

# Solving Problems: Intelligent Search

**Instructor: B. John Oommen**

*Chancellor's Professor*

*Fellow: IEEE; Fellow: IAPR*

School of Computer Science, Carleton University, Canada

The primary source of these notes are the slides of Professor Hwee Tou Ng from Singapore. **I sincerely thank him for this.**

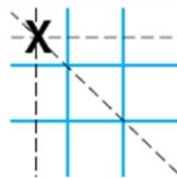
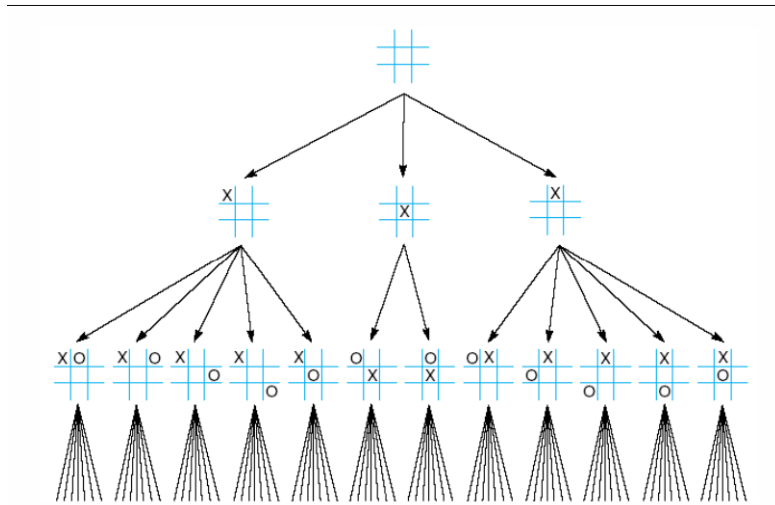
# Heuristic Search

- Problem with DFS and BFS: No way to **guide** the search
- Solution can be anywhere in tree.
- In the worst case all possible states will be traversed
- One “solution” to this problem
  - Probe the search space
  - Where is the final state **likely** to be
- This of course will be problem specific
- A function is usually created that evaluates:
  - How **good** the current solution is
  - This function is used to help **guide** the search process
- This guided search called a **Heuristic Search**

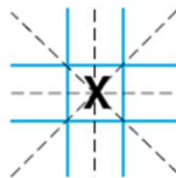
# A Heuristic

- Derived from the Greek: *heuriskein*: “to find”; “to discover”
- Has been used (and is sometimes still used) to mean:
  - “A process that may solve a given problem, but offers no guarantees of doing so” Newall, Shaw, & Simon 1963
- Heuristics can also be thought of as a “Rule of Thumb”
- Can refer to any technique that improves **average-case** but not necessarily **worst-case** performance
- Here: A function that provides an **estimate of solution cost**

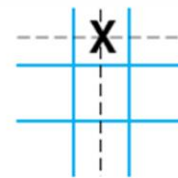
# Advantage of Heuristics



Three wins through a corner square

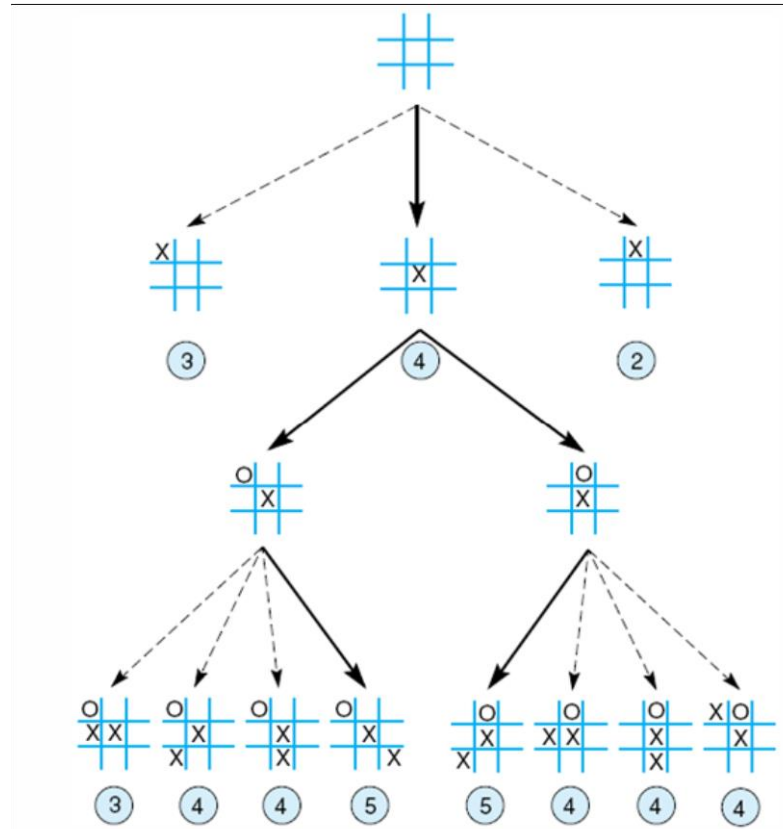


Four wins through the center square



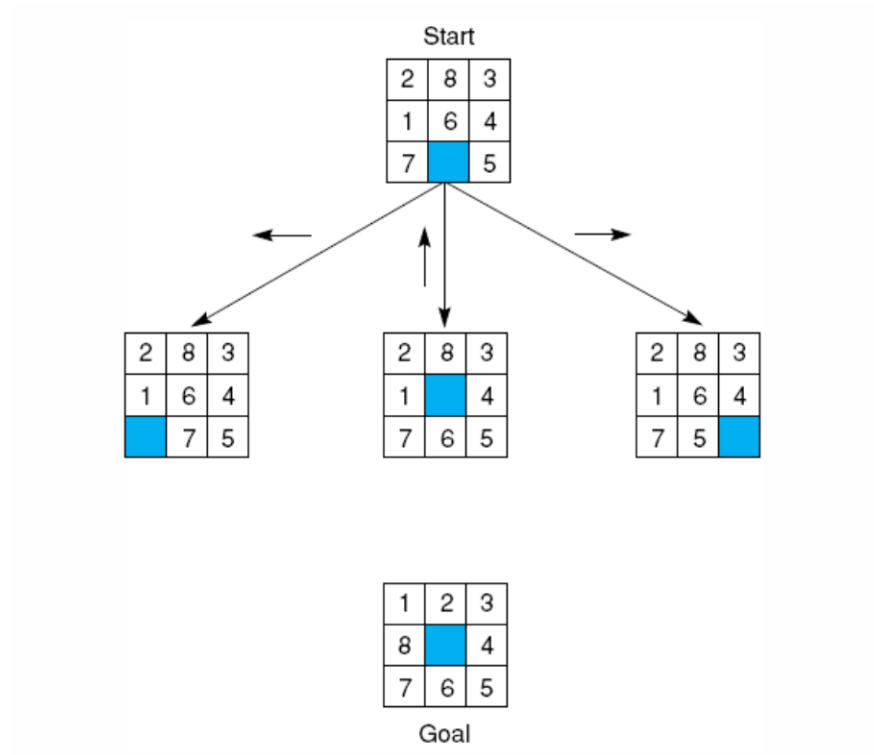
Two wins through a side square

# Advantage of Heuristics: Reduced State Space



# Performance of Heuristics

- Performance of several heuristics...



# Possible Heuristics

- **Count the tiles out of place:**
  - State with fewest tiles out of place is closer to the desired goal
- **Distance Summation:**
  - Sum all the distance by which the tiles are out of place
  - State with the shortest distance is closer to the desired goal
- **Count reversal Tiles:**
  - If two tiles are next to each other, and the goal requires their position to be swapped. The heuristic takes this into account by evaluating the expression  $(2 * \text{number of direct tiles reversal})$

<table border="1"> <tr><td>2</td><td>8</td><td>3</td></tr> <tr><td>1</td><td>6</td><td>4</td></tr> <tr><td>7</td><td>7</td><td>5</td></tr> </table>	2	8	3	1	6	4	7	7	5	5	6	0
2	8	3										
1	6	4										
7	7	5										
<table border="1"> <tr><td>2</td><td>8</td><td>3</td></tr> <tr><td>1</td><td>7</td><td>4</td></tr> <tr><td>7</td><td>6</td><td>5</td></tr> </table>	2	8	3	1	7	4	7	6	5	3	4	0
2	8	3										
1	7	4										
7	6	5										
<table border="1"> <tr><td>2</td><td>8</td><td>3</td></tr> <tr><td>1</td><td>6</td><td>4</td></tr> <tr><td>7</td><td>5</td><td>7</td></tr> </table>	2	8	3	1	6	4	7	5	7	5	6	0
2	8	3										
1	6	4										
7	5	7										
	Tiles out of place	Sum of distances out of place	2 x the number of direct tile reversals									

1	2	3
8	7	4
7	6	5

Goal

# Best-first Search

- **Idea:** use an **evaluation function**  $f(n)$  for each node
  - Estimate of “desirability”
  - Expand most desirable unexpanded node
- **Implementation:**  
Order the nodes in fringe in decreasing order of desirability
- **Special cases:**
  - Greedy best-first search
  - A\* search



# Best-first Search

- Combine BFS and DFS using a **heuristic function**
- Expand the branch that has the best evaluation under the heuristic function
- Similar to hill climbing (move in the best direction)
- But can go back to “discarded” branches

