ASSIGNMENT 1
IT5005 ARTIFICIAL INTELLIGENCE
GROUP AG 39

by

WONG JI FONG
ROHITH REDDY SANKURU
SWETA MISHRA

Contents

List of source codes

List of Figures

List of Tables

Chapter 1

Assignment Q1-1

1.1 Overview of the problem

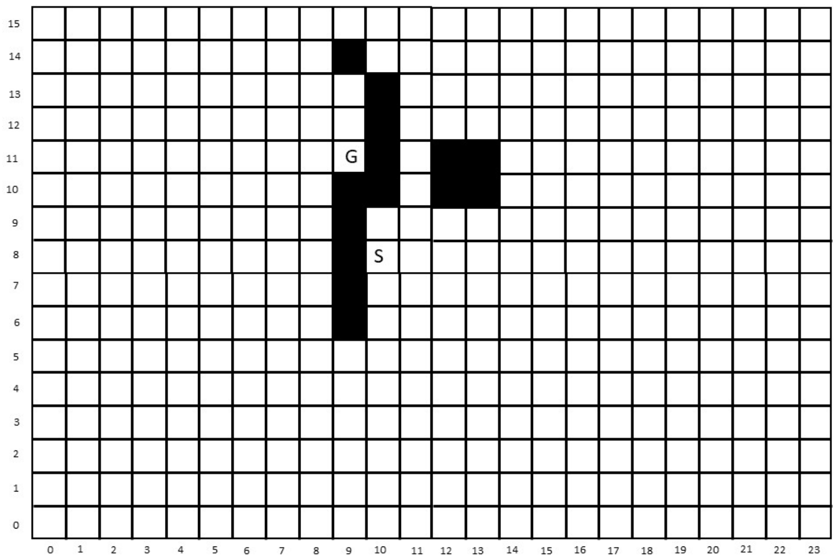

Figure 1.1: Example Maze

Assignment Q1 generally asks students to solve a maze with different algorithms. The maze in question is then solved with the following algorithms

- Breadth-First Search
- Depth-First Search with Cycle-Check

- Iterative-Deepening Search with Cycle-Check
- Uniform-Cost Search
- A* Search
- Greedy Best-first Search

The algorithms are to be evaluated against Figure 1.1 Maze in turn, with respect to nodes generated, number of nodes expanded, maximum frontier size, and path-cost for each entry. In addition, the performance of informed search algorithms are to be evaluated with proposed heuristics.

After, each heuristic is to be evaluated in terms of performance and the data are to be displayed in a visual format.

1.2 Defining the problem class

The problem class was given with the example class below:

```
class Problem:
    """The abstract class for a formal problem. A new domain subclasses this,
    overriding `actions` and `results`, and perhaps other methods.
    The default heuristic is 0 and the default action cost is 1 for all states.
    When you create an instance of a subclass, specify `initial`, and `goal`
↪   states
    (or give an `is_goal` method) and perhaps other keyword args for the
↪   subclass."""

    def __init__(self, initial=None, goal=None, **kwds):
        self.__dict__.update(initial=initial, goal=goal, **kwds)

    def actions(self, state):        raise NotImplementedError
    def result(self, state, action): raise NotImplementedError
    def is_goal(self, state):        return state == self.goal
    def action_cost(self, s, a, s1): return 1
    def h(self, node):               return 0

    def __str__(self):
        return '{}({!r}, {!r})'.format(
            type(self).__name__, self.initial, self.goal)
```

Source code 1.2.1: Problem class

Problem Class 1.2.1 demonstrates the abstract data class to be redefined. We will get to this after defining the Node class.

1.3 Defining the Node Class

Similar to the Problem class, the Node Class has been defined above. However this is to be extended directly as so.

```
# Use the following Node class to generate search tree
import math

class Node:
    "A Node in a search tree."
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.__dict__.update(
            state=state,
            parent=parent,
            action=action,
            path_cost=path_cost
        )

    def __len__(self): return 0 if self.parent is None else (1 +
    ↪  len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost

    # newly defined to assist with trace, and repr redefined
    # for readability
    def __repr__(self):
        """
        Newly defined to assist with trace, and track
        available states throughout the document
        """
        return '<state:(x:{},y:{}) path_cost:{} action:{}>'\
                .format(self.state[0], self.state[1], self.path_cost,
                ↪  self.action)

    def __eq__(self, other: Node) -> bool:
```

```
        """
        Tests for equality between two instances of a Node.
        Nodes with different states or different path costs
        are now defined to be treated seperately.
        Args:
            other (Node):
                Node to compare with (implicitly executed)
                by python when doing comparisons such as node1 == node2
        """
        return self.state == other.state

    def expand_node(self, state: NewState) -> Node:
        """
        Expands on a single state. Method takes a new state and expands
        by instantiating a new node instance and assigning the
        costs as the costs so far, and saving the action taken from
        parent_node -> child_node as child.action.

        Args:
            state (NewState):
                Expects a namedtuple as defined in the cell above.

        Returns:
            Node: Expanded child node
        """
        expanded_node = Node(state.coordinates)
        expanded_node.parent = self
        expanded_node.path_cost = self.path_cost + state.cost
        expanded_node.action = state.action
        return expanded_node

    def expand(
            self,
            permissable_actions: List[NewState]
        ) -> List[Node]:
        """
        Args:
            permissable_actions (List[NewState]):
```

```
                List of possible actions to take

        Returns:
            Generator[Node]:
                Generator of child nodes for this node.
        """
        return map(self.expand_node, permissable_actions)

    def __bool__(self):
        """
        Assist with assigning truthyness to the Node class
        e.g. is node == True if state is truthy
        """
        return True if self.state else False

    def __hash__(self):
        """
        Allows this class to be used in objects that require
        hashes such as sets, keys in dicts.
        """
        return hash(self.state)
```

Source code 1.3.2: Node Class

We will walk through the dunder methods added.

For `__repr__`,

```
def __repr__(self):
    """
    Newly defined to assist with trace, and track
    available states throughout the document
    """
    return '<state:(x:{},y:{}) path_cost:{} action:{}>'\
            .format(self.state[0], self.state[1], self.path_cost, self.action)
```

This assists with debugging by showing the state of each node, their associated path cost, and the action taken to have gotten from the previous node to this node.

For `__eq__`,

```python
def __eq__(self, other: Node) -> bool:
        """
        Tests for equality between two instances of a Node.
        Nodes with different states or different path costs
        are now defined to be treated seperately.
        Args:
            other (Node):
                Node to compare with (implicitly executed)
                by python when doing comparisons such as node1 == node2
        """
        return self.state == other.state
```

This defines the behaviour when an instantiated node class is compared to another node class. More specifically, it returns the result of comparing a `Node.state` and `OtherNode.state`.

For `expand_node()`,

```python
def expand_node(self, state: NewState) -> Node:
    """
    Expands on a single state. Method takes a new state and expands
    by instantiating a new node instance and assigning the
    costs as the costs so far, and saving the action taken from
    parent_node -> child_node as child.action.

    Args:
        state (NewState):
            Expects a namedtuple as defined in the cell above.

    Returns:
        Node: Expanded child node
    """
    expanded_node = Node(state.coordinates)
    expanded_node.parent = self
    expanded_node.path_cost = self.path_cost + state.cost
    expanded_node.action = state.action
    return expanded_node
```

The behaviour is defined to create a node, initialising it with the new coordinates given by the NamedTuple called state. It is then set to have its parent as this node, path cost so far, and action taken to get to the node thereafter. Afterwards, it is returned from the method.

For `expand()`,

```
def expand(
        self,
        permissable_actions: List[NewState]
    ) -> List[Node]:
    """
    Args:
        permissable_actions (List[NewState]):
            List of possible actions to take

    Returns:
        Generator[Node]:
            Generator of child nodes for this node.
    """
    return map(self.expand_node, permissable_actions)
```

This method takes a list of permissable actions (which is defined by the problem as actions that can be taken by the agent), then apply the § 1.3 `expand_node` method to each one. This returns the expansions as generator function to save on memory.

For `__bool__`,

```
def __bool__(self):
    """
    Assist with assigning truthyness to the Node class
    e.g. is node == True if state is truthy
    """
    return True if self.state else False
```

This dunder defines the behaviour when it is tested for truthiness. In the following trivial example, `if Node: print(True)`, should run and True should be printed.

And finally, for `__hash__`,

```
def __hash__(self):
    """
    Allows this class to be used in objects that require
    hashes such as sets, keys in dicts.
    """
    return hash(self.state)
```

This dunder defines that the class should be hashed with the state in order to be able to be used as dict keys or in sets.

