# Chapter 4: Searching

- **Lecture 1** Searching. Graphs. Generic search engine.
- **Lecture 2** Blind search strategies.
- **Lecture 3** Heuristic search, including $A^*$.
- **Lecture 4** Pruning the search space, direction of search, iterative deepening, dynamic programming.
- **Lecture 5** Constraint satisfaction problems, consistency algorithms.
- **Lecture 6** Hill climbing, randomized algorithms.

© David Poole, Alan Mackworth, Randy Goebel, and Oxford University Press 1998-2002

# Searching

- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.
- **Search** is a way to implement don't know nondeterminism.
- So far we have seen how to convert a semantic problem of finding logical consequence to a search problem of finding derivations.

# Search Graphs

- A graph consists of a set $N$ of nodes and a set $A$ of ordered pairs of nodes, called arcs.
- Node $n_2$ is a neighbor of $n_1$ if there is an arc from $n_1$ to $n_2$. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
- Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.

# Example Graph for the Delivery Robot

r123

storage

o125 o123 o119

l3d1

l3d2 l3d3

l2d1 l2d2

l2d3 l2d4

mail ts o103 o109 o111

# Search Graph for SLD Resolution

$a \leftarrow b \wedge c.$ $\quad a \leftarrow g.$

$a \leftarrow h.$ $\quad b \leftarrow j.$

$b \leftarrow k.$ $\quad d \leftarrow m.$

$d \leftarrow p.$ $\quad f \leftarrow m.$

$f \leftarrow p.$ $\quad g \leftarrow m.$

$g \leftarrow f.$ $\quad k \leftarrow m.$

$h \leftarrow m.$ $\quad p.$

$?a \wedge d$

- $yes \leftarrow a \wedge d$
  - $yes \leftarrow b \wedge c \wedge d$
    - $yes \leftarrow j \wedge c \wedge d$
    - $yes \leftarrow k \wedge c \wedge d$
      - $yes \leftarrow m \wedge c \wedge d$
  - $yes \leftarrow g \wedge d$
    - $yes \leftarrow m \wedge d$
    - $yes \leftarrow f \wedge d$
      - $yes \leftarrow m \wedge d$
      - $yes \leftarrow p \wedge d$
        - $yes \leftarrow d$
          - $yes \leftarrow m$
          - $yes \leftarrow p$
            - $yes \leftarrow$
  - $yes \leftarrow h \wedge d$
    - $yes \leftarrow m \wedge d$

# Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier of paths from the start node that have been explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the search strategy.

# Problem Solving by Graph Searching

# Graph Search Algorithm

**Input:** a graph,

a set of start nodes,

Boolean procedure $goal(n)$ that tests if $n$ is a goal node

$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;

**while** *frontier* is not empty:

**select** and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;

**if** $goal(n_k)$

**return** $\langle n_0, \ldots, n_k \rangle$;

**for every** neighbor $n$ of $n_k$

**add** $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;

**end while**

- We assume that after the search algorithm returns an answer, it can be asked for more answers and the procedure continues.
- Which value is selected from the frontier at each stage defines the search strategy.
- The *neighbors* defines the graph.
- *is_goal* defines what is a solution.

# Depth-first Search

- **Depth-first search** treats the frontier as a stack
- It always selects one of the last elements added to the frontier.
- If the frontier is $[p_1, p_2, \ldots]$
  - $p_1$ is selected. Paths that extend $p_1$ are added to the front of the stack (in front of $p_2$.
  - $p_2$ is only selected when all paths from $p_1$ have been explored.

# Illustrative Graph — Depth-first Search

# Complexity of Depth-first Search

- Depth-first search isn't guaranteed to halt on infinite graphs or on graphs with cycles.
- The space complexity is linear in the size of the path being explored.
- Search is unconstrained by the goal until it happens to stumble on the goal.

