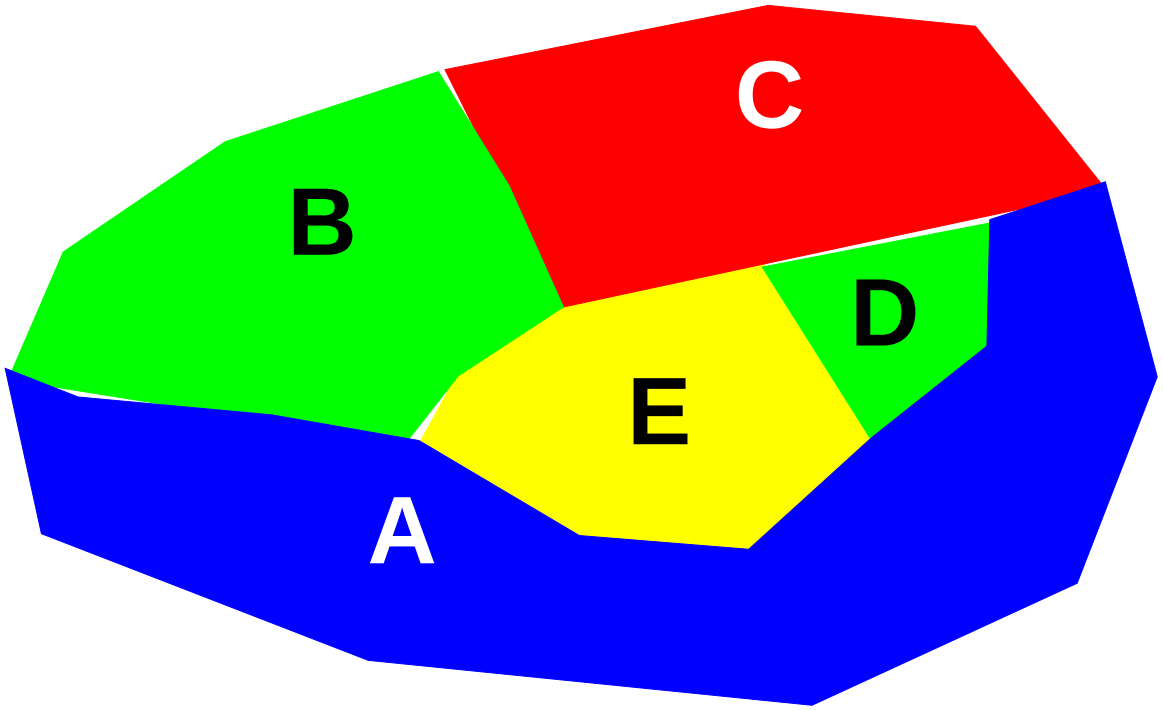
The Map Coloring Problem



Neighboured countries must have different colours!

Problem Representation

What to represent?	**How** to represent it?
Countries and Colors	Numbers or Symbols
Neighborhood Relation	Graph
Candidates	Function cand: Countries → Colors e.g. (a→blue, b→red,…)

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
```

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

```
first_cand()
   val: the first candidate
```

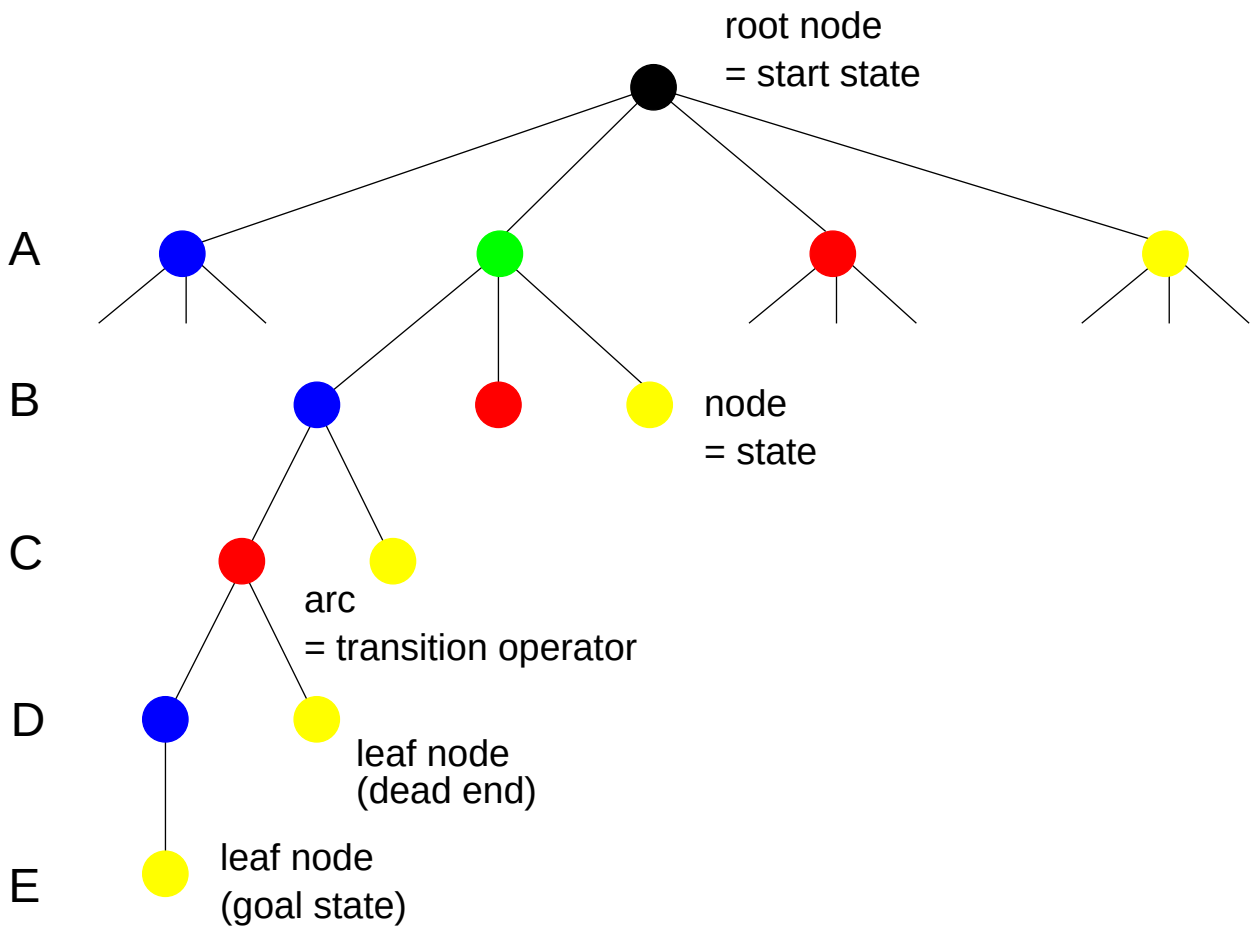
```
next_cand(cand)
   arg: a candidate
   val: the next candidate
```

