Pattern Database Heuristics for Greedy Search

Carmen St. Jean

Background

Greedy search and A* search are search algorithms whose efficiency depends on the quality of their heuristic function, which is the lower bound estimate of the remaining solution cost. These algorithms can be applied to finding solutions to combinatorial problems, such as the 15-puzzle. Pattern databases can be used to improve the efficiency of these algorithms by storing precomputed heuristic values (Culberson and Schaeffer 1998).

A pattern database is built by enumerating all configurations of a partial solution. That partial solution is made by selecting n elements of the goal permutation to form a pattern, while the other elements are abstracted. At any point in the search, the minimal number of actions to place those n elements in their goal positions can be looked up in the pattern database.

The Problem

For the 15-puzzle, there are many possible ways to design a pattern database. The tiles which form the pattern database – i.e., kept from abstraction – are referred to as the refined tiles. One method for building the pattern database looks at a specific start configuration and refines the tiles with the largest Manhattan distances to their goal positions. We refer to this abstraction as the "customized abstraction" (Holte, Grajkowski, and Tanner 2005). Another method, called fringe, is to abstract away the tiles that are not part of the bottom-most row or right-most column in the goal configuration.

In most cases, the customized abstraction makes for a stronger heuristic than the fringe abstraction, since it takes the initial configuration into account. Although A* can efficiently use either form of abstraction, greedy search may actually perform better with the fringe pattern database, even though it is a weaker heuristic. We investigated this claim by comparing these two abstractions in their use with A* and greedy search.

Approach

Pattern databases for the standard 4-by-4 sliding tile puzzle proved to be too computationally intensive to produce when seven tiles were refined, which is the number of tiles required for the fringe pattern database. Therefore, we approached the problem using 3-by-4 versions of the sliding tile puzzle, where the tile labels span from "1" to "11," as seen in Figure 1a.

	1	2	3
4	5	6	7
8	9	10	11

2	6	3	4
10	9	5	
8	7	1	11

Figure 1: a 3-by-4 version of the sliding tile puzzle in (a) its goal configuration and (b) an example of a random start configuration.

Pattern Databases

A pattern found within a pattern database is a partial specification of a search state where some of the elements have been abstracted. The target pattern is a partial specification of the goal state. The pattern database is obtained by applying all permutations to the target pattern. Every pattern knows its exact solution cost for the target pattern. This solution cost is an admissible heuristic. The pattern database is produced before search begins.

During search, a state finds its heuristic value by looking up the solution cost for the pattern obtained from abstracting that state. If multiple, disjoint pattern databases are used, then the state must be abstracted separately according to each pattern database's abstraction. The heuristic value is then the sum of the solution costs returned from each disjoint pattern database.

Abstractions

As mentioned, there are numerous abstraction methods that are available when building a pattern database. To be precise, there are 462 different abstractions possible for the 3-by-4 puzzle when six tiles are refined and the others are abstracted, though some of these may be symmetries with one another. We looked at the aforementioned fringe and customized abstractions, described in further detail below, as well as a small sampling of other abstractions for comparison.

With greedy and A* being different algorithms, we suspected that abstractions which work well for one may not work well for another. In fact, we anticipated very few abstractions to work either well for both or badly for both. Since greedy sorts its queue only on heuristic value, it will always prefer nodes with the heuristic value of zero. A*, on the other hand, sorts its queue on cost and heuristic value, so it may not always jump for any node with heuristic value of zero.

Some abstractions may return a heuristic value of zero only when they are close to the actual goal, while others may return zero when the actual goal is quite far away. In the latter case, zero will be misleading, since future nodes are likely to have non-zero heuristic values. A* may handle this well because it might not explore such nodes in the first place, while it will cause greedy to expand more nodes before reaching the goal, thereby producing a longer solution. Thus, it would make sense for each algorithm to perform differently on the various abstractions.

Fringe

Creation of the fringe pattern database is independent of the start configuration of a problem since all sliding tile puzzles have the same goal configuration. The puzzle is placed into its goal configuration and the right-most column and bottom-most row are refined, while all other tiles are abstracted, as in Figure 2a. This is the target pattern used to produce the pattern database.

Figure 2b shows the fringe abstraction applied to the state shown in Figure 1b, as would be done during search. The cost associated with the resulting pattern would be the heuristic value of the original state.

	A	A	3
A	A	A	7
8	9	10	11

A	A	3	A
10	9	A	
8	7	A	11

Figure 2: (a) the fringe abstraction and (b) the pattern resulting from applying the fringe abstraction to the state in Figure 1b.

