COMS 4701 – Artificial Intelligence – Fall 2014

Assignment 3: Crossword Puzzle Solver

(version 2, November 3, 2014)

Due: November 14, 2014
(Includes November 3 deadline: see problem 3)

If your .bashrc, .profile, or .bash_profile file contains a line such as "ulimit -u 256", remove it or you will not be able to run the code for this assignment.

Introduction

In this assignment, you will develop a crossword puzzle solver.

A crossword puzzle consists of a grid of overlapping entries, each associated with a natural language clue which suggest possible words to fill the entry. According to the style used by the New York Times, which will be the source of our puzzles, all squares in the grid are members of two separate entries, one vertically-aligned ("down") and one horizontally ("across"). This means that the final word choices must not only be appropriate to the clues, but also fit with each other so that entries which share a square assign the same letter to that square.

Conceptually, we represent a puzzle as a constraint satisfaction problem. The variables of the problem are the entries, the domain of each variable is the set of word possibilities of appropriate length generated using a collection of natural language processing components, and the constraint is that entries sharing a square must place the same letter in that square.

Keep in mind that we don't just want to assign the variables to whatever words happen to complete the puzzle. We are more interested in making choices that fit the clues well. For each variable, each word possibility provided by the components is paired with a score; we want to maximize the total score of our solution. Of course, it would be computationally infeasible to try every possible combination of words. The algorithm we will use to approximate an optimal solution is a slightly modified version of Procedure 6.4 of Matthew Ginsberg's "Dr. Fill: Crosswords and an Implemented Solver for Singly Weighted CSPs."

Getting started

We've provided starter code in the ~cs4701/Project3/student directory. Run the following command to copy the given code into your own `Project3` directory:

```
cp -r /home/cs4701/Project3/student Project3
```

You must also modify certain variables:

```
export PATH=/usr/lib/jvm/java-1.7.0-openjdk-amd64/bin:$PATH
export PYTHONPATH=/home/cs4701/python/lib/python2.7/site-packages/
export APP_HOME=/home/cs4701/Project3/source_code/derek
export LIB=$APP_HOME/lib
export CP=$LIB/lucene-core-4.6.0.jar:$LIB/lucene-analyzers-common-
4.6.0.jar:$LIB/lucene-queryparser-4.6.0.jar:$LIB/json.jar:
$APP_HOME/classes/
```

The starter code includes a complete yet unsophisticated crossword puzzle solver. `solve_a_puzzle.py` is the entry point and takes two files as input. It constructs a `puzzle` object (defined by `puzzle.py`) from an input `.puz` file, and asks `components_eval.py` to generate possible words for each clue from the NLP components that are specified in the second input file, a list of component paths. It then asks `solver.py`, which contains a n implementation of a crude search algorithm, to find a solution. Once you've run the above commands, you should be able to run the following from the `puzzle_rui` subdirectory:

```
python solve_a_puzzle.py ../test_month/May0214.puz component_list >
output
```

