Constraint-Satisfaction Problems
19 October 2020
Programming Assignment 1: Search

Programming Assignment 2: Multi-agent

Exam 1

Programming Assignment 3: Uncertainty
  Out soon!

Programming Assignment 4: Classification

Exam 2

Final project / paper
The story so far
The story so far

Search and planning

- Define a state space
- Define a goal test
- Find path from start to goal
The story so far

Search and planning

- Define a state space
- Define a goal test
- Find path from start to goal

Game trees

- Define utilities
- Find path from start that maximizes utility
The story so far

Search and planning
- Define a state space
- Define a goal test
- Find path from start to goal

Game trees
- Define utilities
- Find path from start that maximizes utility

Markov decision processes
- Define rewards, utility = (discounted) sum of rewards
- Find policy that maximizes utility
The story so far

Search and planning
   Define a state space
   Define a goal test
   Find path from start to goal

Game trees
   Define utilities
   Find path from start that maximizes utility

Markov decision processes
   Define rewards, utility = (discounted) sum of rewards
   Find policy that maximizes utility

Reinforcement learning
   Just like MDPs, but $T$ and $R$ are not known in advance
The story so far

Search and planning
Define a state space
Define a goal test
Find path from start to goal

Game trees
Define utilities
Find path from start that maximizes utility

Markov decision processes
Define rewards, utility = (discounted) sum of rewards
Find policy that maximizes utility

Reinforcement learning
Just like MDPs, but $T$ and $R$ are not known in advance

Today: Constraint satisfaction
Find solution that satisfies constraints
Not just for finding a sequential plan
Constraint-satisfaction problems
What's search for?

**Planning**: sequences of actions

Want: *path* to the goal

Paths have various costs and depths

Heuristics give problem-solving guidance
What’s search for?

**Planning**: sequences of actions

Want: *path* to the goal

Paths have various costs and depths

Heuristics give problem-solving guidance

**Identification**: assignments to variables

The *goal* itself is important, not the path

All paths at the same depth (for some formulations)

CSPs are a specialized class of identification problems
Constraint-satisfaction problems

Standard search problems:

State is a “black box”—any data structure
Goal test can be any function over states
Successor function can also be anything
Constraint-satisfaction problems

Standard search problems:

State is a “black box” – any data structure
Goal test can be any function over states
Successor function can also be anything
Constraint-satisfaction problems

Standard search problems:
State is a “black box” – any data structure
Goal test can be any function over states
Successor function can also be anything

Constraint satisfaction problems (CSPs):
State is defined by variables $X_i$ with values from a domain $D$
Goal test is a set of constraints specifying allowable combinations of values for subsets of variables
Constraint-satisfaction problems

Standard search problems:

State is a “black box” – any data structure
Goal test can be any function over states
Successor function can also be anything

Constraint satisfaction problems (CSPs):

State is defined by variables $X_i$ with values from a domain $D$
Goal test is a set of constraints specifying allowable combinations of values for subsets of variables
Constraint-satisfaction problems

Standard search problems:
- State is a “black box” – any data structure
- Goal test can be any function over states
- Successor function can also be anything

Constraint satisfaction problems (CSPs):
- State is defined by variables $X_i$ with values from a domain $D$
- Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

Simple example of a formal representation language
Constraint-satisfaction problems

Standard search problems:

State is a “black box” – any data structure
Goal test can be any function over states
Successor function can also be anything

Constraint satisfaction problems (CSPs):

State is defined by variables $X_i$ with values from a domain $D$
Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

Simple example of a formal representation language

Allows useful general-purpose algorithms with more power than standard search algorithms
Spectrum of representations

Atomic

Factored

Structured

Search, game-playing

CSPs, planning, propositional logic, Bayes nets, neural nets

First-order logic, databases, probabilistic programs
CSP examples
Example: Map coloring

Variables: WA, NT, Q, NSW, V, SA, T

Domains: \( D = \{ \text{red, green, blue} \} \)

Constraints: Adjacent regions must have different colors

Implicit: WA \neq NT

Explicit: (WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \ldots\}
Example: Map coloring

Variables: WA, NT, Q, NSW, V, SA, T

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

Constraints: Adjacent regions must have different colors

Implicit: WA $\neq$ NT

Explicit: $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \ldots\}$

Solutions are assignments satisfying all constraints, e.g.,

$\{WA = \text{red, } NT = \text{green, } Q = \text{red, } NSW = \text{green, } V = \text{red, } SA = \text{blue, } T = \text{green}\}$
Constraint graphs

**Binary CSP**: Each constraint relates (at most) two variables.

**Constraint graph**: Nodes are variables; arcs show constraints.

General-purpose CSP algorithms can use the graph structure to speed up search.

