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.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
```

"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

```
cd /content/drive/MyDrive/Colab Notebooks/aima-python-master
```

```
/content/drive/MyDrive/Colab Notebooks/aima-python-master
```

```
pip install qpsolvers #run it
```

```
Requirement already satisfied: qpsolvers in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: quadprog>=0.1.8 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: scipy>=1.2.0 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (
```

```
pip install ipythonblocks  #run it
```

```
Collecting ipythonblocks
  Downloading ipythonblocks-1.9.0-py2.py3-none-any.whl (13 kB)
Requirement already satisfied: notebook>=4.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipython>=4.0 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: requests>=1.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pexpect in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7/dist
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/lib/py
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter-client>=5.2.0 in /usr/local/lib/python3.7
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: nbformat in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7/dist
Requirement already satisfied: jupyter-core>=4.4.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-packa
```

```
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/
Requirement already satisfied: pyzmq>=13 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/l
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/di
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/dist
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.7/
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.7/di
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/python3
Requirement already satisfied: importlib-resources>=1.4.0 in /usr/local/lib/pyth
Requirement already satisfied: attrs>=17.4.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/di
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0 in
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dis
Requirement already satisfied: zipp>=3.1.0 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python
Installing collected packages: ipythonblocks
Successfully installed ipythonblocks-1.9.0
```

"python" is not defined (reportUndefinedVariable)

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subprocess

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
pip install -r requirements.txt #run it
```

```
Requirement already satisfied: keras-preprocessing>=1.1.1 in /usr/local/lib/pyth
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.7/d
Requirement already satisfied: gast>=0.2.1 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.7/dist
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.7/d
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/loca
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/loc
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/dis
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/li
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/p
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.7
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-
Collecting asgiref<4,>=3.3.2
  Downloading asgiref-3.5.0-py3-none-any.whl (22 kB)
```

```
  Downloading asgiref-3.5.0-py3-none-any.whl (22 kB)
Requirement already satisfied: sqlparse>=0.2.2 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.7/di
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.7/
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: qtpy in /usr/local/lib/python3.7/dist-packages (f
Building wheels for collected packages: image
  Building wheel for image (setup.py) ... done
  Created wheel for image: filename=image-1.5.33-py2.py3-none-any.whl size=19496
  Stored in directory: /root/.cache/pip/wheels/56/88/e6/897194cfe8c08a8b9afd881d
Successfully built image
Installing collected packages: pluggy, coverage, asgiref, tf-estimator-nightly,
  Attempting uninstall: pluggy
    Found existing installation: pluggy 0.7.1
    Uninstalling pluggy-0.7.1:
      Successfully uninstalled pluggy-0.7.1
  Attempting uninstall: coverage
    Found existing installation: coverage 3.7.1
    Uninstalling coverage-3.7.1:
      Successfully uninstalled coverage-3.7.1
  Attempting uninstall: pytest
    Found existing installation: pytest 3.6.4
    Uninstalling pytest-3.6.4:
      Successfully uninstalled pytest-3.6.4
ERROR: pip's dependency resolver does not currently take into account all the pa
datascience 0.10.6 requires coverage==3.7.1, but you have coverage 6.3.2 which i
datascience 0.10.6 requires folium==0.2.1, but you have folium 0.8.3 which is in
coveralls 0.5 requires coverage<3.999,>=3.6, but you have coverage 6.3.2 which i
Successfully installed asgiref-3.5.0 coverage-6.3.2 django-3.2.12 image-1.5.33 p
```

"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

```
from search import *
from notebook import psource, heatmap, gaussian_kernel, show_map, final_path_colors, d

# Needed to hide warnings in the matplotlib sections

import warnings
warnings.filterwarnings("ignore")
```

CONTENTS

- Genetic Algorithm

Conyent from given links has been used in this notebook:
https://www.uio.no/studier/emner/matnat/ifi/INF3490/h18/assignments/
https://classes.engr.oregonstate.edu/mime/fall2017/rob537/hw_samples/hw2_sample2.pdf

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.

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`.


```
%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import lines
import math

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.

```
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
```

"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)
Open in tab View source
%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`
View Problem (⌥F8) No quick fixes ava

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

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`
View Problem (⌥F8) No quick fixes ava

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.

```
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_c
        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:]]
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

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.

```
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):
```


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

```python
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.

```
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

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**.

```
romania_locations = romania_map.locations
print(romania_locations)
```

```
{'Arad': (91, 492), 'Bucharest': (400, 327), 'Craiova': (253, 288), 'Drobeta': (
```

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.

```
# 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 k2, v2 in

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.

```
show_map(romania_graph_data)
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

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.

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

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.

```
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(no
        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.

```
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: right + 1])
        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 state2  """
        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)
```

"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)
Open in tab View source
%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py
View Problem (⌥F8) No quick fixes ava

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.

```
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', 'Hir:
```

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

```
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] - coordinates_2
            distances[name_2][name_1] = np.linalg.norm(
                [coordinates_1[0] - coordinates_2[0], coordinates_1[1] - coordinates_2
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab   View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8)   No quick fixes ava

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.

```
def 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
    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.state))
        if problem.value(neighbor.state) <= problem.value(current.state):
          current.state = neighbor.state # Note that it is based on negative path cost
        iterations -= 1

    return current.state
```

An instance of the TSP_problem class will be created.

```
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.