# Best-first Search Algorithm

- Initialize **OPEN** to initial state, **CLOSED** to Empty list
- Until a Goal is found or no nodes left in **Open** do:
  - Pick the best node in **OPEN**
  - Generate its successors, place node in **CLOSED**
  - For each successor do:
    - If not previously generated (not found in **OPEN** or **CLOSED**)
      - Evaluate
      - Add to **OPEN**

**OPEN:** Generated nodes who's children have not been evaluated yet

» Implemented as a priority queue (heap structure)

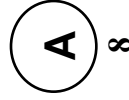
**CLOSED:** Nodes that have been examined

» Used to see if a node has been visited if searching a graph instead of a tree

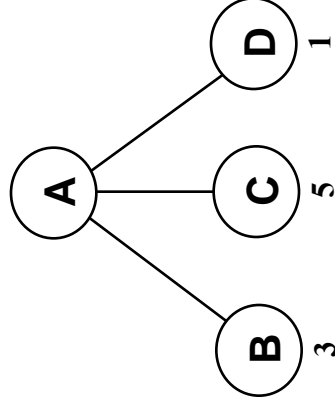
» Same as in DFS and BFS

# Example of BestFS

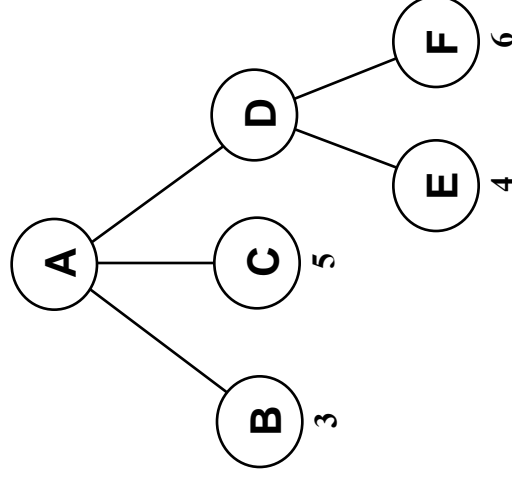
Step 1



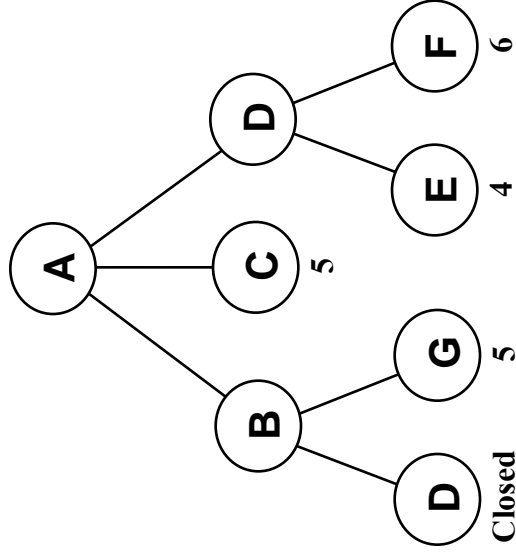
Step 2



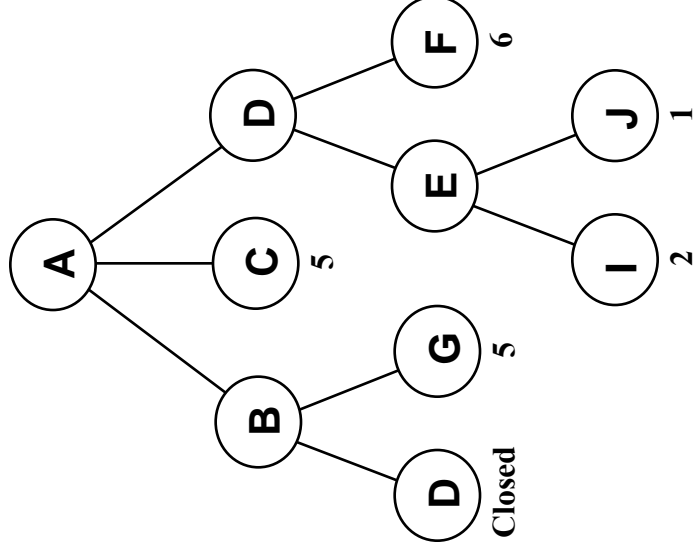
Step 3



Step 4



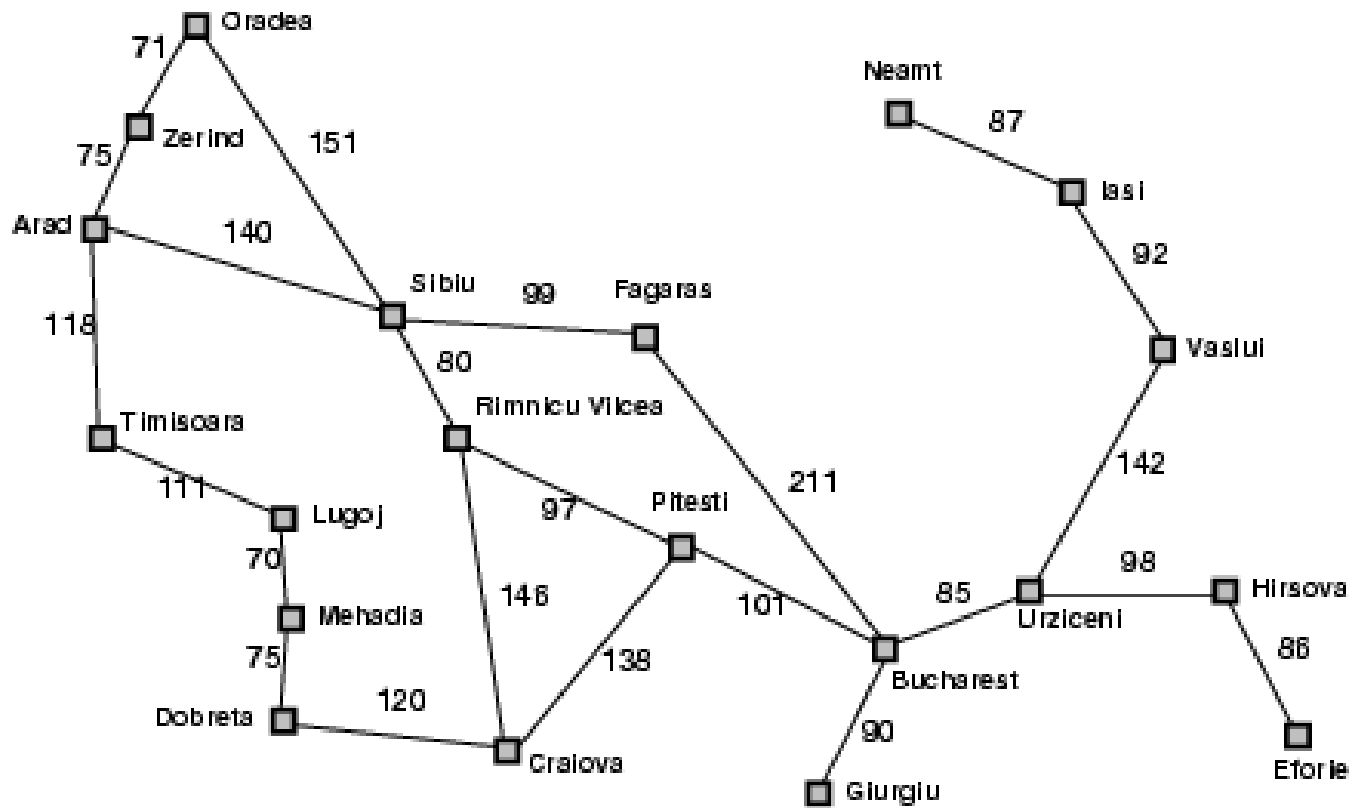
Step 5



# Greedy Best-first Search

- Evaluation function  $f(n) = h(n)$  (**heuristic**)
- An **estimate** of cost from  $n$  to *goal*
- $h_{SLD}(n)$  = straight-line distance from  $n$  to Bucharest
- Greedy Best-first Search expands the node that **appears** to be closest to goal

# Romania: Step Costs in Km



Straight-line distance  
to Bucharest

<b>Arad</b>	366
<b>Bucharest</b>	0
<b>Craiova</b>	160
<b>Dobreta</b>	242
<b>Eforie</b>	161
<b>Fagaras</b>	176
<b>Giurgiu</b>	77
<b>Hirsova</b>	151
<b>Iasi</b>	226
<b>Lugoj</b>	244
<b>Mehadia</b>	241
<b>Neamt</b>	234
<b>Oradea</b>	380
<b>Pitesti</b>	10
<b>Rimnicu Vilcea</b>	193
<b>Sibiu</b>	253
<b>Timisoara</b>	329
<b>Urziceni</b>	80
<b>Vaslui</b>	199
<b>Zerind</b>	374