If you open `output` and scroll to the bottom, you should see some evaluation results:

```
=================
Solution Generated      Actual Solution
LASERBEAM.IDOLS CLICKBAIT.ANGST
E---RO--E.RAMOS HENRIETTA.NORAH
AUSTRALIA.ESTER INSOMNIAC.DRIVE
DOIN..ELLEN.TAB RAPS..MLKJR.NAB
EOOYLOU..BETSEY APOSTLE..LEANNE
R---OP.RUE.NINA CETERA.AGO.NINA
...-PUSI-.JONAS ...DOTTIE.JONAS
MOVIESTOOKSIXTY FIVETHIRTYEIGHT
AGNES.EJ----... ADAYS.RESEWN...
NINE.RPA.LOOSES CORE.RED.LETSAT
TENANT..ALLENDE ENDSIT..ALLENDE
INN.AELSS..ISEE STA.KELPS..DAVE
L--LF.OHTEA---- OLLIE.OUTSPOKEN
L--ET.SORNC---- FIONA.STRAINERS
ARENA.STODE---- FESTS.STOPPEDBY



=================
 runtime : 0.28
 correct_answer_was_candidate : 48.00
 total_words : 70.00
 matching_words_before_fill : 6.00
 total_squares : 197.00
 runtime_before_fill : 0.28
 matching_squares_before_fill : 44.00

=================
 comps_eval
        ../components/lucene_wiki_component.py
                MRAR : 0.05
                Precision : 0.12
```

```
                Attempt_ratio : 0.94
        ../components/lucene_cwg_component.py
                MRAR : 0.31
                Precision : 0.65
                Attempt_ratio : 0.93
        ../components/lucene_otsys_component.py
                MRAR : 0.40
                Precision : 0.71
                Attempt_ratio : 0.93
        ../components/fill_in_the_blank_component.py
                MRAR : 0.26
                Precision : 0.50
                Attempt_ratio : 0.06
avg_eval
        avg_Precision : 0.50
        avg_Attempt_ratio : 0.71
        avg_MRAR : 0.25
```

1) Implement Dr. Fill Procedure 6.4

As you can see by the terrible performance of the initial solver, the solver algorithm in the starter code is very crude. At each step it arbitrarily selects an unassigned variable and assigns it to the highest-ranked remaining word choice for that variable, or to a string of "-" characters if there are no remaining options. (This special character represents a blank square; of course, if an intersecting entry assigns the square to an actual letter, that will take precedence.)

To improve the solver, you will replace this with an implementation of Dr. Fill Procedure 6.4.

Our version of the algorithm is defined as follows:

Define a function solve(C, S, B, n, P), where:
- C is the CSP defined by our puzzle
- S is some partial solution (eg assignments of some subset of the variables)
- B is the best solution seen thus far besides S
- n is some positive integer
- P is the set of assignments that have been pitched

solve(C, S, B, n, P):
    if S assigns every variable, return whichever of S and B has higher total score
    v <- the variable with the maximum difference between the scores of its 1st and 2nd highest-scoring word possibilities that are not in P
    d <- the highest-scoring word possibility of v that is not in P
    S' <- S plus the assignment of v to d
    C' <- propagation of the assignment of v to d in C (ie, set v to d in the CSP, make sure the constraint is still satisfied, and eliminate word possibilities for intersecting entries that would conflict with it)
    if C' is still valid, B <- solve(C', S', B, n, P)
    if |P|<n, P' <- P plus the assignment of v to d, and B <- solve(C, S, B, n, P')
    return B

We start by calling solve(C_0, {}, {}, n, {}), where C_0 is the initial CSP with no assignments.

(The difference between this algorithm and the full Dr. Fill procedure is the absence of a post-processing step.)

With the existing code as your guide, replace `solve_recursive` in `solver.py` with your implementation of this algorithm. Add whatever arguments and helper functions are necessary, and feel free to make modifications to other parts of the code as long as you don't change the format of the printed solution or evaluation results. Of course, you are not allowed to have your code peek at the solution.

Your implementation must perform well in terms of *runtime* and *accuracy*. However long the components may need to evaluate, the CSP solving process should not take more than a couple minutes to generate fairly good results. It may be helpful to compare your runtime and accuracy metrics with those of your fellow students if you have doubts about the performance of your solver.

In your writeup, discuss the performance of your solver under various n parameters. **In particular, find** 1) the lowest setting of n that lets your solver complete every puzzle in `monday_puzzles` with greater than 75% accuracy in terms of correct squares (before filling blank squares) and 2) the highest setting of n that lets your solver complete each of the puzzles in `monday_puzzles` in less than 20 minutes. Record both of these n values and the respective CSP solver runtime and per-square accuracy, before filling blank squares, on all four of the puzzles in `monday_puzzles`. You may want to use a command like this:

```
for f in ../monday_puzzles/*.puz; do ./solve_a_puzzle.py $f
component_list | grep 'runtime_before_fill\|total_squares\|
matching_squares_before_fill'; done
```

Briefly address the following question: What does n represent?

