CS 452/552: Assignment 2

Due: November 6, 2023, by 11:00 PM (Central)

Overview

Your goal for this assignment is to write a program that can solve crossword puzzles formulated as constraint satisfaction problems. The puzzle data is provided in input files, and your solver should allow the user to test various improvements for the backtracking search algorithm discussed in class. Program requirements can be found starting on page 2; note that students registered for CS 552 have **additional requirements** that can be found on page 9. Grading notes and some **hints** can be found on page 11.

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "backtracking search in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must not make use of any non-standard or third-party libraries.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**

Program Requirements

Your program should be runnable from the command line, and it should be able to process command-line arguments to update various program options and parameters as needed. The command-line arguments are listed below:

- `-d <FILENAME>`: Reads dictionary data from the text file named `<FILENAME>` (specified as a `String`); the filename may include path information, so do not assume that the file is located in the same directory as your source code. File format details are on page 3.
- `-p <FILENAME>`: Reads puzzle data from the text file named `<FILENAME>` (specified as a `String`); the filename may include path information, so do not assume that the file is located in the same directory as your source code. File format details are on page 3.
- `-v <INTEGER>`: Specifies a verbosity level, indicating how much output the program should produce; default is `0` (see the **Output** section for details).
- `-vs <STRING>` **or** `--variable-selection <STRING>`: Specifies how variables should be ordered for variable selection in backtracking; `<STRING>` should be one of:
  - `static`: Use fixed ordering.
  - `mrv`: Select variables using the **minimum remaining values** heuristic (also called the **most constrained variable** heuristic); ties may be broken arbitrarily.
  - `deg`: Select variables using the **degree** heuristic (also called the **most constraining variable** heuristic); ties may be broken arbitrarily.
  - `mrv+deg`: Select variables using the **minimum remaining values** heuristic, with ties broken using the **degree** heuristic.

  If this argument is not provided, then `static` ordering should be used by default.
- `-vo <STRING>` **or** `--value-order <STRING>`: Specifies the order in which a variable's values should be iterated; `<STRING>` should be one of:
  - `static`: Use fixed ordering.
  - `lcv`: Order values using the **least constraining value** heuristic.

  If this argument is not provided, then `static` ordering should be used by default.
- `-lfc` **or** `--limited-forward-check`: Indicates that limited forward checking should be applied when checking constraints for consistency; see page 4 for details.
- `--preprocess`: [**Only required for CS 552 students**] Enables the use of constraint propagation as a preprocessing step (which is disabled by default); see page 9 for details.

The `-d` and `-p` options (with arguments) are required; the remaining ones are optional (meaning that they might not be specified when the program is run, **not** that you don't have to support them!). You can assume that your program will only be run with valid arguments (so you do not need to include error checking, though it may be helpful for your own testing). Your program must be able to handle command-line arguments in **any** order (e.g., do **not** assume that the first argument will be `-d`).

For example, if you write a Java program with `main` method in a class named `Solve`, then your program could be run as:

```
shell$ java Solve -d dictionary-small.txt -p xword00.txt
```

A more detailed run might specify additional options:

```
shell$ java Solve -d dictionary-small.txt -p xword00.txt -vs mrv -v 3
```

Any program that does not work this way will lose points from the assignment grade. In particular, your program should not require that the source code be edited (and possibly re-compiled) in order to run with different inputs, and your program should not ask the user for any information via standard input after it starts running.

File Formats

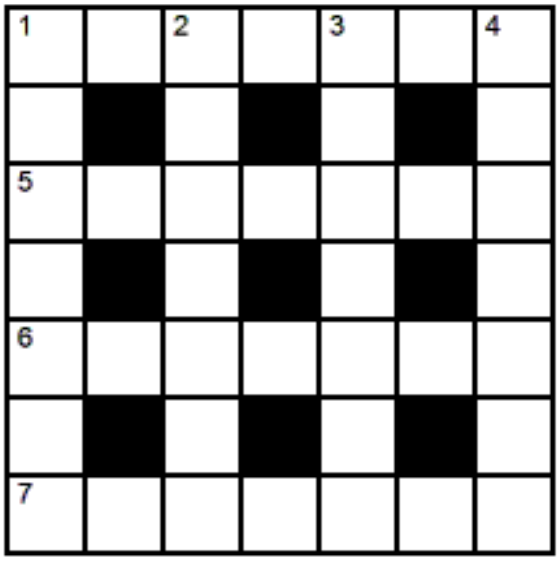
After processing command-line arguments, your program should open the specified files to load the dictionary data and the puzzle structure. The dictionary file consists of a list of words, with one word on each line. As an example, the `dictionary-small.txt` file contains 8 words, with the file contents shown below:

```
ANOTHER
DEVELOP
DETRACT
LEATHER
PROGRAM
THEOREM
TORNADO
VERBOSE
```

