CS5011 - Practical 1

[190026921]

Feb 09, 2022

Word Count: 2030

Contents

1 Introduction and Overview **1**

1.1 Implementation status 1

1.2 Compiling and running instructions 1

2 Design and Implementation **2**

2.1 PEAS model and problem definition 2

2.2 System architecture 3

2.3 Key implementation details 4

3 Test Summary **7**

4 Evaluation and Conclusions **7**

4.1 Basic, intermediate, and advanced agents 7

4.2 Advanced agent in detail 9

4.3 Concluding remarks 11

5 Appendix **12**

1 Introduction and Overview

The objective of this practical was the implementation of several search algorithms in the context of a marine navigation route planner: a ferry agent needs to make their way across a grid consisting of triangular shapes, given a starting cell and a goal cell.

1.1 Implementation status

All requirements were implemented, including the advanced ferry agent.

- **Basic ferry agent:** Attempted, fully working
- **Intermediate ferry agent:** Attempted, fully working
- **Advanced ferry agent:** Attempted, fully working; implemented bidirectional search with different search algorithms for backward search

1.2 Compiling and running instructions

1.2.1 Basic usage

As required by the specification, the program can be compiled and run from the `src` directory with the following commands:

```
javac *.java
java A1main <Algo> <ConfID> [<2nd Algo>]
```

where

Parameter	Meaning	Options
`<Algo>`	Search algorithm	BFS | DFS | BestF | AStar | BDS
`<ConfID>`	Configuration	JCONF00 | ... | JCONF05 | CONF0 | ... | CONF24
`<2nd Algo>`	2nd Algo for BDS	BFS | DFS | BestF | AStar

Note that you can only specify the `<2nd Algo>` parameter if you are using bidirectional search (`Algo=BDS`) (see Section 2).

1.2.2 Advanced usage (testing and evaluation)

I further implemented a number of JUnit tests as well as my evaluation, which require the user to specify the Java class path during compilation and running. A Makefile is provided with the relevant commands.

All Make commands must be run from the project root (containing the `src` directory) and `make compile` must be run first to compile the project.

The following commands are available:

```
compile      Compile all java files
clean        Remove java class files
test         Run JUnit tests
stacscheck   Run stacscheck checks (only on school server)
evaluation   Run the evaluation (run experiments, create plots)
help         Print available commands
```

Note, that it is only necessary to compile the project with `make compile` if you want to run the unit tests or the evaluation. Note that the graphs are generated using a python script (requiring the libraries specified in `evaluation/requirements.txt`).

2 Design and Implementation

2.1 PEAS model and problem definition

The PEAS model can be applied to the problem at hand as follows:

- **Performance measures:** Two performance measures are used: path cost and number of explored nodes. The path cost is the number of steps necessary to reach the goal cell from the start. The number of explored nodes is the total number of nodes (or cells) that had to be visited before a path to the goal is found. While the path cost can serve as a measure of *effectiveness* of the agent, the number of explored nodes relates to the agent's *efficiency.*
- **Environment:** The environment in this problem is deterministic, fully observable, static and known. The agent operates on a `N`×`N` grid of triangles in alternating orientation (see Figure 1), starting with an upward pointing triangle in cell `(0,0)`. The agent is given a start coordinate and goal coordinate given in the format `(row, column)`. Some cells on the grid may be islands, which the agent cannot enter or pass through.
- **Actuators:** The agent can move in the direction of the edges of the triangle, but not across the corners, one cell at a time. In other words, for upward pointing triangles, the agent can move one cell to the right, bottom, or left. For downward pointing triangles, the agent can move to the right, left, or up.
- **Sensors:** The agent has a sensor that asserts if the current cell is the goal cell, thereby ending the search. The agent further is aware of which cells have been visited before.

The problem is defined as follows. Given an agent, a map, a start coordinate, and a goal coordinate, which order of valid actions can be taken for the agent to reach the goal from the start.

