University of St Andrews
CS5011
School of Computer Science
KY16 9SX

Practical 2 - Search

Student ID: [redacted]

April 21, 2025
Word Count: 2991 (Source: Overleaf)

1. Introduction

Flight route planning is a critical task in aviation, where efficiency and cost-effectiveness are key. This report examines the implementation and evaluation of various search algorithms to determine optimal flight routes. Using the Oedipus system, a polar-grid-based representation of a planet, we simulate the task of identifying the most efficient route from a departure airport (start state) to a destination airport (goal state) through airspace in polar coordinates.

The implemented algorithms included informed and uninformed search. Each algorithm is assessed based on solution quality (path cost and number of steps) and computational efficiency (time and space complexity). Agents are implemented as graph search algorithms, different from tree search, by maintaining a list of explored nodes to prevent re-visiting [1].

This report explores the theoretical properties, empirical performance, and practical applications of these algorithms to flight route planning, highlighting their strengths and limitations for the problem domain.

2. Design & Implementation Part A

Problem Domain To support modular and reusable code, the search problem is implemented in *FlightRouteProblem.java*, which defines the goal-checking, cost and transition functions. The class is initialised with the map size, initial state, and goal state. Transitions are handled by the *getSuccessors(state)* function, which combines allowed actions and transitions. It returns valid successor coordinates in polar form and is used by *SearchAlgorithm.expand()* to generate only valid nodes. Successors are returned in standard (NoTie) sorting order. The base class *SearchAlgorithm.java*, located in the *search* directory, provides reusable methods such as *expand()* and *print()*. The *expand()* method generates successor nodes with updated path costs. Specific algorithms extend this class by overriding the abstract *search()* method to define their frontier type and behaviour. For sorting, the *util* folder contains custom comparator classes. *TieBreakerComparator*, extending Java's *Comparator*, is used by Breadth First Search (BFS) and Depth First Search (DFS) to sort successors before adding them to the frontier. For Uniform Cost Search (UCS), a separate *UCSComparator* includes node age, the order of expansion, enabling age-aware exploration.

Breadth-First Search The `BFSearch.java` class implements the BFS strategy for flight route planning. It initialises the frontier with the start node and expands the nodes in a breadth-first manner using Java's *Deque* class. To enforce the First-In-First-Out (FIFO) policy, `poll()` retrieves nodes from the head, and `offer()` inserts nodes at the tail. This ensures all nodes at the current depth are explored before proceeding to the next level. If tie-breaking is enabled, successors are prioritised on the basis of criteria such as lower distance and angle. Compared to DFS and UCS, BFS guarantees the shortest path in terms of steps but can be memory-intensive, as it stores all nodes at the current depth. In this problem domain, BFS turned out to be very memory efficient. This implementation of BFS differs from the lecture version by checking the goal state when selecting a node for expansion, similar to other methods. This results in an additional loop, as the goal state is not checked directly on successor nodes, which is feasible in traditional BFS.

Depth-First-Search The `DFSearch.java` class implements Depth-First Search (DFS) for the flight route planning problem. While BFS uses a queue (FIFO) for level-by-level exploration, DFS employs a stack (Last-In-Last-Out (LIFO)) to explore as deeply as possible before backtracking. The frontier is implemented using Java's *Deque* class, with `push()` and `pop()` managing node insertion and removal. Successors are added in reverse order to preserve the intended depth-first traversal. If tie-breaking is enabled, it influences the order of node insertion in DFS, ensuring consistent handling of nodes.

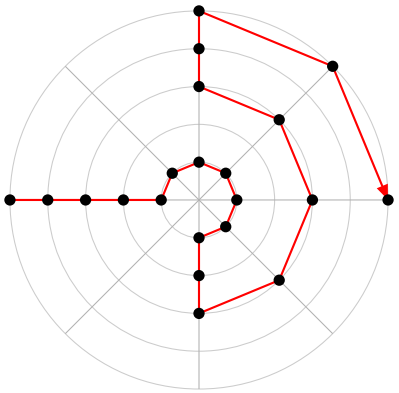

Figure 1: Example of the expansion order of DFS with tie breaking.

