CS:4420 Artificial Intelligence
Spring 2019

Course Project

Due: Friday, May 10 by 11:59pm

Project Policies

Each team can choose a project among those listed in the following sections. Since some projects are a bit more challenging than others, choosing them will earn you a number of bonus points (out of 100), as indicated for each project.

All projects involve developing an AI tool *and* evaluating it experimentally as specified below. *The tool must be developed in OCaml.* If you wish, you may use your scripting language of your choice for auxiliary tasks such as generating test sets, running experiments with the tool, collecting and formatting results, and so on. More specific instructions on how to structure your source code may be posted on Piazza later.

Each team can determine internally how to divide the project work among its members. However, in addition, it must officially designate:

- a *coordinator* to organize work sessions, make sure everyone knows where and when to meet and understands who is supposed to be doing what;
- a *recorder* to assemble and turn in the project material;
- a *checker* to check the material to be submitted for correctness and verify that everyone in the group understands both the solutions adopted and the strategies used to obtain them.

Each team member must have one of the roles above. In teams of two, one person will take two roles. Your project report should document who played which role.

All team members are personally responsible for the good functioning of the team and for resolving internal conflicts, especially in the case of non-participating team members. All students will be asked to submit independently an evaluation of how well they and their teammates performed as team members. Each evaluation is confidential and will not be shown to other team members. The evaluations will be incorporated into the calculation of an individual project grade for each team member. Instructions on one to submit evaluations will be posted later on Piazza.

Cheating policy. *Check the syllabus about our cheating policy. Team members are expected to make sure that both they* ***and their teammates*** *comply to the policy. Any verified instance of cheating will result in disciplinary actions against* ***all*** *members of the team.*

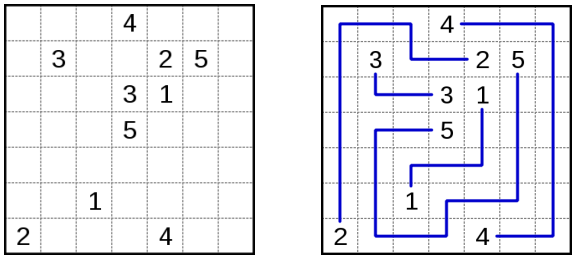

Figure 1: A Numberlink problem and a corresponding solution.

What to submit. Each team must turn in the following material at the end of the project.

1. A *technical report in PDF format*, describing the project and its results. The report should follow as appropriate A. Sherman and C. Brown's general guidelines for writing tech reports.[1] The report should describe experimental results as instructed below and also provide any conclusions you drew from those results.

2. *All the source code* developed in the project, with detailed documentation and appropriate README files, so as to allow a third party to understand, install and run your code and rerun your experiments.

3. The set of *all the data files* used in the experimental evaluation.

All the material must be submitted through ICON as usual but as a zip archive of a single folder called `ProjectMaterial`. *Only the team recorder should submit.*

Project 1: The Numberlink Puzzle (0 bonus points)

Numberlink is a pencil-and-paper puzzle based on an $m \times m$ grid. The puzzle starts with n pairs of distinct marks placed on arbitrary cells of the grid. The marks can be numbers, letters, colors, etc. The goal of the game is to connect two identical marks with a single continuous line on the grid. Lines cannot cross each other or themselves and cannot split. Also, they cannot go through a mark and every line must begin and end with the same mark. See Figure 1 for an example problem and solution which use numbers as marks.

One version of the puzzle considers as acceptable problems only those that have a unique solution which, in addition, is such that all cells are covered (as in Figure 1). A more relaxed version, which we adopt here, admits any initial grid configuration with n pairs of distinct marks and does not require the lines fill up the grid. This version makes it easier to generate problems although some of them may be unsolvable.

Develop a program to solve Numberlink puzzles automatically. The program should take a problem as input and print on the standard output a solution, if there is one, or the string `UNSOLVABLE`, otherwise. An input problem is a text file of the form exemplified below, where the first line is a numeral m for the width of the grid, followed by m lines of m characters each.

[1] Available at `http://www.cs.rochester.edu/u/brown/Crypto/writing/TR_how_to.html` .

```
7
000D000
0C00BE0
000CA00
000E000
0000000
00A0000
B000D00
```

```
BBBDDDD
BCBBBED
BCCCAED
BEEEAED
BEAAAED
BEAEEED
BEEEDDD
```

sample input sample output

Possible characters for the input file are `0` (zero) for the empty cell and letters for the marks. Each letter in the input file must occur exactly twice for the input to be well-formed. The output must be printed on the standard output in the form above where the line connecting two occurrences of a letter is represented by the same letter, and any empty cells are still represented by `0`.