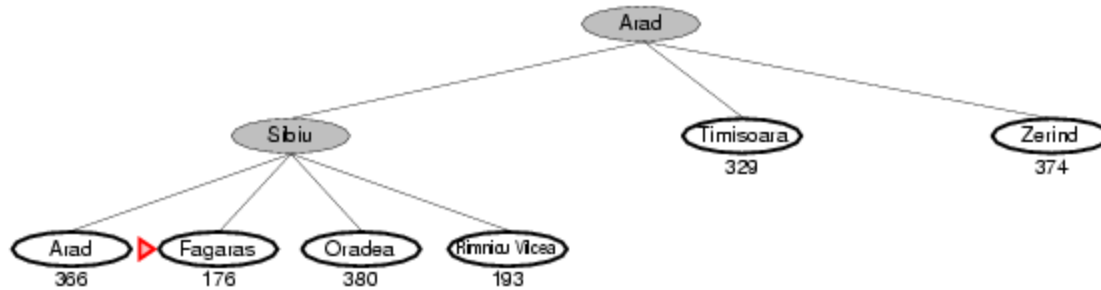
# Example: Greedy Best-first Search



# Example: Greedy Best-first Search

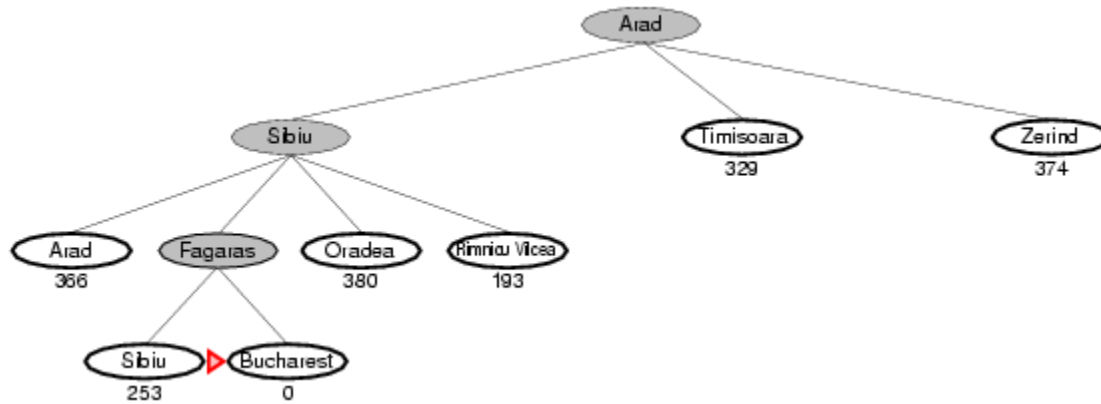


# Example: Greedy Best-first Search





# Example: Greedy Best-first Search



# Properties: Greedy Best-first Search

- **Complete?**

- No – can get stuck in loops
- lasi → Neamt → lasi → Neamt →

- **Time?**

- $O(b^m)$
- But a good heuristic can give dramatic improvement

- **Space?**

- $O(b^m)$
- Keeps all nodes in memory

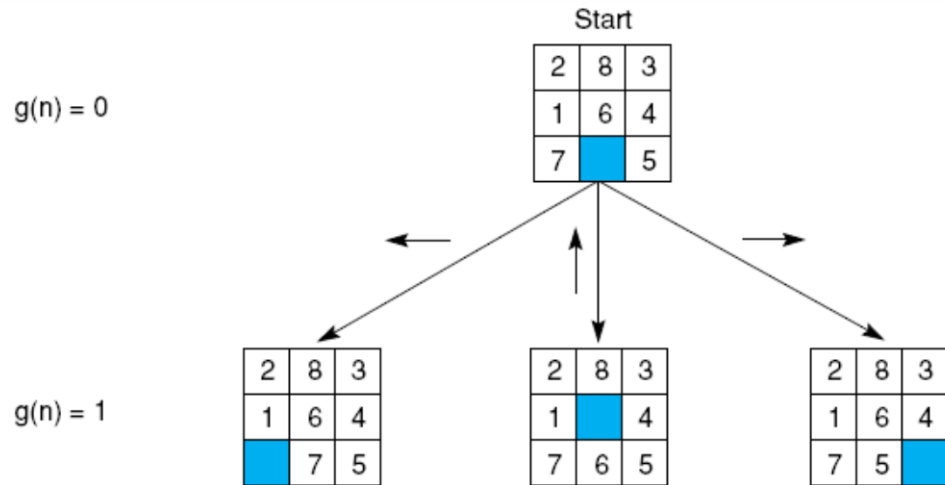
- **Optimal?**

- No

# A\* Search

- A modification of the Best-first Search
- Used when searching for the **Optimal** path
- Idea: Avoid expanding paths that are “expensive”
- The heuristic function **f(S)** is broken into two parts:
- Evaluation function  $f(n) = g(n) + h(n)$ 
  - $g(n)$  = Cost **so far** to reach  $n$
  - $h(n)$  = **Estimated cost** from  $n$  to goal
  - $f(n)$  = Estimated total cost of path through  $n$  to goal

# How A\* Works



Values of  $f(n)$  for each state,

**6**

**4**

**6**

where:

$$f(n) = g(n) + h(n),$$

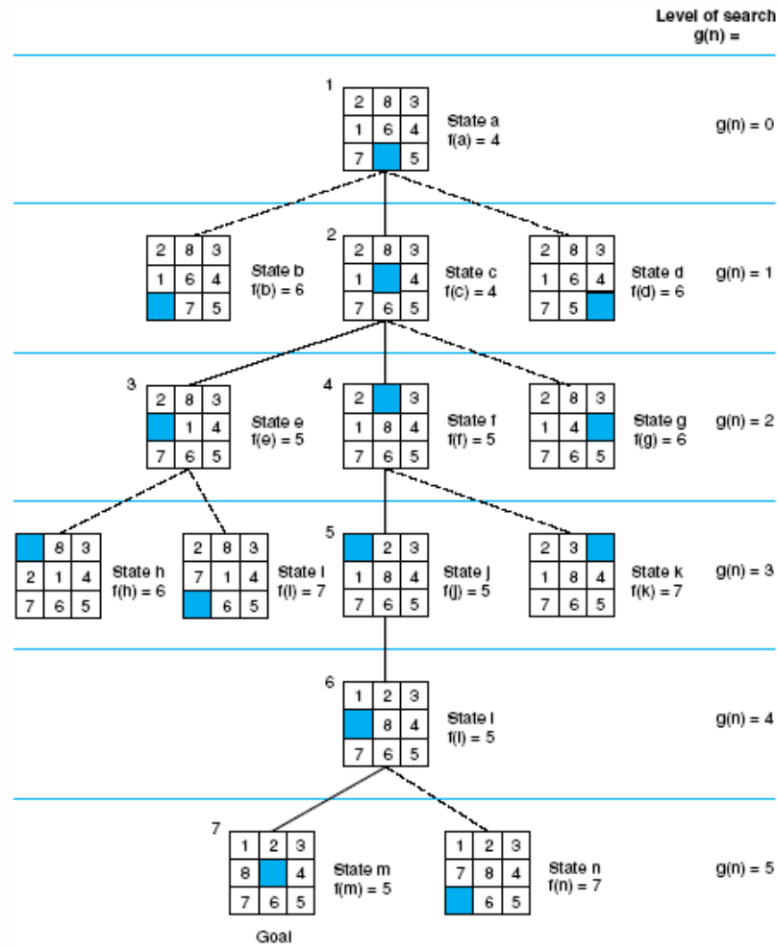
$g(n)$  = actual distance from  $n$   
to the start state, and

$h(n)$  = number of tiles out of place.

1	2	3
8		4
7	6	5

Goal

# How A\* Works



# A\* Algorithm

- Initialize **OPEN** to initial state
- Until a Goal is found or no nodes left in **OPEN** do:
  - Pick the best node in **OPEN**
  - Generate its successors (recording the successors in a list);
  - Place in **CLOSED**
  - For each successor do:
    - If not previously generated (not found in **OPEN** or **CLOSED**)
      - Evaluate, add to **OPEN** , and record its parent
    - If previously generated ( found in **OPEN** or **CLOSED**), and if the new path is better then the previous one
      - Change parent pointer that was recorded in the found node
    - If parent changed
      - Update the cost of getting to this node
      - Update the cost of getting to the children
        - Do this by recursively “regenerating” the successors using the list of successors that had been recorded in the found node
      - Make sure the priority queue is reordered accordingly

# Properties of A\*

- Becomes simple Best-first Search if  $g(S) = 0$  for every S
- When a child state is formed
  - $g(S)$  can be incremented by 1
  - Or be weighted based on the production system operator generated the state
- Is **Breadth-first Search** if  $g += 1$  per generation and  $h=0$  always

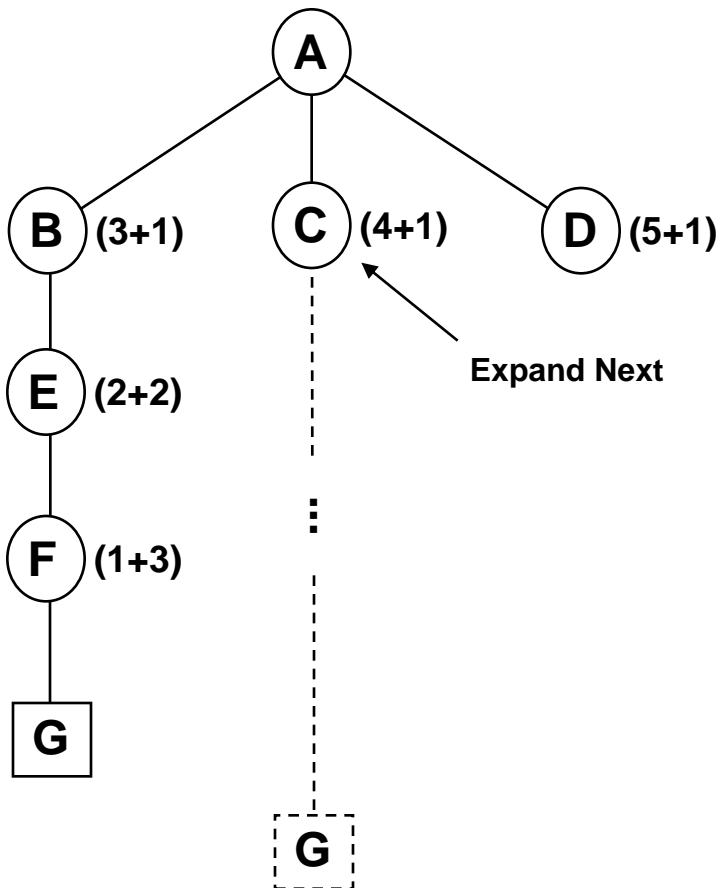
# Properties of A\*

- If  $h$  is the **perfect** estimator of the distance to the Goal (say,  $H$ )
  - A\* will immediately find and traverse the optimal path to the solution
  - Will need **NO** backtracking
- If  $h$  never **overestimates**  $H$ 
  - A\* will find an optimal path to the solution (if it exists)
  - Problem lies in **finding** such an  $h$