It is important to note that both the *Tie* and *No-Tie* options prioritise expansion to the north, biasing the algorithm towards the pole and then outward in a spiral pattern, as illustrated in Figure 1. The key difference is that with tie breaking enabled, DFS first explores every odd-numbered layer of the spiral before expanding southward to the next, missing all goals on evenly numbered layers on the first expansion.

Uniform-Cost Search The UCS algorithm differs from BFS and DFS by expanding the node with the lowest path cost using a priority queue, guaranteeing an optimal cost-based solution. It maintains a cost map (`frontierCosts`) to track the cheapest known path to each state, allowing efficient replacement if a lower-cost route is found. A custom comparator (`UCSComparator`) is used to break ties by preferring newer nodes, determined by an incrementing *age* assigned at expansion. This enables fine-grained ordering when path costs are equal. Unlike BFS, which explores nodes by depth, or DFS, which may follow costly deep paths, UCS prioritises nodes dynamically by cost, improving efficiency in domains with variable path costs like oedipus.

3. Design & Implementation Part B

Best-First Search The `BestFSearch.java` implementation utilises a *PriorityQueue* with a custom *BestFSComparator* that orders nodes based on their f-score. Unlike UCS, which only considers path cost, Best-First-Search (BestF) uses the euclidean distance to the goal as its primary sorting criterion. The implementation maintains a *frontierCosts* map to track the best known costs to each state, similar to UCS. The comparator includes tie-breaking and age-based ordering, ensuring deterministic node expansion when f-scores are equal. However, since it only considers the heuristic value, the algorithm is not guaranteed to find optimal solutions, as will be discussed in Section 6.

A* Search The A-Star Search (A*) implementation closely resembles the structure of BestF, but combines both path cost and heuristic values. In `AStarSearch.java`, each node's f-score is calculated as the sum of the path cost and the Euclidean distance to the goal. The implementation uses the same *BestFSComparator*, but the f-score now represents $f(n) = g(n) + h(n)$. The frontier management and node expansion process mirrors the BestF implementation, but with the crucial difference that A* maintains admissibility by considering actual path costs, ensuring optimal solutions for the Oedipus domain.

SMA* Search The `SMAStarSearch.java` implementation extends A* by incorporating memory constraints via a `maxSize` parameter. It uses two lists: a priority queue for leaf nodes and a *HashMap* for non-leaf (hidden) nodes in the frontier, ensuring efficient node expansion and resulting in worst-case $O(2logx)$ time [2]. Key features include:

- Storing forgotten nodes and their f-scores within parent nodes for later expansion.
- Re-expanding forgotten nodes when necessary, using the forgotten successor list to efficiently update f-scores.
- A priority queue-based approach for leaf nodes, with a culling heuristic to manage memory when the frontier exceeds the limit.

Unlike traditional methods relying on a single priority queue, this implementation efficiently handles both node expansion and culling. The culling heuristic removes the worst leaf node, minimising re-expansion and maintaining efficiency. The forgotten f-cost table further optimises memory usage by storing f-scores of culled nodes for re-expansion with minimal overhead. As described by Lovinger *et al.* [2], the algorithm's memory-efficient culling strategy re-evaluates nodes only when necessary, balancing time and space complexity. This dual priority queue structure ensures optimality while adhering to memory constraints, making `SMAStarSearch.java` suitable for large search spaces with memory limitations.

4. Design & Implementation Part C

Bidirectional A* enhances classical A* by expanding nodes from both the start and goal states, stopping when the frontiers intersect. This reduces the number of node expansions by halving the search depth. The theoretical foundation of bidirectional heuristic search (Bi-HS) lies in the exponential growth of node expansions with depth. Expansion from both ends effectively reduces depth and improves efficiency [3].

Ensuring optimality in bidirectional search is more challenging than in unidirectional search. The primary issue is identifying the correct meeting point and reconstructing the optimal path. Although unidirectional A* ensures optimality with an admissible heuristic, Bi-HS requires stricter termination conditions. Shaham *et al.* [3] addressed this with a must-expand graph, where the minimum expansions correspond to a minimum vertex cover. Empirical results have demonstrated that these algorithms, while theoretically optimal, exhibit variations in runtime depending on factors such as heuristic quality and search-space symmetry, which could explain the marginally higher solution cost in Table 5.

