Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Fodor, Zsófia and Katona, Áron
Group: 30433
Email: fodorzsofi@yahoo.com katonaaron01@gmail.com

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Chapter 1

A1: Search

1.1 Introduction

Our aim is to implement multiple search algorithms, and compare them with themselves and with the ones that were already discussed: A*, DFS, BFS, UCS.

We chose the following searching strategies:

Uninformed

- Bidirectional Search
- Depth Limited Search
- Iterative Deepening Depth-First Search

Informed

- Iterative Deepening A*
- Recursive Best-First Search

1.2 Bidirectional Search

Bidirectional search simultaneously searches from the start state to the goal state (**forward searching**) and from the goal state to the start state (**backward searching**) hoping that these two searches will meet. The time and space complexity was reduced from $O(b^d)$ to $O(2 * b^{d/2}) = O(b^{d/2}) << O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution.

1.2.1 Implementation

The algorithm uses the **Breadth-First Search policy**, meaning that it takes the shallowest node first. It stops when the forward and backward search intersect. If no solutions is found, than it returns *None*.

We used two sets one representing the explored nodes that are still in the queue and another one, that contains the visited nodes that were popped out of the queue. Entering the while loop, we firstly analyze the forward search. Take the next node from the corresponding queue and if it is not yet visited we put it in the visited set. After we have iterated through the nodes of the backward search, if there is a meeting point then we return the current path and the path accumulated by the backward search in reverse order.

Next we analyze the backward search the same way as we did for the forward search, the only difference being that we replace each action with its opposite action, we reverse the list containing them and add it to the accumulated path.

The motivation behind using sets is that they are more efficient than lists in verifying whether they contain an arbitrary element.

Source code: Appendix A.2.

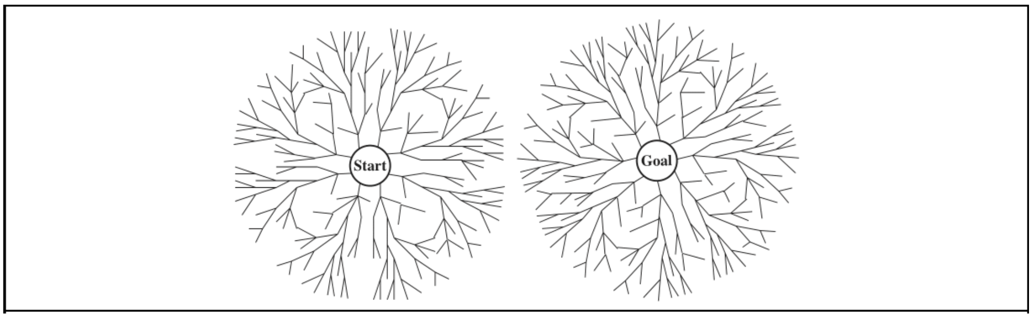

Figure 1.1: Visualization of the bidirectional search algorithm. [1]

1.3 Iterative Deepening Depth-First Search

1.3.1 Depth-Limited Search

The depth-limited search is similar to the depth-first search, with the only difference, that a **depth bound** is added. Thus the algorithm searches for the goal state until the search tree's depth reaches the bound. If the bound is reached, the search backtracks to the parent node and continues searching in the next successor of the parent node. Therefore if there is a path to the goal, whose length is smaller than or equal to the bound, it will be found.

Implementation

To represent the nodes we used the `Node` class, which is a simple data class whose `state`,`action` and `cost` fields are used. More information about the class can be found in Appendix A.1.

The algorithm is a modified version of the recursive **depth-first search**. At each level of recursion, the function calls itself for all of the successors. The function returns not only the found goal node on success and None on error, but also an additional boolean parameter which specifies whether the search terminated because of a cutoff, or not. In each level of recursion, the limit is decremented. When it reaches 0, the returned cutoff value is true.

Source code: Appendix A.3.

1.3.2 Iterative Deepening Search

The iterative deepening search strategies apply a search algorithm multiple times, with increasing bounds until the goal is reached. The calculation of the bound depends on the applied algorithm. We used the strategy of iterative deepening for the following search algorithms: depth-first search (1.3.3) and A* (1.4).

1.3.3 Iterative Deepening Depth-First Search

Starting from an initial bound of 0, we call the depth-limited search and verify the returned result. In case a cutoff occurred, we increase the bound by one, and start again the search. If the goal node was found, the path from the start node which ends in the goal node is reconstructed and returned.

