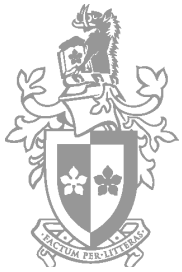
Tree Based Search

COS30019–Introduction to AI

Swinburne University of Technology

Alex Cummaudo 1744070

Semester 1, 2016



Abstract

A tree-based search can be implemented with both uninformed (brute-force) and informed (heurstic-based) searches. This assignment implements a small program demonstrating both types via an AI solver of an n by m puzzle. Utilised in the solver is Breadth-First and (Iterative-Deepening) Depth-First uninformed searches, Greedy-Best First and (Iterative-Deepening) A* Search informed searches.

Acknowledgements

I would like to acknolwedge Bruce (1998) and Mackworth (2013) for their coursework materials on iterative-deepening A* searches, as well as Anderson (2012) for his notes on his space and time complexity for the uninformed searches used in this assignment. Russell and Norvig (2009) was used for reference in all required search implementations also, chiefly in their provided psuedo-code for the algorithms.

Additionally, the article provided by Red Blob Games (2014) helped me optimise my search algorithms significantly. For random state generation, I utilised several theorms devised by Gong (2000), which you can see implemented in the `RandomState.swift` file.

When researching variant heuristics, I utilised works by Deza and Deza (2009) for implementing Euclidean distance, Cantrell (2000) for Chebyshev distance, and lastly Krause (2012) for Manhattan distance implementation.

Code History

The history for the source code provided is version controlled via git and can be found on GitHub. Refer to `http://github.com/alexcu/puzzle-solver/commits/master`.

Prerequisites

- Swift 2.2, which you can download from at `https://swift.org/download/`.
- X11 windowing system[1]

[1] If using a Linux distribution such as Ubuntu, you should have the required X11 libraries already installed. If using OS X, you can use Xquartz. You can download Xquartz from `http://www.xquartz.org`.

Contents

1 Features

A more extensive list of search features can be shown using the `--help` switch.

1.1 Search Algorithms

The solver implements the following search methods:

- Uninformed
  - Breadth-First Search, `BFS`
  - Depth-First Search, `DFS`
  - Iterative-Deepening Depth First Search (IDDFS), `IDS,CUS1`
  - Bogosort Search, `BOGO,CUS3`
- Informed:
  - Greedy Best-First Search, `GBFS`
  - A* Search, `AS`
  - Iterative-Deepening A-Star Search (IDA*), `IDAS,CUS2`

1.1.1 Thresholds

IDDFS and IDA* searches can utilise the `--threshold` switch when executing, which will specify the initial maximum node path cost, or $pc(n)$, to traverse, for IDS, or the limit to the evaluation function result, $f(n)$, that will be accepted for searches, for `IDAS`). That is, for a threshold of t, the following:

$$pc(n) < t_{\text{IDDFS}} \tag{1}$$

$$f(n) < t_{\text{IDA*}} \tag{2}$$

As these are iterative depth searches, the initial threshold is *doubled* each time nodes are calculated to no longer be within the current threshold. These nodes are temporarily stored in a *fallback* frontier and are used only if there are no more nodes to search for, thereby initialising the next *iteration*. Refer to Section 2.1 for more detail.

1.1.2 Heuristics

By default, the heuristic used for informed searches is the Manhattan Distance heuristic, shown below in Equiation 3. The distance of state, s, of an n by m puzzle is calculated by the sum of the absolute differences of a tile t's coordinates from the state to it's goal state, $\bar{s}$.

$$h(s)_{\text{manhattan}} = \sum_{t=1}^{n \times m} (\mid x_t(s) - x_t(\bar{s}) \mid + \mid y_t(s) - y_t(\bar{s}) \mid) \tag{3}$$

Similarly, the Chebyshev and Euclidean Distances are also avaliable for heuristics calculation. See 4 and 5, respectively. The last avaliable heuristic is the misplaced tiles count, which is simply the number of mismatches in the position tiles

$$h(s)_{\text{chebyshev}} = \sum_{t=1}^{n \times m} (\textbf{max}(\mid x_t(s) - x_t(\bar{s}) \mid , \mid y_t(s) - y_t(\bar{s}) \mid)) \tag{4}$$

$$h(s)_{\text{euclidean}} = \sum_{t=1}^{n \times m} (\sqrt{(x_t(s) - x_t(\bar{s}))^2 + (y_t(s) - y_t(\bar{s}))^2}) \tag{5}$$

The heuristic used on an informed search can be changed to other heuristics by using the `--heuristic` switch and providing one of: `manhattan`, `euclidean`, `manhattan`, `chebyshev` or `misplaced`. Refer to the help for more instructions.

1.2 Graphical Solver

A graphical solver can show both the search method solving the puzzle as well as a solution to the puzzle where state's in the node tree are represented by the n by m matrix as coloured tiles. See Figure 1.1.

The GUI can be initiated using the following switches:

- **`--gui=solved`** Show only the solved puzzle. Traverse the search tree from the root node to the goal node using the `B` or ↑ Keys and down the tree using the `F` or ↓ Keys
- **`--gui=solving`**[2] Show the search algorithm solving the puzzle. When solving the GUI will not be responsive, so should the solver need to be quit, termination at the

[2]Note that when using the solving GUI, the search algorithm may appear to be slower than when not using the GUI. This is due to render lag caused by X11, used to render the GUI.