A puzzle specification file begins with a line containing two integers `ROW` and `COL`, specifying the number of rows and columns in the puzzle. The subsequent lines specify the squares in the crossword puzzle, with each square being a **numbered square**, a **blank square** (represented by `_`), or a **black square** (represented by `#`). Consecutive squares in a row are separated by whitespace within the file. For example, the puzzle file `xword01.txt` and the puzzle grid it represents look like:

Contents of `xword01.txt`:

```
7 7
 1  _  2  _  3  _  4
 _  #  _  #  _  #  _
 5  _  _  _  _  _  _
 _  #  _  #  _  #  _
 6  _  _  _  _  _  _
 _  #  _  #  _  #  _
 7  _  _  _  _  _  _
```



The goal of a crossword puzzle is to fill in the numbered and blank squares with **words**.

- A **word** begins at a numbered square and fills up all blank squares leading across or down from that numbered square until encountering the edge of the puzzle or a black square.
- **Across words** always have a puzzle edge or a black square to the **left of their number**; **down words** always have a puzzle edge or a black square **above their number**.
- If two words intersect at a given blank square, then their letters must match.

CSP Formulation

After loading the data, your program should formulate a constraint satisfaction problem (CSP) representing the crossword puzzle to be solved.

- The CSP should include one variable for each word in the crossword puzzle.
- For each variable, the domain of values will consist of all words of the right length from the dictionary file.
- The CSP should include a constraint for each pair of intersecting words in the grid that requires that the two words match at their intersection.

After formulating the CSP, your program should print the number of variables and number of constraints, along with additional details as appropriate (see page 5 for examples).

Solving the CSP

After formulating the CSP, your program should perform the basic **backtracking search** outlined in Lecture 04-2/3 to find a solution or determine that none exists. The search should terminate when an assignment is found that satisfies all constraints or when it is determined that no such solution exists.

In backtracking search, when considering a potential assignment of $X_i = v_i$, the basic consistency check should determine whether this potential assignment would violate any constraints involving variables X_i and X_j in which X_j has already been assigned a value. By default, constraints involving variables X_i and X_k where X_k has not yet been assigned a value should not be checked here (they should get checked "further down" when values for X_k are being considered).

If the **limited forward check** option (`-lfc` flag) is used, then the consistency check should **also** consider constraints involving X_i and X_k where X_k has not yet been assigned a value. Specifically, if the potential assignment of $X_i = v_i$ can work with some future assignment $X_k = v_k$, then there is no consistency problem with using $X_i = v_i$ (at least in terms of the constraint with X_k). However, if there are no values v_k for which the future assignment $X_k = v_k$ would be consistent with the potential assignment of $X_i = v_i$, then limited forward checking should indicate an inconsistency because the constraint involving X_i and X_k cannot be satisfied with $X_i = v_i$.

Variable selection and value ordering should use the options specified via command-line arguments, or the `static` options by default; see Lecture 04-3 for details. Static variable ordering should consider the variables in the order dictated by their puzzle square numbers; for puzzle square numbers with both an **across word** and a **down word**, the variable for the **across word** should come first. Static value ordering should consider values in alphabetical order.

Additionally, for the heuristics, it is **strongly recommended** that you recompute these values from scratch every time they are needed, instead of trying to maintain and update them during the search. This adds redundant computational work during the backtracking process but it avoids the implementation overhead required to update the heuristic values after each assignment and revert them when backtracking. Once you have the values computed correctly and everything working properly, then you *might* consider efficiency improvements, though such improvements are not required. (Additionally, if you do include efficiency improvements such as domain reduction during search, then you should make sure that these improvements are disabled by default and can be toggled by a supplemental command-line flag.)

Output

The `-v` command-line argument controls the **verbosity level** of the program. The default value is `0`; additional output is produced for each successive verbosity level. Your program's output does not need to exactly match the examples shown below, but it should be similar. Additional output will be posted on Canvas in the next few days which may be helpful in debugging and/or checking your own implementation.