This concludes the definition of the Node class. With this, we can continue to use this as the basis for defining the maze class.

1.4 Defining the Maze class

The Maze class inherits from the parent class, Problem, and thus retains certain attributes and methods from its parent class. In our version of the Maze class, we ask user to also provide the boundaries of the maze as well as a defined `action_cost_map` in order to be able to run the algorithm with the modified problem as given in the problem statement.

```
class Maze(Problem):
    def __init__(self,
                 initial: Node,
                 goal: Node,
                 boundaries: Tuple[int, int],
                 action_cost_map: Optional[dict],
                 **kwds):
        """
        Add type hints and parameter to know boundaries
        given the assumption "Assume that the agent knows
        the boundaries of the maze and has full observability"

        Args:
            initial (Node): Node for the initial state
            goal (Node): Node for the goal state
            boundaries (Tuple[int,int]):
                (rows, cols) of boundaries, assuming (0,0) to (rows,cols)
```

```
                as problem space
            """
            super().__init__(initial=initial,
                             goal=goal,
                             boundaries=boundaries,
                             action_cost_map=action_cost_map,
                             **kwds)

        def action_cost(self, node: Node, action: str) -> int:
            """
            Args:
                node (Node): Current node state
                action (str): Action to take

            Returns:
                int: Cost (s, a, s')
            """
            return self.action_cost_map[action]

        def _transform_permissable_action(self,
                                          actions: Tuple[Tuple[int, int], str],
                                          node: Node) -> Node:
            state, action = actions
            action_cost = self.action_cost(node, action)
            return NewState(action_cost, state, action)

        def actions(self, node: Node) -> List[NewState]:
            """
            Return permissable actions as list of actions
            as (COST, s', ACTION)
            Args:
                node (Node): Agents current state node

            Returns:
                List[NewState]:
                    List of permissable states and actions to expand to
                    for the given node.
            """
```

```
        x = node.state[0]
        y = node.state[1]
        parent = node.parent.state if node.parent else None

        actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']

        permissable_x_y = {((x+1, y), 'UP'),
                           ((x-1, y), 'DOWN'),
                           ((x, y+1), 'RIGHT'),
                           ((x, y-1), 'LEFT')}

        return list(
                    map(
                        lambda actions:
                        ↪  self._transform_permissable_action(actions, node),
                        filter(
                                lambda result:
                                   # Ensuring they permissable actions are:
                                   # * within boundaries
                                   # * not a shaded region
                                   0<=result[0][0]<self.boundaries[0] and
                                   0<=result[0][1]<self.boundaries[1] and
                                   (result[0][0], result[0][1]) not in
                                   ↪  self.shaded_regions,
                               permissable_x_y
                        )
                    )
            )

    def result(self, state: Node, action: str) -> Node:
        """
        Gives the state as a result of taking the action
        specified
        Args:
            state (Node): Node state
            action (str): Action to take from node s

        Returns:
```

```
            Tuple[int, int]:
                New state / coordinate
        """
        permissable_x_y = {
            'UP': (1, 0),
            'DOWN': (-1, 0),
            'RIGHT': (0, 1),
            'LEFT': (0, -1)
        }
        change = permissable_x_y[action]
        return ((state.state[0] + change[0],
                state.state[1] + change[1]), action)

    def h(self, node: Node) -> Union[float, int]:
        """
        Implementing Manhatten distance.
        One of two of the heuristics for Q1.a

        Args:
            node (Node): Current agent node

        Returns:
            Union[float, int]: heuristic value
        """
        return abs(self.goal.state[0]-node.state[0]) + \
                abs(self.goal.state[1]-node.state[1])

    def h2(self, node: Node) -> Union[float, int]:
        """
        Implementing Euclidean distance
        One of two of the heuristics for Q1.a
        Args:
            node (Node): Current agent node

        Returns:
            Union[float, int]: heuristic value
        """
        return ((self.goal.state[0]-node.state[0])**2 +
```

```
                (self.goal.state[1]-node.state[1])**2)**(1/2)



    def __repr__(self):
        """
        Assigning a readable representation of the object.
        """
        return '{}({!r}, {!r})'.format(
            type(self).__name__, self.initial, self.goal)

    def is_cycle(self, node: Node) -> bool:
        """
        Args:
            node (Node): Checks if a node is cyclic along its parent nodes

        Returns:
            bool: True if it is cyclic, else False.
        """
        visited = set()
        state = node.state
        while node.parent:
            if node.parent.state == state:
                return True
            node = node.parent
        return False

    def is_goal(self, node) -> bool:
        """
        Args:
            node (Node):
                Helper function to check if a certain node is the goal
                node based on the parameters of this maze.

        Returns:
            bool:
                True if it is the goal node, false if not
        """
        return node.state == self.goal.state
```

Source code 1.4.3: Maze Class

We will walk through the extensions of the methods from the problem class below.

For `__init__`,

```
def __init__(self,
             initial: Node,
             goal: Node,
             boundaries: Tuple[int, int],
             action_cost_map: Optional[dict],
             **kwds):
    """
    Add type hints and parameter to know boundaries
    given the assumption "Assume that the agent knows
    the boundaries of the maze and has full observability"

    Args:
        initial (Node): Node for the initial state
        goal (Node): Node for the goal state
        boundaries (Tuple[int,int]):
            (rows, cols) of boundaries, assuming (0,0) to (rows,cols)
            as problem space
    """
    super().__init__(initial=initial,
                     goal=goal,
                     boundaries=boundaries,
                     action_cost_map=action_cost_map,
                     **kwds)
```

This calls the `super().__init__()` method to be able to call the parent class Problem 1.2.1 with additional keywords that may not be defined. This is done by passing the `**kwds` parameter to the `__init__` method and subsequently passing it to `Problem.__init__()` by using `super()`.

For `action_cost`,

```
def action_cost(self, node: Node, action: str) -> int:
        """
        Args:
            node (Node): Current node state
            action (str): Action to take

        Returns:
            int: Cost (s, a, s')
        """
        return self.action_cost_map[action]
```

The method `action_cost()` takes an action and calculates the cost for the action to be performed by the agent as defined by the problem space Maze. Because each instance of maze may have different action costs, it applies a different cost for each one if the `action_cost_map` is different upon initialisation.

For `_transform_permissable_action`,

```
def _transform_permissable_action(self,
                                  actions: Tuple[Tuple[int, int], str],
                                  node: Node) -> Node:
    state, action = actions
    action_cost = self.action_cost(node, action)
    return NewState(action_cost, state, action)
```

This is simply a helper function defined for the Maze class to convert actions to more human-readable structure, NewState (which is defined as a NamedTuple). In addition, the cost is returned back so that the any subsequent methods can refer to all action_cost, state and action all at once.

For `actions`,

```
def actions(self, node: Node) -> List[NewState]:
    """
    Return permissable actions as list of actions
    as (COST, s', ACTION)
```

```
    Args:
        node (Node): Agents current state node

    Returns:
        List[NewState]:
            List of permissable states and actions to expand to
            for the given node.
    """
    x = node.state[0]
    y = node.state[1]
    parent = node.parent.state if node.parent else None

    actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']

    permissable_x_y = {((x+1, y), 'UP'),
                       ((x-1, y), 'DOWN'),
                       ((x, y+1), 'RIGHT'),
                       ((x, y-1), 'LEFT')}

    return list(
                map(
                    lambda actions: self._transform_permissable_action(actions,
                    ↪  node),
                    filter(
                            lambda result:
                               # Ensuring they permissable actions are:
                               # * within boundaries
                               # * not a shaded region
                               0<=result[0][0]<self.boundaries[0] and
                               0<=result[0][1]<self.boundaries[1] and
                               (result[0][0], result[0][1]) not in
                               ↪  self.shaded_regions,
                        permissable_x_y
                    )
            )
        )
```

This is essentially where the majority of the class logic occurs. Action applies the logic in the problem space onto the Agent, for which the class Maze then returns a list of actions applicable for the agent. This returns the list of possible actions given, in the form of a list of NewState namedtuples for use in algorithms later.

For `result`,

```
def result(self, state: Node, action: str) -> Node:
    """
    Gives the state as a result of taking the action
    specified
    Args:
        state (Node): Node state
        action (str): Action to take from node s

    Returns:
        Tuple[int, int]:
            New state / coordinate
    """
    permissable_x_y = {
        'UP': (1, 0),
        'DOWN': (-1, 0),
        'RIGHT': (0, 1),
        'LEFT': (0, -1)
    }
    change = permissable_x_y[action]
    return ((state.state[0] + change[0],
            state.state[1] + change[1]), action)
```

Result calculates and returns a list of possible states for which the agent can perform at a given Node or location on in the Maze. As the problem space and list of actions are small, we have instead defined it in actions so that the speed of calculations can be sped up. However in most problem classes, this should be more impactful and helpful in defining actions for more complex relationships.

