



Artificial Intelligence

Laboratory activity

Name: Fodor, Zsofia and Katona, Áron

Group: 30433

Email: fodorzsofi@yahoo.com katonaaron01@gmail.com

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



Contents

1	Introduction	3
2	Bidirectional Search	4
2.1	Implementation	4
3	Iterative Deepening Depth-First Search	5
3.1	Depth-Limited Search	5
3.1.1	Implementation	5
3.2	Iterative Deepening Search	5
3.3	Iterative Deepening Depth-First Search	5
4	Iterative Deepening A* Search	7
4.1	Implementation	7
4.2	Remarks	7
5	Recursive Best-First Search	8
6	Comparison	10
6.1	Bidirectional Search vs Breadth-First Search	10
6.2	Iterative Deepening Depth-First Search	10
6.3	Iterative Deepening A* vs A*	10
6.4	Recursive Best-First Search	10
A	Source Code	13
A.1	Nodes	13
A.2	Bidirectional Search	14
A.3	Depth Limited Search	15
A.4	Iterative Deepening Depth-First Search	16
A.5	Iterative Deepening A*	16
A.6	Recursive Best-First Search	18

Chapter 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

Chapter 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 $O(2 * b^{d/2}) = O(b^{d/2}) \ll O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution.

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.

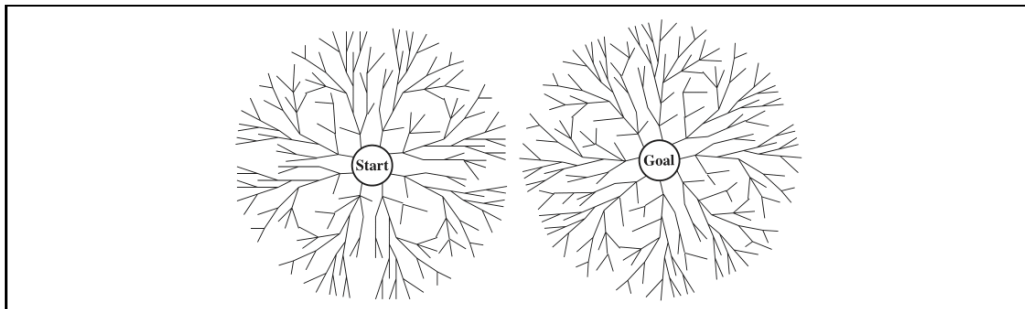


Figure 2.1: Visualization of the bidirectional search algorithm. [1]

Chapter 3

Iterative Deepening Depth-First Search

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.

3.1.1 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.

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 (3.3) and A^* (4).

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.

Chapter 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.

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.

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.

Chapter 5

Recursive Best-First Search

The recursive best-first search, similarly to the depth-limited search (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.