E.g., Tasmania is an independent subproblem!
Example: \textit{n}-queens
Example: $n$-queens
Example: $n$-queens

Formulation 1:

Variables: $X_{ij}$

Domains: $\{0, 1\}$

Constraints:
Example: \( n \)-queens

**Formulation 1:**

**Variables:** \( X_{ij} \)

**Domains:** \{0, 1\}

**Constraints:**

\[
\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}
\]
Example: $n$-queens

Formulation 1:

Variables: $X_{ij}$

Domains: $\{0, 1\}$

Constraints:

\[
\forall i, j, k \ (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}
\]

\[
\forall i, j, k \ (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}
\]

\[
\forall i, j, k \ (X_{ij}, X_{i+k,j+k}) \in \{(0, 0), (0, 1), (1, 0)\}
\]

\[
\forall i, j, k \ (X_{ij}, X_{i+k,j-k}) \in \{(0, 0), (0, 1), (1, 0)\}
\]
Example: $n$-queens

**Formulation 1:**

**Variables:** $X_{ij}$

**Domains:** \{0, 1\}

**Constraints:**

\[
\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}
\]

\[
\forall i, j, k \quad (X_{ij}, X_{k,j}) \in \{(0, 0), (0, 1), (1, 0)\}
\]

\[
\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0, 0), (0, 1), (1, 0)\}
\]

\[
\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0, 0), (0, 1), (1, 0)\}
\]

\[
\sum_{i,j} X_{ij} = N
\]
Example: $n$-queens

**Formulation 2:**

**Variables:** $Q_k$

**Domains:** $\{1, 2, 3, \ldots, N\}$

**Constraints:**

- **Implicit:** $\forall i, j$ non-threatening($Q_i, Q_j$)
- **Explicit:** $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
  ...

![Diagram of a chessboard with queens placed on the board]
Example: Sudoku

**Variables:**
Each (open) square

**Domains:**
{1, 2, ..., 9}

**Constraints:**
Example: Sudoku

**Variables:**
Each (open) square

**Domains:**
{1, 2, ..., 9}

**Constraints:**
9-way alldiff for each column
Example: Sudoku

**Variables:**

Each (open) square

**Domains:**

\{1, 2, ..., 9\}

**Constraints:**

9-way alldiff for each column

9-way alldiff for each row
Example: Sudoku

Variables:
Each (open) square

Domains:
{1, 2, ..., 9}

Constraints:
9-way alldiff for each column
9-way alldiff for each row
9-way alldiff for each region
Example: Sudoku

**Variables:**
Each (open) square

**Domains:**
{1, 2, ..., 9}

**Constraints:**
9-way alldiff for each column
9-way alldiff for each row
9-way alldiff for each region
(or can have a bunch of pairwise inequality constraints)
Example: The Waltz algorithm

The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects.

An early example of an AI computation posed as a CSP.

**Approach:**
- Each intersection is a variable.
- Adjacent intersections impose constraints on each other.
- Solutions are physically realizable 3D interpretations.
Example: The Waltz algorithm

The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects

An early example of an AI computation posed as a CSP

*Approach:*

- Each intersection is a variable
- Adjacent intersections impose constraints on each other
- Solutions are physically realizable 3D interpretations
Varieties of CSPs and constraints
Varieties of CSPs

Discrete variables

Finite domains

Size $d$ means $O(d^n)$ complete assignments

E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)

Infinite domains (integers, strings, etc.)

E.g., job scheduling, variables are start/end times for each job

Linear constraints solvable

Nonlinear constraints undecidable
Varieties of CSPs

Discrete variables

Finite domains

Size $d$ means $O(d^n)$ complete assignments

E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)

Infinite domains (integers, strings, etc.)

