COMP3308/COMP3608, Lecture 3a
ARTIFICIAL INTELLIGENCE

A* Algorithm

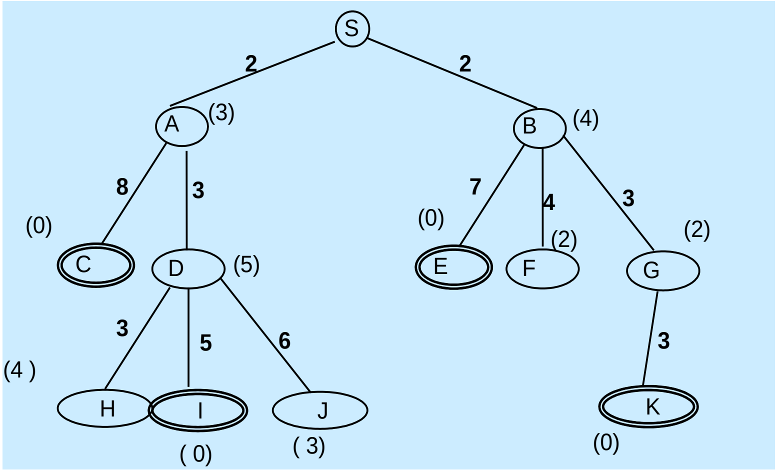
Reference: Russell and Norvig, ch. 3

Outline

- **A* search algorithm**
- **How to invent admissible heuristics**

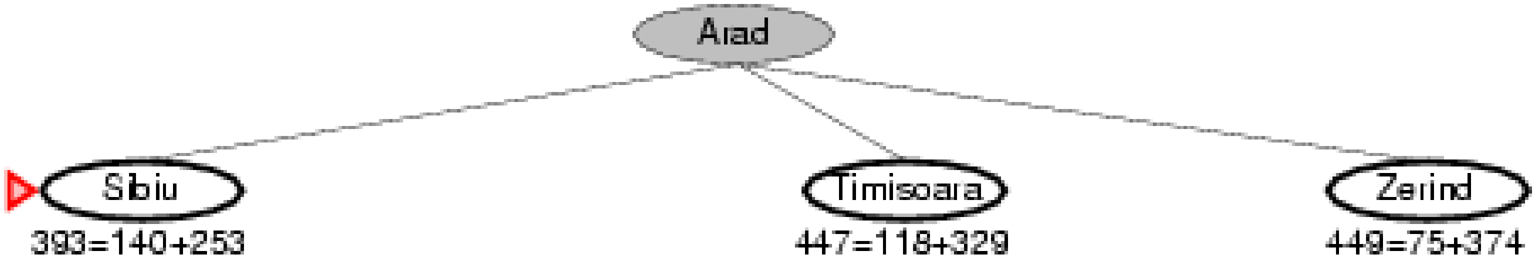
A* Search

- **UCS minimizes the cost so far *g(n)***
- **GS minimizes the estimated cost to the goal *h(n)***
- **A* combines UCS and GS**
- **Evaluation function: *f(n)=g(n)+h(n)***
  - ***g(n)* = cost so far to reach *n***
  - ***h(n)* = estimated cost from *n* to the goal**
  - ***f(n)* = estimated total cost of path through *n* to the goal**

