Goal-Based Agents
Informed Search

Vasant Honavar
Artificial Intelligence Research Laboratory
Department of Computer Science
Bioinformatics and Computational Biology Program
Center for Computational Intelligence, Learning, & Discovery
Iowa State University
honavar@cs.iastate.edu
www.cs.iastate.edu/~honavar/
www.cild.iastate.edu/
www.bcb.iastate.edu/
www.igert.iastate.edu

General recipe for uninformed search

function TREE-SEARCH(problem, open-list) return a solution or failure

open-list ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), open-list)

loop do
  if EMPTY?(open-list) then return failure
  node ← REMOVE-FIRST(open-list)
  if GOAL-TEST[problem] applied to STATE[node] succeeds
    then return SOLUTION(node)
  open-list ← INSERT-ALL(EXPAND(node, problem), open-list)
A strategy is defined by picking the order of node expansion
Finding optimal solutions

- Branch and bound search (BBS) with dynamic programming
- Open list nodes sorted by cost $g(n)$ of cheapest known partial paths to the respective nodes
- Terminate when a node picked from the open list happens to be a goal node
- Questions:
  - Is BBS complete?
    - Yes
  - Is BBS admissible?
    - Yes
  - Under the assumption that each arc cost is bounded from below by some positive constant $\delta$

Informed search

- Informed $\rightarrow$ use problem-specific knowledge
- Which search strategies?
  - Best-first search and its variants
- Heuristic functions
  - How to design them
- Local search and optimization
  - Hill climbing, local beam search, genetic algorithms,…
- Local search in continuous spaces
- Online search agents
A heuristic function

- [dictionary] “A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood.”

Heuristics, those rules of thumb,
Often scorned as sloppy, dumb,
Yet slowly commonsense become!
– Judea Pearl, in *Heuristics*

Why we need heuristics

- Combinatorial explosion
- Uninformed search runs out of time, space, or both quickly
- Perfect rationality unaffordable

```python
function TREE-SEARCH(problem, open-list) return a solution or failure
  open-list ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), open-list)
  loop do
    if EMPTY?(open-list) then return failure
    node ← REMOVE-FIRST(open-list)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    open-list ← INSERT-ALL(EXPAND(node, problem), open-list)
  A strategy is defined by picking the order of node expansion
  
  – Use a heuristic function to guide choice of actions
```

Slides based on material from Russell and Norvig, updated by Vasant Honavar, 2009
Hill-climbing search

- Chooses locally best actions, with random choice to break ties
- Terminates when a state corresponding to a locally optimal evaluation is reached
- Does not look ahead of the immediate neighbors of the current state.
- a.k.a. greedy local search

Hill-climbing search

function HILL-CLIMBING(problem) return a state that is a local maximum
input: problem, a problem
local variables: current, a node.
neighbor, a node.

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return
    STATE[current]
    current ← neighbor
Hill-climbing example

- 8-queens problem (complete-state formulation)
- Successor function:
  - move a single queen to another square in the same column
- Heuristic function $h(n)$:
  - the number of pairs of queens that are attacking each other (directly or indirectly)

Drawbacks of hill-climbing search

- Ridge
  - sequence of local maxima difficult for greedy algorithms to navigate
- Plateau
  - an area of the state space where the evaluation function is flat
Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves
  - The selection probability can vary with the steepness of the uphill move
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima

Best-first search

- General approach informed search:
  - Best-first search: node is selected for expansion based on an evaluation function \( f(n) \)
- Idea: evaluation function measures distance to the goal.
  - Choose node which appears best
- Implementation:
  - Open-list is a queue sorted in decreasing order of desirability.
  - Special cases: greedy search, A* search
Why we need heuristics

- Combinatorial explosion
- Uninformed search runs out of time, space, or both quickly
- Perfect rationality unaffordable
- Can we come up with a compromise?
  - Use a heuristic function to guide choice of actions
- Suppose we could estimate the cost of the cheapest solution obtainable by expanding each node that could be chosen

\[ h : S \rightarrow \mathbb{R}^+ \]
\[ \forall n \in S, h(n) \geq 0 \]
\[ \forall g \in G, h(g) = 0 \]
\[ h(s_1) = 0.8 \quad h(s_2) = 2.0 \quad h(s_3) = 1.6 \]

A heuristic function

In problems requiring the cheapest path from start state to a goal state
- \( h(n) = \) estimated cost of the cheapest path from node \( n \) to a goal node
- \( h(g_1) = h(g_2) = 0 \)
Heuristic function - Example

- E.g for the 8-puzzle
  - Avg. solution cost is about 22 steps
  - Exhaustive search to depth $22 = 3.1 \times 10^{10}$ states
  - A good heuristic function can reduce search

Heuristic functions

Two commonly used heuristics
- $h_1$ = the number of misplaced tiles (not counting blank) relative to the goal (why?)
  \[ h_1(s) = 8 \]
- $h_2$ = the sum of the distances of the tiles (not counting blank) from their goal positions (manhattan distance)
  \[ h_2(s) = 3+1+2+2+2+3+3+2 = 18 \]
Simple examples of heuristic search

- Best first search
  - open list is ordered by \( h \) values of the nodes
- Hill-climbing search
  - Depth first like traversal, with each set of successors ordered by \( h \) values
- A* search
  - BBS-like search with dynamic programming
  - Open list nodes ordered by
    \[
    f(n) = g(n) + h(n)
    \]
Romania with step costs in km

