File: ddc/AIQV/BDDFS.docx

Bidirectional Depth First Search Experiments

Dennis de Champeaux

Unaffiliated

Silicon Valley

2025

Abstract

Search techniques with uninformed, and informed, unidirectional, and bidirectional versions and with additional subversions are core assets of AI. Depth first search belongs to this collection. This paper provides companions: abstract, *bidirectional* depth first search versions. Numerous versions with their control regimes are described. Their performances are shown with applications to a range of examples, among which the knight tour on a chess board and finding paths in grids. Variants include design additions: keeping track of visited nodes, restoring a node before backtracking or not, forward &backward counts, etc. Parallel versions and two-stack versions are also presented. Examples are given with unfavorable and favorable comparisons against unidirectional depth first search. Opportunities for further explorations appear ample/ abundant.

1 Introduction

Search is a core AI asset for planning, reasoning, deduction, theorem proving, problem solving, inferencing, etc.; see [Korf] for an overview. There are search versions having an associated bidirectional version; see [BiDirectionalSearch]. A recent paper [Sturtevant] provides novel insights about versions where a heuristic function is available; these functions help prioritizing which node to elaborate. Bidirectional versions raised the issue whether the two searches will meet half way because Pohl's two paths were missing each other [Pohl]. Hence the paper in which a version is described for which there is the guarantee that the two paths meet in the middle [Holte]. Without a heuristic function best first search, depth first search and iterative deepening are options.

Depth first search lacks a bidirectional companion, which is the topic of this paper, hence of the 'green field' type. A key assumption here is the availability of a goal state in addition to the start state. We give an abstract characterization of such a version. A 'trick' is to apply unidirectional search selectively in forward and backward directions. Not increasing the branching ratio is an obvious concern, which is addressed by *not* operating in the product space of the two directions. A Spartan version of depth first search relies only on a stack and restoring changes, if any, before backtracking. We show the cyclic knight tour on a chess board as an example of this version as well as the bidirectional version. Another knight tour version starts with forward search and switches (relying on a depth parameter) to backward search. This version can be used in situations where, for example, the branching ratio varies on the path from start to goal.

Path finding on grids benefits from additional infrastructure for keeping track of visited nodes. Not restoring changes before backtracking is an option that can be helpful. We show also versions that use two stacks, either system stacks or private stacks and use one or two threads of control.

These versions are tested on different problem families yielding unfavorable and favorable outcomes in the comparisons against unidirectional depth first search. A comparison with bidirectional *best* first search is included. We discuss the results of these experiments.

GitHub has the Java code of 18 versions [GitHub]. An appendix has a table of some of their features. The reported timings below are in milliseconds obtained mostly on an I3/ Win7 laptop.

2 Abstract Algorithms

Several abstract algorithms are described. They all assume that we have available a start state and a goal state, both represented as nodes in a search space. There is also a function *findMoves(nodex)* which produces a set of adjacent nodes of the node *nodex.* Nodes have attributes so that we can record, for example, the parent node of a node with which we can obtain solution paths. The algorithms are flexible and allow unidirectional search in either direction. It is also possible to decide at each iteration whether to go forward or backward. The first abstract algorithm is invoked with:

```
move(startNode,goalNode)
```

This calls up:

```
move(fx, bx) {
   if ( fx = bx ) exit with a solution
   decide to go forward or backward and set
       successorNodes = either findMoves(fx) or findMoves(bx)
   if ( empty successorNodes ) return // back track
   for each node y in successorNodes do { if (go forward) move(y, bx) else move(fx, y) }
} // end of move
```

An actual implementation needs the *findNodes* function and infrastructure for the decision at run time whether to go forward or backward. Alternation is an option. Keeping track of the encountered number of nodes in a direction, or their path lengths, etc. can be used as well. Limiting the recursion depth can be an option with additional infrastructure.

If we keep track of nodes visited the algorithm can be extended by terminating when a node is encountered from the other side:

```
move(fx, bx) {
   if ( fx = bx ) exit with a solution
   decide to go forward or backward and set
       successorNodes = either findMoves(fx) or findMoves(bx)
   if ( empty successorNodes ) return // back track
   for each node y in successorNodes do {
      if (go forward) {
          if ( y was visited in the backward search ) exit with a solution
          else move(y, bx)
      } else {
          if ( y was visited in the forward search ) exit with a solution
          else move(fx, y)
      }
} // end of move
```

Our versions have infrastructure for parameters:

- Terminating with a single solution or finding all solutions
- Using a repository for keeping track of visited nodes or not
- Restoring optionally changes, if any, of a visited node before backtracking

The *findMoves* function operating in our grid examples is parametrized as well. Filtering out nodes in the 'wrong' direction can be turned on and off yielding searches that are '*eased*' or '*hampered*'. The *findMoves* function generates a list of child nodes; its ordering can be scrambled or not, which can make finding a solution path task harder with some combinations of the parameters.

Obviously we can present only a part of all combinations that are possible, which opens up opportunities for more research.

Below we describe also two-stack versions for sequential and parallel processing.

3 Correctness

Termination is not guaranteed when the search space is not finite. Both explorations can go on forever. When the search space is finite there can still be trouble due to loops. If there are no loops we have two

cases: there is a path or not between the start and goal state. If there is a path it will be found using an induction argument on the length of the path. If there is not a path the algorithm will terminate because there are no loops. These arguments assume that there are no resource conflicts. Loops can cause failures as shown.

The effectiveness of the algorithm (regarding finding a solution by meeting in the middle of a path) depends on the structure of the search space, the forward and backward branching ratios, which can be arbitrary complex, and hence prevents further analysis.

4 Comparison against best first search

Before comparing depth first search against bidirectional variants we compare bidirectional *depth* first search – with an example – against bidirectional *best* first search. Both bidirectional variants alternate forward and backward search. We employ path finding on grids made by an array of size 2000 x [10-100]. The start state is a corner of the array and the goal state the corner at the other side reached by the diagonal. The searches of both algorithms are *eased*, i.e. moving in one of the 4 directions is blocked, the list of generated child nodes is scrambled and termination occurs when a path is found. Table 1 has the timings obtained on an I3 machine. The timing results support further investigations of bidirectional *depth* first search.

Size X x 2000	Bidirectional *best* first search Grid2BF	Bidirectional *depth* first search Grid2
10	35	13
20	60	10
30	73	11
40	97	11
50	118	12
60	186	15
70	179	11
80	196	10
90	217	10
100	228	13
Table 1. Timings of finding a single path on rectangular grids of different sizes.		

The codes of these versions Grid2BF and Grid2 are available at [GitHub].