```
last_cand?(cand)
   args: cand is a candidate
   val: true iff cand is the last one
```

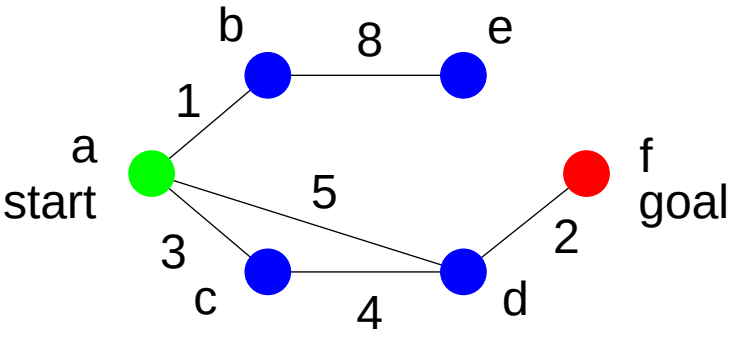
The Generate-and-Test Algorithm

```
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.
```

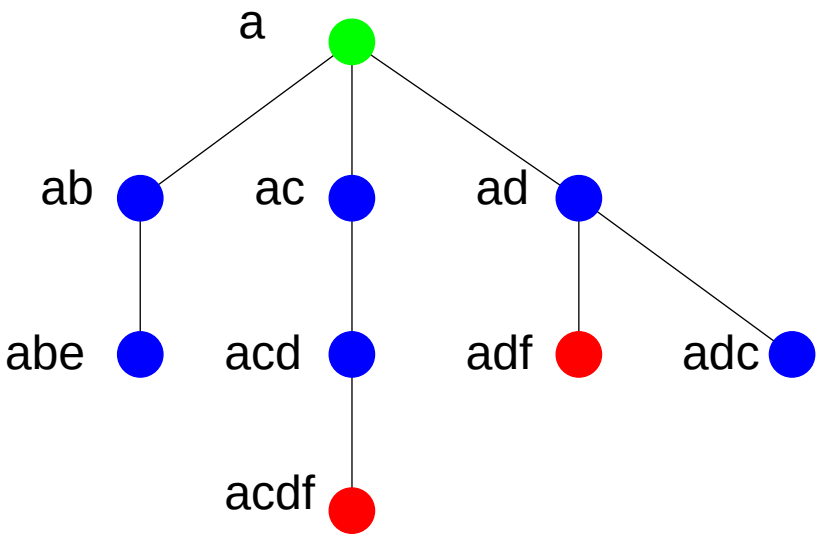
The State-Space Approach



The Routing Problem



Search Tree



Problem states are noncyclic paths in the road map

Finding The Successors for the Routing Problem

```
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.
```

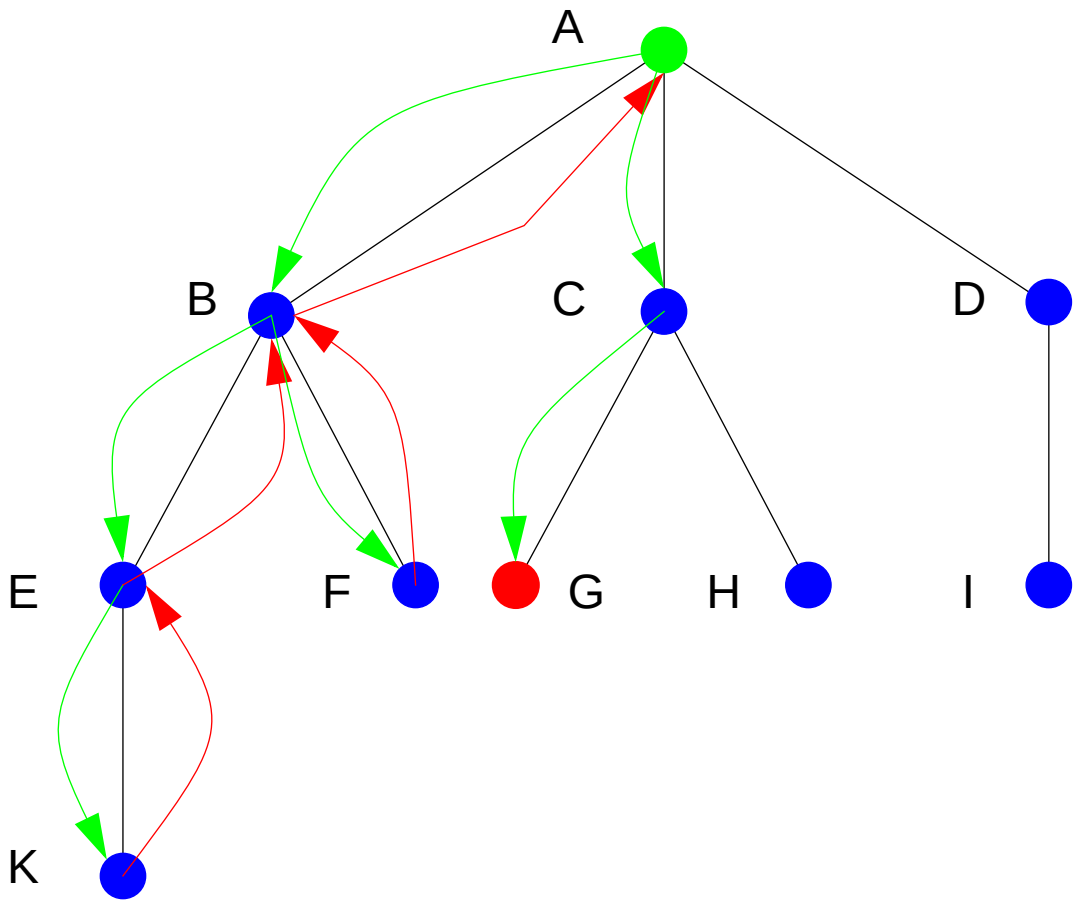
Expanding a Node