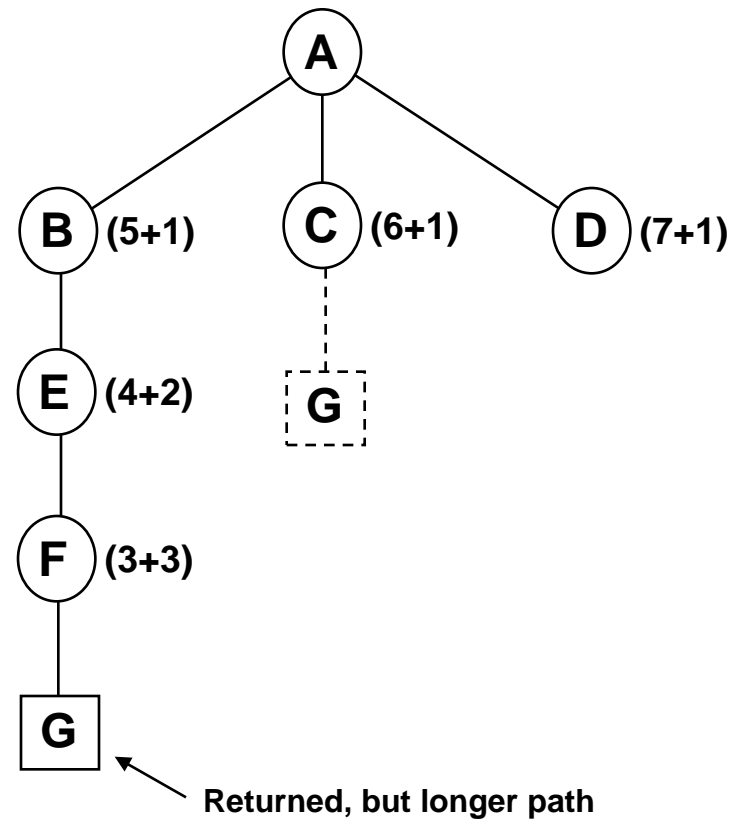


# h Under/Over Estimates H

## h Underestimates H



## h Overestimates H



Goal is G

# Importance of Heuristic Function

- If we have the exact Heuristic Function  $H$ 
  - The search gets solved optimally
- Exact  $H$  is usually **very hard** to find
  - In many cases it would be a solution to an NP problem in polytime
  - Which is probably not possible to compute in less time than it would take to do the exponential sized search
- Next best: Guarantee  $h$  underestimates distance to the  $Sol^n$ .
  - A minimum path to the Goal is then guaranteed

# Heuristic Function vs. Search Time

- The better the heuristic, the less searching
  - Improves the average time complexity
- However, to compute such a heuristic
  - Can figure out a good algorithm
  - Usually costs computation cycles
  - This could be used to process more nodes in the search
  - Trade-off between complex heuristics vs. more search done

