Constraint Satisfaction Problems (CSPs)

Introduction and Backtracking Search

This lecture topic (two lectures)
Chapter 6.1 – 6.4, except 6.3.3

Next lecture topic (two lectures)
Chapter 7.1 – 7.5

(Please read lecture topic material before and after each lecture on that topic)
Outline

- What is a CSP?
- Backtracking Search for CSP
- Variable selection (ordering)
  - Minimum Remaining Values (MRV) heuristic
  - Degree Heuristic
- Value selection (ordering)
  - Least Constraining Value (LCV) heuristic
You Will Be Expected to Know

- Basic definitions (section 6.1)

- Backtracking search (6.3)

- Variable ordering or selection (6.3.1)
  - minimum-remaining values
  - degree heuristic

- Value ordering or selection (6.3.1)
  - least-constraining-value
Constraint Satisfaction Problems

- What is a CSP?
  - Finite set of variables $X_1, X_2, \ldots, X_n$
  - Nonempty domain of possible values for each variable $D_1, D_2, \ldots, D_n$
  - Finite set of constraints $C_1, C_2, \ldots, C_m$
    - Each constraint $C_i$ limits the values that variables can take,
    - e.g., $X_1 \neq X_2$
  - Each constraint $C_i$ is a pair <scope, relation>
    - Scope = Tuple of variables that participate in the constraint.
    - Relation = List of allowed combinations of variable values.
      May be an explicit list of allowed combinations.
      May be an abstract relation allowing membership testing and listing.

- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain specific expertise).
Sudoku as a Constraint Satisfaction Problem (CSP)

• Variables: 81 variables
  – A1, A2, A3, …, I7, I8, I9
    – Letters index rows, top to bottom
    – Digits index columns, left to right

• Domains: The nine positive digits
  – A1 ∈ \{1, 2, 3, 4, 5, 6, 7, 8, 9\}
    – Etc.

• Constraints: 27 \textit{Alldiff} constraints
  – \textit{Alldiff}(A1, A2, A3, A4, A5, A6, A7, A8, A9)
    – Etc.
Random Binary CSP  
(adapted from http://www.unitime.org/csp.php)

- A random binary CSP is defined by a four-tuple \((n, d, p_1, p_2)\)
  - \(n\) = the number of variables.
  - \(d\) = the domain size of each variable.
  - \(p_1\) = probability a constraint exists between two variables.
  - \(p_2\) = probability a pair of values in the domains of two variables connected by a constraint is incompatible.
    - Note that R&N lists compatible pairs of values instead.
    - Equivalent formulations; just take the set complement.
  - \((n, d, p_1, p_2)\) are used to generate randomly the binary constraints among the variables.

- The so called model B of Random CSP \((n, d, n_1, n_2)\)
  - \(n_1 = p_1n(n-1)/2\) pairs of variables are randomly and uniformly selected and binary constraints are posted between them.
  - For each constraint, \(n_2 = p_2d^2\) randomly and uniformly selected pairs of values are picked as incompatible.

- The random CSP as an optimization problem (minCSP).
  - Goal is to minimize the total sum of values for all variables.
CSPs --- what is a solution?

- A *state* is an *assignment* of values to some or all variables.
  - An assignment is *complete* when every variable has a value.
  - An assignment is *partial* when some variables have no values.

- **Consistent assignment**
  - assignment does not violate the constraints

- A *solution* to a CSP is a complete and consistent assignment.

- Some CSPs require a solution that maximizes an *objective function*.

- Examples of Applications:
  - Scheduling the time of observations on the Hubble Space Telescope
  - Airline schedules
  - Cryptography
  - Computer vision -> image interpretation
  - Scheduling your MS or PhD thesis exam 😊
CSP example: map coloring

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D_i=${red,green,blue}
- Constraints: adjacent regions must have different colors.
  - E.g. $WA \neq NT$
CSP example: map coloring

- Solutions are assignments satisfying all constraints, e.g.
  \( \{WA=\text{red}, NT=\text{green}, Q=\text{red}, NSW=\text{green}, V=\text{red}, SA=\text{blue}, T=\text{green} \} \)
Graph coloring

- More general problem than map coloring

- Planar graph = graph in the 2d-plane with no edge crossings

- Guthrie’s conjecture (1852)
  
  *Every planar graph can be colored with 4 colors or less*

  - Proved (using a computer) in 1977 (Appel and Haken)
Constraint graphs

- Constraint graph:
  - nodes are variables
  - arcs are binary constraints

- Graph can be used to simplify search
  e.g. Tasmania is an independent subproblem

(will return to graph structure later)
Varieties of CSPs

• Discrete variables
  
  – Finite domains; size $d \Rightarrow O(d^n)$ complete assignments.
    • E.g. Boolean CSPs: Boolean satisfiability (NP-complete).