# Breadth-first Search

- Breadth-first search treats the frontier as a queue.
- It always selects one of the earliest elements added to the frontier.
- If the frontier is $[p_1, p_2, \ldots, p_r]$:
  - $p_1$ is selected. Its neighbors are added to the end of the queue, after $p_r$.
  - $p_2$ is selected next.

# Illustrative Graph — Breadth-first Search

1

2 3

4 5 6 7

8 9 10 11 12 13 14

15 16

# Complexity of Breadth-first Search

- The branching factor of a node is the number of its neighbors.
- If the branching factor for all nodes is finite, breadth-first search is guaranteed to find a solution if one exists. It is guaranteed to find the path with fewest arcs.
- Time complexity is exponential in the path length: $b^n$, where $b$ is branching factor, $n$ is path length.
- The space complexity is exponential in path length: $b^n$.
- Search is unconstrained by the goal.

# Lowest-cost-first Search

- Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} |\langle n_{i-1}, n_i \rangle|$$

- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- It finds a least-cost path to a goal node.
- When arc costs are equal $\Longrightarrow$ breadth-first search.

# Heuristic Search

- **Idea:** don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: **heuristics.**
- $h(n)$ is an estimate of the cost of the shortest path from node $n$ to a goal node.
- $h(n)$ uses only readily obtainable information (that is easy to compute) about a node.
- $h$ can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
- $h(n)$ is an underestimate if there is no path from $n$ to a goal that has path length less than $h(n)$.

# Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, we can use the straight-line distance from $n$ to the closest goal as the value of $h(n)$.
- If the graph is one of queries for a derivation from a KB, one heuristic function is the number of atoms in the query.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.

# Best-first Search

- **Idea:** select the path whose end is closest to a goal according to the heuristic function.
- Best-first search selects a path on the frontier with minimal $h$-value.
- It treats the frontier as a priority queue ordered by $h$.

# Illustrative Graph — Best-first Search

s

g

# Complexity of Best-first Search

- It uses space exponential in path length.
- It isn't guaranteed to find a solution, even if one exists.
- It doesn't always find the shortest path.

# Heuristic Depth-first Search

- It's a way to use heuristic knowledge in depth-first search.
- Idea: order the neighbors of a node (by $h$) before adding them to the front of the frontier.
- It locally selects which subtree to develop, but still does depth-first search. It explores all paths from the node at the head of the frontier before exploring paths from the next node.
- Space is linear in path length. It isn't guaranteed to find a solution. It can get led up the garden path.

# $A^*$ Search

- $A^*$ search uses both path cost and heuristic values
- $cost(p)$ is the cost of the path $p$.
- $h(p)$ estimates of the cost from the end of $p$ to a goal.
- Let $f(p) = cost(p) + h(p)$. $f(p)$ estimates of the total path cost of going from a start node to a goal via $p$.

$$\underbrace{\underbrace{start \overset{\text{path } p}{\longrightarrow} n}_{cost(p)} \underbrace{\overset{\text{estimate}}{\longrightarrow} goal}_{h(n)}}_{f(p)}$$

# $A^*$ Search Algorithm

- $A^*$ is a mix of lowest-cost-first and best-first search.
- It treats the frontier as a priority queue ordered by $f(n)$.
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

# Admissibility of $A^*$

If there is a solution, $A^*$ always finds an optimal solution —the first path to a goal selected— if

- the branching factor is finite
- arc costs are bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than $\epsilon$), and
- $h(n)$ is an underestimate of the length of the shortest path from $n$ to a goal node.

# Why is $A^*$ admissible?

- If a path $p$ to a goal is selected from a frontier, can there be a shorter path to a goal?
- Suppose path $p'$ is on the frontier. Because $p$ was chosen before $p'$, and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

- Because $h$ is an underestimate

$$cost(p') + h(p') \leq cost(p'')$$

for any path $p''$ to a goal that extends $p'$

- So $cost(p) \leq cost(p'')$ for any other path $p''$ to a goal.

