File: ddc/AIQV/BDDFS.docx

2022 May 2

Bi-directional Depth First Search

Dennis de Champeaux

ddc2@ontooo.com

2022

Abstract

Search techniques with uninformed, and informed, unidirectional, and bi-directional versions and with additional subversions are core assets of AI. This paper describes a bi-directional depth first search version: 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 bi-directional version; see [BiDirectionalSearch]. Depth first search is an exception. This paper gives an abstract characterization of a bi-directional depth first search algorithm. The 'trick' is to apply unidirectional search in the product space of nodes. Not increasing the branching ratio is an obvious concern and is addressed by doing forward search *or* backward search at each iteration. We implemented different versions of the abstract algorithm for different problem families. The comparisons against unidirectional depth first search yielded unfavorable and favorable outcomes. We discuss the results of these experiments.

We describe also a parallelized version for an example where one thread does forward search and the 2nd thread does backward search.

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, while we also allow to go forward or backward at every iteration. 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 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. Suppressing nodes in *findNodes* that go in the wrong direction can be an option. Recognizing nodes encountered earlier can be used to improve performance.

The parallel version uses a move operator with a different signature: an argument indicating the direction of the search and a 2nd argument with a forward or backward node.

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 because 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, 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.

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

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 I3 machine were as follows:

Forward:	132442
Backward:	134882
Bi-direction:	218579

The code of Knight3 is available at [GitHub].

Grid experiment

The search was done on a 2-dimensional 6x10 array with the start state in one corner and the goal state at the opposite side at the end of the diagonal. Forward and back search find the same number of solutions: 8678016, with virtual identical timings of 2000 milliseconds. The bi-directional version produces over 1.4B solutions with a timing of 223137. This result reminds of the pioneering bi-directional heuristic search experience by Ira Pohl [Pohl]: the search trees did not meet in the middle of the search space. The first three solutions demonstrate what is going on with the start state in the top left corner and the goal state in the bottom right corner:

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

Each time the forward search terminates with a solution. The backward search creates irrelevant paths. The code of Grid is available at [GitHub].

<u>Grid 2 experiment</u>

The code of Grid was modified so that the array grid tile was *not* restored when a recursive call came back. The ordering of the output of the *findMoves* function was randomized to make the task more challenging, which is reflected in the snake-like path shown below.

The forward search generated two solutions with timing 1. The visited tiles are:

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

The backward solution generated also two solutions also with timing 1 and a similar pattern of visited tiles. The bi-directional version generated 6 solutions with timing 0. The visited tiles are:

```
f   f   f   f   f   b   b   b

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

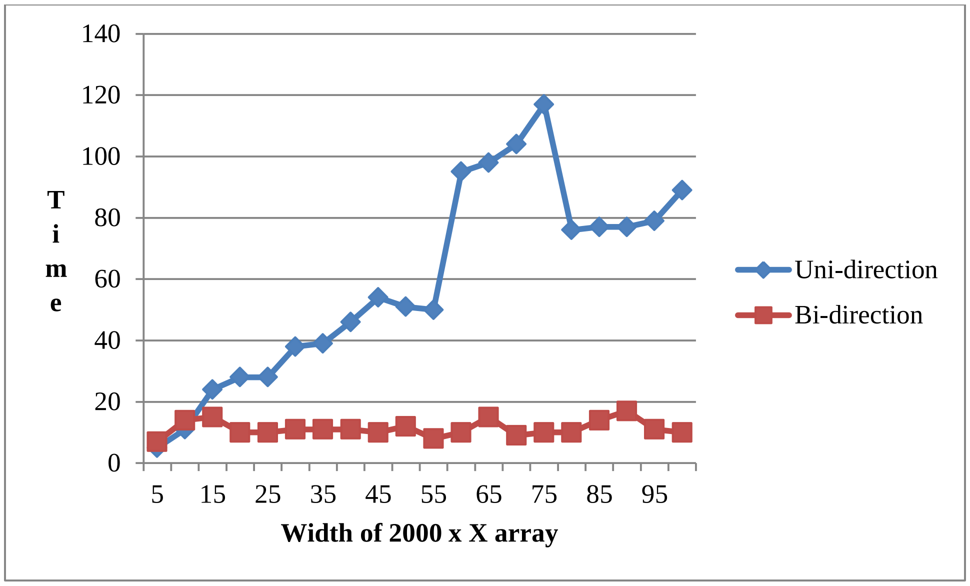
This example demonstrates an example where bi-directional depth search is doing better. Here an example solution of the bi-directional version; the array element where the forward search finds a backward search element is marked with F:

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

The code of Grid2 is available at [GitHub].

<u>Grid 2 next experiment</u>

We used for this experiment the same code as in the previous experiment but used larger grids. The next graph has the timings for unidirectional and bi-direction search for finding paths in arrays in arrays 2000 x [5-50]. Unidirectional search finds each time two solutions, and the bi-direction search finds a various number of solutions, respectively: 9, 12, 15, 25, 37, 12, 64, 18, 23, 63, 32, 33, 42, 24, 36, 40, 30, 49, 40, 15. Hence the time differences are actually more in favor for bi-direction than shown.



The bi-direction search distinctly outperforms unidirectional search in this example.

Grab experiment

This experiment used also a 6x10 array grid but prevented in the bi-directional version the forward exploration to enter the territory of the backward exploration, and similar for the backward exploration. The unidirectional search found 2 solution paths, while the bi-directional version found also 2 paths. The unidirectional forward search had to visit the whole array as shown by:

```
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   f   f   f   f

f   f   f   f   f   f   f   f   f   f

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

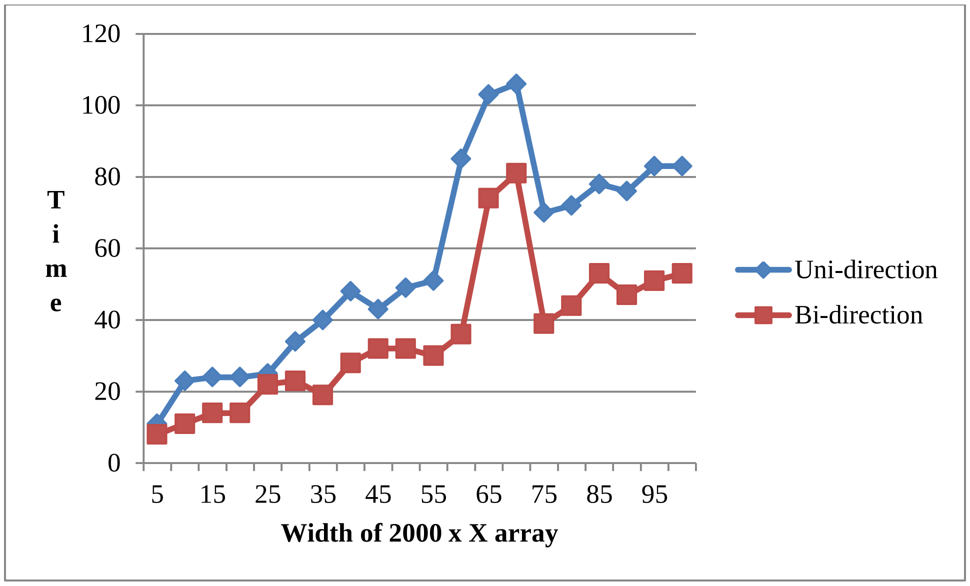
In contrast the bi-directional version visited only:

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

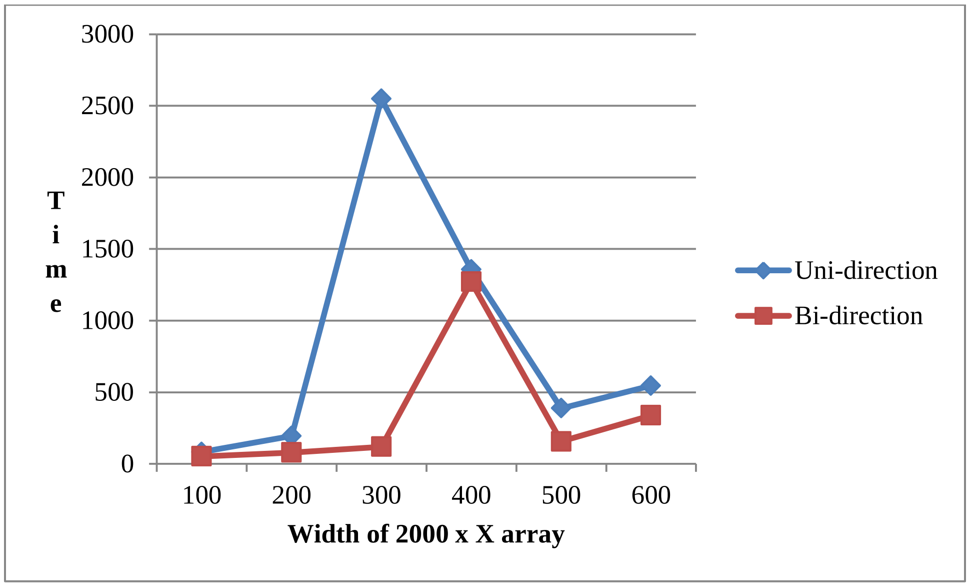
The code of Grab is available at [GitHub].

Grab next experiment

Similar to the Grid 2 experiment, we used larger arrays than in the previous experiment. The next graph has the timings for unidirectional and bi-direction search for finding paths in arrays in arrays 2000 x [5-50]. Unidirectional search finds each time two solutions, while the bi-direction search finds a varying number of solutions: for array size 5: 9 and subsequently 5, 2, 3, 8, 5, 9, 11, 8, 12, 9, 12, 8, 10, 14, 15, 7, 9, 24, 20.



The next graph shows larger ranges than the previous graph. The unidirectional search finds each time two solutions, while the bi-direction search finds a varying number of solutions: for array size 100: 20 and subsequently 29, 55, 82, 80, 105.



<u>Grid3 experiment</u>

This experiment used also a 6x10 array grid but has a barrier half way the length of the array with a pinhole gap in the middle of the barrier. The unidirectional search found 2 solution paths, while the bi-directional version found only 1 path. Scrambling of the move operators was not done as shown by the following non-snake paths:

Uni:

```
 f  f  f  f  f  f
                f
```

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

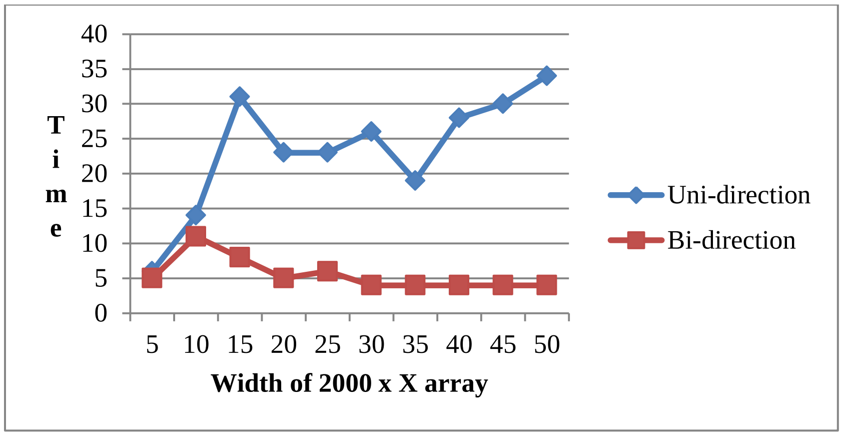
Bi:

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

The code of Grid3 is available in [GitHub].

<u>Grid3 next experiment</u>

Similar to the Grid2 and Grab experiment, we used larger arrays than in the previous experiment. 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 bi-direction search finds a single solution.



