Generation: 969         Fitness: 15
Current best: GenetiW aDgorithm         Generation: 970         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 971         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 972         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 973         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 974         Fitness: 14
Current best: GenetiK kLgorithm         Generation: 975         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 976         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 977         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 978         Fitness: 14
Current best: GenetiK kLgorithm         Generation: 979         Fitness: 14
Current best: GenetiK kLgorithm         Generation: 980         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 981         Fitness: 14
Current best: GenetiK kLgorithm         Generation: 982         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 983         Fitness: 14
Current best: GenetiK oDgorithm         Generation: 984         Fitness: 14
Current best: GenetiO kDgorithm         Generation: 985         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 986         Fitness: 14
Current best: GenetiO aDgorithm         Generation: 987         Fitness: 14
Current best: GenetiW oDgorithm         Generation: 988         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 989         Fitness: 14
Current best: GenetiK oDgorithm         Generation: 990         Fitness: 14
Current best: GenetiO aDgorithm         Generation: 991         Fitness: 14
Current best: GenetiK pDgorithm         Generation: 992         Fitness: 14
Current best: GenetiO aDgorithm         Generation: 993         Fitness: 14
Current best: GenetiK kDgorithm         Generation: 994         Fitness: 14
Current best: GenetiW aDgorithm         Generation: 995         Fitness: 14
Current best: GenetiW pDgorithm         Generation: 996         Fitness: 14
Current best: GenetiV pDgorithm         Generation: 997         Fitness: 14
Current best: GenetiO aDgorithm         Generation: 998         Fitness: 14
Current best: GenetiK aDgorithm         Generation: 999         Fitness: 14
Current best: GenetiK GDgorithm         Generation: 1000                Fitn
ess: 14
```

```
Current best: GenetiK GDgorithm        Generation: 1001                 Fitn
ess: 14
Current best: GenetiK GDgorithm        Generation: 1002                 Fitn
ess: 14
Current best: GenetiK GDgorithm        Generation: 1003                 Fitn
ess: 14
Current best: GenetiK MDgorithm        Generation: 1004                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1005                 Fitn
ess: 14
Current best: GenetiO kDgorithm        Generation: 1006                 Fitn
ess: 14
Current best: GenetiO kDgorithm        Generation: 1007                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1008                 Fitn
ess: 14
Current best: GenetiV pDgorithm        Generation: 1009                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1010                 Fitn
ess: 14
Current best: GenetiO pDgorithm        Generation: 1011                 Fitn
ess: 14
Current best: GenetiV kDgorithm        Generation: 1012                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1013                 Fitn
ess: 14
Current best: GenetiV aDgorithm        Generation: 1014                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1015                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1016                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1017                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1018                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1019                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1020                 Fitn
ess: 14
Current best: GenetiO kDgorithm        Generation: 1021                 Fitn
ess: 14
Current best: GenetiK aDgorithm        Generation: 1022                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1023                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1024                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1025                 Fitn
ess: 14
Current best: GenetiW GDgorithm        Generation: 1026                 Fitn
ess: 14
```

```
Current best: GenetiO GDgorithm        Generation: 1027                 Fitn
ess: 14
Current best: GenetiW GDgorithm        Generation: 1028                 Fitn
ess: 14
Current best: GenetiK GDgorithm        Generation: 1029                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1030                 Fitn
ess: 14
Current best: GenetiW GDgorithm        Generation: 1031                 Fitn
ess: 14
Current best: GenetiW UDgorithm        Generation: 1032                 Fitn
ess: 14
Current best: GenetiO aEgorithm        Generation: 1033                 Fitn
ess: 14
Current best: GenetiX aTgorithm        Generation: 1034                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1035                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1036                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1037                 Fitn
ess: 14
Current best: GenetiO nDgorithm        Generation: 1038                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1039                 Fitn
ess: 14
Current best: GenetiK GDgorithm        Generation: 1040                 Fitn
ess: 14
Current best: GenetiK a gorithm        Generation: 1041                 Fitn
ess: 14
Current best: GenetiK GTgorithm        Generation: 1042                 Fitn
ess: 14
Current best: GenetiO GDgorithm        Generation: 1043                 Fitn
ess: 14
Current best: GenetiO nDgorithm        Generation: 1044                 Fitn
ess: 14
```

```
Current best: GenetiK GDgorithm        Generation: 1045                 Fitn
ess: 14
Current best: GenetiK aDgorithm        Generation: 1046                 Fitn
ess: 14
Current best: GenetiW GDgorithm        Generation: 1047                 Fitn
ess: 14
Current best: GenetiK GDgorithm        Generation: 1048                 Fitn
ess: 14
Current best: GenetiO aDgorithm        Generation: 1049                 Fitn
ess: 14
Current best: GenetiO ADgorithm        Generation: 1050                 Fitn
ess: 15
Current best: GenetiK aDgorithm        Generation: 1051                 Fitn
ess: 14
Current best: GenetiK GDgorithm        Generation: 1052                 Fitn
ess: 14
Current best: GenetiK aDgorithm        Generation: 1053                 Fitn
ess: 14
Current best: GenetiM aDgorithm        Generation: 1054                 Fitn
ess: 14
Current best: GenetiK aDgorithm        Generation: 1055                 Fitn
ess: 14
Current best: GenetiK algoritkm        Generation: 1056                 Fitn
ess: 14
Current best: GenetiK algorithm        Generation: 1057                 Fitn
ess: 15
Current best: GenetiK algorithm        Generation: 1058                 Fitn
ess: 15
Current best: GenetiK algorithm        Generation: 1059                 Fitn
ess: 15
Current best: GenetiK algorithm        Generation: 1060                 Fitn
ess: 15
Current best: GenetiK algorithm        Generation: 1061                 Fitn
ess: 15
Current best: GenetiK algorithm        Generation: 1062                 Fitn
ess: 15
Current best: GenetiM algorithm        Generation: 1063                 Fitn
ess: 15
Current best: Genetic aDgorithm        Generation: 1064                 Fitn
ess: 15
Current best: Genetiw algorithm        Generation: 1065                 Fitn
ess: 15
Current best: Genetic aDgorithm        Generation: 1066                 Fitn
ess: 15
Current best: Genetic aDgorithm        Generation: 1067                 Fitn
ess: 15
Current best: Genetic aDgorithm        Generation: 1068                 Fitn
ess: 15
Current best: Genetic aRgorithm        Generation: 1069                 Fitn
ess: 15
Current best: GenetiW lDgorithm        Generation: 1070                 Fitn
ess: 14
```