Implementation Details The implementation uses two priority queues for the forward and backward frontiers, employing an enhanced evaluation function based on Shaham *et al.* [3]. The forward search uses the Euclidean distance to the goal as its base heuristic, while the backward search uses the Euclidean distance to the start. The implementation of the main part is shown in Algorithm 1. Key components not shown in the pseudocode include:

- Two priority queues (`forwardFrontier`, `backwardFrontier`) for frontier management
- Separate explored sets (`forwardExplored`, `backwardExplored`) to prevent cycles
- Cost maps tracking the best known path costs in both directions
- Optimality-preserving termination condition

The recalculation of the f-score is done inside *expandForward()* and *expandBackward()* using the balanced approach shown in Equation 1 below:

$$f(n) = \begin{cases} g(n) + h(n), & \text{if no solution found} \\ \max(g(n) + h(n), \frac{C^* + g(n) - g_{opp}(n)}{2}), & \text{otherwise} \end{cases} \tag{1}$$

where C^* is the cost of the best path found so far, and $g_{opp}(n)$ is the path cost from the opposite direction. The search terminates as defined in Equation 2 as seen in line 4 of Algorithm 1.

Algorithm 1 Bidirectional A* Search - Main Loop

```
while not forwardFrontier.isEmpty() and not backwardFrontier.isEmpty() do
    minForwardF ← forwardFrontier.peek().getFScore()
    minBackwardF ← backwardFrontier.peek().getFScore()
    if bestPathCost < ∞ and minForwardF + minBackwardF ≥ bestPathCost − ϵ then
        if meetingPointForward ≠ null and meetingPointBackward ≠ null then
            return constructSolution(meetingPointForward, meetingPointBackward)
        end if
        return null
    end if
    if minForwardF ≤ minBackwardF then
        expandForward()
    else
        expandBackward()
    end if
end while
```

$$f_f(u) + f_b(v) \geq C^* - \epsilon \tag{2}$$

where f_f and f_b are the forward and backward f-scores respectively, guaranteeing optimality of the solution.

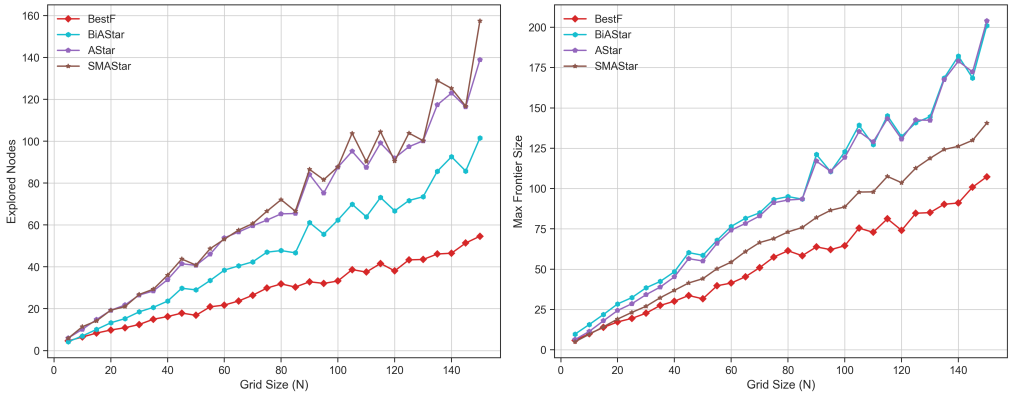

Figure 2: Performance of BiDirectional A* compared to other informed search algorithms.

Empirical performance at runtime varies according to heuristic consistency and the symmetry of the search space [3]. The algorithm performs particularly well in symmetric search spaces such as the Oedipus domain, where both forward and backward heuristics are consistent, showing a reduction in node expansions compared to unidirectional A*, which is shown in Figure 2 on the left. However, in highly asymmetric spaces, the performance advantage diminishes, as one direction may dominate the other in terms of heuristic value, leading to more expansions than necessary.