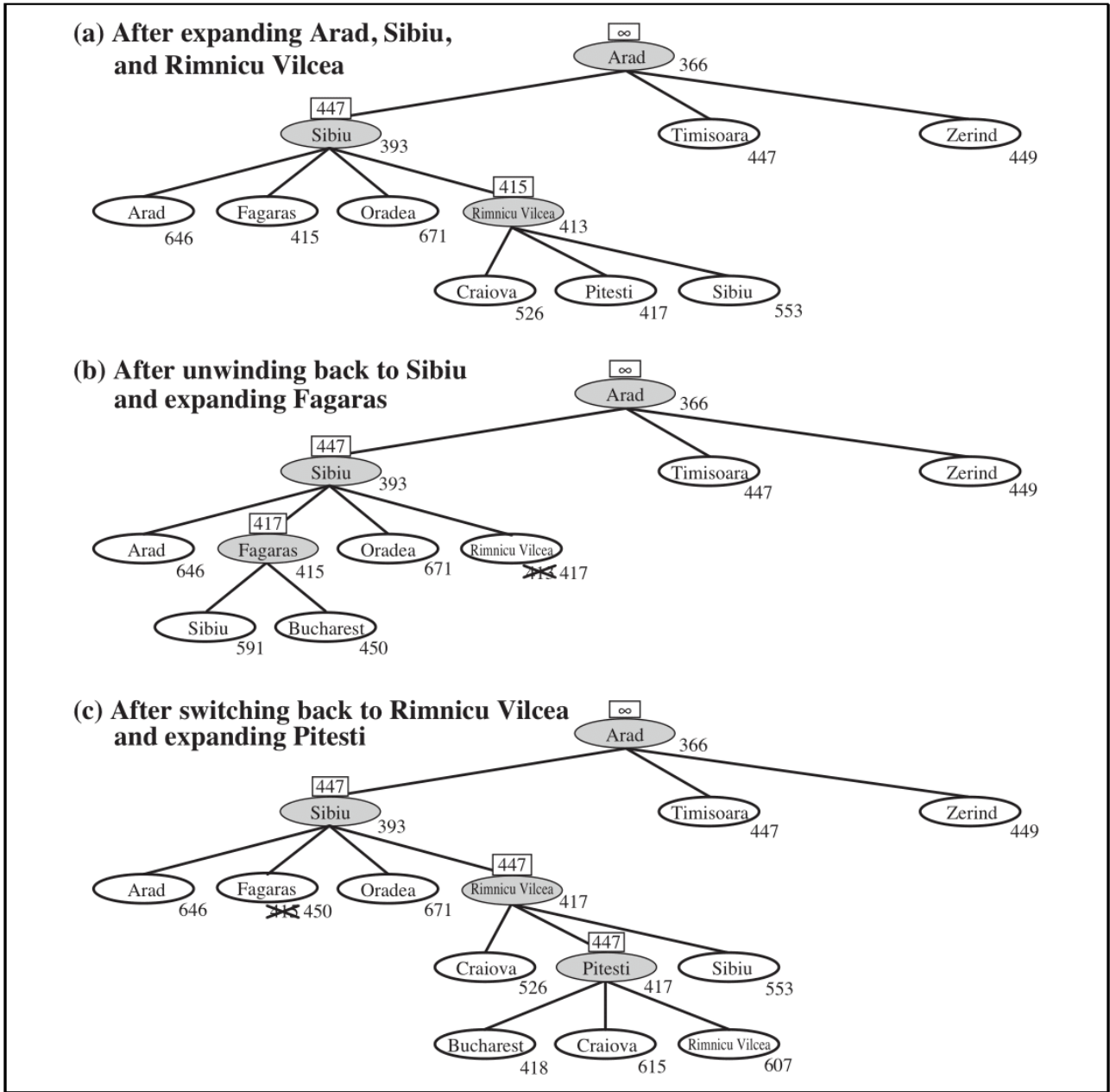


Figure 5.1: Finding a path from Arad to Bucharest by using recursive best-first search. The number above a node is its f-limit, and the number on the right of it is its f-value. The searching branch was switched two times. [1]

Chapter 6

Comparison

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

6.1 Bidirectional Search vs Breadth-First Search

As stated in chapter 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.

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.

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.

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 than the other one, thus increasing the speed.

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

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

A.1 Nodes

To represent the search nodes, we used different strategies across the various search strategies. 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.

```
67 class Node:
68     """
69     Node used for search algorithms.
70     """
71
72     def __init__(self, state, action, cost, parent, f=0):
73         """
74         state: (x,y) coordinates
75         action: direction from Directions of game.py
76         cost: cost of reaching the node
77         parent: parent node
78         f: the backed-up f-value
79         """
80         self.state = state
81         self.action = action
82         self.cost = cost
83         self.parent = parent
84         self.f = f
```

Listing A.1: The Node class

```
87 def reconstructPath(node):
88     """
89     Reconstructs a path, whose end node is given as a parameter, by
90     iterating through the parent references.
91
92     node: search node, instance of Node
93     """
94     if not node:
95         return None
96
```

```

97     path = []
98     while node.parent is not None:
99         path = path + [node.action]
100        node = node.parent
101    path.reverse()
102    return path