```
Current best: GenetiK aNgorithm         Generation: 1071                 Fitn
ess: 14
Current best: GenetiO aDgorithm         Generation: 1072                 Fitn
ess: 14
Current best: GenetiO aDgorithm         Generation: 1073                 Fitn
ess: 14
Current best: GenetiO eDgorithm         Generation: 1074                 Fitn
ess: 14
Current best: GenetiS aNgorithm         Generation: 1075                 Fitn
ess: 14
Current best: GenetiO aDgorithm         Generation: 1076                 Fitn
ess: 14
Current best: GenetiO aDgorithm         Generation: 1077                 Fitn
ess: 14
Current best: GenetiW aGgorithm         Generation: 1078                 Fitn
ess: 14
Current best: GenetiO aDgorithm         Generation: 1079                 Fitn
ess: 14
Current best: GenetiK GDgorithm         Generation: 1080                 Fitn
ess: 14
Current best: GenetiK aNgorithm         Generation: 1081                 Fitn
ess: 14
Current best: GenetiS nDgorithm         Generation: 1082                 Fitn
ess: 14
Current best: GenetiK aNgorithm         Generation: 1083                 Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1084                 Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1085                 Fitn
ess: 14
Current best: GenetiK nDgorithm         Generation: 1086                 Fitn
ess: 14
Current best: GenetiK aDgorithm         Generation: 1087                 Fitn
ess: 14
Current best: GenetiK aNgorithm         Generation: 1088                 Fitn
ess: 14
```

```
Current best: GenetiK gDgorithm         Generation: 1089                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1090                Fitn
ess: 14
Current best: GenetiK nDgorithm         Generation: 1091                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1092                Fitn
ess: 14
Current best: GenetiO aDgorithm         Generation: 1093                Fitn
ess: 14
Current best: GenetiO gDgorithm         Generation: 1094                Fitn
ess: 14
Current best: GenetiO aDgorithm         Generation: 1095                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1096                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1097                Fitn
ess: 14
Current best: GenetiO gDgorithm         Generation: 1098                Fitn
ess: 14
Current best: GenetiN gDgorithm         Generation: 1099                Fitn
ess: 14
Current best: GenetiO gNgorithm         Generation: 1100                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1101                Fitn
ess: 14
Current best: GenetiN gDgorithm         Generation: 1102                Fitn
ess: 14
Current best: GenetiK gNgorithm         Generation: 1103                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1104                Fitn
ess: 14
Current best: GenetiO aNgorithm         Generation: 1105                Fitn
ess: 14
Current best: GenetiK gNgorithm         Generation: 1106                Fitn
ess: 14
Current best: GenetiK gNgorithm         Generation: 1107                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1108                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1109                Fitn
ess: 14
Current best: GenetiO gDgorithm         Generation: 1110                Fitn
ess: 14
Current best: GenetiO gDgorithm         Generation: 1111                Fitn
ess: 14
Current best: GenetiK aNgorithm         Generation: 1112                Fitn
ess: 14
Current best: GenetiK gDgorithm         Generation: 1113                Fitn
ess: 14
Current best: GenetiO gNgorithm         Generation: 1114                Fitn
ess: 14
```

```
Current best: GenetiK gNgorithm        Generation: 1115                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1116                 Fitn
ess: 14
Current best: GenetiO aNgorithm        Generation: 1117                 Fitn
ess: 14
Current best: GenetiK aNgorithm        Generation: 1118                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1119                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1120                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1121                 Fitn
ess: 14
Current best: GenetiO gNgorithm        Generation: 1122                 Fitn
ess: 14
Current best: GenetiO gNgorithm        Generation: 1123                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1124                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1125                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1126                 Fitn
ess: 14
Current best: GenetiK gNgorithm        Generation: 1127                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1128                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1129                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1130                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1131                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1132                 Fitn
ess: 14
```

```
Current best: GenetiK gDgorithm        Generation: 1133                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1134                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1135                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1136                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1137                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1138                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1139                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1140                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1141                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1142                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1143                 Fitn
ess: 14
Current best: GenetiK glgorithm        Generation: 1144                 Fitn
ess: 15
Current best: GenetiO glgorithm        Generation: 1145                 Fitn
ess: 15
Current best: GenetiK gDgorithm        Generation: 1146                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1147                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1148                 Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1149                 Fitn
ess: 14
Current best: GenetiS gDgorithm        Generation: 1150                 Fitn
ess: 14
Current best: GenetiS gDgorithm        Generation: 1151                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1152                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1153                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1154                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1155                 Fitn
ess: 14
Current best: Genetie gDgorithm        Generation: 1156                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1157                 Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1158                 Fitn
ess: 14
```

```
Current best: GenetiK gDgorithm        Generation: 1159                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1160                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1161                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1162                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1163                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1164                Fitn
ess: 14
Current best: GenetiK gHgorithm        Generation: 1165                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1166                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1167                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1168                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1169                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1170                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1171                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1172                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1173                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1174                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1175                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1176                Fitn
ess: 14
```