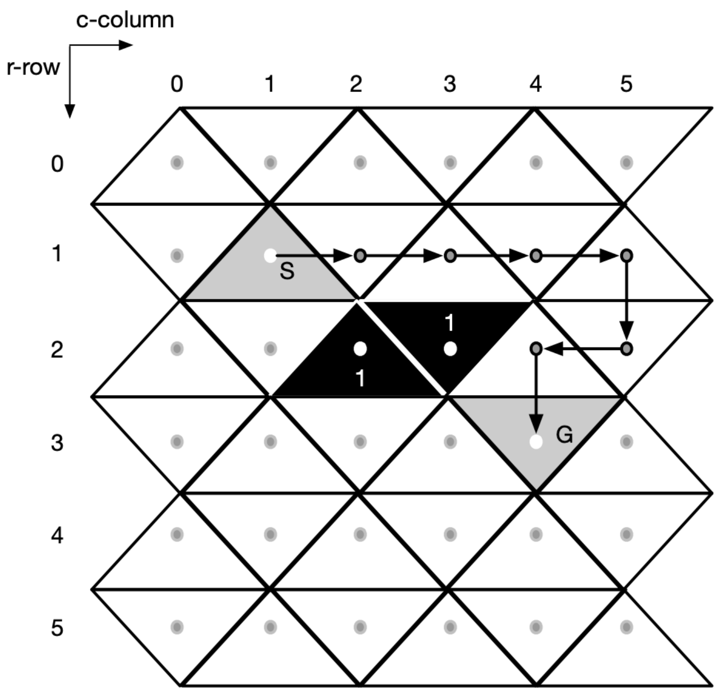

Figure 1: Example map as given in the practical specification.

2.2 System architecture

Across the system, I make extensive use of polymorphism and abstract classes for the different search algorithms. The full class diagram of all search algorithms is given in Figure 2.

Crucially, all implemented search algorithms are a modification of the general search algorithm as discussed in the lectures. As such, it was possible to implement all algorithms as sub-classes of the general search and only make minor modifications by overriding certain methods. For `BestFirstSearch` and `AStarSearch` I created an additional abstract class `InformedSearch` as the two methods make some common modifications to the general search.

Setting up the code in this polymorphic way has several advantages.

First and foremost, it makes the implementation of new search algorithms extremely efficient and uncomplicated, since any new algorithm only needs to implement the exact parts that change compared to the general search. This can go so far that two different algorithms might only need to provide a custom constructor while keeping all other methods the same. For instance, `BreadthFirstSearch` and `BestFirstSearch` only differ in terms of which frontier they specify.

Secondly, the common abstract class `Search` acts as a promise as to which methods are provided by any of the implemented algorithms, thus leveraging the polymorphisms. The reason I created two different classes for `Search` and `GeneralSearch` is that this allows for specifying an alternative set of search algorithms not based on the general search with the same interface.

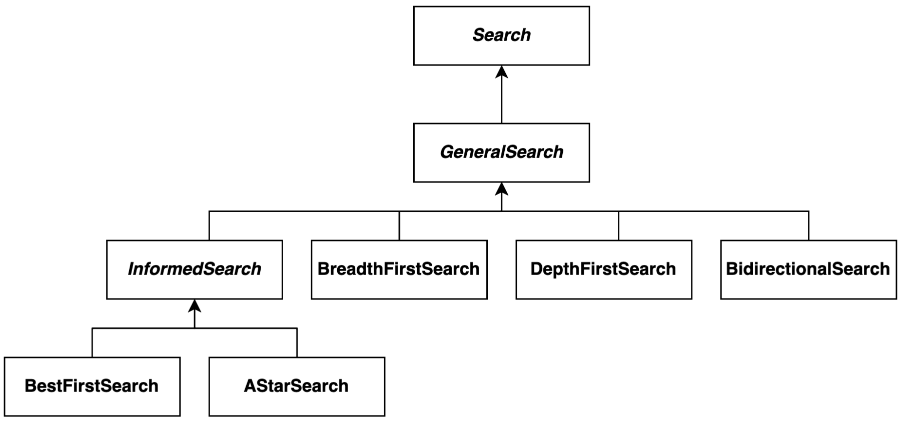

Figure 2: Class diagram of the Search classes.

A similar approach was taken for the implementation of the frontiers (see Figure 3). A common abstract parent class `Frontier` is defined providing a uniform interface for its sub-classes, `StackFrontier`, `QueueFrontier`, and `PriorityQueueFrontier`. They only differ in their underlying data structure (i.e. stack, queue, or priority queue). The polymorphism of the `Frontier` class enables much of the structure and sub-classing in the `Search` classes.