```
hill_climbing(tsp)
```

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

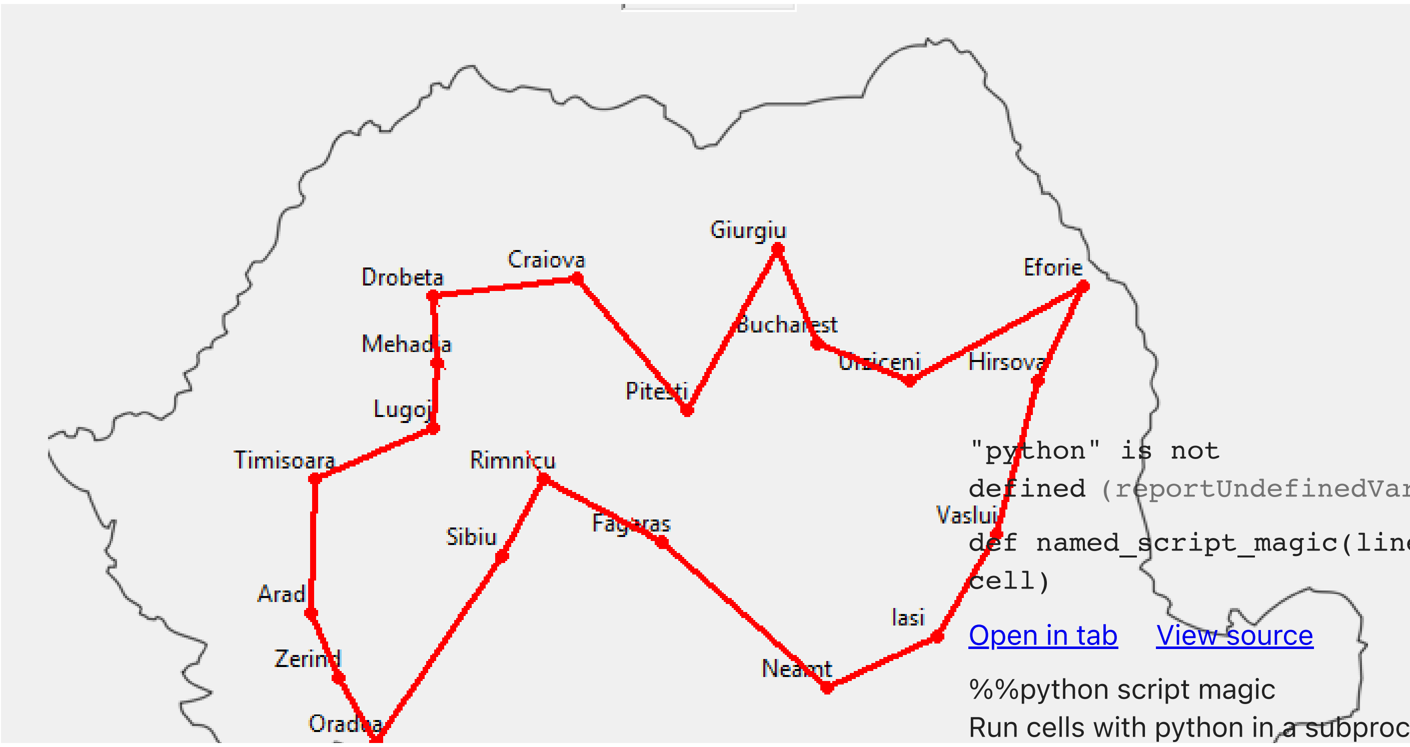
```
"python" is not
defined (reportUndefinedVari
def named_script_magic(line,
cell)
```

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

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



"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)
Open in tab View source
%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py
View Problem (⌥F8) No quick fixes ava

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.

```
def hill_climbing(problem):
    """From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better. [Figure 4.2]"""
    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
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current = neighbor
    return current.state


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


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(math.exp(delta_e / T)):
            current = next_choice
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab   View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py

View Problem (⌥F8)   No quick fixes ava

```
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(math.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.

```
psource(exp_schedule)
```

```
def exp_schedule(k=20, lam=0.005, limit=100):
    """One possible schedule function for simulated annealing"""
    return lambda t: (k * math.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.

```
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.

```
directions4
```

```
{'E': (1, 0), 'N': (0, 1), 'S': (0, -1), 'W': (-1, 0)}
```

Define a problem with these parameters.

```
problem = PeakFindingProblem(initial, grid, directions4)
```

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

```
solutions = {problem.value(simulated_annealing(problem)) for i in range(100)}
```

```
max(solutions)
```

```
9
```

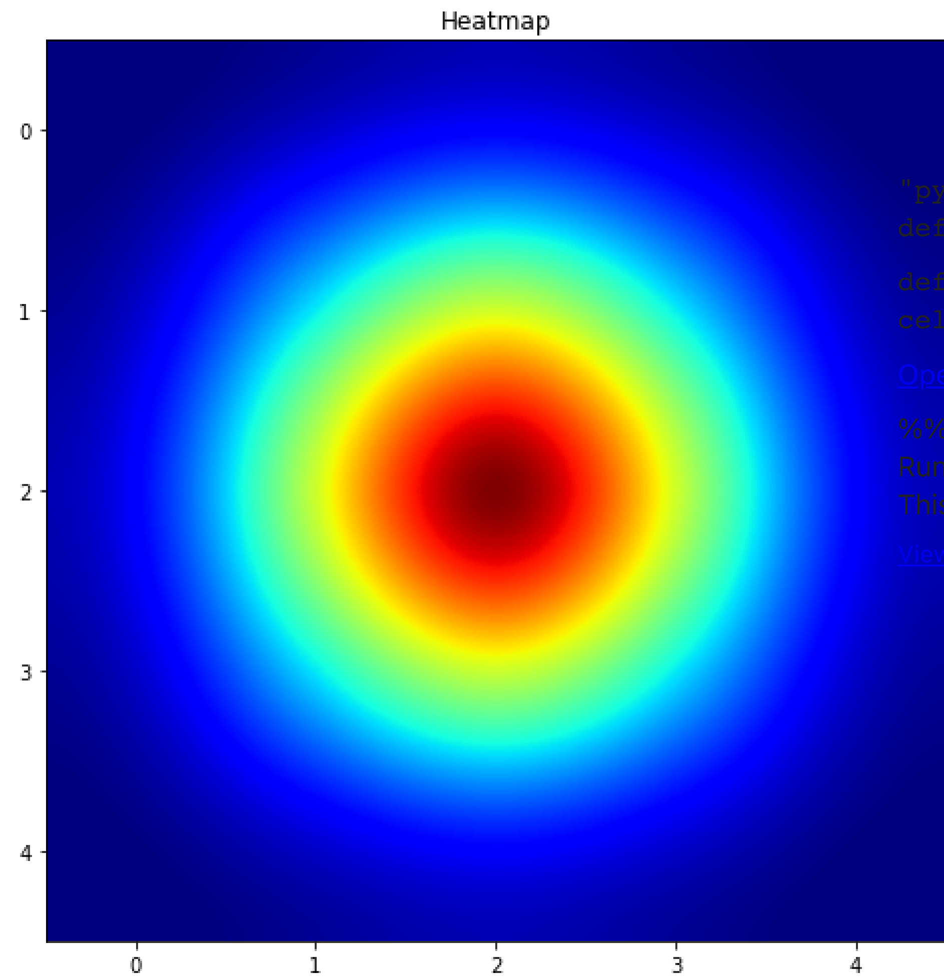
Hence, the maximum value is 9.

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.

```
grid = gaussian_kernel()
```

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

```
heatmap(grid, cmap='jet', interpolation='spline16')
```



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

```
directions8
```

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

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

```
problem = PeakFindingProblem(initial, grid, directions8)
```

```
%%timeit
solutions = {problem.value(simulated_annealing(problem)) for i in range(100)}
```

```
1 loop, best of 5: 275 ms per loop
```

```
max(solutions)
```

```
9
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

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

```
%%timeit
solution = problem.value(hill_climbing(problem))
```

```
10000 loops, best of 5: 114 µs per loop
```

```
solution = problem.value(hill_climbing(problem))
solution
```