We hypothesized that greedy would do well with the fringe pattern database heuristic. The reasoning behind this is that a smaller 2-by-3 puzzle remains when the tiles of the fringe are in their goal location. This smaller puzzle is trivial to solve and, more importantly, can be solved without disturbing the tiles on the fringe. Since the subsequent actions will not move the fringe tiles, the pattern database will return a heuristic value of zero for those actions. Therefore, a heuristic value of zero from the pattern database will be followed by more heuristic values of zero, indicating that the overall goal is indeed close.

Customized

Customized pattern databases are created specially for each individual start configuration. First, the Manhattan distance for each tile to its goal position must be computed. The six tiles which are furthest from their goals are selected to be the refined tiles of the pattern. Since the fringe pattern database keeps six tiles refined, an equivalent number of tiles should be chosen for the customized pattern database. Next, the goal configuration is abstracted according to those six tiles. This gives us the target pattern from which the whole pattern database is built. This process can be seen in Figure 3.

2_2	6_2	3_1	4_4
10_3	9_1	5_1	
8_0	7_3	1_3	11_0

2	6	A	4
10	A	A	
A	7	1	A

	1	2	A
4	A	6	7
A	A	10	A

Figure 3: (a) the state shown in Figure 1b with subscripts denoting the Manhattan distance for each tile, (b) the closest tiles of that state abstracted, and (c) the same abstraction applied to the goal configuration.

We hypothesized that greedy would not do well with the customized abstraction because it cannot always guarantee that a heuristic value of zero will soon lead to the goal. The refined tiles may have their goal positions scattered around the board; they could even be isolated from each other or the blank tile. Placing these tiles in their goal positions could be placing the abstracted tiles far from their goal positions, requiring further actions. Worse, actions taken to move the abstracted tiles into their goal position could move the refined tiles away from their goal position. When that occurs, a heuristic value greater than zero will be returned from the pattern database. Therefore, a heuristic value of zero from the pattern database could be misleading, because it could very soon lead to non-zero heuristic values.

Disjoint Fringe

	A	A	A
A	A	A	A
8	9	10	11

	A	2	3
A	A	6	7
A	A	A	A

Figure 4: the two abstractions that together form the disjoint fringe abstraction.

As with regular fringe, the creation of the disjoint fringe pattern databases is independent of individual start configurations. The disjoint fringe abstraction is made of two abstractions: (1) the bottom-most row and (2) a 2-by-2 square in the top-right corner of the puzzle.

These are shown in Figure 4 in dull blue and cyan, respectively. Figure 5 represents the abstractions applied to the random start configuration shown in Figure 1b.

A	A	A	A
10	9	A	
8	A	A	11

2	6	3	A
A	A	A	
A	7	A	A

Figure 5: the disjoint fringe abstractions applied to the state in Figure 1b. The heuristic value for this state would be $h = cost + cost$.

Disjoint Customized

The disjoint customized pattern database is formed of two pattern databases of size four each, so that it has the same number and sizes of pattern databases as the disjoint fringe. As with the regular customized pattern database, this pattern database is constructed for a specific start configuration by keeping refining the tiles which are furthest from their goal positions refined, with distances measured in Manhattan distance.

A	A	A	4
10	A	A	
A	7	1	A

2	6	3	A
A	A	5	
A	A	A	A

Figure 6: the disjoint customized abstractions applied to the state in Figure 1b, *before* applying the abstractions to the goal configuration to produce the pattern database.

The two abstractions which form the disjoint customized pattern database are (1) the four tiles which are furthest from their goals and (2) the next four tiles which are furthest from their goals. Figure 6 shows these abstractions in magenta and orange, respectively, for the start configuration shown in Figure 1b. These abstractions must be applied to the goal state, as in Figure 7, to produce the pattern databases.

	1	A	A
4	A	A	7
A	A	10	A

	A	2	3
A	5	6	A
A	A	A	A

Figure 7: the abstractions obtained in Figure 6 applied to the goal to obtain the target patterns for building the pattern databases. The heuristic value for the original state in Figure 1b would be $h = cost + cost$.

Assorted

We produced some additional abstractions that we anticipated greedy search having some difficulty with. The idea was to produce abstractions which did not have key characteristics of the fringe abstraction.

As mentioned earlier, an advantage to the fringe pattern is that solving the fringe tiles leaves behind a smaller puzzle that is easy to solve. Thus, we strove designed patterns that do not have smaller 2-by-2 or 2-by-3 abstracted regions, seen in Figures 8a and 8b.