```
expand(node)                    /* updates the node
   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;
         succ := cons(k,succ)
   node.successors := succ;
   return succ.


cyclic(node,state)      /* returnes true iff state leads to a cycle
   return (n ≠ nil and
      ((state = node.state) or
      cyclic(node.predecessor,state)))
```

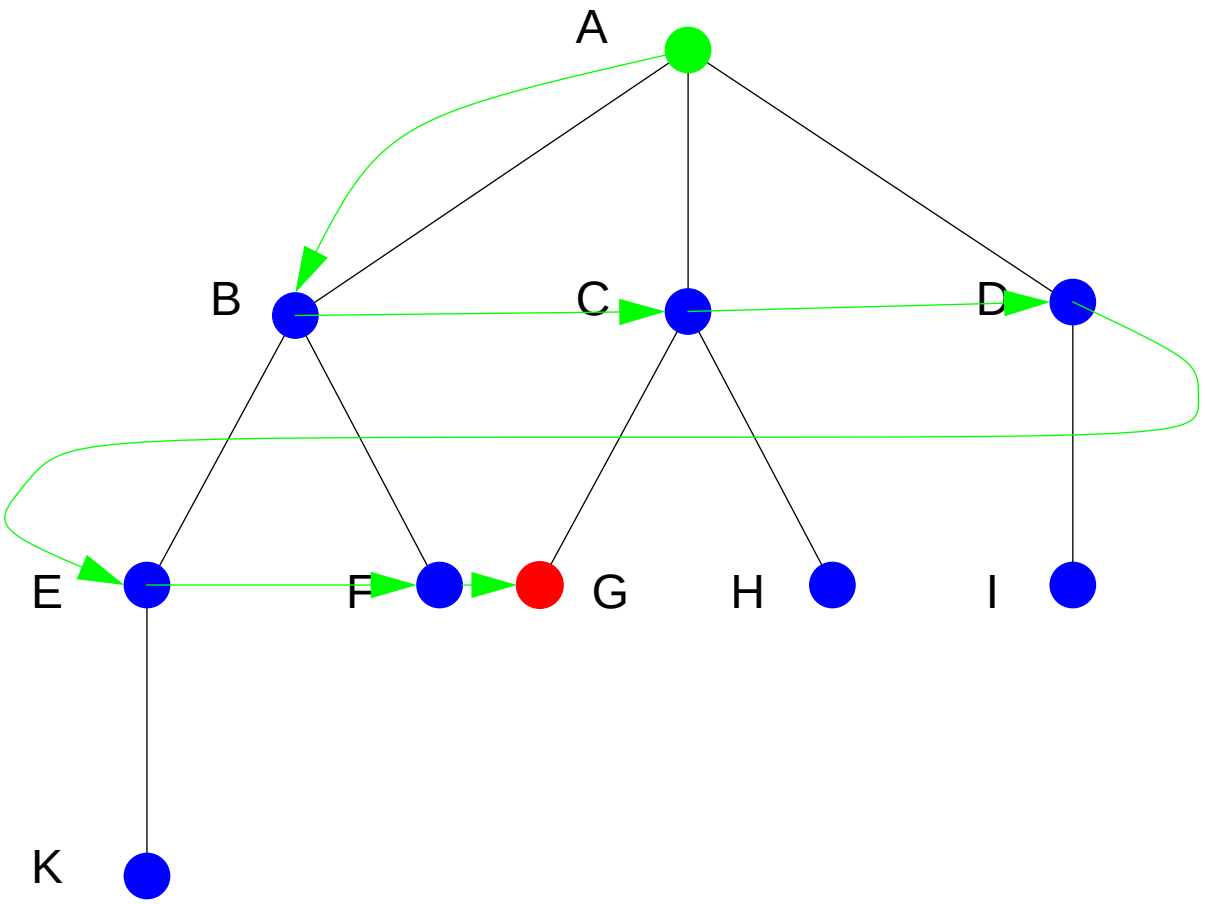
The Search Tree: Basic Functions