- $h_{SLD}$ = straight-line distance heuristic.
- $h_{SLD}$ cannot be computed from the problem description, requires additional information or knowledge on the part of the agent.
- In this example $f(n) = h(n)$
  - Expand node that is closest to goal
  = Greedy best-first search

A* search

- Best-known form of best-first search
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ the cost (so far) to reach the node.
  - $h(n)$ estimated cost to get from the node to the goal
  - $f(n)$ estimated total cost of path through $n$ to goal
A* search

- A* search using an admissible heuristic
  - A heuristic is admissible if it never overestimates the cost to reach the goal
  - Are optimistic

Formally:
1. \( h(n) \leq h^*(n) \) where \( h^*(n) \) is the true cost from \( n \)
2. \( h(n) \geq 0 \) so \( h(G) = 0 \) for any goal \( G \).

e.g. \( h_{SLD}(n) \) never overestimates the actual road distance

Romania example
Admissible heuristics

- A heuristic \( h(n) \) is admissible if for every node \( n \),
  \( h(n) \leq h^*(n) \), where \( h^*(n) \) is the true cost to reach the goal state from \( n \).
- An admissible heuristic never overestimates the cost to reach the goal,
  i.e., it is optimistic.
- Example: \( h_{SLD}(n) \) (never overestimates the actual road distance)
- Theorem: If \( h(n) \) is admissible, \( A^* \) using \textsc{tree-search} is admissible
  (provided each arc cost is bounded from below by a positive constant \( \delta \))
- Proof Sketch:
  - Show that \( A^* \) terminates with a solution
  - Show that \( A^* \) terminates with an optimal solution

Completeness of \( A^* \): \( A^* \) Terminates with a solution

- Assumption: Cost of each arc is bounded from below by a positive constant
  \( \forall n,j, c(n,j) \geq \delta > 0 \)
- Any time a path gets extended, its cost increases by at least
- There is a solution, i.e., a bounded cost path to a goal (by assumption)
- Termination condition: a path terminating in a goal is picked off the front
  of the list of partial paths \( L \)
- There are two kinds of paths other than those that terminate in a goal:
  - Paths that lead to dead ends – discovered and discarded (if their cost is lower than that of a path to a goal)
  - Paths that are infinite (demoted behind a path leading to a goal because sooner or later their cost exceeds the cost of a path leading to a goal)
  - See notes for a formal proof.
Admissibility of A*: A* cannot terminate with a suboptimal goal

Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

\[ f(G_2) = g(G_2) \text{ since } h(G_2) = 0 \]
\[ g(G_2) > g(G) \text{ since } G_2 \text{ is suboptimal} \]
\[ f(G) = g(G) \text{ since } h(G) = 0 \]
\[ f(G_2) > f(G) \text{ from above} \]

Hence $f(G_2) > f(n)$, and A* will never select $G_2$ for expansion. That is, A* tree search is admissible provided the heuristic function is admissible.

See notes for a more formal proof.
What if the search space is not a tree?

function Graph-Search(problem, fringe) returns a solution, or failure

closed ← an empty set
fringe ← INSERT(Make-Node(INITIAL-STATE[problem]), fringe)

loop do
  if fringe is empty then return failure
  node ← REMOVE-FRONT(fringe)
  if GOAL-TEST(problem)(STATE[node]) then return SOLUTION(node)
  if STATE[node] is not in closed then
    add STATE[node] to closed
    fringe ← INSERTALL(Expand(node, problem), fringe)

Graph search

- Previous proof breaks down
- Possible solutions:
  - Discard more expensive paths when there are multiple paths to a node
    - Adds messy book-keeping
    - Admissible if $h$ is admissible
  - Ensure that optimal path to any repeated state is always first followed
    - Extra requirement on $h(n)$: consistency (equivalently monotonicity)

\[ \forall n, \forall n' \text{ where } n' \text{ is a child of } n, \]

\[ h(n) \leq h(n') + c(n, n') \]
Consistency

If $h$ is consistent, we have

$$h(n) \leq c(n, m) + h(m)$$

$f(n)$ is non-decreasing along any path

$$f(m) = g(m) + h(m)$$
$$= g(n) + c(n, m) + h(m)$$
$$\geq g(n) + h(n) + \delta$$
$$> f(n)$$

→ First goal selected for expansion must be an optimal goal
→ $A^*$ using graph search is admissible provided the heuristic used is consistent.

Admissibility of $A^*$ with a consistent heuristic

- $A^*$ expands nodes in order of increasing $f$ value
- $f$-contours: Contours of nodes in the state space with equal $f$ values
- Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
Properties of A* search

• Complete?: YES
  – Since bands of increasing $f$ are added
  – Unless there are infinitely many nodes with $f < f(G)$
    • Not possible when the branching factor is finite and arc costs are bounded from below by $\delta$

• Time complexity:
  – exponential in the length of the solution (in the worst case)

• Space complexity:
  – Need to maintain all generated nodes in memory
  – Space is a more serious problem than time

Memory-bounded heuristic search

• Some solutions to A* space problems (maintain completeness and optimality)
  – Iterative-deepening A* (IDA*)
    • Cutoff information is the $f$-cost ($g+h$) instead of depth
    • Can expand too many nodes when the arc costs are unequal
  – (simple) Memory-bounded A* ((S)MA*)
    • Drop the worst-leaf node when memory is full

  – Recursive best-first search (RBFS)
    • Recursive algorithm that attempts to mimic standard best-first search with linear space
(simple) memory-bounded A*

- Use all available memory.
  - i.e. expand best leaf until available memory is full
  - When full, SMA* drops worst leaf node (highest $f$-value)
  - Backup the $f$-value of the forgotten node to its parent

- What if all leafs have the same $f$-value?
  - Same node could be selected for expansion and deletion
  - SMA* solves this by expanding newest best leaf and deleting oldest worst leaf.