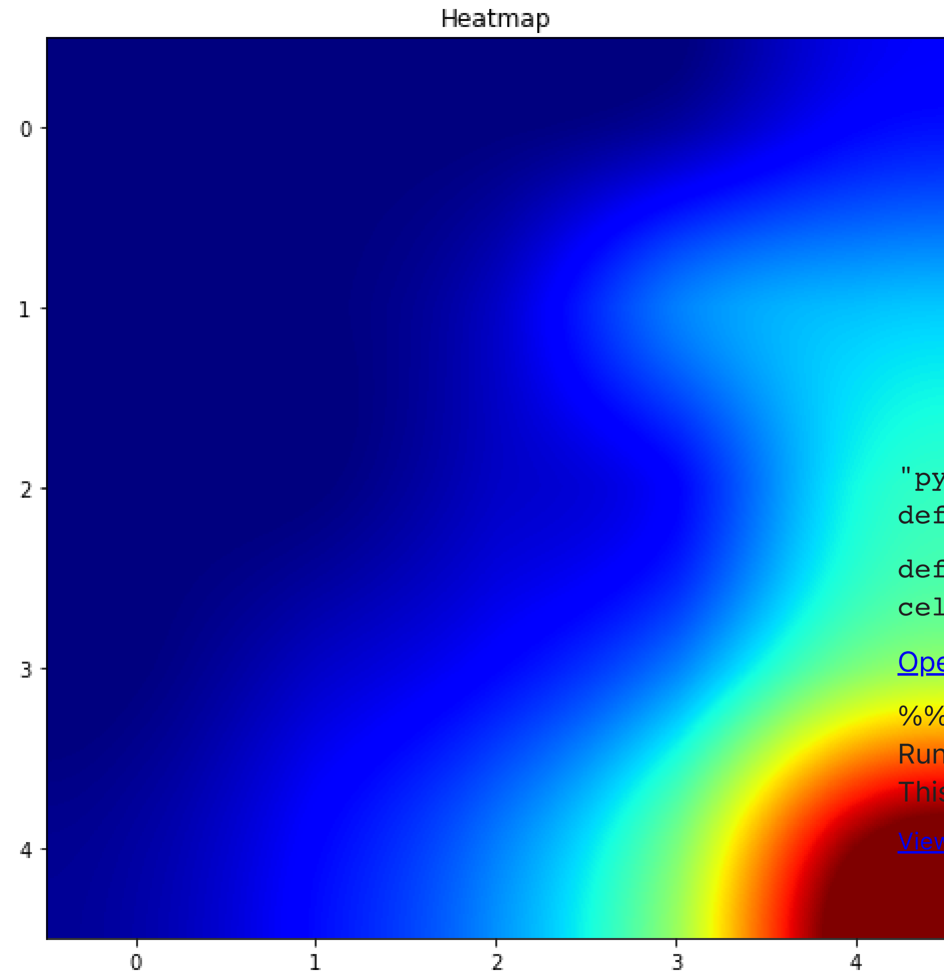
```
1.0
```

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.

```
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]]
```

```
heatmap(grid, cmap='jet', interpolation='spline16')
```



"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)
Open in tab View source
%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py
View Problem (⌥F8) No quick fixes ava

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.

```
problem = PeakFindingProblem(initial, grid, directions8)
```

Solution by Hill Climbing

```
solution = problem.value(hill_climbing(problem))
```

```
solution
```

0

Solution by Simulated Annealing

```
solutions = {problem.value(simulated_annealing(problem)) for i in range(100)}
max(solutions)
```

```
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:

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
psource(genetic_algorithm)
```

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

```
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`:


```
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`:

```
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_
        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

$$27^{17} = 2153693963075557766310747$$

"python" is not defined (reportUndefinedVari

`def named_script_magic(line, cell)`

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

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.

```
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.

```
# 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.

```
max_population = 100
```

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

```
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.

```
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
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

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.

```
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.

```
parents = select(2, population, fitness_fn)
```

```
# The recombine function takes two parents as arguments, so we need to unpack the prev
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.

```
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.

```
population = [mutate(recombine(*select(2, population, fitness_fn)), gene_pool, mutatic
```

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

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

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

Let's print this out

```
print(current_best)
```

```
['c', 'm', 'U', 'e', 'I', 'E', 'U', 'q', 'B', 'Z', 'p', 'o', 'I', 'o', 'O', 'D',
```

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

```
current_best_string = ''.join(current_best)
print(current_best_string)
```

```
cmUeIEUqBZpoIoODp
```

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

```
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 correct
# 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.

```
def genetic_algorithm_stepwise(population, fitness_fn, gene_pool=[0, 1], f_thres=None,
    for generation in range(ngen):
        population = [mutate(recombine(*select(2, population, fitness_fn)), gene_pool,
        # stores the individual genome with the highest fitness in the current populat
        current_best = ''.join(max(population, key=fitness_fn))
        print(f'Current best: {current_best}\t\tGeneration: {str(generation)}\t\tFitne

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

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

```
"python" is not defined (reportUndefinedVari
def named_script_magic(line,
cell)
Open in tab  View source
%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py
View Problem (⌥F8)  No quick fixes ava
```

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.

```
psource(genetic_algorithm)
```

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

```
population = init_population(max_population, gene_pool, len(target))
solution, generations = genetic_algorithm_stepwise(population, fitness_fn, gene_pool,
```

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.

[ ] ↳ *19 cells hidden*

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`.

```
psource(NQueensProblem)
```

```
"python" is not
defined (reportUndefinedVari
def named_script_magic(line,
cell)
```

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

```
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)
```

```
"python" is not
defined (reportUndefinedVari
def named_script_magic(line,
cell)
```

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

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.

```
    def conflicted(self, state, row, col):
```

```
nqp = NQueensProblem(8)
```

```
            for c in range(col))
```

Let's use `depth_first_tree_search` first.

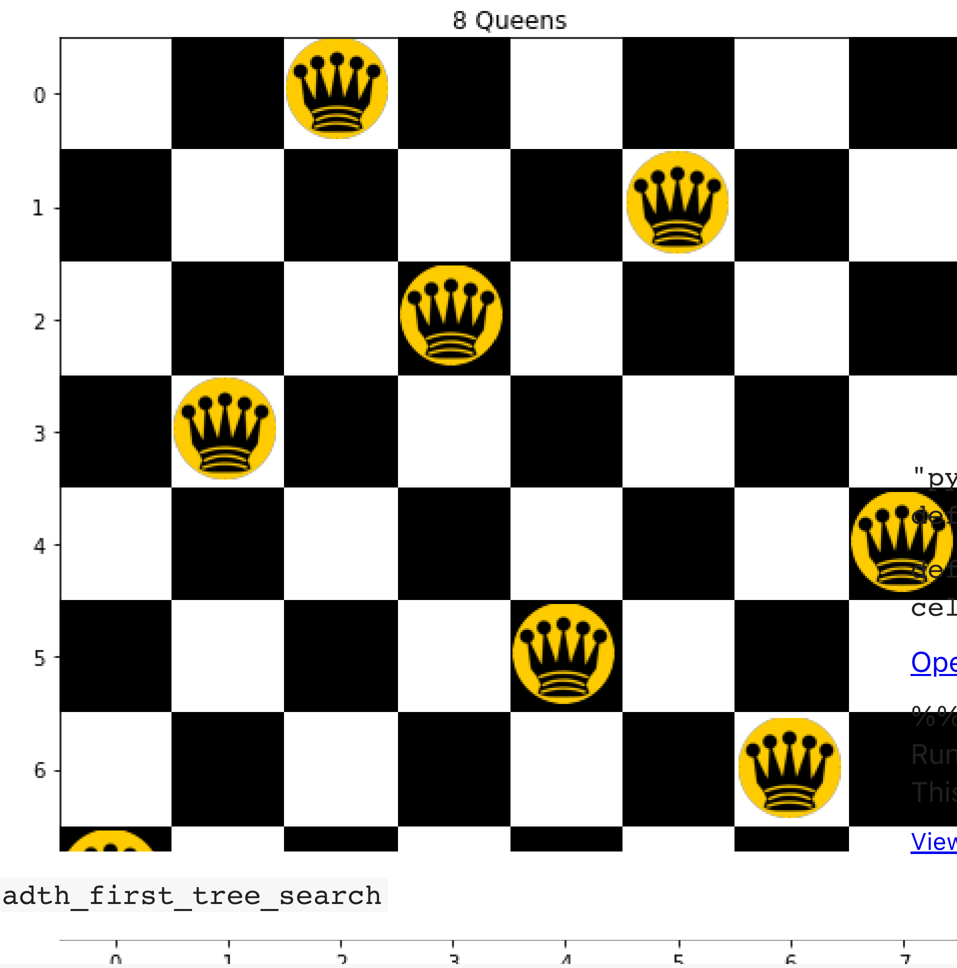
We will also use the %%timeit magic with each algorithm to see how much time they take.

```
%%timeit
depth_first_tree_search(nqp)
```

```
100 loops, best of 5: 2.28 ms per loop
```

```
dfts = depth_first_tree_search(nqp).solution()
```

```
plot_NQueens(dfts)
```