```
Current best: GenetiO gDgorithm        Generation: 1177                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1178                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1179                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1180                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1181                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1182                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1183                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1184                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1185                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1186                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1187                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1188                Fitn
ess: 14
Current best: Genetie lDgorithm        Generation: 1189                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1190                Fitn
ess: 14
Current best: Genetic gDgorithm        Generation: 1191                Fitn
ess: 15
Current best: GenetiO gDgorithm        Generation: 1192                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1193                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1194                Fitn
ess: 14
Current best: Genetie lDgorithm        Generation: 1195                Fitn
ess: 14
Current best: GenetiO ODgorithm        Generation: 1196                Fitn
ess: 14
Current best: GenetiK lDgorithm        Generation: 1197                Fitn
ess: 14
Current best: GenetiO gDgorithm        Generation: 1198                Fitn
ess: 14
Current best: GenetiK gDgorithm        Generation: 1199                Fitn
ess: 14
```

The genetic algorithm was able to converge! We implore you to rerun the above cell and play around with `target, max_population, f_thres, ngen` etc parameters to get a better intuition of how the algorithm works. To summarize, if we can define the problem states in simple array format and if we can create a fitness function to gauge how good or bad our approximate solutions are, there is a high chance that we can get a satisfactory solution using a genetic algorithm.

- There is also a better GUI version of this program `genetic_algorithm_example.py` in the GUI folder for you to play around with.

Usage

Below we give two example usages for the genetic algorithm, for a graph coloring problem and the 8 queens problem.

Graph Coloring

First we will take on the simpler problem of coloring a small graph with two colors. Before we do anything, let's imagine how a solution might look. First, we have to represent our colors. Say, 'R' for red and 'G' for green. These make up our gene pool. What of the individual solutions though? For that, we will look at our problem. We stated we have a graph. A graph has nodes and edges, and we want to color the nodes. Naturally, we want to store each node's color. If we have four nodes, we can store their colors in a list of genes, one for each node. A possible solution will then look like this: ['R', 'R', 'G', 'R']. In the general case, we will represent each solution with a list of chars ('R' and 'G'), with length the number of nodes.

Next we need to come up with a fitness function that appropriately scores individuals. Again, we will look at the problem definition at hand. We want to color a graph. For a solution to be optimal, no edge should connect two nodes of the same color. How can we use this information to score a solution? A naive (and ineffective) approach would be to count the different colors in the string. So ['R', 'R', 'R', 'R'] has a score of 1 and ['R', 'R', 'G', 'G'] has a score of 2. Why that fitness function is not ideal though? Why, we forgot the information about the edges! The edges are pivotal to the problem and the above function only deals with node colors. We didn't use all the information at hand and ended up with an ineffective answer. How, then, can we use that information to our advantage?

We said that the optimal solution will have all the edges connecting nodes of different color. So, to score a solution we can count how many edges are valid (aka connecting nodes of different color). That is a great fitness function!

Let's jump into solving this problem using the `genetic_algorithm` function.

First we need to represent the graph. Since we mostly need information about edges, we will just store the edges. We will denote edges with capital letters and nodes with integers:

In [105…
```
edges = {
    'A': [0, 1],
    'B': [0, 3],
    'C': [1, 2],
    'D': [2, 3]
}
```

Edge 'A' connects nodes 0 and 1, edge 'B' connects nodes 0 and 3 etc.

We already said our gene pool is 'R' and 'G', so we can jump right into initializing our population. Since we have only four nodes, `state_length` should be 4. For the number of individuals, we will try 8. We can increase this number if we need higher accuracy, but be careful! Larger populations need more computating power and take longer. You need to strike that sweet balance between accuracy and cost (the ultimate dilemma of the programmer!).

In [106…
```
population = init_population(8, ['R', 'G'], 4)
print(population)
```

```
[['R', 'G', 'G', 'G'], ['G', 'G', 'R', 'R'], ['R', 'G', 'G', 'G'], ['G',
'R', 'G', 'R'], ['G', 'G', 'R', 'R'], ['R', 'G', 'R', 'R'], ['G', 'G', 'R',
'G'], ['R', 'G', 'G', 'R']]
```

We created and printed the population. You can see that the genes in the individuals are random and there are 8 individuals each with 4 genes.

Next we need to write our fitness function. We previously said we want the function to count how many edges are valid. So, given a coloring/individual `c`, we will do just that:

In [107…
```
def fitness(c):
    return sum(c[n1] != c[n2] for (n1, n2) in edges.values())
```

Great! Now we will run the genetic algorithm and see what solution it gives.

In [108…
```
solution = genetic_algorithm(population, fitness, gene_pool=['R', 'G'])
print(solution)
```

```
['R', 'G', 'R', 'G']
```

The algorithm converged to a solution. Let's check its score:

In [109…
```
print(fitness(solution))
```

```
4
```

The solution has a score of 4. Which means it is optimal, since we have exactly 4 edges in our graph, meaning all are valid!

NOTE: Because the algorithm is non-deterministic, there is a chance a different solution is given. It might even be wrong, if we are very unlucky!

Eight Queens

Let's take a look at a more complicated problem.

In the *Eight Queens* problem, we are tasked with placing eight queens on an 8x8 chessboard without any queen threatening the others (aka queens should not be in the same row, column or diagonal). In its general form the problem is defined as placing *N* queens in an NxN chessboard without any conflicts.

First we need to think about the representation of each solution. We can go the naive route of representing the whole chessboard with the queens' placements on it. That is definitely one way to go about it, but for the purpose of this tutorial we will do something different. We have eight queens, so we will have a gene for each of them. The gene pool will be numbers from 0 to 7, for the different columns. The *position* of the gene in the state will denote the row the particular queen is placed in.

For example, we can have the state "03304577". Here the first gene with a value of 0 means "the queen at row 0 is placed at column 0", for the second gene "the queen at row 1 is placed at column 3" and so forth.

We now need to think about the fitness function. On the graph coloring problem we counted the valid edges. The same thought process can be applied here. Instead of edges though, we have positioning between queens. If two queens are not threatening each other, we say they are at a "non-attacking" positioning. We can, therefore, count how many such positionings are there.

Let's dive right in and initialize our population:

```
In [110…
population = init_population(100, range(8), 8)
print(population[:5])
```

```
[[2, 1, 3, 6, 4, 6, 6, 6], [1, 6, 7, 6, 6, 4, 0, 1], [6, 7, 3, 0, 1, 3, 0,
0], [1, 3, 0, 6, 0, 5, 4, 0], [4, 1, 2, 7, 1, 5, 7, 0]]
```

We have a population of 100 and each individual has 8 genes. The gene pool is the integers from 0 to 7, in string form. Above you can see the first five individuals.

Next we need to write our fitness function. Remember, queens threaten each other if they are at the same row, column or diagonal.

Since positionings are mutual, we must take care not to count them twice. Therefore for each queen, we will only check for conflicts for the queens after her.

A gene's value in an individual `q` denotes the queen's column, and the position of the gene denotes its row. We can check if the aforementioned values between two genes are the same. We also need to check for diagonals. A queen *a* is in the diagonal of another queen, *b*, if the difference of the rows between them is equal to either their difference in columns (for the diagonal on the right of *a*) or equal to the negative difference of their columns (for the left diagonal of *a*). Below is given the fitness function.

In [111…

```
def fitness(q):
    non_attacking = 0
    for row1 in range(len(q)):
        for row2 in range(row1+1, len(q)):
            col1 = int(q[row1])
            col2 = int(q[row2])
            row_diff = row1 - row2
            col_diff = col1 - col2

            if col1 != col2 and row_diff != col_diff and row_diff != -col_di
                non_attacking += 1

    return non_attacking
```

Note that the best score achievable is 28. That is because for each queen we only check for the queens after her. For the first queen we check 7 other queens, for the second queen 6 others and so on. In short, the number of checks we make is the sum 7+6+5+...+1. Which is equal to 7*(7+1)/2 = 28.

Because it is very hard and will take long to find a perfect solution, we will set the fitness threshold at 25. If we find an individual with a score greater or equal to that, we will halt. Let's see how the genetic algorithm will fare.

In [112…