For `h` which is the Manhatten distance:

```
def h(self, node: Node) -> Union[float, int]:
    """
```

```
    Implementing Manhatten distance.
    One of two of the heuristics for Q1.a

    Args:
        node (Node): Current agent node

    Returns:
        Union[float, int]: heuristic value
    """
    return abs(self.goal.state[0]-node.state[0]) + \
            abs(self.goal.state[1]-node.state[1])
```

This is the implementation of one of the possible heuristics used for informed search algorithms.

For `h2` which is the Euclidean distance:

```
def h2(self, node: Node) -> Union[float, int]:
    """
    Implementing Euclidean distance
    One of two of the heuristics for Q1.a
    Args:
        node (Node): Current agent node

    Returns:
        Union[float, int]: heuristic value
    """
    return ((self.goal.state[0]-node.state[0])**2 +
            (self.goal.state[1]-node.state[1])**2)**(1/2)
```

This is the implementation of one of the possible heuristics used for informed search algorithms.

For `__repr__`:

```
def __repr__(self):
    """
```

```
    Assigning a readable representation of the object.
    """
    return '{}({!r}, {!r})'.format(
        type(self).__name__, self.initial, self.goal)
```

This is a dunder method to simply be able to view more descriptive information in a human readable form.

For `is_cycle`:

```
def is_cycle(self, node: Node) -> bool:
    """
    Args:
        node (Node): Checks if a node is cyclic along its parent nodes

    Returns:
        bool: True if it is cyclic, else False.
    """
    visited = set()
    state = node.state
    while node.parent:
        if node.parent.state == state:
            return True
        node = node.parent
    return False
```

This method tests if a Node is a cycle in the search tree. This is used for algorithms with cycle check instead of visited data structure.

Finally, for `is_goal`:

```
def is_goal(self, node) -> bool:
    """
    Args:
        node (Node):
            Helper function to check if a certain node is the goal
            node based on the parameters of this maze.
```

```
    Returns:
        bool:
            True if it is the goal node, false if not
    """
    return node.state == self.goal.state
```

This defines a goal-check method for which the problem space Maze recognises as the goal. In this method, we define goal as when a Node shares the same state as the goal node that is passed into the initialising process.

1.5 Maze setup

As per the problem statement, the maze is defined as so: This corresponds to

```
start_node = Node((8,10))
goal_node = Node((11,9))

maze = Maze(start_node,
            goal_node,
            boundaries=(16, 24),
            action_cost_map={'LEFT': 10, 'RIGHT': 10, 'UP': 1, 'DOWN': 1},
            shaded_regions = {(7, 9),(6, 9),(10, 12),
                              (10, 13),(11, 12),(10, 9),
                              (12, 10),(9, 9),(13, 10),
                              (8, 9),(11, 13),(10, 10),
                              (14, 9),(11, 10)})
```

Source code 1.5.4: Instantiating Maze class

the maze diagram as provided in the initial problem statement.

1.6 Methodology, Running the algorithms and findings

For brevity on the exact implementation in code of each algorithm, they are all placed as references in Appendix A. This section will cover the general approaches and specific considerations used in each algorithm as well as our findings.

For each algorithm, we have extracted:

- Count of Nodes generated for each algorithm
- List of actions taken
- Maximum frontier size created by each algorithm

In addition, the path cost, for which is returned as the final node for each algorithm implemented as functions.

For each function, the problem class instance, Maze is instantiated. It is then used to define the limits as well as permissive actions for which the agent can act in the problem space.

For Breadth-first Search (BFS Appendix A.1), as the Node is defined as its `state`, BFS runs as normal. This changes when an additional `path_cost` is assigned in the equality comparison, as BFS would treat each expanded node as unvisited if they have different path costs. This had to be debugged by removing the `path_cost` equality check and instead have it left to each algorithm that leverages on `path_cost` to perform this check.

For Depth-First Search with cycle check (DFS with cycle check Appendix A.2), we have implemented the recursive version. This is as the implementations could play a part in determining the performance of each algorithm. Compared to Breadth First search, this could inform us on if a recursive algorithm may have a performance reduction as compared to a non-recursive one.

For Iterative Deepening search (IDS with Depth limited search Appendix A.3), this algorithm implementation also uses the `Maze.is_cycle` check to check for cycles. Though theoretically it should save on memory, running this algorithm proved to be difficult. It required restarting the kernel as due to its recursive nature and exponentially growing depth of tree, should it not find its goal early on, the algorithm takes exponentially longer time to perform the is_cycle checks. We will also see later on in section 1.7 that the memory saved trade-off does not make this algorithm a prime candidate for this problem.

We have selected to implement a generalised informed search algorithm (Appendix A.4) to generalise across the following algorithms:

- Uniform Cost Search
- A* Search

- Greedy Best First Search

This was made possible by allowing a callable `f` to be passed to the function to be used as the heuristic for the informed search algorithm. For the remaining algorithms, the `partial` function (from functools - standard library) was then used to define them by currying the appropriate cost function into the generalised algorithm as defined below:

$$\begin{aligned} \text{Uniform Cost Search} \quad & f(n) = g(n) + 0 \\ \text{A* Search} \quad & f(n) = g(n) + h(n) \\ \text{Greedy Best First Search} \quad & f(n) = h(n) \end{aligned} \tag{1.1}$$

For each algorithm that required heuristics, these were ran on both Manhatten and Euclidean heuristics.

1.7 Algorithm comparison summary

Algorithm	Nodes generated	Nodes expanded	Maximum Frontier size	Path Cost
Breadth first search	207	175	31	39
Depth first search with cycle check	67529	67443	115	509
Iterative deepening search	11773	4385	179	445
Uniform Cost Search	123	91	34	39
A* Search (Manhatten distance h)	98	65	34	39
A* Search (Euclidean distance h2)	98	65	34	39
Greedy Best First Search (Manhatten distance h)	38	22	17	61
Greedy Best First Search (Euclidean distance h2)	38	22	17	61

Table 1.1: Summary of algorithm performances

The following conclusions can be drawn. Based on this table and Figure 1.2, we can draw the following conclusions.

First, A* search almost always performs better than all the other algorithms for minimising path cost. This is consistent with theory, as A* search is an informed search algorithm that takes into account both heuristics and path costs when selecting nodes. As a result, it minimises path cost as much as possible. It loses out to Greedy best first search in all other areas (nodes generated, nodes expanded and maximum frontier size). However despite this tradeoff, it can be considered the best performing algorithm as it found the most optimal solution as path cost typically presents itself as an important factor in many real world applications and decision making. As a result, we can say that this algorithm is most suitable for searches with varying path costs (per this problem statement).

Second, iterative deepening search and Depth first search with cycle check most often generates paths with the highest path cost. This is consistent as neither of these algorithms look at path costs and revisits nodes for doing `is_cycle` checks. In particular, iterative deepening searches usually generates more than average nodes for its algorithm runs. Though this is true, in theory, both Depth first search with cycle check and Iterative deepening search should consume less memory as neither implements a visited set. This is the trade off for the using cycle checks compared to instead of a datastructure to store visited nodes.

Heuristics-wise, both heuristics are equal in this case. This makes sense as given the start and end point, the heuristics values are almost always much lower than the actual path cost. When we change the problem statement, the heuristics value could factor in when making the choice to replace nodes. An example of a modified maze can be found in the appendix A.8. In the modified maze, the action costs are modified to make Down much more expensive compared to in reality. This is so that the Heuristics would report a much lower value as compared to the actual path cost in the modified maze example. The results of running the informed search algorithms can be seen in Table 1.2.

Algorithm	Nodes generated	Nodes expanded	Maximum Frontier size	Path Cost
Greedy best first search (manhatten distance)	34	22	13	520
Greedy best first search (Euclidean distance)	36	22	15	520
A* search (manhatten distance)	234	195	40	390
A* search (Euclidean distance)	237	197	41	390

Table 1.2: Summary of heuristic performances

As a result, Manhatten distance performs better for both heuristics based informed search algorithms in this problem statement example. This can be seen in the lower nodes generated for both algorithms for Manhatten distance versus Euclidean distance for almost all facets to be evaluated. This is likely as the maze could be better represented using a Manhatten distance which uses a Manhatten 'block-like' approach to modelling the problem space.