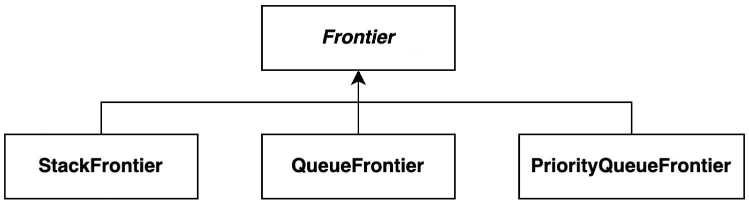

Figure 3: Example map as given in the practical specification.

2.3 Key implementation details

When the user runs the program, the requested algorithm is processed by an abstract factory that returns an instance of the corresponding search algorithm class (if it exists, else an exception is thrown). Due to the polymorphic nature of the `Search` sub-classes described earlier, all implement the `runSearch` method to start the search.

The `runSearch` method is implemented in the `GeneralSearch` class and used by all search algorithms and only overridden by bidirectional search (discussed later). The procedure defined here corresponds to that discussed in the lectures, and is further given in Algorithm 1. It initializes the algorithm by adding the start node to the frontier and then performs iterations as long as the frontier contains nodes.

Note that the actual iteration of the main loop is a separate function to allow sub-classes to override it in case modifications are necessary. The search iteration for the general case is provided in Algorithm 2.

Algorithm 1 runSearch procedure

```
procedure runSearch
    insert startNode into frontier                     ▷ initial node is added
    while frontier not empty do                        ▷ as long as there are nodes to be explored
        if searchIteration() then
            return                                     ▷ search iteration terminated (success)
        end if
    end while
    return                                             ▷ No solution found
end procedure
```

Algorithm 2 searchIteration procedure

```
procedure searchIteration
    remove Node from frontier
    add Node to explored
    if terminationCheck() then                         ▷ termination check depends on algorithm
        return true
    end if
    expand newNodes from Node
    insert all newNodes into frontier                  ▷ insert behaviour depends on algorithm
    return false
end procedure
```

How do we implement the different search algorithms from this general search? In the simplest cases, `BreadthFirstSearch` (BFS) and `DepthFirstSearch` (DFS), it suffices to initialize the class with a different frontier. In the case of the BFS, this is a queue, in the case of DFS, a stack. For example, the constructor for BFS would look comparable to Algorithm 3. This change in the constructor alone is a sufficient modification to the general search algorithm since the general search algorithm calls the generic `insert` and `remove` methods on the respective frontier. All the logic as to which nodes are inserted or removed upon calling this method are purely handled by the frontier defined in the constructor. In the case of the queue frontier of BFS, this means nodes are returned in a FIFO manner, while the stack frontier of DFS operates on a LIFO policy.

Algorithm 3 BreadthFirstSearch constructor

```
procedure constructBFS(map, start, goal)
    super(map, start, goal)
    frontier = new QueueFrontier()
end procedure
```

For the informed searches `BestFirstSearch` (BestF) and `AStarSearch` (AStar), an additional modification is necessary. Before new nodes are inserted into the frontier, their costs ought to be calculated. For BestF this is the Manhattan distance and for AStar the Manhattan distance plus the path cost for each node to be inserted.

Algorithm 4 expand procedure

```
procedure EXPAND(Node)
    get successorStates from Node
    initialize array newNodes
    for all State in successorStates do
        make newNode with State                          ▷ add newNode to newNodes
        ADDTOSUCCESSORS(NEWNODE, NEWNODES)               ▷ if a condition is met
    end for
    return newNodes
end procedure
```

Lastly, AStar modifies `GeneralSearch` by changing the `addToSuccessors` method that is called from the `expand` procedure, which creates an array of successor nodes to the current node to be added to the frontier (given in Algorithm 4). While in the general case, the `addToSuccessors` adds a node to the array of successor nodes unless that node has already been explored, the AStar algorithm expands that behaviour (see Algorithm 5). If the node has not been explored yet AND the frontier does not contain the node already, the node is a successor (so far, no difference). However, if this is NOT the case, the method checks if the node already exists in the frontier, but with a higher path cost. If so, the node already in the frontier is replaced with the updated version of the node (with the lower path cost).