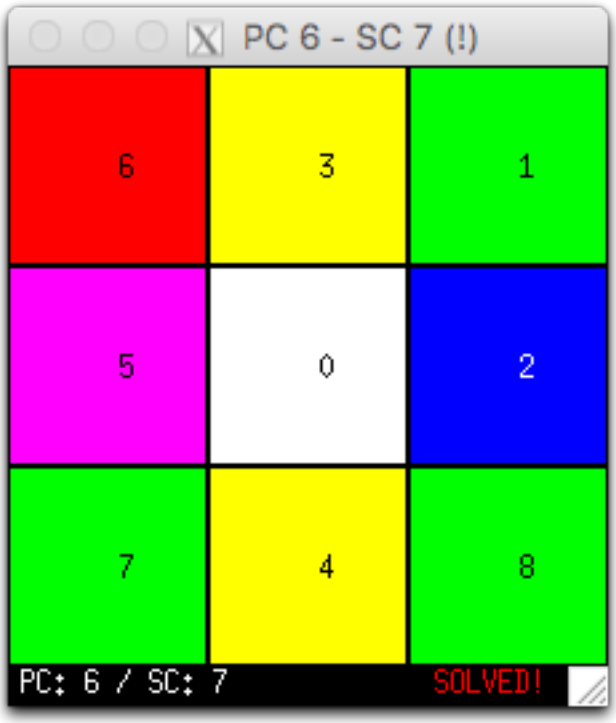


Figure 1.1: GUI of a puzzle solver

command line using `^C` is required

2 Discussion

2.1 Search Traversal

The puzzle is solved using the `SearchMethod` protocol's `traverse:` method, which is inherited via protocol extension to all concrete implementations of the variant search methods defined in Section 1.1. Refer to Listing 1 below for the implementation of search traversal.

Depending on on the search's frontier type which is used in the search, the search will alter the way in which nodes are popped and pushed to and from the frontier. Refer to Section 2.3 for more.

Listing 1: Search traversal for searches is implemented using a protocol extension method

```
func traverse(node: Node) -> Node? {
    // Clear the existing frontier we have as this is a new search
    self.frontier.collection.removeAll()
    // Maintain a dictionary of hash values to test where the node has come
    // from for repeated state checking; that is, check if the node is on a
    // path with a repeated state below. The root node has not come from
    // anything, thus nil
    var cameFrom: Dictionary<Int, Int?> = [node.hashValue: nil]
    // Push the first node on
    self.frontier.push(node)
    // While the search method condition is true
    while !frontier.isEmpty {
        // Force unwrap of optional as frontier isn't empty
        let currentNode = self.popFrontier()!
        // Goal test
        if self.isGoalState(currentNode) {
            SearchMethodObservationCenter.sharedCenter
                .notifyObservers(currentNode, isSolved: true)
            return currentNode
        } else {
            // Only add the children whos hash values are not stored as keys
            // in the nodes that this node has come from
            let childrenToAdd = currentNode.children.filter {
                !(cameFrom.keys.contains($0.hashValue))
            }
            // Update prvious states for the children we are about to add
            for child in childrenToAdd {
                cameFrom.updateValue(child.hashValue,
                                     forKey: currentNode.hashValue)
```

```
                }
                // Notify change
                SearchMethodObservationCenter.sharedCenter
                    .notifyObservers(currentNode, isSolved: false)
                // Push new children
                self.pushToFrontier(childrenToAdd)
            }
        }
        return nil
    }
```

2.2 Iterative Depth Search Implementation

Where an iterative depth search is used, the pushing and popping of a frontier may cause a node to be pushed to its *fallback* frontier (a FIFO frontier), that is, the frontier that stores all nodes that exceed the current threshold.

An iterative search with threshold, t, a concrete frontier, f, and a fallback frontier, $\bar{f}$, would use the node threshold comparator function[3], $T(n)$, to calculate values from a node, n, and see which frontier should be used. Thus:

$$f = \{\ n \mid T(n) < t\ \} \tag{6}$$
$$\bar{f} = \{\ n \mid T(n) \geq t\ \} \tag{7}$$

Hence, for example, consider the following where each frontier's nodes are mapped to its original $T(n)$ values for $t = 4$:

$$f \mapsto T(n) = \{\ 1, 2, 2, 3, 3, 3\ \}$$
$$\bar{f} \mapsto T(n) = \{\ 4, 6, 7, 7, 8, 8, 10, 11\ \}$$

Eventually, all nodes in f will reduce to an empty set after the frontier is fully popped of all its nodes. When this occurs, the threshold doubles to 8. All nodes in $\bar{f}$ are re-evaluated

[3]Both comparators for `IDAS` and `IDS` are noted in Section 1.1.1.

according to Equation 6 and the next iteration occurs:

$$f \mapsto T(n) = \{\ 4, 6, 7, 7\ \}$$
$$\bar{f} \mapsto T(n) = \{\ 8, 8, 10, 11\ \}$$

Note that Equation 6 will always apply to nodes to decide which frontier they are to be allocated to. This is demonstrated in Listings 2 and 3.