- SMA* is complete if solution is reachable, admissible if optimal solution is reachable within the available memory bound

Recursive best-first search (RBFS)

- Similar to IDA* but keeps track of the $f$ value of the best-alternative path available from any ancestor of the current node
  - If current $f$ values exceeds this alternative $f$ value then backtrack to the best alternative path
  - During backtracking (unwinding of recursion) change $f$ value of each node to best $f$-value of its children
  - Remembers the $f$ value of the best leaf in the “forgotten” subtree and hence can revisit that subtree, if warranted, at some point in the future
Recursive best-first search

**function** RECURSIVE-BEST-FIRST-SEARCH(problem) **return** a solution or failure

**return** RFBS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

**function** RFBS(problem, node, f_limit) return a solution or failure and a new f-cost limit

if GOAL-TEST[problem](STATE[node]) then **return** node

successors ← EXPAND(node, problem)

if successors is empty then return failure, ∞

for each s in successors do

\[ f[s] \leftarrow \max(g(s) + h(s), f[node]) \]

**repeat**

best ← the lowest f-value node in successors

if \( f[best] > f\_limit \) then **return** failure, \( f[best] \)

alternative ← the second lowest f-value among successors

result, \( f[best] \leftarrow \) RBFS(problem, best, min(f_limit, alternative))

if result = failure then **return** result

---

### Recursive best-first search: Example

1. After expanding Arad, Sibiu, Rimnicu Vilcea

- Path until Rumnicu Vilcea is already expanded
- Above node; f-limit for every recursive call is shown on top.
- Below node: \( f(n) \)
- The path is followed until Pitesti which has a \( f \) value worse than the \( f\_limit \).
Recursive best-first search: Example

- Unwind recursion and store best $f$-value for current best leaf Pitesti
  
  \[ \text{result, } f[\text{best}] \leftarrow \text{RBFS}(\text{problem, best, min}(f\_limit, \text{alternative})) \]

- best is now Fagaras. Call RBFS for new best
  - best value is now 450

---

Recursive best-first search: example.

- Unwind recursion and store best $f$-value for current best leaf Fagaras
- best is now Rimnicu Viclea (again). Call RBFS for new best
  - Subtree is again expanded.
  - Best alternative subtree is now through Timisoara.
- Solution is found since because 447 > 417.
RBFS evaluation

- RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- Like A*, admissible if \( h(n) \) is admissible
- Space complexity is \( O(b^d) \).
- Time complexity difficult to characterize
  - Depends on accuracy if \( h(n) \) and how often best path changes.
- IDA* and RBFS suffer from too little memory

---

Learning to search better

- All previous algorithms use fixed strategies.
- Agents can learn to improve their search by exploiting experience gained during the search (e.g., by analyzing missteps)
- (we will see examples of this when we consider learning agents)
Heuristic functions revisited

- E.g for the 8-puzzle
  - Avg. solution cost is about 22 steps
  - Exhaustive search to depth 22: $3.1 \times 10^{10}$ states.
  - A good heuristic function can reduce search

Heuristic functions

- E.g for the 8-puzzle knows two commonly used heuristics
  - $h_1 = \text{the number of misplaced tiles}$
    - $h_1(s) = 8$
  - $h_2 = \text{the sum of the distances of the tiles from their goal positions (manhattan distance)}.$
    - $h_2(s) = 3+1+2+2+2+3+3+2 = 18$
How good is a heuristic?

• Effective branching factor $b^*$
  – $N = \text{the number of nodes generated by a heuristic search algorithm (e.g., A*)}$
  – The effective branching factor of search = the branching factor of a tree of depth $d$ needs to have in order to contain $N+1$ nodes.

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

– Measure is fairly constant for sufficiently hard problems.
  • Can thus provide a good guide to the heuristic’s overall usefulness.
  • $b^* = 1$ for A* search with a perfect heuristic

Calculation of effective branching factor

$N=20$, $d=4$

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d = \frac{(b^*)^{d+1} - 1}{(b^* - 1)}$$

Solve for $b^* : 21 = 1 + b^* + (b^*)^2 + (b^*)^3 + (b^*)^4$

$b^* \approx 1.5$
Dominance

• If \( h_2(n) \geq h_1(n) \) for all \( n \), then \( h_2 \) dominates \( h_1 \) (both admissible) then \( h_2 \) is more informative than \( h_1 \).

• If \( h_2 \) is more informative than \( h_1 \), then every partial path that is expanded by \( A^* \) using \( h_2 \) is necessarily expanded by \( A^* \) using \( h_1 \).