5. Testing

The following tests provide comprehensive coverage of the base functionality of the problem domain and provided classes. The reason we test the base classes is that the *SearchAlgorithm* class uses the methods provided by those and only overrides the *search()* function. Since it is explicitly stated that coordinates with $d = 0$ are valid coordinates but cannot be reached, the FlightRoute problem accepts those coordinates and handles them as tested in the *Edge Case Tests*. The test cases provide coverage of problem and object initialisation and parameter validation, edge cases and boundary conditions, cost and distance calculations and goal state verification, node linking in search trees and comparison and hash code generation, ensuring

that all valid input coordinates lead to a solution, or terminating gracefully if no solution can be found. Note that *I have passed all the provided public tests (old SMA* version) and my own tests*, as shown in B and C.

5.1. FlightRouteProblem Test Cases

Test Case	Purpose	Pre-conditions	Post-conditions
1. Constructor Validation Tests			
testConstructor	Validate initialization and parameter checks	Grid size N=5, valid/invalid coordinates, tie-breaking flag	Valid params: object created, Invalid: IllegalArgumentException, properties set
2. Successor Generation Tests			
testGetSuccessors	Verify successor state generation	Mid position, near pole (d=1), boundary (d=N-1)	Normal: 4 successors, near pole: 3, boundary: 3
3. Edge Case Tests			
testMinGridsize testSameStartGoal testNorthPole testOuterBounds	Verify behavior in edge cases	Grid size N=1, same start=goal, pole position, boundary	N=1: no successors, same coords: isGoal true, pole: no successors, boundary: valid successors
4. Cost and Goal Tests			
testGetCost testIsGoal	Validate cost and goal detection	Meridian movement, parallel movement, goal coordinate	Correct cost, goal state correctly identified, non-goal rejected

5.2. PolarCoordinate Test Cases

Test Case	Purpose	Pre-conditions	Post-conditions
1. Constructor and Factory Tests			
testConstructor testFromString	Verify object creation and string parsing	Basic inputs, valid/invalid string formats, non-numeric	Valid: object created, Invalid: exception, properties set
2. Validation Tests			
testIsValid	Validate coordinate within grid constraints	Grid size N=5, valid/invalid d values, angles, boundary values	Valid coords: true, invalid d/angle: false, boundary handled
3. Distance Calculation Tests			
testDistanceTo testEuclideanDist	Verify distance calculations between coordinates	Meridian, parallel, arbitrary points, known distances	Arc length, step length, Euclidean distance
4. Object Comparison Tests			
testEquals testHashCode	Verify equality and hash consistency	Equal coords, different coords, nulls, different types	Equal: true, different: false, consistent hash codes, null/type safety

5.3. Node Test Cases

Test Case	Purpose	Pre-conditions	Post-conditions
1. Node Creation Tests			
testConstructor	Verify node creation with initialization	Full constructor, state-only for root, valid PolarCoordinate, valid parent node	Fields set correctly, root: null parent, 0.0 pathCost, "start" action
2. Path Management Tests			
testGetPath testGetStatePath testGetDepth	Verify path tracking and depth calculation	Connected nodes A→B→C, known path costs, known states, linear structure	Correct node/state sequence, correct depth values, path integrity maintained

6. Experiments & Analysis

To conduct the experiments, Table 4 shows the setup used. After carrying out the first experiments using the start and end points given in the appendix (results in A), a more thorough analysis was conducted to gain more insight into each algorithm for increasing grid sizes and more start/end conditions. The resulting trends discussed in this section align with the results of the first experiments (Figure 6), with the only difference being that the solution percentage of Simplified Memory Bounded A-Star Search (SMA*) is only 30% and the two cases using pole coordinates fail for all algorithms. Table 5 shows the average results

Parameter	**Setting**
Grid sizes	5 to 150 in increments of 5
Start/goal configurations	100 randomly generated valid polar coordinates, d (1 to $N-1$), angle: multiple of 45°
Tie-breaking	With and without tie-breaking enabled
Experimental Environment	
Language & compiler	Java (OpenJDK 23.0.2, also tested on OpenJDK 17)
Computer specs	Apple M3 (ARM64), 16 GB RAM, macOS 24.3.0 (also tested on Lab Machine)

Table 4: Summary of experimental setup and environment.

over all grid sizes for the experiments carried out. SMA* was initialised with a memory bound equal to the size of the grid, resulting in the algorithm finding a solution only 68% of the time during the experiments, resulting in lower values for *Goal Depth* and *Path Cots*, as 32% of the paths were too long to find a solution with SMA*.