```
solution = genetic_algorithm(population, fitness, f_thres=25, gene_pool=rang
print(solution)
print(fitness(solution))
```

```
[3, 7, 5, 7, 2, 1, 6, 4]
25
```

Above you can see the solution and its fitness score, which should be no less than 25.

This is where we conclude Genetic Algorithms.

N-Queens Problem

Here, we will look at the generalized cae of the Eight Queens problem.
We are given a `N` x `N` chessboard, with `N` queens, and we need to place them in such a way that no two queens can attack each other.
We will solve this problem using search algorithms. To do this, we already have a `NQueensProblem` class in `search.py`.

In [113…

```
psource(NQueensProblem)
```

```python
class NQueensProblem(Problem):
    """The problem of placing N queens on an NxN board with none attacking
    each other. A state is represented as an N-element array, where
    a value of r in the c-th entry means there is a queen at column c,
    row r, and a value of -1 means that the c-th column has not been
    filled in yet. We fill in columns left to right.
    >>> depth_first_tree_search(NQueensProblem(8))
    <Node (7, 3, 0, 2, 5, 1, 6, 4)>
    """

    def __init__(self, N):
        super().__init__(tuple([-1] * N))
        self.N = N

    def actions(self, state):
        """In the leftmost empty column, try all non-conflicting rows."""
        if state[-1] != -1:
            return []  # All columns filled; no successors
        else:
            col = state.index(-1)
            return [row for row in range(self.N)
                    if not self.conflicted(state, row, col)]

    def result(self, state, row):
        """Place the next queen at the given row."""
        col = state.index(-1)
        new = list(state[:])
        new[col] = row
        return tuple(new)

    def conflicted(self, state, row, col):
        """Would placing a queen at (row, col) conflict with anything?"""
        return any(self.conflict(row, col, state[c], c)
                   for c in range(col))

    def conflict(self, row1, col1, row2, col2):
        """Would putting two queens in (row1, col1) and (row2, col2) conflict?"""
        return (row1 == row2 or  # same row
                col1 == col2 or  # same column
                row1 - col1 == row2 - col2 or  # same \ diagonal
                row1 + col1 == row2 + col2)  # same / diagonal
    def goal_test(self, state):
        """Check if all columns filled, no conflicts."""
        if state[-1] == -1:
            return False
        return not any(self.conflicted(state, state[col], col)
                       for col in range(len(state)))

    def h(self, node):
        """Return number of conflicting queens for a given node"""
        num_conflicts = 0
        for (r1, c1) in enumerate(node.state):
            for (r2, c2) in enumerate(node.state):
                if (r1, c1) != (r2, c2):
                    num_conflicts += self.conflict(r1, c1, r2, c2)

        return num_conflicts
```

In `csp.ipynb` we have seen that the N-Queens problem can be formulated as a CSP and can be solved by the `min_conflicts` algorithm in a way similar to Hill-Climbing. Here, we want to solve it using heuristic search algorithms and even some classical search algorithms. The `NQueensProblem` class derives from the `Problem` class and is implemented in such a way that the search algorithms we already have, can solve it. Let's instantiate the class.

In [114…
```
nqp = NQueensProblem(8)
```

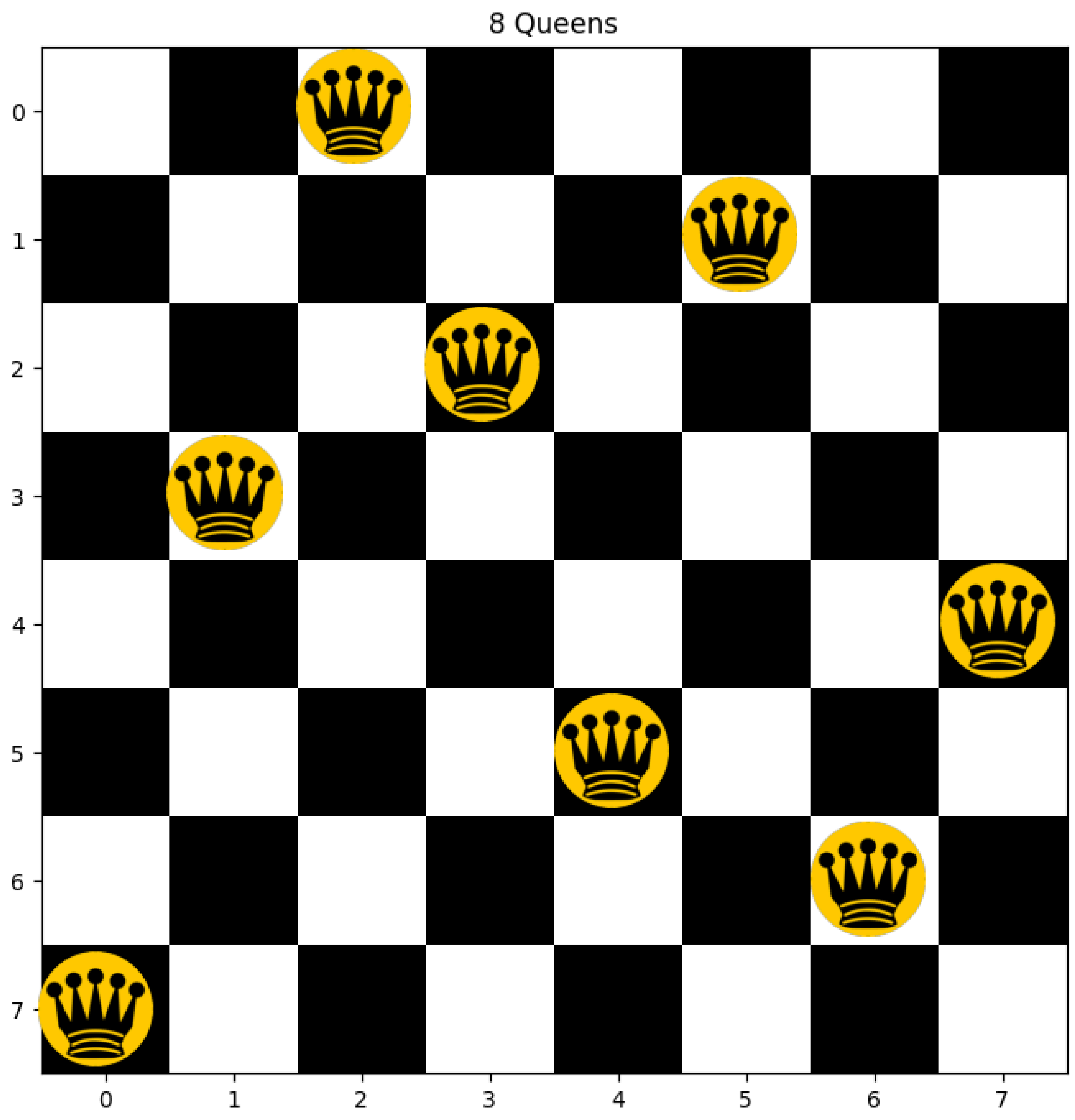
Let's use `depth_first_tree_search` first.
We will also use the %%timeit magic with each algorithm to see how much time they take.

In [115…
```
%%timeit
depth_first_tree_search(nqp)
```