Include clear instructions for how we can change n. This shouldn't be harder than changing a constant in your code or providing a certain command line argument.

If you notice any opportunities to improve performance in a creative way, go for it. Include a description of what you tried and how well it worked. We will award bonus points if your approach seems particularly thoughtful and/or successful.

2) Fill in blank squares

You may notice that this algorithm often results in entries having a few blank squares. In this case, if any word at all matches the letters in the entry, it's a good idea to fill in the blanks rather than leaving empty squares. For instance, if we see an entry set to COMP-TER, we should replace that with COMPUTER.

Come up with a "second round" procedure that tries to fill in the blanks by matching partially-completed clues against a list of words. This list is provided by `answers_cwg_otsys.txt`.

Please don't alter the way you generate your solution from part 1. Your code should evaluate the results both before and after the second round. The comments at the bottom of `solve_puzzle` in `solver.py` lay out the most obvious way to do this.

Note that this step should only insert new letters in blank squares; it should never replace existing letters.

Compare your solver's performance before and after the second round, and briefly describe the algorithm your second round uses.

3) Add two custom components

You will be implementing two components, one from the list of component ideas we've provided, and one that you come up with on your own.

The first step is determining which component you will implement from the appended list of component ideas. We want to have as much coverage of these components as possible, so we will be coordinating who does which components. Each component is assigned a number. Please take note of the numbers of the top 5 components you are interested in. Send an email to ajm2209@columbia.edu with a subject line formatted as follows: `4701 components, yourUNI, choice1, choice2, choice3, choice4, choice5`

For instance, if my top choices are 14, 1, 11, 6, 3, I will write this subject line:
`4701 components, ajm2209, 14, 1, 11, 6, 3`

These subject lines will be processed automatically, so please be sure to match this format exactly.

Please send this email before 11:59 PM on November 3. If we do not receive your email by this deadline, or if it missing information, you may be assigned to an arbitrary component and/or lose points on this assignment.

We encourage you to include, in body of your email, a short description (one sentence is fine) of the original component that you want to develop, and a list of 10 clues from the puzzles provided that you would expect it to handle. If you don't include this in your November 3 email, please send it in another email, with subject line "4701 components," by November 8.

You must figure out on your own what sort of strategy is appropriate for implementing your components. You may want to experiment with different approaches to maximize performance. Computational demands should also be considered. In general, your component should generate correct word options several times (at least 5-10) on the puzzles in test_month.

It's most likely that our datasets will be sufficient for your component, but it's fine if you download another dataset to use, provided that it will fit on your CLIC disk space. The instructors will need access to any external data that you've downloaded and any code that you write.

The component API is as follows:

A component reads from standard input a sequence of tab-delimited lines of the entry ID, the clue, and the number of squares, eg:

```
1A	Intelligent	5
2D	Unhappy	3
```

In response to each line, the component should print tab-delimited lines of the clue ID, an appropriate word choice, and a confidence value for that word choice. So for the above example, we might print:

```
1A	smart	.8
1A	quick	.5
1A	witty	.3
2D	sad	.9
2D	low	.1
```

For each component you create, your writeup should include a description of the component, a list of the clues in `test_month` which trigger the component, and a quantitative summary of the component's performance and its impact on the performance of your solver. Lastly, please include a file of at least 10 clues from the puzzles which trigger the component, so that we can see it in action by running the following:

```
python yourcomponent.py < yourexamples
```

Submission

You will submit your code in the same hidden folder that you used in the last two assignments.

Place all your code and your writeup in this folder. You don't need to put datasets in this folder, but you may if you have sufficient disk space. (An easier solution is just to make a symbolic link or to set up your code with full paths.) The instructors must have access to all code and data that you created. Your writeup should clearly indicate how to solve a particular .puz file using your solver and your components. Indicate the exact commands we must run.