# Why is $A^*$ admissible?

- There is always an element of an optimal solution path on the frontier before a goal has been selected. This is because, in the abstract search algorithm, there is the initial part of every path to a goal.
- $A^*$ halts, as the minimum $g$-value on the frontier keeps increasing, and will eventually exceed any finite number.

# Summary of Search Strategies

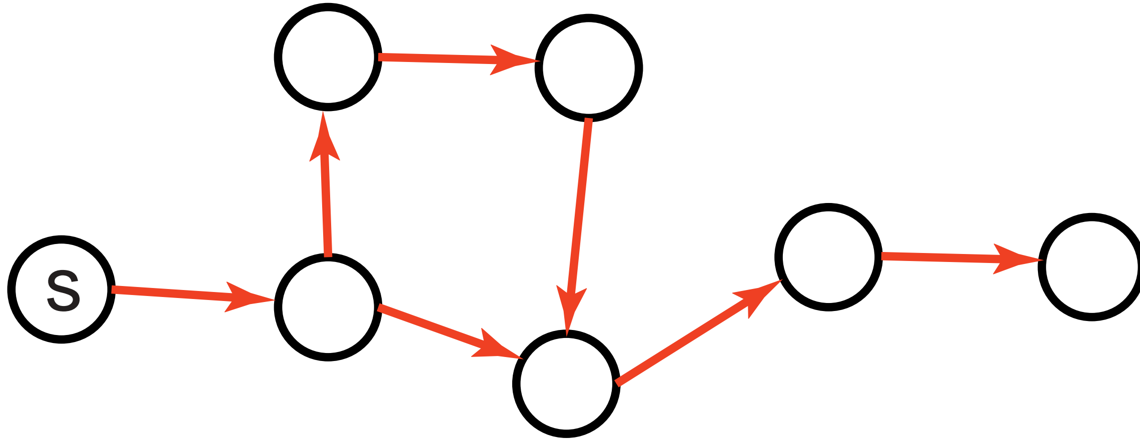
| Strategy | Frontier Selection | Halts? | Space |
|---|---|---|---|
| Depth-first | Last node added | No | Linear |
| Breadth-first | First node added | Yes | Exp |
| Heuristic depth-first | Local min $h(n)$ | No | Linear |
| Best-first | Global min $h(n)$ | No | Exp |
| Lowest-cost-first | Minimal $g(n)$ | Yes | Exp |
| $A^*$ | Minimal $f(n)$ | Yes | Exp |

# Cycle Checking



- You can prune a path that ends in a node already on the path. This pruning cannot remove an optimal solution.
- Using depth-first methods, with the graph explicitly stored, this can be done in constant time.
- For other methods, the cost is linear in path length.

# Multiple-Path Pruning




- You can prune a path to node *n* that you have already found a path to.
- Multiple-path pruning subsumes a cycle check.
- This entails storing all nodes you have found paths to.

# Multiple-Path Pruning & Optimal Solution

Problem: what if a subsequent path to $n$ is shorter than the first path to $n$?

- You can remove all paths from the frontier that use the longer path.
- You can change the initial segment of the paths on the frontier to use the shorter path.
- You can ensure this doesn't happen. You make sure that the shortest path to a node is found first.

# Multiple-Path Pruning & $A^*$

Suppose path $p$ to $n$ was selected, but there is a shorter path to $n$. Suppose this shorter path is via path $p'$ on the frontier.

Suppose path $p'$ ends at node $n'$.

$cost(p) + h(n) \leq cost(p') + h(n')$ because $p$ was selected before $p'$.

$cost(p') + d(n', n) < cost(p)$ because the path to $n$ via $p'$ is shorter.

$$d(n', n) < cost(p) - cost(p') \leq h(n') - h(n).$$

You can ensure this doesn't occur if $|h(n') - h(n)| \leq d(n', n)$.

# Monotone Restriction

- Heuristic function $h$ satisfies the **monotone restriction** if $|h(n') - h(n)| \leq d(m, n)$ for every arc $\langle m, n \rangle$.
- If $h$ satisfies the monotone restriction, $A^*$ with multiple path pruning always finds the shortest path to a goal.