```
591 µs ± 1.54 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [116…
```
dfts = depth_first_tree_search(nqp).solution()
```

In [117…
```
plot_NQueens(dfts)
```



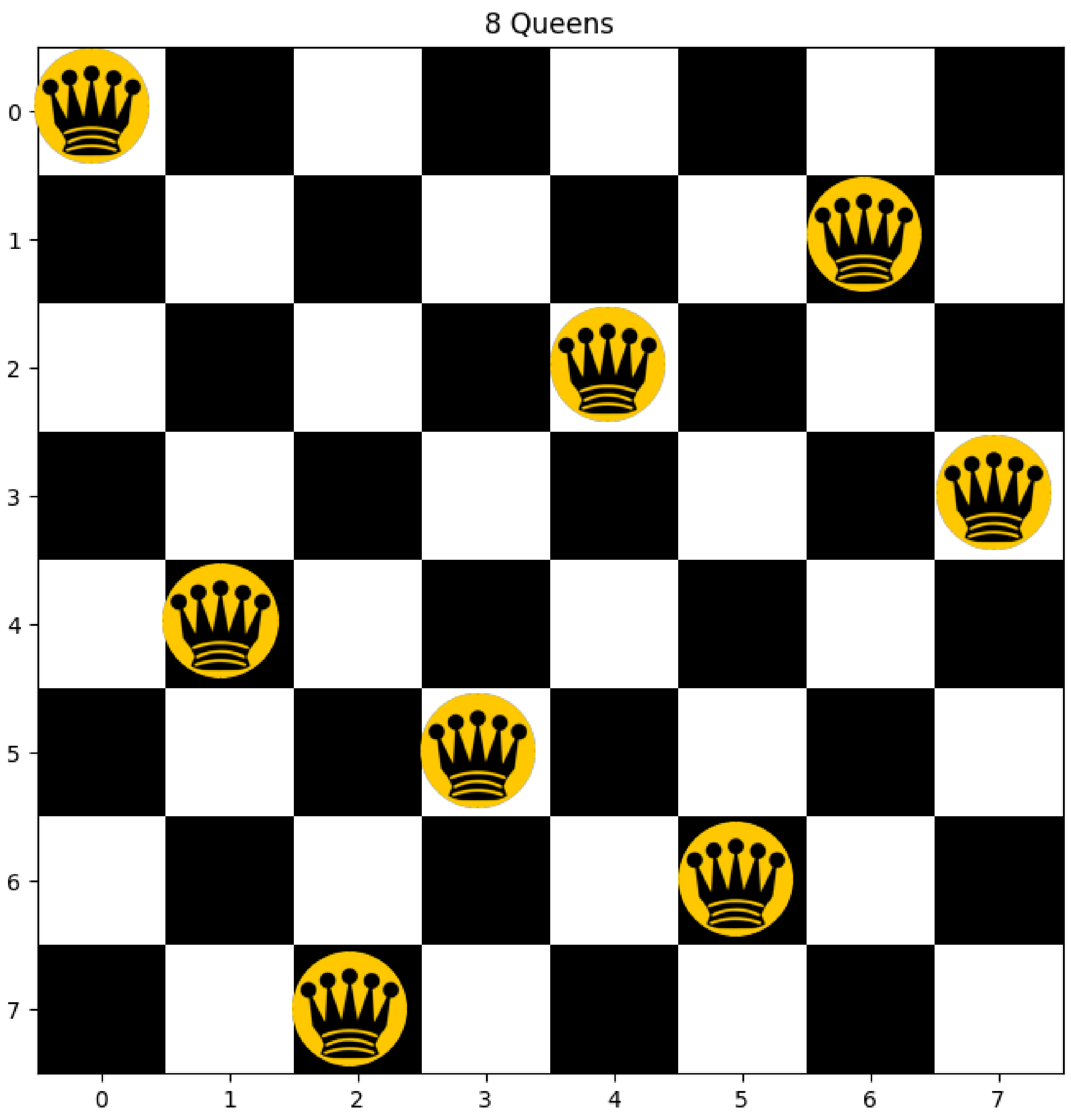
`breadth_first_tree_search`

```
In [118…  %%timeit
          breadth_first_tree_search(nqp)
```

```
10.4 ms ± 29.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [119…  bfts = breadth_first_tree_search(nqp).solution()
```

```
In [120…  plot_NQueens(bfts)
```