5 Knight tour experiments

The knight tour on a chessboard is a classic example that can be solved with depth first search. All tiles must be reached by knight jumps and no tile may be visited twice. A cyclic tour has the additional constraint that the start tile can be reached from the last visited tile by a knight jump; see Figure 1 for an example with a jump back from tile 64 to tile 1.

1	12	9	6	3	14	17	20
10	7	2	13	18	21	4	15
41	64	11	8	5	16	19	22
28	25	42	39	56	23	44	37
63	40	27	24	43	38	55	58
26	29	50	61	32	57	36	45
49	62	31	52	47	34	59	54
30	51	48	33	60	53	46	35

Figure 1. An 8x8 chess board with a cyclic knight tour

The code regarding the specifics of this example is a bit intricate because there is infrastructure to prevent tiles to become unreachable and about checking that a pre-solution satisfies the cyclic requirement. Otherwise it follows the pattern described above.

We obtained (with a C implementation) over 600M solutions on an 8x8 board after 2 months of processing on an I5/ Ubuntu laptop, after which we give up; it looks like that there are at least 128 times more solutions.

The number of solutions on a 6x6 board is 9862. The timings of a Java implementation in milliseconds on an I3/Win7 laptop were as follows:
Uni-direction: 7427
Bi-direction: 7485
On an I5/Win10 laptop:
Uni-direction: 5672
Bi-direction: 5656
These timings were obtained by taking the medians of several runs.

The bi-direction version alternates forward and backward search yielding similar timings; hence there is here no benefit for using the bidirectional version. The code of Knight3 is available at [GitHub].

An attempt to improve by using two threads of control for the forward and backwards paths was not successful. Synchronization infrastructure slowed down unidirectional search already so much that there was no hope for using parallelism – we believe, but we will be delighted when proven wrong.

Cloning the board for the different successor locations of the 2-tile (in Figure 1: 3, 5, 39, 25 and 41) facilitates finding all solutions faster with parallel processing.

6 Forward followed by backward search

An alternative design does forward search up to a specified depth and subsequently switches to backward search. A setting where this approach can payoff – in another domain – is where the search space is not uniform; for example the forward/ backward branching ratio at the start side is different from what it is at the goal side of the search space.

An abstract characterization of this variant (assuming no loops or infrastructure that avoid looping):

```
goForward(0, startNode, goalNode)
```

The *goForward* call expands, using a preset depth level parameter *depthBi*:

```
goForward(depth, startNode, goalNode) {
   if ( depthBi == depth ) {
      goBackward(startNode, goalNode)
      reset goalNode // for next backward search
      return }
   else { depth++
   // explore successor nodes
   for each successor node nx of startNode {
       goForward(depth, nx, goalNode)
       restore nx }
   return
} // end of goForward
```

The *goBackward* call expands (also simplified) into:

```
goBackward(startNode, goalNode) {
     if ( startNode == goalNode ) {
        process solution/s
        return }
     // explore successor nodes
     for each successor node nx of goalNode {
        goBackward(startNode, nx)
        restore nx }
     return
```

```
} // end of goBackward
```

The implementation of this version for the knight tour has also improvements; for example avoiding a successor node that would make a not yet visited location inaccessible. We exploit this enhancements by turning it off selectively to emulate challenges in the underlying search space. The following results were obtained:

Processing time in milliseconds on an I3 machine for all 9862 solutions on a 6x6 array:

55567 with only handicapped backward search

Assisted by initial enhanced forward search:

42118 with depth 10
16450 with depth 20
15787 with depth 25

This shows that a version which has the combination of forward search to a certain depth followed by backward search can outperform unidirectional search. The code of Knight5 is available at [GitHub].

7 Single stack grid

Sequential search versions on 2D grids employing a single system stack is the topic of this section.

7.1 Grid2 experiment

This experiment aims to find one or more paths in a 2-dimensional grid. We use arrays sized X x Y with the start state in one corner and the goal state at the opposite side at the end of the diagonal. The termination condition – for a single solution – is encountering a tile visited earlier on a path from the other side. This was implemented through a direction attribute in the class GN2 that has details about the tile of a node. The direction attribute captures whether the tile has not yet been visited or alternatively whether a forward or backward search was involved.

Backtracking has two options: restoring a vacated tile to not-visited first, or keep the visited status. The *findNodes* function produces a sequence of successor states; that sequence can be scrambled or not. More significant is whether backward moves are allowed (*hampered* mode), or not (*eased* mode).

We start with a small example. The ordering of the sequence generated by the *findNodes* function is here scrambled. Both searches are *hampered* since moves in the wrong direction are also generated, to the left in forward search and to the right in backward search. *Hampered* search can get into trouble as elaborated further down.

A solution on a 6 x 25 grid with bidirectional search with restored, scrambled and *eased*:

```
 1  2  5  6  7                              23 22
    3  4     8  9                           24 21                 7  6  5
               10 11                     26 25 20 19           9  8     4
                  12       19 20          27       18 17    11 10        3
                  13       18 21 24 25 26 28          16 13 12           2
                  14 15 16 17 22 23    27 28          15 14              1
```

Similar but more difficult with *hampered* (moves in the wrong direction are allowed):

```
 1  2  7  8 13 14                                    18 17    13 12 11 10
    3  6  9 12 15             39 38       23 22 21 20 19 16 15 14  7  8  9
31  4  5 10 11 16 17    44 43 40 37       24 25                    6  5
30    24 23 20 19 18 46 45 42 41 36       27 26                       4  3
29 28 25 22 21                   35 34    28 29                          2
   27 26                            33 32 31 30                          1
```

The number of moves was high at 4193. This was caused because both search directions got into trouble. Backtracking at one side wiped out progress at the other side. Both sides have dead-zones; at the left side by the 26-27 move, at the right side by a move (not shown) from 30 to the right. Backtracking out of the larger dead-zone at the right side yielded the escape resulting in the path with the 18-46 pair. It turns out that the escape at the right side was 'lucky'. The next example using a 6 x 35 grid has two dead-zones from which an escape is not possible, causing an infinite loop.

```
1  2        53 52 49 48     44 43 36 35 34 33 30 29              43 44 45 46     14 13 12  9  8
   3     55 54 51 50 47 46 45 42 37        32 31 28           41 42 53 52 47     15     11 10  7
5  4     56       63 64        41 38           26 27           40 39 54 51 48     16 17 18  5  6
6        57 58 59 62            40 39     23 24 25                38     50 49 30 29 20 19  4  3
7              60 61               20 21 22                      37 36 35 34 31 28 21 22 23  2
8  9 10 11 12 13 14 15 16 17 18 19                                    33 32 27 26 25 24  1
```