• Typical search costs (average number of nodes expanded) using the two heuristics for 8-puzzle (averaged over 100 instances for each depth):
  - \( d=12 \): IDA*: 3,644,035 nodes
    - \( A^*(h_1) = 227 \) nodes
    - \( A^*(h_2) = 73 \) nodes
  - \( d=24 \): IDA*: too many nodes
    - \( A^*(h_1) = 39,135 \) nodes
    - \( A^*(h_2) = 1,641 \) nodes

Combining admissible heuristics

\[
h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}
\]
Inventing admissible heuristics: Relaxation

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem:
  - Relaxed 8-puzzle for \( h_1 \): a tile can move anywhere
    - As a result, \( h_1(n) \) gives the shortest solution
  - Relaxed 8-puzzle for \( h_2 \): a tile can move to any adjacent square.
    - As a result, \( h_2(n) \) gives the shortest solution.
- The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem
- Heuristic function based on exact cost of the solution of the relaxed problem is consistent

Inventing admissible heuristics: Subproblems

- Admissible heuristics can be derived from the solution cost of a sub problem of a given problem
- This cost is a lower bound on the cost of the real problem
- Pattern databases store the exact solution to for every possible sub problem instance.
  - The complete heuristic is constructed using the patterns in the DB
Inventing admissible heuristics: learning

- Admissible heuristic can be learned from experience:
  - Experience = solving lots of 8-puzzles
  - An inductive learning algorithm can be used to predict costs for other states that arise during search (we will revisit this when we consider learning agents)

Local search and optimization

- Previously: systematic exploration of search space.
  - Path to goal is solution to problem
- For some problems path is irrelevant
  - Pathless search problems
  - E.g 8-queens

- Different algorithms can be used
  - Local search
Local search and optimization

- Local search
  - Maintain a single current state and move to neighboring state(s)
- Advantages:
  - Uses very little memory
  - Find often reasonable solutions in large or infinite state spaces
- Are useful for pure optimization problems
  - Find the best state according to some *objective function*.

Hill-climbing search

- Chooses locally best actions, with random choice to break ties
- Terminates when a state corresponding to a locally optimal evaluation is reached
- Does not look ahead of the immediate neighbors of the current state.
- a.k.a. *greedy local search*
Local search and optimization

Hill-climbing search

function HILL-CLIMBING( problem) return a state that is a local maximum
input: problem, a problem
local variables: current, a node.
neighbor, a node.

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
 neighbour ← a highest valued successor of current
if VALUE[neighbor] ≤ VALUE[current] then return
STATE[current]
current ← neighbor
Hill-climbing example

- 8-queens problem (complete-state formulation)
- Successor function:
  - move a single queen to another square in the same column
- Heuristic function $h(n)$:
  - the number of pairs of queens that are attacking each other (directly or indirectly)

Drawbacks of hill-climbing search

- Ridge
  - sequence of local maxima difficult for greedy algorithms to navigate
- Plateau
  - an area of the state space where the evaluation function is flat
Hill-climbing variations

- **Stochastic hill-climbing**
  - Random selection among the uphill moves
  - The selection probability can vary with the steepness of the uphill move
- **First-choice hill-climbing**
  - Stochastic hill climbing by generating successors randomly until a better one is found
- **Random-restart hill-climbing**
  - Tries to avoid getting stuck in local maxima

Simulated annealing

- **Escape local maxima** by allowing occasional “bad” moves
  - but gradually decrease their size and frequency
- **Origin of idea**
  - annealing of metal
- **Bouncing ball analogy:**
  - Shaking hard (= high temperature)
  - Shaking less (= lower the temperature)
  - The probability $P(s)$ of being in state $s$ $P(s) \propto e^{\frac{E(s)}{kT}}$
    where $E(s)$ is the “energy” at state $s$, given by the objective function applied to state $s$, $k$ a positive constant and $T$ the “temperature”
- If $T$ decreases slowly enough, then simulated annealing is guaranteed to find a global optimum with probability approaching 1
Simulated annealing

- Escape local maxima by allowing occasional “bad” moves
- If $T$ decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Used for VLSI layout, airline scheduling, convex optimization, etc.

---

**function** SIMULATED-ANNEALING( problem, schedule) **return** a solution

**state**

**input:** problem, a problem

schedule, a mapping from time to temperature

**local variables:** current, a node.

next, a node.

$T$, a “temperature” controlling the probability of downward steps

$\text{current} \leftarrow \text{MAKE-NODE(INITIAL-STATE}[\text{problem}])$

**for** t $\leftarrow$ 1 to $\infty$ **do**

$T \leftarrow \text{schedule}[t]$

if $T = 0$ **then** return current

next $\leftarrow$ a randomly selected successor of current

$\Delta E \leftarrow \text{VALUE}[\text{next}] - \text{VALUE}[\text{current}]$

if $\Delta E > 0$ **then** current $\leftarrow$ next

else current $\leftarrow$ next only with probability $e^{\Delta E / T}$
Local beam search

- Keep track of $k$ states instead of one
  - Initially: $k$ random states
  - Next: determine all successors of $k$ states
  - If any of successors is goal → finished
  - Else select $k$ best from successors and repeat

- Major difference from random-restart search
  - Information is shared among $k$ search threads

- Can suffer from lack of diversity
  - Stochastic variant: choose $k$ successors with probability proportional to difference between the state’s value relative to the value of the current state