# Iterative Deepening

- So far all search strategies that are guaranteed to halt use exponential space.
- **Idea:** let's recompute elements of the frontier rather than saving them.
- Look for paths of depth 0, then 1, then 2, then 3, etc.
- You need a depth-bounded depth-first searcher.
- If a path cannot be found at depth $B$, look for a path at depth $B + 1$. Increase the depth-bound when the search fails unnaturally (depth-bound was reached).

# Iterative Deepening Complexity

Complexity with solution at depth $k$ & branching factor $b$:

| level | breadth-first | iterative deepening | # nodes |
|---|---|---|---|
| 1 | 1 | $k$ | $b$ |
| 2 | 1 | $k-1$ | $b^2$ |
| $k-1$ | 1 | 2 | $b^{k-1}$ |
| $k$ | 1 | 1 | $b^k$ |
| | $\geq b^k$ | $\leq b^k \left(\frac{b}{b-1}\right)^2$ | |

# Direction of Search

The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.

**Forward branching factor:** number of arcs out of a node.

**Backward branching factor:** number of arcs into a node.

Search complexity is $b^n$. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.

Note: sometimes when graph is dynamically constructed, you may not be able to construct the backwards graph.

# Bidirectional Search

- You can search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$. This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.

# Island Driven Search

**Idea:** find a set of islands between $s$ and $g$.

$$s \longrightarrow i_1 \longrightarrow i_2 \longrightarrow \ldots \longrightarrow i_{m-1} \longrightarrow g$$

There are $m$ smaller problems rather than 1 big problem.

This can win as $mb^{k/m} \ll b^k$.

The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.

You can solve the subproblems using islands $\Longrightarrow$ **hierarchy of abstractions.**

# Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node $n$ to a goal.

This can be built backwards from the goal:

$$dist(n) = \begin{cases} 0 & \text{if } is_goal(n) \\ \min_{\langle n,m\rangle \in A}(|\langle n,m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

This can be used locally to determine what to do.

There are two main problems:

- You need enough space to store the graph.
- The *dist* function needs to be recomputed for each goal.

# Constraint Satisfaction Problems

- **Multi-dimensional Selection Problems**

- Given a set of variables, each with a set of possible values (a domain), assign a value to each variable that either
  - satisfies some set of constraints:
    **satisfiability problems** — "hard constraints"
  - minimizes some cost function, where each assignment of values to variables has some cost:
    **optimization problems** — "soft constraints"

- Many problems are a mix of hard and soft constraints.

# Relationship to Search

- The path to a goal isn't important, only the solution is.
- Many algorithms exploit the multi-dimensional nature of the problems.
- There are no predefined starting nodes.
- Often these problems are huge, with thousands of variables, so systematically searching the space is infeasible.
- For optimization problems, there are no well-defined goal nodes.

# Posing a Constraint Satisfaction Problem

A CSP is characterized by

- A set of variables $V_1, V_2, \ldots, V_n$.
- Each variable $V_i$ has an associated domain $\mathbf{D}_{V_i}$ of possible values.
- For satisfiability problems, there are constraint relations on various subsets of the variables which give legal combinations of values for these variables.
- A solution to the CSP is an $n$-tuple of values for the variables that satisfies all the constraint relations.

# Example: scheduling activities

Variables: $A$, $B$, $C$, $D$, $E$ that represent the starting times of various activities.

Domains: $\mathbf{D}_A = \{1, 2, 3, 4\}$, $\mathbf{D}_B = \{1, 2, 3, 4\}$, $\mathbf{D}_C = \{1, 2, 3, 4\}$, $\mathbf{D}_D = \{1, 2, 3, 4\}$, $\mathbf{D}_E = \{1, 2, 3, 4\}$

Constraints:

$$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge$$
$$(C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge$$
$$(E < C) \wedge (E < D) \wedge (B \neq D).$$

# Generate-and-Test Algorithm

Generate the assignment space $\mathbf{D} = \mathbf{D}_{V_1} \times \mathbf{D}_{V_2} \times \ldots \times \mathbf{D}_{V_n}$.
Test each assignment with the constraints.