"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

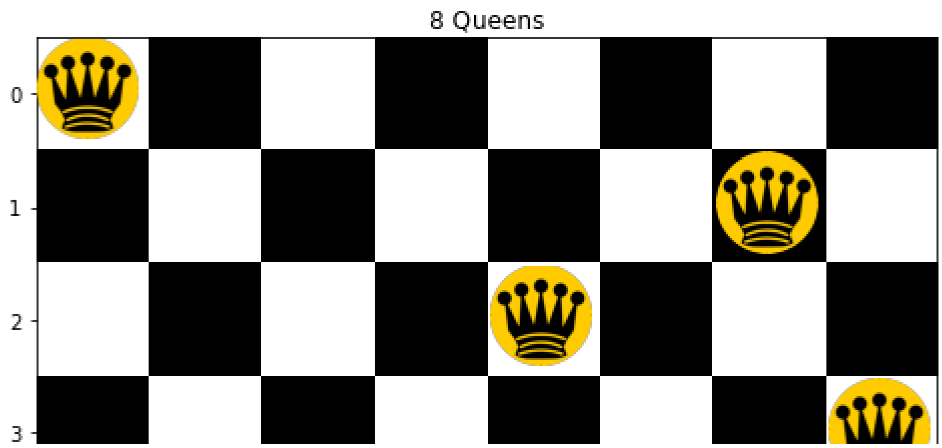
breadth_first_tree_search

```
%%timeit
breadth_first_tree_search(nqp)
```

```
10 loops, best of 5: 41.2 ms per loop
```

```
bfts = breadth_first_tree_search(nqp).solution()
```

```
plot_NQueens(bfts)
```



`uniform_cost_search`

"python" is not defined (reportUndefinedVari

```
%%timeit
uniform_cost_search(nqp)
```

def named_script_magic(line, cell)

Open in tab   View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for `%%script py`

View Problem (⌥F8)   No quick fixes ava

```
1 loop, best of 5: 337 ms per loop
```

```
ucs = uniform_cost_search(nqp).solution()
```

```
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.

```
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
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

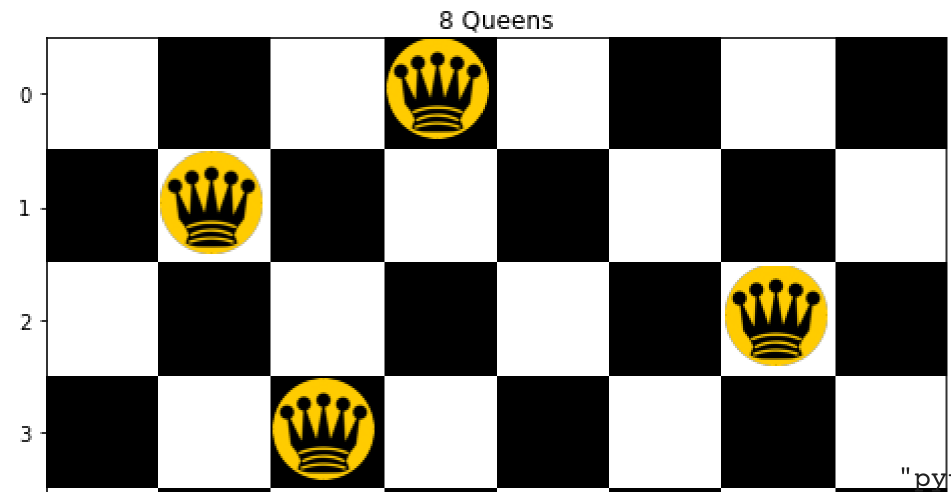
```
%%timeit
astar_search(nqp)
```

```
100 loops, best of 5: 4.15 ms per loop
```

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

```
astar = astar_search(nqp).solution()
```

```
plot_NQueens(astar)
```



Double-click (or enter) to edit

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

ICE - 4

Task 1 [20%]

Run the whole Romania_map code and analyze the code of following section:

Hill climbing

Simulated annealing

Genetic algorithm

N-Queens problem

Task 2 [30%]

Implement the given graphs for Hill climbing by modifying the romania_map code.

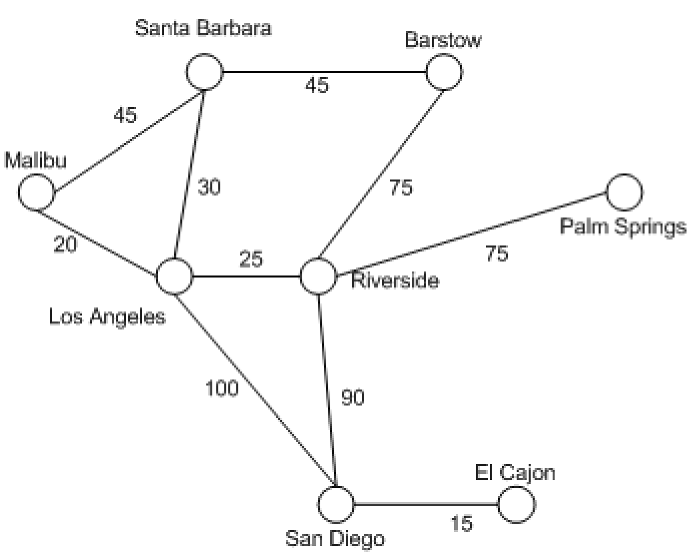


Fig 1: santa_barbara_map

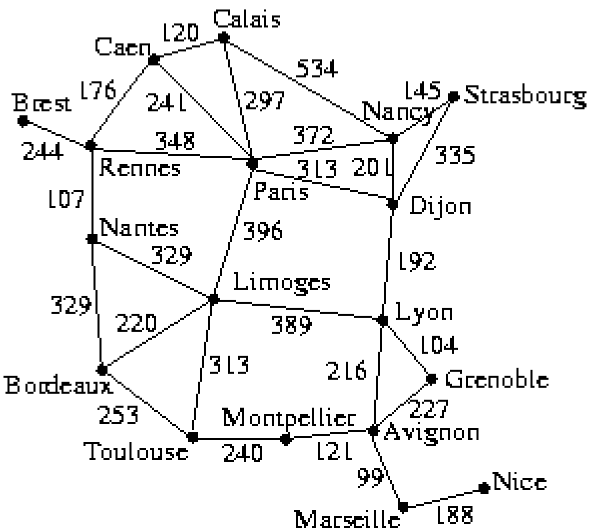


Fig 2: brest_map

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
#Implementing santabarbara graph with hill climbing method
```