```

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

A.2 Bidirectional Search

```

228 def bidirectionalSearch(problem):
229     """
230     Simultaneously searches from both the start and the goal positions.
231     Stops when the two frontiers intersect.
232     The shallowest node is taken first (BFS).
233     """
234
235     forward = util.Queue()
236     start = problem.getStartState()
237     exploredForward = {start}
238     forward.push((start, []))
239
240     backward = util.Queue()
241     goal = problem.goal
242     exploredBackward = {goal}
243     backward.push((goal, []))
244
245     visitedForward = set()
246     visitedBackwards = set()
247
248     while not forward.isEmpty() and not backward.isEmpty():
249
250         # Forward searching
251         currentNode, currentActions = forward.pop()
252
253         if currentNode not in visitedForward:
254             visitedForward.add(currentNode)
255             if currentNode in exploredBackward:
256                 while not backward.isEmpty():
257                     node, actions = backward.pop()
258                     if node == currentNode:
259                         solution = currentActions + actions.reverse()
260                         return solution
261
262                 for (childState, childAction, childCost) in problem.
263                     getSuccessors(currentNode):
264                         forward.push((childState, currentActions + [childAction]))
265                         exploredForward.add(childState)
266
267         # Backward searching
268         currentNode, currentActions = backward.pop()
269
270         if currentNode not in visitedBackwards:
271             visitedBackwards.add(currentNode)
272             if currentNode in exploredForward:
273

```

```

274         while not forward.isEmpty():
275             node, actions = forward.pop()
276             if node == currentNode:
277                 backwardActions = [reverseAction(action) for action
in currentActions]
278                 backwardActions.reverse()
279                 solution = actions + backwardActions
280                 return solution
281
282         for (childState, childAction, childCost) in problem.
getSuccessors(currentNode):
283             backward.push((childState, currentActions + [childAction]))
284             exploredBackward.add(childState)
285         return None
286
287 def reverseAction(action):
288     """Changes the action to its inverse."""
289
290     if action == 'North':
291         return 'South'
292     elif action == 'South':
293         return 'North'
294     elif action == 'West':
295         return 'East'
296     else:
297         return 'West'
298

```

search.py

A.3 Depth Limited Search

```

366 def depthLimitedSearch(problem, limit):
367     """
368     Search the deepest nodes in the search tree first. The search depth is
limited by the given parameter.
369
370     returns a (node, cutoff) tuple:
371     - node is the goal node containing reference to its parent.
372     - cutoff is True if no solution was found in the given limit, False
otherwise.
373     """
374
375     start_node = Node(problem.getStartState(), None, 0, None)
376     visited = {start_node.state}
377     return recursiveDLS(problem, visited, start_node, limit)
378
379 def recursiveDLS(problem, visited, node, limit):
380     """
381     Helper function for depthLimitedSearch(problem, limit).
382
383     returns a (node, cutoff) tuple:
384     - node: the goal node containing reference to its parent, or False if no
solution was found.
385     - cutoff: True if no solution was found in the given limit, False
otherwise.
386     """
387

```

```

388
389     if problem.isGoalState(node.state):
390         return node, False
391     if limit == 0:
392         return None, True
393
394     cutoff_occurred = False
395     for child_state, child_action, child_cost in problem.getSuccessors(node.
state):
396         if child_state not in visited:
397
398             child = Node(child_state, child_action, child_cost, node)
399
400             visited.add(child_state)
401             (result, cutoff) = recursiveDLS(problem, visited, child, limit -
1)
402             visited.remove(child_state)
403
404             if cutoff:
405                 cutoff_occurred = True
406             elif result:
407                 return result, False
408
409     if cutoff_occurred:
410         return None, True
411     else:
412         return None, False

```

search.py

A.4 Iterative Deepening Depth-First Search

```

416 def iterativeDeepeningSearch(problem):
417     """
418     Search the deepest nodes in the search tree first. The search depth is
419     limited, but the limit is increased in each
420     iteration.
421
422     returns:
423     - node: the goal node containing reference to its parent, or False if no
424       solution was found.
425     """
426     depth = 0
427     while True:
428         (result, cutoff) = depthLimitedSearch(problem, depth)
429         if not cutoff:
430             return reconstructPath(result)
431         depth += 1

