Bidirectional Depth First Search Experiments

Dennis de Champeaux

Unaffiliated

Silicon Valley

2024

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 has its companion: an abstract, *bidirectional* depth first search version. 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 like keeping track of visited nodes and infrastructure for capturing forward/ backward status. Parallel versions and a two stack version 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, after 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 after backtracking is an option that can be helpful. We show also parallel versions with threads for forward and backward search. Yet another pseudo-parallel version uses two stacks for the forward and backward searches.

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 16 versions. 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 *findNodes(nodex)* which produces a set of adjacent nodes of the node *nodex.* [This function is called `findMoves` in the codes.] 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 findNodes(fx) or findNodes(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 findNodes(fx) or findNodes(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 when backtracking

The *findNodes* 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 *findNodes* 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.

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 when there is no provision for registering visited nodes. 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.

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 the bidirectional variant we compare bidirectional depth first search – with an example – against bidirectional *best* first search. The bidirectional variant alternates 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 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. A 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.

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 the situation at the goal side of the search space.

An abstract characterization of this variant is:

```
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
         return }
      else { depth++
      // explore successor nodes
      for each successor node nx of startNode {
          goForward(depth, nx, goalNode)
          restore }
      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 }
      return
   } // end 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 these enhancements by turned them 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 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 tiles 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.

We start by restoring a vacated tile to not-visited when backtracking. 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, 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 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:

```
  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 by both paths getting into trouble. Backtracking at one side wiped out progress at the other side. The forward path marched in the wrong direction after 18 and entered a dead-zone with the move of 26-27. It was never able to break out. The backward path moved first in the wrong direction (to the right at 30). It took forever to backtrack out of that dead-zone. We will revisit this example further down.

An eased bidirectional search:

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

The solution is found in the backward search either by 39-14 or 39-16.

Similar but more difficult with a hampered unidirectional search (move count 9567716 and timing 4408):

```
 1 10 11 12 15 16 19 20 21 24 25 26 41 42          52 53
 2  9  8 13 14 17 18    22 23    27 40 43 46 47 48 51 54
 3  4  7                         28 39 44 45    49 50 55 56       69 70
    5  6                         29 38 37 36             57       68 71
                                 30 31 34 35       60 59 58       67 72
                                    32 33          61 62 63 64 65 66  1
```

We proceed by showing the timing results on larger arrays in Table 2.

Size	Bidirectional alternate	Unidirectional
6 x 45	2	10340
6 x 50	3	failed
6 x 55	4	23
6 x 60	2	18
6 x 65	3	511
6 x 70	10	295
6 x 75	5	12
6 x 80	4	11
6 x 85	6	178
6 x 90	5	148
6 x 95	5	failed
6 x 100	4	3

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.

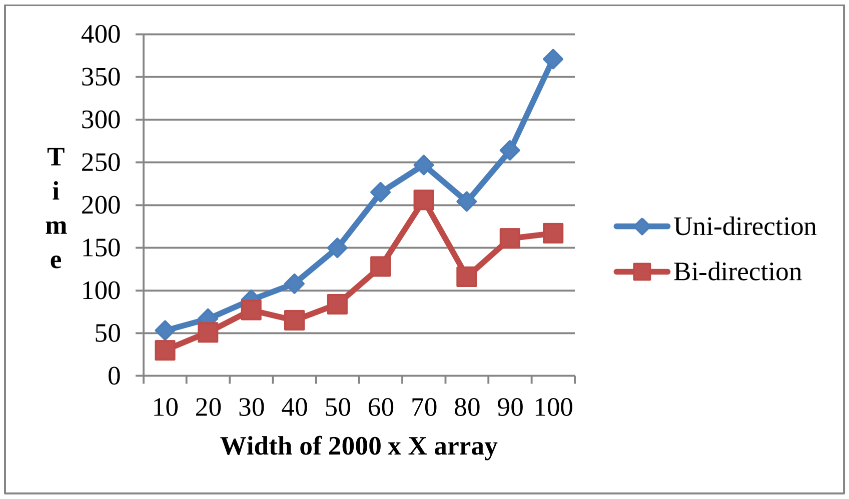
Size	Bidirectional alternate	Bidirectional shortest path
6 x 45	2	3
6 x 50	3	474
6 x 55	4	3516
6 x 60	2	failed
6 x 65	3	4
6 x 70	10	failed
6 x 75	5	failed
6 x 80	4	failed
6 x 85	6	failed
6 x 90	5	failed
6 x 95	5	failed
6 x 100	4	failed

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 provide more opportunities to enter dead-end zones.

