B551 Homework 1

Assigned: Sep. 1, 2011
Due: Sep. 15, 2011

1 Directions

Please read all the instructions for the homework carefully; the assignment is not difficult, but should be completed with attention to detail. You may wish to skim this document for the general structure of the homework and then re-read it closely for important details.

In this homework, you will be asked to write two functions to complete a search routine for a word puzzle.

- The first function focuses on an aspect of basic search (the creation of successor states in the search tree), and you should be able to write it after the lectures on basic search. (The assignment framework uses a type of heuristic search called A* search, but you should not need to understand it to complete the first problem.)
- The second function focuses on heuristics and can be written after the heuristic-search lecture.

You will have two weeks to complete the assignment, including a week after the heuristic-search lecture, so you should have plenty of time to finish the homework by the due date. (If you wish to try your hand at the second part of the assignment early, the relevant sections of the book are listed on the course schedule.)

Section 2 of this document describes the word game we are playing. Section 3 describes the programming framework and the functions you must implement.

1.1 Submission

Complete your programming work in the *wordgame.py* file and hand it in electronically on OnCourse. (Do not alter the other Python files or submit other files.) Don't forget to complete the written assignment available on the class webpage!

2 The Game

For this homework, we are interested in a single-player word game that appears in many magazines, newspapers, books of puzzles, and so on. The object of this game is to start with one English word, for instance, "mare", and end up with another English word, for instance, "colt". To move between these words you may change one letter at a time, but every time you change a letter it must result in a real English word. For instance, to solve the "mare" to "colt" puzzle, we would generate the following words:

- mare
- care
- core
- cole
- colt

We could not start this solution by changing, say, "mare" into "mave", because "mave" is not an English word.

3 Programming

3.1 Setup

1. Download the file *hw1.zip* from OnCourse. When you unzip this file you will have three Python files and one text file containing a spelling dictionary, one word per line (*words.txt*).

    - *fibheap.py* defines a useful data structure for the search problem (a special kind of heap for keeping the search fringe up-to-date). You don't have to understand this code to complete the assignment, but it may be of interest to you if you like data structures and algorithms.
    - *astar_fibheap.py* defines a class that operates a generic A* search procedure – that is, it is not designed for solving any particular problem, but just to search a tree of A* states. You should read through the code and gain a reasonably good grasp of how this works, once A* search is covered in class – in-depth knowledge of this code is not required to complete the assignment, but seeing A* code in action is useful to your understanding of the class material.
    - *wordgame.py* is our primary interest. It defines an inherited A* search class that operates on the word game described above. The file also contains some infrastructure to support the search and operate on user input from the command

line. However, two crucial functions of the A* class are missing and the search will not run with the file as-is. You will (eventually) need to understand much of the code in this file and fix the search by implementing the missing parts of the search definition.

2. You may run the program with this command from the Windows command line (assuming the Python executable is on your PATH):
python wordgame.py WORD1 WORD2

   Or, on any UNIX/Mac system's command line:
./wordgame.py WORD1 WORD2

   However, the program will not run successfully until you complete the first programming problem.

3. If you wish, you may generate documentation for these files by running the following command:
pydoc -w fibheap astar_fibheap wordgame

   This will generate an HTML file for each Python source file, describing the corresponding file's functions and classes in overview.

3.2 Programming Problems

REMINDER: Submit only your modified wordgame.py file to OnCourse. Do not modify or submit other Python files.

1. **Successor function for the word game**

   The successor function (*WordAStar.successors()*) for the puzzle is currently unimplemented. Considering that you want all the legal English words which can be arrived at by changing one letter in the current word, how would you define the successor function for this search problem? You do not need to write down an answer for this question, but consider it before you start coding.

   Implement the successor function.

   Notes:

   - Your function should accept a *WordState* object, which just stores the current word, as an argument, and return a list of *WordState* objects which are **legal** successors to the argument.
   - Remember that we are ignoring case (i.e., "Mare" is automatically changed to "mare" when input).
   - Notice that *WordAStar*'s *self.dictionary* object is a set containing all the legal words from the dictionary file *words.txt* which are of the same length as the start

and goal words. Python sets are hash-table based and can be used for quick lookups to check membership. If you have a variable named *spam* and want to find out if its value is contained in the set named *breakfast*, you write *if spam in breakfast*. See the Python documentation on sets for more information.

- Also note the global *ALPHABET* variable, which is a list containing all the lowercase letters of the alphabet and may be useful.

You will be graded on correctness rather than efficiency, but try to consider running time. Penalties may be assessed for very inefficient solutions.

2. **Heuristic function for the word game**

Heuristic functions and heuristic search will be covered in class on Thursday, September 8.

Once you have implemented the successor function, the search will run and generate solutions (try it!). However, it will have to search quite a bit of the tree to do so – probably thousands of nodes. That is because the search is using a "null" heuristic function that returns 0 for all states. This heuristic is admissible but not very accurate. If you execute *wordgame.py* without arguments, it will describe a command-line switch that allows you to use a custom heuristic; however, the custom heuristic is currently unimplemented.

What do you think is a reasonable heuristic for a word's distance from the goal word? Remember that an admissible heuristic will never overestimate the distance to the goal. A good way to do this is to relax some constraint on the problem. What constraints does this problem have? Which of them is a good candidate for relaxation? Again, you do not have to write answers to these questions, but you should consider them before writing any code.

When you have designed a good heuristic function, implement it in the *WordAStar.heuristic_custom()* method. Try running your code by using the *-r* switch to *wordgame.py*. A good heuristic should provide a noticeable decrease in the number of nodes searched and not produce any error messages about decreasing nodes in the fringe.

Notes:

- Your function should accept a *WordState* object and return a number that is smaller when the word is closer to the goal.
- Note that the goal word for the search is stored in *WordAStar*'s *self.gw* member.

4 Example Test Cases

Use these test cases to acquire experimental data. Use that data when answering the written questions.

- *sat* to *roc*
- *arc* to *lot*
- *word* to *pare*
- *hare* to *fray*
- *taupe* to *brown*
- *smith* to *felid*
- *campus* to *coffee*
- *sweets* to *pastry*