```
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



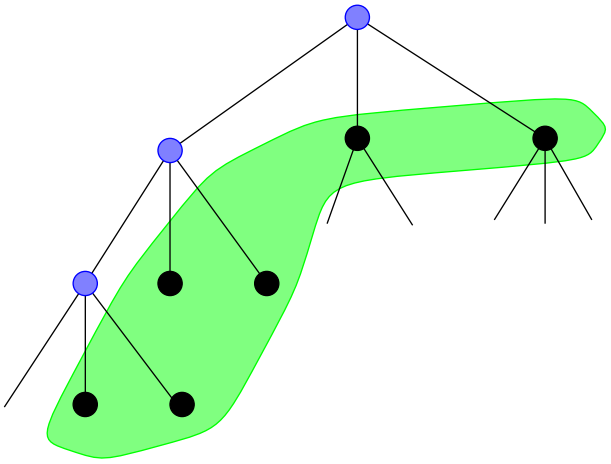
Depth-First-Search

```
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

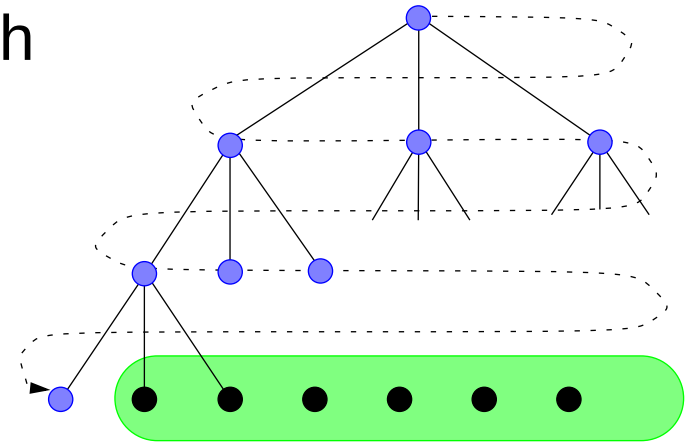
```
bfs()     /* returns goal node or failure
   openList := [root()];
   while openList ≠ []
      actnode := delete_first(openList);
      if goal?(actnode) then return actNode;
      openList := append(openList,expand(actnode));
   return fail.
```

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

Property	DFS	BFS	ID
Structure of OpenList	Stack	Queue	Stack
Complete?	no	yes	yes
Optimal?	no	yes	yes
Time Complexity	$O(b^n)$	$O(b^n)$	$O(b^n)$
Space Complexity	$O(b^n)$	O(bn)	O(bn)

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