Genetic algorithms

- A successor state is generated by combining two parent states
- Start with $k$ randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function)
  - Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation
Genetic algorithms

- Variant of local beam search with *genetic recombination*

```plaintext
function GENETIC_ALGORITHM( population, FITNESS-FN) return an individual

input: population, a set of individuals

FITNESS-FN, a function which determines the quality of the individual

repeat

new_population ← empty set

loop for i from 1 to SIZE(population) do

x ← RANDOM_SELECTION(population, FITNESS_FN)
y ← RANDOM_SELECTION(population, FITNESS_FN)

child ← REPRODUCE(x, y)
if (small random probability) then child ← MUTATE(child)
add child to new_population

population ← new_population

until some individual is fit enough or enough time has elapsed

return the best individual
```

Slides based on material from Russell and Norvig, updated by Vasant Honavar, 2009
Genetic Algorithms

- GA emphasizes combining information from good parents (crossover)
- EA emphasize generating variants of current solutions (mutation)
- many variants, e.g., reproduction models, operators

Genetic algorithms

- The original genetic algorithm due to John Holland is now known as the simple genetic algorithm (SGA)
- Other GA use different:
  - Representations
    - E.g. program parse trees (genetic programming)
  - Different variants and combinations of genetic operators
    - E.g. evolutionary programs
  - Selection mechanisms
## SGA summary

<table>
<thead>
<tr>
<th>State Representation</th>
<th>Binary strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recombination</td>
<td>N-point or uniform</td>
</tr>
<tr>
<td>Mutation</td>
<td>Bit-flipping with fixed probability</td>
</tr>
<tr>
<td>Parent selection</td>
<td>Fitness-Proportionate</td>
</tr>
<tr>
<td>Survivor selection</td>
<td>All children replace parents</td>
</tr>
<tr>
<td>Speciality</td>
<td>Emphasis on crossover</td>
</tr>
</tbody>
</table>

**Genotype space** = \{0,1\}^L

![Representation diagram]

**State space** → **Genotype space** = \{0,1\}^L

**Encoding** (representation) → **Decoding** (inverse representation)
1. Select parents for the mating pool  
   (size of mating pool = population size)  
2. Shuffle the mating pool  
3. For each consecutive pair apply crossover with  
   probability $p_c$, otherwise copy parents  
4. For each offspring apply mutation (bit-flip with  
   probability $p_m$ independently for each bit)  
5. Replace the whole population with the resulting  
   offspring

SGA operators: 1-point crossover

- Choose a random point on the two parents  
- Split parents at this crossover point  
- Create children by exchanging tails  
- $P_c$ typically in range (0.6, 0.9)
SGA operators: mutation

- Alter each gene independently with a probability $p_m$
- $p_m$ is called the mutation rate
  - Typically between $1/pop\_size$ and $1/chromosome\_length$

<table>
<thead>
<tr>
<th>parent</th>
<th>1 1 1 1 1 1 1 1 1 1 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>child</td>
<td>0 1 0 0 1 0 1 0 0 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

SGA operators: Selection

- Main idea: better individuals get higher chance
  - Chances proportional to fitness
  - Implementation: roulette wheel technique
    - Assign to each individual a part of the roulette wheel
    - Spin the wheel $n$ times to select $n$ individuals

- fitness(A) = 3
- fitness(B) = 1
- fitness(C) = 2

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/6 = 50%</td>
<td>1/6 = 17%</td>
<td>2/6 = 33%</td>
</tr>
</tbody>
</table>
An example after Goldberg '89 (1)

- Simple problem:
  - max $x^2$ over {0,1,...,31}
- GA approach:
  - Representation: binary code, e.g. 01101 ↔ 13
  - Population size: 4
  - 1-point crossover, bitwise mutation
  - Roulette wheel selection
  - Random initialization
- We show one generation of GA

$x^2$ example: selection

<table>
<thead>
<tr>
<th>String no.</th>
<th>Initial population</th>
<th>$x$ Value</th>
<th>Fitness $f(x) = x^2$</th>
<th>$Prob_i$</th>
<th>Expected count</th>
<th>Actual count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01101</td>
<td>13</td>
<td>169</td>
<td>0.14</td>
<td>0.58</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11000</td>
<td>24</td>
<td>576</td>
<td>0.49</td>
<td>1.97</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>01000</td>
<td>8</td>
<td>64</td>
<td>0.06</td>
<td>0.22</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10011</td>
<td>19</td>
<td>361</td>
<td>0.31</td>
<td>1.23</td>
<td>1</td>
</tr>
<tr>
<td>Sum Average Max</td>
<td></td>
<td></td>
<td><strong>1170</strong></td>
<td><strong>1.00</strong></td>
<td><strong>4.00</strong></td>
<td><strong>4</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>293</strong></td>
<td><strong>0.25</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>576</strong></td>
<td><strong>0.49</strong></td>
<td><strong>1.97</strong></td>
<td><strong>2</strong></td>
</tr>
</tbody>
</table>
### X² example: crossover