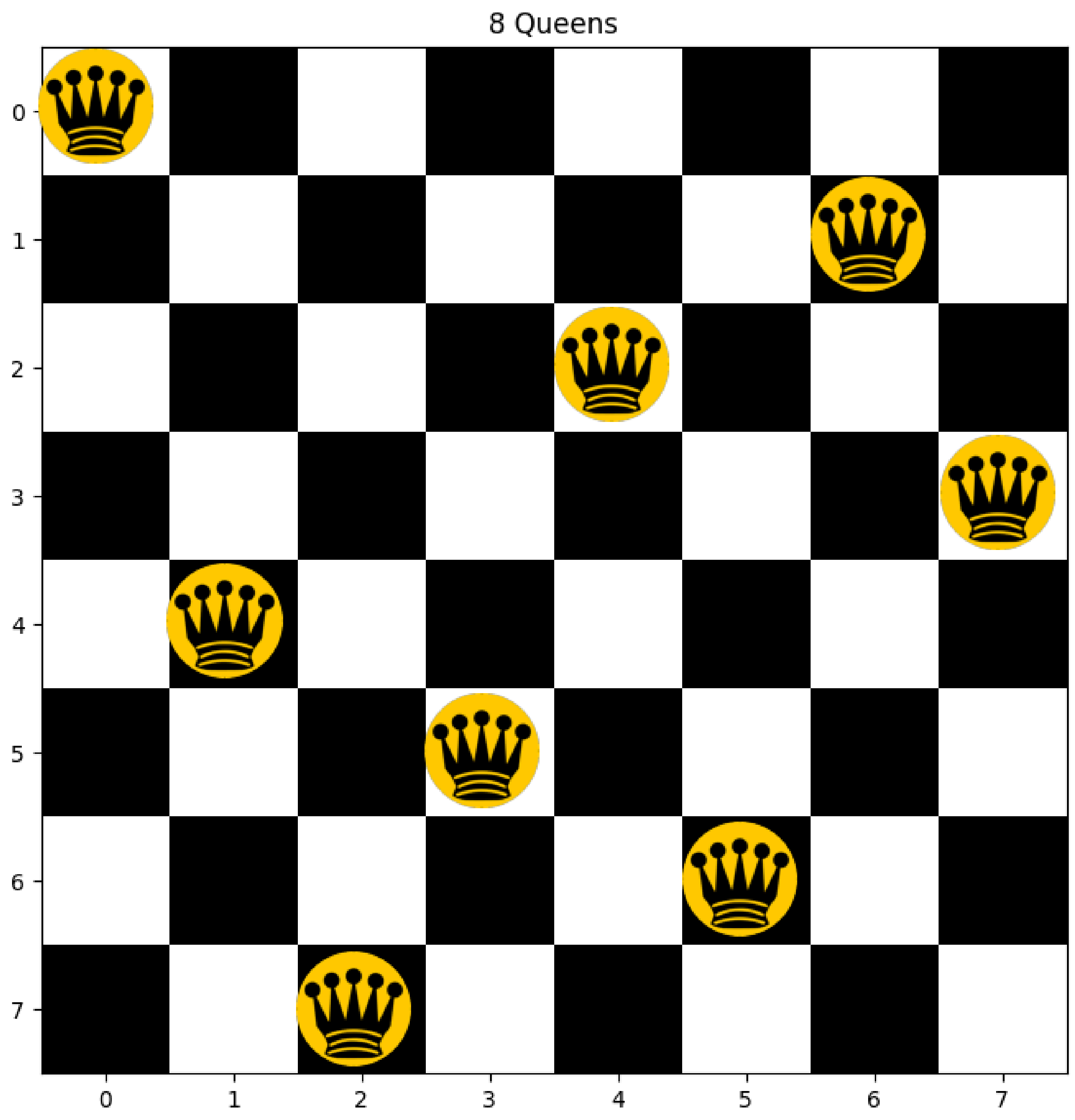
`uniform_cost_search`

In [121…
```
%%timeit
uniform_cost_search(nqp)
```

```
86.6 ms ± 315 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [122…
```
ucs = uniform_cost_search(nqp).solution()
```

In [123…
```
plot_NQueens(ucs)
```



`depth_first_tree_search` is almost 20 times faster than `breadth_first_tree_search` and more than 200 times faster than `uniform_cost_search`.

We can also solve this problem using `astar_search` with a suitable heuristic function. The best heuristic function for this scenario will be one that returns the number of conflicts in the current state.

In [124…
```
psource(NQueensProblem.h)
```

```
    def h(self, node):
        """Return number of conflicting queens for a given node"""
        num_conflicts = 0
        for (r1, c1) in enumerate(node.state):
            for (r2, c2) in enumerate(node.state):
                if (r1, c1) != (r2, c2):
                    num_conflicts += self.conflict(r1, c1, r2, c2)

        return num_conflicts
```

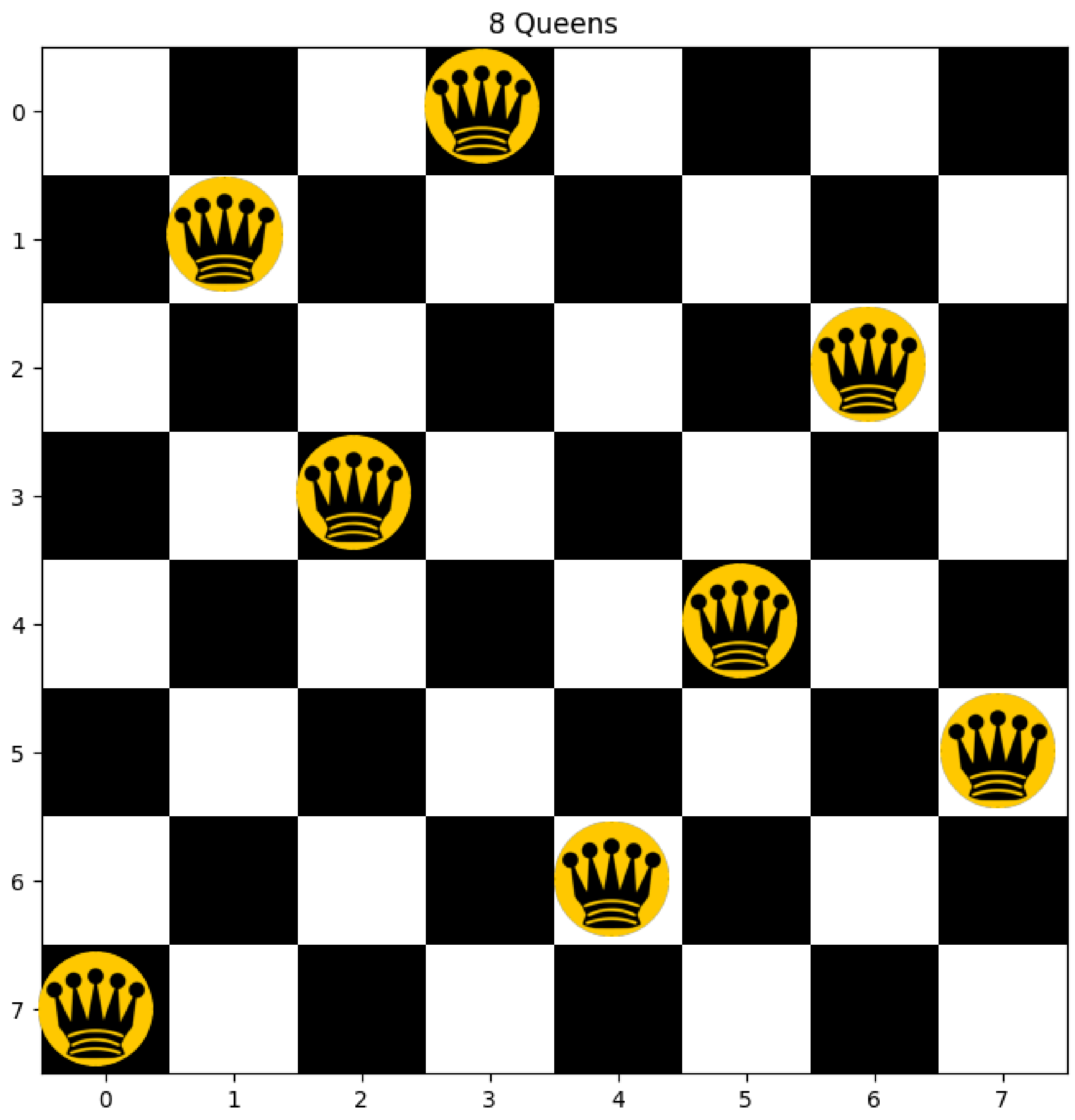
In [125…
```
%%timeit
astar_search(nqp)
```

```
987 µs ± 4.37 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

`astar_search` is faster than both `uniform_cost_search` and `breadth_first_tree_search`.