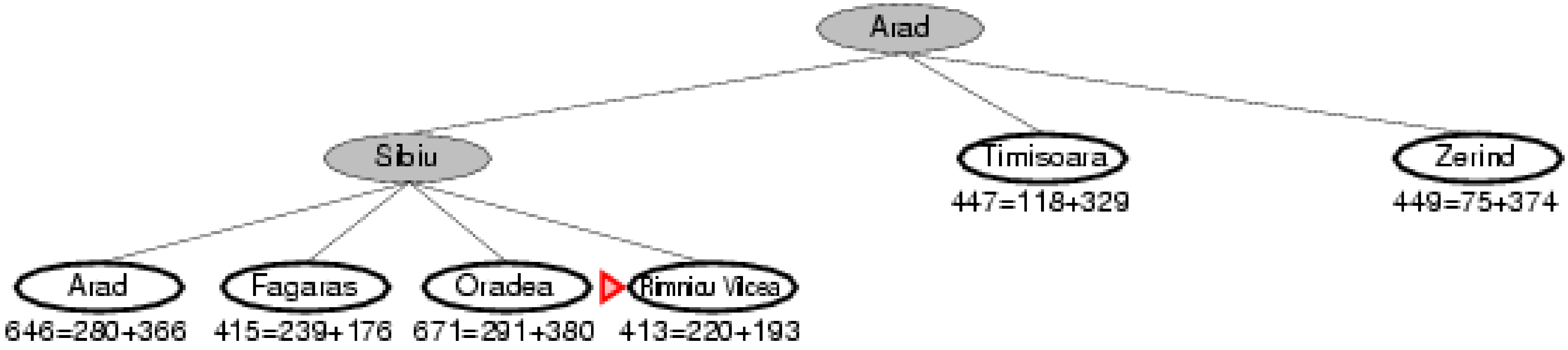


A* Search for Romania Example

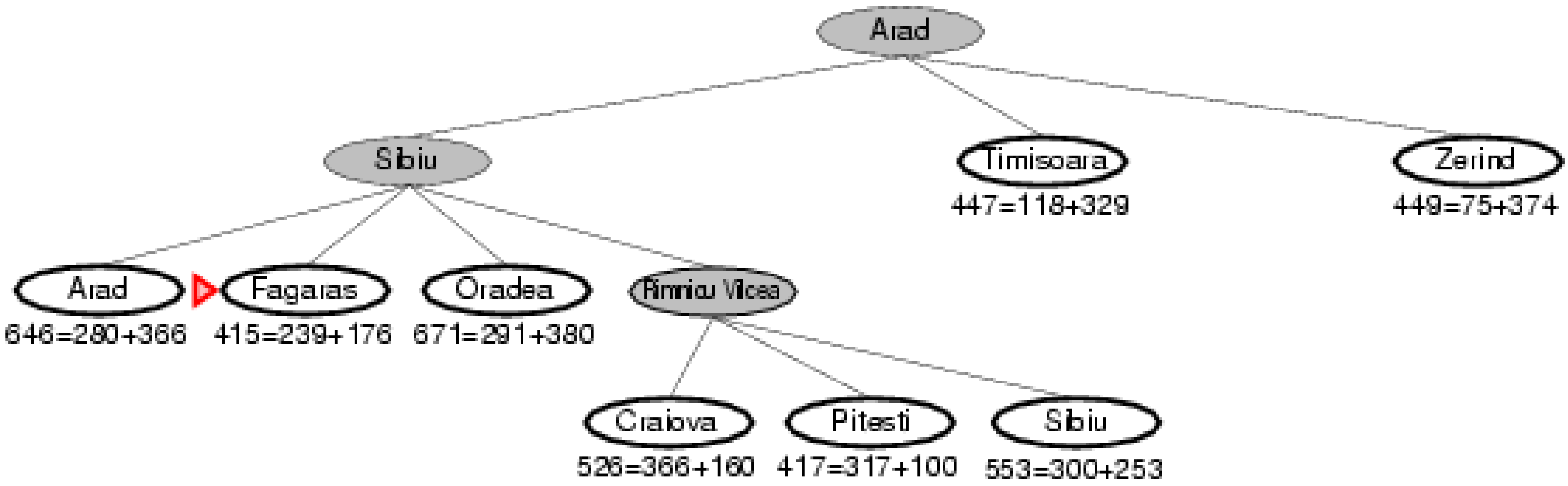
A* Search for Romania Example



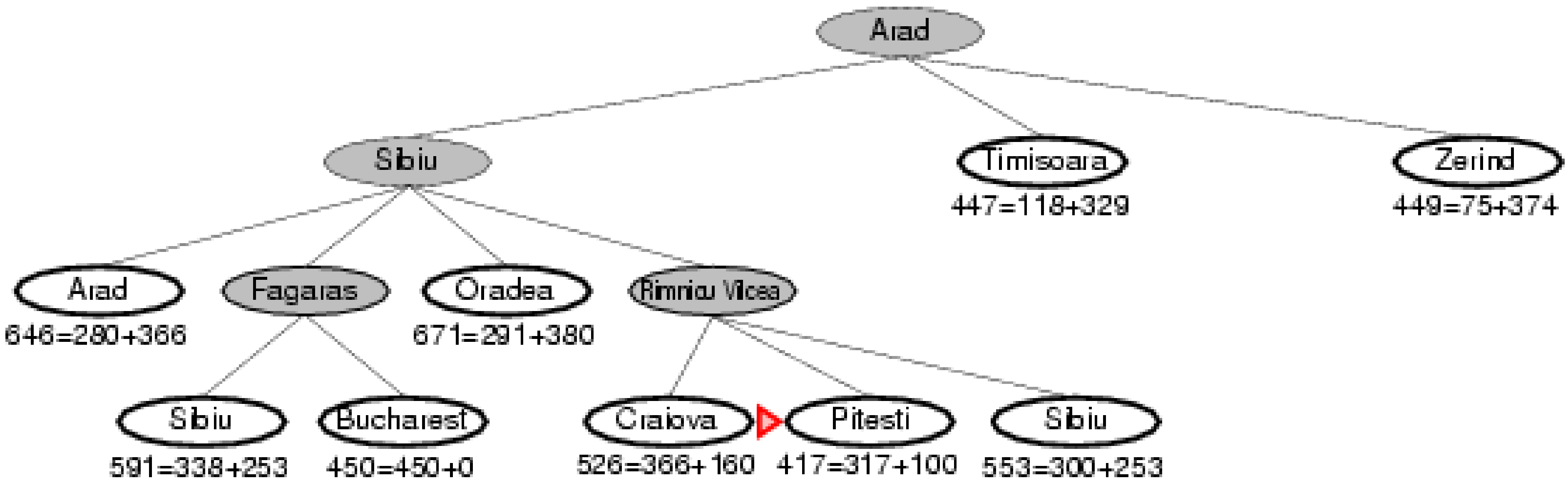