<table>
<thead>
<tr>
<th>String no.</th>
<th>Mating pool</th>
<th>Crossover point</th>
<th>Offspring after xover</th>
<th>$x$ Value</th>
<th>Fitness $f(x) = x^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 1 1 0 1</td>
<td>4</td>
<td>0 1 1 0 0</td>
<td>12</td>
<td>144</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 0 0</td>
<td>4</td>
<td>1 1 0 0 1</td>
<td>25</td>
<td>625</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 0 0</td>
<td>2</td>
<td>1 1 0 1 1</td>
<td>27</td>
<td>729</td>
</tr>
<tr>
<td>4</td>
<td>1 0 0 1 1</td>
<td>2</td>
<td>1 0 0 0 0</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>1754</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>439</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td><strong>729</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Slides based on material from Russell and Norvig, updated by Vasant Honavar, 2009

### X² example: mutation

<table>
<thead>
<tr>
<th>String no.</th>
<th>Offspring after xover</th>
<th>Offspring after mutation</th>
<th>$x$ Value</th>
<th>Fitness $f(x) = x^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 1 1 0 0</td>
<td>1 1 1 0 0</td>
<td>26</td>
<td>676</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 0 1</td>
<td>1 1 0 0 1</td>
<td>25</td>
<td>625</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 1 1</td>
<td>1 1 0 1 1</td>
<td>27</td>
<td>729</td>
</tr>
<tr>
<td>4</td>
<td>1 0 0 0 0</td>
<td>1 0 1 0 0</td>
<td>18</td>
<td>324</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td></td>
<td></td>
<td></td>
<td>2354</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td>588.5</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td></td>
<td></td>
<td></td>
<td>729</td>
</tr>
</tbody>
</table>

Slides based on material from Russell and Norvig, updated by Vasant Honavar, 2009
The simple GA

- Has been subject of many (early) studies
  - still often used as benchmark for novel GAs
- Shows many shortcomings, e.g.
  - Representation is too restrictive
  - Mutation & crossovers only applicable for bit-string & integer representations
  - Selection mechanism sensitive for converging populations with close fitness values
  - Generational population model (step 5 in SGA repr. cycle) can be improved with explicit survivor selection

Alternative Crossover Operators

- Performance with 1 Point Crossover depends on the order that variables occur in the representation
  - more likely to keep together genes that are near each other
  - Can never keep together genes from opposite ends of string
  - This is known as *Positional Bias*
  - Can be exploited if we know about the structure of our problem, but this is not usually the case
**n-point crossover**

- Choose n random crossover points
- Split along those points
- Glue parts, alternating between parents
- Generalisation of 1 point (still some positional bias)

Parents:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Children:

```
0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0
```

---

**Uniform crossover**

- Assign 'heads' to one parent, 'tails' to the other
- Flip a coin for each gene of the first child
- Make an inverse copy of the gene for the second child
- Inheritance is independent of position

Parents:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Children:

```
0 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 1 1 0 1 1 0 1 0 0 1 0 1 0 0 1 1 0 0 1 0 0
```

---

*Slides based on material from Russell and Norvig, updated by Vasant Honavar, 2009*
Crossover OR mutation?

- it depends on the problem
- in general, it is good to have both
- Mutation generates variants of good states by making small changes (analogous to hill climbing)
- Crossover generates states by combing features of good states
- Mutation-only-EA is possible, crossover-only-EA would not work

Crossover OR mutation? (cont’d)

- **Crossover helps explore the state space:**
  - Discover promising areas in the search space, i.e. gaining information about the state space
  - Makes a big jump to an area somewhere “in between” two (parent) states
- **Mutation helps exploit the known parts of the state space**
  - Mutation makes small changes to already explored states so as to reach nearby states
- There is co-operation AND competition between them
**Crossover OR mutation? (cont’d)**

- Only crossover can combine information from two explored states
- Only mutation can introduce new information (alleles or values of state variables)
- Crossover does not change the allele frequencies of the population (If we start with 50% 0’s on first bit in the population, we will have 50% 0’s after performing $n$ crossovers)
- To hit the optimum you often need a ‘lucky’ mutation

---

**Integer representations**

- Some problems naturally have integer variables
- Others take *categorical* values from a fixed set e.g. {blue, green, yellow, pink}
- $n$-point / uniform crossover operators work
- Extend bit-flipping mutation to allow
  - “creep” i.e. more likely to move to similar value
  - Random choice (esp. categorical variables)
Permutation Representations

- **Ordering/sequencing problems**
- **Task is (or can be solved by) arranging some objects in a certain order**
  - Example: sort algorithm: important thing is which elements occur before others (order)
  - Example: Travelling Salesman Problem (TSP): important thing is which elements occur next to each other (adjacency)
- **Solutions are generally expressed as a permutation:**
  - if there are $n$ variables then the representation is as a list of $n$ integers, each of which occurs exactly once

Permutation representation: TSP example

- **Problem:**
  - Given $n$ cities
  - Find a complete tour with minimal length
- **Encoding:**
  - Label the cities 1, 2, … , $n$
  - One complete tour is one permutation (e.g. for $n = 4$ \([1,2,3,4], [3,4,2,1]\) are OK)
- **Search space is BIG:**
  - for 30 cities there are $30! \approx 10^{32}$ possible tours
Mutation operators for permutations

- Normal mutation operators lead to invalid states
- e.g. bit-wise mutation: let gene $i$ have value $j$
  - changing to some other value $k$ would mean that $k$ occurred twice and $j$ no longer occurred
- Must change at least two values
- Mutation parameter now reflects the probability that some operator is applied once to the state rather than individually in each position