In [126…
```
astar = astar_search(nqp).solution()
```

In [127…
```
plot_NQueens(astar)
```



AND-OR GRAPH SEARCH

An *AND-OR* graph is a graphical representation of the reduction of goals to *conjunctions* and *disjunctions* of subgoals.
An *AND-OR* graph can be seen as a generalization of a directed graph. It contains a number of vertices and generalized edges that connect the vertices.
Each connector in an *AND-OR* graph connects a set of vertices V to a single vertex, v_0.
A connector can be an *AND* connector or an *OR* connector. An **AND** connector connects two edges having a logical *AND* relationship, while and **OR** connector connects two edges having a logical *OR* relationship.
A vertex can have more than one *AND* or *OR* connector. This is why *AND-OR* graphs can be expressed as logical statements.

AND-OR graphs also provide a computational model for executing logic programs and you will come across this data-structure in the `logic` module as well. *AND-OR* graphs can be searched in depth-first, breadth-first or best-first ways searching the state sapce linearly or parallely.
Our implementation of *AND-OR* search searches over graphs generated by non-deterministic environments and returns a conditional plan that reaches a goal state in all circumstances. Let's have a look at the implementation of `and_or_graph_search`.

In [128…

```
psource(and_or_graph_search)
```

```python
def and_or_graph_search(problem):
    """[Figure 4.11]Used when the environment is nondeterministic and completely observab
le.
    Contains OR nodes where the agent is free to choose any action.
    After every action there is an AND node which contains all possible states
    the agent may reach due to stochastic nature of environment.
    The agent must be able to handle all possible states of the AND node (as it
    may end up in any of them).
    Returns a conditional plan to reach goal state,
    or failure if the former is not possible."""

    # functions used by and_or_search
    def or_search(state, problem, path):
        """returns a plan as a list of actions"""
        if problem.goal_test(state):
            return []
        if state in path:
            return None
        for action in problem.actions(state):
            plan = and_search(problem.result(state, action),
                              problem, path + [state, ])
            if plan is not None:
                return [action, plan]

    def and_search(states, problem, path):
        """Returns plan in form of dictionary where we take action plan[s] if we reach state
s."""
        plan = {}
        for s in states:
            plan[s] = or_search(s, problem, path)
            if plan[s] is None:
                return None
        return plan

    # body of and or search
    return or_search(problem.initial, problem, [])
```

The search is carried out by two functions `and_search` and `or_search` that recursively call each other, traversing nodes sequentially. It is a recursive depth-first algorithm for searching an *AND-OR* graph.
A very similar algorithm `fol_bc_ask` can be found in the `logic` module, which

carries out inference on first-order logic knowledge bases using *AND-OR* graph-derived data-structures.
AND-OR trees can also be used to represent the search spaces for two-player games, where a vertex of the tree represents the problem of one of the players winning the game, starting from the initial state of the game.
Problems involving *MIN-MAX* trees can be reformulated as *AND-OR* trees by representing *MAX* nodes as *OR* nodes and *MIN* nodes as *AND* nodes.
`and_or_graph_search` can then be used to find the optimal solution. Standard algorithms like `minimax` and `expectiminimax` (for belief states) can also be applied on it with a few modifications.

Here's how `and_or_graph_search` can be applied to a simple vacuum-world example.

In [129…
```
vacuum_world = GraphProblemStochastic('State_1', ['State_7', 'State_8'], vac
plan = and_or_graph_search(vacuum_world)
```

In [130…
```
plan
```

Out[130…
```
['Suck',
 {'State_7': [], 'State_5': ['Right', {'State_6': ['Suck', {'State_8':
[]}]}]}]
```

In [131…
```
def run_plan(state, problem, plan):
    if problem.goal_test(state):
        return True
    if len(plan) is not 2:
        return False
    predicate = lambda x: run_plan(x, problem, plan[1][x])
    return all(predicate(r) for r in problem.result(state, plan[0]))
```

In [132…
```
run_plan('State_1', vacuum_world, plan)
```

Out[132…
```
True
```

ONLINE DFS AGENT

So far, we have seen agents that use **offline search** algorithms, which is a class of algorithms that compute a complete solution before executing it. In contrast, an **online search** agent interleaves computation and action. Online search is better for most dynamic environments and necessary for unknown environments.
Online search problems are solved by an agent executing actions, rather than just by pure computation. For a fully observable environment, an online agent cycles through three steps: taking an action, computing the step cost and checking if the goal has been reached.
For online algorithms in partially-observable environments, there is usually a tradeoff between exploration and exploitation to be taken care of.

Whenever an online agent takes an action, it receives a *percept* or an observation that tells it something about its immediate environment. Using this percept, the agent can augment its map of the current environment. For a partially observable environment, this is called the belief state.
Online algorithms expand nodes in a *local* order, just like *depth-first search* as it does not have the option of observing farther nodes like *A* search*. Whenever an action from the current state has not been explored, the agent tries that action.
Difficulty arises when the agent has tried all actions in a particular state. An offline search algorithm would simply drop the state from the queue in this scenario whereas an online search agent has to physically move back to the previous state. To do this, the agent needs to maintain a table where it stores the order of nodes it has been to. This is how our implementation of *Online DFS-Agent* works. This agent works only in state spaces where the action is reversible, because of the use of backtracking.
Let's have a look at the `OnlineDFSAgent` class.