6.1. Empirical Results

Space Complexity To analyse the space complexity, the average and maximum frontier size during the search process is evaluated and presented in Figure 3. DFS explores significantly longer paths than BFS, requiring more memory to support backtracking. This leads to a considerably higher space complexity for DFS. The shaded region for DFS shows its high standard deviation, reflecting the variability of the paths in 100 test inputs. This stood out compared to all other algorithms and the reasons can be found in the way DFS expands outwards, as shown in Figure 1 and explained in the time complexity analysis. Empirical results show that all algorithms perform well below their theoretical worst-case bounds. While BFS has

-	Use Tie	Explored	Goal Depth	Path Cost	Frontier Size	Max Depth
BFS	False	300.26	28.03	*75.71*	15.36	28.03
BFS	True	300.38	28.03	*69.38*	15.29	28.03
DFS	False	291.59	154.28	1 778.85	219.14	232.54
DFS	True	297.92	157.83	2 670.02	218.40	236.67
UCS	Both	319.94	46.27	63.33	111.58	64.67
BestF	Both	28.09	28.09	88.33	54.00	28.09
A*	Both	65.54	46.27	63.33	93.01	48.45
SMA*	False	*67.92*	24.33	41.79	70.64	42.10
SMA*	True	*68.85*	24.33	41.78	70.64	42.05
BiA*	Both	48.70	42.21	65.24	95.09	43.99

Table 5: Average number of explored nodes, path length, path cost, maximum frontier size, and maximum depth for each algorithm (with and without tie-breaking).

a theoretical space complexity of $O(b^d)$, it exhibits sub-linear frontier growth. DFS, with a worst-case of $O(b^m)$, scales approximately linearly but maintains larger frontiers due to deeper average paths, over five times longer than those found by BFS.

BestF and A* also exhibit sub-linear frontier scaling, despite theoretical exponential bounds. BestF demonstrates better space efficiency than A* due to finding shallower solutions at higher costs, which is outlined in the Optimality Analysis (6.1). UCS and A* have comparable space complexity due to both looking at the path cost.

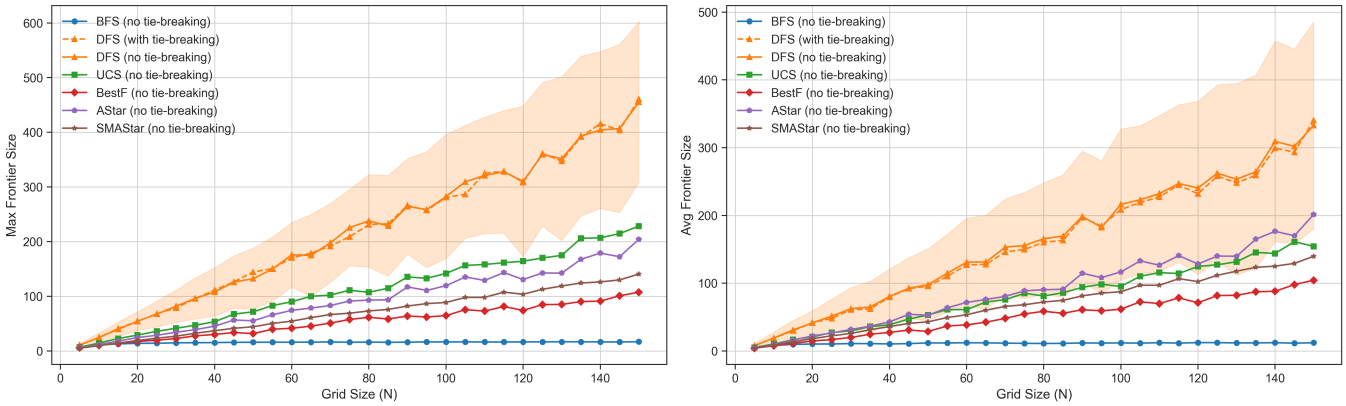

Figure 3: Maximum and average frontier sizes during testing.