A* Search for Romania Example



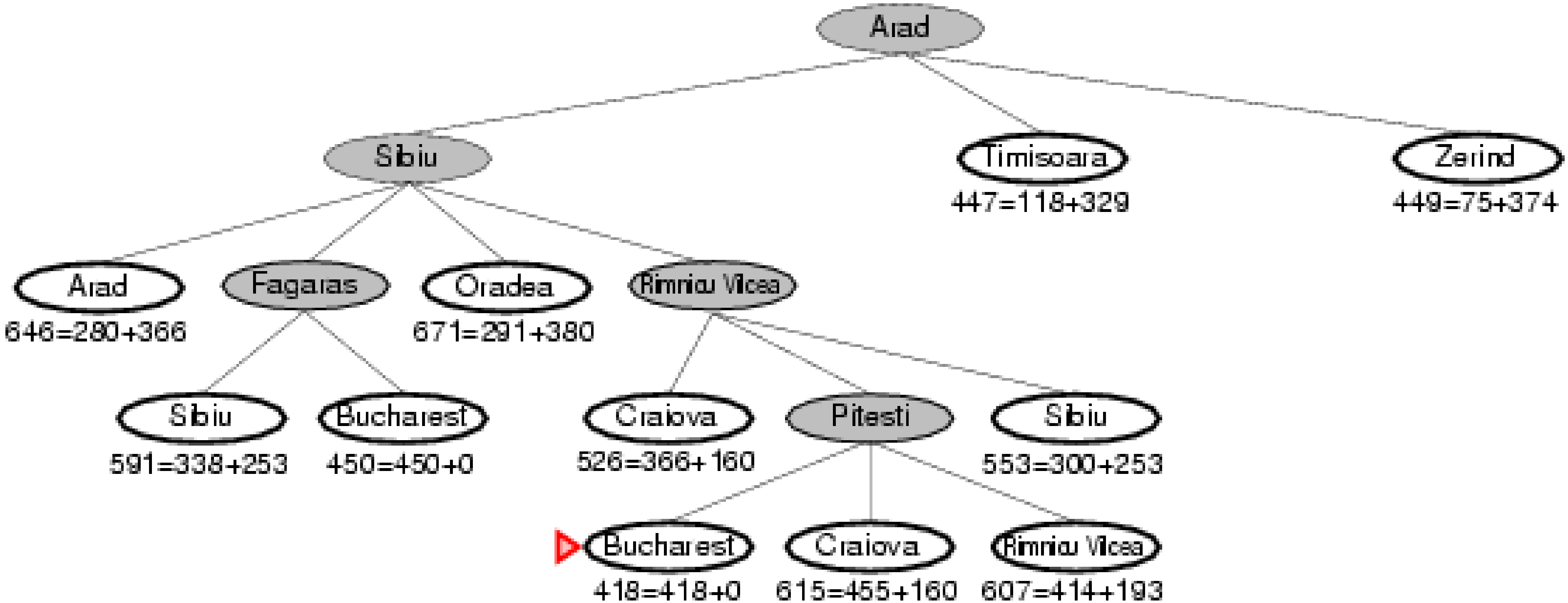
A* Search for Romania Example



A* Search for Romania Example



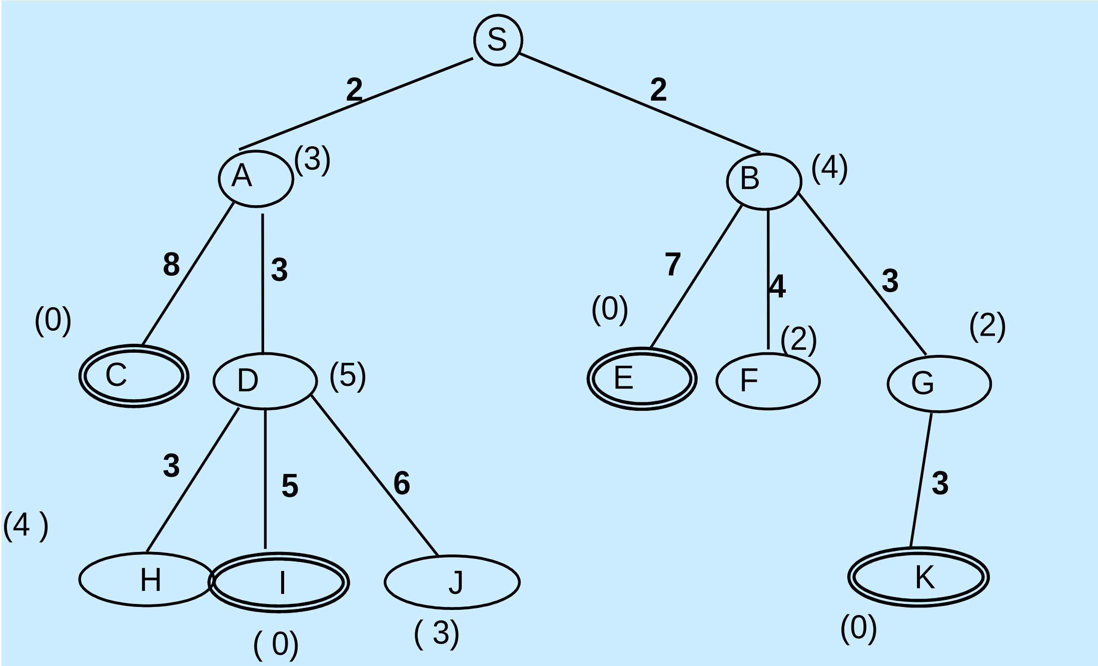
A* Search for Romania Example



Bucharest is selected for expansion and it is a goal node => stop
Solution path: Arad-Sibiu-Riminicu Vilcea-Pitesti-Bucharest, cost=418