The algorithm is complete, because depth-first search, and in particular depth-limited search is guaranteed to find a solution, if the distance to the goal node is inside the bound. The bound is incremented from 0 indefinitely, thus if the solution exists, it will be found.

The algorithm is optimal for unit action costs, because if no solution was found for bound b, the length of the shortest path from the start node to the goal node is at least $b + 1$, and if there is any solution with length $b + 1$, all of them are optimal solutions, and one of them will be found by the depth-limited search.

Source code: Appendix A.4.

1.4 Iterative Deepening A* Search

Iterative deepening is a preferred algorithm when we have a larger search state space than the one that can fit in memory and the depth of our solution is not known.

Iterative Deepening A* is similar to A* the difference being that we do not keep all reached states in memory, at a cost of visiting some states multiple times. It is a very frequently used algorithm for problems that do not fit in memory, as stated above.

1.4.1 Implementation

The idea behind the algorithm was that we search the node with the lowest combined cost and heuristic first ($f = g + h$). The algorithm is limiting the size of the frontier by using this calculated f value as a bound.

We used a set in which we stored the visited nodes. Searching in sets is faster than searching in lists, that is why we chose set. Besides this set, we used a list called path, that stored for each node its state, the action that needs to be performed to get there from the previous node and the cost.

We have a utility function that we call repeatedly until the returned value is either a very large number (∞) or the *None* state and we change the value of the bound to the returned value if none of these conditions is satisfied.

The utility function begins with taking the last element from the path, computing the f value of this element and comparing it to the bound. If it is larger than the bound, we return the f value. If the state of the node is the goal state, it means we reached our goal, so we return the *None* state. If none of these conditions are fulfilled, then we iterate through each child of the current node, add it to the visited set and to the path list, then we call the utility function on it. We make the adjustments according to the returned value of this call, remove the node from the visited list and the path. Lastly we return the minimum value which represents the minimum cost of all values that exceeded the current bound.

1.4.2 Remarks

This algorithm was actually easy to implement after we understood the idea behind it.

Comparing the number of expanded nodes we can see that it is much larger than for the A* algorithm, because of the fact that we do not keep all reached states in memory risking the fact that we might visit some nodes more than once.

1.5 Recursive Best-First Search

The recursive best-first search, similarly to the depth-limited search (1.3.1), searches for the deepest node first, but stops after surpassing a bound. But in this case the bound is placed on the f-value of a node, instead of its depth.

The f-value of a node is the maximum between the f-value of its parent and the cost of reaching the node + the heuristic, i.e. $f(n) = max\{g(n) + h(n), f(n.parent)\}$. When a branch of the search is cut off because the limit on the f-value, while backtracking, the f-value of a parent node will be updated with the f-value of its child. This way, the f-value of a node will become a more and more accurate estimation of the true cost of reaching the goal node from it.

This algorithm uses the principle of best-first searching. It only expands the node with the smallest f-value. If the node is on the current searching branch, than it will be expanded, otherwise the search tree backtracks to the level of the node with the smallest f-value. Therefore at each level of recursion we take the first two successors with the lowest f-value, call the search for the best option, and supply the bound as the f-limit of the alternative successor. This is repeated until the solution is found, or until the f-value of the best node will become greater than the f-limit. Thus it always considers a best and an alternative path, and switches between them, if the alternative path becomes the best one, in terms of lowest f-value.

The advantage of this algorithm is that it uses linear space: used for storing the nodes along the search path, and also the sibling of each node. The disadvantage is that it regenerates already visited nodes, which could happen very frequently. Thus it trades speed for storage.

For the implementation the `Node` class (A.1) was used for storing the nodes, and a priority queue was used for obtaining the best successor of a node. The recursive function returns the result and None, if the goal node was found. If the goal was not found, it returns None and the f-value of the deepest node reached before surpassing the limit.

Source code: Appendix A.6.

1.6 Comparison

Consider b the branching factor and d the depth of the shallowest solution.

1.6.1 Bidirectional Search vs Breadth-First Search

As stated in chapter 1.2, the bidirectional search reduces the size of the frontier and the running time from $O(b^d)$ to $O(b^{d/2})$. To find a solution with length d, the two frontiers contain in the worst case only the nodes with the depth $d/2$ from either nodes. Thus both the space and the running time is reduced.

