File: ddc/AIQV/BDDFS.docx

2022 May 2/ Aug 27/ Sep 25/ Oct 8/ Nov 12

Bidirectional Depth First Search

Dennis de Champeaux

Unaffiliated

2022

Abstract

Search techniques with uninformed, and informed, unidirectional, and bidirectional versions and with additional subversions are core assets of AI. This paper describes a bidirectional depth first search version (and variants): a candidate addition to the collection of search techniques. Examples are shown with unfavorable and favorable comparisons against unidirectional depth first search.

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]. Depth first search is an exception and hence this paper is of the 'green field' type. We give an abstract characterization of a bidirectional depth first search algorithm. A 'trick' is to apply unidirectional search selectively in forward and backward directions. Not increasing the branching ratio is an obvious concern and is addressed by *not* operating in the product space of the two directions. In addition, we have a parallel version with forward and back threads and a semi-parallel version using two stacks. These versions are tested on different problem families yielding unfavorable and favorable outcomes in the comparisons against unidirectional depth first search. We discuss the results of these experiments.

Abstract Algorithm

The algorithm assumes 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*. Nodes have attributes so that we can record, for example, the parent node of a node with which we can obtain solution paths. The algorithm is flexible and allows unidirectional search in either direction. We also allow at each iteration deciding to go forward or backward. The abstract algorithm starts with:

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

This invokes:

```
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)
   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 number of nodes encountered in both directions can be used as well; etc.

If we keep track of node 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)
   for each node y in successorNodes do {
      if (go forward) {
          if ( y was visited in the backward search ) exit with a solution
          move(y, bx)
      } else {
          if ( y was visited in the forward search ) exit with a solution
          move(fx, y)
      }
} // end of move
```

These two versions have a single stack. We will discuss also a single thread two-stack version serving forward and backward search. Yet another version exploits Java's multi-threading functionality with two threads serving respectively forward and backward search.

Some of our implemented 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 a search domain that is used for keeping track of visited nodes when back tracking
- Filtering optionally the nodes generated by *findNodes* to change the difficulty level of a problem domain; unfiltered is labeled as 'hampered', filtered as 'eased'
- Scrambling optionally the list of nodes generated by the *findNodes* function. Scrambling can make finding a solution path task harder or even impossible with some combinations of these parameters.

These parameter settings yield a large space for experiments that can be applied to each problem domain that we will encounter. Obviously we can present only a part of all combinations that are possible, which opens up opportunities for more research.

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.

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.

Knight tour experiment

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 some dead-end recursions and about checking that a pre-solution satisfies the cyclic requirement. Otherwise it follows the pattern described above.

We obtained over 40M solutions on an 8x8 board after a week of processing, after which we give up. The number of solutions on a 6x6 board is 9862. The timings of a Java implementation in milliseconds on an I5 machine were as follows:
Forward: 28536
Bi-direction: 36970

Using the bidirectional mode is here obviously not an improvement.

The code of Knight3 is available at [GitHub].

Grid2 experiment

This experiment aims to find one or more paths in a 2-dimensional grid. We start with an array sized 6 x 12 with the start state in one corner and the goal state at the opposite side at the end of the diagonal. The termination condition is obtaining a single solution by 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). 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 (to the left in forward search and to the right in backward search).

Here an example of a path obtained with these specs with a bi-directional search:

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

This solution is found during the backward move by the node labeled ‘B’ and encountering the node labeled ‘F’.

A unidirectional forward search yielded:

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

The latter pattern shows clearly that in this version the *findNodes* function includes 'bad' moves; i.e. here moves backward to the left.