Example:

$$
\begin{aligned}
\mathbf{D} &= \mathbf{D}_A \times \mathbf{D}_B \times \mathbf{D}_C \times \mathbf{D}_D \times \mathbf{D}_E \\
&= \{1, 2, 3, 4\} \times \{1, 2, 3, 4\} \times \{1, 2, 3, 4\} \\
&\quad \times \{1, 2, 3, 4\} \times \{1, 2, 3, 4\} \\
&= \{\langle 1, 1, 1, 1, 1 \rangle, \langle 1, 1, 1, 1, 2 \rangle, \ldots, \langle 4, 4, 4, 4, 4 \rangle\}.
\end{aligned}
$$

Generate-and-test is always exponential.

# Backtracking Algorithms

Systematically explore **D** by instantiating the variables in some order and evaluating each constraint predicate as soon as all its variables are bound. Any partial assignment that doesn't satisfy the constraint can be pruned.

Example Assignment $A = 1 \wedge B = 1$ is inconsistent with constraint $A \neq B$ regardless of the value of the other variables.

# CSP as Graph Searching

A CSP can be seen as a graph-searching algorithm:

- Totally order the variables, $V_1, \ldots, V_n$.
- A node assigns values to the first $j$ variables.
- The neighbors of node $\{V_1/v_1, \ldots, V_j/v_j\}$ are the consistent nodes $\{V_1/v_1, \ldots, V_j/v_j, V_{j+1}/v_{j+1}\}$ for each $v_{j+1} \in \mathbf{D}_{V_{j+1}}$.
- The start node is the empty assignment $\{\}$.
- A goal node is a total assignment that satisfies the constraints.

# Consistency Algorithms

Idea: prune the domains as much as possible before selecting values from them.

A variable is domain consistent if no value of the domain of the node is ruled impossible by any of the constraints.

Example: $\mathbf{D}_B = \{1, 2, 3, 4\}$ isn't domain consistent as $B = 3$ violates the constraint $B \neq 3$.

# Arc Consistency

- A **constraint network** has nodes corresponding to variables with their associated domain. Each constraint relation $P(X, Y)$ corresponds to arcs $\langle X, Y \rangle$ and $\langle Y, X \rangle$.

- An arc $\langle X, Y \rangle$ is **arc consistent** if for each value of $X$ in $\mathbf{D}_X$ there is some value for $Y$ in $\mathbf{D}_Y$ such that $P(X, Y)$ is satisfied. A network is arc consistent if all its arcs are arc consistent.

- If an arc $\langle X, Y \rangle$ is *not* arc consistent, all values of $X$ in $\mathbf{D}_X$ for which there is no corresponding value in $\mathbf{D}_Y$ may be deleted from $\mathbf{D}_X$ to make the arc $\langle X, Y \rangle$ consistent.

# Example Constraint Network

$A$ $\{1,2,3,4\}$

$B$ $\{1,2,4\}$

$C$ $\{1,3,4\}$

$D$ $\{1,2,3,4\}$

$E$ $\{1,2,3,4\}$

$A \neq B$

$A = D$

$B \neq D$

$B \neq C$

$C < D$

$E < A$

$E < D$

$E < C$

$E < B$

# Arc Consistency Algorithm

The arcs can be considered in turn making each arc consistent.

An arc $\langle X, Y \rangle$ needs to be revisited if the domain of $Y$ is reduced.

Three possible outcomes (when all arcs are arc consistent):

- One domain is empty $\Longrightarrow$ no solution
- Each domain has a single value $\Longrightarrow$ unique solution
- Some domains have more than one value $\Longrightarrow$ may or may not be a solution

# Finding solutions when AC finishes

- If some domains have more than one element $\Longrightarrow$ search
- Split a domain, then recursively solve each half.
- We only need to revisit arcs affected by the split.
- It is often best to split a domain in half.