Listing 2: Pushing a node in an iterative deepening search's frontier must perform $T(n)$ to see *which* frontier that node should be inserted to

```
func pushToFrontier(node: Node) {
    // If the node we want to push exceeds the threshold we currently have?
    if self.nodeThresholdComparatorBlock(node) > self.threshold {
        // Then we will push to our fallback - essentially add this node to the
        // end of the queue and we will pop it only if we are completely out of
        // nodes that are out of the threshold range
        self.fallbackFrontier.push(node)
    } else {
        // We can push this node to the concrete frontier as it is within the
        // threshold; essentially act as a normal frontier
        self.frontier.push(node)
    }
}
```

Listing 3: Popping a node in an iterative deepening search see if f is empty and if so double the threshold and reapply Equation 6 to $\bar{f}$

```
func popFrontier() -> Node? {
    // We have some nodes in our frontier?
    var poppedNode: Node?
    if !self.frontier.isEmpty {
        poppedNode = self.frontier.pop()
    }
    // Second if to see if we just made the entire frontier empty by popping
    // above in nodeToPop
    if self.frontier.isEmpty {
        // Since we are now out of nodes, we have to **iteratively** traverse
        // nodes in the fallback frontier instead (i.e., nodes which were too
        // much beyond the threshold before but are now okay to test.
```

```
            // Therefore append the fallback frontier's collection to the
            // collection of frontier and double the threshold to support new nodes
            // for iterative deepening.

            // Double the threshold with each iteration
            self.threshold *= 2

            // Get the nodes which are in the fallback that do not exceed threshold
            let nextIterationNodes = self.fallbackFrontier.collection.filter {
                self.nodeThresholdComparatorBlock($0) < self.threshold
            }

            // Add in all nodes in the fallback
            self.frontier.collection
                .appendContentsOf(nextIterationNodes)
            // Remove all nodes from the fallback part of this iteration
            // We do this by reassigning the collection to the same collection
            // filtered by not containing nodes in nextIterationNodes
            self.fallbackFrontier.collection =
                self.fallbackFrontier.collection.filter {
                    !nextIterationNodes.contains($0)
                }

            // Pop if were never able to pop before (i.e., popped node is still nil)
            if poppedNode == nil {
                poppedNode = self.frontier.pop()
            }
        }
        return poppedNode!
    }
```

2.3 Frontiers

There are four frontiers in the program, namely:

- a FIFO frontier used for `BFS` search,
- a LIFO frontier used for `DFS` and `IDS` searches,
- an evaluated frontier used for informed searches, and
- a random frontier used for BOGO search.

The evaluated frontier will use an evaluation function to decide the index in which to insert in its collection, as shown in Listing 4.

Listing 4: Implementation of inserting at the correct index for evaluated frontier; the collection ensures that nodes are inserted by their distance to goal

```
mutating func push(node: Node) {
    // Evaluate the distance of the state
    let distanceToGoal = self.distanceToGoal(node)
    // Find the index to insert at by finding that whose distance to goal result
    // is equal to or larger than the distanceToGoal calculated for this
    // particular state. If not found, then it's added to the end.
    // E.g.:
    //
    //   Insert 7 where f(n) for all nodes in collection are [ 3, 5, 8 ]
    //
    // In this example, indexToInsert will return:
    //
    //   3 >= 7 -> no
    //   5 >= 7 -> no
    //   8 >= 7 -> yes therefore index 2
    //
    // So insert at index 2, which will result in [ 3, 5, 7, 8 ]
    let indexToInsert = (self.collection.indexOf { node -> Bool in
        self.distanceToGoal(node) >= distanceToGoal
    } ?? self.collection.count)
    self.collection.insert(node, atIndex: indexToInsert)
}
```

2.4 Random State Generation

As seen in `RandomState.swift`, random n by m states were generated for extensive testing. It was ensured that these generated states are *solvable* using a State's `isSolvable` computed property.

To implement `isSolveable`, three theorms devised by (Gong, 2000) were used. The number of inversions, k, of a state is calculated by of a state, s by finding the sum of the

each tile, t, at index i that is higher than the *next* tile position, t_{i+1}.

$$k(s) = \sum_{i=0}^{n \times m} (\textbf{count}(K)) \text{ where } K = \{ t \mid t_i > t_{i+1} \text{ and } t_i \neq 0 \} \tag{8}$$

The implementation of Equation 8 is given in Listing 5 below.

Listing 5: Calculating the number of inversions of a given state

```
let inversions = self.sequence.enumerate().reduce(0) {
    (memo, value) -> Int in
        let startAt = value.index + 1
        let element = value.element
        let numInversionsForElement =
            self.sequence.suffixFrom(startAt).filter({ num in
                element > num && num != kEmptyTile
            }).count
        return memo + numInversionsForElement
    }
```

By using Equation 8, we can then utilise three theorms to see if a given state is solvable. Theorms 2 and 3 make use of the number of the row of the blank tile, i, from the bottom of the row, that is, for an n by m state $m - i$.

Theorm 1 "If n is odd, then every legal configuration corresponds to a sequence with an even number of inversions":

$$n \not\equiv k(s) \pmod 2 \quad \text{where } n \bmod 2 \neq 0 \tag{9}$$

Theorm 2 "If n is even, then every legal configuration with the blank tile in the i'th row where $m - i$ is even corresponds to a sequence with an even number of inversions":

$$n \equiv m - i \pmod 2 \equiv k(s) \pmod 2 \quad \text{where } n \bmod 2 = 0 \tag{10}$$

Theorm 3 "If n is even, then every legal configuration with the blank tile in the i'th row where $m - i$ is odd corresponds to a sequence with an odd number of inversions":

$$n \not\equiv m - i \pmod 2 \equiv k(s) \pmod 2 \quad \text{where } n \bmod 2 = 0 \tag{11}$$

Refer to `State.swift:140-204` for the three theorms used to calculate solvability and `IsStateSolvableTests.swift` for test coverage of these theorms.

2.5 Time Complexity Performance Testing

The codebase was ensured for optimal unit test coverage. For tests, refer to the the `test` directory. Each test generated a random state of a variant state size as demonstrated in Section 2.4, and the results were averaged over 10 different tests for each.

