The Map Coloring Problem

Neighbours countries must have different colours!
Problem Representation

<table>
<thead>
<tr>
<th><strong>What</strong> to represent?</th>
<th><strong>How</strong> to represent it?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Countries and Colors</td>
<td>Numbers or Symbols</td>
</tr>
<tr>
<td>Neighborhood Relation</td>
<td>Graph</td>
</tr>
<tr>
<td>Candidates</td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td>cand: Countries → Colors</td>
</tr>
<tr>
<td></td>
<td>e.g. (a→blue, b→red,...)</td>
</tr>
</tbody>
</table>

**Basic Functions:**

neighbored?(x, y)
- **args:** x, y are countries
- **value:** true iff x and y are neighbors

color(cand, x)
- **args:** cand is a candidate, x is a country
- **value:** cand(x) (the color of x w.r.t. cand)
Basic Functions

admissible?(cand)
  args: cand is a candidate
  val: true iff cand is admissible

for all x, y in countries:
  not (neighbored(x, y)
       and color(cand, x) = color(cand, y))

first_cand()
  val: the first candidate

next_cand(cand)
  arg: a candidate
  val: the next candidate

last_cand?(cand)
  args: cand is a candidate
  val: true iff cand is the last one
The Generate-and-Test Algorithm

gen_and_test() /* returns a solution or failure
    cand := first_cand();
    if admissible(cand) then return cand;
    while not last_cand(cand)
        cand := generate_next(cand);
        if admissible(cand) then return cand;
    return fail.
The State-Space Approach

- Root node = start state
- Node = state
- Arc = transition operator
- Leaf node (dead end)
- Leaf node (goal state)
The Routing Problem

Problem states are noncyclic paths in the road map
Finding The Successors for the Routing Problem

```plaintext
code
expand(node) /* updates the node

    succ := [];
    for c in Cities
        if (road(n.city,c) and not cyclic(n,c))
            then k := make_node();
                k.city := c;
                k.predecessor := n;
                succ := cons(k,succ)
        node.successors := succ.
```

Expanding a Node

\[\text{expand}(\text{node}) \quad /* \quad \text{updates the node}\\
\qquad \text{succ} := [];\\n\qquad \text{for } i := 1 \text{ to } N \\
\quad \quad \text{newstate} := \text{operator}(i, \text{node.state}),\\n\quad \quad \text{if not } \text{cyclic}(\text{node}, \text{newstate})\\n\quad \quad \quad \text{then } k := \text{make_node}();\\n\quad \quad \quad k.\text{state} := \text{newstate};\\n\quad \quad \quad k.\text{predecessor} := n;\\n\quad \quad \quad \text{succ} := \text{cons}(k, \text{succ})\\n\qquad \text{node.successors} := \text{succ};\\n\quad \quad \text{return } \text{succ}.\\n\]

\[\text{cyclic}(\text{node}, \text{state}) \quad /* \quad \text{returns true iff state leads to a cycle}\\n\quad \text{return } (n \neq \text{nil} \quad \text{and}\\n\quad \quad ((\text{state} = \text{node.state}) \quad \text{or}\\n\quad \quad \quad \text{cyclic}(\text{node.predecessor}, \text{state})))\]
The Search Tree: Basic Functions

root()  
    value: the root node

goal?(node)  
    arg.: a node  
    value: true, iff node is a goal node

expand(node)  
    arg.: a node  
    value: list of successors of node
Depth First Search
Breadth First Search
Depth-First-Search

dfs() /* returns goal node or failure 
openList := [root()];
while openList <> []
    actnode := delete_first(openList);
    if goal?(actnode) then return actNode;
    openList := append(expand(actnode),openList);
return fail.