Output Level 0

Verbosity level `0` should print the results of the search process after it ends (i.e., `SUCCESS!` or `FAILED`), along with the time taken and the number of distinct recursive backtracking calls that were made. If the search is successful, then your program should also print the final solution, with black squares in the grid printed as spaces to improve readability.

Example output from running the CSP solver on the `xword01.txt` puzzle with the `dictionary-small.txt` dictionary is shown below on the left, with the completed puzzle grid shown on the right:

```
shell$ java Solve -d dictionary-small.txt -p xword01.txt
SUCCESS! Solving took 11ms (15 recursive calls)

DEVELOP
E E E R
TORNADO
R B T G
ANOTHER
C S E A
THEOREM
```

D	E	V	E	L	O	P
E	■	E	■	E	■	R
T	O	R	N	A	D	O
R	■	B	■	T	■	G
A	N	O	T	H	E	R
C	■	S	■	E	■	A
T	H	E	O	R	E	M

Example output from a failed search with verbosity level `0`:

```
shell$ java Solve -d dictionary-small.txt -p xword02.txt
FAILED; Solving took 1ms (1 recursive calls)
```

Output Level 1

Verbosity level `1` prints some additional information about the steps that the program performs along with the numbers of variables and constraints in the CSP:

```
shell$ java Solve -d ../a02-data/dictionary-small.txt -p ../a02-data/xword00.txt -v 1
* Reading dictionary from [../a02-data/dictionary-small.txt]
* Reading puzzle from [../a02-data/xword00.txt]
* CSP has 4 variables
* CSP has 4 constraints
* Attempting to solve crossword puzzle...

SUCCESS! Solving took 11ms (7 recursive calls)

DEVELOP
E     R
V     O
E     G
L     R
O     A
PROGRAM
```

Output Level 2

Verbosity level 2 also prints the number of words in the dictionary and the puzzle itself. Additionally, during backtracking search, the program should print, in a tree-like format, the variable selected within a backtracking call and the value(s) that are tried for the variable, as well as whether or not those values are found to be consistent with the current partial assignment:

```
shell$ java Solve -d ../a02-data/dictionary-small.txt -p ../a02-data/xword00.txt -v 2
* Reading dictionary from [../a02-data/dictionary-small.txt]
** Dictionary has 8 words

* Reading puzzle from [../a02-data/xword00.txt]
** Puzzle
 1  _  _  _  _  _  2
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 3  _  _  _  _  _  _

* CSP has 4 variables
* CSP has 4 constraints
* Attempting to solve crossword puzzle...
** Running backtracking search...
Backtrack:
  Trying values for X1a
  Assignment { X1a = ANOTHER } is consistent
  Backtrack:
    Trying values for X1d
    Assignment { X1d = ANOTHER } is consistent
    Backtrack:
      Trying values for X2d
      Assignment { X2d = ANOTHER } is inconsistent
      Assignment { X2d = DEVELOP } is inconsistent
      Assignment { X2d = DETRACT } is inconsistent
      Assignment { X2d = LEATHER } is inconsistent
      Assignment { X2d = PROGRAM } is inconsistent
      Assignment { X2d = THEOREM } is inconsistent
      Assignment { X2d = TORNADO } is inconsistent
      Assignment { X2d = VERBOSE } is inconsistent
    Assignment { X1d = DEVELOP } is inconsistent
    Assignment { X1d = DETRACT } is inconsistent
    Assignment { X1d = LEATHER } is inconsistent
    Assignment { X1d = PROGRAM } is inconsistent
    Assignment { X1d = THEOREM } is inconsistent
    Assignment { X1d = TORNADO } is inconsistent
    Assignment { X1d = VERBOSE } is inconsistent
  Assignment { X1a = DEVELOP } is consistent
  Backtrack:
    Trying values for X1d
    Assignment { X1d = ANOTHER } is inconsistent
    Assignment { X1d = DEVELOP } is consistent
    Backtrack:
      Trying values for X2d
```

```
      Assignment { X2d = ANOTHER } is inconsistent
      Assignment { X2d = DEVELOP } is inconsistent
      Assignment { X2d = DETRACT } is inconsistent
      Assignment { X2d = LEATHER } is inconsistent
      Assignment { X2d = PROGRAM } is consistent
      Backtrack:
        Trying values for X3a
        Assignment { X3a = ANOTHER } is inconsistent
        Assignment { X3a = DEVELOP } is inconsistent
        Assignment { X3a = DETRACT } is inconsistent
        Assignment { X3a = LEATHER } is inconsistent
        Assignment { X3a = PROGRAM } is consistent
        Backtrack:
          Assignment is complete!

SUCCESS! Solving took 16ms (7 recursive calls)

DEVELOP
E     R
V     O
E     G
L     R
O     A
PROGRAM
```

