Constraint Satisfaction Problems (CSPs)

- A set of **variables**
- A **domain** for each variable
- A set of **constraints** or **evaluation function**
- Two kinds:
  1. **Satisfiability** Problems: Find an assignment that satisfies constraints (hard constraints)
  2. **Optimization** Problems: Find an assignment that optimises the evaluation function (soft constraints)
- A **solution** to a CSP is an assignment to the variables that satisfies all constraints
- A solution is a **model** of the constraints.

CSPs as Graph searching problems

Two ways:
- **Complete** Assignment:
  - nodes: assignment of value to all variables
  - neighbors: change one variable value

- **Partial** Assignment:
  - nodes: assignment to first \( k - 1 \) variables
  - neighbors: assignment to \( k^{th} \) variable

But,
- these search spaces can get extremely large (thousands of variables), so the branching factors can be big!
- path to goal is not important, only the goal is
- no predefined starting nodes

**Classic CSP: Crossword Construction**

Fill in all horizontal and vertical slots with words or phrases
Classic CSP: Crossword Construction

Dual Representations

Two ways to represent the crossword as a CSP

- **Primal** representation:
  - nodes represent word positions: 1-down... 6-across
  - domains are the words
  - constraints specify that the letters on the intersections must be the same.

- **Dual** representation:
  - nodes represent the individual squares
  - domains are the letters
  - constraints specify that the words must fit

Real World Example Domains

- Transportation Planning (Pascal Van Hentenryck)
  - https://www.youtube.com/watch?v=SxvM0jG3qLA
- Ride-sharing scheduling
- Air Traffic Control
  - http://dx.doi.org/10.1017/S026988912000215
- Disaster Recovery
- Factory process management
- Scheduling (courses, meetings, etc)
- ...
Constraints:
- Can be **N-ary** (over sets of $N$ variables - e.g. “dual representation” for crossword puzzles with letters as domains)
- Here: Consider only **Unary** and **Binary** (e.g. “primal representation” for crossword puzzles with words as domains)

Solutions:
- Generate and test
- Backtracking
- Consistency
- Hill-Climbing
- Randomized incl. Local Search

**Example**

Delivery robot: activities $a,b,c,d,e$, times $1,2,3,4$.
- $A$: variable representing the time activity $a$ will occur
- $B$: variable representing the time activity $b$ will occur

Domains:
- $D_A = \{1,2,3,4\}$
- $D_B = \{1,2,3,4\}$

Constraints:
- $(B \neq 3) \land (C \neq 2) \land (A \neq B) \land (B \neq C) \land (C < D) \land (A = D) \land (E < A) \land (E < B) \land (E < C) \land (E < D) \land (B \neq D)$

**Generate and Test**

Exhaustively go through all combinations, check each one
- $D = D_A \times D_B \times D_C \times D_D \times D_E$
- $D = \{< 1,1,1,1,1>, < 1,1,1,1,2>, ..., < 4,4,4,4,4 >\}$

- test: $< 1,1,1,1,1 > \ldots$ fail $\neg(A \neq B)$
- test: $< 1,1,1,1,2 > \ldots$ fail $\neg(A \neq B)$
- test: $< 1,1,1,1,3 > \ldots$ fail $\neg(A \neq B)$
- ...
- test: $< 1,2,1,1,1 > \ldots$ fail $\neg(C < D)$
- test: $< 1,2,1,1,2 > \ldots$ fail $\neg(C < D)$
- ...

**Backtracking**

Can use the fact that large portions of the state space can be pruned.
1. Order all variables
2. Evaluate constraints into the order as soon as they are grounded

- e.g. Assignment $A = 1 \land B = 1$ is inconsistent with constraint $A \neq B$ regardless of the value of the other variables.
Backtracking - Example

- Efficiency depends on order of variables!
- Finding optimal ordering is as hard as solving the problem
- Idea: push failures as high as possible
- Cut off large branches of the tree as soon as possible

Consistency

- More general approach
- Look for inconsistencies.
- E.g. C=4 in example inconsistent with any value of D (C < D)
- Backtracking will "re-discover" this for every value of A,B
- Graphical representation