Breadth-First-Search

bfs() /* returns goal node or failure 
openList := [root()];
while openList ≠ []
    actnode := delete_first(openList);
    if goal?(actnode) then return actNode;
    openList := append(openList,expand(actnode));
return fail.
Size of the OpenList

- depth-first-search
- breadth-first-search
Iterative Deepening

dfs(M)  /* M is maximal search depth
    openList := [root()];
    while openList ≠ []
      actnode := delete_first(openList);
      if goal?(actnode) then return actNode;
      if depth(actnode) ≤ M
        then succ := expand(actNode)
        else succ := [];
      openList := append(succ, rest(openList));
    return fail.

id()   /* Iterative Deepening
    M := 1;
    while (dfs(M) = fail)
      M := M+1
    return dfs(M)
## Properties of Search Methods

<table>
<thead>
<tr>
<th>Property</th>
<th>DFS</th>
<th>BFS</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure of OpenList</td>
<td>Stack</td>
<td>Queue</td>
<td>Stack</td>
</tr>
<tr>
<td>Complete?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Optimal?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Time Complexity</td>
<td>$O(b^n)$</td>
<td>$O(b^n)$</td>
<td>$O(b^n)$</td>
</tr>
<tr>
<td>Space Complexity</td>
<td>$O(b^n)$</td>
<td>$O(bn)$</td>
<td>$O(bn)$</td>
</tr>
</tbody>
</table>
Informed Search Methods: Heuristic Search

- Use knowledge about the problem to improve basic search: Make an intelligent guess!

- Example: Breaking the Cesar-Code

  ARTIFICIALINTELLIGENCE

  ↓+1

  BSUJGJDJBMJOUFMMJHFODF

  Solution: Compare with frequency distribution of letters in English texts

- A Heuristic is not a guarantee for a better solution
Heuristic Functions

- The Routing Problem: Choose the city that is next to the goal (straight line distance)
- Chess Playing: Choose the move that achieves the best-valued board state

heuristic function:

\[ f : \text{Nodes} \rightarrow \mathbb{N} \]

\( f(N) \) is an estimate for the distance of N to a goal node
Heuristic Functions: The 8-Puzzle

\[ h_1(S) = \text{number of misplaced tiles} \]
Heuristic Functions: The 8-Puzzle

\[ h_2(S) = \text{"Manhattan Block Distance"} \]
Assessing the Quality of a Heuristic Function

$h_2$ is "better" than $h_1$ if

$h_1(n_1) > h_1(n_2) \Rightarrow h_2(n_1) > h_2(n_2)$
Finding The Successors with Heuristics

```plaintext
expand(node, h)  /* expands the node, h is a functional argument */
succ := [];
for i := 1 to N /* N = number of operators */
    newstate := operator(i, node.state),
    if not cyclic(node, newstate)
        then k := make_node();
        k.state := newstate;
        k.predecessor := n;
        k.hval := h(newstate);  /* Store the heuristic value */
        succ := cons(k, succ)
node.successors := succ;
return succ.
```
Best-First-Search

bfs(h) /* returns goal node or failure
openList := [root(h)];
while openList <> []
    actnode := delete_first(openList);
    if goal?(actnode) then return actNode;
    succ := expand(actNode,h);
    for n in succ
        openList := insert_pq(n,openList,h)
    return fail.

insert_pq(Node,NodeList,h)
    /* inserts Node into NodeList according to its h-value
A heuristic function \( h \) is fair iff for each \( k \geq 0 \) there are only finitely many nodes \( N \) with \( h(N) \leq k \).
The Effective Branching Rate:
A Measure for the Efficiency of Heuristic Functions

\[ m = 1 + r + r^2 + \ldots + r^t \]
\[ r \approx m^{1/t} \]

- \( m \) = Number of Nodes Expanded
- \( t \) = Solution Depth
- \( r \) = Effective Branching Rate
# Efficiency of Different Strategies