With the **limited forward check** option, the search process is faster:

```
shell$ java Solve -d ../a02-data/dictionary-small.txt -p ../a02-data/xword00.txt -v 2 -lfc
* Reading dictionary from [../a02-data/dictionary-small.txt]
** Dictionary has 8 words

* Reading puzzle from [../a02-data/xword00.txt]
** Puzzle
 1  _  _  _  _  _  2
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 3  _  _  _  _  _  _

* CSP has 4 variables
* CSP has 4 constraints
* Attempting to solve crossword puzzle...
** Running backtracking search...
Backtrack:
  Trying values for X1a
  Assignment { X1a = ANOTHER } is inconsistent
  Assignment { X1a = DEVELOP } is consistent
  Backtrack:
    Trying values for X1d
    Assignment { X1d = ANOTHER } is inconsistent
    Assignment { X1d = DEVELOP } is consistent
    Backtrack:
```

```
      Trying values for X2d
      Assignment { X2d = ANOTHER } is inconsistent
      Assignment { X2d = DEVELOP } is inconsistent
      Assignment { X2d = DETRACT } is inconsistent
      Assignment { X2d = LEATHER } is inconsistent
      Assignment { X2d = PROGRAM } is consistent
      Backtrack:
        Trying values for X3a
        Assignment { X3a = ANOTHER } is inconsistent
        Assignment { X3a = DEVELOP } is inconsistent
        Assignment { X3a = DETRACT } is inconsistent
        Assignment { X3a = LEATHER } is inconsistent
        Assignment { X3a = PROGRAM } is consistent
        Backtrack:
          Assignment is complete!

SUCCESS! Solving took 14ms (5 recursive calls)

DEVELOP
E     R
V     O
E     G
L     R
O     A
PROGRAM
```

Some additional examples are shown below:

```
shell$ java Solve -d ../a02-data/dictionary-large.txt -p ../a02-data/xword01.txt \
                  --variable-selection static
SUCCESS! Solving took 63ms (2235 recursive calls)

BEACONS
E I V Y
AIMLESS
C L R T
OVERSEE
N S E M
SYSTEMS
```

```
shell$ java Solve -d ../a02-data/dictionary-large.txt -p ../a02-data/xword01.txt \
                  --variable-selection mrv
SUCCESS! Solving took 63ms (11 recursive calls)

COMMITS
O O L P
MODELLI
M E U N
ILLUVIA
T L I C
SPINACH
```

Testing

When you have finished your implementation, your program should be able to solve the smallest puzzles in most cases. You should test this as follows:

1. Test the simplest puzzle grid (`xword00.txt`) against all three of the dictionaries. A proper implementation should solve the puzzle in all three cases very quickly.
2. Test the second-simplest puzzle grid (`xword01.txt`) against the `small` and `medium` dictionaries. A proper implementation should solve the puzzle in the `small` and `medium` cases very quickly, while the `large` case may take a few seconds depending on your implementation.
3. Test the more complex puzzle grid (`xword02.txt`) against the `small` dictionary. A proper implementation should very quickly determine that the problem can't be solved, as the dictionaries lack the words necessary to fill the grid. Trying to solve this puzzle with either the `medium` or `large` dictionaries will take a long time with default search options, but should be faster with appropriate heuristic improvements and the limited forward check option.

Required for CS 552 Students: Constraint Propagation

The requirements described in this section are **required** for graduate students (i.e., those registered for CS 552); they are *optional* for students registered for CS 452.

As noted in class, **constraint propagation** can be used before backtracking search to potentially reduce the size of the domains for variables and thereby reduce the branching factor in the search itself. If the `--preprocess` command-line argument is specified, then your program should perform constraint propagation by enforcing **arc consistency** across all variables using the `AC-3` algorithm (outlined in Lecture 04-4) prior to beginning the backtracking search process.