However the implementation of the bidirectional search is difficult if the actions cannot be reversed fast. E.g. North $\leftrightarrow$ South.

1.6.2 Iterative Deepening Depth-First Search

vs Depth-First Search: The depth-first search does not find always the optimal solution, and may not find any solution even if it exists in the graph. However the iterative deepening version is optimal and complete, because each path is considered in increasing order of depth.

vs Breadth-First Search: The running time has the same $O(b^d)$ complexity. For the iterative deepening depth-first search the space complexity is reduced from $O(b^d)$ to $O(bd)$, because the deepest node is taken first. However because of the iterative deepening, the number of expanded nodes is higher.

1.6.3 Iterative Deepening A* vs A*

Iterative Deepening A* is similar to A* the difference being that we do not keep all reached states in memory, at a cost of visiting some states multiple times. It is a very frequently used algorithm for problems that do not fit in memory.

1.6.4 Recursive Best-First Search

vs A*: RBFS uses only linear space, but suffers from frequently regenerating the nodes. Given enough time it could solve those problems that could not be solved by A* because of running out of memory.

vs Iterative Deepening A* The two algorithms are solutions for the same problem of reducing the size of the frontier of A*. Both suffer from revisiting and regenerating nodes. However the RBFS is slightly more efficient, because of storing more information that the other one, thus increasing the speed.