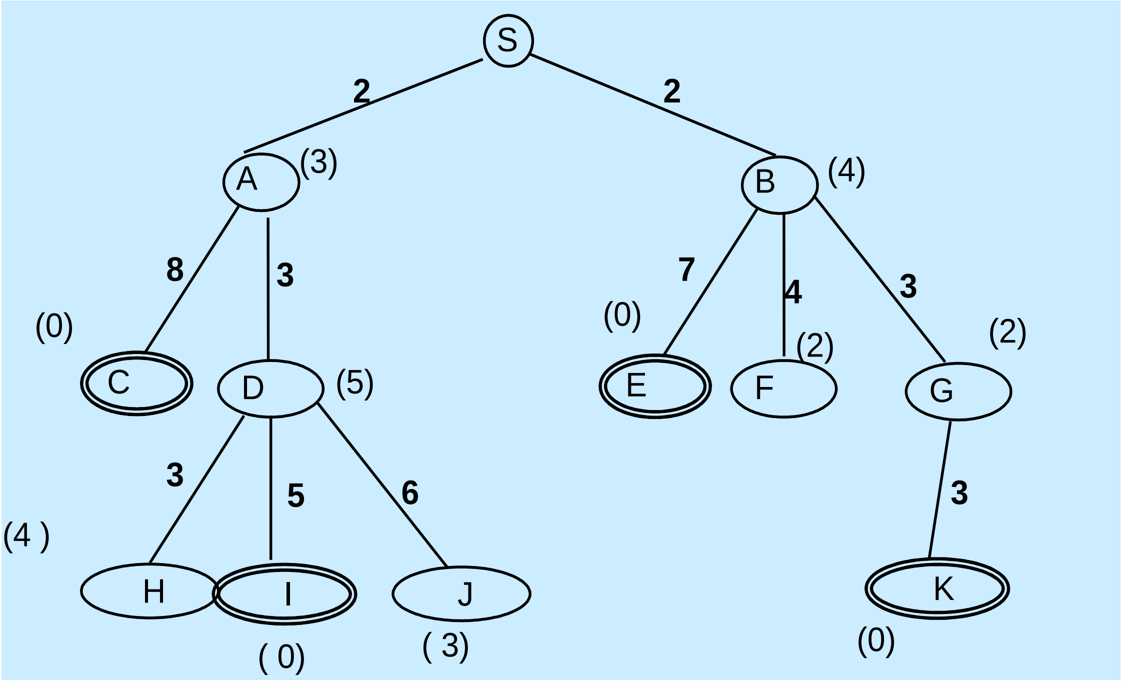
A* Search – Another Example

- **Given:**
  - **Goal nodes: C, I, E and K**
  - **Step path cost: along the links**
  - ***h* value of each node: in brackets ()**
  - **Same priority nodes -> expand the last added first**
- **Run A***
  - **list of expanded nodes =?**
  - **solution path =?**
  - **cost of the solution:=?**