With a verbosity level is 2, your program should print out the sizes of each variable's domain both before and after enforcing arc consistency. An example is shown below:

```
shell$ java Solve -p ../a02-data/xword00.txt -d ../a02-data/dictionary-small.txt \
                  -v 2 --preprocess
* Reading dictionary from [../a02-data/dictionary-small.txt]
** Dictionary has 8 words

* Reading puzzle from [../a02-data/xword00.txt]
** Puzzle
 1  _  _  _  _  _  2
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 _  #  #  #  #  #  _
 3  _  _  _  _  _  _

* CSP has 4 variables
* CSP has 4 constraints
* Attempting to solve crossword puzzle...
** Preprocessing: constraint propagation
  X1a: Revised domain has 2 values (was 8)
  X1d: Revised domain has 2 values (was 8)
  X2d: Revised domain has 3 values (was 8)
  X3a: Revised domain has 3 values (was 8)
```

```
** Running backtracking search...
Backtrack:
  Trying values for X1a
  Assignment { X1a = DEVELOP } is consistent
  Backtrack:
    Trying values for X1d
    Assignment { X1d = DEVELOP } is consistent
    Backtrack:
      Trying values for X2d
      Assignment { X2d = PROGRAM } is consistent
      Backtrack:
        Trying values for X3a
        Assignment { X3a = PROGRAM } is consistent
        Backtrack:
          Assignment is complete!

SUCCESS! Solving took 13ms (5 recursive calls)

DEVELOP
E     R
V     O
E     G
L     R
O     A
PROGRAM
```

With arc consistency, a proper implementation should be able to solve any combination of puzzle and dictionary files very quickly (even without the limited forward check option and heuristics), either by finding a solution or determining that none exists; be sure to test this.

```
shell$ java Solve -p ../a02-data/xword02.txt -d ../a02-data/dictionary-large.txt \
                  -v 0 --preprocess
SUCCESS! Solving took 102ms (75 recursive calls)

ARRR CROP  APPS
NOAH LOVES ROOT
DAVYJONESLOCKER
IRISES REAM EMU
    LEAST AVAST
TALKLIKEA HER
ATOI NIE MAGOG
BASSOON PENGUIN
 DEMON PAD INGA
  NEC PIRATEDAY
BOOTY ANKLE
ACT TICK LAALAA
SHIVERMETIMBERS
IRMA SAYSO BACH
NEET  NEON AHOY
```

Grading

At a high level, your assignment grade will depend on the following:

1. Proper file I/O and data representation
2. Proper CSP formulation
3. Basic backtracking search:
   - Builds partial assignment recursively
   - Uses `static` options for variable selection and value ordering
   - Does basic consistency checking
4. Additional improvements:
   - Support for limited forward check option
   - Variable selection heuristics `mrv`, `deg`, and `mrv+deg`
   - Additional value ordering heuristics: `lcv`
   - Constraint propagation for preprocessing

In order to earn points in one of these categories, your program needs to be correct or nearly correct across all preceding categories. For example, a program that formulates an **incorrect** CSP for crossword puzzles cannot earn full credit for the backtracking search category, even if its backtracking search is correct for the CSP that was formulated. Similarly, a program with a **non-working** implementation of backtracking search cannot earn points for its implementation of additional variable selection heuristics. This means that you should focus your implementation efforts on these categories **in order**, and that you should be reasonably sure that each category is working properly **before** moving on to the next one. Simply throwing together a bunch of partially correct code for each of the categories is likely to result in a **bad grade**!

Hints

- Think about how you should represent the variables and constraints in your CSP formulation. You will need to use both of these types heavily in your backtracking search, so make sure that you design them appropriately.
- In a basic backtracking search, the CSP formulation and in particular the variable domains remain **fixed** during the search; only the partial assignment changes. This makes the implementation **simpler**, though it can lead to some redundant or unnecessary checking of values on later variable assignments that can be eliminated with forward checking and other techniques. However, your own implementation **does not need to modify the CSP formulation during search**, and in fact I would **discourage** you from attempting this – implementing inference during search requires some additional book-keeping and is not part of this assignment. (If you want to undertake the challenge of adding inference during search, then make sure you have a backup of your working basic implementation in case things break!)
- For variable selection and value ordering, it is **strongly recommended** that you implement the `static` options first and **thoroughly test** your program before attempting to implement the other variable selection and value ordering heuristics (i.e., don't worry about these heuristics until you have a basic backtracking solver that works!).
- In your implementation, you may find that certain operations need to be done several times in close proximity, particularly when you add in advanced heuristics. Don't worry too much if your program ends up repeating the same work a few times; first write your code to do the correct things, and then figure out if it makes sense or is even necessary to try to improve performance. Remember: "Premature optimization is the root of all evil."