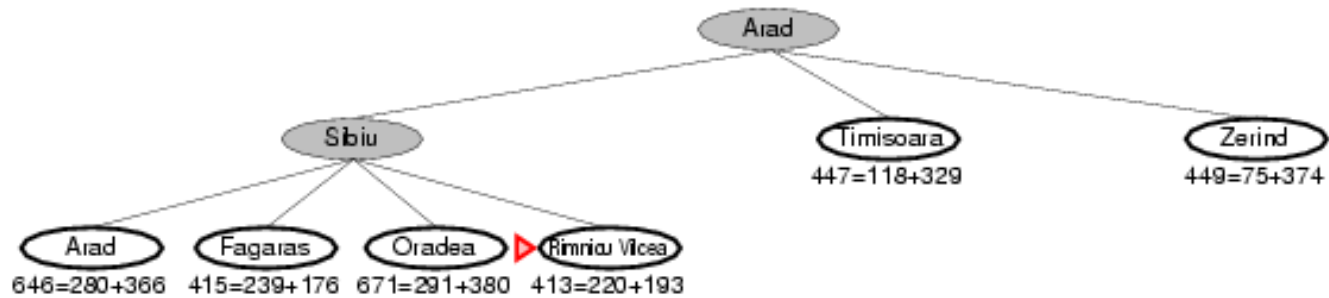
# Example: A\* Search

▶ Arad  
366=0+366

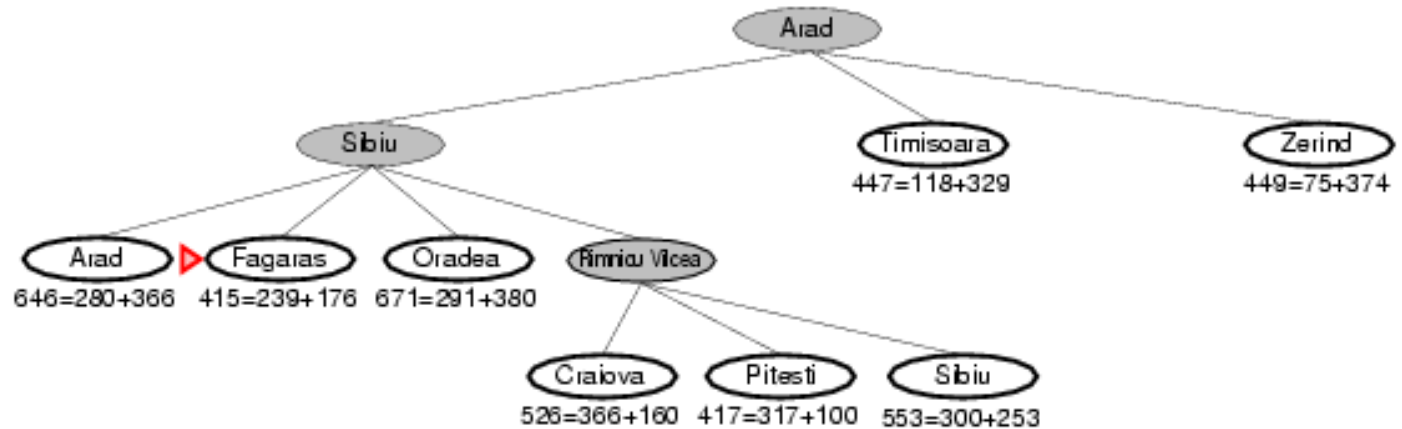
# Example: A\* Search



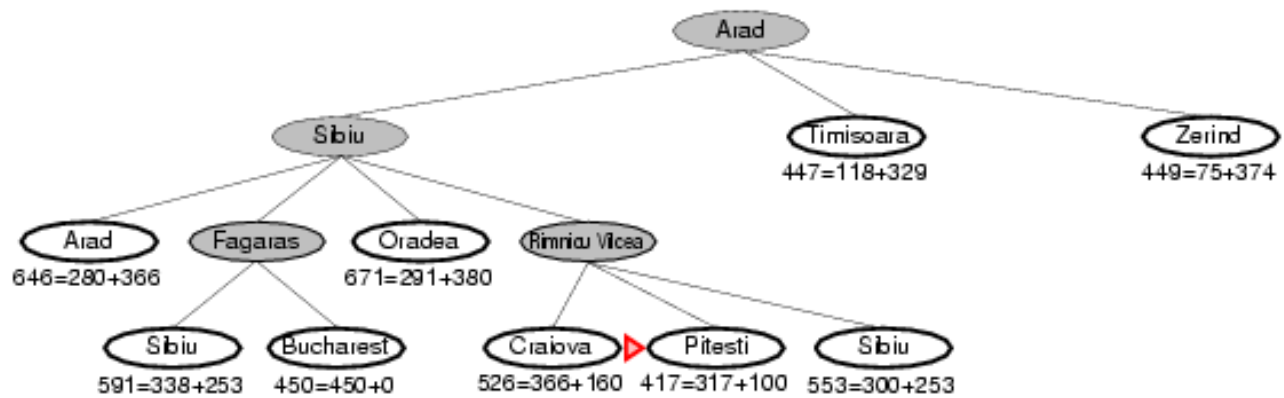
# Example: A\* Search



# Example: A\* Search

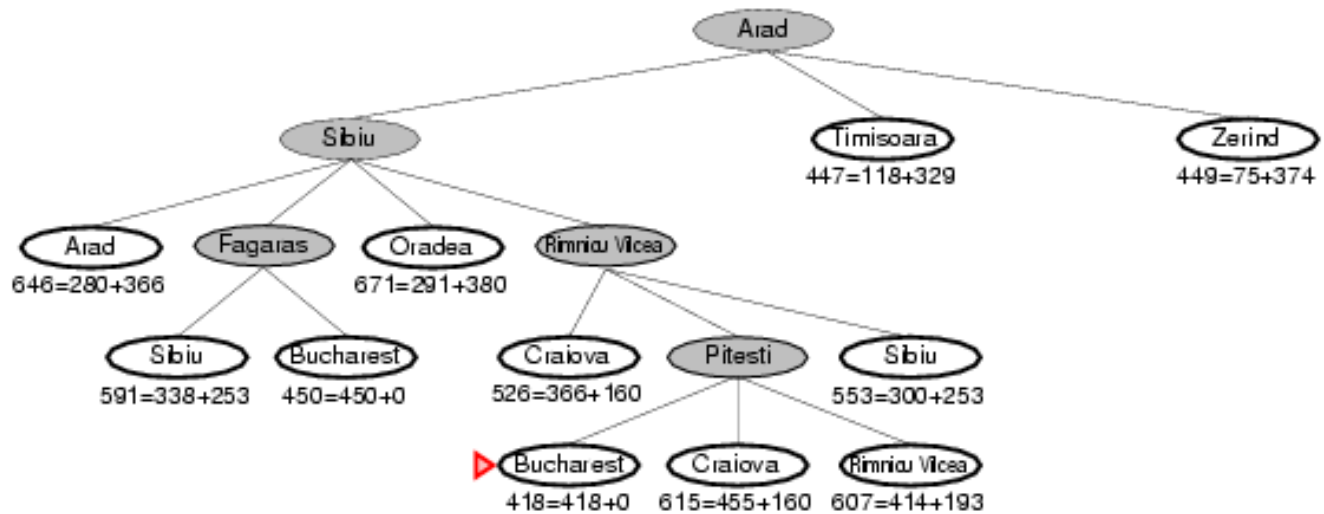


# Example: A\* Search





# Example: A\* Search



## Other Example: A\* Search

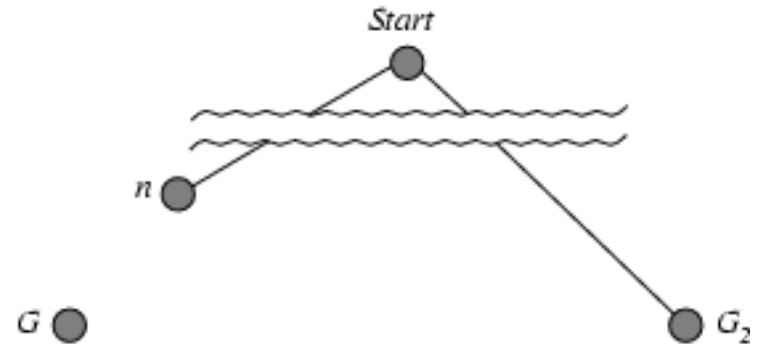
- Please see the other Powerpoint in the folder...

# 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 TREE-SEARCH is **optimal**

# Proof: Optimality of $A^*$

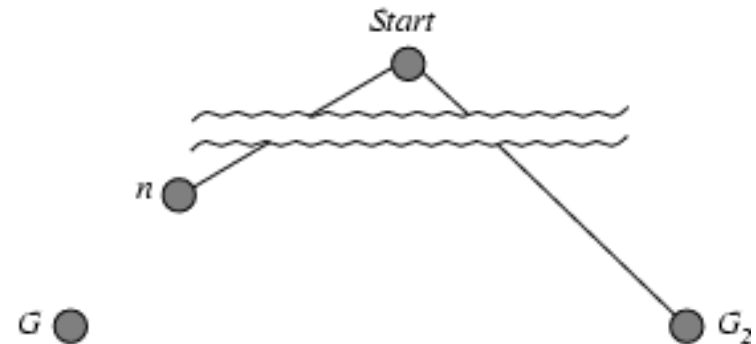
- Suppose some suboptimal goal  $G_2$  has been generated and is in the fringe.
- Let  $n$  be an unexpanded node in the fringe such that  $n$  is on a shortest path to an optimal goal  $G$ .



- $f(G_2) = g(G_2)$                       Since  $h(G_2) = 0$
- $g(G_2) > g(G)$                       Since  $G_2$  is suboptimal (2)
- $f(G) = g(G)$                          Since  $h(G) = 0$  (3)
- $f(G_2) = g(G_2) > g(G)$  (from (2)) =  $f(G)$  (from (3))
- $f(G_2) > f(G)$                          From above

# Proof: Optimality of $A^*$

- 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  $G$ .



- $f(G_2) > f(G)$
- $h(n) \leq h^*(n)$
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$

Hence  $f(G_2) > f(n)$ .

From above

Since  $h$  is admissible

Thus  $A^*$  will never select  $G_2$  for expansion

# Consistent Heuristics

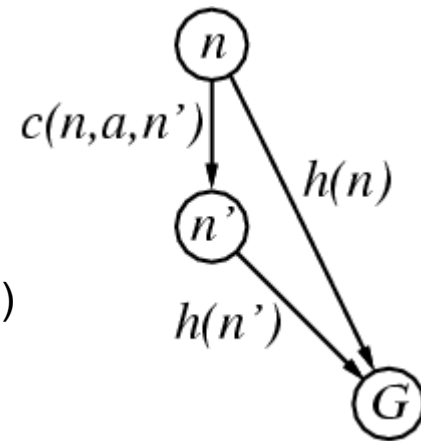
- A heuristic is **consistent** if for every node  $n$ , every successor  $n'$  of  $n$  generated by any action  $a$ ,

$$h(n) \leq c(n,a,n') + h(n') \quad (4)$$

- If  $h$  is consistent, we have

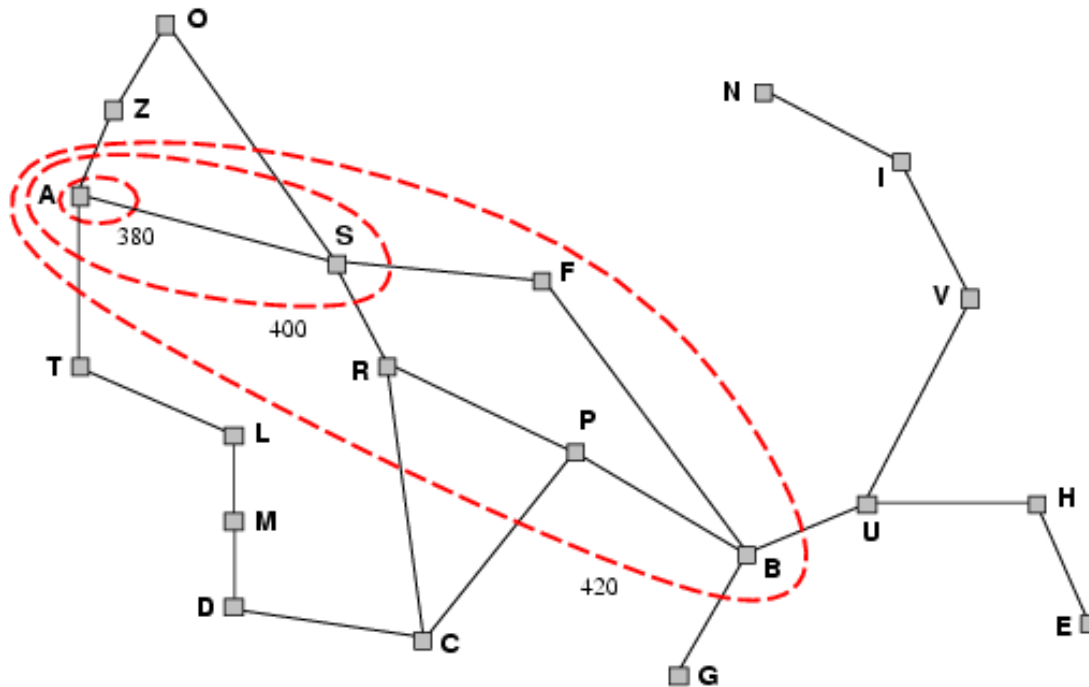
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \quad (\text{By (4)}) \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

- i.e.,  $f(n)$  is non-decreasing along any path.
- **Theorem:** If  $h(n)$  is consistent, A\* using GRAPH-SEARCH is optimal.
- Essentially since: At the very end –  $h(G) = 0$ .



# Optimality of A\*

- A\* expands nodes in order of increasing  $f$  value
- Gradually adds " $f$ -contours" of nodes
- Contour  $i$  has all nodes with  $f=f_i$ , where  $f_i < f_{i+1}$



# Properties of A\*

- **Complete?**
  - Yes (unless there are infinitely many nodes with  $f \leq f(G)$ )
- **Time?**
  - Exponential
- **Space?**
  - Keeps all nodes in memory
- **Optimal?**
  - Yes



# Admissible Heuristics

## The 8-puzzle:

- $h_1(n)$  = number of misplaced tiles
- $h_2(n)$  = total Manhattan distance  
(i.e., No. of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- $h_1(S) = ?$  8
- $h_2(S) = ?$   $3+1+2+2+2+3+3+2 = 18$

# Dominance

- If  $h_2(n) \geq h_1(n)$  for all  $n$  (both admissible)
- then  $h_2$  **dominates**  $h_1$
- $h_2$  is better for search
  
- Typical search costs (average number of nodes expanded):
  
- $d=12$  IDS = 3,644,035 nodes  
     $A^*(h_1) = 227$  nodes  
     $A^*(h_2) = 73$  nodes
- $d=24$  IDS = too many nodes  
     $A^*(h_1) = 39,135$  nodes  
     $A^*(h_2) = 1,641$  nodes

# Relaxed Problems

- A problem with fewer restrictions on the actions is called a **relaxed problem**
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then  $h_1(n)$  gives the shortest solution
- If the rules are relaxed so that a tile can move to **any adjacent square**, then  $h_2(n)$  gives the shortest solution

# Beam Search

- Same as BestFS and A\* with one difference
- Instead of keeping the list OPEN unbounded in size, Beam Search **fixes** the size of OPEN
- OPEN only contains the best K evaluated nodes

# Beam Search

- If **new node** considered is not better than any in **OPEN**, and **OPEN** is full, **new node** is not added
- If **new node** is to be inserted in the middle of the priority queue, and **OPEN** is full, **drop** the node at the end of **OPEN** (the one with the **least** priority)

# Local Beam Search

- Keep track of *k states* rather than just one
- Start with *k* randomly generated states
- At each iteration, *all the successors of all *k* states* are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list & repeat.