E.g., job scheduling, variables are start/end times for each job

Linear constraints solvable

Nonlinear constraints undecidable

Continuous variables

E.g., start/end times for Hubble Telescope observations

Linear constraints solvable in polynomial time by linear-programming methods
Varieties of constraints

*Unary constraints* involve a single variable (equivalent to reducing domains), e.g.,

\[ SA \neq \text{green} \]

*Binary constraints* involve pairs of variables, e.g.,

\[ SA \neq WA \]

*Higher-order constraints* involve three or more variables, e.g.,

Sudoku row constraints
Real-world CSPs

Scheduling problems, e.g., which class is offered when and where?

Hardware configuration / circuit layout

Transportation scheduling

Factory scheduling

Fault diagnosis

… and many more!

Many real-world problems involve real-valued variables…
Solving CSPs
Standard search formulation

States defined by the values assigned so far (partial assignments)

- **Initial state**: the empty assignment, {}
- **Successor function**: assign a value to an unassigned variable
- **Goal test**: the current assignment is complete and satisfies all constraints

We’ll start with the straightforward, naïve approach, then improve it.
Search methods

What would BFS do?

Level 1

Level 2

... 

Level N  solution!
Search methods

What would BFS do?
What would DFS do?
Let’s see!
Demo: Naive (depth-first) search
Search methods

What would BFS do?

What would DFS do?

Let’s see!

What problems does naïve search have?
Backtracking search
Backtracking search is the basic uninformed algorithm for solving CSPs. It’s depth-first search with two improvements:
Backtracking search is the basic uninformed algorithm for solving CSPs. It’s depth-first search with two improvements:

Idea 1: One variable at a time

Variable assignments are commutative, e.g.,

[WA = red then NT = green] same as [NT = green then WA = red]

so fix the ordering to get a smaller branching factor.

Only need to consider assignments to a single variable at each step
**Backtracking search** is the basic uninformed algorithm for solving CSPs. It’s depth-first search with two improvements:

**Idea 1:** One variable at a time

Variable assignments are commutative, e.g.,

\[\text{WA} = \text{red} \text{ then } \text{NT} = \text{green} \] same as \[\text{NT} = \text{green} \text{ then } \text{WA} = \text{red}\]

so fix the ordering to get a smaller branching factor.

Only need to consider assignments to a single variable at each step

**Idea 2:** Check constraints as you go

Only consider values that don’t conflict with previous assignments

Might have to do some computation to check the constraints

“Incremental goal test”
**Backtracking search** is the basic uninformed algorithm for solving CSPs. It’s depth-first search with two improvements:

**Idea 1:** One variable at a time

Variable assignments are commutative, e.g.,

\[
\text{[WA = red then NT = green]} \text{ same as [NT = green then WA = red]}
\]

so fix the ordering to get a smaller branching factor.

Only need to consider assignments to a single variable at each step

**Idea 2:** Check constraints as you go

Only consider values that don’t conflict with previous assignments

Might have to do some computation to check the constraints

“Incremental goal test”

Can solve \( n \)-queens for \( n \approx 25 \)
Backtracking example
Backtracking example
Backtracking example
Backtracking example
Backtracking example
Demo: Backtracking
Improving backtracking

General-purpose ideas give huge gains in speed
Improving backtracking

General-purpose ideas give huge gains in speed

*Filtering*:

Can we detect inevitable failure early?

*Ordering*: 
Improving backtracking

General-purpose ideas give huge gains in speed

*Filtering:*
  Can we detect inevitable failure early?

*Ordering:*
  Which variable should be assigned next?
  In what order should its values be tried?
Filtering
**Filtering**: Keep track of domains for unassigned variables and cross off bad options

**Forward checking**: Cross off values that violate a constraint when added to the existing assignment
**Filtering**: Keep track of domains for unassigned variables and cross off bad options

**Forward checking**: Cross off values that violate a constraint when added to the existing assignment
**Filtering:** Keep track of domains for unassigned variables and cross off bad options

**Forward checking:** Cross off values that violate a constraint when added to the existing assignment
**Filtering**: Keep track of domains for unassigned variables and cross off bad options

**Forward checking**: Cross off values that violate a constraint when added to the existing assignment
**Filtering**: Keep track of domains for unassigned variables and cross off bad options