<u>GridT experiment</u>

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 is a grid laying on a torus. 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   f   f   f       f   f   f
  f   f   f   f   f   f       f   f   b
```

A bi-directional solution with less visited node is:

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

The first column is an example where the forward path moved to the other side.

GridT next experiment

This experiment used a 2000 x X array torus grid for X = (5,10,15,20,40,60,80,100). The unidirectional version solved only the first three examples and failed due to stack overflow on the remaining examples. The performance is shown in the next table with the timings and with the number of solutions in brackets.

X	Unidirectional	Bi-directional
5	21 (3)	21 (20)
10	27 (3)	25 (30)
15	37 (3)	24 (32)
20	-	31 (40)
40	-	31 (62)
60	-	19 (6)
80	-	16 (70)
100	-	22 (121)

This example shows that bi-directional depth first search can be qualitatively better by obtaining solutions where the unidirectional version fails.

The code of GridT is available at [GitHub].

Gridp experiment

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 difference is using two threads of control to explore respectively forward from the start state and backwards from the goal state. The move operation uses instead of a forward node and a backward node, a Boolean indicating the direction to be pursued and a forward or backward node. The two threads of control start respectively with `move(true, startState)` and `move(false, goalState)`. Termination is as with Grid3 when a search direction finds a grid location visited by the other search direction.

Expected speedup was *not* obtained as shown by the timings on the I3 machine for the range 2000 x X:

X	100	200	300	400	500	600	700	800	900	1000
Grid3	3	3	4	3	5	3	5	3	5	3
Gridp	4	4	4	5	6	4	5	5	3	5

Repeating this test on a newer I5 machine, while there was a speed up for both versions, the ratios of (sum of times Gridp)/ (sum of times Grid3) hardly changed with the average 1.24 in favor of the sequential version Grid3. The sequential version visits less grid tiles because it maintains a count of the encountered tiles for each side and decides dynamically which side to explore, which is not happening in the parallel version. For example, the sequential version visited on the 1000 x 2000 grid 2500 forward and 2499 backward nodes while the parallel version terminated with 2500 forward and 4185 backward nodes.

The code of Gridp is available at [GitHub].

Discussion

The comparisons of uni- and bi-directional depth first search yielded different outcomes. Meeting in the middle of the search space can be challenging as shown by the cyclic knight tour. Bi-directional search takes there more effort than unidirectional search. Another implementation for finding paths between corners of a small 6 x 10 rectangular grid yielded an astonishing difference: 10M paths for unidirectional search versus over 1.4B paths for bi-directional search with close to identical timings per solution path.

Adjusting the second implementation by remembering visited nodes (tiles in the rectangular grid) (instead of restoring them after returning from a recursive call) changed the outcome. This adjustment relies on marking grid tiles with a forward or backward indicator, which resembles maintaining the collections of closed nodes in other search variants. This version favors the bi-directional version.

A third implementation – using also a rectangular grid – has a modified algorithm that reduces the number of nodes/tiles visited in the bi-directional version. By keeping track of the frontiers of the territory of the two explorations, an exploration avoids entering the other side. This modification favors also the bi-directional version.

A fourth implementation used a barrier midway the rectangular grid with a pinhole gap. The bi-directional version exploited this constraint better than the unidirectional version.

A fifth implementation is a parallelized modification of the fourth implementation. Expected speed up was not obtained because the two threads are not ‘aware’ of each other’s progress.

Conclusion and next challenges

We described a novel bi-directional depth first algorithm and the results of several implementations; one with an unfavorable comparison against unidirectional depth first search and others with favorable comparisons. A two thread parallel version did not deliver the expected speed up and requires better ‘cooperation’ between them.

Bi-directional depth first search has turned out to be way more challenging than initially envisioned. This could be the reason why it has not been explored earlier. 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 is how to create the different instantiations of the abstract algorithm ‘automatically’, i.e. without ‘manual’ coding. 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

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