Subsequently the code of Grid2 was modified so that the array grid tile was *not* restored when a recursive call came back. 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 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 revisited 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: 28
Uni-direction: 17

The code of Grid2 is available at [GitHub].

8 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.

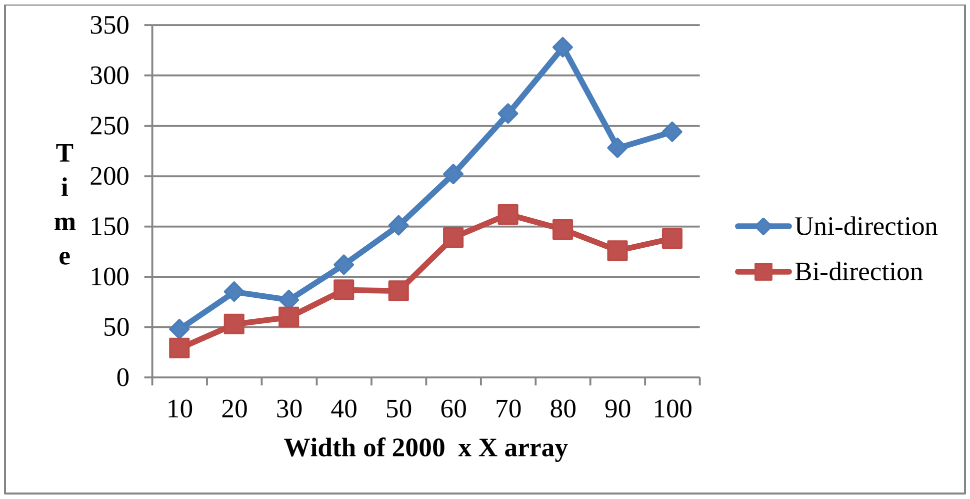
Size	Grid2	Grab
6 x 45	2	5
6 x 50	3	4
6 x 55	4	12
6 x 60	2	5
6 x 65	3	2
6 x 70	10	5
6 x 75	5	5
6 x 80	4	2226
6 x 85	6	5
6 x 90	5	5
6 x 95	5	9
6 x 100	4	5

Table 4 Bidirectional timings of Grid2 & Grab

The timing for the 6 x 80 search space must be caused by ‘bad luck’ due to one of the search directions getting stuck in a dead-zone. 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: avoid moves in the wrong direction, no restore after 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.

9 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.

```
xxxxxxxxxxxxxxxxxxx
xS       x        x
x        x        x
x        x        x
x                 x
x        x        x
x        x       Gx
xxxxxxxxxxxxxxxxxxx
```

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 when backtracking, scrambling and both directions were hampered (allowing moves in the wrong direction).

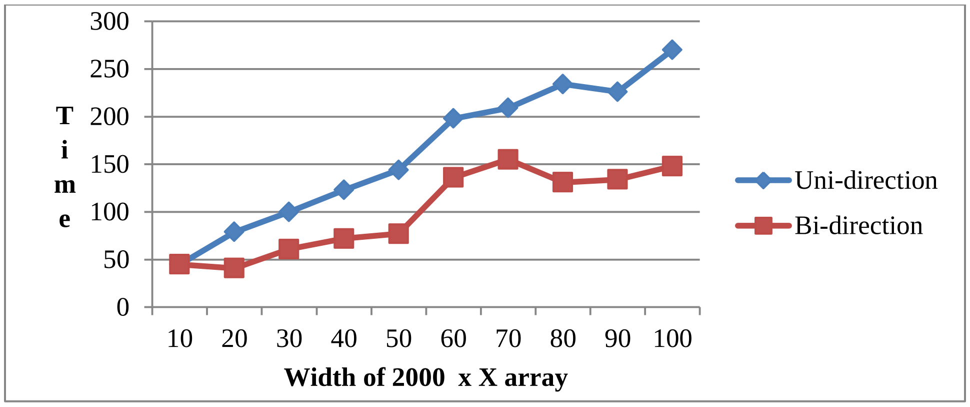
Size	Bidirectional	Unidirectional
6 x 45	5	19
6 x 50	9	failed
6 x 55	8	failed
6 x 60	9	1073
6 x 65	4	failed
6 x 70	9	6
6 x 75	8	failed
6 x 80	7	failed
6 x 85	5	failed
6 x 90	12	failed
6 x 95	6	777
6 x 100	14739	failed

Table 5. Bi- and unidirectional timings of Grid3.