```
santa_barbara_map = UndirectedGraph(dict(
    SantaBarbara=dict(Barstow=45, LosAngeles=30, Malibu=45),
    ElCajon=dict(SanDiego=15),
    Barstow=dict(SantaBarbara=45, Riverside=75),
    Riverside=dict(Barstow=75, PalmSprings=75, SanDiego=90, LosAngeles= 25),
    PalmSprings=dict(Riverside=75),
    SanDiego=dict(ElCajon=15, Riverside=90, LosAngeles=100 ),
    LosAngeles=dict(SanDiego=100, Riverside=25, SantaBarbara=30, Malibu=20),
    Malibu=dict(LosAngeles=20, SantaBarbara=45)))

santa_barbara_map.locations = dict(
    SantaBarbara=(90, 30), ElCajon=(250, 240), Barstow=(220, 30),
    Riverside=(150, 125), PalmSprings=(310, 80), SanDiego=(160, 250),
    LosAngeles=(90, 130), Malibu=(10, 90))
```

```
santa_barbara_problem = GraphProblem('SantaBarbara ', 'ElCajon', santa_barbara_map)
```

```
santa_barbara_locations = santa_barbara_map.locations
```

```
print(santa_barbara_locations)
```

```
{'SantaBarbara': (90, 30), 'ElCajon': (250, 240), 'Barstow': (220, 30), 'Riversi
```

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

santa_barbara_graph_data = {  'graph_dict' : santa_barbara_map.graph_dict,
                        'node_colors': node_colors,
                        'node_positions': node_positions,
                        'node_label_positions': node_label_pos,
                         'edge_weights': edge_weights
                     }
```

```
show_map(santa_barbara_graph_data)
```

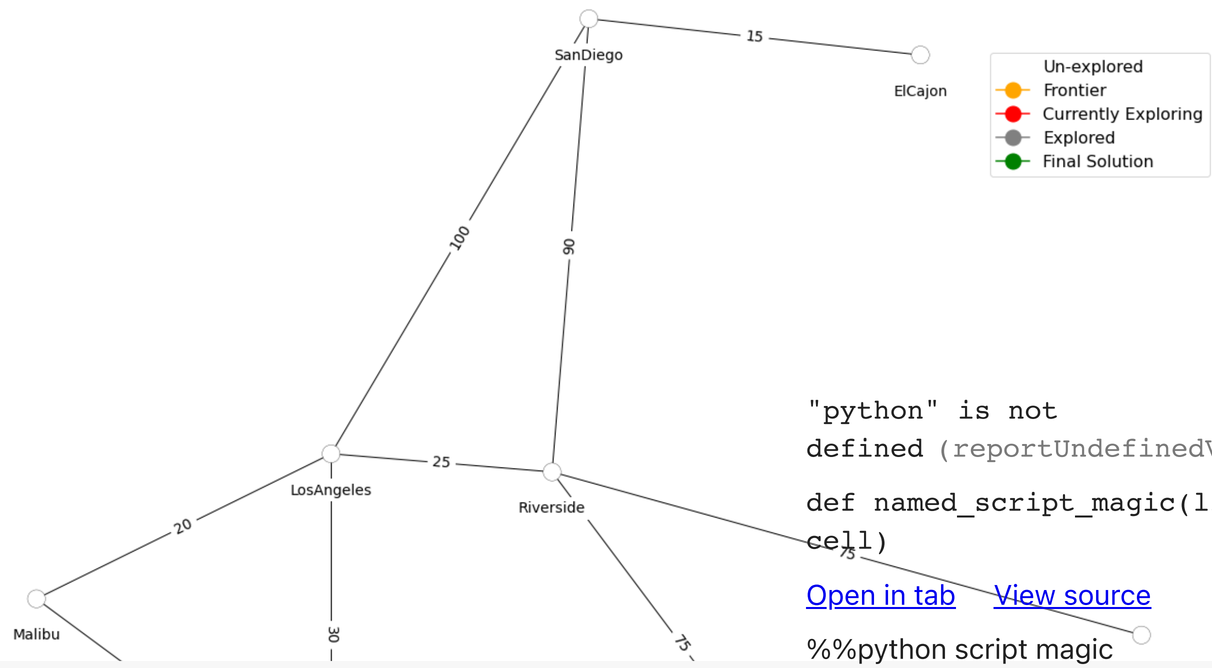
"python" is not defined (reportUndefinedVari

```
def named_script_magic(line, cell)
```

[Open in tab](#) [View source](#)

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

[View Problem (⌥F8)](#) No quick fixes ava



"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

[Open in tab](#) [View source](#)

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

[View Problem (⌥F8)](#) No quick fixes ava

```
distances = {}
all_cities_santa_barabara = []

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

all_cities_santa_barabara.sort()
print(all_cities_santa_barabara)
```

```
['Barstow', 'ElCajon', 'LosAngeles', 'Malibu', 'PalmSprings', 'Riverside', 'SanD
```