Results from aggregating these time performance tests are shown in Tables A.1 to A.4 and are graphically represented in Figures 2.1 to 2.5. All tests for iterative deepening searches had an initial threshold value of 4.

It can be determined that the 3×3 states generally take the longest for most tests. The maximum number of solvable states, n, as shown by Gong (2000), is shown in Equation 12:

$$n_{n \times m} = \frac{(n \times m)!}{2} \tag{12}$$

For the 3×3 tests, it can be shown that there are more legal configurations than the largest other dimensional state, 4×2 by applying Equation 12:

$$n_{3\times3} = \frac{(3 \times 3)!}{2} = 181440 > n_{2\times4} = \frac{(2 \times 4)!}{2} = 20160$$

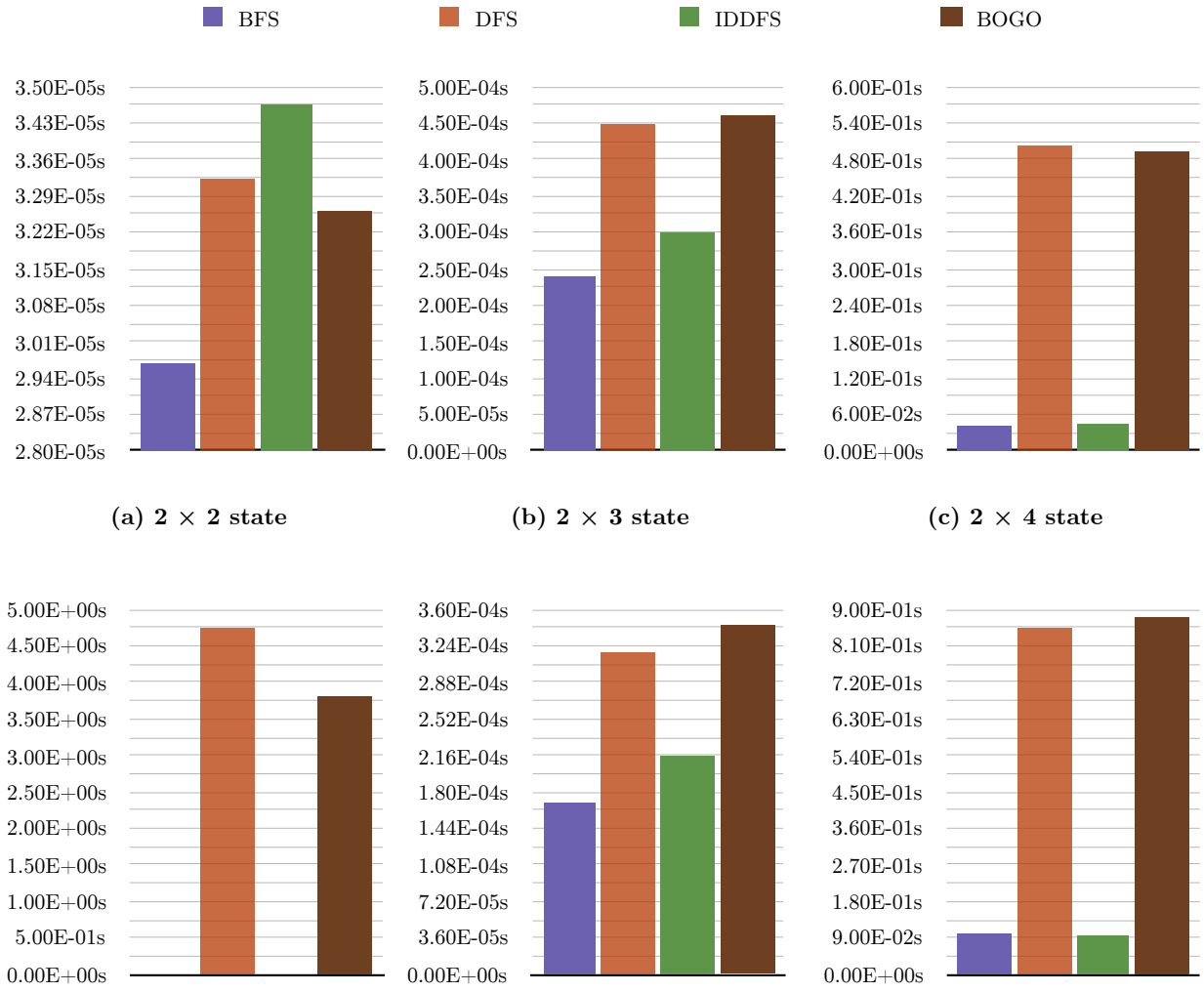
Uninformed searches are expectedly slower than their informed counterparts. BFS is shown to be the fastest here, consistently scoring speeds almost half of the other informed searches. Figure 2.1(c) and (f) do show, however that IDDFS is somewhat close when one dimension is *signficantly* larger than the other. Mackworth (2013) and Anderson (2012) demonstrate how both the time complexities of BFS and IDDFS share similar properties—that is, $O(b^m)$—which reinforces these similarities.

The non-optimal properites DFS are proved by these search tests, and Figure 2.1 highlights how poor DFS can perform on even the smallest of states. Where a solution is not on the initial vector, significant time is wasted—IDDFS, by combining the best of both DFS and BFS, emphasises the improvements of adding an increasing threshold to stop this problem.