Section 3 mentioned already that loops can create trouble. We added a repository (a `Hashtable`) to keep track of visited locations. This facilitated avoiding a loop and increased the ability to backtrack out of a dead-zone. Rerunning the 6 x 35 grid example yielded a solution: We show a fragment of the path (eliminated a segment at the left side) with a solution at the 27-51 pair:

```
1  2                 36 35 34 33 30 29                                    14 13 12  9  8
   3                 37       32 31 28 50 49 48    44 43 42 41           15     11 10  7
5  4              41 38          26 27 51    47 46 45       40 39         16 17 18  5  6
6                 40 39    23 24 25                        38          30 29 20 19  4  3
7                    20 21 22                              37 36 35 34 31 28 21 22 23  2
8  9           17 18 19                                          33 32 27 26 25 24  1
```

There is still a dead-zone at the left side but an escape happened at the right side. While the addition of infrastructure yielded an improvement, we found another example with the *hampered* mode where still no solution was obtained. While tracking visited locations and avoiding them subsequently was beneficial for this 6 x 35 example it failed for other examples, hence we disabled this functionality.

We proceed by showing timing results on larger arrays in Table 2 using the *hampered* mode.

Size	Bidirectional alternate	Unidirectional
6 x 45	2	10340
6 x 50	3	NA
6 x 55	3	22
6 x 60	3	20
6 x 65	3	698
6 x 70	10	372
6 x 75	5	10
6 x 80	4	NA
6 x 85	7	196
6 x 90	5	180
6 x 95	5	NA
6 x 100	5	2
Table 2. Bi- and unidirectional timings of Grid2		

The situation changes dramatically when the *findNodes* function filters out moves in the wrong direction. The timings on the 6 x 45 array for bi- and unidirectional search become both *one*.

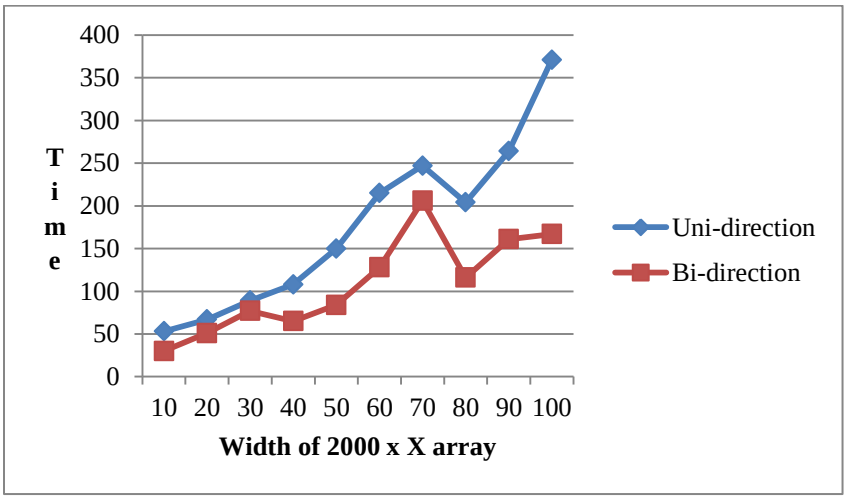
The bidirectional version uses here straight alternation. An alternative is exploiting that we track the length of the path from the start for the forward search and similarly from the goal for the backward search. This allows deciding to go forward or backward based on which side has the current shortest path length. Table 3 has the unidirectional column of Table 2 replaced by the findings of the other way to go forward or backward.

Size	Bidirectional alternate	Bidirectional shortest path
6 x 45	2	2
6 x 50	3	658
6 x 55	3	4924
6 x 60	3	NA
6 x 65	3	4
6 x 70	10	NA
6 x 75	5	NA
6 x 80	4	NA
6 x 85	7	NA
6 x 90	5	NA
6 x 95	5	NA
6 x 100	5	NA
Table 3. Two versions for different procedures that decide going forward or backward in Grid2		

It is certainly quite remarkable why the alternative procedure (aimed at improving meeting in the middle of the solution path) has such a poor outcome. It is an opportunity for a deeper investigation. For now, we conjecture that the different way to decide going forward or backward provides more opportunities to enter dead-zones.

Subsequently the code of Grid2 was modified so that the array grid tile was *not* restored before a backtrack. Hence, this created more opportunities to recognize a tile from the other side. Encountering a tile again from the own direction caused this exploration to be ignored. The search mode was *eased*.The ordering of the output of the *findMoves* function was scrambled as before to make the task more challenging.

We used for this experiment larger grids. The next graph has the timings for unidirectional and bidirectional search for finding paths in arrays in arrays 2000 x [10-100]. Search did not stop here after finding a solution. Unidirectional search finds each time two solutions, and the bidirectional search finds a various number of solutions, respectively: 68, 176, 641, 1640, 17739, 9125, 14639, 2380, 2008, and 3389. Hence the time differences are actually more in favor for bidirectional than shown, although, as mentioned above, subsequent explorations benefit from the earlier visited tiles.



The bidirectional search distinctly outperforms unidirectional search also in this example.

Still, we found also with the parameters *eased* and restore on a 2000 x 2000 grid the timings:
Bi-direction: 38

Uni-direction: 14

The code of Grid2 is available at [GitHub].

7.2 Grab experiments

This experiment uses first a 6 x 12 array grid and marks the grid elements as before. This time however (in the bi-directional version) the forward exploration is not permitted to enter the territory of the backward exploration, and similar for the backward exploration. Forward and backward search directions alternate. Backtracking in this version entails restoring the vacated tile to not-visited. The ordering of the sequence generated by the *findNodes* function is scrambled. Both searches are *hampered* since moves in the wrong direction are also generated. For the bidirectional search, single solution we obtain;

```
 f  f     f  f  f  f
    f  f  f  f  f  f  f  b  b  b  b
             f  f  f  F  b     b  b
             f  f  f  B  b  b  b
                      b  b  b  b  b
                            b  b  b

 1  2     6  7 16 17
    3  4  5  8 15 18 19 14 13 12 11
             9 14 13 20 15     9 10
            10 11 12 19 16  7  8
                     18 17  6  3  2
                            5  4  1
```

The trace shows a forward move (13 to 14) and a backward move (7 to 8) going in the wrong direction.