Goal: Each domain has a single element, and all constraints are satisfied.
Consistency:
Constraint Network (CN)

- **Constraint Network** (CN)
- **domain constraint** is unary constraint on values in a domain, written \( \langle X, c(X) \rangle \).
- A node in a CN is **domain consistent** if no domain value violates any domain constraint.
- A CN is **domain consistent** if all nodes are domain consistent.
- Arc \( \langle X, c(X, Y) \rangle \) is a constraint on \( X \).
- An arc \( \langle X, c(X, Y) \rangle \) is **arc consistent** if for each \( X \in D_X \), there is some \( Y \in D_Y \) such that \( c(X, Y) \) is satisfied.
- A CN is **arc consistent** if all arcs are arc consistent.
- A set of variables \( \{X_1, X_2, X_3, \ldots, X_N\} \) is **path consistent** if all arcs and domains are consistent.

AC-3

- Alan Mackworth 1977!
- Makes a CN **arc consistent** (and **domain consistent**)
- To-Do Arcs Queue (TDA) has all inconsistent arcs

1. **Make** all domains domain consistent
2. **Put** all arcs \( \langle Z, c(Z, \_\_\_\_\_\_\_\_\_\_\_) \rangle \) in TDA
3. **repeat**
   a. **Select** and remove an arc \( \langle X, c(X, Y) \rangle \) from TDA
   b. **Remove** all values of domain of \( X \) that don’t have a value in domain of \( Y \) that satisfies the constraint \( c(X, Y) \)
   c. If any were removed, **Add** all arcs \( \langle Z, c'(Z, X) \rangle \) to TDA \( \forall Z \neq Y \) until TDA is empty

When AC-3 Terminates

- AC-3 **always** terminates with one of these three conditions:
  - Every domain is empty: there is **no solution**
  - Every domain has a single value: **solution!**
  - Some domain has more than one value: split it in two, run AC-3 recursively on two halves. Don’t have to start from scratch - only have to put back all arcs \( \langle Z, c'(Z, X) \rangle \) if \( X \) was the domain that was split.
- Connection between domain splitting and search.
Constraint Satisfaction: Example

{1,2,3,4} {1,2,4}
{1,2,3,4} {1,3,4}
{1,2,3,4}
AB
D C
E
A ≠ B
B ≠ D
C < D
A = D
E < A
B ≠ C
C < D
E < B
E < C
E < D

Goal: each domain has a single element, and all constraints are satisfied.

Example: Crossword Puzzle

Words:
ant, big, bus, car, has book, buys, hold, lane, year
beast, ginger, search, symbol, syntax

Variable Elimination

- Idea: **eliminate** the variables one-by-one passing their constraints to their neighbours
- When there is a single variable remaining, if it has no values, the network was **inconsistent**.
- The variables are eliminated according to some **elimination ordering**
- Different elimination orderings result in different size intermediate constraints.

Variable elimination (cont.)

- **Variable Elimination** Algorithm:
  - If there is only one variable, return the intersection of the (unary) constraints that contain it
  - Select a variable X
    - Join the constraints in which X appears, forming constraint R
    - Project R onto its variables other than X: call this R₂
    - Place new constraint R₂ between all variables that were connected to X
    - Remove X
    - Recursively solve the simplified problem
    - Return R joined with the recursive solution
Example network

\begin{itemize}
  \item $A \neq B$
  \item $B < E$
  \item $E - A \text{ is odd}$
  \item $D < C$
\end{itemize}

Example: arc-consistent network

\begin{itemize}
  \item $A \neq B$
  \item $E \neq C$
  \item $E \neq D$
  \item $D < C$
  \item $A < D$
  \item $B < E$
  \item $E - A \text{ is odd}$
\end{itemize}