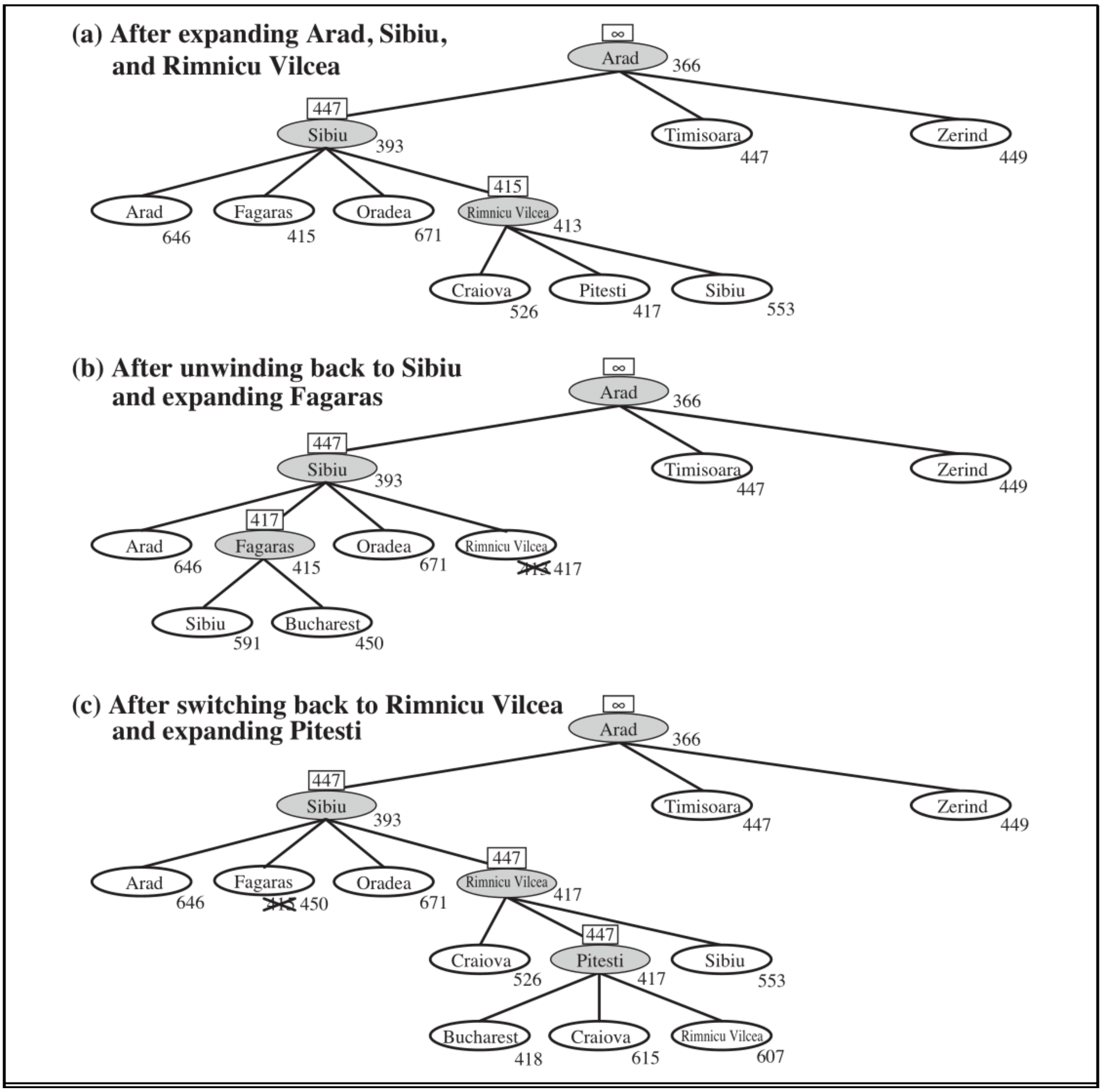

Figure 1.2: Finding a path from Arad to Bucharest by using recursive best-first search. The number above a node is its f-limit, and the number on the right of it is its f-value. The searching branch was switched two times. [1]

Chapter 2

A2: Logics

2.1 Introduction

2.2 Logic via algebra

2.3 Solving lady and tigers with algebra

2.3.1 The first trial

Representation

In **propositional logic** we can represent the state of the i-th room by the following predicates:

- li: There is a lady in room i.
- ti: There is a tiger in room i.
- mi: Message on the door of room i

Then one must specify, that if the room contains a lady, it does not contain a tiger, and viceversa.

```
l1 -> -t1.
l2 -> -t2.
```

For representing the rooms in **modular arithmetics** we can define:

- $ri = [[$"There is a lady in room i"$]] = 1 + [[$"There is a tiger in room i"$]]$
- $mi = [[$"The message on the door of room i is true"$]]$

In this representation the appearance of a lady and a tiger in the same room is implicitly exclusive.

Knowledge base

The following statements are given:

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** In this room there is a lady, and in the other room there is a tiger.

3. **Message on door 2:** In one of these rooms there is a lady, and in one of these rooms there is a tiger.

4. One of the messages is true, but the other one is false.

The first statement enumerates all four possibilities. It does not give more information than the one included in the representation.

The second statement represents the truth value of the first message. It can be written the following forms:

```
m1 <-> l1 & t2.
```

Propositional logic

```
m1 = r1 * (r2 + 1).
```

Modular arithmetics

One can observe that logical equivalence was replaced with equality, conjunction with multiplication, and negation with an addition by one.

The second message states that at least one tiger and at least one lady exists. Because there are only two rooms, this means that one room contains a lady, and the other one contains the tiger. Thus we can use the XOR operator between `r1` and `r2` to force exclusivity, which in the case of the modulo 2 algebra it is represented by the "+" symbol.

```
m2 <-> (l1 | l2) & (t1 | t2).
```

Propositional logic

```
m2 = r1 + r2.
```

Modular arithmetics

As for the fourth statement, we simply need to specify that the two messages are not equal, i.e. the first is equal with the negation of the second. Or we can say that the two does not take the same value simultaneously (XOR).

```
m1 <-> -m2.
```

Propositional logic

```
m1 + m2 = 1.
```

Modular arithmetics

Resolution

The advantage of the modular arithmetics is that it can be resolved by algebraic methods, taking into consideration the rules of the boolean ring.

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 * (r2 + 1) & (2.1a) \\ m2 = r1 + r2 & (2.1b) \\ m1 + m2 = 1 & (2.1c) \end{cases}$$

By replacing $m1$ and $m2$ in 2.1c we obtain:

$$r1 * (r2 + 1) + r1 + r2 = 1$$

Which can be rewritten as

$$r2 * (r1 + 1) + r1 + r1 = 1$$

Knowing that $x + x = 0$ and $x + 0 = x$, we can omit the $r1 + r1$ term form the equation and obtain:

$$r2 * (r1 + 1) = 1$$

In order to satisfy the equation, both operands of the multiplication must be 1. Otherwise the product would be 0. Therefore the result will be:

$$\begin{cases} r1 = 0 \\ r2 = 1 \end{cases}$$

2.4 Simplifying modular arithmetic expressions programatically

2.5 Solving systems of modular arithmetic equations programatically

2.6 Translating propositional logic to modular arithmetic programatically

Bibliography

[1] Stuart Russell and Peter Norvig. Solving problems by searching. In *Artificial Intelligence: A Modern Approach*, chapter 3, pages 63–105. Pearson, 4 edition, 2020.

