AI Search: Assignment 1

P. van Heerden Z. Mohamed

August 2018

Contents

1 Overview of implemented functionality

1.1 Algorithms

The following *algorithms* were implemented as part of the core assignment:

- Depth-first iterative deepening (implicit)
- A* (explicit, graph-search formulation)

- Iterative deepening A* (implicit)
- Dijkstra's Algorithm (as a special case of A*, via `NullHeuristic`)
- Bidirectional A* (explicit, graph-search formulation)

1.2 Heuristics

The following *heuristics* were implemented:

- Misplaced Tiles
- Manhattan Distance (with configurable admissibility)
- Euclidean Distance (with configurable admissibility)
- Linear Conflict

1.3 Domains

Support for the following *domains* have been implemented:

- N-puzzle (with configurable N)
- Path-finding on a $N \times N$ grid (with configurable N)

1.4 Additional functionality

- The class `GridRepl` provides a real-eval-print-loop style interface for configuring a path-finding problem, and displaying the final solution in an intuitive manner[1].

2 Architecture of the engine

2.1 Domains as a core data structure

When looking to solve problems with search, it is crucial the problem space is efficiently and elegantly represented within our framework. This must be done without making it difficult for agents to operate on graphs of nodes, not needing any more domain specific information out of the data structure. Our approach to this is the `Domain` interface and it's two direct implementations, `Board` and `Grid`. Agents can make use of domains to generate and traverse the state space in a way that is sensible for the given domain, without the agents themselves having to do much more than provide a new constructor to support this.

The `Board` domain is used to represent the state of a N-puzzle game. It has two concrete implementations, `ExplicitBoard` and `ImplicitBoard`. The explicit

[1] Future work could expand upon this functionality, perhaps as an API that can be exposed and then be consumed by visualization services.

variant is used by agents that require an explicit graph of the state space (through parent pointers). In our framework, the algorithms that require such a representation are A*, Dijkstra's Algorithm, and Bidirectional A*. The implicit variant is required by the IDA* and IDDFS agents.

The `Grid` domain is used to represent the state of a path-finding problem. Since the domain is different in the sense that the large grid remains static, having an explicit variant seems like a sub-optimal use of memory if the underlying grid was to be copied on each generation of a new neighbor. Instead, the authors' make a minor modification to the `ExplicitGrid` implementation, opting to maintain a reference to a single grid. After making this change, the `ExplicitGrid` implementation still complied with the behaviour required from agents using explicit data structures.

2.2 Agents as domain-agnostic algorithms

Agents are the algorithms driving the search. They have been designed as to be as general as possible, in essence only requiring a new constructor to support a new domain[2]. All agents implement the `Agent` interface, and the naming convention of agents makes the mapping from agent to algorithm clear. IDA*, for example, is implemented by the agent `IDAStarAgent`. After a new agent is initialized, the `Agent#solve()` method can be called, after which the agent will respond with an array of domain specific actions that must be taken to achieve the goal, if the goal can be achieved.

2.3 Heuristics as clairvoyant helpers

Heuristics drive informed search strategies. To allow the reuse of heuristics between agents, heuristics all implement the `Heuristic` interface and provide a single method, `Heuristic#getHeuristicCostEstimate(Domain)`. All the heuristics—as listed in Section 1.2—are implemented in this fashion. This also allows for easy comparison of heuristics, as they are usable by any agent (given that the heuristic makes sense for the domain the agent has been initialized for).

3 Experiments

3.1 Verify IDA* against Korf

The authors verified the results from [1] against our implementation of IDA*. In all the examples ran for comparison, both solutions provided the same initial estimate as well as the same length on the optimal paths. A comparison of nodes explored for the first ten examples can be found in Figure 1.

The authors' implementation was using the Manhattan distance heuristic, as was done by Korf during his investigation.

[2] In the source tree, agents operating on different domains have been kept in separate translation units for clarity sake.

	Nodes Explored		
Example	**Korf**	**The Authors**	**Author-Korf Ratio**
1	276361933	202882460	73%
2	15300442	11749617	77%
3	565994203	479872314	85%
4	62643179	156396225	**250%**
5	11020325	11111651	**100%**
6	32201660	34191997	**101%**
7	387138094	367429253	94%
8	39118937	56395496	**144%**
9	56395496	1632771	98%
10	198758703	189933071	95%

Figure 1: Comparison of nodes explored by Korf and the authors' implementation of IDA*.