1.8 Graph comparing Algorithm performances

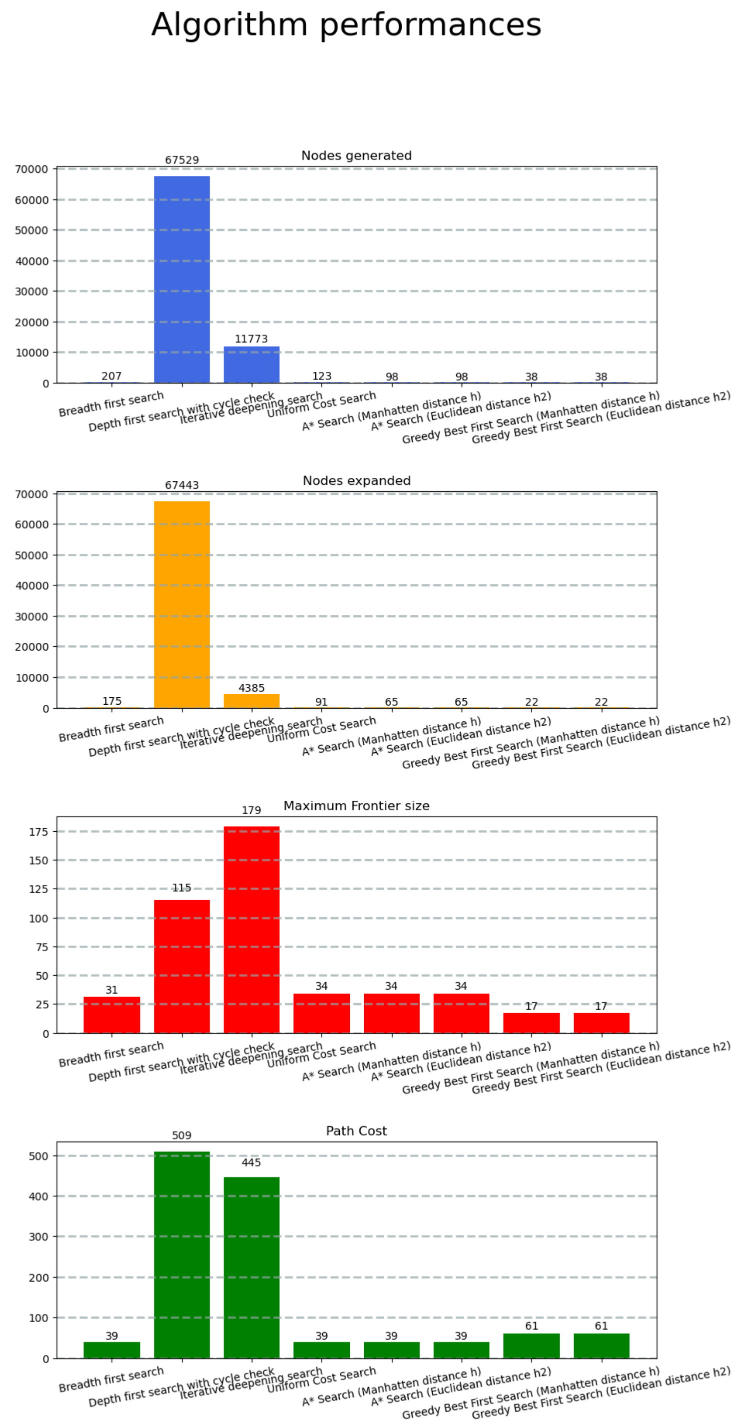

Figure 1.2: Visual summary algorithm performances

Chapter 2

Assignment Q2-1

2.1 Overview of the problem

Assignment Q2-1 explores local search in the problem space of genome assembly. In this problem, there are a number of 'reads' that consists of 4 letters, AGCT. The order for which to order each read (whether before or after another read) is unknown, but what is known is that some prefixes matches the suffix of other reads. This problem explores approaches to the Travelling Salesman Problem.

For brevity, the full TSP class as defined in our assignment is copied in its entirety to Appendix B.1. The rest of this document will refer and copy out methods as snippets for which may refer to helper methods that are defined seperately in the same class.

2.2 Defining the problem space

For simplifying the problem, a few restrictions were applied. First, only reads of length $5 \leq x \leq 30$ are to be considered valid weights.

Second, the weight of an edge between read A and read B should be the negated value of x, i.e. $-x$. The example given was, if read A is "TACTAGT" and read B is "TAGTCCCCT", then an edge is drawn FROM read A TO read B (i.e., $A \rightarrow B$) with weight of -4. This is because the 4-suffix "TAGT" is also the 4-prefix of read B; in other words, the last 4 characters of read A (a substring of length 4) overlap with the first 4 characters of read B (a substring of length 4).

With these in mind, the number of edge weights can then be found and plotted in a histogram for Exploratory Data Analysis (EDA). This is shown in Figure 2.1.

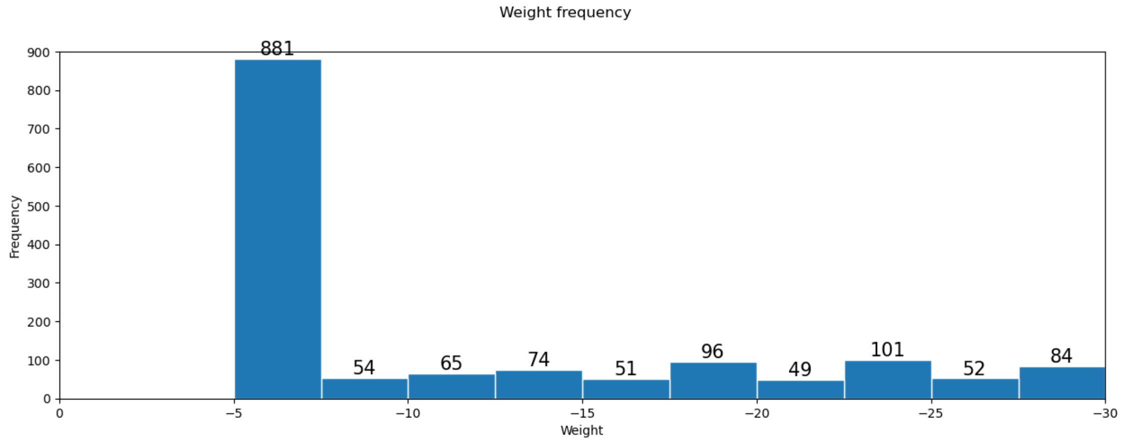

Figure 2.1: Edge Weight Frequencies / Histogram

Please note, TSP is the state for the problem is defined as a fully joined path. This is in contrast to assignment exercise Q1-1 where a state is one node at one location. Here, we define state here as "one possible TSP path going through all the nodes" which is a type of configuration state. As a result, one Node defined here may have two distinct meanings. While in the TSP class (with exception to `find_lowest_path` and the dependent methods), a Node is defined as **a possible path that goes through all of the reads exactly once**. This is the basis for which Simulated Annealing will be run on, as it iterates through multiple configuration states and hence multiple possible paths and thus ways to join the reads.

2.3 Creating the directed graph

We approached the problem by using a Greedy Algorithm. No specific algorithm was picked, however an approach similar to Greedy Best First Search was used.

This algorithm is defined in the TSP class in source code 2.3.5.

```python
def find_lowest_path(self, initial_node: Node) -> Node:
    """Greedy algorithm to find the lowest path

    Args:
        initial_node (Node): Initial starting node

    Returns:
        solution_node (Node): if found
    """
    # instantiate weights lookup
    self.calculate_weights()

    visited = [0 for i in range(len(self.weights))]
    visited[self._get_node_index(initial_node.state)] = 1
    current_node = (0, initial_node)

    total_number_of_nodes = len(self.weights)
    travelled_nodes = 1
    while travelled_nodes < total_number_of_nodes:
        path_cost, read = current_node
        neighbours = self.weights[read.state]
        found_neighbours = False
        while neighbours:
            cost, neighbour = heappop(neighbours)
            neighbour_index = self._get_node_index(neighbour)
            if not visited[neighbour_index]:
                current_node = (cost, read.expand_node(neighbour, cost))
                travelled_nodes += 1
                visited[neighbour_index] = 1
                found_neighbours = True
                break
        if not found_neighbours:
            not_visited_index = visited.index(0)
            current_node = (0,
                            read.expand_node(
                                    self._convert_node_index(not_visited_index),
                                    ↪   0)
                            )
            travelled_nodes += 1
            visited[not_visited_index] = 1
    return current_node
```

Source code 2.3.5: Greedy algorithm for TSP

2.4 Selection of scheduling strategy and parameters

The AIMA4e [2] repository provides a schedule called `exp_schedule`. Experimentation with this strategy yields that there was not a natural exit condition to the algorithm provided. As such, we have explored the use of other cooling strategies. Referring to Cohn, Harry and Fielding, Mark James (1999) [1] who performed comparisons on different schedules, Lundys scheduling strategy seem to yield the highest probability in returning the global minima. As a result, it was chosen as this assignments scheduling strategy.