Appendix A

Source Code: Search

The code snippets below are from `search.py`.

A.1 Nodes

To represent the search nodes, we used different strategies across the various search algorithms. In some cases we used tuples containing the state, the action and the cumulative cost of a node, while in other cases we used the `Node` class defined below.

In the latter case, we stored in a node a reference to its parent. This information is used for reconstructing the path from the start node to the goal node, in the `reconstructPath(node)` method.

```python
class Node:
    """
    Node used for search algorithms.
    """

    def __init__(self, state, action, cost, parent, f=0):
        """
        state: (x,y) coordinates
        action: direction from Directions of game.py
        cost: cost of reaching the node
        parent: parent node
        f: the backed-up f-value
        """
        self.state = state
        self.action = action
        self.cost = cost
        self.parent = parent
        self.f = f
```

Listing A.1: The Node class

```python
def reconstructPath(node):
    """
    Reconstructs a path, whose end node is given as a parameter, by iterating through the parent references.

    node: search node, instance of Node
    """

    if not node:
        return None

```

```
    path = []
    while node.parent is not None:
        path = path + [node.action]
        node = node.parent
    path.reverse()
    return path
```

Listing A.2: Method for reconstructing the path from the start to the goal node.

A.2 Bidirectional Search

```
def bidirectionalSearch(problem):
    """
    Simultaneously searches from both the start and the goal positions.
    Stops when the two frontiers intersect.
    The shallowest node is taken first (BFS).
    """

    forward = util.Queue()
    start = problem.getStartState()
    exploredForward = {start}
    forward.push((start, []))

    backward = util.Queue()
    goal = problem.goal
    exploredBackward = {goal}
    backward.push((goal, []))

    visitedForward = set()
    visitedBackwards = set()

    while not forward.isEmpty() and not backward.isEmpty():

        # Forward searching
        currentNode, currentActions = forward.pop()

        if currentNode not in visitedForward:
            visitedForward.add(currentNode)
            if currentNode in exploredBackward:
                while not backward.isEmpty():
                    node, actions = backward.pop()
                    if node == currentNode:
                        solution = currentActions + actions.reverse()
                        return solution

            for (childState, childAction, childCost) in problem.
   getSuccessors(currentNode):
                forward.push((childState, currentActions + [childAction]))
                exploredForward.add(childState)

        # Backward searching
        currentNode, currentActions = backward.pop()

        if currentNode not in visitedBackwards:

            visitedBackwards.add(currentNode)
            if currentNode in exploredForward:

```

```
                while not forward.isEmpty():
                    node, actions = forward.pop()
                    if node == currentNode:
                        backwardActions = [reverseAction(action) for action in currentActions]
                        backwardActions.reverse()
                        solution = actions + backwardActions
                        return solution

            for (childState, childAction, childCost) in problem.getSuccessors(currentNode):
                backward.push((childState, currentActions + [childAction]))
                exploredBackward.add(childState)
    return None


def reverseAction(action):
    """Changes the action to its inverse."""

    if action == 'North':
        return 'South'
    elif action == 'South':
        return 'North'
    elif action == 'West':
        return 'East'
    else:
        return 'West'
```

A.3 Depth Limited Search

```
def depthLimitedSearch(problem, limit):
    """
    Search the deepest nodes in the search tree first. The search depth is limited by the given parameter.

    returns a (node, cutoff) tuple:
    - node is the goal node containing reference to its parent.
    - cutoff is True if no solution was found in the given limit, False otherwise.
    """

    start_node = Node(problem.getStartState(), None, 0, None)
    visited = {start_node.state}
    return recursiveDLS(problem, visited, start_node, limit)


def recursiveDLS(problem, visited, node, limit):
    """
    Helper function for depthLimitedSearch(problem, limit).

    returns a (node, cutoff) tuple:
    - node: the goal node containing reference to its parent, or False if no solution was found.
    - cutoff: True if no solution was found in the given limit, False otherwise.
    """

    if problem.isGoalState(node.state):
```

```
            return node, False
        if limit == 0:
            return None, True

        cutoff_occurred = False
        for child_state, child_action, child_cost in problem.getSuccessors(node.
        state):
            if child_state not in visited:

                child = Node(child_state, child_action, child_cost, node)

                visited.add(child_state)
                (result, cutoff) = recursiveDLS(problem, visited, child, limit -
        1)
                visited.remove(child_state)

                if cutoff:
                    cutoff_occurred = True
                elif result:
                    return result, False

        if cutoff_occurred:
            return None, True
        else:
            return None, False
```

