Massachusetts Institute of Technology

16.412J/6.834J Cognitive Robotics

Problem Set #2 Due: in class Wed, 3/9/05

Background

In order for cognitive robots to act responsively, intelligently and safely within the real world, they must be able to reason at lightening speeds. Towards this end, performance evaluation of novel reasoning algorithms is used as an important measure of progress. In particular, an important requirement for any publication of a novel algorithm is a careful performance analysis compared to the state of the art. Likewise, many reasoning disciplines have developed the practice of publishing survey evaluation articles or by running competitions, such as the AI Planning competition, to evaluate performance. These evaluations are facilitated by sets of common benchmarks, many of which are publicly available over the web.

In these evaluations a number of performance metrics are used, depending on the properties of the algorithms. However, the most common are speed and memory usage. Whenever possible, processor independent metrics are used for evaluation. For example, the speed of a suite of systematic search algorithms might be measured in terms of the number of search nodes expanded, and memory in terms of the maximum number of search nodes placed on the search queue at any point in time. In other cases, the algorithms being compared are sufficiently different that the actual processor speed and memory usage are the only common denominators. To gain insight, performance is often measured as a function of a range of parameters that summarize certain characteristics of the problem being solved. For example, for constraint satisfaction problems, a common parameter is the ratio of constraints to number of variables that appear in the problem.

Objective

A recently emerging category of reasoning algorithm that has not been well characterized is the set of algorithms for solving finite domain constraint optimization problems (COP) – in particular valued constraint satisfaction problems (valued CSPs) and optimal CSPs. The objective of this problem set is for the *class to collectively evaluate* a set of three representative algorithms from this category, and to assemble a *survey article of the results.* The three algorithms to be evaluated are:

Algorithm A: Conflict directed A* Search
Algorithm B: Branch-and-bound using Russian Doll Search
Algorithm C: Decomposition and Dynamic Programming

Provided below are descriptions of the algorithm variants that we are asking you to evaluate, and relevant references for those algorithms. Each algorithm is to be evaluated against a set of benchmarks provided below. These benchmarks include both randomly generated and real world problems of varying levels of complexity.

Logistics

Each of you will be responsible for implementing one (and only one) of the algorithms, (A, B or C), which you will do together with a partner, unless you choose to implement the algorithm alone. Each team will then work with one other team to write up an evaluation of the relative performance of the two algorithms on the common benchmarks.

Please choose a partner with whom you want to work with, in order to implement one of the three algorithms (A, B, or C). We plan to have roughly two teams implement each algorithm, and will make the assignment of algorithms to teams next Wednesday (Feb 23rd). In class next week we will ask you to provide us the name of your partner and any preference you have of the algorithm you implement. Please review the material below and the references provided to help you to make your decision.

For the implementation, you are free to choose your favorite programming language. For one of the algorithms, Algorithm A, we provide a software component that is written in C/C++. Hence those teams responsible for Algorithm A will need to be familiar with C/C++ and/or interfacing to C++.

In order to compare the relative performance of the algorithms, we will form “super-teams,” by pairing each team working on an algorithm (e.g., A) with another team working on a different algorithm (e.g., B).

Algorithm Evaluation

Each team of two students should evaluate its implementation on each of the example problems described at the end of this document. Note that it may not be possible to solve every example problem with your algorithm. For each example that your approach could solve, indicate the value of the optimal solution, the required run-time and memory usage. In addition to absolute processor time and memory usage, please come up with processor independent measures of performance (e.g., in terms of search tree nodes) and report your algorithm’s performance in terms of those units. For each example that could not be solved by your implementation, indicate whether the program ran out of time or ran out of memory. You may terminate the program on any example that takes more than 10 minutes (600 seconds) to solve.

Each “super-team” will write up an evaluation comparing their two algorithms against each other (e.g., A vs. B).

Materials Submitted on Due Date

On the due date, each super-team (that is, group of four students) should submit to the course secretary, Brian O' Conaill (oconaill@Mit.edu), a hardcopy report, and an electronic version of the report, according to the guidelines specified below.

In addition, each super-team should submit an electronic copy of the implementation of their two algorithms, including a short summary of how to run the algorithms.