Be sure to set up permissions appropriately. You may lose points otherwise. To set the permissions on your code, assuming it is in ~/myprojectfolder, run:

```
chmod -R a+rx ~/myprojectfolder
```

Grading

Part 1: Implement Dr. Fill (50 points)
Part 2: Fill in the blanks (20 points)
Part 3: Two components (15 points each)
Bonus points:
We will run each solver against a predetermined puzzle using the starter code components. We will award 10 bonus points to the most accurate solver under 20 minutes and the fastest solver

that produces 75% accuracy on Monday puzzles, using the n values you report in your writeup.
We will also award up to 10 bonus points for any major improvements to the system that you can come up with. (Please specify in your writeup if you attempt something.)
Finally, we will award 10 bonus points to the most original component.

Deliverables

Problem 1:
Alter solver.py to use Dr. Fill. Write up a general summary of the performance, the 2 particular n values requested and results using those n values, a description of what n does and how to adjust it in your code, and (optionally) any creative improvements you made.
Problem 2:
Again, alter solver.py to add a "second round" fill-in-the-blank procedure. Write up your strategy and compare performance before and after.
Problem 3:
By end of day on November 3, send component summary email (see problem 3)
Create two instructor-approved components (see instructions)
For both components, list clues in `test_month` which trigger the component and summarize the component's performance. Also include a list of example inputs to your component.

Appendices

print_puzzle.py

The starter code includes this piece of code to display the .puz crossword puzzle input files. `print_puzzle.py` will display the completed puzzle as well as a list of its clues. This may be useful for understanding the type of clues you will be presented with.

For example:

```
$ ./print_puzzle.py ../test_month/May0214.puz
CLICKBAIT.ANGST
HENRIETTA.NORAH
INSOMNIAC.DRIVE
RAPS..MLKJR.NAB
APOSTLE..LEANNE
CETERA.AGO.NINA
...DOTTIE.JONAS
FIVETHIRTYEIGHT
ADAYS.RESEWN...
CORE.RED.LETSAT
ENDSIT..ALLENDE
STA.KELPS..DAVE
OLLIE.OUTSPOKEN
FIONA.STRAINERS
FESTS.STOPPEDBY
```

```
1A      Modern traffic director?        CLICKBAIT
1D      President beginning in 1995     CHIRAC
2D      Delaware Valley Indians LENAPE
3D      Hip place       INSPOT
4D      Strabismus      CROSSEDEYES
...
```

evalute_components.py

A useful script for evaluating components is `evaluate_components.py`. This will save individual component evaluation results in the directory `component_eval_results`. To test your components on every example puzzle in `test_month`, run from in `puzzle_rui`:

```
mkdir component_eval_results
python evaluate_components.py ../test_month component_list
```

Of course, you must add the path to your component to `component_list` so that it will be included.

To test a single component on every puzzle in `test_month` , use `component_test.py`:

```
python component_test.py ../test_month path_to_my_component
```

Starter components and datasets

We've provided 6 NLP components in the `components` directory, and we have also supplied a number of datasets that you may use for your own components.

Two of thee starter components, `lucene_cwg_component.py` and `lucene_otsys_component.py`, search datasets of past crossword puzzle solutions and return words that are paired with clues similar to the ones being searched for. `lucene_wiki_component.py` searches a local index of Wikipedia articles, and `fill_in_the_blank_component.py` also uses this index for the purpose of responding to "fill in the blank" style clues. All of these components perform this task with the help of a java script, `edu.umich.si.si561.SearchDriver`. The script takes two normal command line arguments – a number, representing the maximum number of options to identify, as well as a clue of one or more words. The system properties provided (as JVM arguments prefixed with -D) determine which dataset it searches.

Note that for any of these commands to work, CP (the classpath) must be set as previously specified.