Example: eliminating $C$

\begin{tabular}{c|c|c}
  $r_1 : C \neq E$ & $C$ & $E$ \\
  3 & 2 & \\
  3 & 4 & \\
  4 & 2 & \\
  4 & 3 & \\
\end{tabular}

\begin{tabular}{c|c|c}
  $r_2 : C > D$ & $C$ & $D$ \\
  3 & 2 & \\
  4 & 2 & \\
  4 & 3 & \\
\end{tabular}

\begin{tabular}{c|c|c}
  $r_3 : r_1 \bowtie r_2$ & $C$ & $D$ & $E$ \\
  3 & 2 & 2 & \\
  3 & 2 & 4 & \\
  4 & 2 & 2 & \\
  4 & 2 & 3 & \\
  4 & 3 & 2 & \\
  4 & 3 & 3 & \\
\end{tabular}

\begin{tabular}{c|c|c}
  $r_4 : \pi_{(D,E)} r_3$ & $D$ & $E$ \\
  2 & 2 & \\
  2 & 3 & \\
  2 & 4 & \\
  3 & 2 & \\
  3 & 3 & \\
\end{tabular}

\text{new constraint}

Resulting network after eliminating $C$

\begin{itemize}
  \item $A \neq B$
  \item $E \neq D$
  \item $D < C$
  \item $A < D$
  \item $B < E$
  \item $E - A \text{ is odd}$
  \item $r_4(E,D)$
\end{itemize}
Local Search

Back to CSP as Search (Local Search):

- **Maintain** an assignment of a value to each variable.
- At each step, select a **neighbor** of the current assignment (e.g., one that improves some **heuristic** value).
- Stop when a **satisfying** assignment is found, or return the best assignment found.

Requires:
- What is a neighbor?
- Which neighbor should be selected?

(Some methods maintain **multiple assignments**.)

Local Search for CSPs

- Aim is to find an assignment with **zero unsatisfied** constraints.
- Given an assignment of a value to each variable, a **conflict** is an unsatisfied constraint.
- The goal is an assignment with **zero conflicts**.
- **Heuristic** function to be minimized: the number of conflicts.

Greedy Descent Variants

- Find the variable-value pair that **minimizes** the number of conflicts at every step.
- Select a variable that participates in the **most** number of conflicts. Select a value that **minimizes** the number of conflicts.
- Select a variable that appears in **any** conflict. Select a value that **minimizes** the number of conflicts.
- Select a variable at **random**. Select a value that **minimizes** the number of conflicts.
- Select a variable and value at **random**; **accept** this change if it **doesn’t increase** the number of conflicts.

GSAT (Greedy SATisfyability)

Let \( n \) be random assignment of values to all variables
\( h(n) \) is number of un-satisfied constraints

**repeat**
- evaluate neighbors, \( n' \) of \( n \).
- can’t change the same variable twice in a row
- \( n = n^*, \) where \( n^* = \arg \min_{n'}(h(n')) \)
  even if \( h(n^*) > h(n) \)!
**Until stopping criteria is reached**

- e.g. start with \( A = 2, B = 2, C = 3, D = 2, E = 1 \) .... \( h=3 \)
  - change B to 4 ...
  - \( h = 1 \)
  - local minimum
  - change D to 4 (\( h=2 \))
  - change A to 4 (\( h=2 \))
  - change B to 2 (\( h=0 \))
Problems with Greedy Descent

- A **local minimum** that is not a global minimum
- A **plateau** where the heuristic values are uninformative
- A **ridge** is a local minimum where \( n \)-step look-ahead might help

Randomized Greedy Descent

As well as downward steps we can allow for:

- **Random steps:** move to a random neighbor.
- **Random restart:** reassign random values to all variables.

Which is more expensive computationally?
A mix of the two = **stochastic local search**

1-Dimensional Ordered Examples

Two 1-dimensional search spaces; step right or left:

(a) ![Graph](image1.png)
(b) ![Graph](image2.png)

Which method would most easily find the **global** minimum?

What happens in hundreds or **thousands of dimensions**?

What if different parts of the search space have **different structure**?