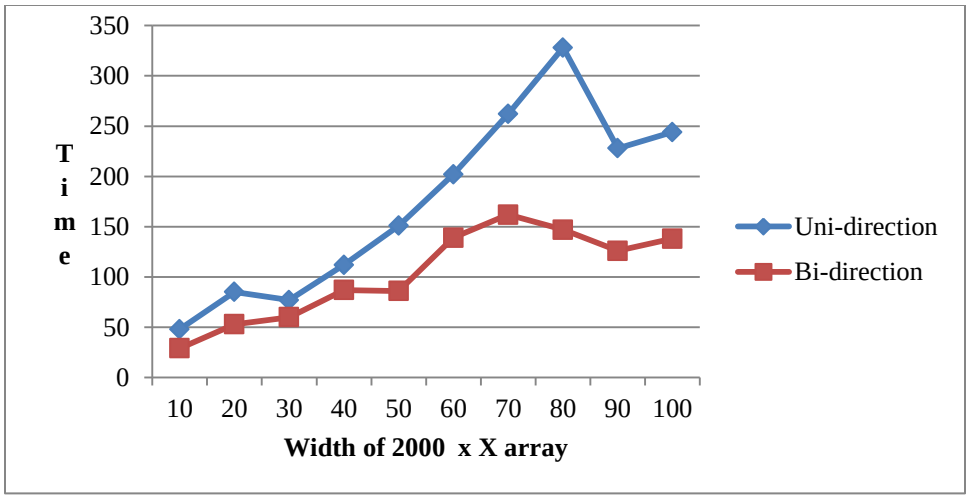
We repeat the test we did for Grid2, see Table 4.

Size	Grid2	Grab
6 x 45	2	5
6 x 50	3	4
6 x 55	3	12
6 x 60	3	3
6 x 65	3	2
6 x 70	10	5
6 x 75	5	3
6 x 80	4	2226
6 x 85	7	5
6 x 90	5	5
6 x 95	5	9
6 x 100	5	5
Table 4 Bidirectional timings of Grid2 & Grab		

The timing for the 6 x 80 search space must be caused by ‘bad luck’ due to the search directions getting stuck in dead-zones. The code of Grab is available at [GitHub].

Similar to the Grid2 experiment, we applied the Grab version also to larger arrays. The next graph has the timings for unidirectional and bi-direction search for finding paths in arrays 2000 x [10-100]. However we changed the settings to: *eased*, no restore before backtracking and continue after finding a solution.

Unidirectional search finds each time two solutions, while the bi-directional search finds a varying number of solutions: for array size 10: 20 and subsequently: 6, 5, 7, 17, 10, 13, 12, 13, 17.



The bidirectional search outperforms in this example also. The additional constraint in the Grab version not to enter the territory of other search direction appears not to have made a tangible difference with the Grid2 version, as shown in Table 4.

7.3 Grid3 experiments

This experiment used also first a 6 x 12 array grid but has a barrier half way the length of the array with a pinhole gap in the middle of the barrier. This version experiments with a hash-table that maintains the status of occupied grid tiles.

```
xxxxxxxxxxxxxxxxx
xS       x       x
x        x       x
x        x       x
x                x
x        x       x
x        x      Gx
xxxxxxxxxxxxxxxxx
```

Scrambling of the move operators was not done as shown by the following non-snake paths. Moves in the wrong direction were blocked, search terminated after a solution was found, backtracking implied restoration of the original situation. The bidirectional version has a module that can block a search path to cross the mid-boundary. Unidirectional:

```
f  f  f  f  f  f  f
                  f
                  f
                  f  f  f  f  f  f
                                 f
                                 b
```

Bidirectional:

```
f  f  f  f  f  f  f
                  f
                  f
               b  b
                  b
                  b  b  b  b  b  b
```

The solution path in the bidirectional version is found in the pin hole in the middle.

Making the search more difficult by scrambling and making both sides *hampered* yielded for the unidirectional search:

```
f     f  f
f     f  f  f  f  f
f     f  f  f  f  f
f  f  f  f  f  f  f  f
f  f                  f  f  f
f  f                        f  f  b

1     15 16
2     14 17 18 23 24
3     13 12 19 22 25
4  9 10 11 20 21 26 27
5  8                 28 29 30
6  7                       31 32  1
```

The bidirectional search generated:

```
f                        b  b  b  b  b
f  f  f  f     f  F  B                 b
         f  f  f  f           f        b
                  f  f  f  f           b
                           b  b  b
                           b  b  b

1                       14 13 12 11 10
2  3  4  5     9 10 15                 9
         6  7  8 11        16          8
               12 13 14 15              7
                           4  5  6
                           3  2  1
```

The code of Grid3 is available in [GitHub].

We proceed by showing the timing results on larger arrays in Table 5. We used restoration before backtracking, scrambling and both directions were *hampered* (allowing moves in the wrong direction).

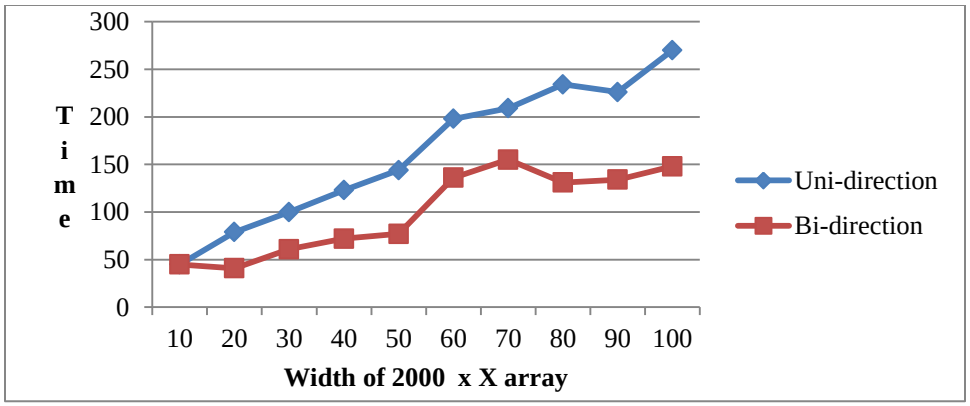
Size	Bidirectional	Unidirectional
6 x 45	7	26
6 x 50	9	75224
6 x 55	8	5330887
6 x 60	10	1016
6 x 65	4	2168497 *
6 x 70	9	6
6 x 75	8	21759
6 x 80	7	2565559
6 x 85	5	NA
6 x 90	12	NA
6 x 95	6	874
6 x 100	14739	174743

Table 5. Bi- and unidirectional timings of Grid3. The entry with * is obtained not on an I3/win7 machine but on a faster I5/win10 machine.

The timings in Table 2, Table 3 and Table 5 show the consequence of the halfway barrier: unidirectional search 'struggles' even more to find solutions. The 6 x 100 entry in Table 5 of the bidirectional column is likely caused by entering a dead-zone; and even more so for the entries in the unidirectional column.

Similar to the Grid2 and Grab experiment, we used larger arrays than in the previous experiment. Again the search is *eased*, the moves sequence was scrambled and back tracking did not restore the grid.