Insert Mutation for permutations

- Pick two values at random
- Move the second to follow the first, shifting the rest along to accommodate
- Note that this preserves most of the order and the adjacency information
Swap mutation for permutations

- Pick two values at random and swap their positions
- Preserves most of adjacency information (4 links broken), disrupts order more

\[ \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \quad \rightarrow \quad 1 & 5 & 3 & 4 & 2 & 6 & 7 & 9 \end{array} \]

Inversion mutation for permutations

- Pick two values at random and then invert the substring between them.
- Preserves most adjacency information (only breaks two links) but disrupts order information

\[ \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \quad \rightarrow \quad 1 & 5 & 4 & 3 & 2 & 6 & 7 & 8 \end{array} \]
Scramble mutation for permutations

- Pick a subset of genes at random
- Randomly rearrange the alleles in those positions

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\rightarrow
\begin{array}{cccccccc}
1 & 3 & 5 & 4 & 2 & 6 & 7 & 8 \\
\end{array}
\]

(note subset does not have to be contiguous)

Crossover operators for permutations

- “Normal” crossover operators will often lead to inadmissible solutions

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 & 5 \\
5 & 4 & 3 & 2 & 1 \\
\end{array}
\rightarrow
\begin{array}{cccc}
1 & 2 & 3 & 2 \\
5 & 4 & 3 & 4 & 5 \\
\end{array}
\]

- Many specialized operators have been devised which focus on combining order or adjacency information from the two parents
Order 1 crossover

• Idea is to preserve relative order that elements occur in
• Informal procedure:
  1. Choose an arbitrary part from the first parent
  2. Copy this part to the first child
  3. Copy the values that are not in the first part, to the first child:
     • starting right from cut point of the copied part,
     • using the order of the second parent
     • and wrapping around at the end
  4. Analogous for the second child, with parent roles reversed

Order 1 crossover example

• Copy randomly selected set from first parent

1 2 3 4 5 6 7 8 9

9 3 7 8 2 6 5 1 4

• Copy rest from second parent in order 1, 4, 9, 3, 7, 8, 2, 6, 5

1 2 3 4 5 6 7 8 9

9 3 7 8 2 6 5 1 4

3 8 2 4 5 6 7 1 9
Population Models

- SGA uses a Generational model:
  - each individual survives for exactly one generation
  - the entire set of parents is replaced by the offspring
- At the other end of the scale are Steady-State models:
  - one offspring is generated per generation
  - one member of population replaced
- Generation Gap
  - the proportion of the population replaced
  - 1.0 for SGA, 1/pop_size for steady state GA

Fitness Based Competition

- Selection can occur in two places:
  - Selection from current generation to take part in mating (parent selection)
  - Selection from parents + offspring to go into next generation (survivor selection)
- Selection operators work on whole individual
  - i.e. they are representation-independent
Implementation example: SGA

- Expected number of copies of an individual $i$
  \[ E(n_i) = \mu \cdot f(i)/\langle f \rangle \]
  \[ \mu = \text{pop.size}, \]
  \[ f(i) = \text{fitness of } i, \langle f \rangle \text{ avg. fitness in pop.} \]
- Roulette wheel algorithm:
  - Given a probability distribution, spin a 1-armed wheel $n$ times to make $n$ selections

Fitness-Proportionate Selection

- Problems include
  - One highly fit member can rapidly take over if rest of population is much less fit: Premature Convergence
  - At end of runs when fitness values are similar, selection pressure is lost
- Solutions
  - Windowing: $f'(i) = f(i) - \beta t$
    - where $\beta$ is worst fitness in this (last n) generations
  - Sigma Scaling: $f'(i) = \max\left( f(i) - \langle f \rangle - c \cdot \sigma_i, 0.0 \right)$
    - where $c$ is a constant, usually 2.0
Rank – Based Selection

- Attempt to avoid problems of FPS by basing selection probabilities on relative rather than absolute fitness
- Rank population according to fitness and then base selection probabilities on rank where fittest has rank $\mu$ and worst rank 1
- This imposes a sorting overhead on the algorithm, but this is usually negligible compared to the fitness evaluation time

Tournament Selection

- All methods above rely on global population statistics
  - Could be a bottleneck esp. on parallel machines
  - Relies on presence of external fitness function which might not exist: e.g. evolving game players
- Informal Procedure:
  - Pick $k$ members at random then select the best of these
  - Repeat to select more individuals
Further variants

Genetic Programming

• Candidate solutions represented by parse trees of programs e.g. (+ 2 3 (* 2 5))
• Genetic operators manipulate the population guided by a fitness function

• Optional slides on online (real-time) search algorithms
Exploration problems

- Until now all algorithms were offline
  - Offline = solution is determined before executing it.
  - Online = interleaving computation and action
- Online search is necessary for dynamic and semi-dynamic environments
  - It is impossible to take into account all possible contingencies.
- Used for exploration problems:
  - Unknown states and actions
  - e.g. any robot in a new environment, a newborn baby,…

Online search problems

- Agent knowledge:
  - ACTION(s): list of allowed actions in state s
  - C(s,a,s'): step-cost function (! After s' is determined)
  - GOAL-TEST(s)
- An agent can recognize previous states
- Actions are deterministic
- Access to admissible heuristic h(s)
  e.g. manhattan distance
Online search problems

- Objective: reach goal with minimal cost
  - Cost = total cost of traveled path
  - Competitive ratio = comparison of cost with cost of the solution path if search space is known.