Test Problem: The Eight-Puzzle

<table>
<thead>
<tr>
<th>$t$</th>
<th>effective branching rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
</tr>
<tr>
<td>2</td>
<td>2.45</td>
</tr>
<tr>
<td>4</td>
<td>2.87</td>
</tr>
<tr>
<td>6</td>
<td>2.73</td>
</tr>
<tr>
<td>8</td>
<td>2.80</td>
</tr>
<tr>
<td>10</td>
<td>2.79</td>
</tr>
<tr>
<td>12</td>
<td>2.78</td>
</tr>
<tr>
<td>14</td>
<td>2.83</td>
</tr>
</tbody>
</table>

$t$ = Solution Depth  
ID = Iterative Deepening  
BeFS($h$) = Best First Search Using Heuristics $h$
A cost function is monotonous if it increases towards the leaf nodes
Improved Uniform Cost Search

merge(oldNodes,newNodes) /* returns merged list
    result := oldNodes;
    for n in newNodes
        result := insert(n,result)
    return result.

insert(new,nodes) /* insert single node into nodelist
    s := new.state;
    if new.cost < mincost(s)
        then mincost(s) := new.cost;
        nodes := nodes - [k in nodes | k.state = s];
        nodes := insert_pq(new,nodes)
    return nodes.
A*-Search

A* is UCS with

\[ f(N) = c(N) + h(N) \]

Function \( h \) is **admissible** iff

\[ h(N) \leq \text{actual\_cost}(N, G) \]

If \( h \) is admissible, then A* finds an optimal solution
Search Tree for IDA*
Iterative Deepening A* (IDA*)

ida*()
    m := 1;
    while (dfs(m,mnew) = fail)
        m := mnew;
    return dfs(m,mnew).

dfs(m,mnew)
    openList := [root()];
    mnew := root().cost;
    while openList <> []
        actnode := delete_first(openList);
        if goal?(actnode) then return actNode;
        if actNode.cost £ m
            then succ := expand(actNode)
        else succ := [];
        mnew := min(mnew,actNode.cost)
    return fail.
Search Tree for SMA*
Search Tree for SMA*
The Knapsack Problem

Starting Point for Iteration

10 → 7, 6

7, 6 → 7, 6

7, 6 → 6, 7

6, 7 → 6, 7

6, 7 → 2, 4

2, 4 → 2, 4

2, 4 → 2, 4

2, 4 → 4, 2

4, 2 → 4, 2

4, 2 → 4, 2

4, 2 → 4, 2

4, 2 → 4, 2
Hill Climbing

hc() /* returns a (locally) optimal node
actNode := root();
choose n in actNode.nb: n.gain > actNode.gain;
while (n ≠ nil)
    actNode := n;
    choose n in actNode.nb: n.gain > actNode.gain;
return actNode.
\textbf{Threshold accepting}

\begin{verbatim}
\texttt{ta() /* returns a (locally) optimal node}
    \begin{verbatim}
        T := init_threshold();
        actNode := root();
        \textbf{choose} n \textbf{in} actNode.nb:
            n.gain > actNode.gain - T;
        \textbf{while} (n \neq \texttt{nil})
            actNode := n;
            \textbf{choose} n \textbf{in} actNode.nb:
                n.gain > actNode.gain - T;
            T := decrease_threshold(T);
        \textbf{return} actNode.
    \end{verbatim}
\end{verbatim}
\end{verbatim}
The Great Deluge

deluge() /* returns a (locally) optimal node
    W := init_waterlevel();
    actNode := root();
    choose n in actNode.nb:
        n.gain > W;
    while (n ≠ nil)
        actNode := n;
        choose n in actNode.nb:
            n.gain > W;
        W := rise_waterlevel(W);
    return actNode.
Simulated Annealing

sa() /* returns a (locally) optimal node
    T := init_temp();
    actNode := root();
    while (T > eps)
        choose n in actNode.nb;
        delta := n.gain - actNode.gain;
        if delta > 0
            then actNode := n
        else r := random(0,1);
            if r < exp(delta/T) then actNode := n;
        T := temp_decrease(T);
    return actNode.