50.021 -AI

Alex

Week 08: Search

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Constraint satisfaction problems

Start situation: see map coloring example on slide 5. We have variables that can take some values. Goal is to find an assignment $var_1 = value_1$, $var_2 = value_2$,$var_3 = value_3$,$var_4 = value_4$,$var_5 = value_5$ – which satisfies some constraint. For map coloring the constraints are inequality between neighboring variables, e.g. $NSW \neq V$ - which means that the color of New South Wales is not equal to that of Victoria.

What is a CSP?

- a special case of problems solvable by search
- we have variables, each variable can take values from some domain. goal: find an assigment of values for all variables that satisfies constraints
- its special structure allows for efficient algorithms:
  - heuristics such variable ordering by minimum remaining value
  - state space reduction by forward checking
  - state space reduction by constraint propagation
  - efficient algorithms for tree-structured CSPs
  - ideas for nearly tree-structured CSPs

Formally a CSP is defined by:

- a set of variables X_i
- a domain of possible values D_i for every variable X_i
- a set of constraints between sets of variables $C_k(X_{i_1}, X_{i_2}, \ldots, X_{i_r})$ – here r variables (e.g. all the variables sum up to some constant value)

- goal is choosing for every variable a value from its domain such that all the constraints are satisfied.

A binary CSP is a CSP problem where every constraint involves only one or two variables such as $NSW \neq V$: $C_k(X_r)$, $C_k(X_r, X_s)$.

The constraint graph – see slide 6 – is a representation of the set of all constraints.

- nodes are the variables
- edges if two variables are part of a constraint (which can involve more than 2)

See slide 7 for discrete vs continuous variables.

Size of state space for discrete variables: if every variable has d possible values, and we have n variables, then d^n possible assignments.

See slide 10 for problems that can be solved. Assignment, time tabling can also involve factory planning - which machines to use for which job.

One theoretically important CSP is 3-SAT [1]: given n variables x_i find an assignment such that a conjunctive normal form with at most 3 variables in every clause is satisfied.

[1] it is NP-complete, NP means: all decision problems whose solutions can be verified in polynomial time

$$(x_1 \vee \neg x_2 \vee \neg x_5) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge \ldots$$

a conjunctive normal form with at most 3 variables in every clause is a concatenation of logical expressions by an "and" ($\wedge$) such that every logical expression contains at most 3 variables x_i, or their negation $\neg x_i$, joined by and "or" $\vee$

solving CSP by search

simplest idea: root node contains state of nothing assigned, can be represented by an empty set $\{\}$. expand a node: choose an unassigned variable, generate child nodes: one child node for every possible value assigned to that variable.

Observations:

1. n variables ... all are assigned at depth n. So all possible solutions lie at depth n.
2. can use DFS – why no gain in using BFS?

- naive approach: at depth k can choose $n-k$ variables for assignment, and each of them may have d values, so $n!d^n$ leaves
- can simplify: can consider in one node only one variable to be assigned. d^n leaves.
- CSPs are commutative: the solution is the same, no matter what the order in which we assign values to variables.
- This gives one freedom of choice: we can choose an order which variable to be assigned first, and which last. Example: at depth 1 we assign values to variable C, at depth 2 to variable F, at depth 3 to variable A ...
- another freedom of choice: given a variable, which values to be assigned should be explored first in DFS?

vanilla backtracking search - see slide 13

Backtracking means: when DFS found no solution and it walks up the tree backwards - this means that assignments done $\{var = value\}$ have to be removed when one walks up the tree – example on slide 21 where there is 0 value left for SA

Idea is recursive:

- select an unassigned variable (1st freedom above),
- select an order of values (2nd freedom above)
- then for every value: initialize variable to value, call recursive routine with this new assignment
- if recursive routine returns with a failure, then remove assignment and return a failure

An example for Capability: n-Queens with $n \approx 25$ (?)

Example slide 13

Lets explore the freedom - General purpose methods improve search speed by looking at the following questions:

- which variable should be assigned next?
- given a variable, in what order we should explore the possible values?
- can we detect unavoidable failure early?

some heuristics

order of variables:

MRV heuristic: slide 19: MRV – minimum remaining value: choose the variable first which has the least number of values in its domain which can be assigned.

why it works? We will learn soon: there are ways to check whether the domain of a variable can be pruned/shrunk based on what variables have already assigned values.

if one variable has no possible values left, then MRV will pick it and produce an early failure.

if one variable has only one value left, then we should assign it right away, and not wait, as this assignment is for free. if assigning this single variable forces a failure in constraints, then it will do so no matter how we choose all other variables. so we can have hope to sometimes skip trying out many possible values for other variables.