An adversary can make things difficult

- Assume an adversary who can construct the state space while the agent explores it
  - Visited states S and A. What next?
  - Fails in one of the state spaces
  - No algorithm can avoid dead ends in all state spaces.
Online search agents

- The agent maintains a map of the environment.
  - Updated based on percept input.
  - This map is used to decide next action.

An online version can only expand the node it is physically in (local order)

---

**Online DF-search**

```plaintext
function ONLINE_DFS-AGENT(s) return an action
input: s', a percept identifying current state
static: result, a table indexed by action and state, initially empty
        unexplored, a table that lists for each visited state, the action not yet tried
        unbacktracked, a table that lists for each visited state, the backtrack not yet tried
        s,a, the previous state and action, initially null

if GOAL-TEST(s') then return stop
if s' is a new state then unexplored[s'] ← ACTIONS(s)
if s is not null then do
    result[a,s] ← s'
    add s to the front of unbacktracked[s]
if unexplored[s] is empty then
    if unbacktracked[s] is empty then return stop
    else a ← an action b such that result(b, s')=POP(unbacktracked[s])
else a ← POP(unexplored[s])
s ← s'
return a
```
Online DF-search, example

- Assume maze problem on 3x3 grid.
- $s' = (1,1)$ is initial state
- Result, unexplored (UX), unbacktracked (UB), ... are empty
- $S,a$ are also empty

S' = (1,1)

- GOAL-TEST((1,1))?  
  - $S$ not = $G$ thus false
- (1,1) a new state?
  - True
  - ACTION((1,1)) -> UX[(1,1)]
    - {RIGHT, UP}
  
- $s$ is null?
  - True (initially)
- UX[(1,1)] empty?
  - False
- POP(UX[(1,1)])->a
  - $A$=UP
- $s = (1,1)$
- Return a
Online DF-search, example

\[ S' = (2,1) \]

- \( \text{GOAL-TEST}((2,1)) \)?
  - \( S \) not = \( G \) thus false
- \( (2,1) \) a new state?
  - True
  - \( \text{ACTION}((2,1)) \rightarrow \text{UX}((2,1)) \)
    - \{DOWN\}
- \( s \) is null?
  - false \( (s=(1,1)) \)
  - \( \text{result}[$\text{UP},(1,1)] \leftarrow (2,1) \)
  - \( \text{UB}((2,1)) = \{(1,1)\} \)
- \( \text{UX}((2,1)) \) empty?
  - False
- A=DOWN, \( s=(2,1) \) return A

\[ S' = (1,1) \]

- \( \text{GOAL-TEST}((1,1)) \)?
  - \( S \) not = \( G \) thus false
- \( (1,1) \) a new state?
  - false
- \( s \) is null?
  - false \( (s=(2,1)) \)
  - \( \text{result}[$\text{DOWN},(2,1)] \leftarrow (1,1) \)
  - \( \text{UB}((1,1)) = \{(2,1)\} \)
- \( \text{UX}((1,1)) \) empty?
  - False
- A=RIGHT, \( s=(1,1) \) return A
Online DF-search, example

• GOAL-TEST((1,2))?
  – S not = G thus false
• (1,2) a new state?
  – True,
    UX[(1,2)]=[RIGHT, UP, LEFT]
• s is null?
  – false (s=(1,1))
  – result[RIGHT,(1,1)] <- (1,2)
  – UB[(1,2)]=((1,1))
• UX[(1,2)] empty?
  – False
• A=LEFT, s=(1,2) return A

S’=(1,2)
Online DF-search

- Worst case each node is visited twice.
- An agent can go on a long walk even when it is close to the solution.
- An online iterative deepening approach solves this problem.
- Online DF-search works only when actions are reversible.

Online local search

- Hill-climbing is already online
  - One state is stored.
- Bad performance due to local maxima
  - Random restarts impossible.
- Solution: Random walk introduces exploration (can produce exponentially many steps)
Online local search

• Solution 2: Add memory to hill climber
  – Store current best estimate $H(s)$ of cost to reach goal
  – $H(s)$ is initially the heuristic estimate $h(s)$
  – Afterward updated with experience (see below)

• Learning real-time $A^*$ (LRTA*)

Learning real-time $A^*$

function LRTA*-COST($s,a,s',H$) return an cost estimate
  if $s'$ is undefined the return $h(s)$
  else return $c(s,a,s') + H[s']$

function LRTA*-AGENT($s'$) return an action
  input: $s'$, a percept identifying current state
  static: result, a table indexed by action and state, initially empty
  $H$, a table of cost estimates indexed by state, initially empty
  $s,a$, the previous state and action, initially null

  if GOAL-TEST($s'$) then return stop
  if $s'$ is a new state (not in $H$) then $H[s'] \leftarrow h(s')$
  unless $s$ is null
    result[$a,s$] $\leftarrow s'$
    $H[s] \leftarrow$ MIN_LRTA*-COST($s,b$,result[$b,s$],$H$)
    $b \in$ ACTIONS($s$)
    $a$ $\leftarrow$ an action $b$ in ACTIONS($s$) that minimizes LRTA*-COST($s',b$,result[$b,s$],$H$)
  $s \leftarrow s'$
  return $a$
Recent developments

• Resource (memory, time) bounded search algorithms