**Forward checking**: Cross off values that violate a constraint when added to the existing assignment
Filtering: Keep track of domains for unassigned variables and cross off bad options

Forward checking: Cross off values that violate a constraint when added to the existing assignment
**Filtering:** Keep track of domains for unassigned variables and cross off bad options

**Forward checking:** Cross off values that violate a constraint when added to the existing assignment
Demo: Backtracking with forward checking
Ordering
Variable ordering: *Minimum remaining values* (MRV):

Choose the variable with the fewest legal values left in its domain

*Why min rather than max?*

Also called “most constrained variable”

“Fail-fast” ordering
Variable ordering: **Minimum remaining values** (MRV):

Choose the variable with the fewest legal values left in its domain

Why min rather than max?

Also called “most constrained variable”

“Fail-fast” ordering
**Variable ordering:** *Minimum remaining values* (MRV):

Choose the variable with the fewest legal values left in its domain

*Why min rather than max?*

Also called “most constrained variable”

“Fail-fast” ordering
Variable ordering: *Minimum remaining values* (MRV):

Choose the variable with the fewest legal values left in its domain

Why min rather than max?

Also called “most constrained variable”

“Fail-fast” ordering
Value ordering: Least-constraining value

Given a choice of variable, choose the least constraining value
I.e., the one that rules out the fewest values in the remaining variables
Note that it may take some computation to determine this! (E.g., rerunning filtering)

Why least rather than most?

Combining these ordering ideas makes 1000 queens feasible
Demo: Ordering
Iterative improvement
Iterative algorithms for CSPs

*Local search* methods typically work with “complete” states, i.e., all variables assigned.
Iterative algorithms for CSPs

*Local search* methods typically work with “complete” states, i.e., all variables assigned.
Iterative algorithms for CSPs

*Local search* methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:

- Take an assignment with unsatisfied constraints
- Operators *reassign* variable values
- No tree, no frontier!
Iterative algorithms for CSPs

*Local search* methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:

- Take an assignment with unsatisfied constraints
- Operators *reassign* variable values
- No tree, no frontier!
Iterative algorithms for CSPs

*Local search* methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:

- Take an assignment with unsatisfied constraints
- Operators *reassign* variable values
- No tree, no frontier!

Algorithm: While not solved,

*Variable selection:* randomly select any conflicted variable

*Value selection:* *min-conflicts* heuristic:

Choose a value that violates the fewest constraints
Example: 4-queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen in column

Goal test: no attacks

Evaluation: $h(n) =$ number of attacks
Demo: Iterative improvement – coloring
Summary

CSPs are a special kind of search problem:

- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:

- Ordering
- Filtering

Iterative min-conflicts is often effective in practice
Local search
Local search

Tree search keeps unexplored alternatives on the frontier (ensures completeness)

*Local search*: Improve a single option until you can’t make it better

*New successor function*: local changes

Generally much faster and more memory efficient (but incomplete and suboptimal)
Hill climbing

Simple, general idea:

Start wherever.
Repeat: Move to the best neighboring state.
If no neighbors better than current, quit.
Hill climbing diagram

- Objective function
- Global maximum
- Shoulder
- Local maximum
- "Flat" local maximum
- Current state
- State space
Hill climbing quiz

Starting from X, where do you end up?
Starting from Y, where do you end up?
Starting from Z, where do you end up?
Simulated annealing

**Idea:** Escape local maxima by allowing downhill moves

But make them rarer as time goes on

```java
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
             schedule, a mapping from time to “temperature”
    local variables: current, a node
                    next, a node
                    T, a “temperature” controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] − VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^{ΔE/T}
```
**Genetic algorithms** use a natural selection metaphor

Keep best $N$ hypotheses at each step (selection) based on a fitness function

Also have pairwise crossover operators, with optional mutation to give variety

Possibly the most misunderstood, misapplied (and even maligned) technique in AI
Example: $n$-queens

Why does crossover make sense here?

When wouldn’t it make sense?

What would mutation be?

What would a good fitness function be?
Acknowledgments

The lecture incorporates material from:

Pieter Abbeel, Dan Klein, et al., University of California, Berkeley;
ai.berkeley.edu