Part 1 Develop your program using greedy best-first search and A* search to find solutions. Design your representation of states and operators along the lines of what we have studied in the search chapter. Use the sum of the lengths of all lines in a state s as the actual cost $g(s)$ of s. Choose or design a heuristic function h for the two search strategies, making sure it is admissible.

Test the performance of A* with each of the two strategies by running them on a number of test cases. Generate your own test cases for different grid sizes. For each value of m from 8 to 12 generate randomly at least $5m$ problems with an $m \times m$ grid. Each problem should contain n letter pairs, with n chosen randomly between $(m-3)$ and m. The letters should be put on the grid also randomly, with at most one letter per cell. Choose a reasonable time limit for all test cases (at least 100 seconds per test).

Run your experiments and collect statistics so that you can provide the following information in your report.

1. For each of the two search strategies, indicate the number of solved problems, unsolvable problems, and problems that timed out.

2. For each value of m from 10 to 12, also provide these aggregate statistics for the solved problems with a $m \times m$ grid: average number of expanded nodes, average cost of the solution, average depth of the solution in the search tree, the effective branching factor of the tree, and the average actual running time on the testing machine

Hint: To help you produce correct code, also write a separate program that takes two text files, one with Numberlink problem and one with a solution for it, and checks that the problem is well formed and the solution is indeed a solution of that problem. Use this program to validate your randomly generated problems and their corresponding solution, if any.

Part 2 We expect that A* will not be effective for large enough problems as it will require too much memory. Implement one of the improvements of A* described in the textbook (IDA*, SMA*). Using a heuristics of your choice, run a number of test cases to see how large a problem you can solve. Use the experience you have gathered in Part 1 to fine-tune your implementation so that it

is as scalable as possible. Report the most difficult problem (in terms of solution depth) that your implementation can solve in 10 minutes.

Note: In both Part 1 and Part 2, except for (i) the data structures that implement a state, (ii) the heuristic function, and (iii) the procedures that manipulate states, your implementation of the search procedure should be independent from the specific domain.

Project 2: The Numberlink Puzzle Reloaded (5 bonus points)

Consider the puzzle described in Project 1 but model it as a constraint satisfaction problem and develop a CSP solver to solve instances of the puzzle. Use the same input/output behavior and format as in Project 1.

Part 1 Generate input problems as described for Project 1. For each input problem, start with an initial assignment for the variables that corresponds to the initial grid configuration. Compute a complete assignment that corresponds to a solution of the problem.

Implement your solver by using some of the CSP techniques described in Chapter 5 of the textbook. Start with a simple backtracking algorithm. Then improve on it by adding heuristics for variable and value ordering. After that, add constraint propagation techniques (corresponding to function INFERENCE in Figure 6.5 of the textbook) such as some appropriate form of arc-consistency. Your propagations should reflect the general constraints for this puzzle — lines cannot cross, etc. If feasible, also implement conflict-directed backjumping. For this part, do not aim at finding a least-cost solution; computing any solution is enough.

Run your experiments and collect statistics so that you can provide the following information in your report.

1. For each improved version of your CSP solver, indicate the number of solved problems, unsolvable problems, and problems that timed out.

2. For each value of m from 10 to 12, also provide these aggregate statistics for the solved problems with a $m \times m$ grid: average number of guesses, average number of domain values removed from consideration by constraint propagation, average number of domain values removed from consideration by conflict-directed backjumping (if implemented), and the average actual running time on the testing machine

Part 2 Modify your program so that it is able to generate the least-cost solution if a problem is satisfiable, where the cost of a solution state s is again the sum of the lengths of all lines in s. You can do that by adapting to your CSP implementation the main idea in *branch-and-bound* search.[2]

During the search, maintain a global value b that is a strict upper bound on the cost of the optimal solution. Modify the solver so that, once it finds a solution s for the problem with cost $c < b$, it records that solution and continues the search using c as the new global upper bound. Doing this will force the solver to go through all the solutions, if any, while keeping track of the least-cost solution found so far. Once the search is over, the solver prints the current solution if it has one and prints `UNSOLVABLE` otherwise. Use the bound to cut search space as well, as follows.

[2] See `https://en.wikipedia.org/wiki/Branch_and_bound` .