```

search.py

A.5 Iterative Deepening A*


```

433 def iterativeDeepeningAStarSearch(problem, heuristic=nullHeuristic):
434     """
435     Search the node that has the lowest combined cost and heuristic first.
436     It differs from A* by limiting the size of
437     the frontier, using a bound on the f value.
438     """
439
440     start = problem.getStartState()
441     bound = heuristic(start, problem)
442     path = [(start, None, 0)]
443     visited = {start}
444
445     while True:
446         t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path,
447         visited, bound)
448
449         if t is None:
450             return [action for (state, action, cost) in path if action is
451             not None]
452         if t == sys.maxint:
453             return None
454
455         bound = t
456
457 def iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited,
458 bound):
459     """
460     Utility function for iterativeDeepeningAStarSearch.
461     Returns the minimum cost of all values that exceeded the current bound.
462     """
463
464     state, action, cost = path[-1]
465     f = cost + heuristic(state, problem)
466
467     if f > bound:
468         return f
469     if problem.isGoalState(state):
470         return None
471
472     min = sys.maxint
473
474     for child_state, child_action, child_cost in problem.getSuccessors(state
475 ):
476         if child_state not in visited:
477             visited.add(child_state)
478             path.append((child_state, child_action, child_cost + cost))
479
480         t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path,
481         visited, bound)
482
483         if t is None:
484             return None
485         if t < min:
486             min = t
487
488         path.pop()
489         visited.remove(child_state)
490
491     return min

```

A.6 Recursive Best-First Search

```

301 def recursiveBestFirstSearch(problem, heuristic=nullHeuristic):
302     """
303     Similarly to the DFS, search the deepest nodes in the search tree first,
304     but uses the f_limit variable to keep
305     track of the f values. The f value is the largest reached g(n) + h(n)
306     value upon one path after the search is
307     stopped because of the f_limit. This f values is backed up to the parent
308     of a node upon backtracking. The f_limit of
309     the best child path is the f value of the best alternative path.
310
311     returns:
312     - node: the goal node containing reference to its parent, or False if no
313       solution was found.
314     """
315     start_node = Node(problem.getStartState(), None, 0, None)
316     start_node.f = 0
317     visited = {start_node.state}
318     result, _ = RBFS(problem, heuristic, visited, start_node, float('inf'))
319
320     return reconstructPath(result)
321
322 def RBFS(problem, heuristic, visited, node, f_limit):
323     """
324     Helper function of recursiveBestFirstSearch(problem, heuristic).
325
326     returns:
327     - node: the goal node containing reference to its parent, or None if no
328       solution was found.
329     - f-cost: the f value obtained on the path.
330     """
331
332     if problem.isGoalState(node.state):
333         return node, None
334
335     successors = util.PriorityQueueWithFunction(lambda n: n.f)
336
337     for child_state, child_action, child_cost in problem.getSuccessors(node.state):
338         if child_state not in visited:
339             path_cost = child_cost + node.cost
340             child_f = max(path_cost + heuristic(child_state, problem), node.f)
341             child = Node(child_state, child_action, path_cost, node, child_f)
342             successors.push(child)
343
344     if successors.isEmpty():
345         return None, float('inf')
346
347     while True:
348         best = successors.pop()

```

```

346     if best.f > f_limit:
347         return None, best.f
348
349     if successors.isEmpty():
350         alternative_f = float('inf')
351     else:
352         alternative = successors.pop()
353         alternative_f = alternative.f
354         successors.push(alternative)
355
356     visited.add(best.state)
357     result, best.f = RBFS(problem, heuristic, visited, best, min(f_limit
358 , alternative_f))
359     visited.remove(best.state)
360
361     if result:
362         return result, best.f
363
364     successors.push(best)

```

search.py

Intelligent Systems Group