Algorithm 5 AStarSearch.addToSuccessors procedure

```
procedure ASTARSEARCH.ADDTOSUCCESSORS(Node, newNodes)
    if Node not explored AND not in frontier then        ▷ Same as in general search
        add Node to newNodes and return ;
    end if
    for all existingNode in frontier do                  ▷ This is the adaptation for AStar
        if existingNode equals Node then
            replace existingNode with Node in frontier
        end if
    end for
end procedure
```

In Algorithm 4 we glossed over the details of how we get the `successorStates` from the current node. In detail, a `getSuccessorStates` method has been implemented in the `State` class, that returns an array of all states on adjacent coordinates (if on the map and not an island). This method is used during the `expand` procedure to create the array of successor states.

Advanced Agent: Bidirectional Search

For the advanced agent, bidirectional search was implemented. The main idea behind the algorithm, as detailed in [Norvig and Russel, 2010, p.90], is to run two simultaneous searches:

a forward search from start to goal, and a backward search from goal to start. Once the two search frontiers meet, we can construct a full path from the two partial paths. Bidirectional search is very appealing in the case at hand, since the forward and backward search can make use of any of the previously implemented algorithms.

`BidirectionalSearch` (BDS) is instantiated with two search algorithms - one for forward and one for backward search. Forward search always uses BFS to ensure that the searches meet before reaching their respective goal. The second algorithm can be set to BFS, DFS, BestF, or AStar with the additional command line argument (see Section 1).

The `runSearch` and `searchIteration` methods of BDS follow the same steps as in general search, however, each step is executed both on the forward and the backward search instance. Furthermore, the search iteration now includes a `frontierIntersectionCheck`, that ends the search if the frontiers of the two searches meet.

Once forward and backward searches intersect in their frontiers, the solution path needs to be constructed. The `constructPath` method does so by traversing the backward search tree in order (i.e. from intersection point to the goal) and appending each node to the forward search tree.

3 Test Summary

To test the correctness of the application, the majority of methods and classes were tested extensively with JUnit unit tests with a special focus on critical sections. Overall, I wrote 26 unit tests (all of which are passing). A test coverage report is included in Table 3 in the appendix as well as in the `doc/coverage` folder of the submission.

Note that some of the JUnit tests are of a much broader scope than a typical unit test. For example, `SearchTest.testRunAll` runs all available algorithms on all configurations, ensuring that for each configuration, the results line up.

In addition to the JUnit test, I used `stacscheck` with the six provided test cases for BFS, all of which are passing.

4 Evaluation and Conclusions

In this last part, the implemented search strategies are evaluated in terms of their efficiency (number of explored nodes) and their effectiveness (path cost).

4.1 Basic, intermediate, and advanced agents

Figure 4 compares the path costs achieved by the algorithms for the 25 given configurations. Note that among the algorithms in this Figure, BFS, AStar and BDS (since it uses BFS in both directions) always find the optimal path, which is why they all yield the same lowest path cost.