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 is likely caused by entering dead-zone.

Similar to the Grid2 and Grab experiment, we used larger arrays than in the previous experiment. This time bad moves were blocked, 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.

10 GridT experiment

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 when backtracking. The performance is shown in Table 6with the timings and with the number of solutions in brackets.

X	Unidirectional	Bidirectional
5	25 (3)	24 (22)
10	48 (3)	36(59)
15	58 (3)	50 (139)
20	77 (3)	61 (191)
40	167 (3)	90 (4323)
60	194 (3)	148(29421)
80	217 (3)	126(44514)
100	295 (3)	167 (3399)
600	333 (3)	794 (353721)

Table 6. Bi- and unidirectional timings of GridT.

We see here also that the bidirectional version is doing mostly 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	Unidirectional	Bidirectional
5	37 (4)	1 (57)
10	53 (4)	7 (629)
15	63 (4)	10 (107)
20	83 (4)	24 (3331)
40	115 (4)	93 (53)

Table 7. Bi- and unidirectional timings of GridT.

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

The code of GridT is available at [GitHub].

11 Parallel Gridp & Grid3 and Gridpt & GridT experiments

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 threads of control; one for forward search and one for backward search. In pseudo code:

- Create forward and backward nodes
- Create a forward thread for the forward node and a backward thread for the backward node
- Start both threads, which will execute `move(true, startState)` and `move(false, goalState)`
- Wait for both threads to terminate

A thread executes code that checks whether a solution is found, whether an earlier node is encountered or it applies recursion on nodes produced by the *findNodes* function.

This experiment used a 2000 x X array grid for X = (100, 200, .., 1000) and has also a barrier half way the length of the array with a pinhole gap in the middle of the barrier. The settings are: scrambled, eased, no restore and terminate with one solution. Termination is as with Grid3 when a search direction finds a grid location visited by the other search direction.

The comparison of Grid3 and Gridp yielded the following timings:

X	100	200	300	400	500	600	700	800	900	1000
Grid3	13	12	9	37	9	12	148	164	10	241
Gridp	9	9	9	11	12	12	12	11	14	14

The parallel version is here mostly the winner. The code of Gridp is available at [GitHub]. This result inspired us to try out the torus domain with the Gridpt version. We used the same data range in the comparison of GridT versus Gridpt; see Table 8.

X	Gridpt	GridT
5	22 (3)	34 (22)
10	22 (509)	36 (59)
15	30 (7069)	50 (139)
20	32 (8026)	61 (191)
40	44 (9163)	90 (4323)
60	112 (9954)	148 (29421)
80	83 (9112)	126(44514)
100	94 (9425)	167(3399)
600	493(25711)	794 (353721)

Table 8. Timings for Gridpt and GridT

Yet again the two thread parallel version performs better. The code of Gridpt is available on [GitHub].

12 Two stack Gridpt & GridBt experiment

Given the success of using Gridpt with its two Java threads for forward and backward search we wondered whether we could do a better job by using a non-parallel version to avoid the synchronization overheads. Hence we tried a version that managed two stacks and used forward and backward functions each employing their own stack. Backtracking by a function would not impact achievements made by the peer code and its stack at the other side.

Simplified in pseudo code for GridBt:

```
• Create forward and backward stacks
• Create a forward node and a backward node and put them on the stacks
• Loop:
    o Terminate when a stack is empty
    o Decide forward or backward
    o Pop element and handle successors:
        ▪ Skip, process solution, or push successor
```

We employed this version, GridBt, also to the grid with the torus challenge. We obtained the following timing comparison against Gridpt; see Table 9.

X	Gridpt	GridBt
5	22 (3)	58 (12)
10	22 (509)	67 (345)
15	30 (7069)	97 (6171)
20	32 (8026)	109 (8853)
40	44 (9163)	142 (8929)
60	112 (9554)	254 (8733)
80	83 (9112)	288 (9166)
100	93(9425)	422 (9303)
600	493 (25711)	1471 (9706)

Table 9. Timings for Gridpt and GridBt

This version GridBt is here certainly *not* competitive against the two other versions. Still it has an advantage that is not captured by the timing data. We encountered runs where we had to increase Java stack size to avoid a stack overflow for Gridpt. That was *not* required for GridBt. Hence it can be used when stack overflows prevent obtaining solutions. This topic is revisited in Section 15 with Gridp5 and GridB5. The code of GridBt is available on [GitHub].