The equation for the strategy is defined as:

$$T_0 = \frac{T_0}{1 + n\beta T_0} \tag{2.1}$$

To which translated to code, yielded the two following methods:

```
def lundy_schedule(self,
                     temperature: int=2000,
                     beta: float = 0.8,
                     n: int=0,
                     limit: int=9999999) -> Union[int, float]:
    """From paper "Cohn, Harry and Fielding, Mark James" (1999),
    Lundy's schedule was chosen. Schedule modified with a hard
    limit defined by the limit parameter.

    Args:
        temperature (int): starting temperature
        beta (float): hyperparameter, rate.
        n (int): Current iteration
        limit (int): limit to set to exit

    Returns:
        temperature (Union[int, float]):
            Either the temperature for this current iteration
            or 0 if limit is reached
    """
    return (temperature / (1 + n * beta * temperature)
            if n < limit else 0)

def probability(self, p):
        """Return true with probability p.
        From AIMA4e Repo
        """
        return p > random.uniform(0.0, 1.0)
```

```python
def simulated_annealing(self,
                        temperature: int= 40,
                        beta: float= 0.8,
                        limit: int= 9999999) -> List[Tuple[str, int]]:
    """Simulated annealing
    Args:
        temperature (int): Temperature to start with
        beta (int): Variable hyperparameter
        limit (int): Limit to run the simulated annealing for

    Returns:
        List[Tuple[str, int]]:
            Order if found or limit reached
    """
    current = self.initial
    n = 0
    self.reads = []
    while True:
        # Pass keyword args direct to schedule for
        # hyperparameter tuning
        T = self.lundy_schedule(
            temperature=temperature,
            beta=beta,
            n=n,
            limit=limit
        )
        if T == 0:
            return current.state
        neighbour = current.expand(self)
        if not neighbour:
            return current.state

        n += 1
        current_value = self.value(current)
        delta_e = self.value(neighbour[0]) - current_value
        if delta_e < 0 or self.probability(math.exp(-delta_e / T)):
            current = neighbour[0]
```

```
            self.cost = "Cost = " + str('%0.3f' % (self.value(current)))
            self.reads.append(self.value(current))
            print(self.cost)
            self.final = current
        else:
            self.reads.append(current_value)
```

Source code 2.4.6: Lundys Schedule and Simulated Annealing

As the algorithm provides a parameter n that can be parameterised, this provides a way to implement a limit for which the algorithm may use to as an exit strategy should a temperature not be found within the given number of iterations. We have exposed n from the method as a parameter for assisting in hyperparameter tuning by providing a cut-off for number of iterations. The parameter β works similar to `exp_schedule`'s λ in that it provides the schedule a rate to 'bounce' out of a possible local minima to continue its search for a global minima.

Integrating this into the simulated annealing method as provided in AIMA4e [2], we get the method for which utilises the schedule to determine if to take a neighbouring state.

Some changes were made to check that $\Delta e < 0$ instead of greater than so that the algorithm will automatically take the neighbouring state should it yield a more negative path cost. Otherwise, the probability function remains the same, where should the neighbouring state yield a higher (less negative) cost, then we take it with probability based on eq. (2.2).

$$\begin{gathered} X < e^{-\Delta e/T} \\ \text{Where } X \sim U(0,1) \end{gathered} \tag{2.2}$$

. In addition to this, a way to generate neighbour states was required. To do so, two-opt was used, as defined in code below (adapted from AIMA4e[2]):

```
def two_opt(self, state):
        """Neighbour generating function for Traveling Salesman Problem"""
        # remove the ordered costs due to changes that will be made
        if not isinstance(state, Node):
```

```
            neighbour_state = list(map(lambda x: x[0], state))
        else:
            neighbour_state = list(map(lambda x: x[0], 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 executed in given state"""
    return [self.two_opt]
```

2.5 Running Simulated Annealing with Lundys Schedule

The result of running simulated annealing using Lundys schedule can be seen in Figure 2.2 and Figure 2.3. These correspond to using a seeded state and random state respectively. We can observe that they have similar effects; that is that the simulated annealing heuristic will try to find the global minima within the time given. For these, this translates to a hard limit of 10,000 iterations placed to reduce time spent finding the optimal amount.

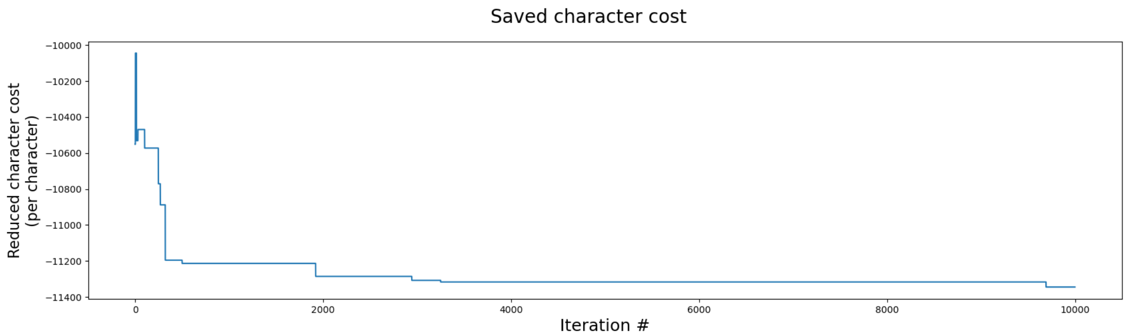

Figure 2.2: Simulated Annealing using a seeded node

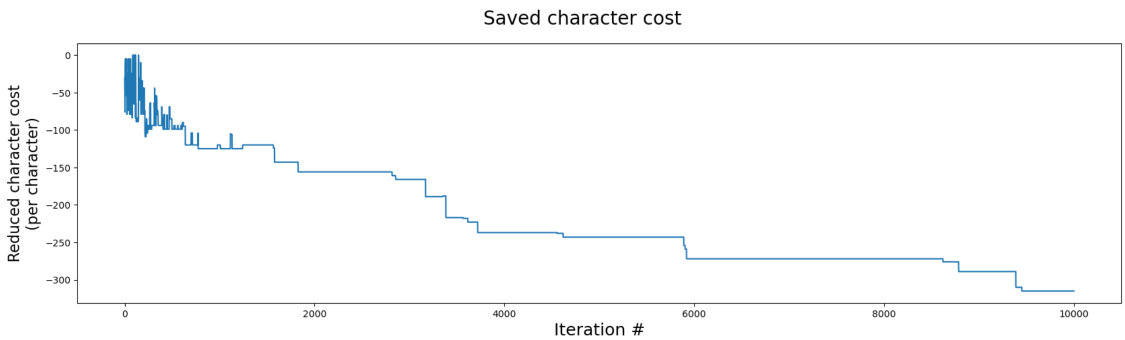

Figure 2.3: Simulated Annealing using a random node

Different hyperparameters were chosen to get to this result. With a lower β value, a result similar to a random walk can be observed (Figure 2.4). Hyperparameter tuning then becomes important to understand the optimal rate for which the schedule may be able to find the global minima. More testing can be done in this area in further runs.

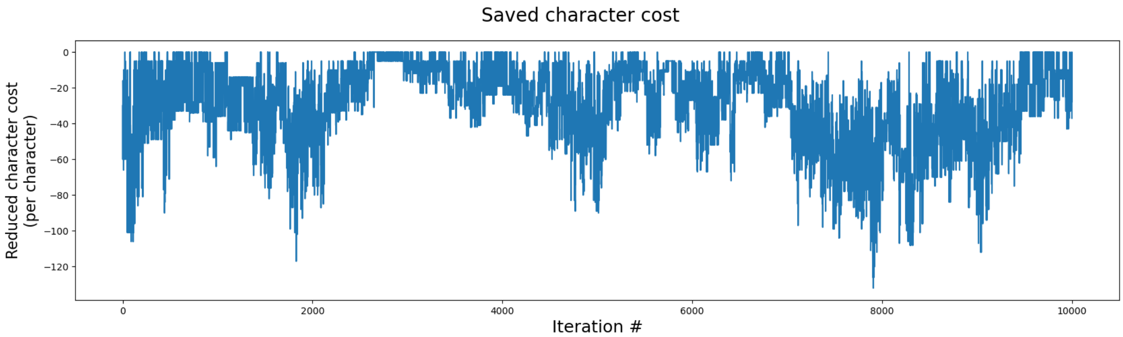

Figure 2.4: Simulated Annealing using a low beta, 0.0000002546

The lowest cost path we had found had a path cost of -12011 resulting in a string 197182 characters long. This can be found in the file attached as the variable shortest_str, which can be saved in a text file as shortest_str.txt.