When juxtaposed to the informed searched, immediatley it can be seen that the times improve significantly (Figure 2.2). By averaging across all heuristics, Figure 2.2 highlights that IDA* perofmrs signficiantly well in all n by m states, but *not* for n by n states—in fact

both A* and IDA* are outperformed by GBFS. Compare 2.2(a) and (d)—GBFS outperforms A* and IDA*, which could be due to implementation factors of heuristics calculations, as some heuristics in A* and IDA* significantly disadvantage them to GBFS (contrast the signficiantly high values of the Chebyshev Distance heuristic among A* and IDA* in Figures 2.3 and 2.4, and then the relatively *lower* GBFS average of Chebyshev in Figure 2.5.

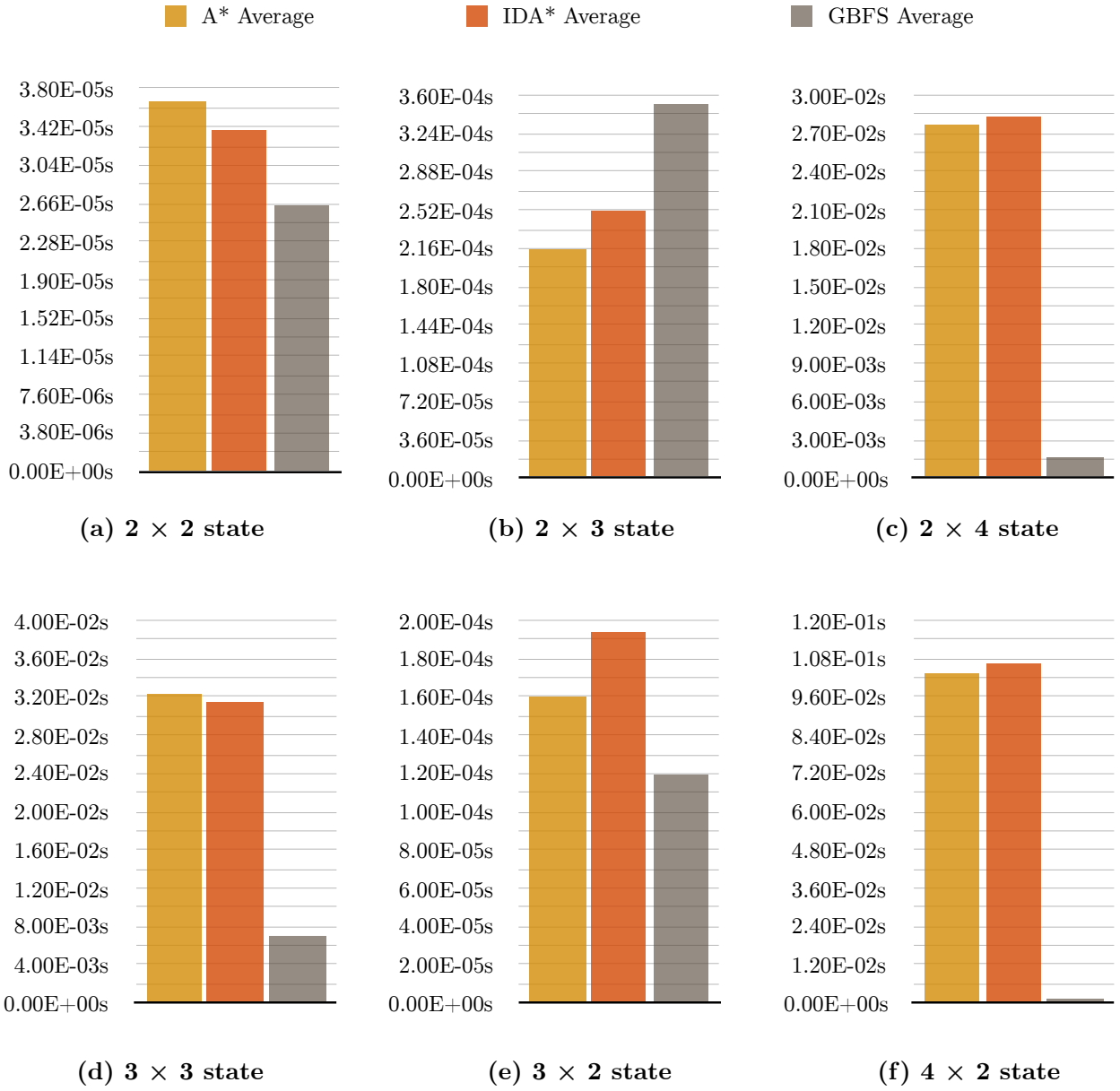
Indeed, Chebyshev is shown to be quite a poor heuristic consistently for A* and IDA* regardless of state, especially in Figures2.3(d) and Figures2.4(d) where it is a significant outlier. For most heuristics, Manhattan is generally the best performer, although when it comes to n by n matrices using A*, Euclidean comes very close—compare Figure 2.3(a) and (d) and Figure 2.4(a) and (d).