Time Complexity Figure 4 reports the average number of nodes expanded across grid sizes for all algorithms, which serves to approximate the time taken for each algorithm. Empirical results show substantial improvements over theoretical worst-case bounds, especially for informed search methods, which outperform the uninformed search.

BFS, DFS, and UCS, despite their theoretical exponential complexity, exhibit near-linear scaling in practice. However, DFS shows significant variability, as indicated by its standard deviation in Figure 4. Although its average performance appears comparable to BFS and UCS, this masks a key behaviour: DFS prioritises northward expansion before spiralling outward (Section 2), leading to inconsistent performance depending on goal location. When the goal lies in an even-numbered layer, DFS often expands the entire spiral before backtracking, skewing the average results due to the placement of the goal.

Informed algorithms show greater efficiency gains. A*, despite a theoretical $O(b^d)$ bound, achieves sub-linear node expansion, highlighting the strength of its heuristic. BestF explores the fewest nodes overall, though at the cost of optimality. SMA*, while memory-bounded, performs comparably to A*, indicating

that memory constraints do not constrain the performance, but limit the percentage of goals reached.

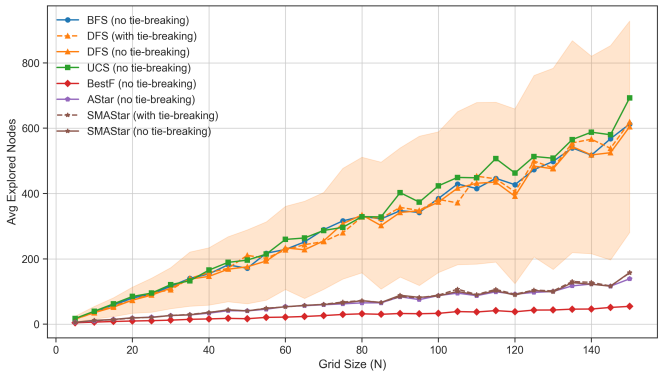

Figure 4: The number of explored nodes as an estimate of time complexity of the algorithms.

Optimality Analysis UCS always returned optimal paths, as guaranteed by its cost-driven expansion. BFS found the shortest-step solution on unweighted grids but does not guarantee cost optimality. DFS frequently returned suboptimal routes, especially in situations where the goal is on an evenly numbered layer. SMA* seems to find better paths than A*, but this is related to failing to find a solution for path depths exceeding the memory bound. The higher we set the maximum frontier size, the closer we get to approach the performance of A*. BestF finds shallower goals on average compared to A*, because for grid sizes greater than 6, moving between meridians reduces the euclidean distance by greater than 1, so BestF prefers to use the arcs for traversal. The direct route using A* is more costly in terms of depth, as only 4 steps are needed to go around the arc to the other side, but $2 * N$ steps are needed to go through the centre, but the resulting covered distance is lower due to the more direct path.

Effect of Tie-Breaking Figure 3 shows that the average frontier size is slightly lower for DFS if tie breaking is enabled, but Table 5 paints a clearer picture, showing that tie breaking results in a higher path cost for DFS, because it encourages the spiral exploration pattern. Tie breaking also has a minor influence on BFS and UCS, indicated in Table 5, but does not affect informed search.

7. Evaluation and Conclusion

This report evaluates various search algorithms for flight route planning within the Oedipus airspace domain. The tested algorithms include uninformed and informed searches like A* and BFS. Performance was assessed based on solution quality (path cost and number of steps) and computational efficiency (nodes explored and frontier size).