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

```
tsp = TSP_problem(all_cities_santa_barabara)
```

```
hill_climbing(tsp)
```

```
['Barstow',
 'ElCajon',
 'LosAngeles',
 'Malibu',
 'PalmSprings',
 'Riverside',
```

```
 'SanDiego',
 'SantaBarbara']
```

```
#Implementing brest_map with hill climbing method
```

```
brest_map = UndirectedGraph(dict(
    Brest=dict(Rennes=244),
    Rennes=dict(Caen=176, Paris=348, Nantes=107, Brest=244),
    Nantes=dict(Rennes=107, Limoges=329, Bordeaux=329),
    Bordeaux=dict(Nantes=329, Limoges= 220, Toulouse= 253),
    Toulouse=dict(Bordeaux=253, Limoges= 313, Montpellier= 240),
    Montpellier=dict(Toulouse=240, Avignon=121),
    Avignon=dict(Montpellier=121, Lyon=216, Grenoble=227, Marseille=99),
    Grenoble=dict(Avignon=227, Lyon=104),
    Lyon=dict(Grenoble=104, Avignon=216, Limoges= 389, Dijon=192),
    Dijon=dict(Lyon=192, Paris=313, Nancy=201, Strasbourg= 335),
    Strasbourg=dict(Dijon=335, Nancy=145 ),
    Nancy=dict(Strasbourg=145, Dijon=201, Paris=372, Calais= 534),
    Calais=dict(Nancy=534, Paris=297, Caen=120),
    Caen= dict(Calais=120, Paris=241, Rennes=176),
    Paris= dict(Caen=241, Calais= 297, Dijon= 313, Nancy=372, Rennes=348, Limoges=396)
    Limoges= dict(Paris=396, Nantes=329, Bordeaux=220, Toulouse=313, Lyon=389),
    Marseille= dict(Avignon=99, Nice=188),
    Nice= dict(Marseille=188)))

brest_map.locations = dict(
    Brest=(25, 77), Rennes=(58, 88), Nantes=(59, 133),
    Bordeaux=(64, 199), Toulouse=(108, 232), Montpellier=(156, 234),
    Avignon=(198, 230), Grenoble=(227, 205), Lyon=(203, 176),
    Dijon=(209, 117), Strasbourg=(238, 65), Nancy=(210, 84),
    Calais=(124, 35), Caen=(89, 44), Paris=(137, 97),
    Limoges=(119, 164), Marseille=(213, 267), Nice=(255, 259))
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
brest_problem = GraphProblem('Bordeaux', 'Strasbourg', brest_map)
```

```
brest_locations = brest_map.locations
print(brest_locations)
```

```
{'Brest': (25, 77), 'Rennes': (58, 88), 'Nantes': (59, 133), 'Bordeaux': (64, 19
```

```
node_colors = {node: 'white' for node in brest_map.locations.keys()}
node_positions = brest_locations
node_label_pos = { k:[v[0],v[1]-10]  for k,v in brest_locations.items() }
edge_weights = {(k, k2) : v2 for k, v in brest_map.graph_dict.items() for k2, v2 in v.

brest_graph_data = {  'graph_dict' : brest_map.graph_dict,
                      'node_colors': node_colors,
                      'node_positions': node_positions,
```

```
                            'node_label_positions': node_label_pos,
                             'edge_weights': edge_weights
                        }
```

```
show_map(brest_graph_data)
```

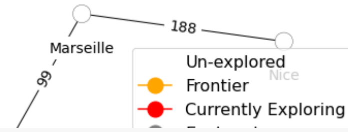
"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava




```
distances = {}
all_cities_brest_map = []

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

all_cities_brest_map.sort()
print(all_cities_brest_map)
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab \ View source

```
['Avignon', 'Bordeaux', 'Brest', 'Caen', 'Calais', 'Dijon', 'Grenoble', 'Limoges
```

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

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
tsp = TSP_problem(all_cities_brest_map)
```

```
hill_climbing(tsp)
```

```
['Avignon',
 'Bordeaux',
 'Brest',
 'Caen',
 'Calais',
 'Dijon',
 'Grenoble',
 'Limoges',
 'Lyon',
 'Marseille',
 'Montpellier',
 'Nancy',
 'Nantes',
 'Nice',
 'Paris',
 'Rennes',
 'Strasbourg',
 'Toulouse']
```

Task 3 [50%]

Solving TSP using Hill climbing:

In this exercise, you will attempt to solve an instance of the traveling salesman problem (TSP) using Hill climbing

In Hill climbing, a random sequence of cities, is generated. Then, all successor states of the solution are evaluated, where a successor state is obtained by switching the ordering of two cities adjacent in the solution. Finally, the best successor is chosen as the new state, and its successors are evaluated, and this continues until a satisfactory solution is obtained.

Here the traveling salesman, wants to go to the major cities in some region of the world in the shortest time possible, but is faced with the problem of finding the shortest tour among the cities. In this instance of the TSP, a number of European cities are to be visited. Their relative distances are given in the data file found at the csv file named "european_cities.csv".

```
"python" is not defined (reportUndefinedVari
def named_script_magic(line, cell)
```

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

Write a simple hill climber to solve the TSP. Run the algorithm several times to measure its performance. Report the length of the tour of the best, worst and mean of 10 runs (with random starting tours) both with the 6 first cities, 12 first cities, 18 first cities and with all 24 cities. Show the cities travelled each time and show the visualization.

Use python programming language and you must write your program from scratch

Read the .csv file in your notebook first. This is a sample picture of how the .csv file looks like.

	Barcelona	Belgrade	Berlin	Brussels	Bucharest	Budapest
Barcelona	0	1528.13	1497.61	1062.89	1968.42	1498.79
Belgrade	1528.13	0	999.25	1372.59	447.34	316.41
Berlin	1497.61	999.25	0	651.62	1293.40	689.06
Brussels	1062.89	1372.59	651.62	0	1769.69	1131.52
Bucharest	1968.42	447.34	1293.40	1769.69	0	639.77
Budapest	1498.79	316.41	689.06	1131.52	639.77	0

Extra materials and hint:

Pseudocode for Hill Climbing:

1: Start from a random state (random order of cities)

2: Generate all successors (all orderings obtained with switching any two adjacent cities)

3: Select successor with lowest total cost

4: Go to step 2

```
"python" is not
defined (reportUndefinedVari

def named_script_magic(line,
cell)
```

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

```
import numpy as np
import pandas as pd
import csv
from itertools import permutations, cycle
import time
import random
```

```
cd /content/drive/MyDrive/Colab
```

```
[Errno 2] No such file or directory: '/content/drive/MyDrive/Colab'
/content/drive/MyDrive/Colab Notebooks/aima-python-master
```

```
df = pd.read_csv('european_cities-1.csv')
```

```
df
```

	Barcelona;Belgrade;Berlin;Brussels;Bucharest;Budapest;Copenhagen;Dublin;Hamb
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab   View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for %%script py

View Problem (⌥F8)   No quick fixes ava

```
df = pd.read_csv('european_cities-1.csv', delimiter=';')
```

19

```
df
```

	Barcelona	Belgrade	Berlin	Brussels	Bucharest	Budapest	Copenhagen	Dublin
0	0.00	1528.13	1497.61	1062.89	1968.42	1498.79	1757.54	1469.29
1	1528.13	0.00	999.25	1372.59	447.34	316.41	1327.24	2145.39
2	1497.61	999.25	0.00	651.62	1293.40	689.06	354.03	1315.16
3	1062.89	1372.59	651.62	0.00	1769.69	1131.52	766.67	773.20
4	1968.42	447.34	1293.40	1769.69	0.00	639.77	1571.54	2534.72
5	1498.79	316.41	689.06	1131.52	639.77	0.00	1011.31	1894.95
6	1757.54	1327.24	354.03	766.67	1571.54	1011.31	0.00	1238.38
7	1469.29	2145.39	1315.16	773.20	2534.72	1894.95	1238.38	0.00
8	1471.78	1229.93	254.51	489.76	1544.17	927.92	287.97	1073.36
9	2230.42	809.48	1735.01	2178.85	445.62	1064.76	2017.17	2950.11
10	2391.06	976.02	1204.00	1836.20	744.44	894.29	1326.33	2513.69
11	1137.67	1688.97	929.97	318.72	2088.42	1450.12	955.13	462.60
12	504.64	2026.94	1867.69	1314.30	2469.71	1975.38	2071.75	1449.96
13	725.12	885.32	840.72	696.61	1331.46	788.56	1157.89	1413.37
14	3006.93	1710.99	1607.99	2253.26	1497.56	1565.19	1558.52	2792.41
15	1054.55	773.33	501.97	601.87	1186.37	563.93	838.00	1374.91
16	831.59	1445.70	876.96	261.29	1869.95	1247.61	1025.90	776.83
17	1353.90	738.10	280.34	721.08	1076.82	443.26	633.05	1465.61
18	856.69	721.55	1181.67	1171.34	1137.38	811.11	1529.69	1882.22