(a) 2 × 2 state **(b) 2 × 3 state** **(c) 2 × 4 state**

(d) 3 × 3 state **(e) 3 × 2 state** **(f) 4 × 2 state**

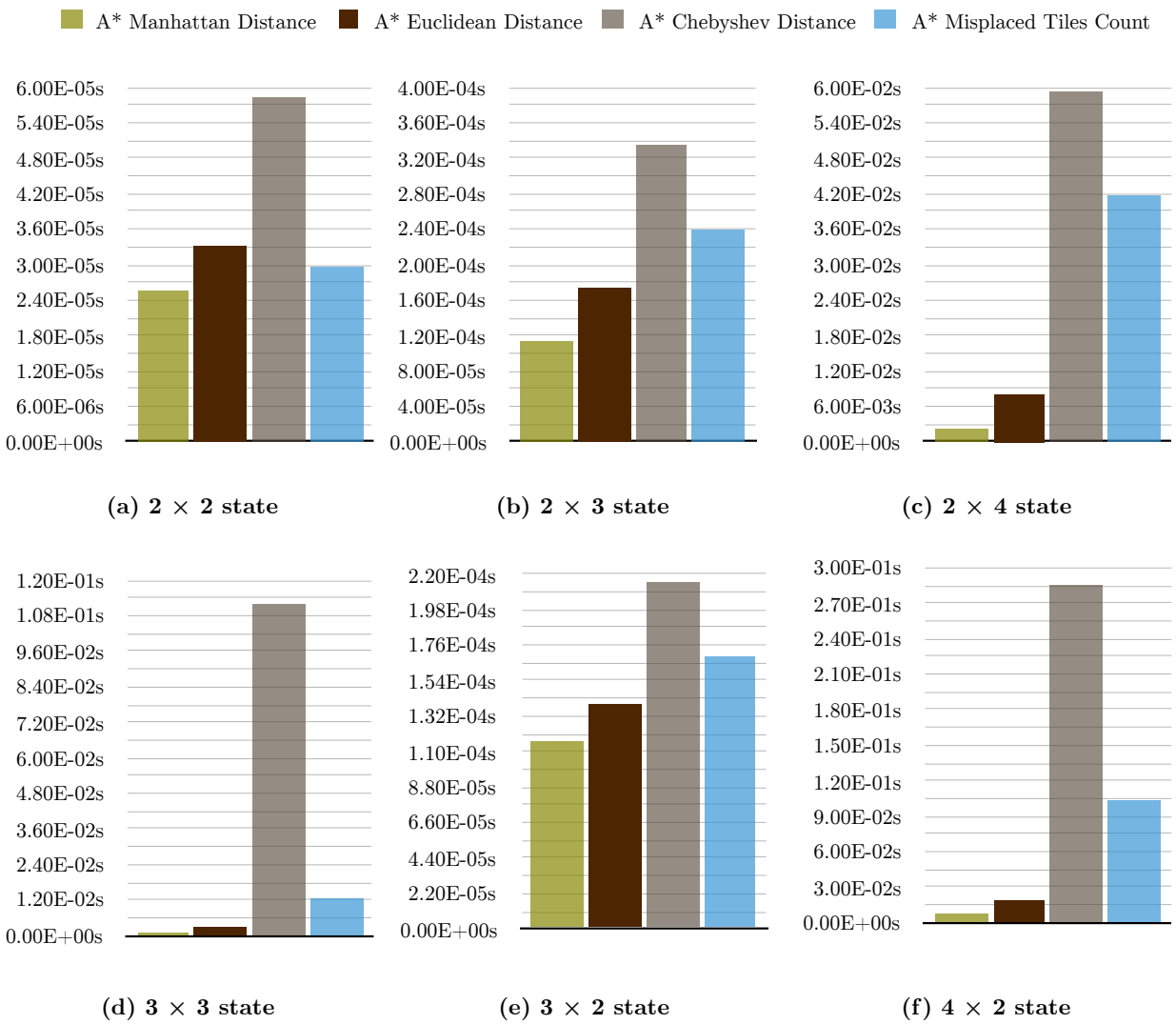
Figure 2.1: Aggregated performance test results for uninformed search methods