# Local Search Algorithms

- In many optimization problems, the **path** to the goal is irrelevant; the goal state itself is the solution
- State space = set of “complete” configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use **local search algorithms**
- keep a single “current” state, try to improve it

# Hill Climbing Search

- “Like climbing Everest in thick fog with amnesia”

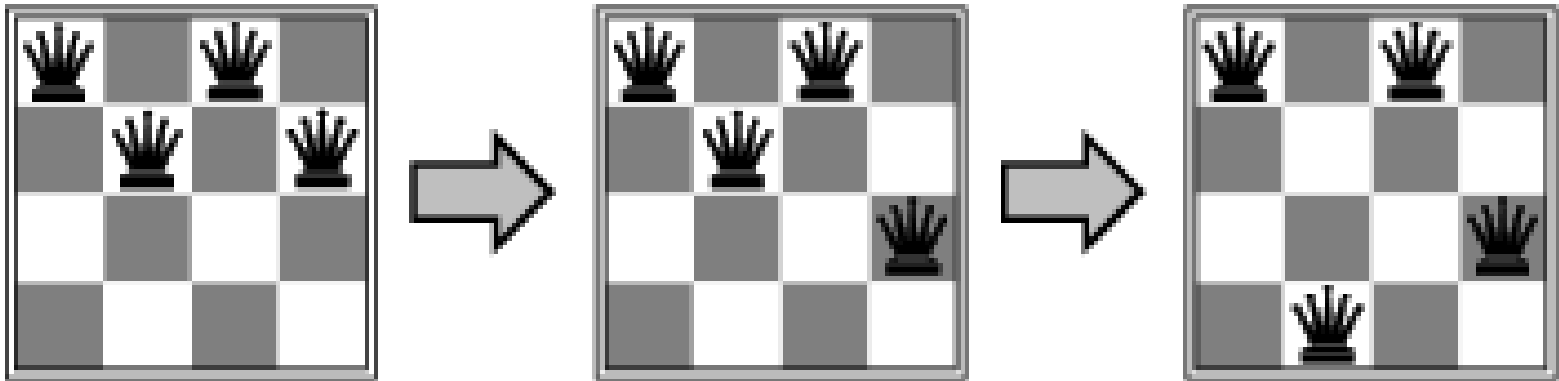
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                  neighbor, a node