The next graph has the timings for unidirectional and bi-direction search for finding paths in arrays 2000 x [5-50]. Unidirectional search finds each time two solutions, while the bidirectional search finds the following number of solutions: 46, 429,95, 25, 980, 199, 3519, 32, 1736, 29593.



Yet again we find a problem family where the bi-directional search version wins.

7.4 GridT experiments

This experiment uses similarly a rectangular array, but without borders, because the edges refer back to the other side; it can be seen as a grid laying on a torus. We use first: a single solution, scrambled, both directions *eased*, no restore; while this grid is a torus, the short sides are not crossed because the directions are *eased*, otherwise the start and goal nodes would be very close. An example of a unidirectional solution starting in the top left and ending in the bottom right is:

```
f  f                    f  f     F
f  f                       f     f
f  f  f     f  f           f     f
f     f  f  f  f           f  f  f
f     f  f     f  f  f        f  f
f  f                 f  f        B

1  8                    25 26    36
2  9                       27    35
3 10 11    17 18           28    34
4    12 15 16 19           29 30 33
5    13 14    20 21 22        31 32
6  7                 23 24        1
```

The numbers show that the path uses the lack of boundaries with the moves: 7-8, 24-25 and 36-1.

A bidirectional solution is:

```
f        f        F  b  b  b
f        f  f  f  f  b  b
f
f
f     b  b  b  b  b
f  f  f  f        B  b     b  b  b

1       10       15  8  5  4
2       11 12 13 14  7  6
3
4
5    15 14 13 12 11
6  7  8  9       10  9     3  2  1
```

This is a solution indeed because the two tiles F and B are adjacent on the torus.

The next experiment used a 2000 x X array torus grid for X = (5,10,15,20,40,60,80,100,600). We used multiple solutions, scrambled, both directions *eased* and no restore before backtracking. The performance is shown in Table 6 with the timings and with the number of solutions in brackets.

X	Bidirectional	Unidirectional
5	16 (22)	36(3)
10	34 (59)	48 (3)
15	47 (139)	58 (3)
20	56 (191)	69 (3)
40	83 (4323)	117 (3)
60	148 (29421)	185 (3)
80	126 (44514)	207 (3)
100	151 (53399)	295 (3)
600	794 (353721)	1226 (3)
Table 6. Bi- and unidirectional timings of GridT. Number of solutions in brackets.		

We see here also that the bidirectional version is doing a better job.

Another test uses the combination: single solution, both directions *hampered*, scrambled, and restore.

The unidirectional version produced:

```
1                                   2




                                    1
```

The bidirectional version generated the solution:

```
1                                       2
                                    4   3
                                    5
                                    4
                                    3   2
                                        1
```

Both of them exploited the torus topology for the search space successfully. Both of the paths being short is quite unexpected.

Using the array X x 2000 for the combination of multiple solutions, both directions *hampered*, scrambled, and restore yielded to following results with the timings and number of solutions in brackets; see Table 7.

X	Bidirectional	Unidirectional
5	1 (57)	37 (4)
10	7 (629)	53 (4)
15	10 (107)	63 (4)
20	24 (3331)	83 (4)
40	93 (53)	115 (4) *7m
Table 7. Bi- and unidirectional timings of GridT. Number of solutions in brackets.		

Hampering (allowing search moves in the wrong direction) required increasing the Java stack to 7M for the unidirectional 40 x 2000 example. The bidirectional version outperforms again the unidirectional version.

The code of GridT is available at [GitHub].

8 Two private or system stacks for sequential and parallel versions

This section has versions using two system or private stacks with either sequential or parallel processing. There are comparisons between these processing modes and comparisons against the earlier, single stack versions. Performances with array layouts having additional barriers are presented also.

8.1 Grid2p4 experiment

The *7.1* Grid2 section has an application of bidirectional *hampered* search on a 6 x 25 grid. Early on the forward search maneuvers itself in a dead-zone and it never gets out of it. The backward search enters an even larger dead-zone, but ultimately it breaks out through backtracking and a solution was found with move-count 4193. Grid2p4 has two threads for the forward and backward search that use system stacks.

Each invocation of this version on the 6 x 25 grid gives a different solution due to the random scheduling of the Java threads. The timings are usually 1, while the move-count varies like: 118, 267, 83, 101, etc. Here an example of a solution where the forward path is longer than the backward path:

```
 1
 2  3
    4 13 14
 6  5 12 15 16
 7  8 11    17 20 21 22       27 28 29          35 36     7  6
    9 10    18 19    23 24 25 26    30 31 32 33 34 37  9  8  5  4  3  2  1
```

Here an example where the forward path is shorter:

```
 1                27 28 29 30 48                          14 13 12 11 10
 2  3          25 26 35 34 31 47 46 39 38 37              15 16        9
    4 13 14    24 23 36 33 32    45 40 41 36 33 32 31 20 19 18 17     7  8
 6  5 12 15 16    22              44 43 42 35 34    30 21 22 23 24  5  6
 7  8 11    17 20 21                              29 28 27 26 25  4
    9 10    18 19                                                 3  2  1
```

Lacking a heuristic caused going in the wrong direction. Changing the parameter from *hampered* to *eased* yields a solution with zero timing and a 49 move-count:

```
 1
 2  3
    4  5  6                                          31 32        6  5
          7                                          30 33 34     7  4
          8                16 17 18       23 24    28 29 13 12     8  3
          9 10 11 12 13 14 15    19 20 21 22 25 26 27 15 14 11 10  9  2  1
```

We proceed with more examples of using two stacks, system or private, and using multi-threading.

The version Grid2p4 is available in [GitHub].

8.2 Sequential processing two stacks GridB2 versus one stack Grid2

We have shown above numerous examples above where our bidirectional version (with some parameters) outperforms the unidirectional version. Still we noticed that there were occurrences of repeated backtracking at one direction that could undo good progress at the other side – as shown above. This inspired trying a different way: using depth first search twice using two private stacks; one for forward search and one for backward search. In pseudo code (simplified):

- Create forward and backward nodes
- Enter a loop that checks weather the search has terminated otherwise it alternates forward & backward processing – assume forward:
    - If forward stack empty set `done-forward` true and continue
    - Obtain a forward stack element and enter a loop to explore successor nodes
    - Backtrack with a pop when the element has been fully investigated; restoration or not first is an option
        - When a successor node has been visited in the forward search, skip
        - When a successor node has been visited in the backward search register a solution and pop the stack
        - Otherwise push the successor node on the stack and continue

The version GridB2 operates on a grid, like Grid2, but uses two private stacks.