13 Grid4, GridB & Gridp4 experiments

Grid4 resembles Grid3, but instead of one barrier it has 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 – using the ‘normal’ version – was able to solve a size 5 x 12 array grid with the settings: bidirectional, restoration when 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 when backtracking, *no* solution was obtained, producing:

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

  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 stacks with two simulated threads, we did get a solution with:

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

  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
```

Instrumenting Gridp4, which is using parallel threads, a solution was obtained also:

```
  1  2  3          11 10  9  8
        4       13 12        7
        5 12 13 14 15        6
        6 11       16        5
        7 10       17    21  4
        8  9       18 19 20  3  2  1

  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 10.

20 x X	Grid4	GridB	Gridp4
1600	10	9	8
1800	fail	8	8
2000	fail	11	6

Table 10 Timings for 3 versions. Grid4 & GridB use 1 thread, Gridp4 uses 2 threads.

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

14 Grid2p4 experiment

The Grid2 section has an application to bidirectional hampered search on a 6 x 25 grid as discussed in Section 7. 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 timing 74 and move-count 54470. Given the success of Gridp4 with its two threads the version Grid2p4 is like Grid2p but without its barriers. 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 can cause 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
```

The version Grid2p4 is available in [GitHub].

15 Gridp5, Gridpp & GridB5 experiment

This experiment challenges versions with larger grids, 20 x 20000, and with up to10 barriers. Gridp5 employs two threads using the Java system stack. Gridpp employs two threads with two ‘private’ stacks. GridB5 has similarly two ‘private’ stacks but uses only one thread of control. Tables 11, 12 and 13 have the timing to get a single zig-zag solution while negotiating 10 barrier obstacles.

20 x 20000	Restore after backtrack	No restore after backtrack
Eased	70	79
Hampered	359 (1279)	367 (821)

Table 11 Timings of Gridp5 for finding a solution path. See the text for the timings in brackets.

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 11 measures when a solution is found; the numbers in brackets measures when the forward and backward threads have unwound their stacks and the evoking thread has regained control.

20 x 20000	Restore after backtrack	No restore after backtrack
Eased	80	81
Hampered	336	308

Table 12 Timings of Gridpp for finding a solution path.

Gridpp using ‘private’ stacks instead of Gridp5 relying on the system stacks is clearly a distinct improvement over Gridp5.

20 x 20000	Restore after backtrack	No restore after backtrack
Eased	64	51
Hampered	368	342

Table 13 Timings of GridB5 for finding a solution path.

The two alternating paths of the GridB5 version do not depend on the Java system stack but, instead, they employ also the ‘private ‘stack functionality provided by the Java language. Hence there is also no need to increase the system stack and unwinding the own stacks is apparently way easier as shown in Table 13. The Gridpp version was designed and implemented after the comparison between Gridp5 and GridB5 with the expectation that it would be better than both of them. Not clear is why Gridpp with its two threads doesn’t have a substantial better timing than GridB5.

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

16 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 found 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. We experimented with having a single thread of control managing one or two stacks and with two threads of control each managing its own stack. There are variants how a single thread of control selects between forward and backward explorations; alternation is an option, shortest current path length is an option, etc.

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.

Yet another 'trick' is using forward search up to a specified depth and subsequently switch to backward search – as shown for the knight tour domain.

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

17 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 (with16 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 backtracking forgets visited nodes, 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 ‘private’ stacks with one or two threads of control appear to be the better design choices.

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 better 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 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 contexts like them.

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.
DOI: 10.1109/ACDSA59508.2024.10467324

[DdC3] de Champeaux, D., "AI Knowledge and Meaning Representation for Chat-systems", 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 14 provides some details about the different codes. Versions 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
qqGridp	grid	2	2		Compare against Grid3
Gridpt	torus	2	2		Compare against GridT
GridBt	grid	2	1	alternate	Compare against Gridpt, 2 Java stacks
Grid4	grid	1	1	alternate	B grid with two barriers
GridB	grid	2	1	alternate	Compare against Grid4, 2 Java stacks
Gridp4	grid	2	2		Compare against GridB
Grid2p4	grid	2	2		Compare against Grid2
Gridp5	grid	2	2		Compare against Gridpp & GridB5
Gridpp	grid	2	2		Compare against GridB5, 2 Java stacks
GridB5	grid	2	1	alternate	Compare against Gridpp, 2 Java stacks

Table 14 List of versions available on [GitHub]. B = bidirectional, U = unidirectional