Bibliography

[1] M. J. Cohn H. & Fielding, “Simulated annealing: Searching for an optimal temperature schedule”, *SIAM Journal On Optimization*, vol. 9, no. 3, pp. 779–802, 1999.

[2] P. N. Stuart Russell, “Artificial Intelligence: A Modern Approach, 4th edition”, Pearson, 2021.

Contributions

Wong Ji Fong

- Programming for the assignment (Jupyter notebook Q1-1, Q2-1). Exception is for Depth first search with cycle check for which contribution was only editing to fit existing format
- Reporting for the assignment (this document):
  - Q1-1 and Q2-1 Overview of the problems, problem classes and provision of explanations of extensions and method implementations for each class
  - Section 1.5 explanation of problem setup and parameters for Q1-1
  - Section 1.6 on methodology and approaches for writing the algorithms for Q1
  - Section 1.7 on comparing the performances of the different algorithms
  - Section 2.4 for defining the problem space and disambiguation of different 'state's within the realms of TSP
  - Section 2.3 for explanation on seeding the algorithm with a path generated with greedy search algorithm
  - Section 2.4 write-up on selection of scheduling strategy and parameters for hyperparameter tuning.
  - Section 2.5 running of simulated annealing with chosen schedule

Rohith Reddy Sankuru

- Had contributed to the conversion of the Uniform Cost search, Greedy Best Fit Search, and A* Search codes to a generic algorithm with a change in the parameter f. (that is, the evaluation function).
- Had completed the code checking and analysis with the original algorithm, leading to some code changes such as "checking the initial state for goal conditions," suggested changes in code in Depth Limited Search where goal check was initially done when the child is generated but later changed to do goal check when the child node gets expanded.

- The original algorithm advised changing the code so that it only checked the "cycle check" condition for the parent node, as opposed to the prior version, which checked the condition for all children nodes.

- When using Iterative-Deepening-Search, we originally utilized a fail condition and broke out of the loop, but subsequently adjusted it to break only if we found a solution. I modified the infinite loop to continue to a maximum depth of the boundaries of the maze. (To do this, substitute a cutoff condition for a fail-condition at each depth as suggested in the original algorithm.)

- Tested the code and verified that all the generated reports and results met the requirements.

Sweta Mishra

- Implemented depth first search algorithm. As the state variable has a path cost associated, it is possible for a single node to be added with multiple costs in the tree which can result in an infinite loop. This was countered by using a cycle check in the code .i.e. we had to resort to implementing graph search instead of tree search. An alternative way to achieve the same is by implementing a set of visited nodes.

- Assisted in implementing breadth first search.

- Tested the code involving the data structure implementing the Problem, Node and Maze classes and correcting the row , column order in accordance with the problem statement.

- Came up with the comparison of implemented algorithms based on performance, heuristic values and confirmed it's correctness with AIMA 4e repositories.

Appendix A

Algorithms

A.1 Breadth-First Search

```
def breadth_first_search(
        maze: Maze
    ) -> (Node, List[Tuple[int, int]], int):
    """
    Args:
        maze (Maze): Maze class to run the search on

    Returns:
        solution (Node):
            Node for which the solution was found
        algo_actions (List[Tuple[Tuple[int, int], str]]):
            List of coordinate,actions taken by the algo
        maximum_frontier_size (int):
            Maximum frontier size of this algo
        nodes_generated (int):
            Number of nodes generated by the algorithm
    """
    initial_node = maze.initial
    if maze.is_goal(initial_node):
        algo_actions.append((initial_node.state, 'expanded'))
        return initial_node, algo_actions, 1, 1

    frontier = deque([initial_node])
    visited = set([initial_node])
```

```python

    algo_actions = []
    current_frontier_size = 1
    maximum_frontier_size = 1
    nodes_generated = 1

    while frontier:
        frontier_node = frontier.popleft()
        current_frontier_size -= 1

        actions = maze.actions(frontier_node)
        for child in frontier_node.expand(actions):
            if maze.is_goal(child):
                return (child,
                        algo_actions,
                        maximum_frontier_size,
                        nodes_generated)
            if child not in visited:
                visited.add(child)
                frontier.append(child)
                algo_actions.append((child.state, 'generated'))
                nodes_generated += 1

                current_frontier_size += 1
                if current_frontier_size > maximum_frontier_size:
                    maximum_frontier_size = current_frontier_size
        algo_actions.append((frontier_node.state, 'expanded'))
    raise Exception('Unable to find target')
```

A.2 Depth First Search with cycle check

```python
def depth_first_search_with_cycle_check(
        maze: Maze,
        curnode: Node,
        current_frontier_size : int,
        algo_actions: List[Tuple[int, int]] = [],
        maximum_frontier_size: int = 0,
        nodes_generated: int = 0
    ) -> bool:
    """
    Args:
        maze (Maze): Maze class to run the search on
        curnode (Node): Node for which we are searching on
        current_frontier_size (int): frontier size for this interation
        algo_actions (List[Tuple[Tuple[int, int], str]]):
            List of coordinate,actions taken by the algo
        maximum_frontier_size (int): Maximum frontier size at any time
        nodes_generated (int): Number of nodes generated so far

    Returns:
        solution (Node):
            Node for which the solution was found
        expanded_nodes (List[Tuple[int, int]]):
            Nodes for which was visited
        maximum_frontier_size (int):
            Maximum frontier size of this algo
        nodes_generated (int):
            Explored nodes
    """
    if maze.is_goal(curnode):
        return curnode, algo_actions, maximum_frontier_size, nodes_generated

    if current_frontier_size > maximum_frontier_size:
        maximum_frontier_size = current_frontier_size

    actions = maze.actions(curnode)
    if not maze.is_cycle(curnode):
```

```
        for child in curnode.expand(actions):
            nodes_generated += 1
            algo_actions.append((child.state, 'generated'))
            (solution_node,
             algo_actions,
             frontier_size,
             nodes_generated) = depth_first_search_with_cycle_check(
                                     maze,
                                     child,
                                     1 + current_frontier_size,
                                     algo_actions,
                                     maximum_frontier_size,
                                     nodes_generated
                                 )
            if frontier_size > maximum_frontier_size:
                maximum_frontier_size = frontier_size

            if isinstance(solution_node, Node):
                return (solution_node,
                        algo_actions,
                        maximum_frontier_size,
                        nodes_generated)
    algo_actions.append((curnode.state, 'expanded'))
    return None, algo_actions, maximum_frontier_size, nodes_generated
```