  – Infinite domains (integers, strings, etc.)
    • E.g. job scheduling, variables are start/end days for each job
    • Need a constraint language e.g. $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$.
    • Infinitely many solutions
    • Linear constraints: solvable
    • Nonlinear: no general algorithm

• Continuous variables
  
  – e.g. building an airline schedule or class schedule.
  – Linear constraints solvable in polynomial time by LP methods.
Varieties of constraints

- Unary constraints involve a single variable.
  - e.g. $SA \neq green$

- Binary constraints involve pairs of variables.
  - e.g. $SA \neq WA$

- Higher-order constraints involve 3 or more variables.
  - Professors A, B, and C cannot be on a committee together
  - Can always be represented by multiple binary constraints

- Preference (soft constraints)
  - e.g. red is better than green often can be represented by a cost for each variable assignment
  - combination of optimization with CSPs
CSPs Only Need Binary Constraints!!

- Unary constraints: Just delete values from variable’s domain.
- Higher order (3 variables or more): reduce to binary constraints.
- Simple example:
  - Three example variables, X, Y, Z.
  - Domains \(D_x=\{1,2,3\}\), \(D_y=\{1,2,3\}\), \(D_z=\{1,2,3\}\).
  - Constraint \(C[X,Y,Z] = \{X+Y=Z\} = \{(1,1,2), (1,2,3), (2,1,3)\}\).
  - Plus many other variables and constraints elsewhere in the CSP.

  - Create a new variable, \(W\), taking values as triples (3-tuples).
  - Domain of \(W\) is \(D_w = \{(1,1,2), (1,2,3), (2,1,3)\}\).
    - \(D_w\) is exactly the tuples that satisfy the higher order constraint.
  - Create three new constraints:
    - \(C[X,W] = \{ [1, (1,1,2)], [1, (1,2,3)], [2, (2,1,3)] \} \).
    - \(C[Y,W] = \{ [1, (1,1,2)], [2, (1,2,3)], [1, (2,1,3)] \} \).
    - \(C[Z,W] = \{ [2, (1,1,2)], [3, (1,2,3)], [3, (2,1,3)] \} \).
  - Other constraints elsewhere involving X, Y, or Z are unaffected.
CSP Example: Cryptharithmetic puzzle

\[
\begin{array}{c}
T \ W \ O \\
+ T \ W \ O \\
\hline
F \ O \ U \ R
\end{array}
\]

Variables: \( F, T, U, W, R, O, X_1, X_2, X_3 \)
Domains: \( \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)
Constraints
\[
alldiff(F, T, U, W, R, O) \\
O + O = R + 10 \cdot X_1, \text{ etc.}
\]
CSP Example: Cryptarithmic puzzle

\[
\begin{array}{c}
T W O \\
+ T W O \\
\hline
F O U R
\end{array}
\]

Variables: $F, T, U, W, R, O, X_1, X_2, X_3$
Domains: \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
Constraints
\[
\text{alldiff}(F, T, U, W, R, O)
\]
\[
O + O = R + 10 \cdot X_1, \text{ etc.}
\]
CSP Example: Cryptarithmetic puzzle

Variables: $F, T, U, W, R, O, X_1, X_2, X_3$
Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Constraints
- $\text{alldiff}(F, T, U, W, R, O)$
- $O + O = R + 10 \cdot X_1$, etc.

A Solution:
- $F=1$, $T=7$, $U=6$, $W=3$, $R=8$, $O=4$
- $X_1=0$, $X_2=0$, $X_3=1$

$$
\begin{align*}
7 & \quad 3 & \quad 4 \\
+ & \quad 7 & \quad 3 & \quad 4 \\
\hline
1 & \quad 4 & \quad 6 & \quad 8
\end{align*}
$$
CSP Example: Cryptarithmetmic puzzle

- Try it yourself at home:

\[
\begin{align*}
\text{S E N D} \\
+ \text{M O R E} \\
\hline
\text{M O N E Y}
\end{align*}
\]

- (A frequent request from college students to parents!)
CSP as a standard search problem

- A CSP can easily be expressed as a standard search problem.

- Incremental formulation
  - Initial State: the empty assignment {} 
  - Actions: Assign a value to an unassigned variable provided that it does not violate a constraint 
  - Goal test: the current assignment is complete (by construction it is consistent) 
  - Path cost: constant cost for every step (not really relevant) 

- Can also use complete-state formulation
  - Local search techniques (Chapter 4) tend to work well
CSP as a standard search problem

- Solution is found at depth $n$ (if there are $n$ variables).

- Consider using BFS
  - Branching factor $b$ at the top level is $nd$
  - At next level is $(n-1)d$
  - ....