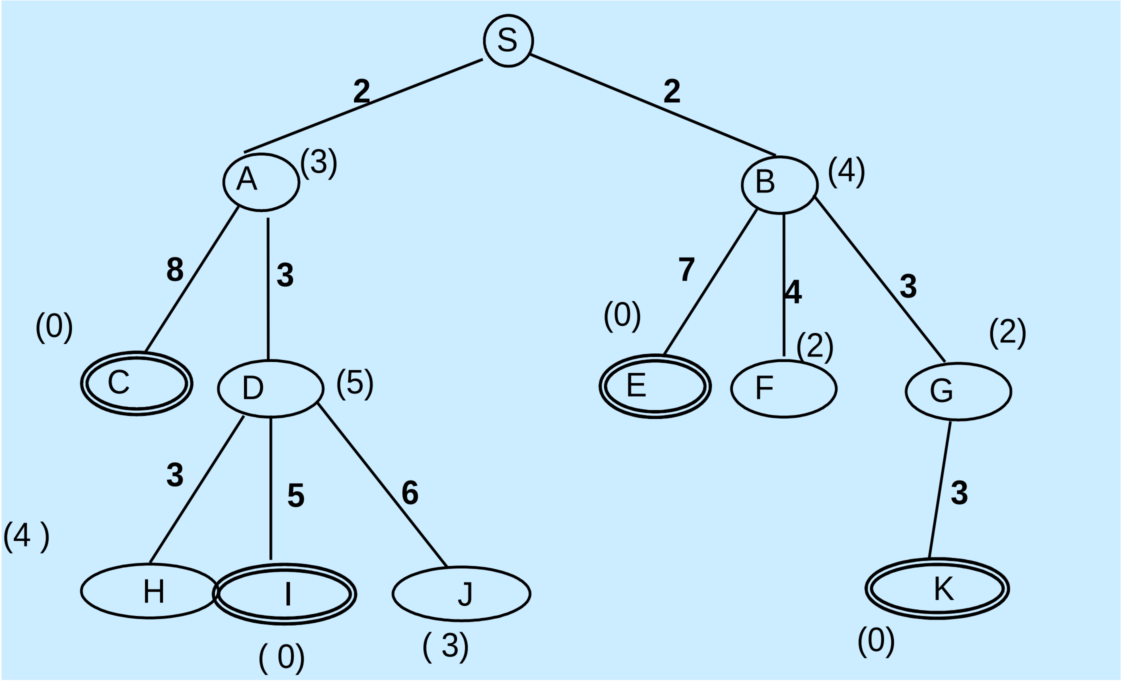
Solution

- **Fringe: S**
- **Expanded: nill**



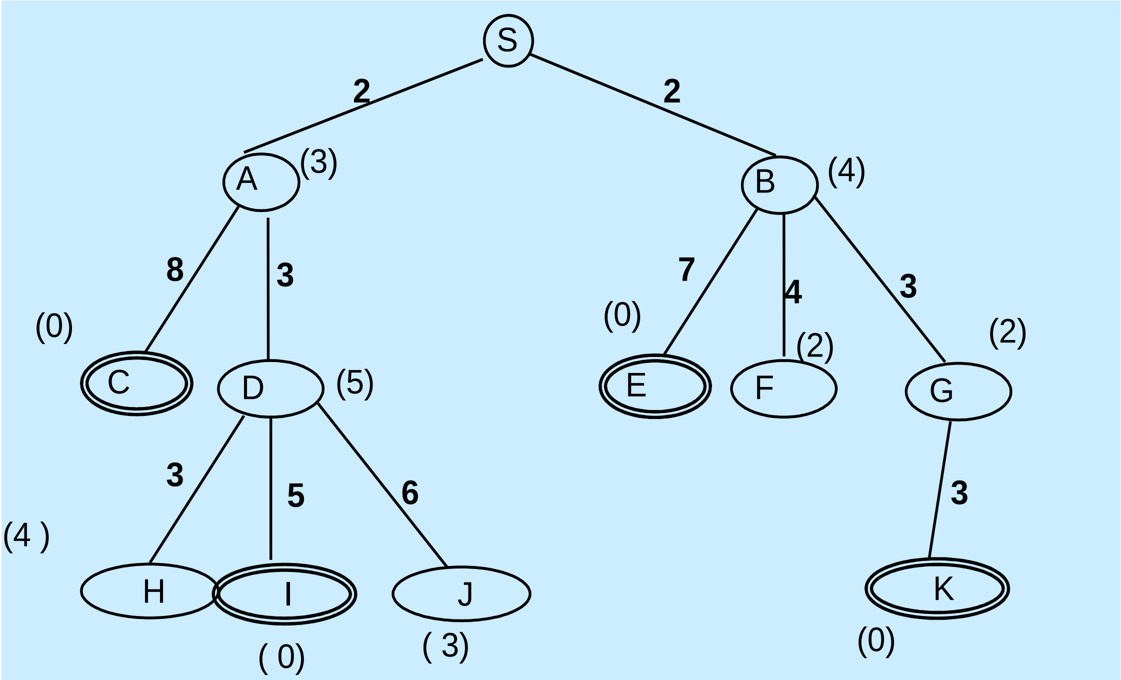
Solution

- **Fringe: (A, 5), (B, 6) //keep the fringe in sorted order**
- **Expanded: S**



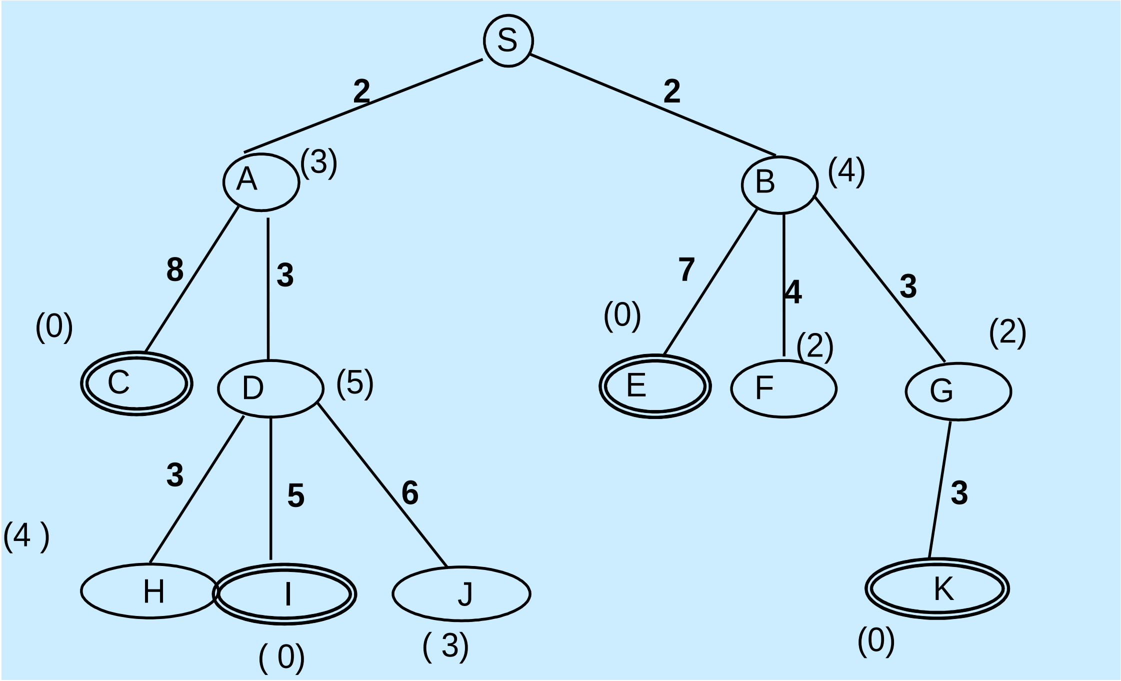
Solution

- **Fringe: (B, 6), (C, 10), (D, 10) //the added children are in blue**