A.3 Depth limited search with Iterative Deepening

```python
import sys

sys.setrecursionlimit(500000)

def depth_limited_search(
        maze: Maze,
        limit: int,
        algo_actions: List[Tuple[Tuple[int, int], str]],
        nodes_generated: int=1
    ) -> (Union[Node, str],
          List[Tuple[int, int]],
          int,
          int):
    """
    Args:
        maze (Maze): Maze to solve
        l (int): limit for which to apply as cutoff
        expanded (list): To track expanded nodes
        nodes_generated (int): number of nodes generated. Includes initial

    Returns:
        Union[Node, str]:
            str returned on failure to find solution;
            else node returned if found
        algo_actions (List[Tuple[Tuple[int, int], str]]):
            List of coordinate,actions taken by the algo
        max_frontier_size int:
            Maximum frontier size of this iteration
            of depth limited search
        size_of_frontier int:
            Remaining length of frontier on exiting this function
    """
    frontier = [maze.initial]
    if maze.is_goal(maze.initial):
        algo_actions.append((maze.initial.state, 'expand'))
```

```
        return maze.initial, algo_actions, 1, 1

    solution = 'FAIL'

    current_frontier_size = 1
    max_frontier_size = 1

    while frontier:
        frontier_node = frontier.pop()
        current_frontier_size -= 1

        if maze.is_goal(frontier_node):
            return frontier_node, algo_actions, max_frontier_size,
            ↪ nodes_generated

        if len(frontier_node) > limit:
            solution = 'CUTOFF'
            return solution, algo_actions, max_frontier_size, nodes_generated
        else:
            actions = maze.actions(frontier_node)

            if not maze.is_cycle(frontier_node):
                for child in frontier_node.expand(actions):
                    algo_actions.append((frontier_node.state, 'generated'))
                    nodes_generated += 1

                    frontier.append(child)
                    current_frontier_size += 1
                    if current_frontier_size > max_frontier_size:
                        max_frontier_size = current_frontier_size
            algo_actions.append((frontier_node.state, 'expanded'))
    return solution, algo_actions, max_frontier_size, nodes_generated

def iterative_deepening_search(
        maze: Maze,
        nodes_generated:int=0
    ) -> (Union[Node, str], List[Tuple[int, int]], int):
    """
```

```
    Args:
        maze (Maze): Maze problem class to solve
        depth (int): Depth to limit the search to

    Returns:
        Union[Node, str]:
            Node or exit message
        Tuple[List[Tuple[int, int]], str]:
            List of coordinate,actions taken by the algo
        int:
            # of explored nodes
    """
    algo_actions = []
    max_frontier_size = 0
    all_frontier_size = 0
    nodes_generated = 0
    depth = 0

    # As we assumed that the agent knows the boundaries
    # of the maze and has full observability, max d
    # will only be all the cells in the boundaries.
    # Agent is not assumed to know shaded areas hence not calculated in
    while depth < maze.boundaries[0] * maze.boundaries[1]:
        depth += 1

        (result,
         algo_actions,
         frontier_size,
         nodes_generated) = depth_limited_search(maze, depth, algo_actions,
         ↪  nodes_generated)

        if frontier_size > max_frontier_size:
            max_frontier_size = frontier_size

        if isinstance(result, Node) or result == 'FAIL':
            return result, algo_actions, max_frontier_size, nodes_generated
    return 'FAIL', algo_actions, max_frontier_size, nodes_generated
```

A.4 Generalised Informed Search

```
def informed_search(
        maze: Maze,
        f: Callable
    ) -> (Node, Tuple[List[Tuple[int, int]], str], int):
    """Uses priority queues (in python, heapq module) to minimise
    the target function.

    Args:
        maze (Maze): Maze to run search on
        f (Callable): Target function to minimise.

    Returns:
        child (Node): Solution node
        algo_actions (Tuple[List[Tuple[int, int]], str]):
            List of coordinate,actions taken by the algo
        max_frontier_size (int): Maximum frontier size
        explored (int): Sum of nodes explored
    """
    initial_node = maze.initial

    frontier = [(f(initial_node), initial_node)]
    visited = {maze.initial.state: maze.initial}

    algo_actions = []
    current_frontier_size = 1
    max_frontier_size = 1
    nodes_generated = 0

    while frontier:
        heuristic, frontier_node = heappop(frontier)
        current_frontier_size -= 1

        actions = maze.actions(frontier_node)
        if maze.is_goal(frontier_node):
            return (frontier_node,
                    algo_actions,
```

```
                    max_frontier_size,
                    nodes_generated)

        for child in frontier_node.expand(actions):
            if (child.state not in visited or
                child.path_cost < visited[child.state].path_cost):
                nodes_generated += 1
                algo_actions.append((child.state, 'generated'))

                visited[child.state] = child
                heappush(frontier, (f(child), child))
                current_frontier_size += 1
                if current_frontier_size > max_frontier_size:
                    max_frontier_size = current_frontier_size
        algo_actions.append((frontier_node.state, 'expanded'))
    raise Exception('Unable to find target')
```

A.5 Uniform Cost Search

Generalised informed search where $f(n)$ is defined as $f(n) = g(n) + 0$:

```
uniform_cost_search = partial(informed_search, f=lambda node: node.path_cost + 0)
```

A.6 A* Search

Generalised informed search where $f(n)$ is defined as $f(n) = g(n) + h(n)$:

```
# Manhatten distance heuristic
astar_search_h = partial(informed_search, f=lambda node: node.path_cost +
↪  maze.h(node))
# Euclidean distance heuristic
astar_search_h2 = partial(informed_search, f=lambda node: node.path_cost +
↪  maze.h2(node))
```

A.7 Greedy Best First Search

Generalised informed search where $f(n)$ is defined as $f(n) = h(n)$:

```
# Manhatten distance heuristic
gbfs_search_h = partial(informed_search, f=lambda node: maze.h(node))
# Euclidean distance heuristic
gbfs_search_h2 = partial(informed_search, f=lambda node: maze.h2(node))
```

A.8 Modified Maze

```
modified_maze = Maze(start_node,
            goal_node,
            boundaries=(16, 24),
            action_cost_map={'LEFT': 10, 'RIGHT': 10, 'UP': 10, 'DOWN': 100},
            shaded_regions = {(7, 9),(6, 9),(10, 12),
                              (10, 13),(11, 12),(10, 9),
                              (12, 10),(9, 9),(13, 10),
                              (8, 9),(11, 13),(10, 10),
                              (14, 9),(11, 10)})
```

Appendix B

Travelling Salesman Problem

B.1 Travelling Salesman Problem Class

```python
class ProblemException(Exception):
    pass

# Code to generate neighbours, value of states, etc.
class TSP(Problem):
    #Implement TSP class here
    def load_data(self) -> dict:
        """Loads data from path provided

        Args:
            path (str): local path to file

        Returns:
            dict:
                dict in the following schema:
                {READ_INDEX (str): READ_SEQUENCE (str)}
        """
        data = {}
        with open(self.data_path) as file:
            entries = map(dict, csv.DictReader(file))
            for entry in entries:
                data[entry['read_index']] = entry['read_sequence']
        self.data = data