# Hill Climbing

Many search spaces are too big for systematic search.

A useful method in practice for some consistency and optimization problems is **hill climbing:**

- Assume a heuristic value for each assignment of values to all variables.
- Maintain an assignment of a value to each variable.
- Select a “neighbor” of the current assignment that improves the heuristic value to be the next current assignment.

# Selecting Neighbors in Hill Climbing

- When the domains are small or unordered, the neighbors of a node correspond to choosing another value for one of the variables.

- When the domains are large and ordered, the neighbors of a node are the adjacent values for one of the dimensions.

- If the domains are continuous, you can use
  **Gradient ascent:** change each variable proportional to the gradient of the heuristic function in that direction. The value of variable $X_i$ goes from $v_i$ to $v_i + \eta \frac{\partial h}{\partial X_i}$.
  **Gradient descent:** go downhill; $v_i$ becomes $v_i - \eta \frac{\partial h}{\partial X_i}$.

# Problems with Hill Climbing

**Foothills** local maxima that are not global maxima

**Plateaus** heuristic values are uninformative

**Ridge** foothill where *n*-step lookahead might help

**Ignorance of the peak**

Plateau

Ridge

Foothill

# Randomized Algorithms

- Consider two methods to find a maximum value:
  - Hill climbing, starting from some position, keep moving uphill & report maximum value found
  - Pick values at random & report maximum value found
- Which do you expect to work better to find a maximum?
- Can a mix work better?

# Randomized Hill Climbing

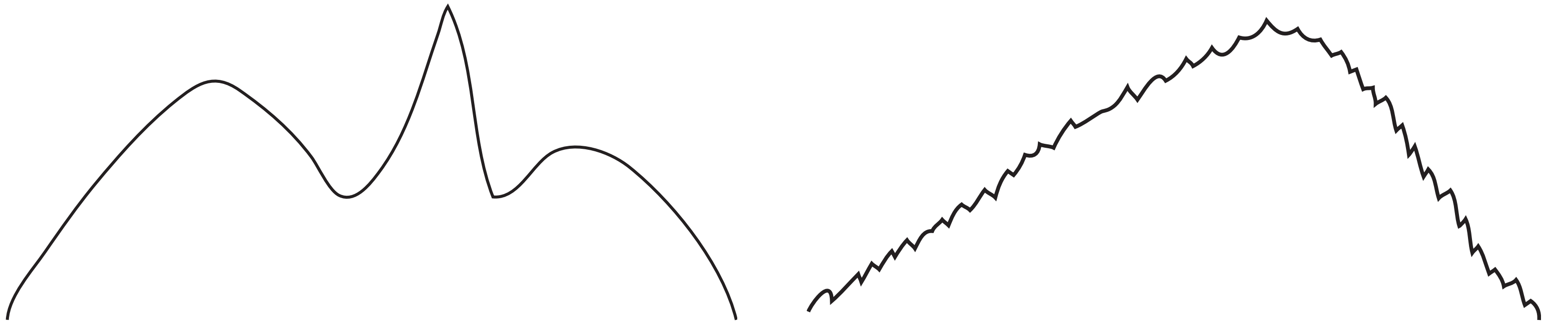
As well as uphill steps we can allow for:

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

Which is more expensive computationally?

# 1-Dimensional Ordered Examples

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



- Which method would most easily find the maximum?
- What happens in hundreds or thousands of dimensions?
- What if different parts of the search space have different structure?

# Stochastic Local Search for CSPs

- Goal is to find an assignment with zero unsatisfied relations.
- Heuristic function: the number of unsatisfied relations.
- We want an assignment with minimum heuristic value.
- 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

# Greedy Descent

- It may be too expensive to find the variable-value pair that minimizes the heuristic function at every step.
- An alternative is:
  - Select a variable that participates in the most number of conflicts.
  - Choose a (different) value for that variable that resolves the most conflicts.