if MRV has a tie, then one does often a tiebreak by: **degree heuristic**: how many other variables are affected by this variable through constraints - choose the variable with the largest degree, the one which affects most others at the same time by having constraints

Given one variable, what order to choose? **LCV - least constraining value** slide 21: - the value first that rules out the fewest values (fewest values ruled out, highest chance to find a solution when one has no extra knowledge)

An example for Capability: n-Queens with $n \approx 1000$ (?)

state space reduction - forward checking

Idea to reduce the search space: when one variable X got assigned a value: $X = x$, you can check all constraints $C(X, Y)$ between this variable X and all other variables Y that are yet unassigned: can remove all values from $dom(Y)$ that are conflicting $X = x$. Can help to detect failures early – if all states in $dom(Y)$ are conflicting $X = x$, then no solution possible, can backtrack right away.

Compare to vanilla backtracking, where one could assign and try out values for variables $Z_1, Z_2, \dots, Z_n$.

slides 22-25: on slide 25: one knows its a failure without assigning values to T.

state space reduction - constraint propagation

forward checking propagates information only from an assigned variable to unassigned ones. but one can do more: see slide 26

what happened on slide 26?

suppose one assigns $X = x$, now one can prune the domain of an unassigned variable Y, and remove some possible values. But pruning changed the domain of Y - this might now affect other unassigned variables Z, if they have constraints with Y!!

So one can check now whether one can prune domain of Z because the domain of Y was pruned, and both have a constraint.
Lets formalize this. Arc consistency- see slide 27.

if $X \to Y$ is not arc-consistent. what does it mean?

not for every value x of X exists some allowed value y for Y.

negation and quantors:

$$\neg\forall x: \; A(x) \Leftrightarrow \exists x: \; \neg A(x)$$
$$\neg\exists x: \; A(x) \Leftrightarrow \forall x: \; \neg A(x)$$

$\to$ there exists some value x of X such that the following statement is false: "exists some allowed value y for Y"

$\to$ there exists some value x of X such that there is no allowed value y for Y

but then we can remove that value x from the domain $dom(X)$!!

Slides: 27-30 gives an application example.

so enforcing arc consistency is: removing values that can never lead to a solution.

Technical idea: maintain queue of variables where domains have changed lately – bcs if a variable domain has changed, then it makes sense to check all variables that are neighbors/have constraints to it.
see slides 27-29

Caveat: after "removing values that can never lead to a solution" for variable X, the domain $dom(X)$ has been shrunk. This can again affect any variables Y that have a constraint with X (maybe we can remove values from y now that can not be satisfied by the remaining values in X). so typically arc-consistency checks are run until no domain changes anymore. if a domain changes, one must check all neighbors in the graph.

slide 31 – AC-3: an arc consistency algorithm for binary CSP. Remove-first means just: pop off element off the queue.

analysis of connected components

slides 32-32: a graph can be broken down into connected components, and they can be solved independently

tree-structured CSPs

see slides 34:

definition: a CSP is tree-structured if its constraint graph has no cycles.

tree-structured CSPs allow for an efficient solution in $\mathcal{O}(nd^2)$ - see slide 35

the backward pass of the algorithm removes for every edge the inconsistent values in the parent.

Why that rocks? Now: no matter what we choose in the parent, there is always a value in the child so that the constraint is satisfied.

In the forward pass: we can never get stuck, we will never backtrack! Never need to try out in vain.

nearly tree-structured CSPs

Question: is the graph such that one can remove k nodes and the result is tree structured?

if yes: try out for the k nodes all possible sets of assignments (variable ordering!!!), then solve the remaining tree structured CSP.

Local search Algorithms - min conflicts

Alternative to the algorithms seen so far:

Initialize all variables with random values. Compute a heuristic how close one is to the goal, try local search for values for variables.

A use case where local search is favorable: when constraints are changing, e.g. flight plans and other schedules. Local search can start from the current solution in the hope that one can limit changes to the current plan!

Min-conflicts algorithm: What works often well: Choose a random variable that is involved in conflicts, select a value as assignment such that the number of conflicts/violated constraints is minimized for this value.

other Modifications:

Tabu-search: keep a list of recently visited states, forbid to go there back.

prevent oscillations

Constraint weighting:

- each constraint is initialized with weight 1
- loss of current assignment: sum of weights for all violated constraints
- apply some heuristic to find a pair of variable and value, so that if this variable gets this value, then the loss will be small (approximately minimized)
- after assignment: increase weight of every violated constraint
- helps to find and focus on constraints which are hard ... these will get high weights over time.