```

```
    def calculate_weight(self, a_string: str, b_string: str) -> int:
        """
        Args:
            a_string (str): String to compare as from node
            b_string (str): String to compare as to node

        Returns:
            int:
                Weight with -ve of a -> b
        """
        if a_string == b_string:
            return -30
        for i in range(1, 26):
            if a_string[i:] == b_string[:-i]:
                return -(30-i)
        return 0

    def calculate_weights(self) -> None:
        """
        Args:
            csv_data (dict):
                Data from csv with the following schema
                {READ_NAME (str): READ_SEQUENCE (str)}

        Attributes:
            weights:
                Dict with the following schema:
                {('READ_#', 'READ_#2'): WEIGHT_OF_READ#_TO_READ#2}
            adjacency_matrix:
                Save read to read costs
            stat:
                Save reads in flat form to report statistics
        """
        weights = {}
        set_added = set()
        stat = []

        adjacency_matrix = [[0 for j in range(len(self.data))] for i in
        ↪  range(len(self.data))]
```

```
        for sequences in combinations(self.data.items(), 2):
            ((read_name_1, sequence_1), (read_name_2, sequence_2)) = sequences
            if read_name_1 == read_name_2:
                continue

            weight_1_2 = self.calculate_weight(sequence_1[-30:],sequence_2[:30])
            if weight_1_2:
                weight_lib = weights.get(read_name_1, [])
                heappush(weight_lib, (weight_1_2, read_name_2))
                weights[read_name_1] = weight_lib
                adjacency_matrix[self._get_node_index(read_name_1)]\
                                [self._get_node_index(read_name_2)] = weight_1_2
                stat.append(weight_1_2)

            weight_2_1 = self.calculate_weight(sequence_2[-30:],sequence_1[:30])
            if weight_2_1:
                weight_lib = weights.get(read_name_2, [])
                heappush(weight_lib, (weight_2_1, read_name_1))
                weights[read_name_2] = weight_lib
                adjacency_matrix[self._get_node_index(read_name_2)]\
                                [self._get_node_index(read_name_1)] = weight_2_1
                stat.append(weight_2_1)
        self.weights = weights
        self.adjacency_matrix = adjacency_matrix
        self.stat = stat


    def is_cycle(self, node: Node):
        read = node.state
        while node.parent is not None:
            if node.parent.state == read:
                return True
            node = node.parent
        return False

    def _get_node_index(self, read: str) -> int:
        """
        Helper methods to get read index given read name
```

```python
        """
        return int(read.split('_')[1])

    def _convert_node_index(self, read_index: int) -> str:
        """
        Helper methods to get read name given node index
        """
        return f'read_{read_index}'

    def find_lowest_path(self, initial_node: Node) -> Node:
        """Greedy algorithm to find the lowest path

        Args:
            initial_node (Node): Initial starting node

        Returns:
            solution_node (Node): if found
        """
        # instantiate weights lookup
        self.calculate_weights()

        visited = [0 for i in range(len(self.weights))]
        visited[self._get_node_index(initial_node.state)] = 1
        current_node = (0, initial_node)

        total_number_of_nodes = len(self.weights)
        travelled_nodes = 1
        while travelled_nodes < total_number_of_nodes:
            path_cost, read = current_node
            neighbours = self.weights[read.state]
            found_neighbours = False
            while neighbours:
                cost, neighbour = heappop(neighbours)
                neighbour_index = self._get_node_index(neighbour)
                if not visited[neighbour_index]:
                    current_node = (cost, read.expand_node(neighbour, cost))
                    travelled_nodes += 1
                    visited[neighbour_index] = 1
```

```
                    found_neighbours = True
                    break
            if not found_neighbours:
                not_visited_index = visited.index(0)
                current_node = (0,
                                read.expand_node(
                                            self._convert_node_index(not_visited_index),
                                            0)
                                )
                travelled_nodes += 1
                visited[not_visited_index] = 1
        return current_node

    def get_read_order(self, node: Node) -> List[str]:
        """Given a node, traces back the order
        for which the graph was traversed.

        Args:
            node (Node): Solution node

        Returns:
            List[str]:
                List of reads for which was traversed
                in order
        """
        order = []
        while node.parent is not None:
            order.append((node.state, node.cost))
            node = node.parent
        order.append((node.state, 0))
        return order[::-1]

    def recreate_str(self, order: List[Tuple[str, int]]) -> str:
        """Recreates the string based on the order list
        Args:
            order (List[Tuple[str, int]]):
                order and cost from n-1 to n of the read
```

```python

        Returns:
            str:
                Final reconstructed string
        """
        final_string: str = self.data[order[0][0]]
        for i in range(0, len(self.weights)-1):
            final_string += self.data[order[i+1][0]][-order[i+1][1]:]
        return final_string

    def value(self, order: Union[List[Tuple[str, int]], Node]) -> int:
        """Calculates the total characters shortened
        given a read order in the form of
        List[tuple[read_name (str), cost_to_this_read (int)]]

        Args:
            order (List[Tuple[str, int]]):
                order and cost from n-1 to n of the read

        Returns:
            int:
                Cost of traversal
        """
        if isinstance(order, Node):
            order = order.state
        if len(order) < len(self.data):
            raise ProblemException(
                'Order is incomplete; there are less entries of traversal'
            )
        return reduce(lambda x, y: x + y, map(lambda x: x[1], order))

    def regenerate_order_weights(self, order: List[str]) -> List[Tuple[str,
    ↪   int]]:
        """Calculate edge weights for a given read order

        Args:
            order:
                Order of weights for a given list of reads
```

```

        Returns:
            List[Tuple[str, int]]:
                Order with costs in tuple index 1
        """
        entries = [(order[0], 0)]
        for i in range(1, len(order)):
            cost = tsp.adjacency_matrix[tsp._get_node_index(order[i-1])]\
                    [tsp._get_node_index(order[i])]
            entries.append((order[i], cost))
        return entries

    def path_cost(self,
                  cost: int,
                  state1: List[Tuple[str, int]],
                  action: str,
                  state2: List[Tuple[str, int]]) -> int:
        """Recalculate weights given order of state2, then
        finds the total cost for the weights and return the result
        """
        return self.value(state2)

    def print_path(self, order: List[Tuple[str, int]]) -> None:
        """Convienience function to print final path taken

        Args:
            order (List[Tuple[str, int]]):
                Order of nodes visited and costs

        Returns:
            None
        """
        i_prev = 0
        mssg = ''
        for idx in range(len(order)-1):
            read, cost_to_read = order[idx][0], order[idx+1][1]
            mssg += f'{read}\n   |\n   | Cost: {cost_to_read}\n   v\n'
        mssg+= f'{order[-1][0]}'
```

```
            print(mssg)

    # Below are methods adapted from the aima4e python directory
    # https://github.com/aimacode/aima-python/blob/master/gui/tsp.py
    def two_opt(self, state):
        """Neighbour generating function for Traveling Salesman Problem"""
        # remove the ordered costs due to changes that will be made
        if not isinstance(state, Node):
            neighbour_state = list(map(lambda x: x[0], state))
        else:
            neighbour_state = list(map(lambda x: x[0], 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 executed in given state"""
        return [self.two_opt]

    def result(self, state, action):
        return self.regenerate_order_weights(action(state))

    def probability(self, p):
        """Return true with probability p."""
        return p > random.uniform(0.0, 1.0)

    def exp_schedule(self,
                     temperature: int=40,
                     k: int=20,
                     lam: float=0.005,
                     limit: int=100) -> Union[int, float]:
        """Adapted from exp_schedule aima4e repo, but exposing parameters
        for hyperparameter tuning
```

```
        Args:
            k (int):
            temperature (int): temperature
            lam (float): lambda to set change / 'bounce' rate variation / degree
            limit (int): limit to set to exit
        """
        return -k * math.exp(-lam * temperature) if temperature < limit else 0

    def lundy_schedule(self,
                       temperature: int=2000,
                       beta: float = 0.8,
                       n: int=0,
                       limit: int=9999999) -> Union[int, float]:
        """From paper "Cohn, Harry and Fielding, Mark James" (1999),
        Lundy's schedule was chosen. Schedule modified with a hard
        limit defined by the limit parameter.

        Args:
            temperature (int): starting temperature
            beta (float): hyperparameter, rate.
            n (int): Current iteration
            limit (int): limit to set to exit

        Returns:
            temperature (Union[int, float]):
                Either the temperature for this current iteration
                or 0 if limit is reached
        """
        return (temperature / (1 + n * beta * temperature)
                if n < limit else 0)

    def simulated_annealing(self,
                            temperature: int= 40,
                            beta: float= 0.8,
                            limit: int= 9999999):
        """Simulated annealing where temperature is taken as user input"""
        current = self.initial
        n = 0
```

```
        self.reads = []
        while True:
            # Pass keyword args direct to schedule for
            # hyperparameter tuning
            T = self.lundy_schedule(
                temperature=temperature,
                beta=beta,
                n=n,
                limit=limit
            )
            if T == 0:
                return current.state
            neighbour = current.expand(self)
            if not neighbour:
                return current.state

            n += 1
            current_value = self.value(current)
            delta_e = self.value(neighbour[0]) - current_value
            if delta_e < 0 or self.probability(math.exp(-delta_e / T)):
                current = neighbour[0]
                self.cost = "Cost = " + str('%0.3f' % (self.value(current)))
                self.reads.append(self.value(current))
                print(self.cost)
                self.final = current
            else:
                self.reads.append(current_value)
```

Source code B.1.7: Travelling Salesman Problem

B.2 Node Class

B.3 Problem Class

```python
import math
class Node:
    "A Node in a search tree."
    def __init__(self, state, parent=None, action=None, path_cost=0, **kwds):
        self.__dict__.update(state=state, parent=parent, action=action, path_cost=path_cost, **kwds)

    def __repr__(self): return '<{}>'.format(self.state)
    def __len__(self): return 0 if self.parent is None else (1 + len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost
    def __eq__(self, other): return isinstance(other, Node) and self.state == other.state
    def expand_node(self, state, cost):
        new_node = Node(state,
                   path_cost=self.path_cost + cost,
                   cost = cost,
                   parent=self)
        return new_node

# Below are methods adapted from the aima4e python directory
# https://github.com/aimacode/aima-python/blob/master/search.py
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.state, action, next_state))
    return next_node
```

Source code B.2.8: Node class for TSP

```python
# Problem Class
class Problem:
    """The abstract class for a formal problem. A new domain subclasses this,
    overriding `actions` and `results`, and perhaps other methods.
    The default heuristic is 0 and the default action cost is 1 for all states.
    When you create an instance of a subclass, specify `initial`, and `goal` states
    (or give an `is_goal` method) and perhaps other keyword args for the subclass."""

    def __init__(self, initial=None, goal=None, **kwds):
        self.__dict__.update(initial=initial, goal=goal, **kwds)

    def actions(self, state):        raise NotImplementedError
    def result(self, state, action): raise NotImplementedError
    def is_goal(self, state):        return state == self.goal
    def action_cost(self, s, a, s1): return 1
    def h(self, node):               return 0

    def __str__(self):
        return '{}({!r}, {!r})'.format(
            type(self).__name__, self.initial, self.goal)
```

Source code B.3.9: Problem class for TSP. Identical to Q1-1 Problem abstract class