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
file = open('european_cities-1.csv', 'r+')
reader = csv.reader(file,delimiter=";")
distance = np.zeros((24,24))
cities = []

i = 0
for row in reader:
    #print row
    if i == 0:
        for j in range(len(row)):
            cities.append(row[j])
    else:
        for j in range(len(distance)):
```

```
            distance[i-1,j] = float(row[j])
    i += 1

cities
```

```
['Barcelona',
 'Belgrade',
 'Berlin',
 'Brussels',
 'Bucharest',
 'Budapest',
 'Copenhagen',
 'Dublin',
 'Hamburg',
 'Istanbul',
 'Kiev',
 'London',
 'Madrid',
 'Milan',
 'Moscow',
 'Munich',
 'Paris',
 'Prague',
 'Rome',
 'Saint Petersburg',
 'Sofia',
 'Stockholm',
 'Vienna',
 'Warsaw']
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

1. Start from a random state (random order of cities)

2: Generate all successors (all orderings obtained with switching any two adjacent cities)

3: Select successor with lowest total cost

```
def dist(cities,distance_matrix):
    tour = np.zeros(len(cities))
    for i in range(len(cities)-1):
        tour[i] = distance_matrix[int(cities[i]),int(cities[i+1])]
    tour[-1] = distance_matrix[int(cities[-1]), int(cities[0])]
    return np.sum(tour)
# taking random cities first
def hillclimb(cities, distance_matrix,neighbours):
    init = cities
    np.random.shuffle(init)

    best = dist(init,distance_matrix)
    best_neighbour = best.copy()
    count = 0
```

```
    best_sequence = init.copy()
    while count < 20:
      for i in range(neighbours):
        number1 = np.random.randint(N)
        number2 = np.random.randint(N)
 #Generate all successors (all orderings obtained with switching any two adjacent citi
        while number1 == number2:
            number2 = np.random.randint(N)
        value1 = init[number1]
        value2 = init[number2]
        init[number1] = value2
        init[number2] = value1
        temp_neighbour = dist(init,distance_matrix)
        if temp_neighbour < best_neighbour:
          best_neighbour = temp_neighbour


      if best_neighbour < best:
        best = best_neighbour
        best_sequence = init.copy()
        count = 0
      else:
        count += 1
  #3: Select successor with lowest total cost
    return best, best_sequence
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic
Run cells with python in a subproces
This is a shortcut for `%%script py`

View Problem (⌥F8) No quick fixes ava

```
def index_to_city_name(city_names, city_indices):
    best_cities = []
    for i in range(len(city_indices)):
      nums = city_indices[i]
      arr = []
      for n in nums:
        arr.append(city_names[int(n)])
      best_cities.append(arr)
    for arr in best_cities:

        return best_cities
```

Report the length of the tour of the best, worst and mean of 10 runs (with random starting tours) both with the 6 first cities, 12 first cities, 18 first cities and with all 24 cities.

```
# 1. 6 cities  and mean of 10 runs
No_cities = 6
No_runs = 10
No_neighbours = 5
city_vec = np.arange(No_cities )
best = np.zeros(No_runs)
best_sequence = np.zeros((No_runs,No_cities ))
```

```
start_time = time.time()

for i in range(No_runs):

    best[i], best_sequence[i,:] = hillclimb(city_vec,distance,No_neighbours)

elapsed_time = time.time() - start_time


print("The time for ", No_cities, " cities, is ", elapsed_time, " seconds.")
print("The best route is ", np.amin(best))
print("The worst route is ", np.amax(best))
print("The mean distance of the routes is ", np.mean(best))
print("The standard deviaton of the routes is ", np.std(best))
best_sequence
```

```
The time for  6  cities, is  0.028624534606933594  seconds.
The best route is  5018.8099999999995
The worst route is  5135.74
The mean distance of the routes is  5030.502999999999
The standard deviaton of the routes is  35.07900000000003
array([[1., 4., 0., 5., 3., 2.],
       [4., 0., 1., 3., 2., 5.],
       [3., 5., 1., 2., 4., 0.],
       [5., 4., 0., 3., 1., 2.],
       [5., 4., 2., 3., 0., 1.],
       [2., 3., 0., 1., 4., 5.],
       [0., 2., 3., 4., 1., 5.],
       [0., 2., 3., 5., 1., 4.],
       [2., 1., 0., 5., 3., 4.],
       [0., 1., 4., 3., 2., 5.]])
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell):

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
best_sequence
```

```
array([[1., 0., 3., 5., 4., 2.],
       [1., 4., 0., 3., 2., 5.],
       [5., 2., 3., 0., 1., 4.],
       [2., 1., 5., 0., 3., 4.],
       [3., 4., 2., 1., 5., 0.],
       [5., 2., 1., 0., 3., 4.],
       [2., 3., 0., 5., 4., 1.],
       [5., 3., 0., 1., 2., 4.],
       [4., 3., 2., 0., 1., 5.],
       [3., 4., 0., 2., 5., 1.]])
```

```
# 2. 12 cities  and mean of 10 runs
No_cities = 12
No_runs = 10
No_neighbours = 5
city_vec = np.arange(No_cities )
best = np.zeros(No_runs)
```

```
best_sequence = np.zeros((No_runs,No_cities ))

start_time = time.time()

for i in range(No_runs):

    best[i], best_sequence[i,:] = hillclimb(city_vec,distance,No_neighbours)

elapsed_time = time.time() - start_time


print("The time for ", No_cities, " cities, is ", elapsed_time, " seconds.")
print("The best route is ", np.amin(best))
print("The worst route is ", np.amax(best))
print("The mean distance of the routes is ", np.mean(best))
print("The standard deviaton of the routes is ", np.std(best))
best_sequence
```