The results show that BFS and BestF consistently returned the shallowest solutions, with BFS being memory-efficient but exploring more nodes in general. A* and UCS found optimal solutions in terms of path cost, with UCS exploring ten times more nodes than A*. Bidirectional A* improved time efficiency compared to A* by reducing node exploration, highlighting the potential of bidirectional search in complex environments. DFS performed the worst in time and space complexity, proving suboptimal for this problem. SMA* performed similarly to A*, but memory limitations hindered solutions for more distant goals.

To conclude, performance differences between informed and uninformed search were most notable in time complexity, but BFS outperformed informed searches in space complexity due to the domain structure. A* and Bidirectional A* were the most efficient overall, achieving optimal solutions with competitive time and space complexity.

References

[1] Stuart J. Russell, Peter Norvig, and Ernest Davis, *Artificial intelligence: a modern approach* (Prentice Hall series in artificial intelligence), 3rd ed. Upper Saddle River: Prentice Hall, 2010, 1132 pp., ISBN: 9780136042594.

[2] Justin Lovinger and Xiaoqin Zhang, "Enhanced simplified memory-bounded a star (SMA*+)," presented at the GCAI 2017. 3rd Global Conference on Artificial Intelligence, Oct. 2017, pp. 202–190. DOI: 10.29007/v7zc. [Online]. Available: https://easychair.org/publications/paper/TL2M (visited on 04/14/2025).

[3] Eshed Shaham, Ariel Felner, Nathan R. Sturtevant, *et al.*, "Optimally efficient bidirectional search," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 6221–6225, ISBN: 978-0-9992411-4-1. DOI: 10.24963/ijcai.2019/867. [Online]. Available: https://www.ijcai.org/proceedings/2019/867 (visited on 04/20/2025).

A. Assignment Start/Goal Pairs

-	Use Tie	Explored	Goal Depth	Path Cost	Frontier Size	Max Depth
BFS	False	43.28	6.06	8.24	9.53	7.75
BFS	True	43.53	6.06	8.24	9.39	7.75
DFS	False	35.92	18.89	35.26	22.44	28.69
DFS	True	34.69	18.00	35.32	22.56	27.47
UCS	Both	45.06	8.00	7.20	11.00	11.17
BestF	Both	17.58	6.67	8.20	11.67	9.28
A*	Both	19.56	8.00	7.20	12.69	10.61
SMA*	False	14.44	1.33	1.67	7.67	7.17
SMA*	True	10.33	1.33	1.67	7.67	6.44
BiA*	Both	9.75	8.00	7.20	16.31	8.31

Table 6: Average number of explored nodes, path length, path cost, maximum frontier size, and maximum depth for each algorithm (with and without tie-breaking) evaluated on the start/goal pairs given in the assignment handout.