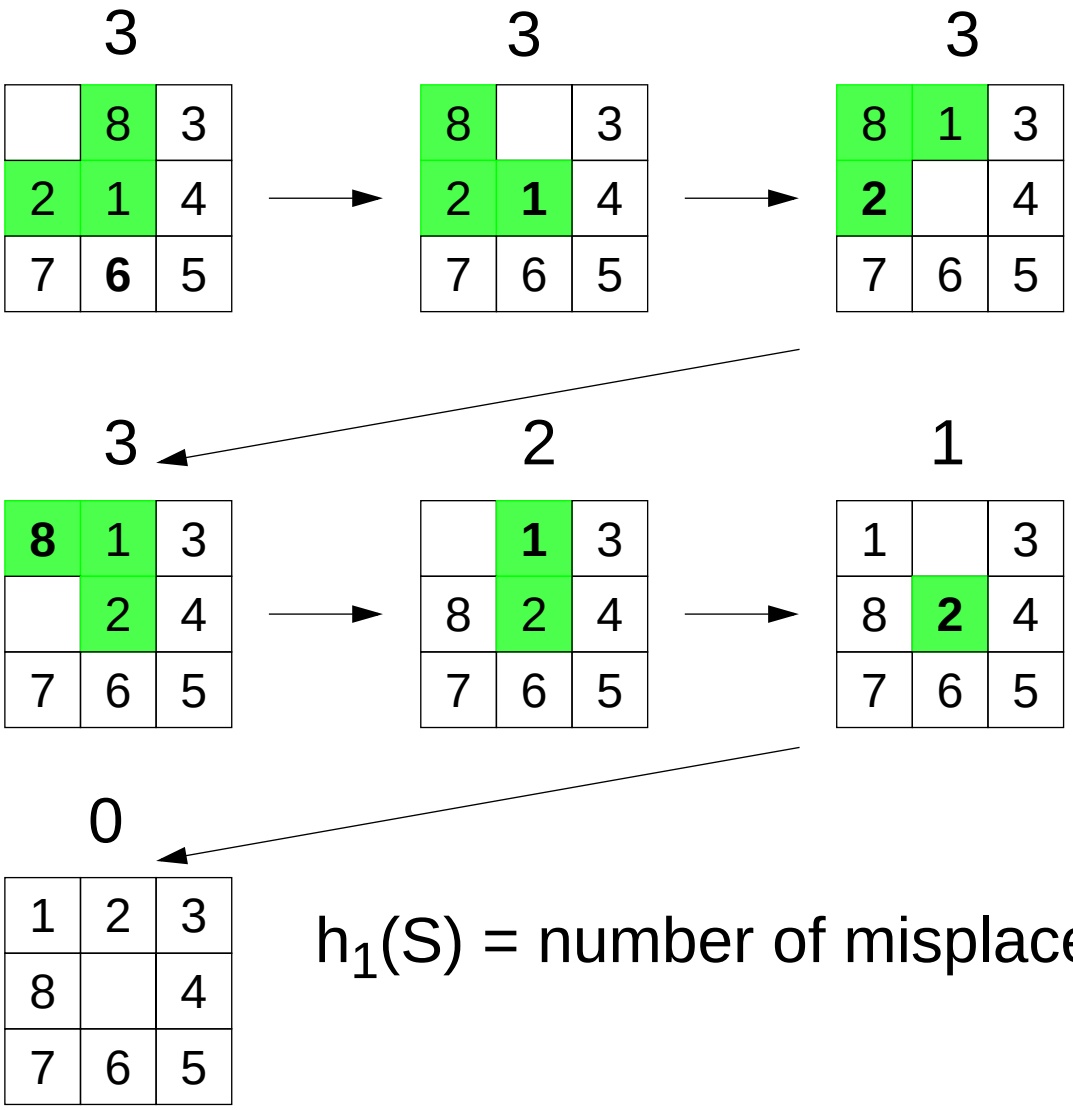
- The Routing Problem: Choose the city that is next to the goal (straight line distance)
- Chess Playing: Choose the move that achieves the best-valued board state

heuristic function:

$$f : \text{Nodes} \rightarrow \mathbb{N}$$

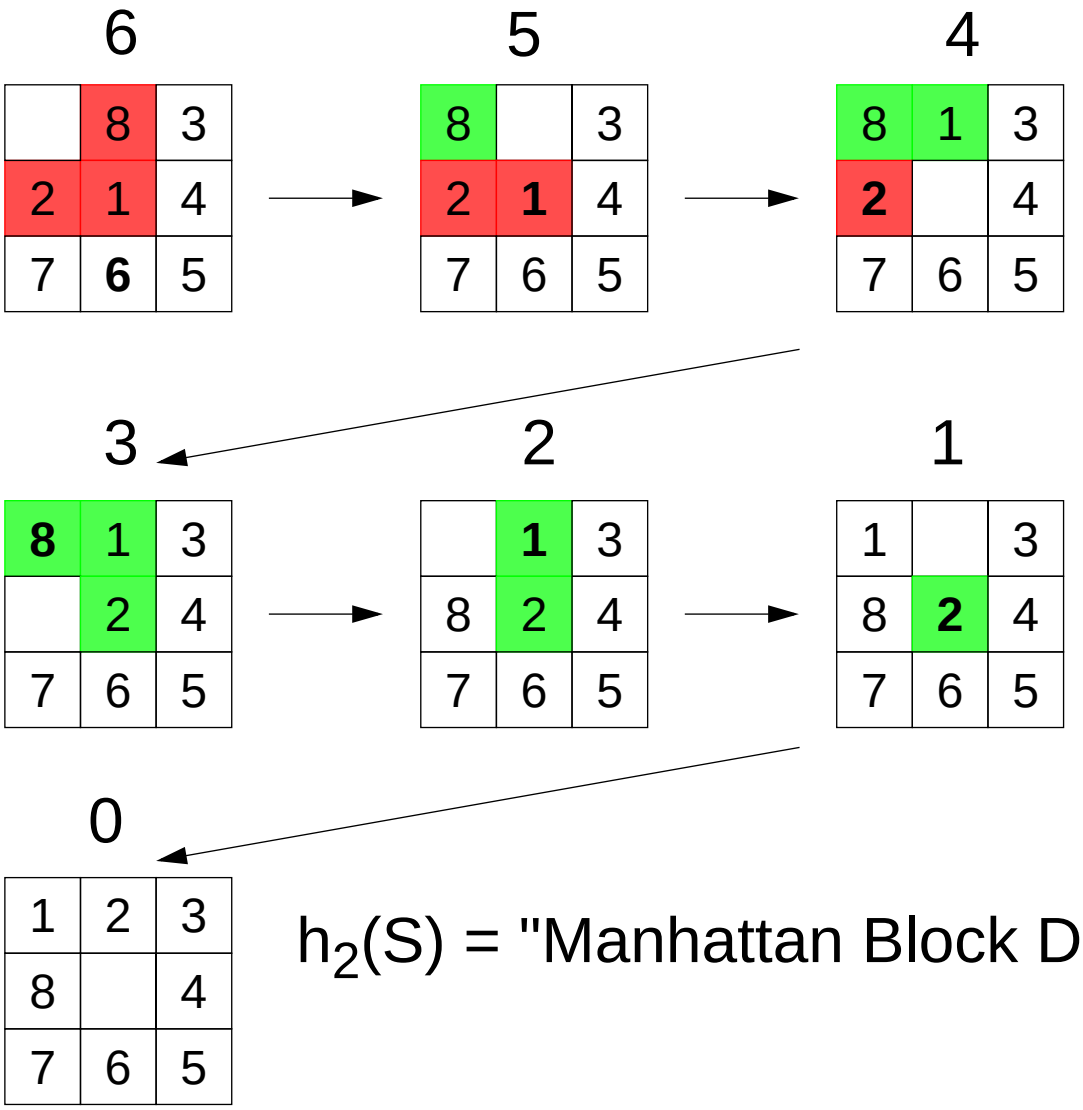
f(N) is an estimate for the distance of N to a goal node

Heuristic Functions: The 8 -Puzzle



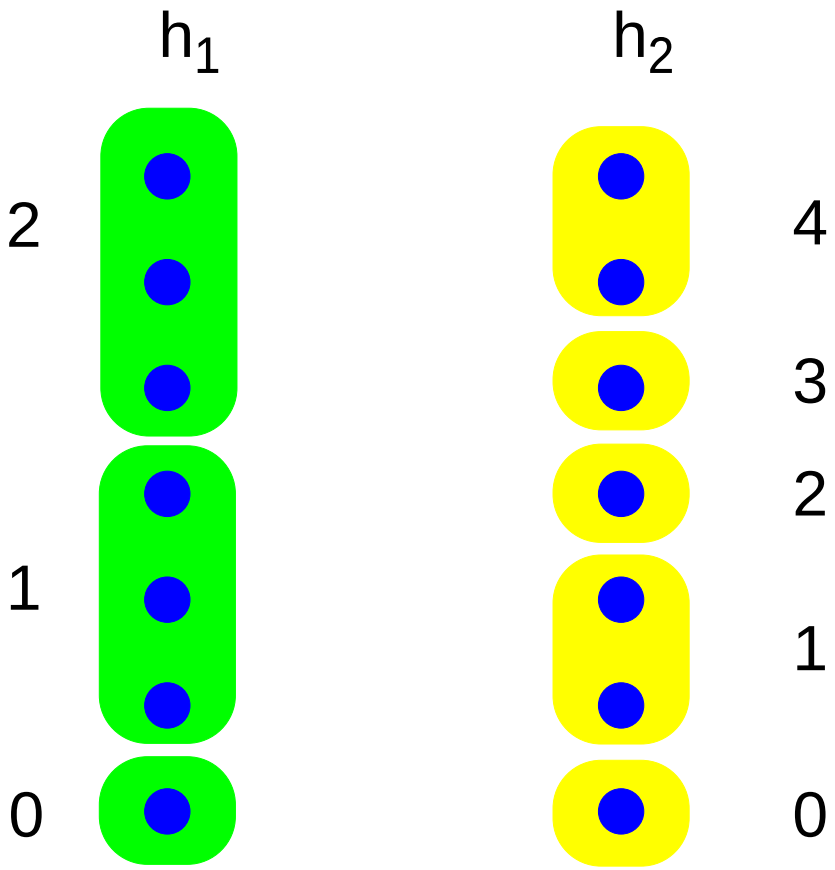
$h_1(S)$ = number of misplaced tiles

Heuristic Functions: The 8 -Puzzle



$h_2(S)$ = "Manhattan Block Distance"

Assessing the Quality of a Heuristic Function



h_2 is "better" than h_1 if

$$h_1(n_1) > h_1(n_2) \Rightarrow h_2(n_1) > h_2(n_2)$$

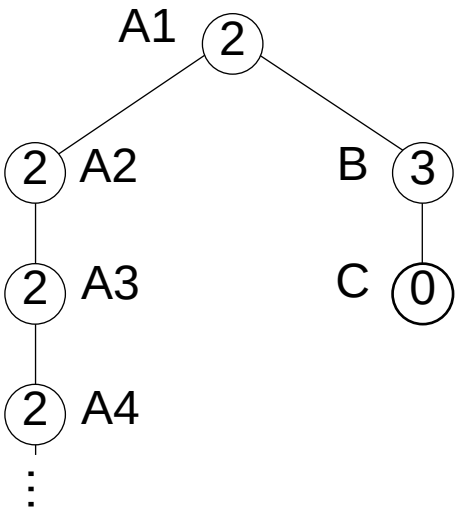
Finding The Successors with Heuristics