High Dimensional Search Spaces

- In high dimensions the search space is less easy to visualize
- Often consists of long, nearly flat "canyons"
- Hard to optimize using local search
- Step-size can be adjusted
**Stochastic Local Search**

Stochastic local search is a mix of:
- **Greedy descent**: move to a lowest neighbor
- **Random walk**: taking some random steps
- **Random restart**: reassigning values to all variables

**Variant: Simulated Annealing**

- Pick a variable at random and a new value at random.
- If it is an improvement, adopt it.
- If it isn’t an improvement, adopt it probabilistically depending on a temperature parameter, $T$.
  - With current assignment $n$ and proposed assignment $n'$ we move to $n'$ with probability $e^{-(h(n')-h(n))/T}$
- Temperature can be reduced.

<table>
<thead>
<tr>
<th>Temperature</th>
<th>1-worse</th>
<th>2-worse</th>
<th>3-worse</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.91</td>
<td>0.81</td>
<td>0.74</td>
</tr>
<tr>
<td>1</td>
<td>0.37</td>
<td>0.14</td>
<td>0.05</td>
</tr>
<tr>
<td>0.25</td>
<td>0.02</td>
<td>0.0003</td>
<td>0.000005</td>
</tr>
<tr>
<td>0.1</td>
<td>0.00005</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Simulated Annealing**

Let $n$ be random assignment of values to all variables Let $T$ be a (high) temperature

```
repeat
  Select neighbor $n'$ of $n$ at random
  If $h(n') < h(n)$ then
    $n = n'$
  else
    $n = n'$ with probability $e^{-(h(n')-h(n))/T}$
reduce $T$
until stopping criteria is reached
```
Tabu lists

- recall GSAT: never choose same variable twice.
- To prevent cycling we can maintain a tabu list of the $k$ last assignments.
- Don’t allow an assignment that is already on the tabu list.
- If $k = 1$, we don’t allow an assignment of to the same value to the variable chosen.
- We can implement it more efficiently than as a list of complete assignments.
- It can be expensive if $k$ is large.

Parallel Search

A total assignment is called an individual.

- maintain a population of $k$ individuals instead of one.
- At every stage, update each individual in the population.
- Whenever an individual is a solution, it can be reported.
- Like $k$ restarts, but uses $k$ times the minimum number of steps.

Beam Search

- Like parallel search, with $k$ individuals, but choose the $k$ best out of all of the neighbors (all if there are less than $k$).
- When $k = 1$, it is greedy descent.
- The value of $k$ lets us limit space and parallelism.

Stochastic Beam Search

- Like beam search, but it probabilistically chooses the $k$ individuals at the next generation.
- The probability that a neighbor is chosen is proportional to its heuristic value: $e^{-h(n)/T}$.
- This maintains diversity amongst the individuals.
- The heuristic value reflects the fitness of the individual.
- Like asexual reproduction: each individual mutates and the fittest ones survive.
Genetic Algorithms

Like stochastic beam search, but pairs of individuals are combined to create the offspring:

- For each generation:
  - Randomly choose pairs of individuals where the fittest individuals are more likely to be chosen.
  - For each pair, perform a cross-over: form two offspring each taking different parts of their parents:
  - Mutate some values.
- Stop when a solution is found.

Crossover

Given two individuals:

\[ X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m \]
\[ X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m \]

- Select \( i \) at random.
- Form two offspring:

\[ X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m \]
\[ X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m \]

The effectiveness depends on the ordering of the variables.
- Many variations are possible.

Comparing Stochastic Algorithms

How can you compare three algorithms when

- one solves the problem 30% of the time very quickly but doesn’t halt for the other 70% of the cases
- one solves 60% of the cases reasonably quickly but doesn’t solve the rest
- one solves the problem in 100% of the cases, but slowly?

How can you compare three algorithms when

- one solves the problem 30% of the time very quickly but doesn’t halt for the other 70% of the cases
- one solves 60% of the cases reasonably quickly but doesn’t solve the rest
- one solves the problem in 100% of the cases, but slowly?

Summary statistics, such as mean run time, median run time, and mode run time don’t make much sense.
Runtime Distribution

- Plots runtime (or number of steps) and the proportion (or number) of the runs that are solved within that runtime.

Next:

- Inference (Poole & Mackworth (2nd Ed.) chapter 5.1-5.3 and 13.1-13.2)