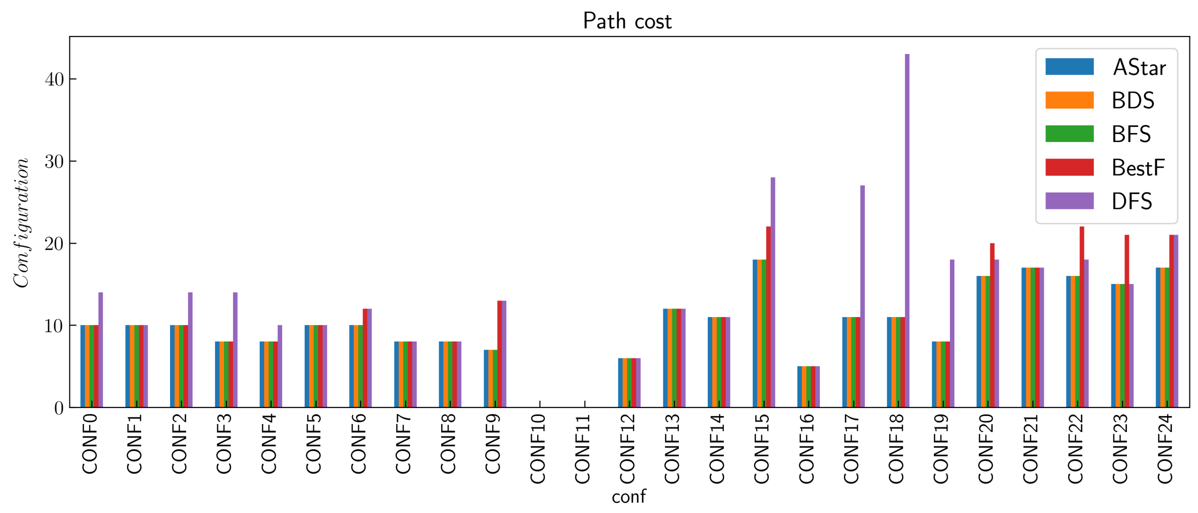

Figure 4: Path cost for 25 given configurations. BDS uses BFS in both directions.

Ranking the algorithms for each configuration w.r.t their path cost, the optimal algorithms (obviously) always rank first, while BDS has a mean rank of 1.61 and BestF of 1.39, making BestF slightly more effective in the evaluated scenarios. These results are given in Table 1.

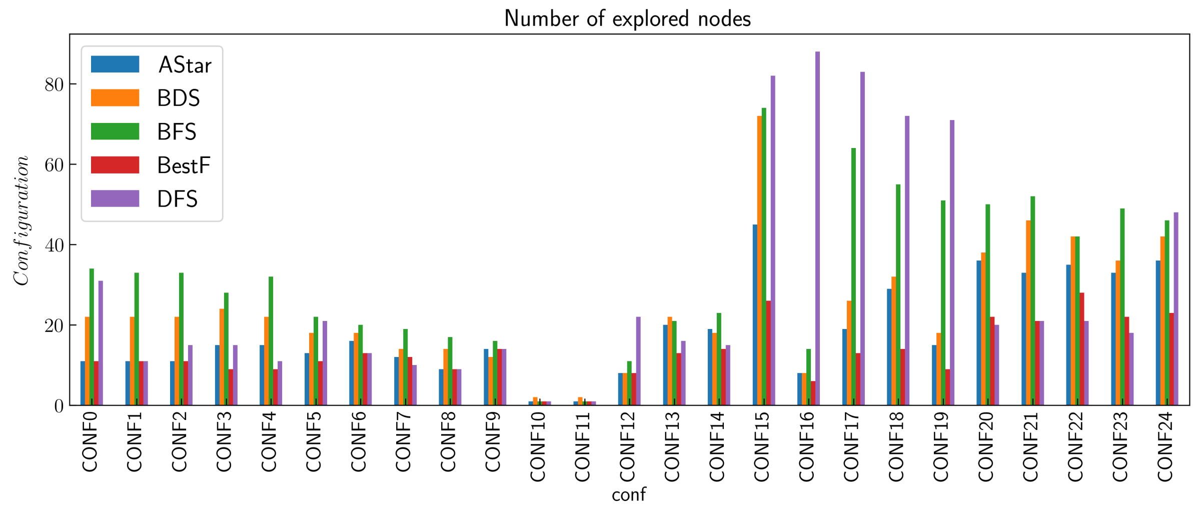

Figure 5: No. of explored nodes for 25 given configurations. BDS uses BFS in both directions.

Figure 5 considers the same five algorithms on the 25 configurations, but in terms of the number of explored nodes. Here, the picture is less obvious, with the performance of the algorithms varying quite drastically across configurations. Ranking them in terms of their node count for each configurations gives some clarity: on average, BestF ranks best (mean 1.2, median 2), followed by AStar (mean 2, median 2), while BFS does worst in terms of efficiency (mean 3.72, median 4). Interestingly, BDS also performs poorly (mean 2.84, 3.0), however, this is likely due to the use of BFS for both search directions. The ranking results are given in Table 1.

Table 1: Average and median rank in terms of path cost and node count across the 25 given configurations. BDS uses BFS in both directions.