We show again the solution of Grid2 on the 6 x 25 grid with restored, scrambled and *hampered*:

```
 1  2  7  8 13 14                                    18 17    13 12 11 10
    3  6  9 12 15             39 38       23 22 21 20 19 16 15 14  7  8  9
31  4  5 10 11 16 17    44 43 40 37       24 25                    6  5
30    24 23 20 19 18 46 45 42 41 36       27 26                       4  3
29 28 25 22 21                   35 34    28 29                          2
   27 26                            33 32 31 30                          1G
```

The timing is 16 and the number of moves is 4193.

A GridB2 solution with these parameters is:

```
 1                      21 22 23 24 31                          12 11 10
 2                      20       25 30 29                       13     9
 3             14 15 16 19       26 27 28 30                    14  7  8
 4             13    17 18                29                 16 15  6  5
 5       10 11 12                         28 27 26 25 24 23    17 18  3  4
 6  7  8  9                                        22 21 20 19  2  1
```

The timing is here 5 and the number of moves is 61.

As discussed above the Grid2 solution has the large number of movements because a dead-zone was entered at the left side from which no escape occurred. A dead-zone at the right side was escaped resulting in the solution with the 18-46 pair. GridB2 avoided these obstacles. It can run also into dead-zones but backtracking at one side does not impact the situation at the other side.

The next experiment used the sizes 2000 x X for X = (5,10,15,20,40,60,80,100,600). We used multiple solutions, scrambled, both directions *eased* and no restore when backtracking. The performance is shown in Table 8 with the timings and with the number of solutions in brackets.

X	GridB2	Grid2
5	58 (30)	32 (26)
10	59 (200)	43 (68)
15	67(88)	44 (251)
20	94 (174)	60 (176)
40	150 (2182)	102 (1640)
60	234 (2929)	169 (9125) *2m
80	295 (4666)	169 (2380) *2m
100	364 (4712)	214 (3389) *2m
600	1321(3555)	3363 (2333) *2m
Table 8. Timings for GridB2 and Grid2 with the *eased* parameter. Number of solutions in brackets. Entries with *2m required increasing the system stack size.		

Since both directions were *eased* (no moves in the wrong direction) there was no risk to enter a dead-zone. It turns out that in this case Grid2 using the system stack yields a better performance than using the two private stacks by GridB2, although the stack had to be increased.

We repeated this test battery with using instead the *hampered* parameter for both versions but terminated after finding one solution; see the results in Table 9.

X	GridB2	Grid2

5	23	NS 1
10	29	NS 1
15	50	NS 11
20	51	NS 10
40	89	82 *2m
60	189	134 *3m
80	97	164 *3m
100	208	154 *4m
600	639	3542 *20m

Table 9. Timings for GridB2 and Grid2 to obtain one solution with the *hampered* parameter. NS = no solution. Entries with *xm required increasing the system stack size.

The results are different. Grid2 failed to find a solution in 4 of the 9 examples. Grid2 is faster than GridB2 in 3 of the 5 tasks where they both found a solution. The system stack had to be increased to terminate tasks.

Subsequently we repeated with the termination after all solutions were found; see Table 10.

X	GridB2	Grid2
5	46 (12)	NS 2
10	43 (22)	NS 6
15	93 (125)	NS 8
20	108 (136)	NS 11
40	160 (407)	146 (28) *2m
60	312 (166)	212 (9) *3m
80	334 (1500)	245 (21) *3m
100	458 (1377)	372 (23) *4m
600	3681 (11347)	3815 (37) *20m

Table 10. Timings for GridB2 and Grid2 with the *hampered* parameter. NS = no solution. Number of solutions in brackets. Entries with *xm required increasing the system stack size.

As before reported in Table 9, Grid2 failed to find a solution in 4 of the 9 tasks. It was faster in the other tasks but required substantially more stack space and found less solutions. How that was achieved needs a deeper analysis.

Based on these results we have the rule of thumb that:

- a search domain without loops can be handled by a version using the system stack
- a search domain with loops should be handled by a version using two private stacks for the forward and backward searches

The code of GridB2 is available at [GitHub].

8.3 Sequential processing two stacks GridBt versus one stack GridT

The comparison of GridBt versus GridT resembles the comparison above in *8.2*. GridBt uses two private stacks while GridT uses (as shown above) the system stack. Different is that a backtrack required first the restoration of a grid element. Both searches are also *hampered*; hence the data in Table 9 corresponds with the data in Table 11 using also X x 2000 grids.

X	GridBt	GridT
5	0 (4)	2 (57)
10	92 (64)	7 (629)
15	80 (124)	14 (107)
20	86 (575)	26 (3331)
40	181 (5)	103 (33) *2m
60	375 (5)	198 (33) *3m
80	550 (2355)	201 (20) *3m
100	562 (1511)	263 (19) *3m
600	3568 (8)	3167 (54) *20m
Table 11. Timings for GridBt and GridT with the *hampered* parameter. Number of solutions in brackets. Entries with *xm required increasing the system stack size.		

There is a qualitative difference in the two tables. Grid2 failed in 5 of the 9 entries, while GridT did not.. Grid2 ran into trouble with the *hampered* parameter with the width 5, 10, 15 & 20 because of dead-zones at both sides. GridT operates on a torus which reduces the chance to enter a dead-zone. GridT could still get in trouble. Consider a search path creating a confined region that is entered; two of those can prevent obtaining a solution. GridT outperformed GridBt but at the expense of requiring large stacks.

The code of GridBt is available at [GitHub].

8.4 Parallel processing with two system-stacks Grid2p versus sequential one stack Grid2

Parallel processing is an 'obvious' option given that our versions engage in forward and backward searches using two system stacks. The implementation is fairly straightforward in Java given that it supports multiple threads. Two threads use the same code while they have – like the sequential versions – a parameter for the target directions. The parallel version Grid2p operates on the same grid as Grid2. Table 12 has the timings for obtaining *one* solution using *hampered* search and without restoration before backtracking (hence keeping the forward/backward status of visited location) on X x 2000 grids. The Grid2p column has two values as explained below.

X	Grid2p	Grid2
5	27 – 14	NS 1
10	41 – 21	NS 1
15	60 – 38	NS 11
20	71 – 44	NS 10
40	98 – 70	82 *2m
60	143 – 108	134 *3m
80	155 – 114	164 *3m
100	268 – 136	154 *4m
600	1001 – 257	3542 *20m
Table 12. Timings for Grid2p and Grid2 with the *hampered* parameter.		

The Grid2 column has the same features as in Table 9: no solution for 4 of the 9 tests. The Grid2p column has timings with (the left side) synchronization active and (the right side) synchronization directives commented out. Synchronization prevents the two threads to access a grid location at the same time (for which the chance is small); the price – in Java – is unfortunate. The parallel version does decrease the risks of getting stuck in dead-zones.

