Constraint Satisfaction Problems
Deepak Kumar
October 2017

Search problems so far

• Single agent
• Deterministic actions
• Fully observable state
• Discrete state space
1. Planning Problem

Given a problem, find a sequence of actions to go from start to goal:

Search(problem) returns \([action_1, action_2, ..., action_n]\)

The state is a Black Box
- problem.getStartState()
- problem.isGoalState()
- problem.getSuccessors()

Path to goal is important
- Paths can have costs/depths
- Heuristics help guide search

2. Identification Problem

- Assignments to variables
- Goal is important, not the path

N-queens - What is the placement of N queens so that all are safe?
Example 2: Map Coloring

- Given three colors: {red, green, blue}
- Color the map so that no two adjacent states have the same color.
Example 2: Map Coloring

- Given three colors: \{red, green, blue\}
- Color the map so that no two adjacent states have the same color.

Example 3: Cryptarithmetic

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

\[
\begin{array}{c|c|c|c|c}
S & E & N & D \\
9 & 5 & 6 & 7 \\
\hline
+ & M & O & R & E \\
1 & 0 & 8 & 5 \\
\hline
\text{M O N E Y} \\
9 & 5 & 6 & 5 & 2
\end{array}
\]
Example 4: Sudoku

```
  S E N D
+   M O R E
  ---------
  M O N E Y
```

Constraint Satisfaction Problems!!

```
  S E N D
+   M O R E
  ---------
  M O N E Y
```
CSP Formulation
(as a special case of search)

• State is defined by n variables
  \[ \{x_1, x_2, ..., x_n\} \]

• Variables can take on values from a domain set
  (One for each variable)
  \[ \{D_1, D_2, ..., D_n\} \]

• Goal test is a set of constraints specifying allowable combinations of values of
  variables (subsets)

• This allows general purpose algorithms without resorting to domain specific
  heuristics.

Example Formulation: N-Queens V1

• **Variables**: \(N^2\) variables \(x_{ij}\) one for each square

• **Domains**: all variables have same domain \(\{0, 1\}\)
  - 0 - no queen
  - 1 - queen

• **Constraints**

1. In a row, any two cells are either empty or exactly one has a queen
   \[ \forall i,j,k (x_{ij}, x_{ik}) \in \{(0,0), (0,1), (1,0)\} \]

2. In a column, any two cells are either empty or exactly one has a queen
   \[ \forall i,j,k (x_{ij}, x_{kj}) \in \{(0,0), (0,1), (1,0)\} \]

3. In a diagonal, any two cells are either empty or exactly one has a queen
   \[ \forall i,j,k (x_{ij}, x_{i+j+k, j+k}) \in \{(0,0), (0,1), (1,0)\} \]
   \[ \forall i,j,k (x_{ij}, x_{i-j-k, j-k}) \in \{(0,0), (0,1), (1,0)\} \]

4. There are exactly N queens
   \[ \sum_{i,j} x_{i,j} = N \]
N-Queens V2

• Variables: $Q_1, Q_2, \ldots, Q_N$

• Domains: $\{1, 2, 3, \ldots, N\}$

• Constraints:
  \[ \forall i,j \text{nonThreatening}(Q_i, Q_j) \]

• Solution (N=4): \{ $Q_1 = 2, Q_2 = 4, Q_3 = 1, Q_4 = 3$ \}

Example: Map-Coloring

• Variables: WA, NT, Q, NSW, V, SA, T

• Domains: $D_i = \{\text{red, green, blue}\}$

• Constraints: adjacent regions must have different colors

  e.g., $WA \neq NT$

  or

  $(WA, NT) \in \{ (\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green}) \}$
Example: Map-Coloring

• Solutions are complete and consistent assignments

\[
\{ WA = \text{red}, \\
NT = \text{green}, \\
Q = \text{red}, \\
NSW = \text{green}, \\
V = \text{red}, \\
SA = \text{blue}, \\
T = \text{green} \}
\]

• Complete – Every variable has a value assigned.

• Consistent – The assignment satisfies all constraints.

Constraint Graph Representation of CSP

• Binary CSP: each constraint relates two variables
• Constraint graph: nodes are variables, arcs are constraints
Start with a basic search algorithm...

**Initial State:** Empty assignment  \{ \}

**Successor Function:** assign a value to an unassigned variable

**Goal Test:** current assignment complete & consistent?

Which search algorithm to use??

Try Breadth-First Search

There are only $d^n$ complete assignments!
Most are not consistent!!
What about Depth-First Search?

DFS with these is called **Backtracking Search**

1. Pick one variable at a time.
2. Check constraints as you go.
   (incremental goal testing)

Will get to the bottom but will be spending a lot of time looking at failure options.
Backtracking Search

1. Pick one variable at a time.

2. Check constraints as you go.
   (incremental goal testing)

Backtracking Search Algorithm

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

function RECURSIVE-BACKTRACKING(assignment, csp) returns 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
```
Backtracking example
### Backtracking Search Algorithm

Can we detect inevitable failure?

Which variable to pick next?

Which value to assign next?

These are general purpose heuristics.

---

**Improving backtracking efficiency**

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
  - Can we exploit problem structure?

---

**Function BACKTRACKING-SEARCH**

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

function RECURSIVE-BACKTRACKING(assignment, csp) returns 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
        end if
    end for
    return failure
```
Forward checking

• Idea
  • Keep track of filtered domains (remaining legal values) for unassigned variables
  • Terminate search when any variable has no legal values

Forward Checking

• Useful general heuristic

• Employs a kind of arc consistency

• Detects many inconsistencies, but not all

• Effective along with other arc consistency strategies