Add another inference to the INFERENCE procedure that removes a value v from the current domain of a variable x if the cost of the successor state obtained by assigning v to x is b or greater. This way, the solver will never move to a state whose cost is certainly worse than the optimal solution cost. Note that for this improvement to be correct your formulation of the puzzle must be such that giving a value to an unassigned variable produces a state with higher cost.

Run the this optimizing solver on the same test set you used earlier, and report for it the same results as in Part 1.

Project 3: Bayesian Networks (10 bonus points)

For this project it is helpful to have already some familiarity with lexer and parser generators such as Lex and Yacc/Bison. You can refer to the OCaml website for materials on how to parse structured text in OCaml[3] and how to build lexers and parsers in OCaml using ocamllex and ocamlyacc, the OCaml versions of Lex and Yacc, respectively.

You will implement and compare experimentally some of the inference methods for Bayesian networks discussed in Chapter 14 of the textbook. You will run your experiments on a selection of the Discrete BNs available from this website: `http://www.bnlearn.com/bnrepository/`. The networks there are grouped by size. Choose at least one network per group except for the groups with more than 100 nodes. The networks are available in various formats. Choose the one that you find easier to parse.

For each network, write a variety of queries, at least 5 per network, with various combination of query and evidence variables. Adopt a text format of your choice to express a query.

Part 1 Write a program that takes as input two file names, one for a file containing a network and one for a file containing a query for that network, and prints on the standard output the probability table corresponding to the query plus additional information necessary to collect the statistics below.

The program should implement the Enumeration and the Variable Elimination methods from the textbook, and use one or the other based on the value of an input flag. Run experiments with a reasonable timeout (at least 3 minutes per query) and collect statistics so that you can provide the following information in your report. For each of the two methods indicate:

1. the total number of solved queries and of queries that timed out;
2. the total number of queries solved by one method but not by the other;
3. the average solving time over the solved queries.

Part 2 Extend your program so that it also implements the inexact inference method Likelihood Weighting. Run the methods for different values (say, 500, 1000, and 2000) for the number N of samples. Collect and report the same statistics as in Part 1 but for this method and for each value of N.

Analyze your data and provide a critical assessment of the strengths and weakness of each the four implemented methods based on your experimental evaluation.

[3] See `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html` and `https://ocaml.org/learn/tutorials/`.

Project 4: Decision Tree Learning (15 bonus points)

You will implement and compare experimentally some of the learning methods for decision trees discussed in Chapter 18 of the textbook. You will run your experiments on a selection of the datasets available from the Machine Learning Repository at: `http://archive.ics.uci.edu/ml/datasets.php`. The datasets can be filtered according to several criteria. Choose 4 multivariate, classification datasets with "Categorical" attribute type and 100 or more examples. Each of these datasets is stored in a text file that consists of one example per line. Each example is a comma-separated sequence of values for the problem attributes. The attributes are documented in a companion file with extension `.name`.

Part 1 Write a program DTL that implements the basic decision tree learning algorithm. The program should take as input (i) a 4-valued flag specifying the data model of one of the 4 datasets to have identified and (ii) the name of a file containing a dataset for the specified data model. For simplicity you can hard code the 4 data model in your program. Information on a data model, specifically the attributes used in the dataset as well as their domains is available in human-readable format in the `.name` files in the ML Repository. The DTL program should print on the standard output a textual representation the decision tree it learned. You can choose any suitable textual format for the tree (for instance, s-expressions or JSON).

Write a separate classifier, a function that takes two file names as input: one for a file containing a decision tree and one for a file containing a corresponding dataset. It should print on the standard output a classification for each example in the dataset according to the input decision tree. Note that the classifier will need to parse the decision tree. You can use publicly available libraries to do that if you adopt s-expressions or JSON. Alternatively, you can write your own parser (see previous project).

Hint: You can use the classifier to do a sanity-check on your decision tree learning function by running a learned tree over the same dataset used to construct it, and verifying that it classifies the examples correctly.

Part 2 Parametrize your implementation of the decision tree learning procedure by a value d for the maximum depth of the learned tree. The new procedure should now force a newly generated tree node to be a leaf if that node is at depth d in the tree (similarly to when there are no more attributes to test on). Use the new procedure to implement a separate program performing model selection as explained in Section 18.4.1 of the textbook where, however, you use tree depth instead of *size*. Use $k = 4$ for k-fold cross validation.

For two of the datasets you used in Part 1, run the model selection program for depth values from 1 to 10 and report the error set on the training data and the validation data as done in Figure 18.9 of the textbook, indicating which model was selected in each case.