A.4 Iterative Deepening Depth-First Search

.

```
def iterativeDeepeningSearch(problem):
    """
    Search the deepest nodes in the search tree first. The search depth is
    limited, but the limit is increased in each
    iteration.

    returns:
    - node: the goal node containing reference to its parent, or False if no
      solution was found.
    """

    depth = 0
    while True:
        (result, cutoff) = depthLimitedSearch(problem, depth)
        if not cutoff:
            return reconstructPath(result)
        depth += 1
```

A.5 Iterative Deepening A*

```
def iterativeDeepeningAStarSearch(problem, heuristic=nullHeuristic):
    """
    Search the node that has the lowest combined cost and heuristic first.
    It differs from A* by limiting the size of
    the frontier, using a bound on the f value.
    """
```

```python

    start = problem.getStartState()
    bound = heuristic(start, problem)
    path = [(start, None, 0)]
    visited = {start}

    while True:
        t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited, bound)

        if t is None:
            return [action for (state, action, cost) in path if action is not None]
        if t == sys.maxint:
            return None

        bound = t


def iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited, bound):
    """
    Utility function for iterativeDeepeningAStarSearch.
    Returns the  minimum cost of all values that exceeded the current bound.
    """
    state, action, cost = path[-1]
    f = cost + heuristic(state, problem)

    if f > bound:
        return f
    if problem.isGoalState(state):
        return None

    min = sys.maxint

    for child_state, child_action, child_cost in problem.getSuccessors(state):
        if child_state not in visited:
            visited.add(child_state)
            path.append((child_state, child_action, child_cost + cost))

            t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited, bound)

            if t is None:
                return None
            if t < min:
                min = t

            path.pop()
            visited.remove(child_state)

    return min
```

A.6 Recursive Best-First Search

```python
def recursiveBestFirstSearch(problem, heuristic=nullHeuristic):
    """
```

```
    Similarly to the DFS, search the deepest nodes in the search tree first, but uses the f_limit variable to keep
    track of the f values. The f value is the largest reached g(n) + h(n) value upon one path after the search is
    stopped because of the f_limit. This f values is backed up to the parent of a node upon backtracking. The f_limit of
    the best child path is the f value of the best alternative path.

    returns:
    - node: the goal node containing reference to its parent, or False if no solution was found.
    """
    start_node = Node(problem.getStartState(), None, 0, None)
    start_node.f = 0
    visited = {start_node.state}
    result, _ = RBFS(problem, heuristic, visited, start_node, float('inf'))

    return reconstructPath(result)


def RBFS(problem, heuristic, visited, node, f_limit):
    """
    Helper function of recursiveBestFirstSearch(problem, heuristic).

    returns:
    - node: the goal node containing reference to its parent, or None if no solution was found.
    - f-cost: the f value obtained on the path.
    """

    if problem.isGoalState(node.state):
        return node, None

    successors = util.PriorityQueueWithFunction(lambda n: n.f)

    for child_state, child_action, child_cost in problem.getSuccessors(node.state):
        if child_state not in visited:
            path_cost = child_cost + node.cost
            child_f = max(path_cost + heuristic(child_state, problem), node.f)
            child = Node(child_state, child_action, path_cost, node, child_f)
            successors.push(child)

    if successors.isEmpty():
        return None, float('inf')

    while True:
        best = successors.pop()

        if best.f > f_limit:
            return None, best.f

        if successors.isEmpty():
            alternative_f = float('inf')
        else:
            alternative = successors.pop()
            alternative_f = alternative.f
            successors.push(alternative)
```

```
        visited.add(best.state)
        result, best.f = RBFS(problem, heuristic, visited, best, min(f_limit, alternative_f))
        visited.remove(best.state)

        if result:
            return result, best.f

        successors.push(best)
```

Appendix B

Source Code: Logics