

Word

Due: 6:00PM 10/15/09

```

E N A L P R I A           B L I M P
E G A I R R A C         D U E K I T E
N V D R U N G Y         P Y S S U R R E Y
W R           L A         E L C Y C I B
E A           I C         E A S G D
L O           D H         L K D
E B           E T         G   A A M
V E           R D         E A U   T T O
A T A H A O E           D O U   O E T
T A Y I G X H T T N E N T R N O N
O K N S P E D C P E O P A O A R O S
R S L S K I E S A O K G O C L C O L K
J E S H U T T L E O C C A M P Y L E M
D K C A B E S R O H C I O W T C L I I
H T E T U H C A R A P E L R E L A G W
V O E N A L P O R D Y H G E J E B H S
  I R Q   W A H S K C I R A H P Y X
T R U C K   O S U B W A Y   T L
T A O B V                               T E S G
S H I P E                               Q E L B
C J D                                   F Y
    
```

AIRPLANE	DOG SLED	MOPED	SKIES
AUTO	ELEVATOR	MOTORCYCLE	SLEIGH
BALLOON	ESCALATOR	PARACHUTE	STAGECOACH
BICYCLE	FEET	RICKSHAW	SUBWAY
BLIMP	GLIDER	ROCKET	SURREY
BORT	HELICOPTER	RUN	SWIM
BUS	HORSEBACK	SHIP	TRAIN
CANOE	HYDROPLANE	SHUTTLE	TRUCK
CARRIAGE	JET PLANE	SKATE	WAGON
CHARIOT	KITE	SKATEBOARD	YACHT

Introduction

You have successfully thwarted a Cylon attack on Earth. However, you cannot rest on your laurels. It is time to go on the counter offensive.

A *word search* is a common childhood puzzle in which a set of words is hidden within a group of letters placed on a grid. Each word appears in a straight line either horizontally, vertically, or diagonally, and either forwards or backwards. For example, TRUCK and SHIP are hidden within the back wheel of the train in the above figure.

Our intelligence indicates that the Cylons are mesmerized by these simple puzzles. We believe that if enough of these puzzles were distributed within the Cylon fleet, the resulting distraction would bring their military to a standstill.

Your goal in this project is not to write a word search solver, but rather a word search *generator*. You will be provided with a set of words and a $m \times n$ board, and you must create a word

search containing the given words. First, you will approach this problem as one of constraint satisfaction, where you must find a feasible configuration of the words on the board. Second, you will approach this problem as one of optimization, where you will be asked to optimize some objective function (e.g., maximize the number of words or minimize the size of the board).

Getting Started

1. **Formulate the word search puzzle generation problem as a constraint satisfaction problem (CSP).** As a first step, formalize the problem of generating a word search puzzle given W words and an $m \times n$ board as a CSP. What are the variables? What are their domains? What are the constraints? Consider at least two formulations of the problem and speculate on the relative strengths and weakness of each. Which do you expect to work better, and why?
2. **Formulate word search puzzle generation as a local search problem.** All CSPs can be formulated as optimization problems. Reformulate the word search puzzle generation problem as an optimization problem. To do so, you must describe the states in your search space and an objective function. The standard objective is to minimize the number of “conflicts”. What kind of conflicts arise in word search puzzle generation? Further, define a neighborhood relation so that word search puzzle generation can be viewed as a local search problem.

Part I: Word Search Puzzle Generation as a CSP

Recall that any CSP can be posed as a search problem on a tree, where each state corresponds to a partial assignment, the initial state is the empty assignment, the goal states are the complete and consistent assignments, and the successor relation extends partial assignments in a consistent manner. With this in mind, you are ready to begin implementing a word search puzzle generator.

1. **Implement backtracking.** This algorithm is depth-first search on a tree, where at each node, one of the as yet unassigned variables is assigned a value. For more information about the backtracking algorithm, see the course notes or page 142 of Russell and Norvig.
 - (a) Implement the backtracking algorithm in the `solveWithBacktracking` method in the `WordSearchGenerator` class. In order to reduce the amount of memory required, each search node should not contain a copy of the entire board. Instead, you should make incremental changes to the board as you traverse the tree. When you backtrack, undo these board changes.
 - (b) Create at least three test problems in the `puzzles` directory and note the performance of your algorithm on each test instance. Document both the time the algorithm required to terminate, as well as the number of nodes visited before a solution was found.

2. **Implement the minimum-remaining values heuristic.** The minimum-remaining-values (MRV) heuristic helps the backtracking algorithm determine which unassigned variable to assign a value to next. It chooses the variable with the smallest domain. For more information on the MRV heuristic, see the course notes or page 143 of Russell and Norvig.
 - (a) Implement the MRV heuristic. Make the heuristic configurable, so that you have the option of reverting back to the original, more naive variable selection method as desired.
 - (b) As above, run your program on the word search test instances you created in the `puzzles` directory and note the runtime of your algorithm and number of nodes visited for each instance. Do the problems that the heuristic performs particularly well or poorly on share any common characteristics?

3. **Implement the least-constraining-value heuristic.** This heuristic helps the backtracking algorithm determine which value to assign to a given variable. The algorithm chooses the value that eliminates the fewest number of possible values for the other unassigned variables. For more information on the least-constraining-value heuristic, see the course notes or page 144 of Russell and Norvig.
 - (a) Implement the least-constraining-value heuristic. Make this heuristic configurable so that you can revert back to randomly selecting a value to assign to the variable.
 - (b) As above, run your program on the word search test instances you created in the `puzzles` directory and note the runtime of your algorithm for each instance. Try using different combinations of MRV and LCV (both, neither, etc.). If the performance of LCV along some dimension (time or number of nodes visited) is worse with this heuristic, try to figure out why that might be? Do you have any ideas for a different heuristic that might perform better?