- end up with $n!d^n$ leaves even though there are only $d^n$ complete assignments!
Commutativity

- CSPs are commutative.
  - The order of any given set of actions has no effect on the outcome.
  - Example: choose colors for Australian territories one at a time
    - \([\text{WA}=\text{red then NT}=\text{green}]\) same as \([\text{NT}=\text{green then WA}=\text{red}]\)

- All CSP search algorithms can generate successors by considering assignments for only a single variable at each node in the search tree
  \(\Rightarrow\) there are \(d^n\) leaves

\(\text{(will need to figure out later which variable to assign a value to at each node)}\)
Backtracking search

- Similar to Depth-first search, generating children one at a time.

- Chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

- Uninformed algorithm
  - No good general performance
Backtracking search (Figure 6.5)

```plaintext
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RECURSIVE-BACTRACKING(assignment, csp)
            if result ≠ failure then return result
        remove {var=value} from assignment
    return failure
```
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search

- Expand \textit{deepest} unexpanded node
- Generate \textit{only one} child at a time.
- \textit{Goal-Test} when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search

• Expand *deepest* unexpanded node
• Generate *only one* child at a time.
• *Goal-Test* when inserted.
  – For CSP, Goal-test at bottom
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future = green dotted circles
Frontier = white nodes
Expanded/active = gray nodes
Forgotten/reclaimed = black nodes
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search

• Expand *deepest* unexpanded node
• Generate *only one* child at a time.
• *Goal-Test* when inserted.
  – For CSP, Goal-test at bottom
Backtracking search

- Expand *deepest* unexpanded node
- Generate *only one* child at a time.
- *Goal-Test* when inserted.
  - For CSP, Goal-test at bottom

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Backtracking search (Figure 6.5)

**function** BACKTRACKING-SEARCH(csp)  **return** a solution or failure

  **return** RECURSIVE-BACKTRACKING({}, csp)

**function** RECURSIVE-BACKTRACKING(assignment, csp)  **return** a solution or failure

  **if** assignment is complete  **then**  **return** assignment

  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)

  **for each** value in ORDER-DOMAIN-VALUES(var, assignment, csp)  **do**

    **if** value is consistent with assignment according to CONSTRAINTS[csp]  **then**

      add {var=value} to assignment

      result ← RECURSIVE-BACKTRACKING(assignment, csp)

    **if** result ≠ failure  **then**  **return** result

      remove {var=value} from assignment

  return failure
Improving CSP efficiency

- Previous improvements on uninformed search → introduce heuristics

- For CSPS, general-purpose methods can give large gains in speed, e.g.,
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
  - Can we take advantage of problem structure?

Note: CSPs are somewhat generic in their formulation, and so the heuristics are more general compared to methods in Chapter 4
Backtracking search (Figure 6.5)

function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
Minimum remaining values (MRV) for next variable

\[ var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE(VARIABLES}[csp],assignment,csp) \]

- A.k.a. most constrained variable heuristic

- **Heuristic Rule**: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values
Degree heuristic for next variable

- **Heuristic Rule**: select variable that is involved in the largest number of constraints on other unassigned variables.

- Degree heuristic can be useful as a tie breaker after MRV.

- *In what order should a variable’s values be tried?*
**Backtracking search (Figure 6.5)**

```plaintext
function BACKTRACKING-SEARCH(csp) return a solution or failure
   return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment according to CONSTRAINTS[csp] then
         add {var=value} to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         remove {var=value} from assignment
   return failure
```
Least constraining value (LCV) for next value

- Least constraining value heuristic

- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments
Minimum remaining values (MRV) vs. Least constraining value (LCV)

- Why do we want the MRV (minimum values, most constraining) for variable selection --- but the LCV (maximum values, least constraining) for value selection?

- Isn’t there a contradiction here?

- MRV for variable selection to reduces the branching factor.  
  - Smaller branching factors lead to faster search.  
  - Hopefully, when we get to variables with currently many values, constraint propagation (next lecture) will have removed some of their values and they’ll have small branching factors by then too.

- LCV for value selection increases the chance of early success.  
  - If we are going to fail at this node, then we have to examine every value anyway, and their order makes no difference at all.  
  - If we are going to succeed, then the earlier we succeed the sooner we can stop searching, so we want to succeed early.  
  - LCV rules out the fewest possible solutions below this node, so we have the most chances for early success.
Summary

- CSPs
  - special kind of problem: states defined by values of a fixed set of variables, goal test defined by constraints on variable values

- Backtracking = depth-first search with one variable assigned per node

- Heuristics
  - Variable ordering and value selection heuristics help significantly

- Variable ordering (selection) heuristics
  - Choose variable with Minimum Remaining Values (MRV)
  - Degree Heuristic --- break ties after applying MRV

- Value ordering (selection) heuristic
  - Choose Least Constraining Value