	Nodes Explored			
Algorithm	**Misplaced Tiles**	**Manhattan Distance**	**Null Heuristic**	**Linear Conflict**
A*	143093	14016	466983	20982
IDA*	691251	11597	34758347	51068
Bidirectional A*	17414	1902	24881	2652
IDDFS	20092580	20092580	20092580	20092580

Figure 2: Comparison of nodes explored by different algorithms using different heuristics in the N-Puzzle domain.

3.2 Algorithm results

3.2.1 N-Puzzle Domain

The reference puzzle used in these results:

- Optimal solution length = 27

$$\begin{Bmatrix} 8 & 6 & 7 \\ 2 & 5 & 4 \\ 3 & 0 & 1 \end{Bmatrix} \tag{1}$$

3.2.2 Path-finding Domain

The reference problem used in these results:

- Optimal path = 62
- Grid size = 40×40

	Nodes Explored			
Algorithm	Manhattan Distance	Manhattan Distance Inadmissible	Null Heuristic	Euclidean Distance
A*	1381	306	3090	1570
IDA*	N/A	N/A	N/A	N/A
Bidirectional A*	7275	1130	22772	14096
IDDFS	N/A	N/A	N/A	N/A

Figure 3: Comparison of nodes explored by different algorithms using different heuristics in the path-finding domain.

- Player position = $\{0, 20\}$
- Goal position = $\{39, 5\}$
- Horizontal obstacle positions = $\{10, 2\} \longrightarrow \{12, 30\}$
- Vertical obstacle positions = $\{0, 27\} \longrightarrow \{10, 30\}$

3.3 Remarks

3.3.1 N/A results

In the path-finding experiment in Figure 3, results for both iterative deepening algorithms have been omitted. This is due to the fact that the execution time for these algorithms may run for several hours or even days.

3.3.2 Linear conflict heuristic

The linear conflict heuristic is quite complex by nature. Once a conflict is identified we have to add the minimum number moves it would take to fix this conflict. This makes the algorithm complex, due to time constraints for every conflict we simply incremented our cost by two. In some cases this makes the heuristic inadmissible which is why it is outperformed by Manhattan distance when in theory the number of nodes explored by linear conflict is a subset of those explored by Manhattan distance.

3.3.3 Move ordering

With regards to our Korf experiment, in every example more or less nodes were explored by us than what was explored by Korf. Since there is not much information about how Korf performed his experiment, with what little information given to us we have concluded that this was due to *move ordering*. The order of which the legal moves are explored by us most likely differs to the way it was explored by Korf resulting in the last depth of IDA* exploring a different number of nodes.

3.3.4 Manhattan inadmissible heuristic

For the inadmissible heuristic in our path-finding results, we simply multiplied our Manhattan distance result by epsilon which was 2 in our results.

3.4 Impact of bidirectional search on node exploration

On the path-finding reference problem using the Euclidean heuristic, as per Figure 3, Bidirectional A* explores roughly 5 times the number of nodes when compared to standard A*.

On the N-puzzle reference problem using the Manhattan Distance heuristic, as per Figure 2, standard A* now performs much worse, exploring roughly 7 times the number of nodes when compared to Bidirectional A*.

Averaging over the results obtained in the tables mentioned above, the ratios hold even when the heuristic is changed. This leads us to believe that the nature of the two state spaces is likely a cause for the stark contrast observed. In the case of the path-finder problem, there is effectively one path out of the room in which the starting position has been placed. Once A* breaks out the room, it is a fairly straight forward descent to the goal. The Bidirectional search, however, wastes time in the backward portion of the search as it must find a way around the walls from the goal position, whilst the forward search is doing the work the standard A* algorithm does in any case.

3.5 Effects of inadmissible heuristics for path-finding

From the reference path-finding problem results we can see a significant decrease in the number of nodes explored when using our inadmissible heuristic. For the inadmissible heuristic we multiplied the Manhattan distance result by 2. For A* we can see a decrease from 1381 nodes explored to 306 and similarly for Bidirectional A* we can see a decrease from 7275 nodes explored to 1130. Even though our heuristic is inadmissible and may overestimate the actual cost by a factor of 2 it still managed to find the optimal path length of 62. Since this is basically making our Manhattan distance heuristic more greedy the higher the factor we multiply it by, we may see less optimal solutions being found in more complex problems.

References

[1] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," 1985.