The code of Grid2p is available at [GitHub].

8.5 Parallel processing with two system-stacks Gridp versus sequential one stack Grid3

Gridp is a parallel version with two system stacks. This test resembles the test in *8.3.* The grid is different: there is a barrier in the middle with a pinhole gap. The grid range has larger widths (100 – 2000) and a longer length (2000). The settings are: *eased*, no restore and terminates with one solution. Table 12 has the timings for Gridp using synchronization protection and also with the removal of the synchronization directive.

X	Gridp	Grid3
100	10	10
200	13 - 11	13
300	15	9
400	21 - 9	31
500	17 -13	10
600	12 - 11	10
700	15 - 13	110
800	18 -15	158
900	16 -12	9
1000	24 -16	222
1200	17 -15	10
1400	330 -17	751
1600	26 - 24	10
1800	30 - 13	439 (0) NA
2000	88 - 20	6252 (0) NA
Table 13. Timings for Gridp and Grid3 with the *eased* parameter.		

The performance of the parallel version of Gridp is more consistent; the Grid3 sequential version failed on X 1800 and 2000. Still there were entries were Grid3 was faster than Gridp.

The code of Gridp is available at [GitHub].

8.6 Two stack experiments with sequential GridB & parallel Gridp4 and one stack Grid4

Sequential GridB uses two private stacks, parallel Gridp4 uses two system stacks. Grid4 with one system stack resembles Grid3. However, all three search in a grid that has not one barrier but two, with pinhole gaps at opposite sides:

```
xxxxxxxxxxxxxxxxxxxxxxxxx
xS      x               x
x       x       x       x
x       x       x       x
x       x       x       x
x       x       x       x
x               x      Gx
xxxxxxxxxxxxxxxxxxxxxxxxx
```

Grid4 solves a size 5 x 12 array grid with the settings: bidirectional, restoration before backtracking enabled, scrambled and *eased*. This yielded the solution:

```
f  f                   b  b
   f                   b  b
   f     f  f          b  b
   f  f  f  f  f       b  b  b  b
      f  f  f  F  B  b  b        b
```

Without restoration before backtracking, *no* solution was obtained, producing:

```
f  f  f
f  f  f                        b
   f  f                  b  b  b
   f  f                  b  b  b
   f  f  f               b  b  b
```

Backtracking at one side caused progress made at the other side to get lost. Both sides entered dead-zones.

Switching to GridB, which is using two private stacks with two simulated threads, we did get a solution with:

```
f  f  f                  b  b
      f              b  b  b
      f              b     b
      f           f  b     b
      f  f  f  f  f  b     b
      f  f                 b  b  b
```

By instrumenting Gridp4, which is using parallel threads and two system stacks, a solution was obtained also:

```
f  f  f           b  b  b  b
      f        b  b        b
      f  f  f  f  f        b
      f  f        f        b
      f  f        f     f  b
      f  f        f  f  f  b  b  b
```

Hence we observe that versions with two threads (simulated or not) are better equipped for handling difficult search domains because backtracking at one side does not demolish progress at the other side.

Testing on larger grids 20 x (1600, 1800, 2000) with restoration when backtracking enabled, scrambled and *eased* produced the timings in Table 14.

20 x X	Grid4	GridB	Gridp4
1600	10	9	8
1800	NA	8	8
2000	NA	11	6
Table 14 Grid4 & GridB use 1 thread, Gridp4 uses 2 threads.			

This result inspired more testing with also restoration, backtracking and *eased* search, see Table 15.

X x 2000	Grid4	GridB	Grid4p
5	9	15	7
10	13	13	8
15	NA	11	7
20	NA	10	7
40	17	9	8
60	NA	8	8
80	9	9	9
100	NA	21	8
600	18 *2m	26	10
Table 15 Grid4 & GridB use 1 thread, Gridp4 uses 2 threads			

The two stack versions GridB sequential and Grid4p parallel are doing a better job than the one stack sequential version Grid4; Gridp4 is here the winner.

The next test with these versions has *hampered* search with restauration before backtracking, see Table 16.

X x 2000	Grid4	GridB	Grid4p
5	NA	31	13
10	NA	23	11
15	NA	38	27
20	NA	47	20 *2m
40	NA	95	44 *3m
60	NA	163	99 *3m
80	NA	174	53 *4m
100	NA	136	41*2m
600	NA	844	812 *12m
Table 16 Grid4 & GridB use 1 thread, Gridp4 uses 2 threads			

The timings in Table 16 show that the single stack, single thread Grid4 version struggles even more to find a solution when the search is *hampered* (in contrast what we obtained in Table 15). The parallel version Grid4p version outperforms decisively the sequential two stack version GridB, although requiring more stack space.

The versions Grid4, GridB and Gridp4 are available in [GitHub].

8.7 Gridp5, Gridpp & GridB5 experiments

This experiment challenges versions with larger grids, 20 x 20000, and with 10 barriers. Gridp5 employs two threads using Java system stacks. Gridpp employs two threads with two ‘private’ stacks. GridB5 has similarly two ‘private’ stacks but uses only one thread of control.

Version	Restore before backtrack	No restore before backtrack
Gridp5	70	79
GridB5	64	51
Gridpp	80	81
Table 17 *Eased* search on 20 x 20000 grid with parallel Gridp5 using system stacks; sequential GridB5 with two private stacks; and parallel Gridpp with two private stacks.		

Table 17 shows that for the *eased* search the sequential version GridB5 is the fastest and Gridpp with its two private stacks the slowest.

Version	Restore before backtrack	No restore before backtrack
Gridp5	359 (1279)	367
GridB5	368	342
Gridpp	336	308
Table 18 *Hampered* search on 20 x 20000 grid with parallel Gridp5 using system stacks; sequential GridB5 with two private stacks; and parallel Gridpp with two private stacks.		

While Gridpp is the slowest for the *eased* search it is the fastest for the *hampered* search.

The two threads of the Gridp5 version recur deeper and deeper with the *hampered* parameter due to the 20000 length of the grid. This required a Java system stack of 20M. The first numbers in the *hampered* row of Table 18 measures when a solution is found; the number in brackets measures when the forward and backward threads have unwound their stacks and the thread that invoked the task regained control.

The versions Gridp5, Gridpp & GridB5 are available in [GitHub].

9 Discussions

Since this is a 'green field' exploration, we can't provide better results than earlier work except by comparisons against 'plain' depth first search. Instead we report how we 'meandered' and obtained progress while learning from setbacks.

