A* intro – AI

5 Oct 2018

Although using the Manhattan distance of puzzles got a fast solution for slider puzzles, the solution was usually not optimal. The reason for this was that the basis for selecting the next puzzle to try was the one that appeared to be closest to the goal. However, this did not account for how many steps it took to get to that puzzle in the first place.

Therefore, for finding an optimal solution (at the expense of perhaps having to search a lot more), the strategy is to make an estimate to the goal (h) and add that to the distance already covered (g) to get a total estimate (f). The letters used, f, g, h are industry standard when talking about the algorithm that incorporates them, A* (pronounced "A Star").

There are some restrictions on what h may be. Specifically, there are two terms that are applicable.

- h is admissible if h never overestimates the remaining distance. That's why the space is not counted in computing the Manhattan distance.

- h is consistent if h obeys the triangle inequality. Conceptually, this means that the estimate for the remaining distance must take into account all the possible ways to get to the goal. It's not consistent to say something like, "My estimate for this puzzle is 12" and then have a back way which would actually reduce the estimate. The back way must already be accounted for in the estimate.

Consistency implies admissibility, but not the other way around. It can be shown that if h is consistent (and hence admissible), then if A* terminates at all, the solution it finds will be optimal.

A* is similar to BFS and DFS, but it is quite finicky in the details. One of the things is that that the frontier `parseMe` is now a priority queue and it is always called the `openSet` (even though it is not a set). The analogue of `dctSeen` is called the `closedSet` (even though it will still be a dictionary). However, it is more than just a name change. Under BFS, we put items into `dctSeen` as soon as we encounter them (and concomitantly put them into `parseMe`). With A*, items go into the `closedSet` only when they are taken out of the `openSet` (and not when they are put in). This guarantees (when h is consistent) that items in the `closedSet` are optimal, meaning that their level is correct. Pseudo code for A* follows.

A* pseudo code

```
def solve(root, goal):

    if already solved  ⇒  we're done
    if not solvable    ⇒  we're done

    Create openSet with root
    Create empty closedSet                            # pzl: level

    while openSet:
        find pzl with lowest estimate in openSet      # (eg. by sorting)
        extract pzl from openSet                      # for processing

        if pzl already in closedSet ⇒ continue        # Already processed
        add pzl to closedSet                          # else we know it's optimal

        for each nbr in the neighbors of pzl:
            if nbr is the goal  ⇒  we're done
            if nbr already in closedSet ⇒ continue    # More efficient
            new estimate = (pzl's level+1) + $h$(nbr,goal)   # nbr's level + rest estimate
            add nbr to openSet                        # add to queue
```

Notes:
If you don't have a special library you import, the normal way to deal with sorting by a parameter is that the list in question will be filled with a tuple where the first element of the tuple is the parameter by which the sort happens followed by any remaining information. If you need ties to be broken (ie. the first parameter is not sufficient), then the 2nd element of the tuple will be used to subsort those parts of the list where the first element of the tuple is the same.

For example, if you wanted to sort a list of words based on their length, you could do:

```
myWordTpls = [(len(w), w) for w in listOfWords]  # prep for sorting
myWordTpls.sort()
myLen, shortestWord = myWordTpls.pop(0)     # remove shortest word
```

h is the Manhattan distance, the `nbr`'s level is g, and the new estimate is f.