The alternative is easier to compute even if it doesn't always maximally reduce the number of conflicts.

# Random Walk

You can add randomness:

- When choosing the best variable-value pair, randomly sometimes choose a random variable-value pair.
- When selecting a variable then a value:
  - Sometimes choose a random variable.
  - Sometimes choose, at random, a variable that participates in a conflict (a red node).
  - Sometimes choose a random variable.
- Sometimes choose the best value and sometimes choose a random value.

# 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?
- 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.

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

1 10 100 1000

# 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 node $n$ and proposed node $n'$ we move to $n'$ with probability $e^{(h(n')-h(n))/T}$
- Temperature can be reduced.

# Tabu lists

- To prevent cycling we can maintain a tabu list of the $k$ last nodes visited.
- Don't allow a node that is already on the tabu list.
- If $k = 1$, we don't allow a node to the same value.
- We can implement it more efficiently than as a list of complete nodes.
- It can be expensive if $k$ is large.

# Parallel Search

- **Idea:** maintain $k$ nodes instead of one.
- At every stage, update each node.
- Whenever one node 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 $k$ nodes, but you choose the $k$ best out of all of the neighbors.
- When $k = 1$, it is hill climbing.
- When $k = \infty$, it is breadth-first search.
- The value of $k$ lets us limit space and parallelism.
- Randomness can also be added.

# Stochastic Beam Search

- Like beam search, but you probabilistically choose the $k$ nodes at the next generation.
- The probability that a neighbor is chosen is proportional to the heuristic value.
- This maintains diversity amongst the nodes.
- The heuristic value reflects the fitness of the node.
- Like asexual reproduction: each node gives its mutations and the fittest ones survive.

# Genetic Algorithms

- Like stochastic beam search, but pairs of nodes are combined to create the offspring:
- For each generation:
  - Randomly choose pairs of nodes where the fittest individuals are more likely to be chosen.
  - For each pair, perform a cross-over: form two offsprings each taking different parts of their parents:
  - Mutate some values
- Report best node found.

# Crossover

- Given two nodes:

$$X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m$$

$$X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m$$

- Select $i$ at random.
- Form two offsprings:

$$X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m$$

$$X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m$$

- Note that this depends on an ordering of the variables.
- Many variations are possible.

# Example: Crossword Puzzle

| 1 | | 2 | | |
|---|---|---|---|---|
| | ■ | | | |
| 3 | | | | |
| | | | | |
| | | 4 | | |
| | | | | |

**Words:**

---

ant, big, bus, car, has

book, buys, hold,

lane, year

beast, ginger, search,

symbol, syntax

# Constraint satisfaction revisited

- A **Constraint Satisfaction problem** consists of:
  - a set of variables
  - a set of possible values, a **domain** for each variable
  - A set of constraints amongst subsets of the variables (relations)
- The aim is to find a set of assignments that satisfies all constraints, or to find all such assignments.

# Example: crossword puzzle

|   | 1 | 2 |   |   |
|---|---|---|---|---|
|   | 3 |   |   |   |
| 4 |   |   | 5 |   |
|   |   |   |   |   |
|   | 6 |   |   |   |
|   |   |   |   |   |
|   |   |   |   |   |

at, be, he, it, on,
eta, hat, her, him,
one,
desk, dove, easy,
else, help, kind,
soon, this,
dance, first, fuels,
given, haste,
loses, sense,
sound, think,
usage

# Dual Representations

Two ways to represent the crossword as a CSP

- First representation:
  - nodes represent the positions 1 to 6
  - domains are the words
  - constraints specify that the letters on the intersections must be the same.
- Dual representation:
  - nodes represent the intersecting squares
  - domains are the letters
  - constraints specify that the words must fit