An obvious prerequisite is the availability not only of a start state but also of a goal state. A solution (or more of them) is obtained when the frontiers of the two search efforts states meet up. There are variants how a single thread of control selects between forward and backward exploration; alternation is an option, shortest path length is an option, etc. We experimented with having a single thread of control managing one or two stacks and with two threads of control each managing a system stack or a private stack.

We used two domains: the knight tour on 8x8 and 6x6 boards and path finding in rectangular grids and on a torus. In both cases we added domain specific infrastructure to block obvious 'bad' candidate moves. We still encountered in the grid domain single stack examples where a search path entered a dead-zone which can take a lot of backtracking to escape. Two stack designs reduce these slowdowns. Two threads with system stacks or private stacks give timings improvements.

Working with 2 dimensional grids yielded the difference *eased* (prevent going in the wrong direction) versus *hampered* (allow going in the wrong direction) searches that capture restricting successor nodes or not.

The knight tour domain inspired using forward search up to a specified depth and switch subsequently to backward search, which can be relevant be when the search space differs at the goal side from the start side.

Meeting in the middle of the search space has not been tracked – likely another opportunity to explore.

10 Conclusion and next challenges

We are not aware of earlier investigations of bidirectional depth first search. Several, novel, abstract algorithms are described. There are four implemented designs (with18 implementations):

1. A depth first forward search combined with a depth first backward search in a sequential setting
2. A depth first forward search switching (using a predefined depth parameter) to a depth first search backward version
3. As in 1 but using two parallel threads as supported by Java

4. As above but using two stacks supporting forward and backward searches in, again, a sequential setting

Path finding in grids (where start and goal locations are known) with varying sizes and with varying obstacles has shown promising results: the bidirectional versions mostly outperform unidirectional versions.

The designs are parametrized regarding how forward and backward search alternate, whether or not visited nodes are remembered, whether or not restoring visited nodes before backtracking, whether the move generator should allow or not bias candidate moves, whether or not the move generator can scramble or not the order of candidate moves, etc.

Using two stacks (system or private) with one or two threads of control appear to be better design choices. Searches in the ease-mode that rely on an (implicit) heuristic can be done with good results using sequential versions. Otherwise when the search is in the hampered-more – lacking an implicit heuristic the parallel versions obtain better results.

We started out with a simple idea, but each experiment generated more ideas. Hence we can recommend this topic because there are many opportunities for further explorations.

Finding additional problem families where to try out this type of search is certainly necessary. Hence, we encourage other minds to pursue this challenge.

A larger challenge remains how to create 'automatically', i.e. without 'manual' coding, problem families and their instantiations so that a search version can be applied to them. The PDDL language [PDDL] facilitates the specification of problem families and their instances but it is not enough. A small step was perhaps our 'Object-Oriented Development Process and Metrics' by describing some semantics of the UML formalism for the analysis phase; see [DdC]. However, the gap remains. Creating routinely the required data structures, the classes and objects with their attributes and functional components, is elusive and requires a generic facility – a kind of LOT (Language of Thought [Fodor]). This problem has surfaced in the recent chat-systems and needs attention as elaborated in [DdC2, DdC3]. Simple problems described in natural language solvable by an AI algorithm – the crown jewels of AI – cannot be solved in chat-systems because it requires the ad hoc creation of data-structures for nodes and the collection of successor nodes. Hence bidirectional depth first search is an addition to the challenge of recruiting AI algorithms without manual coding in chat-systems and similar contexts.

References

[DdC] de Champeaux, D., "Object-Oriented Development Process and Metrics", Prentice Hall; ISBN: 0-13-099755-2, 1997.

[DdC2] de Champeaux, D., "AI Assessment", Conference: 2024 International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA), 2024 February. https://doi.org:10.1109/ACDSA59508.2024.10467324

[DdC3] de Champeaux, D., "Chat-Systems defects and fixes", submitted, 2024.

[Fodor] Fodor, J.A., "The Language of Thought", Harvard University Press,1975.

[GitHub] https://github.com/ddccc/BDDFS

[Holte] Holte, R., A. Felner, G. Sharon, N.R. Sturtevant, J. Chen, "MM: A bidirectional search algorithm that is guaranteed to meet in the middle", Artificial Intelligence, 252 (2017). https://dx.doi.org/10.1016/j.artint.2017.05.004

[Korf] Korf, R.E. "Artificial Intelligence Search Algorithms", https://dl.acm.org/doi/pdf/10.5555/1882723.1882745

[BiDirectionalSearch] https://en.wikipedia.org/wiki/Bidirectional_search

[PDDL] https://www.researchgate.net/publication/332160365_An_Introduction_to_the_Planning_Domain_Definition_Language

[Pohl] Pohl, Ira, "Bi-directional Search", in Meltzer, Bernard; Michie, Donald (eds.), Machine Intelligence, vol. 6, Edinburgh University Press, pp. 127–140, 1971.

[Sturtevant] Sturtevant, N.R. & A. Felner, "A Brief History and Recent Achievements in Bidirectional Search", Thirty-Second AAAI Conference on Artificial Intelligence, Vol. 32 No. 1, 2018. https://doi.org/10.1609/aaai.v32i1.12218

Appendix

Table 19 provides some details about the codes of different versions. These can have parameters for the sizes of grids, whether a search direction is *eased* or *hampered*, whether the ordering of candidate successor state are fixed or scrambled, whether there are range restrictions or not.

Version	Domain	# Stacks	# Threads	Control	Features
Grid2BF	grid	1	1	alternate	Best first search with 2 queues
Grid2	grid	1	1	alternate	B&U with parameters
Knight3	chessboard	1	1	alternate	B&U with dead location check
Knight5	chessboard	1	1		Forward & backward phase
Grab	grid	1	1	alternate	B with range restrictions
Grid3	grid	1	1	alternate	B&U grid with one barrier
GridT	torus	1	1	alternate	B&U grid without boundary
GridB2	grid	2	1	alternate	Compare against Grid2
Gridp	grid	2	2		Compare against Grid3, system stacks
GridBt	torus	2	1	alternate	Compare against GridT
Grid2p	grid	2	2		Compare against Grid2, system stacks
Grid4	grid	1	1	alternate	B grid with two barriers
GridB	grid	2	1		Compare against Grid4, 2 Java stacks
Gridp4	grid	2	2		Compare against GridB, 2 Java stacks
Grid2p4	grid	2	2		Compare against Grid2, system stacks
Gridp5	grid	2	2		Compare against Gridpp & GridB5, system stacks
Gridpp	grid	2	2		Compare against GridB5, 2 Java stacks
GridB5	grid	2	1	alternate	Compare against Gridpp, 2 Java stacks

Table 19 List of versions available on [GitHub]. B = bidirectional, U = unidirectional