Content of Your Reports

Each super-team should submit one report. Your report should include a pedagogical description of each algorithm implemented, including a simple walked through example for illustration. This should be followed by an analysis of your benchmark results of the two algorithms.

Each pair is responsible for writing up the description of their respective algorithm, and for analyzing the performance of their algorithm on the benchmarks. The collective super-team is responsible for the comparison of the two algorithms. To give us a sense of the contribution that each team member has made, please indicate the author or authors of each section of the report, and please provide a summary of the contribution each team member makes to the implementations.

More specifically, each report should consist of the following parts:

Part 1. Describe the two algorithms. Explain the key ideas underlying each algorithm and provide pseudocode for each algorithm. Explain how the two algorithms work by walking through a small, pedagogical example.

Part 2. Briefly describe your implementations, and show a sample run of your programs on an example. The sample run should show the optimal solution, and some meaningful intermediate steps.

Part 3. In tabular form, report the results of running the two algorithms on each benchmark problem. If the algorithm halted, report the value of the optimal solution and the run-time, else indicate whether the algorithm ran out of time or out of memory (see the above section on "algorithm evaluation" for further guidelines).

Part 4. Discuss your results: Which algorithm performed better on which problems? Identify properties of the problems that seem to positively or negatively affect the performance of each algorithm. What do you think are the strengths and weaknesses of one algorithm over the other? How difficult would it be to extend them to generate the next best solution, or all solutions? Can you think of possible extensions or improvements?

Part 5. Summarize each team member's contribution to the implementation and evaluation of the two algorithms.

Appendix. In an appendix to your report, provide a print out of the source code of your programs.

Algorithm A: Conflict-directed A* Search

Implement conflict-directed A* (CDA*) to compute optimal solutions for valued constraint satisfaction problems (VCSPs).

In order to apply CDA* to a VCSP, you first need to write a compiler that maps a VCSP into an optimal constraint satisfaction problem (OCSP). This may be accomplished by applying the transformation shown in lecture by Martin Sachenbacher, and involves introducing a decision variable for each soft constraint.

Next, implement CDA* by using a search queue to enumerate the assignments to the decision variables in best-first order, testing complete assignments for consistency, and using conflicts obtained from inconsistent assignments to guide the expansion of the search tree.

CDA* as described in the paper by Williams and Ragno optimizes child expansion by expanding only the best child and next best sibling of a search node, rather than expanding all of its children at once. If you would like, you can simplify your implementation of the algorithm by omitting this improvement and expanding all children of a search node at once (however, we would be delighted if you implemented the original algorithm).

In order to facilitate implementation of the algorithm, we provide a component, called ISAT, which tests assignments for consistency with a set of constraints given as clauses in propositional state logic, and returns a conflict in the case of inconsistency. Propositional state logic is a propositional logic in which all propositions are assignments to a set of finite domain variables. Note that ISAT is written in C++, hence your implementation of CDA* will either need to make a foreign function call to ISAT or you will need to implement CDA* in C++. You can obtain a CD containing the C/C++ code of ISAT and a description of how to use ISAT from the course secretary, Brian O'Conaill, in Room 33-336. Should you have any further questions regarding ISAT, please do not hesitate to direct your question to Shen Qu, jing2qu@MIT.EDU.

Further reading:

- Brian C. Williams and Robert Ragno: Conflict-directed A* and its Role in Model-based Embedded Systems. To appear in the Special Issue on Theory and Applications of Satisfiability Testing, *Journal of Discrete Applied Math* (http://mers.csail.mit.edu/abstracts/jdam.html)

Algorithm B: Branch-and-Bound using Russian Doll Search

Implement the Russian Doll Search (RDS) algorithm by Verfaillie, Lemaitre and Schiex for solving valued constraint satisfaction problems (VCSPs).

RDS is a branch-and-bound algorithm that replaces one search by *n* searches on successively larger subproblems (*n* is the number of variables in the problem). RDS records the result for each subproblem, and uses it to improve the lower bounds of partial assignments when solving the next largest subproblem.