```
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.
```

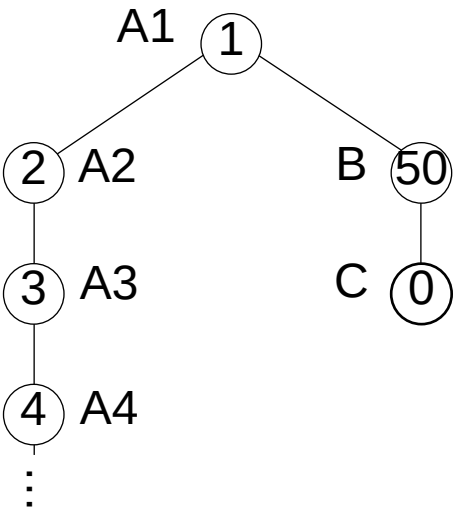
Best-First-Search

```
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
```

Fair and Unfair Heuristics



unfair heuristic function

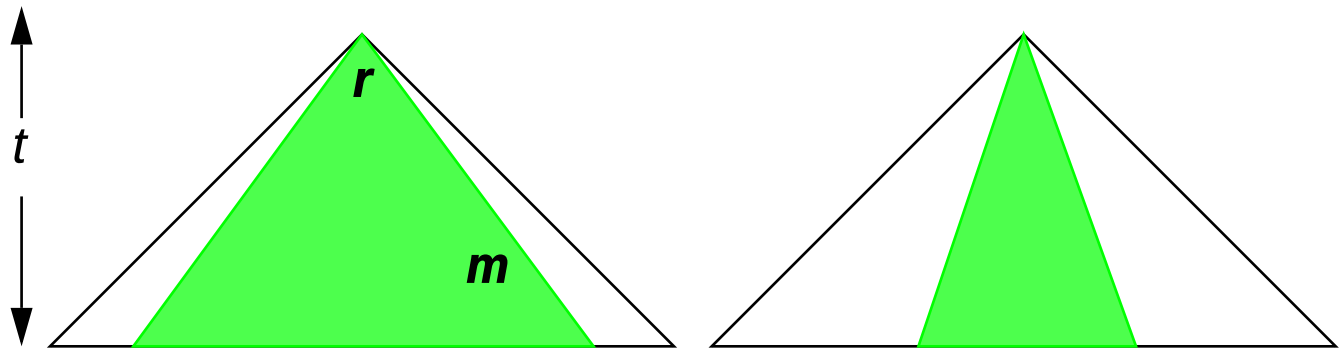


fair heuristic function

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

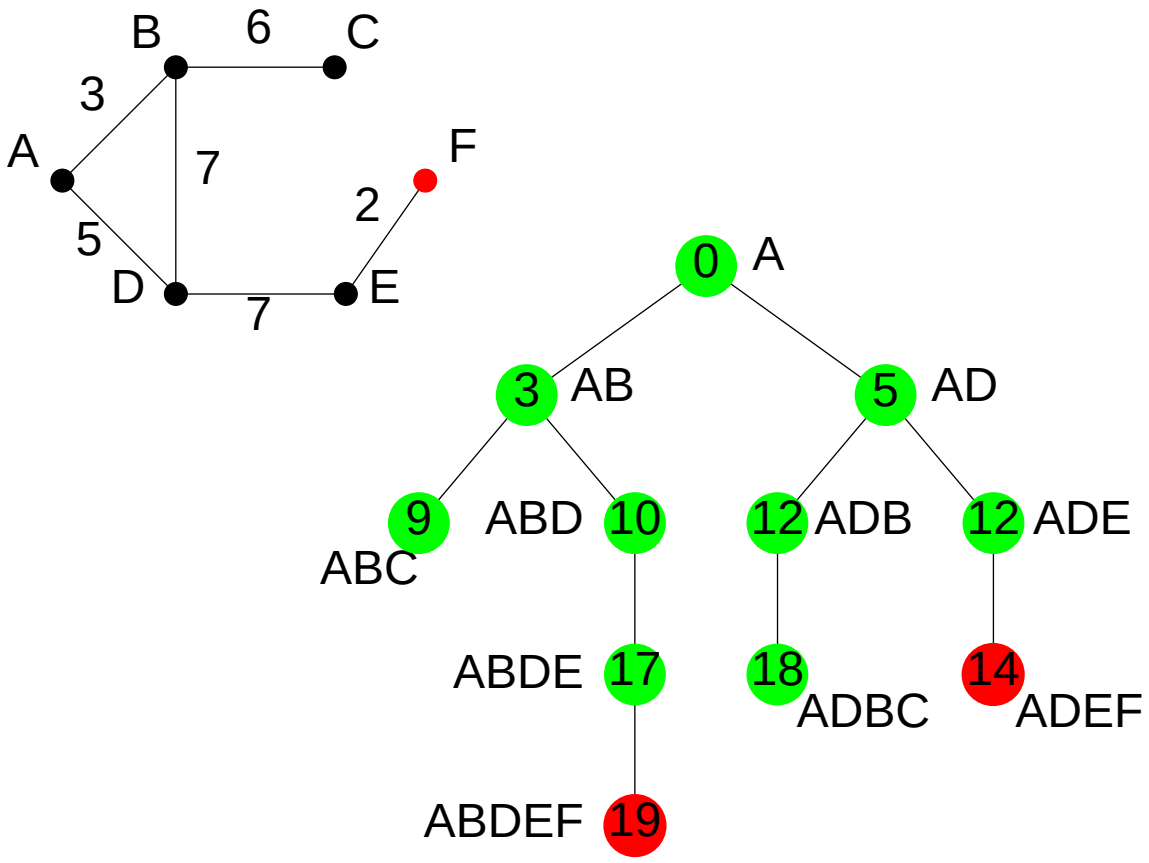
	effective branching rate		
t	ID	BeFS(h_1)	BeFS(h_2)
2	2,45	1,79	1,79
4	2,87	1,48	1,45
6	2,73	1,34	1,30
8	2,80	1,33	1,24
10	2,79	1,38	1,22
12	2,78	1,42	1,24
14	2,83	1,44	1,23

t = Solution Depth

ID = Iterative Deepening

BeFS(*h*) = Best First Search Using Heuristics *h*

Search Tree with Cost Function



A cost function is monotonous if it increases towards the leaf nodes

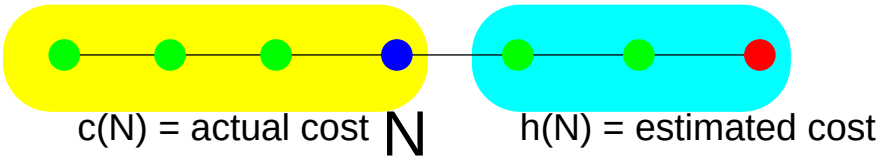
Improved Uniform Cost Search

