B551 Homework 1

Due date: September 6, 2012

Instructions. Read this assignment carefully, complete the programming assignment and answer all of the written questions. Place all of your code in the *wordgame.py* file and your written answers in the *hw1_answers.txt* file. Submit both files on OnCourse.

Honor Code. You must NOT look at previously published solutions of any of these problems in preparing your answers. You may discuss these problems with other students in the class (in fact, you are encouraged to do so) and/or look into other documents (books, web sites), with the exception of published solutions, without taking any written or electronic notes. Computer code may NOT be shared.

Any intentional transgression of these rules will be considered an honor code violation.

1. Word Puzzle

In this homework, you will be asked to write search routines to solve a word puzzle 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.

2. Programming

The package *hw1.zip* from OnCourse contains this document, one Python file (*wordgame.py*) and one text file containing a spelling dictionary, one word per line (*words.txt*).

2.1. Running the program

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
```

This command will run a breadth first search (BFS) to search for a transformation from WORD1 to WORD2. However, it will not run correctly until you have answered question 1.

Different search algorithms can be run using the following flags:

`-r`: BFS with revisited state detection

`-i`: iterative deepening search

`-a`: A* search

However, these algorithms have not yet been implemented.

2.3. Documentation

If you wish, you may generate documentation for these files by running the following command:

```
pydoc -w wordgame
```

This will generate an HTML file describing the file's functions and classes in overview.

2.4. Data structures

The primary data structure defined in wordgame.py is a search tree, defined via the `SearchNode` class. It is a very simple tree, where each node simply contains a state (here, a Python string) and a back-pointer to its parent.

This code also makes use of the Python built-ins `list` (similar to a C array), `set` (an unordered collection with fast membership lookup), and `deque` (double-ended queue with O(1) pushing and popping). Testing if an item `x` is a member of a set `S` is accomplished using the statement "`if x in S`". You will also find the `dict` and `heapq` built-in data structures useful for answering the programming problems. See the Python documentation for more information about these data structures.

3. Problems

Question 1 (20 points)

Programming: Implement the `WordProblem.successors` method, which returns a set of valid English words that a given state (given as a Python string) can be transformed into using a single letter change. Notice that `WordProblem`'s `self.dictionary` member 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. 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.

Verify that *wordgame.py* now runs BFS correctly.

Written questions:

1) What is the worst-case branching factor of BFS on a problem with words of length *n*?
2) How does the effective branching factor compare to the worst-case? Test a handful of examples and discuss your observations.
3) Given a finite dictionary containing *M* words of length *n*, how many nodes might BFS explore in the worst case?

Question 2 (25 points)

Programming: The iterative deepening search (IDS) function is currently broken. Implement it, using the `depth_limited_DFS` function. Note that if the depth limit is hit, the `SearchResullt.successful` property is set to `'cutoff'`. Your implementation should detect this case (rather than a `True` or `False` result) and respond appropriately.

Verify that your system now runs IDS correctly using the `–i` command-line flag.

Written questions:

1) For which problems might IDS return failure? Will it ever return failure for the word puzzle? Why not?
2) Test a handful of example problems, and discuss how the number of nodes generated by IDS compares to regular BFS. Are the paths still optimal?

Question 3 (25 points)

Programming: The BFS with revisited-state detection in the `BFS_revisit` function is currently broken. Implement it, using the data structure of your choice. Make sure that you call `stats.on_revisit()` whenever a revisited state is pruned from the search tree so that the number of pruned states is printed to the console output.

Verify that your system now runs BFS with revisited-state detection correctly using the `–r` command-line flag.

Written questions:

1) Test a handful of example problems, and discuss how revisited state detection compares to regular BFS.
2) Given a finite dictionary containing *M* words of length *n*, how many nodes might BFS with revisited state detection explore in the worst case?

Question 4 (30 points)

Programming: The A* search function in the `astar_search` function is currently broken. Also, the `word_problem_heuristic` function currently returns 0 (the null heuristic). Implement an admissible heuristic and the A* search. You may wish to use the `heapq` module to implement the A* priority queue. Your A* should also discard revisited states properly.

Verify that your system now runs A* correctly using the `-a` command-line flag.

Written questions:

1) Describe the heuristic that you chose. Why is it admissible? Is it also consistent? What is its computational cost? Justify your decision.
2) Test a handful of example problems, and discuss how A* search compares to BFS, BFS+Revisit, and IDS in terms of number of nodes generated and the optimality of the resulting path.

Bonus Question (10 points)

As implemented, IDS will not terminate for problems without a solution (except when the start word is an isolated state). Modify IDS so that it does indeed terminate for all problems without a solution, but does not incur extra storage overhead. Describe your approach and argue that it is sound (give an informal proof).

4. Example Test Cases

Below are some test cases that you can use to acquire experimental data for 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*