(a) 2 × 2 state **(b) 2 × 3 state** **(c) 2 × 4 state**

(d) 3 × 3 state **(e) 3 × 2 state** **(f) 4 × 2 state**

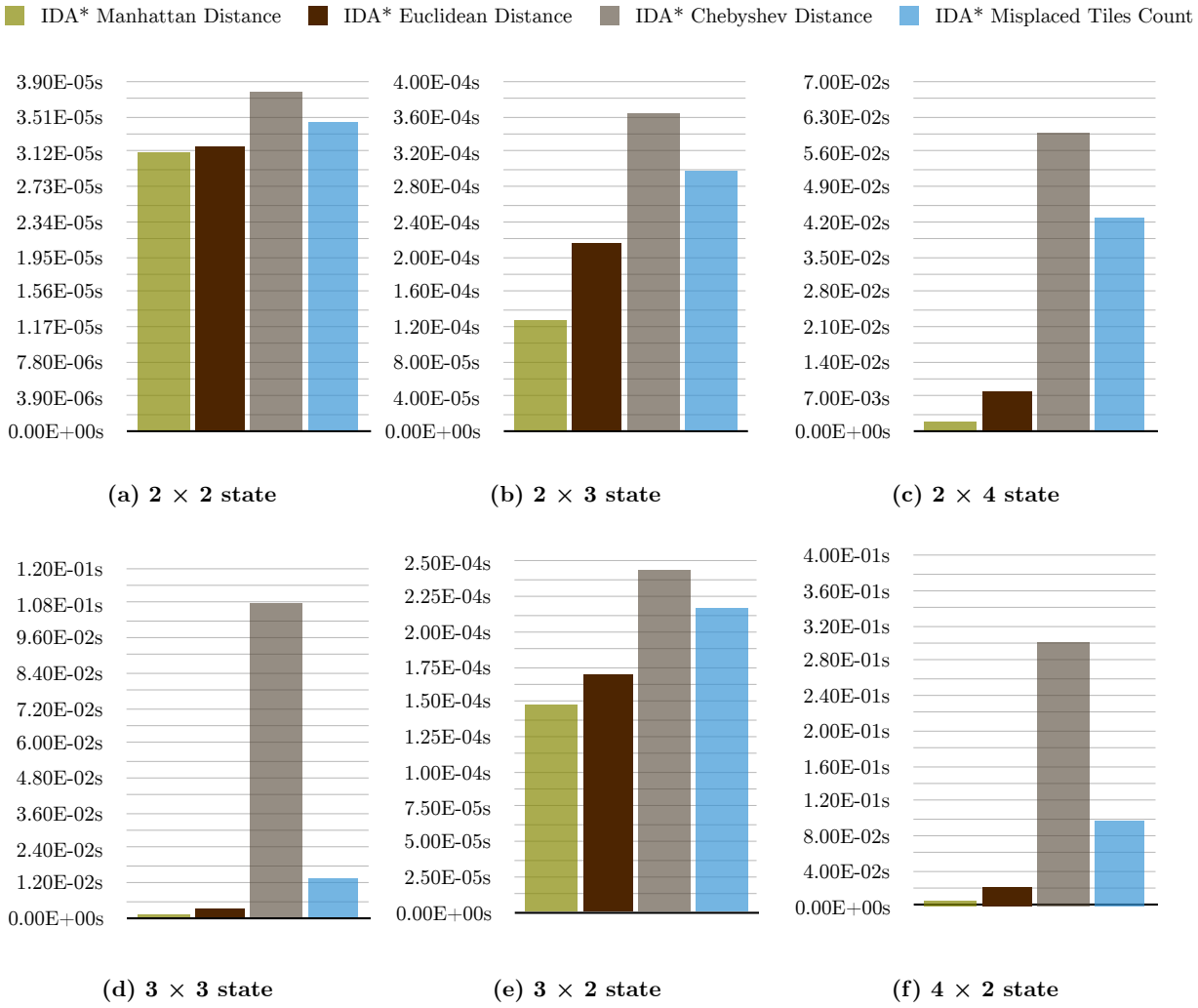
Figure 2.2: Aggregated performance test results for informed search methods averaged over all heuristics



(a) 2 × 2 state **(b) 2 × 3 state** **(c) 2 × 4 state**

(d) 3 × 3 state **(e) 3 × 2 state** **(f) 4 × 2 state**

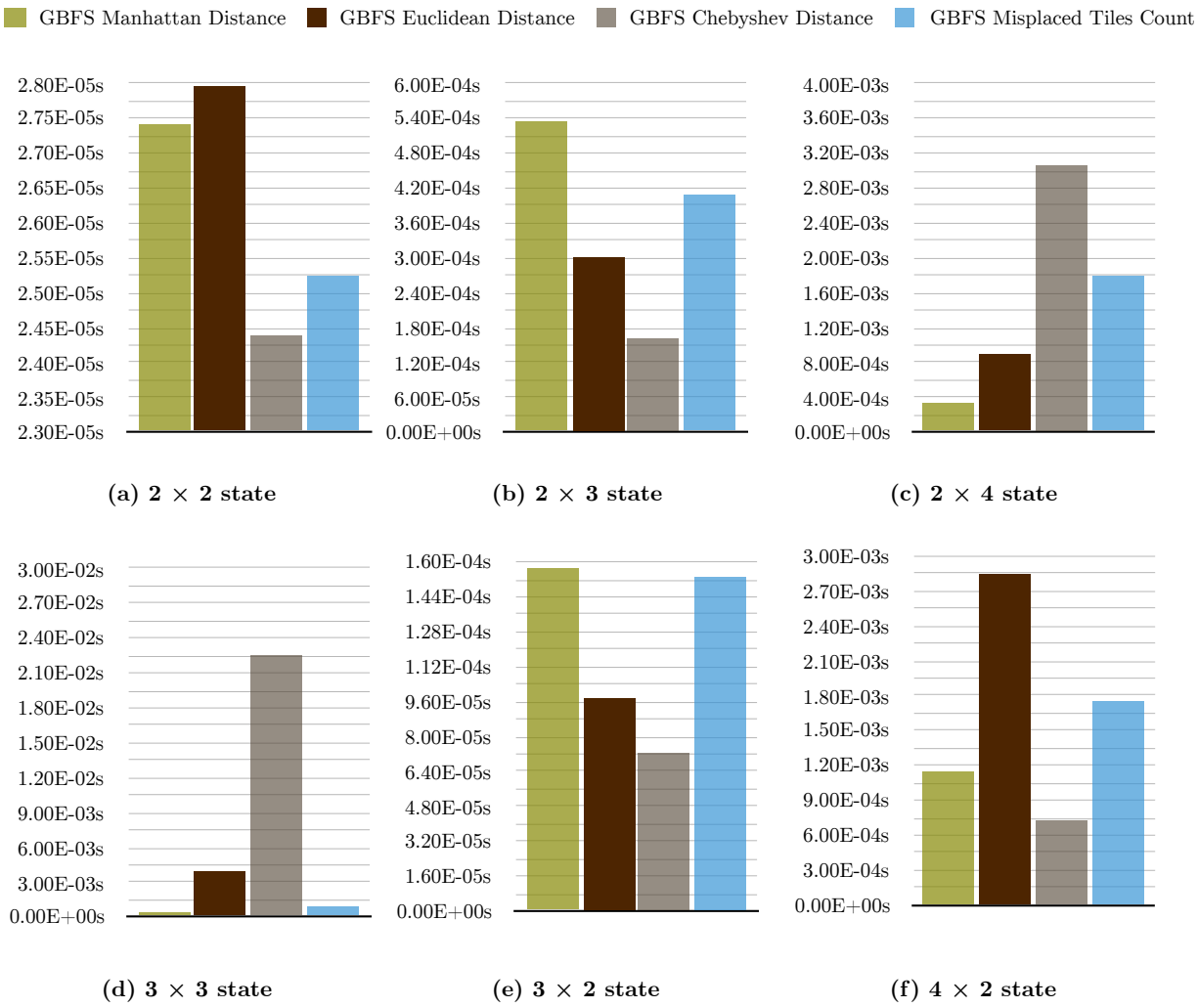
Figure 2.3: Aggregated performance test results for A* over all heuristics