  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
```



# Example: $n$ -queens

- Put  $n$  queens on an  $n \times n$  board
- No two queens on the same row, column, or diagonal

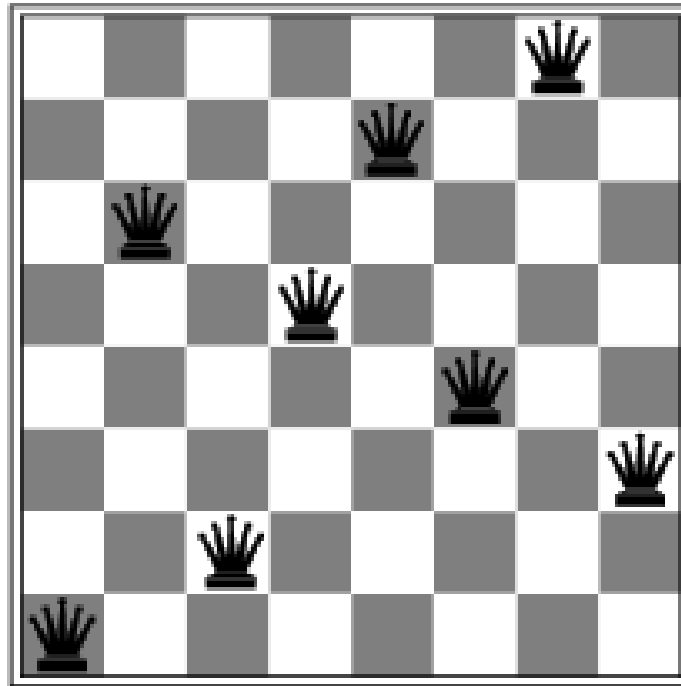


# Example: 8-queens

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

- $h$  = No. of pairs of queens that are attacking each other, either directly or indirectly
- $h = 17$  for the above state

## Hill-climbing Search: 8-queens problem



- A local minimum with  $h = 1$

# Hill Climbing Search

## Simple-Hill-Climber (S)

- Evaluate S; If Goal state return and quit
- Loop until a solution is found or no neighbors left
  - Look at next neighbor NN
  - Evaluate NN
    - If NN is Goal return and quit
    - If NN is better than S,  $S := NN$
    - Reset neighbors

# Hill Climbing Search

## Steepest-Ascent-HC (S)

- Evaluate S; If Goal state return and quit
- $SUCC := S$
- Loop until a solution is found or no neighbors left
  - For all neighbors (NN) of S
    - Evaluate NN
    - If NN is Goal then return NN and quit
    - If NN is better than SUCC then  $SUCC := NN$
  - If SUCC is better than S then
    - $S := SUCC$
    - Reset neighbors

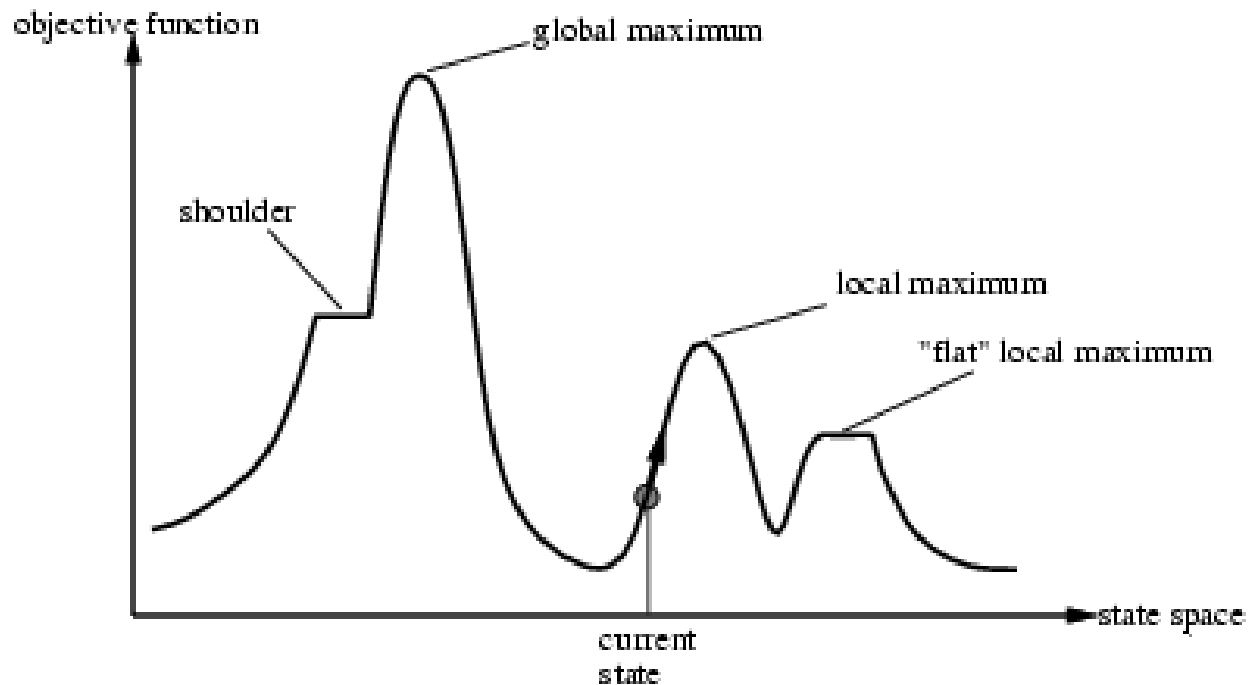
# Hill Climbing Continued

## Stochastic-Hill-Climber (S)

- Evaluate S; If Goal state return and quit
- Loop until a solution is found or no neighbors left
  - Look at some random neighbor RN
  - Evaluate RN
    - If RN is Goal return and quit
    - If RN is better than S
      - $S := RN$
      - Reset neighbors

# Hill Climbing Search

**Problem:** Local maxima or plateau...



# Problems with Hill Climbing

- Hill Climbing will get stuck at local maxima in the space
- Can get stuck on a “plateau”

## Solutions

- Backtrack to earlier node and force it to go in a new direction
- Take a big jump to somewhere else in search space
- Simulated Annealing (Will study this next)
- Genetic Algorithms



# Simulated Annealing Search

- Simulate the annealing process of creating metal alloys
- Start off hot, and cool down slowly which allows the various metals to crystallize into a global uniform structure
- If cooled too fast the metals crystallize in pockets
- If cooled too slowly, a uniform crystallization but wastes time

# Simulated Annealing Search

- Use this idea to try to find global minimum
- Now finding minimum instead of maximum -- but it's the same
- **Wander** from the hill-climbing while system still hot
- **Reduce** to hill climbing as system cools

# Properties: Simulated Annealing

- One can prove:
  - If  $T$  decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching **unity**
- Widely used in VLSI layout, airline scheduling, etc

# Details: Simulated Annealing

- The probability to move to a higher energy state in physics is

$$p = \frac{1}{e^{\Delta E/kT}}$$

where  $k$  is the Boltzmann constant

- Similarly, in SA (when finding the minimum), the probability to move to a state with a higher (worse) heuristic is:

$$p = \frac{1}{e^{\Delta E/T(t)}}$$

where

$\Delta E = (\text{value of current state}) - (\text{value of new state})$

$T(t)$  is the temperature schedule (a function of time  $t$ )

- Temperature monotonically decreases with time,
- Eventually  $T$  reaches 0 when the system becomes simple “hill descending”

# SA Details When Maximizing

- The probability to move to a state with a lower (worse) heuristic function evaluation in SA is

$$p = e^{\Delta E/T}$$

where

$\Delta E = (\text{value of new state}) - (\text{value of current state})$

*(The negation of the  $\Delta E$  used when minimizing)*

$T(t)$  is the temperature schedule (a function of time  $t$ )

- Temperature monotonically decreases with time
- Eventually  $T$  reaches 0 when the system becomes simple “hill climbing”

# Simulated Annealing Algorithm

**Simulated-Annealing** (problem, schedule)      From Russell and Norvig

Current := Initial-State(Problem)

for t := 1 to  $\infty$  do

    T := schedule(t)

    If T = 0 then return Current

    Next := a randomly selected successor of Current

$\Delta E$  := Value(Next) - Value(Current)

    If  $\Delta E > 0$  then

        “Always go to a better solution”

        Current := Next

    Else

        “Leave a better solution for a worse one with prob.  $e^{-\Delta E/T}$ ”