The fringe also features a contiguous pattern – i.e, the refined tiles are connected in one group. Figure 8b features a non-contiguous pattern and Figure 8c is an even more extreme example of discontiguity – a checkerboard pattern with every tile isolated from other tiles of its type.

	A	A	A
4	5	6	A
8	9	10	A

	1	2	A
A	A	A	A
8	9	10	11

	1	A	3
4	A	6	A
A	9	A	11

Figure 8: additional abstractions, referred to as A, B, and C (from left to right).

Evaluation

We used each pattern database as the heuristic for solving 100 different, random start configurations for the 3-by-4 sliding tile puzzle with both A* and greedy search. The fringe pattern database needed to be created only once for all problems, while the customized pattern database had to be generated according to each start configuration. We included A* in our investigation to illustrate the strength of either pattern database heuristic, while greedy was included to explore the proposed problem.

For each search conducted, we recorded the number of nodes that were expanded in order to solve the puzzle. The average number of nodes expanded was then calculated for each combination of algorithm (i.e., greedy or A*) and heuristic (i.e., fringe or customized pattern database) across all 100 problems. We imposed a time limit of ten minutes and recorded the number of times each algorithm failed to find a solution within that limit. Each search was also able to use up to 7500 megabytes of memory.

Results

The results of our evaluations are summarized in Table 1. This table has been sorted by the number of times (out of 100 total tests) that A* failed to find a solution within ten minutes, which describes the strength of the pattern database heuristic for A*. The more often a pattern database leads to a solution for A* within the time limit, the stronger that heuristic is. It is useful to next look the average number of nodes expanded when A* did manage to find a solution because stronger heuristics will lead to less expansion of nodes.

Abstraction	Greedy Nodes Exp.	A* Nodes Exp.	Num. A* Failures	Greedy Ranking
B	1,086,512	3,757,149	1	4
Fringe	2,403,225	3,450,926	4	6
Customized	2,778,947	5,593,762	10	7
A	755,125	5,528,595	22	3
C	2,021,907	6,259,867	26	5
Disjoint Fringe	*44,171*	*218,816*	67	1
Disjoint Customized	196,073	333,928	67	2

Table 1: average number of nodes expanded, sorted by the number of times A* failed to find a solution.

Greedy search always found a solution - albeit a sub-optimal one. It required, in general, fewer expansions than A* did. Sorting by A*'s number of failures seemingly jumbles the average number of nodes expanded for greedy. To help illustrate this, the right-most column of Table 1 shows the ranking for average number of nodes expanded for greedy search. From this column, it is easy to see that the order for greedy search does not correspond to A*'s rankings at all. This verifies that these two search algorithms really are quite different in this domain and do not share many preferences for abstractions, as we expected.

In italics are the abstractions which had the lowest average number of expansions for each algorithm. Both performed best with the disjoint fringe pattern databases. Perhaps this is because the disjoint pattern databases included eight refined tiles in total, while the regular pattern databases included only six refined tiles in total. Larger pattern databases, though more difficult to compute and store, make for more powerful heuristics. Still, the disjoint pattern databases are not a total success story for A*. They may have led to the fewest expansions of nodes on average, but they also had extraordinarily high failure rates for A*.

What is worth mentioning more is that both algorithms preferred the disjoint fringe. We had expected this for greedy, but we had expected A* to prefer the disjoint customized abstraction. Still, the ratio of average number of nodes expanded for disjoint fringe to average number of nodes expanded for disjoint special is much smaller for greedy (0.225) than for A* (0.655). This may mean that greedy search benefited from the disjoint fringe abstractions more than A*.

We have underlined the averages for the regular (i.e., non-disjoint) pattern databases that did best after both of the disjoint pattern databases. As somewhat of a shock, the fringe pattern database gave the best non-disjoint result for A* in terms of nodes expanded, while the A abstraction was best for greedy. The overwhelming success of the A abstraction was surprising since we had designed it in the hope that it would cause greedy to do poorly.

We had hoped to show that the fringe abstraction was a best overall choice for greedy search, which we did not find as far as non-disjoint options are concerned. Fringe came in sixth place out of seven and customized came in dead last for greedy search. From these results, it would seem that other abstractions are better choices than either of these when selecting a heuristic for greedy search. However, the fringe abstraction was better than the customized for greedy, which confirmed at least part of what we hoped to show.