In [133…
```
psource(OnlineDFSAgent)
```

```python
class OnlineDFSAgent:
    """
    [Figure 4.21]
    The abstract class for an OnlineDFSAgent. Override
    update_state method to convert percept to state. While initializing
    the subclass a problem needs to be provided which is an instance of
    a subclass of the Problem class.
    """

    def __init__(self, problem):
        self.problem = problem
        self.s = None
        self.a = None
        self.untried = dict()
        self.unbacktracked = dict()
        self.result = {}

    def __call__(self, percept):
        s1 = self.update_state(percept)
        if self.problem.goal_test(s1):
            self.a = None
        else:
            if s1 not in self.untried.keys():
                self.untried[s1] = self.problem.actions(s1)
            if self.s is not None:
                if s1 != self.result[(self.s, self.a)]:
                    self.result[(self.s, self.a)] = s1
                    self.unbacktracked[s1].insert(0, self.s)
            if len(self.untried[s1]) == 0:
                if len(self.unbacktracked[s1]) == 0:
                    self.a = None
                else:
                    # else a <- an action b such that result[s', b] = POP(unbacktracked[s'])
                    unbacktracked_pop = self.unbacktracked.pop(s1)
                    for (s, b) in self.result.keys():
                        if self.result[(s, b)] == unbacktracked_pop:
                            self.a = b
                            break
            else:
                self.a = self.untried.pop(s1)
        self.s = s1
        return self.a

    def update_state(self, percept):
        """To be overridden in most cases. The default case
        assumes the percept to be of type state."""
        return percept
```

It maintains two dictionaries `untried` and `unbacktracked`. `untried` contains nodes that have not been visited yet. `unbacktracked` contains the sequence of nodes that the agent has visited so it can backtrack to it later, if required. `s` and `a` store the state and the action respectively and `result` stores the final path or solution of the problem.
Let's look at another online search algorithm.

LRTA* AGENT

We can infer now that hill-climbing is an online search algorithm, but it is not very useful natively because for complicated search spaces, it might converge to the local minima and indefinitely stay there. In such a case, we can choose to randomly restart it a few times with different starting conditions and return the result with the lowest total cost. Sometimes, it is better to use random walks instead of random restarts depending on the problem, but progress can still be very slow.
A better improvement would be to give hill-climbing a memory element. We store the current best heuristic estimate and it is updated as the agent gains experience in the state space. The estimated optimal cost is made more and more accurate as time passes and each time the the local minima is "flattened out" until we escape it.
This learning scheme is a simple improvement upon traditional hill-climbing and is called *learning real-time A** or **LRTA***. Similar to *Online DFS-Agent*, it builds a map of the environment and chooses the best possible move according to its current heuristic estimates.
Actions that haven't been tried yet are assumed to lead immediately to the goal with the least possible cost. This is called **optimism under uncertainty** and encourages the agent to explore new promising paths. This algorithm might not terminate if the state space is infinite, unlike A* search.
Let's have a look at the `LRTAStarAgent` class.

In [134…
```
psource(LRTAStarAgent)
```

```python
class LRTAStarAgent:
    """ [Figure 4.24]
    Abstract class for LRTA*-Agent. A problem needs to be
    provided which is an instance of a subclass of Problem Class.

    Takes a OnlineSearchProblem [Figure 4.23] as a problem.
    """

    def __init__(self, problem):
        self.problem = problem
        # self.result = {}      # no need as we are using problem.result
        self.H = {}
        self.s = None
        self.a = None

    def __call__(self, s1):  # as of now s1 is a state rather than a percept
        if self.problem.goal_test(s1):
            self.a = None
            return self.a
        else:
            if s1 not in self.H:
                self.H[s1] = self.problem.h(s1)
            if self.s is not None:
                # self.result[(self.s, self.a)] = s1    # no need as we are using problem.output

                # minimum cost for action b in problem.actions(s)
                self.H[self.s] = min(self.LRTA_cost(self.s, b, self.problem.output(self.s, b),
                                                    self.H) for b in self.problem.actions(self.s))

            # an action b in problem.actions(s1) that minimizes costs
            self.a = min(self.problem.actions(s1),
                         key=lambda b: self.LRTA_cost(s1, b, self.problem.output(s1, b), self.H))

            self.s = s1
            return self.a

    def LRTA_cost(self, s, a, s1, H):
        """Returns cost to move from state 's' to state 's1' plus
        estimated cost to get to goal from s1."""
        print(s, a, s1)
        if s1 is None:
            return self.problem.h(s)
        else:
            # sometimes we need to get H[s1] which we haven't yet added to H
            # to replace this try, except: we can initialize H with values from problem.h
            try:
                return self.problem.c(s, a, s1) + self.H[s1]
            except:
                return self.problem.c(s, a, s1) + self.problem.h(s1)
```

`H` stores the heuristic cost of the paths the agent may travel to.
`s` and `a` store the state and the action respectively.
`problem` stores the problem definition and the current map of the environment is stored in `problem.result`.
The `LRTA_cost` method computes the cost of a new path given the current state `s`, the action `a`, the next state `s1` and the estimated cost to get from `s` to `s1` is extracted from `H`.

Let's use `LRTAStarAgent` to solve a simple problem. We'll define a new `LRTA_problem` instance based on our `one_dim_state_space`.

In [135…
```
one_dim_state_space
```

Out[135…
```
<search.Graph at 0x104412dd0>
```

Let's define an instance of `OnlineSearchProblem`.

In [136…
```
LRTA_problem = OnlineSearchProblem('State_3', 'State_5', one_dim_state_space
```

Now we initialize a `LRTAStarAgent` object for the problem we just defined.

In [137…
```
lrta_agent = LRTAStarAgent(LRTA_problem)
```

We'll pass the percepts `[State_3, State_4, State_3, State_4, State_5]` one-by-one to our agent to see what action it comes up with at each timestep.

In [138…
```
lrta_agent('State_3')
```

```
State_3 Right State_4
State_3 Left State_2
```

Out[138…
```
'Right'
```

In [139…
```
lrta_agent('State_4')
```

```
State_3 Right State_4
State_3 Left State_2
State_4 Right State_5
State_4 Left State_3
```

```
Out[139… 'Left'
```

```
In [140… lrta_agent('State_3')
```

```
State_4 Right State_5
State_4 Left State_3
State_3 Right State_4
State_3 Left State_2
```

```
Out[140… 'Right'
```

```
In [141… lrta_agent('State_4')
```

```
State_3 Right State_4
State_3 Left State_2
State_4 Right State_5
State_4 Left State_3
```

```
Out[141… 'Right'
```

If you manually try to see what the optimal action should be at each step, the outputs of the `lrta_agent` will start to make sense if it doesn't already.

```
In [142… lrta_agent('State_5')
```

There is no possible action for this state.

This concludes the notebook. Hope you learned something new!