```
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

$$f(N) = c(N) + h(N)$$

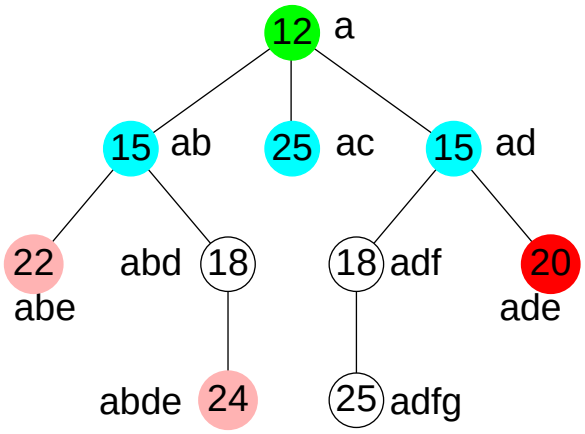


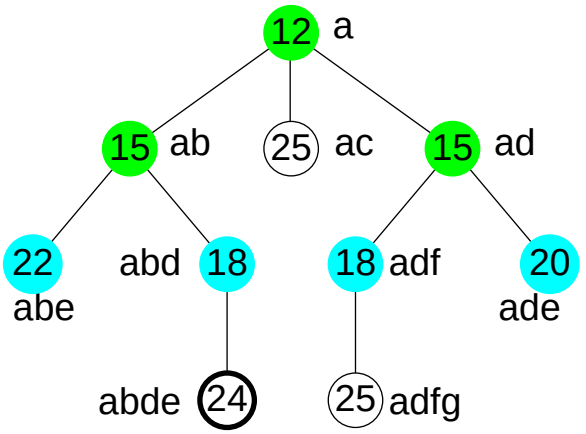
Function h is **admissible** iff

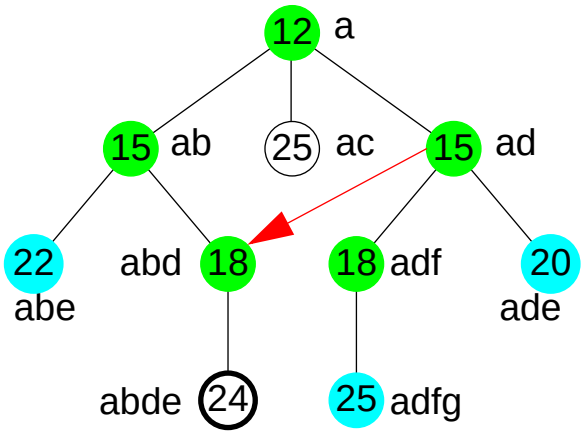
$$h(N) \leq \text{actual\_cost}(N,G)$$

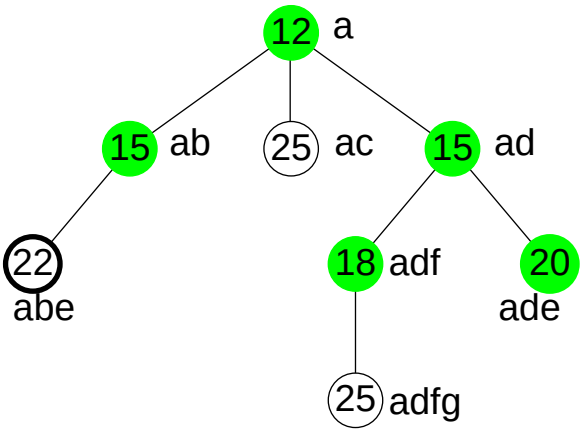
If h is admissible, then A* finds an optimal solution