Further reading:

- Gerard Verfaillie, Michel Lemaitre and Thomas Schiex: Russian Doll Search for Solving Constraint Optimization Problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-96),* pages 181-187, 1996. (http://citeseer.ist.psu.edu/article/verfaillie96russian.html)

- Rina Dechter: *Constraint Processing*. Morgan Kaufmann Publishers, 2003. Chapter 13 (Constraint Optimization)

Algorithm C: Decomposition and Dynamic Programming

Implement an algorithm that solves valued constraint satisfaction problems (VCSPs) by decomposing the problem into a bucket tree, and computing optimal solutions by performing dynamic programming on the tree.

In order to decompose the problem into a bucket tree, represent the network of constraints in the problem as a graph, such that each variable corresponds to a graph node, and each constraint corresponds to a graph hyper-edge that connects all of the variables that appear in the constraint. Next use the greedy min-fill (MF) heuristics described in the lecture by Martin Sachenbacher and in the book by Dechter to compute a bucket tree from the constraint graph.

To perform dynamic programming on the tree, implement a message-passing scheme that processes the tree from bottom to top, combines the constraints in each tree node, projects them on to the variables shared with its parent node, and sends the result to the parent node (this algorithm is called cluster-tree elimination (CTE) in the paper by Kask, Dechter and Larrosa below). Output the value of the optimal solution as the value of the best assignment in the root node of the tree.

Further reading:

- Kalev Kask, Rina Dechter and Javier Larrosa: Unifying Cluster-Tree Decompositions for Automated Reasoning, University of California at Irvine Technical Report, 2003 (http://www.ics.uci.edu/~dechter/publications/r109.html)

- Rina Dechter: *Constraint Processing*. Morgan Kaufmann Publishers, 2003. Chapter 9 (Tree Decomposition)

Benchmark Problems

On the course website (http://web.mit.edu/16.412J/www/html/), we provide a repository of benchmark examples to run your algorithm on. All files are given in the WCSP format, which is a simple, low-level format for instances of Valued Constraint Satisfaction Problems (VCSPs), where a cost is associated with each tuple of a constraint, and the goal is to find a complete assignment with minimum cost. For a description of the format, see:
http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/CpWcspFormats

- Folder "Academic" contains some small, academic problems (n-queens problem, crypto-arithmetic puzzles, and zebra puzzle):

    - 4wqueens.wcsp (4 variables, 10 constraints)
    - 8wqueens.wcsp (8 variables, 36 constraints)
    - 16wqueens.wcsp (16 variables, 136 constraints)
    - donald.wcsp (15 variables, 51 constraints)
    - send.wcsp (11 variables, 32 constraints)
    - zebra.wcsp (25 variables, 19 constraints)

- Folder "Random" contains randomly generated problems, which are sorted into four different problem classes according to the density of the constraint network and the tightness of the constraints (sparse/loose, sparse/tight, dense/loose, and dense/tight):

    - Vcsp40_10_13_60_1-5.wcsp (40 variables, 100 constraints) (5 instances)
    - Vcsp25_10_21_85_1-5.wcsp (25 variables, 64 constraints) (5 instances)
    - Vcsp30_10_25_48_1-5.wcsp (30 variables, 109 constraints) (5 instances)
    - Vcsp25_10_25_87_1-5.wcsp (25 variables, 75 constraints) (5 instances)

- Folder "Celar" contains real-world radio link frequency assignment problems arising in wireless communication networks (courtesy for these examples is Centre d'Electronique de l'Armement, France). For further documentation, see

http://fap.zib.de/problems/CALMA/
http://www.inra.fr/bia/T/schiex/Doc/CELAR.shtml

- CELAR6-SUB0.wcsp (16 variables, 207 constraints)
- CELAR6-SUB1-24.wcsp (14 variables, 300 constraints)
- CELAR6-SUB2.wcsp (16 variables, 353 constraints)

- Folder "Dimacs" contains real-world circuit fault analysis examples from the Second Discrete Mathematics and Computer Science (DIMACS) challenge. For further documentation, see
  http://mat.gsia.cmu.edu/challenge.html
  http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/DIMACS/BF/descr.html

  - ssa0432-003.wcsp (435 variables, 1027 constraints)
  - ssa2670-141.wcsp (986 variables, 2315 constraints)
  - ssa2670-130.wcsp (1359 variables, 3321 constraints)