4. **Implement forward checking.** Modify your program further to implement forward checking. In this modification, whenever a variable is assigned a value, the domains of all other unassigned variables are checked for feasibility. Any value to an unassigned variable that is not feasible is removed from the domain. Forward checking should help reduce the number of nodes visited. For more information on forward checking, see Amy's course notes or page 144 of Russell and Norvig. After implementing this modification, run your program on your test instances and document its performance. Provide any insights you might have about the performance of forward checking in the word search generation problem.

5. **Extra Credit: Implement arc consistency.** Modify your program so that it performs arc consistency checks. In this modification, each pair of unassigned variables is checked for consistency, as follows: consider assigning unassigned variable x a value v in its domain. If there is another unassigned variable whose domain becomes empty when x is assigned v , then v is not a feasible assignment for x , and should be removed from x 's domain. With arc consistency, even fewer nodes should be expanded. For more information on arc

consistency, see Amy's course notes or page 145 of Russell and Norvig. After implementing arc consistency, run your program on each of your test instances and record the results.

Part II: Word Search Puzzle Generation as Local Search

A local search method can be used to solve a CSP by moving from state to state with the goal of minimizing the number of conflicts. For example, to use simulated annealing to solve a CSP, you might allow many conflicts initially, but as your search proceeds, you should allow fewer and fewer conflicts until eventually there are none left at all (hopefully). To avoid a situation in which a solution with no conflicts is not found, it may be necessary to restart simulated annealing multiple times or tweak the scheduled rate of exploration.

Your next task is to develop a variant of simulated annealing and apply it to the word search generation CSP posed as an optimization problem.

1. **Optimize for Minimal Conflicts** Roughly speaking, each iteration of your algorithm should choose a word (presumably one involved in many conflicts) and place it in another location with (hopefully) few conflicts. For example, your algorithm could choose and place words probabilistically, being more likely to choose words involved in many conflicts, and more likely to place words in locations that create few conflicts.

This algorithm should be used when the `solveWithMinConflictsLocalSearch` method is called. The details of your algorithm (e.g., how it makes probabilistic choices) are up to you, but be sure to document your choices in your README file.

2. **Optimize for Maximal Intersections** At this point, you have implemented two programs that generates word search puzzles. You may notice, however, that your word search puzzles are not always very interesting. For example, if you are given 4 words of length less than or equal to 4 to insert in a 4x4 board, your program might place each one in its own row beginning in the leftmost column. One way you could imagine generating more interesting word search puzzles is by maximizing the number of intersections between words.

Modify your local search algorithm so that at each iteration it chooses a word (presumably one involved in many conflicts but few intersections) and place that word in another location with (hopefully) few conflicts and many intersections.

This algorithm should be used when the `solveWithMaxIntersectionsLocalSearch` method is called. As above, the details of your algorithm (e.g., how it makes probabilistic choices) are up to you, but be sure to document your choices in your README file.

3. **Summarize your results.** Specifically, provide examples of word search puzzles generated by your two local search algorithms. Is one set of examples more interesting than the other?

Support Code

You can get the source code you need for this project by running `cs141-install word`, which will copy all of the files in `/course/cs141/src/word` to your home directory in `~/course/cs141/word`.

The support code is configured as an Eclipse workspace. Simply import the word directory as an existing Eclipse project to get started.

We have provided the source for all of the support code for this project. Javadocs for all of the support code are available at <http://cs.brown.edu/courses/cs141/word/index.html>.

Important Classes

WordSearchGenerator contains the three methods that correspond to the three coding portions of the assignment. Your code goes here.

The **WordSearchProblem** class is the input type to your generator methods. It provides you with a **Board** and a **WordBank**

The **WordBank** class provides the list of words you are expected to generate a word search puzzle for.

The **Board** class represents the state of the puzzle. It provides mutator methods that return **BoardChanges** objects. These objects allow you to store incremental changes rather than whole boards at each step. While the code also supports a torus shaped board, you won't be dealing with such boards for this assignment.

The **WordSearchSolution** class is the expected return type for your generator methods. It contains the final **Board**, the **WordBank** with all locations, and a list of incremental changes leading to the solution.

What to Handin

Please hand in the following:

1. All the code necessary to solve Parts I and II of the assignment and any extra credit.
2. A README file that includes the following:
 - (a) Standard bug reports.
 - (b) Anything that might help us read your code: design quirks, conventions, etc.
 - (c) Comments that might help us improve this project in the future.
3. A short report tabulating the results of running each successive modification of your CSP search algorithm on your test instances, and summarizing the output of your two local search algorithms. The file `table.tex` in the support code, part of which compiles as

shown, is a convenient format for reporting statistics on your implementations of the CSP heuristics.

MRV	LCV	Test 1	Test 2	Test 3
F	F	(time, nodes)	(time, nodes)	(time, nodes)
T	F	(time, nodes)	(time, nodes)	(time, nodes)
F	T	(time, nodes)	(time, nodes)	(time, nodes)
T	T	(time, nodes)	(time, nodes)	(time, nodes)

Our handin script turns in everything in the current directory and all subdirectories. To hand in this project, navigate into the directory containing the files and directories you wish to hand in and run `cs141-handin word`.