	Path Cost (Rank)		**Node Count (Rank)**	
	mean	**median**	**mean**	**median**
BFS	1.0	1.0	3.72	4.0
DFS	1.61	2.0	2.44	2.0
BestF	1.39	1.0	1.2	1.0
AStar	1.0	1.0	2.0	2.0
BDS	1.0	1.0	2.84	3.0

4.2 Advanced agent in detail

Comparing different BDS agents by their backward search strategy used, Figure 6 gives the path cost for the 25 configurations. Again, we know that only the BDS-BFS is guaranteed to be optimal, which is why it non-surprisingly ranks first. BDS-AStar and BDS-BestF perform similarly, while BDS-DFS ranks lowest on average (see Table 2).

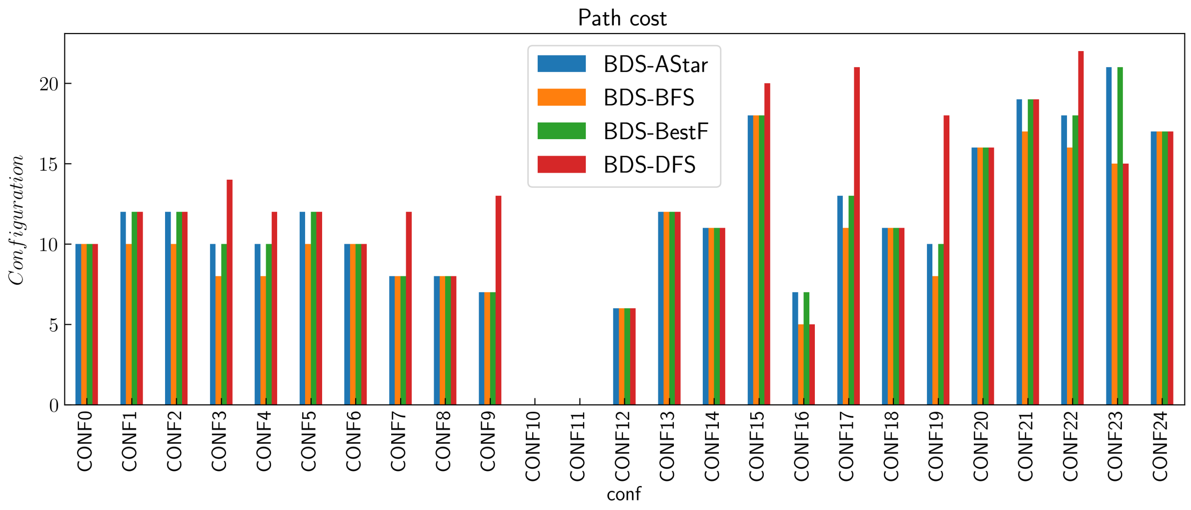

Figure 6: Path cost for 25 given configurations using BDS.

The node counts for the BDS configurations are presented in Figure 7. BDS-DFS appears to explore the least number of nodes on average, which is confirmed by considering the average rank (see Table 2). BDS-AStar and BDS-BestF are least efficient over the configurations, which is somewhat surprising given their high efficiency overall.