        Current := Next only with probability  $e^{-\Delta E/T}$

# SA: Meta Heuristics

- If the solution is better:
  - Always move to it
- If the solution is worse but the slope up is shallow:
  - Try it out
- If the solution is worse but the slope is steep:
  - Don't try it out as readily (with an exponentially decreasing probability)
- As time goes on, don't try worse solutions as frequently
  - Again with an exponentially decreasing probability

# SA Effects

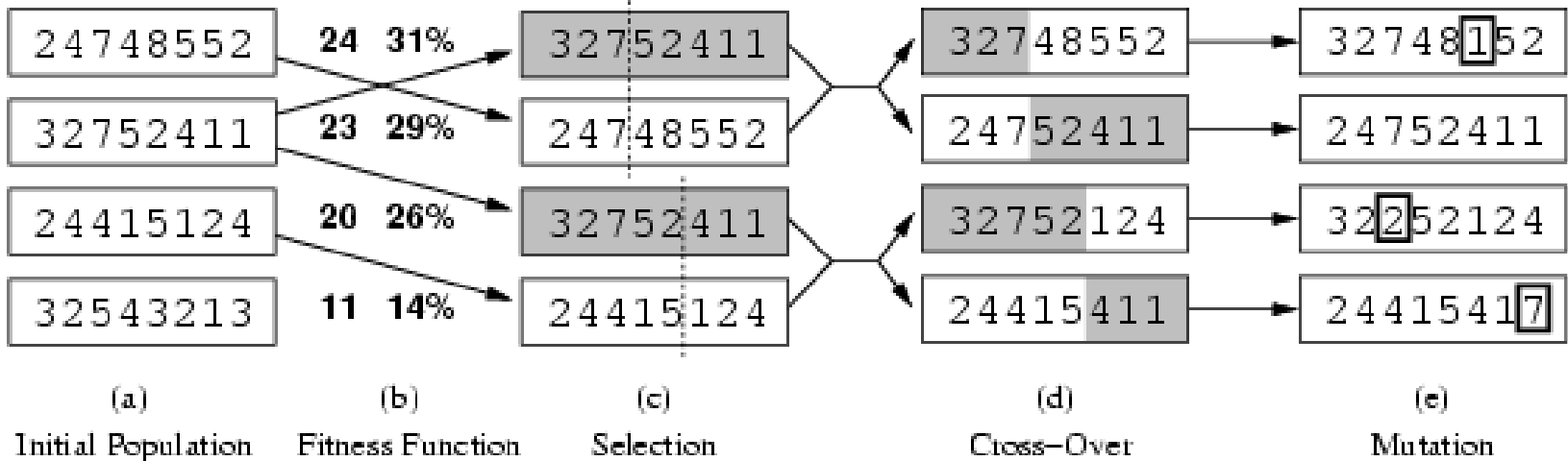
- **At the beginning of the process** (when  $T(t)$  is large)
  - The probability of moving to poorer states, or moving along a plateau is large.
  - So the space can be well searched
  - Local minimums can be passed over
  - Ignore steep ascents
    - This implies that you are in a deep valley, which is assumed to be good
- **As time increases**
  - The search gets trapped in one valley and gets stuck as  $T(t)$  becomes small
  - The probability of getting out of the Valley is too small.
- **At this time**
  - SA becomes “hill descending”
  - Descends to the bottom of that valley - hopefully the global minimum



# Genetic Algorithms

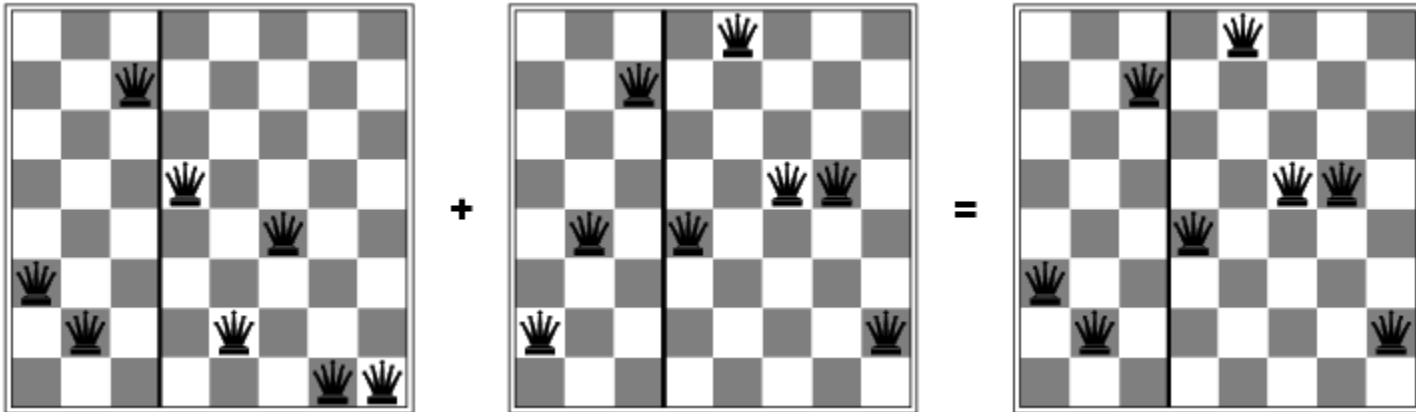
- A successor state is generated by combining two parent states
- Start with  $k$  randomly generated states (**population**)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (**fitness function**). Higher values for better states.
- Produce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

# Genetic Algorithms



- **Fitness function:** Number of non-attacking pairs of queens  
(min = 0, max =  $8 \times 7/2 = 28$ )
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$  etc

# Genetic Algorithms

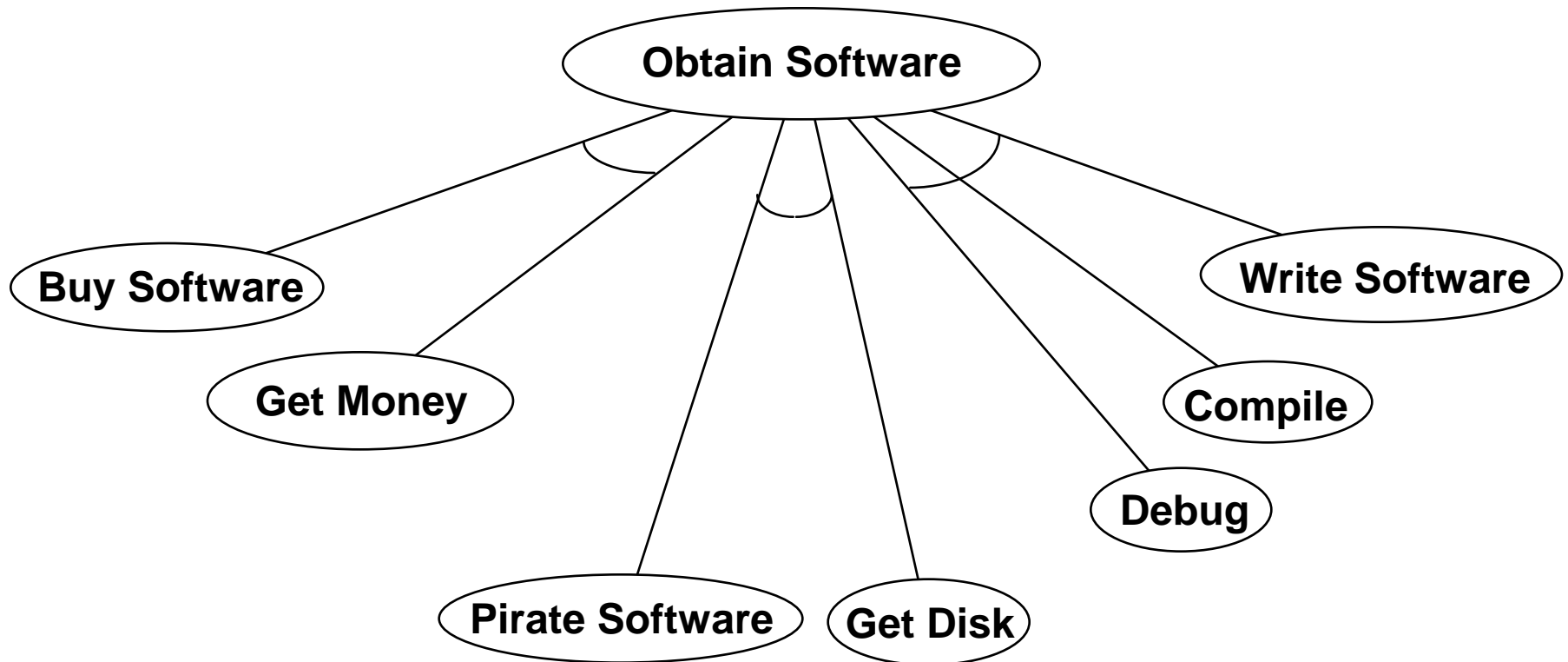


# OR Graphs vs. AND-OR Graphs

- In the previous search techniques, Solution can be found down any path independent of any other path
- This is called an **OR** graph
- However, there may be sub-goals that must **all** be solved for a solution to be found
  - Each sub-goal is its own sub-tree
  - All sub-trees must have its own end state found if the path is to be considered satisfied
- This is called an **AND-OR** graph

# Example of an AND-OR Graph

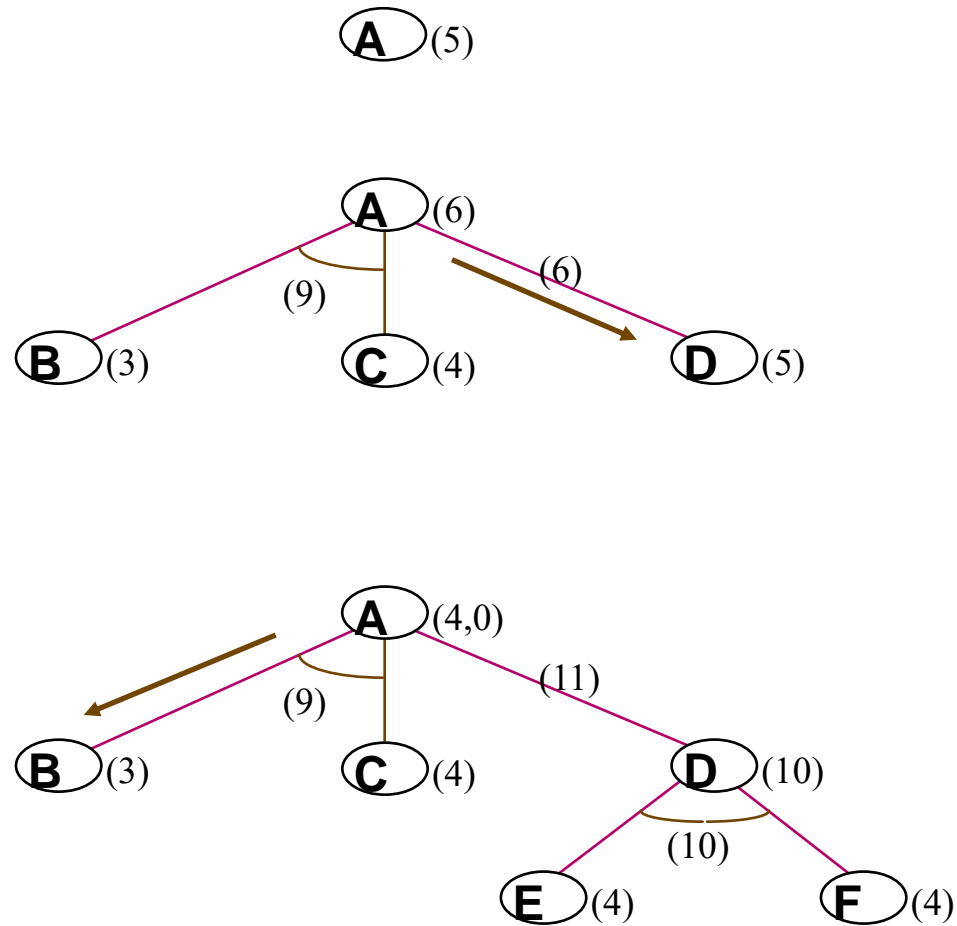
- Getting software to accomplish a task



# Problem Reduction Algorithm

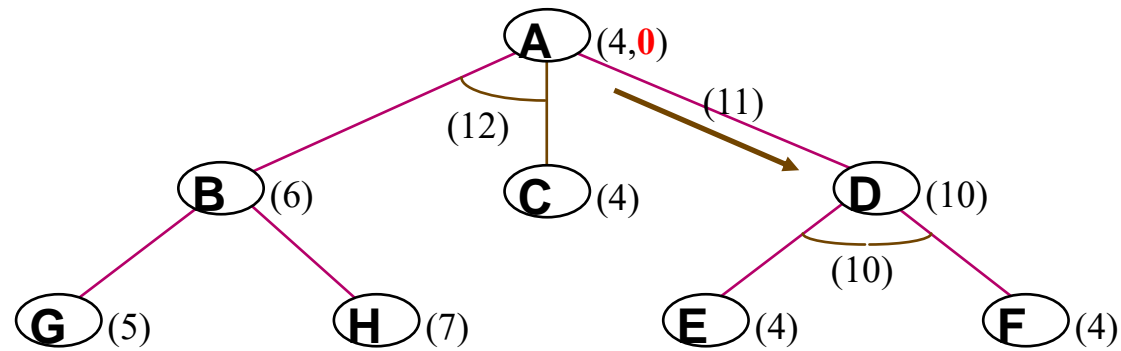
- **Initialize** the graph to the starting node
- Until the starting node is labeled SOLVED or its cost  $>$  FUTILITY do:
  - Start at initial node and traverse best path
    - Accumulate set of nodes on path not expanded or labeled SOLVED
  - Pick an unexpanded node and expand
    - If no successors, node cost = FUTILITY
    - Add successors to graph after computing the heuristic  $f$  for each
    - If  $f = 0$  for any node mark node as SOLVED
  - Propagate change back through path
    - If child is an OR child and is SOLVED mark parent as SOLVED
    - If AND children are all solved, mark parent as SOLVED
    - Change the estimate of  $f$  as determined by children
    - As we back up the tree, change current best path associated with each node (on the original best path) if updated  $f$  values warrant it

# Example of Problem Reduction (AO\*)



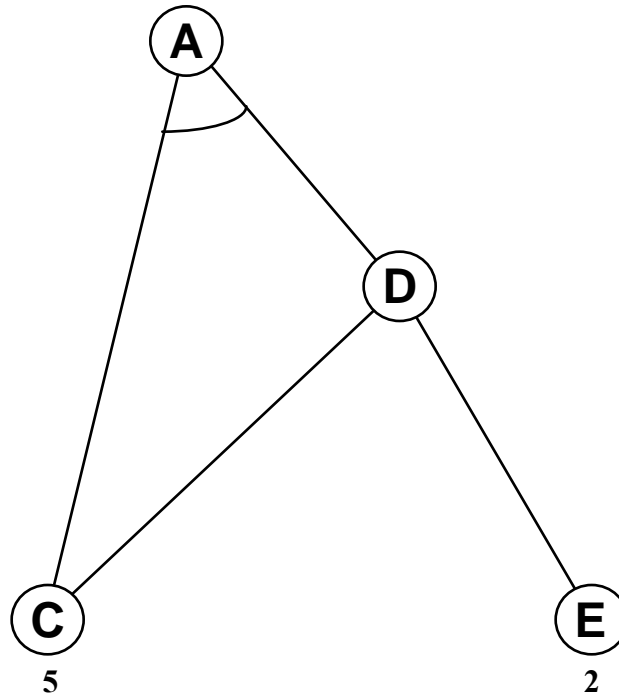
When you calculate costs, remember to use the cost PLUS the depth

# Example of Problem Reduction (AO\*)





# Interacting Sub-goals



# Branch and Bound

- If we know that current path (branch) is already **worse** than some other known path:
  - **Stop** Expanding It (Bound).
- Have already encountered Branch and Bound:
  - A\* stops expanding a branch if its heuristic value  $h$  becomes larger than some other branch

# Constraint Satisfaction Problems and Branch and Bound

- Problems where there are **natural constraints** on the system (fixed resources, impossibility conditions, etc.)
- Constraints: Handled by **Branch and Bound** technique
  - Branch out in your normal search pattern
  - Stop expanding a branch if it fails a constraint (backtracking may occur when that happens)
- Trivial example: Missionaries and Cannibals
  - Do not continue to search along a branch if the Cannibals have just eaten some (or all) of the Missionaries