Also note that there will be two "derek" folders, one in /home/cs4701/Project3/source_code and one in the project3 directory you copied over. The commands described here will work from your own "derek" directory, provided that you have set the specified variables.

To search the CWG dataset, we specify:

```
-Dclue_path_regex="../crossword_corpus/(.+).txt"
-Dclue_index_dir=/home/cs4701/Project3/source_code/derek/cwg_index
```

So a full command searching CWG might be:

```
java -Xmx512m -cp $CP \
-Dclue_index_dir=/home/cs4701/Project3/source_code/derek/cwg_index \
-Dclue_path_regex="../crossword_corpus/(.+).txt" \
edu.umich.si.si561.SearchDriver 2 Apple computer
```

This example command lists up to 2 words related to "Apple computer" in this JSON format:

```
{"limit":2,"query":"Apple computer","clue-search":
{"total":2,"results":[{"score":69.79560089111328,"word":"imac"},
{"score":40.297733306884766,"word":"mac"}],"status":"OK"}}
```

To search OTSYS, we specify the following option:

```
-Dotsys_index_dir=/home/cs4701/Project3/source_code/derek/otsys_index
```

For example:

```
java -Xmx512m -cp $CP \
-Dotsys_index_dir=/home/cs4701/Project3/source_code/derek/otsys_index \
edu.umich.si.si561.SearchDriver 2 Apple computer
```

The result is in the same format:

```
{"limit":2,"query":"Apple computer","otsys-search":
{"total":2,"results":[{"score":88.69232177734375,"word":"imac"},
{"score":49.78369903564453,"word":"mac"}],"status":"OK"}}
```

To search the Wikipedia dataset, we specify the following (N.B: the output format is different):

```
-Dclue_path_regex="../crossword_corpus/(.+).txt"
-Dwiki_index_dir=/home/cs4701/Project3/source_code/derek/wiki_index
```

We have also provided a script, `wikip_search_path.bash`, which performs a search of the Wikipedia index and returns a list of the paths of local versions of these articles.

Finally, to search an acronym dataset, we specify:

```
-Dclue_path_regex="/home/cs4701/Project3/source_code/components/
acronym_DB/acronymsandslang_corpus/(.+).txt"
-Dacronym_index_dir=/home/cs4701/Project3/source_code/derek/
acronym_index
```

Again, be careful about the output format, which varies slightly depending on the dataset. You may want to familiarize yourself with Python's JSON handling, which is very straightforward.

There are a few other datasets we've provided that you should be aware of.

`/home/cs4701/Project3/clues.txt` includes every clue in the CWG dataset, listed by solution (appended with ".txt"). So, for instance, it contains the following:

```
zzzquil.txt
```

```
OTC sleep-aid brand introduced in 2012
http://www.crosswordgiant.com/crossword-clue/1464123/OTC-sleep-aid-brand-
introduced-in-2012
OTC sleep-aid introduced in 2012
http://www.crosswordgiant.com/crossword-clue/1467322/OTC-sleep-aid-
introduced-in-2012
```

`/home/cs4701/Project3/source_code/components/auto_database/listofuniquecars.txt` is a list of the make and model of various cars.

`/home/cs4701/Project3/source_code/components/country_database/countries_acronyms_and_abbreviations.txt` lists acronyms for every country, and `state-country.txt` lists the capitals of various countries.

`/home/cs4701/Project3/source_code/components/president/president.txt` lists leaders, and historical leaders, of various countries.

`/home/cs4701/Project3/source_code/components/rock_band_db/band_list.dat` lists the names of various bands.