# Representations for image interpretation

- First representation:
  - nodes represent the chains and regions
  - domains are the scene objects
  - constraints correspond to the intersections and adjacency
- Dual representation:
  - nodes represent the intersections
  - domains are the intersection labels
  - constraints specify that the chains must have same marking

# Arc Consistency for non-binary relations

- Each relation $R(X_1, \ldots, X_k)$ converted into $k$ hyperarcs:

$$\langle X_1, R(X_1, \ldots, X_k)\rangle$$

$$\ldots$$

$$\langle X_k, R(X_1, \ldots, X_k)\rangle$$

- Hyperarc $\langle X_i, R(X_1, \ldots, X_k)\rangle$ is arc consistent if
  - for every $v_i \in domain(X_i)$
  - there exists $v_1 \in domain(X_1)$, … $v_{i-1} \in domain(X_{i+1})$, $v_{i+1} \in domain(X_{i+1})$ … $v_k \in domain(X_k)$
  - such that $R(X_1, \ldots, X_k)$ is true.

# Variable Elimination

- Idea: eliminate the variables one-by-one passing their constraints to their neighbours

- To eliminate a variable $X_i$:
  - Join all of the relations in which $X_i$ appears.
  - Project the join onto the other variables, forming a new relation.
  - Remember which values of $X_i$ are associated with the tuples of the new relation.
  - Replace the old relations containing $X_i$ with the new relation.

# Variable elimination (cont.)

- When there is a single variable remaining, if it has no values, the network was inconsistent.
- The solutions can be computed from the remembered mappings.
- The variables are eliminated according to some elimination ordering
- Different elimination oderings result in different size relations being generated.

# Example network

A

{1,2,3,4}

$A<D$

D

{1,2,3,4}

$A \neq B$

B

{1,2,3,4}

*E-A is odd*

$E \neq D$

$D<C$

$B<E$

{1,2,3,4}

E

$E \neq C$

{1,2,3,4}

C

# Example: arc-consistent network

A
{1,2}

$A < D$

$A \neq B$

D
{2,3}

$E$-$A$ *is odd*

B
{1,2,3}

$E \neq D$

$D < C$

$B < E$

{2,3,4}
E

$E \neq C$

{3,4}
C

# Example: eliminating $C$

| $r_1 : C \neq E$ | $C$ | $E$ |
|---|---|---|
| | 3 | 2 |
| | 3 | 4 |
| | 4 | 2 |
| | 4 | 3 |

| $r_2 : C > D$ | $C$ | $D$ |
|---|---|---|
| | 3 | 2 |
| | 4 | 2 |
| | 4 | 3 |

| $r_3 : r_1 \bowtie r2$ | $C$ | $D$ | $E$ |
|---|---|---|---|
| | 3 | 2 | 2 |
| | 3 | 2 | 4 |
| | 4 | 2 | 2 |
| | 4 | 2 | 3 |
| | 4 | 3 | 2 |
| | 4 | 3 | 3 |

| $r_4 : \pi_{\{D,E\}} r_3$ | $D$ | $E$ |
|---|---|---|
| | 2 | 2 |
| | 2 | 3 |
| | 2 | 4 |
| | 3 | 2 |
| | 3 | 3 |

# Resulting network after eliminating *C*

A

{1,2}

$A<D$

$A \neq B$

D

{2,3}

$E$-$A$ *is odd*

B

{1,2,3}

$E \neq D$

$r_4(E,D)$

$B<E$

{2,3,4}

E

# Stochastic local search for CSPs

- The following can be used to solve CSPs:
  - hill climbing on the assignments.
    - Choose the best variable then the best value.
    - Choose the best variable-value pair

    Best: satisfies the most constraints
  - random assignments of values.
  - random walks
- A mix works even better.

# Evaluating Algorithms

- Summary statistics such as mean or median of run times are often not useful in comparing algorithms.
- The information about an algorithm performance can be determined from a runtime distribution.
- A runtime distribution specifies the proportion of the instances that have a running time less than any particular run time.