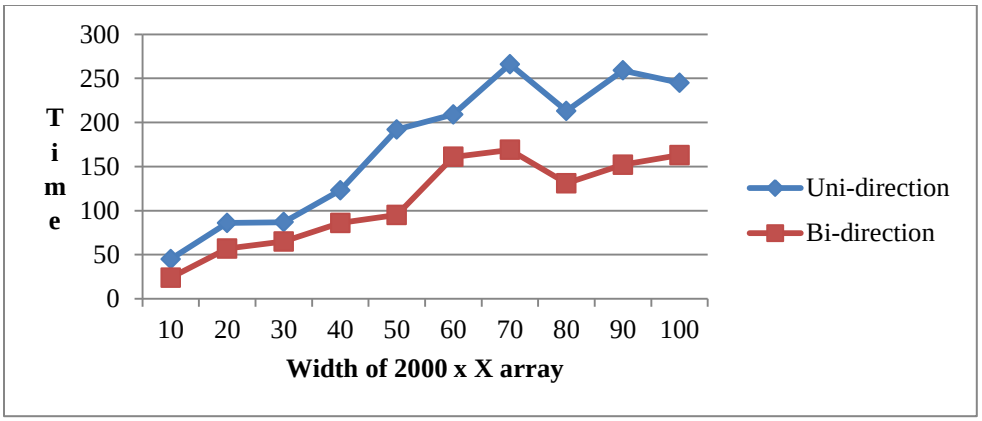
We proceed by showing the timing results on larger arrays in Table 1.

Size	Bidirectional	Unidirectional
6 x 45	6	0
6 x 50	7	failed
6 x 55	6	32
6 x 60	6	217
6 x 65	7	116
6 x 70	10	18
6 x 75	9	15933
6 x 80	11	failed
6 x 85	6	791
6 x 90	6	failed
6 x 95	7	83
6 x 100	7	failed
Table 1. Bi- and unidirectional timings of Grid2		

The situation changes dramatically when the *findNodes* function filters out moves in the wrong direction. The timings on a 6 x 100 array for bi- and unidirectional search become both one.

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: 25, 52, 627, 845, 11064, 6150, 12069, 4312, 24703, and 2975. Hence the time differences are actually more in favor for bidirectional than shown, although, as mentioned above, subsequent explorations benefit from the revisiting tiles earlier.



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

The code of Grid2 is available at [GitHub].

Grab experiments

This experiment uses a 6x12 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 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 bi-direction, single solution we obtain;

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

The trace shows a forward move (6 to 7) going in the wrong direction. We repeat the test we did for Grid2, see Figure 3.

Size	Grid2	Grab
6 x 45	6	2
6 x 50	7	3
6 x 55	6	7
6 x 60	6	5
6 x 65	7	4
6 x 70	10	6
6 x 75	9	5
6 x 80	11	4569
6 x 85	6	4
6 x 90	6	6
6 x 95	7	10
6 x 100	7	9

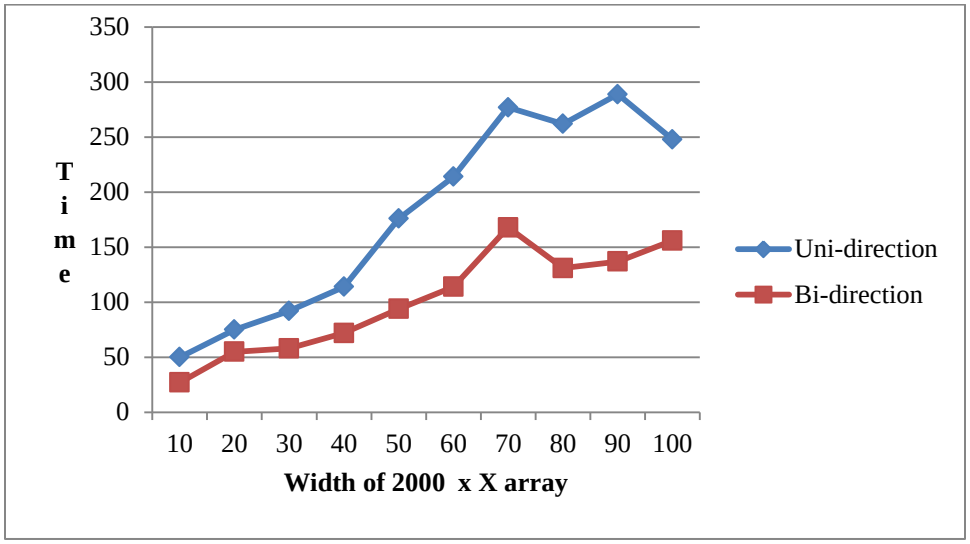
Table 2 Bidirectional timings of Grid2 & Grab

The timing for the 6x80 search space must be caused by ‘bad luck’ from scrambling the output of the *findNodes* function. Using instead the 7*80 search space the timing becomes just 5.