Comprehensive Analysis

Seeing these unexpected results led us to wonder exactly what distinguishes good abstractions from bad ones for either search algorithm. With an eleven tile puzzle and a pattern database where six tiles are left refined, there are exactly 462 possible abstraction choices. We took a sample of 120 random patterns from these 462 possibilities – approximately a fourth – and used these pattern databases to solve the 100 random start configurations. The same 7500 megabyte memory limit and ten minute time limit were applied as before. We recorded the average number of nodes expanded for greedy and A* search.

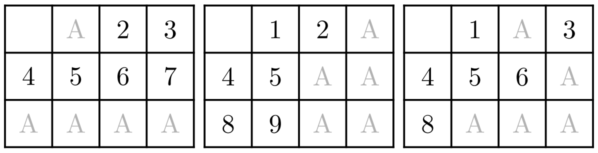

Figure 9: abstractions which did not work well for greedy.

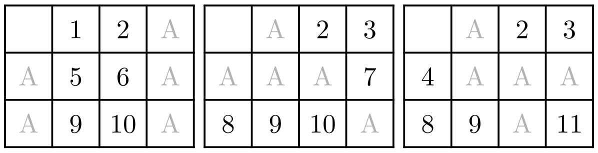

Figure 10: abstractions which worked well for greedy.

We then ranked the different abstractions twice, once for A* and again for greedy search. For both rankings, we first ranked by number of failures, preferring smaller failure counts. Next, we ranked by average number of nodes expanded, preferring smaller averages. We then looked at the best and worst abstractions from these lists to attempt to describe distinguishing properties.

Figure 9 shows a few of abstractions which greedy did poorly with. These abstractions and other badly ranked abstractions tended to feature a single isolated abstracted tile. Another common trend among these was complete abstraction of either (a) the tiles of the bottom-most row, (b) the tiles of the right-most column, or (c) several contiguous tiles in the bottom right corner.

Therefore, it would lead one to conclude that greedy does not like an abstraction where the blank tile is disconnected from a large contiguous group of abstracted tiles. However, Figure 10 seems to contradict these conclusions. These abstractions were good for greedy, but some of them featured the blank separated from the pattern and multiple groupings of refined tiles. They did at least spare some tiles of the bottom row or right column from abstraction, but not all tiles in these regions. Therefore, it is difficult to draw conclusions from these abstractions which worked well for greedy search.

	A	2	3
4	5	6	7
A	A	A	A

	1	A	3
4	5	6	A
8	A	A	A

	A	A	3
4	5	6	7
8	A	A	A

Figure 11: abstractions which worked badly for A*.

The abstractions which worked badly for A*, some of which are shown in Figure 11, are a bit easier to analyze. When some or all of the bottom row's tiles are abstracted, A* does poorly. A* also did not like when single tiles or pairs of tiles were abstracted by themselves, separated from the other abstracted tiles. This agrees with our results, which showed that A* does horribly with the C abstraction, where every abstracted tile is isolated from the other abstracted tiles in a checkerboard pattern.

	1	A	A
A	A	A	7
8	9	10	11

	1	2	A
A	A	A	A
8	9	10	11

	A	2	A
A	A	A	7
8	9	10	11

Figure 12: abstractions which worked well for A*.

Abstractions which worked well for A*, shown in Figure 12, agree with these conclusions. All of them spared the bottom row from abstraction. Furthermore, these abstractions feature at least four connected refined tiles. All or most of the abstracted tiles in these patterns are contiguous. Also, the blank tile was connected to the abstract region for all of these patterns. Thus, A* seems to prefer pattern database abstractions which include the bottom row and abstract large, contiguous regions.

Conclusion

We had hoped to show that, for the 3-by-4 sliding tile domain, greedy search would perform best with a fringe pattern database, while A* would perform best with a customized pattern database. Greedy search does indeed perform better with fringe pattern database abstractions than customized pattern database abstractions. However, A* also performs better with fringe abstractions, which was unexpected.

Both of these algorithms worked best with the disjoint fringe and disjoint customized pattern databases. Disjoint pattern databases aside, the fringe pattern database is definitely not the best of available options for greedy search. It is still difficult to characterize what greedy likes in an abstraction. A* does best when the bottom-most row is included in the pattern and the abstracted tiles form one large, contiguous region.

Future Work

It is unclear whether or not these conclusions apply to pattern databases of different sizes, such as the standard 4-by-4. More investigation is necessary to understand the outcome of different pattern database abstractions for greedy search. Future work might involve solving the 3-by-4 sliding tile puzzle with all 462 abstractions or working with the 4-by-4 domain.

References

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical heuristic search revisited. In Zucker, J.-D., and Saitta, L., eds., *SARA*, volume 3607 of *Lecture Notes in Computer Science*, 121–133. Springer.