B. Stacscheck Results (Old SMAStar tests)

```
Testing CS5011 P2 Practical
- Looking for submission in a directory called 'src': found in current directory
* BUILD TEST - InvalidInputs/build-all : pass
* COMPARISON TEST - InvalidInputs/00_Wrong-No-Tie0/prog-run-00_Wrong-No-Tie0.out : pass
* COMPARISON TEST - InvalidInputs/01_Extra-Args0/prog-run-01_Extra-Args0.out : pass
* COMPARISON TEST - InvalidInputs/02_Less-Args0/prog-run-02_Less-Args0.out : pass
* COMPARISON TEST - InvalidInputs/03_Negative-Oedipus0/prog-run-03_Negative-Oedipus0.out : pass
* COMPARISON TEST - InvalidInputs/04_Negative-Start-d0/prog-run-04_Negative-Start-d0.out : pass
* COMPARISON TEST - InvalidInputs/05_Big-Start-d0/prog-run-05_Big-Start-d0.out : pass
* COMPARISON TEST - InvalidInputs/06_Big-Start-angle0/prog-run-06_Big-Start-angle0.out : pass
* COMPARISON TEST - InvalidInputs/07_Not-45x-Start-angle0/prog-run-07_Not-45x-Start-angle0.out :
↪  pass
* BUILD TEST - NoTie/build-all : pass
* COMPARISON TEST - NoTie/00_BFS0/prog-run-00_BFS0.out : pass
* COMPARISON TEST - NoTie/01_BFS1/prog-run-01_BFS1.out : pass
* COMPARISON TEST - NoTie/02_BFS2/prog-run-02_BFS2.out : pass
* COMPARISON TEST - NoTie/03_BFS3/prog-run-03_BFS.out : pass
* COMPARISON TEST - NoTie/04_DFS0/prog-run-04_DFS0.out : pass
* COMPARISON TEST - NoTie/05_DFS1/prog-run-05_DFS1.out : pass
* COMPARISON TEST - NoTie/06_DFS2/prog-run-06_DFS2.out : pass
* COMPARISON TEST - NoTie/07_UCS0/prog-run-07_UCS0.out : pass
* COMPARISON TEST - NoTie/08_UCS1/prog-run-08_UCS1.out : pass
* COMPARISON TEST - NoTie/09_UCS2/prog-run-09_UCS2.out : pass
* COMPARISON TEST - NoTie/10_BESTF0/prog-run-10_BESTF0.out : pass
* COMPARISON TEST - NoTie/11_ASTAR0/prog-run-08_ASTAR0.out : pass
* COMPARISON TEST - NoTie/12_SMASTAR0/prog-run-09_SMASTAR0.out : pass
* COMPARISON TEST - NoTie/13_SMASTAR1/prog-run-10_SMASTAR1.out : pass
* COMPARISON TEST - NoTie/14-SMAStar-TCONF0/prog-run-14-SMAStar-TCONF0.out : pass
* COMPARISON TEST - NoTie/15-SMAStar-TCONF1/prog-run-15-SMAStar-TCONF1.out : pass
* BUILD TEST - TieBreaking/build-all : pass
* COMPARISON TEST - TieBreaking/00_BFS0/prog-run-00_BFS0.out : pass
* COMPARISON TEST - TieBreaking/01_BFS1/prog-run-01_BFS1.out : pass
* COMPARISON TEST - TieBreaking/02_BFS2/prog-run-02_BFS2.out : pass
* COMPARISON TEST - TieBreaking/03_BFS3/prog-run-03_BFS.out : pass
* COMPARISON TEST - TieBreaking/04_DFS0/prog-run-04_DFS0.out : pass
* COMPARISON TEST - TieBreaking/05_DFS1/prog-run-05_DFS1.out : pass
* COMPARISON TEST - TieBreaking/06_DFS2/prog-run-06_DFS2.out : pass
* COMPARISON TEST - TieBreaking/07_UCS0/prog-run-07_UCS0.out : pass
* COMPARISON TEST - TieBreaking/08_UCS1/prog-run-08_UCS1.out : pass
* COMPARISON TEST - TieBreaking/09_UCS2/prog-run-09_UCS2.out : pass
* COMPARISON TEST - TieBreaking/10_BESTF0/prog-run-10_BESTF0.out : pass
* COMPARISON TEST - TieBreaking/11_ASTAR0/prog-run-08_ASTAR0.out : pass
* COMPARISON TEST - TieBreaking/12_SMASTAR0/prog-run-09_SMASTAR0.out : pass
* COMPARISON TEST - TieBreaking/13_SMASTAR1/prog-run-10_SMASTAR1.out : pass
* COMPARISON TEST - TieBreaking/14-SMAStar-TCONF0/prog-run-14-SMAStar-TCONF0.out : pass
* COMPARISON TEST - TieBreaking/15-SMAStar-TCONF1/prog-run-15-SMAStar-TCONF1.out : pass
43 out of 43 tests passed
```

Figure 5: Terminal Output of Stacscheck tests (containing the older version of SMAStar tests.

C. Unitest Terminal Output

```
(venv) (base) fp@user CS5011-P2-code % ./run.sh test
Compiling the project...
Compilation successful!
Compiling unit tests...
Test compilation successful!

Running unit tests...

PolarCoordinate tests:
JUnit version 4.13.2
.........
Time: 0,003

OK (9 tests)


Node tests:
JUnit version 4.13.2
.....
Time: 0,003

OK (5 tests)


FlightRouteProblem tests:
JUnit version 4.13.2
.............
Time: 0,005

OK (13 tests)
```

Listing 1: Testing output in terminal.