Search Tree for IDA*





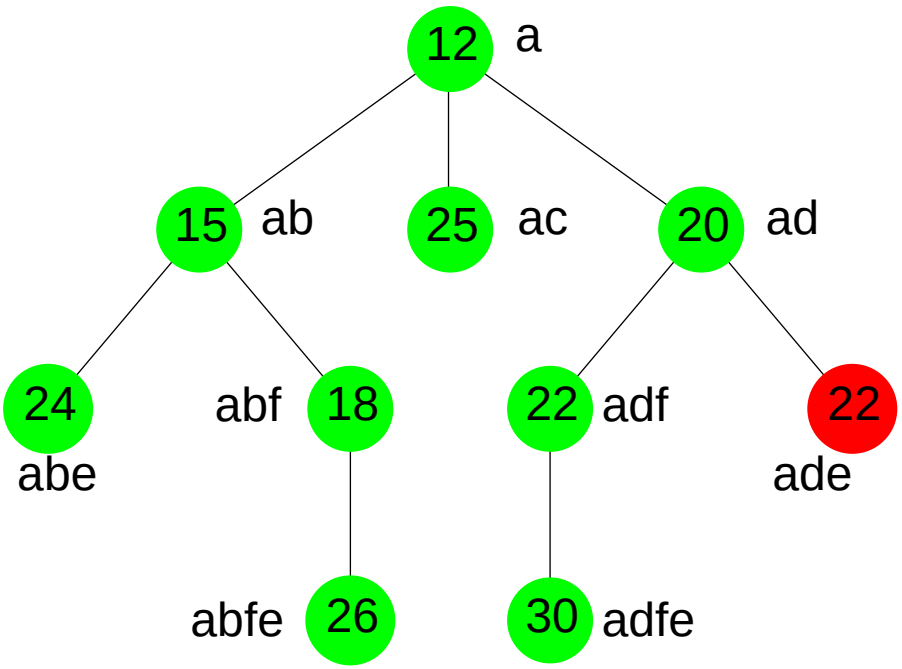




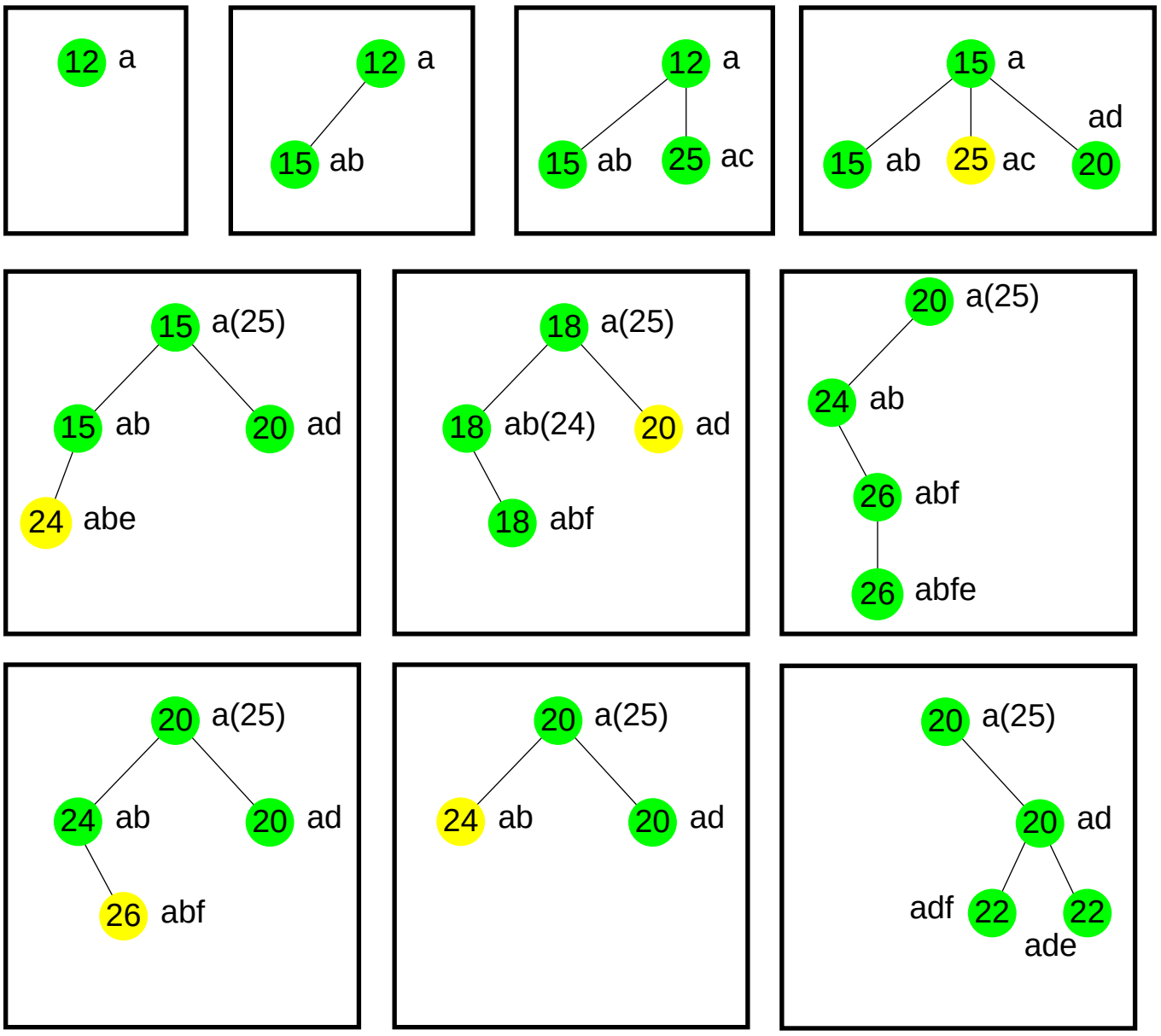
Iterative Deepening A* (IDA*)

```
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.
```

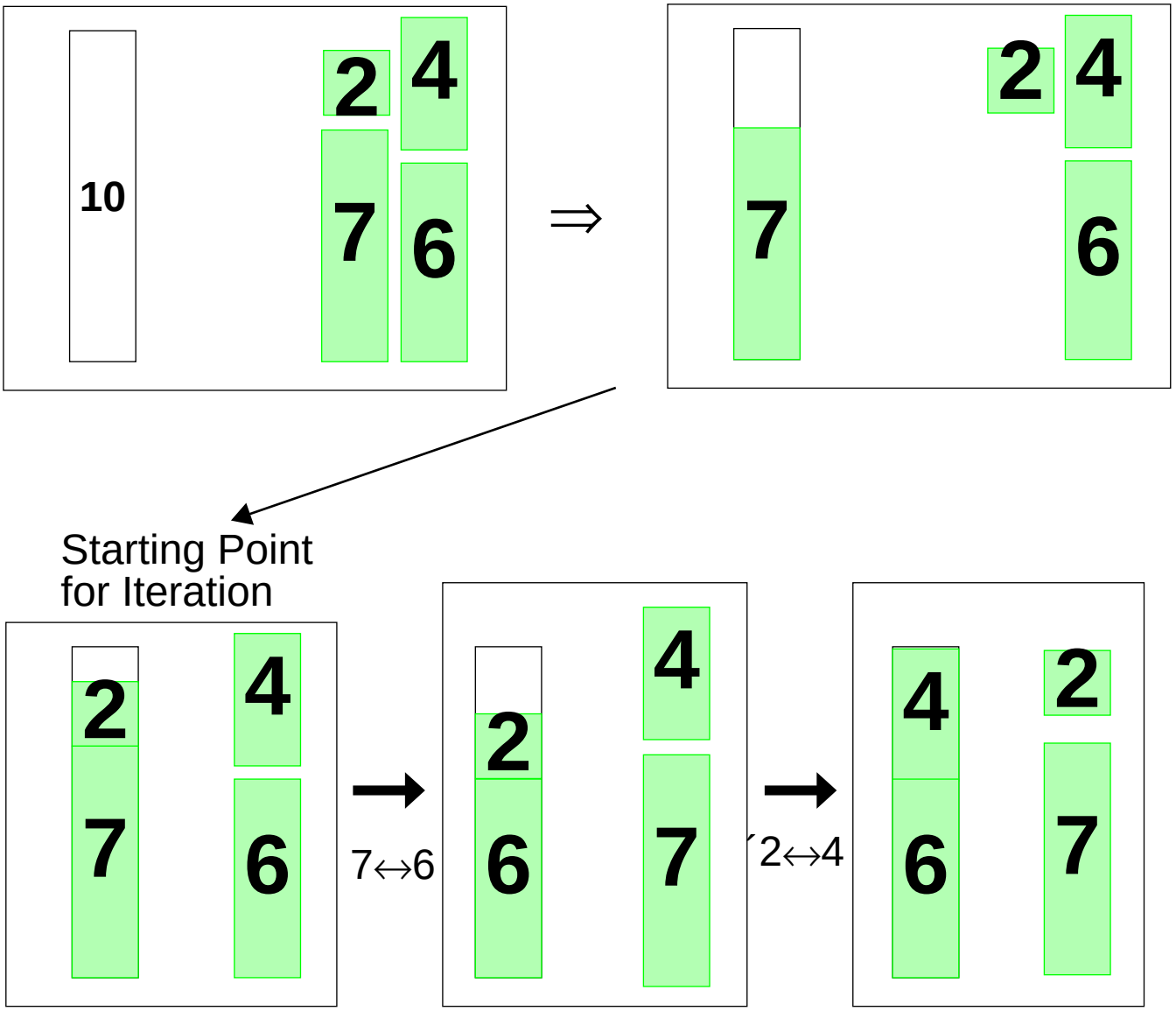
Search Tree for SMA*



Search Tree for SMA*



The Knapsack Problem



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.
```

Threshold accepting

```
ta() /* returns a (locally) optimal node
   T := init_threshold();
   actNode := root();
   choose n in actNode.nb:
      n.gain > actNode.gain - T;
   while (n ≠ nil)
      actNode := n;
      choose n in actNode.nb:
         n.gain > actNode.gain - T;
      T := decrease_threshold(T);
   return actNode.
```

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.
```