Lastly, we have the Python NLTK package installed, and have downloaded the Wordnet corpus. This is used by `wordnet_antonym_component.py` to find solutions to antonym clues. For details on how to use this system, see http://www.nltk.org/howto/wordnet.html

The Wordnet NLTK interface makes it easy to perform operations like getting the hypernyms of a word. The following example demonstrates this for the noun "cat."

```
% python

>>> import nltk
>>> from nltk.corpus import wordnet as wn
>>>
>>> noun='cat.n.01'
>>> print noun," ",
>>> result = wn.synset(noun)
>>> hyper = lambda s: s.hypernyms()
>>> for l in list(result.closure(hyper)):
>>>     print " ",l
```

The last component we've included, `state_capital_component.py`, simply returns the answer to clues asking for the capital of a particular state, e.g., "Capital of New Jersey." This component is provided to give you a simple example of how a component should work.

Component ideas (note that some of these might actually be very hard to implement):

1
Missing last name
Titanic actor Billy (4) ZANE
James who wrote "A Death in the Family" (4) AGEE
Cesar who played the Joker (6) ROMERO

Cheri formerly of "S.N.L." (5) OTERI

2
Missing first name
“Wrecking Ball" singer Cyrus (5) MILEY
Playwright O'Neill (6) EUGENE
Stuntman Knievel (4) EVEL
"Morning Joe" co-host Brzezinski (4) MIKA
Yiddish author Aleichem (6) SHOLOM

3
Roman years
Year John Dryden died (4) MDCC
Early third-century year (4) CCIV
2,502, to ancient Romans (5) MMDII

4
Common French words
When the day's done, to Denis (4) NUIT

5
Prefix
Puncture preceder (3) ACU
Intro to physics? (4) META
Plasm preceder (4) ECTO
Prefix with scope (6) STETHO

6
Suffix
Suffix with miss and dismiss (3) IVE
Suffix with shepherd (3) ESS
Medical suffix (3) OMA
Fox tail? (4) TROT

7
Direction
New Orleans-to-Detroit dir. (3) NNE

8
World capitals
Samoa's capital (4) APIA

9
Common French words
Black: Fr. (4) NOIR
Mrs., in Marseille (3) MME

10

Common German words

11
Common Spanish words

12
Common Portuguese, Italian, Hawaiian, etc. words
"Thank you," in Hawaii (6) MAHALO

13
Brands
Hyundai and Kia (5) AUTOS

14
Competitors
Marriott alternative (4) OMNI

15
Acronym
Canoodling in a restaurant, e.g. (abbr.) (3) PDA
Chemists' org (3) AIC

16
Hypernym
Platinum, for example (5) METAL
Cronus and Hyperion (6) TITANS
Shrek or Fiona (5) OGRES

17
Synonym
Overshoot, say (4) MISS
Gait (4) PACE

18
Roles
1963 Elizabeth Taylor role (9) CLEOPATRA

19
Products
Honda model (6) ACCORD

20
Cities
Largest city in Nebraska (5) OMAHA

21
Songs and performers
"Owner of a Lonely Heart" band (3) YES

22
Famous person
Former South African president (5) BOTHA

23
Partner
Gentleman's partner (4) LADY

24
Dictionary Definitions
Coffee dispenser (3) URN

25
2+ word phrases
On task (4) ATIT
Drink a little here, drink a little there... (6) BARHOP
Went without a copilot (8) FLEWSOLO
An operator may help place one (9) PHONECALL
Flown into a rage (13) GONEBALLISTIC
Wine-producing area of SE France (11) RHONEVALLEY
Food, warmth or a cozy bed (15) CREATURECOMFORT
Money available for nonessentials (9) SPARECASH

26
Counterpart
Bull's counterpart (4) BEAR

27
Alternative
Reebok alternative (4) NIKE

28
Musical key
Key of Mozart's Symphony No. 40 (6) GMINOR

29
Movies
Tom Cruise film set in Chicago (13) RISKYBUSINESS

30
Co-star
Tom Cruise's 'Risky Business' co-star (15) REBECCADEMORNAY

31
Products
Canon camera (3) SLR

32
Characters
Judy's brother on the Jetsons (5) ELROY

33
Fill in the blank (for parts of names or words, rather than phrases in general)
___ Beach, S.C. (6) MYRTLE
Boston ___ Party (3) TEA