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 4, 7, 8, 13, 13, 11, 17, 20, 16.



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.

Grid3 experiments

This experiment used also a 6x12 array grid but has a barrier half way the length of the array with a pinhole gap in the middle of the barrier.

```
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.

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.

The code of Grid3 is available in [GitHub].

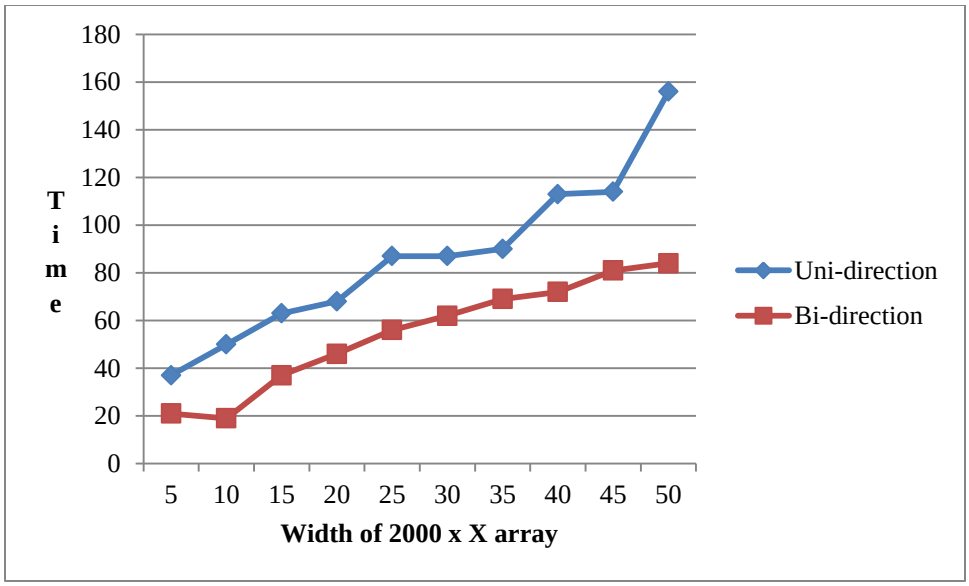
We proceed by showing the timing results on larger arrays in Figure 4. We used restoration when backtracking, scrambling and both directions were hampered (allowed moves in the wrong direction).

Size	Bidirectional	Unidirectional
6 x 45	3	failed
6 x 50	3	failed
6 x 55	2	failed
6 x 60	2	failed
6 x 65	2	failed
6 x 70	5	11618
6 x 75	3	failed
6 x 80	1	337
6 x 85	1	293
6 x 90	2	failed
6 x 95	4	2273
6 x 100	2	49753
Table 3. Bi- and uni-direction timings of Grid3.		

The timings in Table 1 and Table 3 show the consequence of the halfway barrier: uni-directional search ‘struggles’ to find solutions.

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: 11, 43, 22, 50, 3, 101, 35, 2, 101, and 997.



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

GridT experiment

This experiment uses similarly a rectangular array, but it has no borders, because the edges refer back to the other side; it can be seen as a grid laying on a torus. We use: a single solution, scrambled, both directions eased, no restore. An example of a unidirectional snake-path 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: 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 the next table with the timings and with the number of solutions in brackets.

X	Unidirectional	Bidirectional
5	31 (3)	31 (20)
10	42 (3)	36 (57)
15	97 (3)	45 (122)
20	86 (3)	52 (193)
40	119 (3)	97 (18903)
60	199 (3)	144 (29990)
80	211 (3)	129 (46051)
100	234 (3)	164 (55154)
600	fail	fail

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

The code of GridT is available at [GitHub].

Parallel Gridp experiment

We have shown numerous examples where our bidirectional version (with some parameters) outperforms the unidirectional version. Still we noticed that there were occurrences of repeated back tracking at one direction that could undo good progress at the other side. This inspired us to try 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 alternatively it recurses deeper 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	26	16	34	11	10	136	3943	9	415
Gridp	9	9	18	10	10	10	11	11	10	13

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

X	Gridpt	GridT
5	20 (10)	31 (20)
10	23 (480)	36 (57)
15	26 (5277)	45 (122)
20	37 (8143)	52 (193)
40	41 (9121)	97 (18903)
60	113 (9303)	144 (29990)
80	90 (9402)	129 (46051)
100	94 (9478)	164 (55154)
600	483 (9854)	Fail

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

Two 'thread' 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 avoiding the synchronization overheads. Hence we tried a version where we created and managed two stacks and used forward and backward codes employing their own stack. Backtracking by a code would not impact achievements made by the peer code and its stack at the other side.

Simplified in pseudo code:

- Create forward and backward stacks
- Create a forward node and a backward node and put them on the stacks
- Loop:
    - Terminate when a stack is empty
    - Decide forward or backward
    - 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:

X	Gridpt	GridBt
5	20 (10)	52 (12)
10	23 (480)	77 (348)
15	26 (5277)	83 (6116)
20	37 (8143)	109 (8941)
40	41 (9121)	150 (8906)
60	113 (9303)	256 (8716)
80	90 (9402)	346 (9112)
100	94 (9478)	404 (9231)
600	483 (9854)	1555 (9636)

This version GridBt is certainly *not* competitive against the two other versions. Still it has an advantage that is not captured by the timing data. We have encountered runs where we had to increase Java stack space to avoid a stack overflow. That has not been required for GridBt. Hence it can be used when stack overflows prevent obtaining solutions. The code of GridBt is available on [GitHub].

Grid4 & GridB

Grid4 resembles Grid3, but instead of one barrier it has two with pinhole gaps at opposite sides:

```
xxxxxxxxxxxxxxxxxxxxxxxx
xS      x              x
x       x       x      x
x       x       x      x
x       x       x      x
x       x       x      x
x               x     Gx
xxxxxxxxxxxxxxxxxxxxxxxx
```

Grid4 – using the 'normal' version – was able to solve a size 5x12 array grid with the settings: bidirectional, restoration when backtracking enabled, scrambles 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
```

Switching to GridB, which is using two stacks with two 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, we obtained also a solution:

```
 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) appear to be better equipped for handling difficult search domains. The versions Grid4, GridB and Gridp4 are available in [GitHub].

Discussion

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. We relied most on just alternation between forward and backward search. That may not work best when the search space has asymmetric obstacles. We have provided some infrastructure to support alternatives for plain alternation but we have not explored their capabilities. Restorations or not after backtracking has pros and cons, which need further exploration. We used first grid elements to record their (un)visited status. Subsequently we used a data structure that records visited nodes. This has created overlapping functionality, which – we think – allows for other opportunities. Meeting in the middle of the search space has not been tracked – likely another opportunity to explore.

Conclusion and next challenges

We are not aware of earlier investigations of bidirectional depth first search. We have described a novel, abstract bidirectional depth first algorithm and three implemented designs:

- A depth first forward search combined with a depth first backward search in a sequential setting
- As above but using two parallel threads that are supported by Java
- As above but using two stacks supporting forward and backward searches in again a sequential settings

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 in candidate moves, whether or not the move generator can scramble or not the order of candidate moves, etc.

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 versions can be applied to them. The PDDL language [PDDL] is an attempt to facilitate the specification of problem families and their instances. 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, creating the required data structures, the classes and objects with their attributes and functional components, remains elusive and requires a breakthrough.

References

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

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

[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.