(a) 2 × 2 state **(b) 2 × 3 state** **(c) 2 × 4 state**

(d) 3 × 3 state **(e) 3 × 2 state** **(f) 4 × 2 state**

Figure 2.4: Aggregated performance test results for IDA* over all heuristics



(a) 2 × 2 state **(b) 2 × 3 state** **(c) 2 × 4 state**

(d) 3 × 3 state **(e) 3 × 2 state** **(f) 4 × 2 state**

Figure 2.5: Aggregated performance test results for GBFS over all heuristics

References

Chuck Anderson. More Uninformed Search Methods. `http://www.cs.colostate.edu/~anderson/cs440/index.html/doku.php?id=notes:week2b`, 2012. [Online; accessed 4 April 2016].

Rebecca Bruce. Internal State Representation. `http://www.cs.unca.edu/~bruce/Spring98/csci373/ch4/lecture.html`, 1998. [Online; accessed 5 April 2016].

C.D. Cantrell. *Modern Mathematical Methods for Physicists and Engineers*. Cambridge University Press, 2000. ISBN 9780521598279. URL `https://books.google.com.au/books?id=cwEs5AkGp0MC`.

Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-00234-2_1.

Kevin Gong. Analysis of the Sixteen Puzzle. `http://kevingong.com/Math/SixteenPuzzle.html`, 2000. [Online; accessed 2 April 2016].

E.F. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Books on Mathematics. Dover Publications, 2012. ISBN 9780486136066. URL `https://books.google.com.au/books?id=cAzDAgAAQBAJ`.

Alan Mackworth. Iterative Deepening and IDA*. `http://www.cs.ubc.ca/~mack/CS322/lectures/2-Search6.pdf`, 2013. [Online; accessed 5 April 2016].

Red Blob Games. Implementation of A*. `http://www.redblobgames.com/pathfinding/a-star/implementation.html`, 2014. [Online; accessed 3 March 2016].

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, third edition, 2009. ISBN 0136042597.

A Results from Performance Tests

Aggregated results from all performance tests are shown below and are in seconds.

For each informed search, heuristics are labelled by the following: (1) *MD* for Manhattan Distance, (2) *ED* for Euclidean Distance, (3) *CD* for Chebyshev Distance and (4) *MTC* for Misplaced Tile Count.

Table A.1: Uninformed Search Time Performance Statistics

$n \times m$	**BFS**	**DFS**	**IDS**	**BOGO**
2x2	2.9650E-05	3.3200E-05	2.4850E-05	3.2600E-05
2x3	2.4030E-04	4.4810E-04	3.8110E-04	4.5990E-04
2x4	4.1709E-02	5.0072E-01	1.9233E+00	4.9248E-01
3x3	1.2729E-02	4.7581E+00	1.1117E+02	3.8281E+00
3x2	1.6840E-04	3.1785E-04	3.8650E-04	3.4315E-04
4x2	1.0290E-01	8.5634E-01	9.1438E-02	8.7820E-01

Table A.2: `AS` Time Performance Statistics

$n \times m$	**MD**	**ED**	**CD**	**MTC**
2x2	2.5650E-05	3.3350E-05	5.8150E-05	2.9650E-05
2x3	1.1195E-04	1.7165E-04	3.3525E-04	2.4030E-04
2x4	2.0448E-03	8.0126E-03	5.9261E-02	4.1709E-02
3x3	9.7440E-04	2.6800E-03	1.1216E-01	1.2729E-02
3x2	1.1685E-04	1.3850E-04	2.1490E-04	1.6840E-04
4x2	6.8561E-03	1.9199E-02	2.8589E-01	1.0290E-01

Table A.3: `IDAS` Time Performance Statistics

$n \times m$	**MD**	**ED**	**CD**	**MTC**
2x2	3.1100E-05	3.1750E-05	3.8000E-05	3.4650E-05
2x3	1.2935E-04	2.1590E-04	3.6475E-04	2.9725E-04
2x4	2.2764E-03	8.0005E-03	6.0009E-02	4.3006E-02
3x3	1.0713E-03	2.9046E-03	1.0836E-01	1.3130E-02
3x2	1.4795E-04	1.7025E-04	2.4370E-04	2.1605E-04
4x2	7.3250E-03	2.0477E-02	3.0235E-01	9.6859E-02

Table A.4: `GBFS` Time Performance Statistics

$n \times m$	**MD**	**ED**	**CD**	**MTC**
2x2	2.7400E-05	2.7950E-05	2.4400E-05	2.5250E-05
2x3	5.3350E-04	3.0150E-04	1.6315E-04	4.0960E-04
2x4	3.2020E-04	8.9710E-04	3.0497E-03	1.7795E-03
3x3	3.5855E-04	3.8554E-03	2.2598E-02	9.9805E-04
3x2	1.5655E-04	9.6950E-05	7.2350E-05	1.5365E-04
4x2	1.1546E-03	2.8541E-03	7.1830E-04	1.7506E-03