```
The time for  12  cities, is  0.0776054859161377  seconds.
The best route is  10150.67
The worst route is  15034.0
The mean distance of the routes is  13001.927999999998
The standard deviaton of the routes is  1433.4943471796466
array([[ 5.,  0., 10.,  2., 11.,  9.,  4.,  6.,  7.,  8.,  3.,  1.],
       [ 2.,  0.,  1.,  3., 11.,  9., 10.,  7.,  5.,  8.,  4.,  6.],
       [ 4., 10.,  2.,  3.,  0.,  7.,  6.,  1.,  8., 11.,  9.,  5.],
       [11.,  4.,  8.,  7.,  0.,  3., 10.,  2.,  9.,  1.,  5.,  6.],
       [ 8.,  6.,  5., 10.,  3.,  9., 11.,  1.,  0.,  4.,  7.,  2.],
       [ 1., 11.,  5.,  6.,  7., 10.,  8.,  2.,  3.,  0.,  4.,  9.],
       [10.,  8.,  7., 11.,  9.,  5.,  2.,  4.,  1.,  0.,  3.,  6.],
       [ 0.,  5.,  6.,  1.,  9.,  3.,  8.,  7.,  2., 10.,  4., 11.],
       [ 8.,  4.,  1.,  3.,  7., 10.,  2.,  5.,  0., 11.,  9.,  6.],
       [ 3.,  7.,  6., 10.,  4.,  1.,  2., 11.,  5.,  0.,  9.,  8.]])
```



```
# 3. 18 cities  and mean of 10 runs
No_cities = 18
No_runs = 10
No_neighbours = 5
city_vec = np.arange(No_cities )
best = np.zeros(No_runs)
best_sequence = np.zeros((No_runs,No_cities ))

start_time = time.time()

for i in range(No_runs):

    best[i], best_sequence[i,:] = hillclimb(city_vec,distance,No_neighbours)

elapsed_time = time.time() - start_time


print("The time for ", No_cities, " cities, is ", elapsed_time, " seconds.")
print("The best route is ", np.amin(best))
```

```
print("The worst route is ", np.amax(best))
print("The mean distance of the routes is ", np.mean(best))
print("The standard deviaton of the routes is ", np.std(best))
best_sequence
```

```
The time for  18  cities, is  0.04636073112487793  seconds.
The best route is  17434.38
The worst route is  23464.16
The mean distance of the routes is  20326.917
The standard deviaton of the routes is  1983.7282353893631
array([[ 6.,  2., 15., 13., 11.,  1.,  4.,  9.,  5., 10.,  3., 17., 14.,
        12., 16.,  0.,  7.,  8.],
       [10., 14.,  2.,  6.,  7., 11.,  3.,  8., 12.,  1.,  4.,  0., 16.,
        15., 13., 17.,  9.,  5.],
       [ 7.,  9., 14.,  4.,  2.,  1., 17.,  5., 10.,  8., 15.,  3.,  0.,
        11., 16., 13., 12.,  6.],
       [ 1.,  5., 17., 10., 16.,  2., 14.,  0.,  4.,  7., 12., 15., 13.,
         3., 11.,  9.,  8.,  6.],
       [17.,  5., 11., 13., 16.,  4., 10.,  2.,  8.,  7., 12.,  1.,  6.,
         0.,  9.,  3., 14., 15.],
       [ 0.,  7.,  1.,  5., 16.,  9., 10.,  6., 12., 11.,  8., 17., 14.,
         4., 15.,  2., 13.,  3.],
       [ 5., 14.,  6., 13.,  1., 15.,  8., 16., 10., 11.,  0.,  4.,  3.,
         2.,  9., 17., 12.,  7.],
       [ 4., 14., 10.,  3., 16.,  6.,  7.,  9.,  2., 17.,  8., 12., 11.,
         5., 13.,  0., 15.,  1.],
       [ 9., 12., 14., 17., 10.,  3.,  6., 13.,  1.,  2., 16.,  4., 11.,
         0.,  7.,  5.,  8., 15.],
       [ 9., 14.,  8.,  2., 17.,  6., 11., 10.,  4.,  3., 16.,  7., 15.,
         0.,  1.,  5., 12., 13.]])
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

```
# 4. 24 cities  and mean of 10 runs
No_cities = 24
No_runs = 10
No_neighbours = 5
city_vec = np.arange(No_cities )
best = np.zeros(No_runs)
best_sequence = np.zeros((No_runs,No_cities ))

start_time = time.time()

for i in range(No_runs):

    best[i], best_sequence[i,:] = hillclimb(city_vec,distance,No_neighbours)

elapsed_time = time.time() - start_time


print("The time for ", No_cities, " cities, is ", elapsed_time, " seconds.")
print("The best route is ", np.amin(best))
print("The worst route is ", np.amax(best))
print("The mean distance of the routes is ", np.mean(best))
print("The standard deviaton of the routes is ", np.std(best))
```

```
print("The standard deviaton of the routes is ", np.std(best))
best_sequence
```

```
The time for  24  cities, is  0.2030317783355713  seconds.
The best route is  22282.230000000003
The worst route is  32838.369999999995
The mean distance of the routes is  28092.737
The standard deviaton of the routes is  2894.69133060176
array([[12., 13., 23.,  5., 14., 17., 11.,  2.,  8., 15., 18., 16.,  1.,
        20., 10., 19.,  6.,  4., 21.,  7.,  0.,  9., 22.,  3.],
       [ 0.,  1., 15., 19., 12., 21.,  9.,  3., 22., 17., 16.,  6., 18.,
        23., 10.,  5., 14.,  7., 13.,  2., 20.,  4., 11.,  8.],
       [ 6., 20., 10.,  7., 18.,  8., 13., 17.,  3.,  2., 12., 15., 11.,
         4.,  0., 14., 16.,  9., 19., 22., 21., 23.,  1.,  5.],
       [11.,  4.,  1.,  6., 17.,  0., 18., 23., 22., 19.,  8.,  9., 20.,
        21., 14.,  3., 16., 15., 12.,  2., 13.,  7., 10.,  5.],
       [ 5.,  0., 14.,  6., 15., 23., 22.,  7.,  1., 18., 17.,  2.,  4.,
        21., 11.,  3.,  8., 16., 20., 13., 12., 19., 10.,  9.],
       [ 0., 17., 19., 10., 23.,  4.,  1., 22., 20., 21.,  8.,  3., 12.,
         9.,  5., 18., 13., 16., 15.,  7., 14.,  6., 11.,  2.],
       [10.,  4., 13.,  9., 14.,  8.,  2., 11.,  5., 22.,  3., 15., 18.,
        16., 12.,  6.,  1., 23., 21., 19., 17., 20.,  0.,  7.],
       [21.,  7.,  8.,  5., 18.,  4., 22., 23., 15., 20., 11.,  1., 12.,
        19., 10.,  0.,  6., 16., 17.,  2.,  3.,  9., 14., 13.],
       [10., 23.,  5., 22., 18., 13., 12.,  0., 11.,  6., 15.,  8., 20.,
         9.,  1., 17.,  7.,  2.,  3., 16., 21., 14.,  4., 19.],
       [16.,  1., 21.,  6., 11., 19., 14.,  8., 20., 13., 10., 18.,  3.,
        15., 12.,  0.,  5.,  7.,  9., 22.,  4.,  2., 17., 23.]])
```

"python" is not defined (reportUndefinedVari

def named_script_magic(line, cell)

Open in tab View source

%%python script magic

Run cells with python in a subproces

This is a shortcut for %%script py

View Problem (⌥F8) No quick fixes ava

✓ 0s completed at 11:23 PM