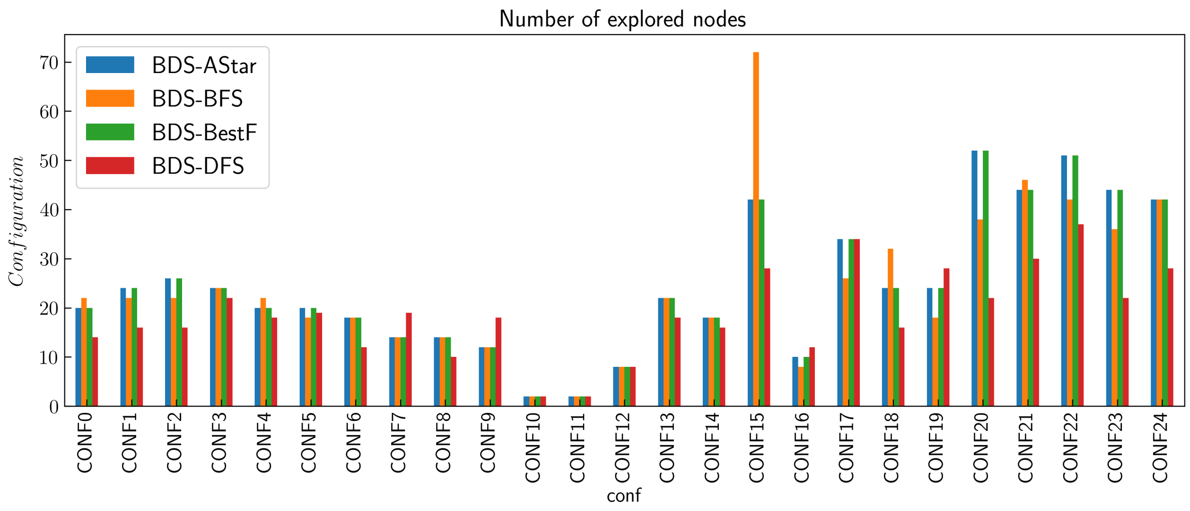

Figure 7: No. of explored nodes for 25 given configurations using BDS.

Table 2: Average and median rank in terms of path cost and node count across the 25 given configurations for different BDS agents.

	Path Cost (Rank)		**Node Count (Rank)**	
	mean	**median**	**mean**	**median**
BDS-BFS	1.0	1.0	1.84	2.0
BDS-DFS	1.74	2.0	1.32	1.0
BDS-AStar	1.48	1.0	2.04	2.0
BDS-BestF	1.48	1.0	2.04	2.0

4.3 Concluding remarks

The evaluation confirmed several theoretical characteristics of the algorithms. As expected, the optimal algorithms (BFS, AStar, BDS-BFS) did indeed consistently yield the lowest path costs and the intermediate agents BestF and AStar clearly outperformed the basic agents in terms of efficiency (node count).

For BDS, interestingly, the use of AStar or BestF for the backward search yielded less efficient results than using BFS or BDS. A more detailed investigation into the reasons for this could be interesting, but is out of scope for this practical.

5 Appendix

Table 3: Test coverage report of the JUnit tests for the application. Overall, 79.2% of classes (19/24) were tested, 72.5% of methods (108/149), and 69.6% of lines (364/523).

Class	**Class, %**	**Method, %**	**Line, %**
A1main	0% (0/1)	0% (0/7)	0% (0/52)
AStarSearch	100% (1/1)	100% (3/3)	100% (19/19)
BestFirstSearch	100% (1/1)	100% (2/2)	100% (8/8)
BidirectionalSearch	100% (1/1)	90.9% (10/11)	92.4% (61/66)
BreadthFirstSearch	100% (1/1)	100% (1/1)	100% (3/3)
Conf	100% (1/1)	100% (5/5)	100% (40/40)
Coord	100% (1/1)	100% (13/13)	100% (30/30)
DepthFirstSearch	100% (1/1)	100% (1/1)	100% (3/3)
Experiment	0% (0/1)	0% (0/1)	0% (0/5)
ExperimentReaderWriter	0% (0/1)	0% (0/10)	0% (0/65)
ExploredSet	100% (1/1)	62.5% (5/8)	72.7% (8/11)
Frontier	100% (1/1)	100% (3/3)	93.3% (14/15)
GeneralSearch	100% (1/1)	100% (8/8)	100% (33/33)
InformedSearch	100% (1/1)	100% (2/2)	100% (4/4)
Map	100% (1/1)	80% (4/5)	93.8% (15/16)
Node	100% (2/2)	93.8% (15/16)	97.1% (33/34)
PriorityQueueFrontier	100% (1/1)	66.7% (8/12)	84.6% (22/26)
QueueFrontier	100% (1/1)	88.9% (8/9)	93.3% (14/15)
Search	100% (1/1)	63.6% (7/11)	85.2% (23/27)
SearchFactory	0% (0/2)	0% (0/4)	0% (0/13)
StackFrontier	100% (1/1)	77.8% (7/9)	86.7% (13/15)
State	100% (1/1)	75% (6/8)	91.3% (21/23)

References

P. Norvig and S. Russel. *Artificial Intelligence: A Modern Approach.* Prentice Hall Upper Saddle River, NJ, USA, 2010.