- **Expanded: S, (A, 5)**



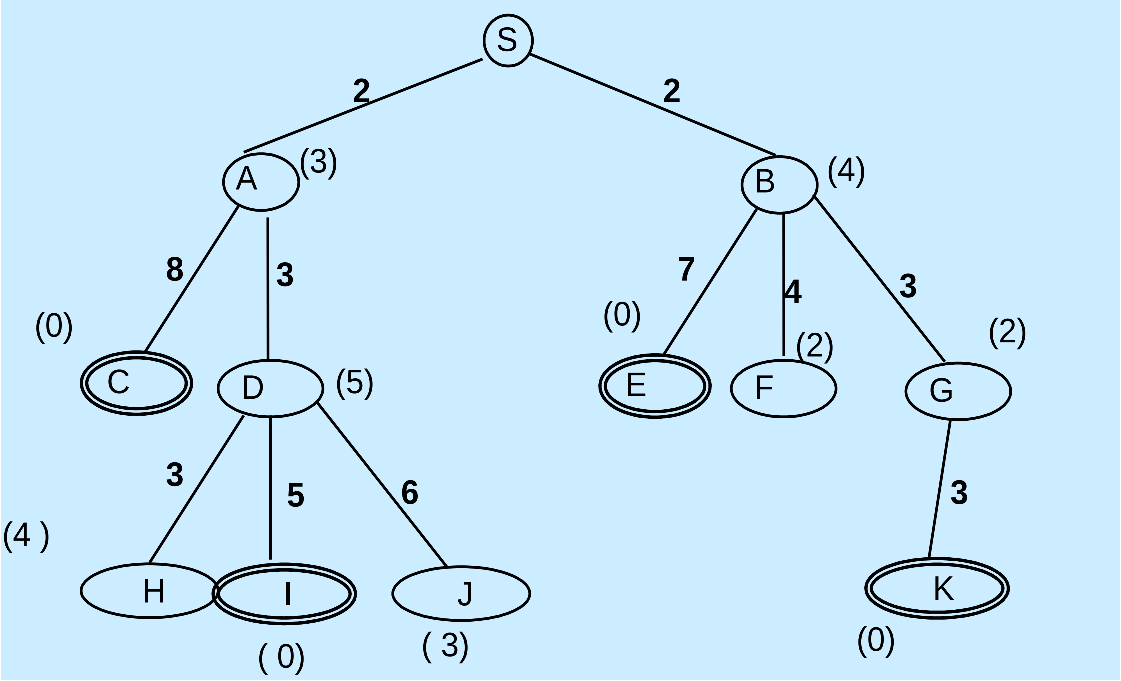
Solution

- **Fringe: (G, 7), (F, 8), (E, 9), (C, 10), (D, 10)**
- **Expanded: S, (A, 5), (B, 6)**



Solution

- **Fringe: (K, 8), (F, 8), (E, 9), (C, 10), (D, 10)**
- **Expanded: S, (A, 5), (B, 6), (G, 7)**

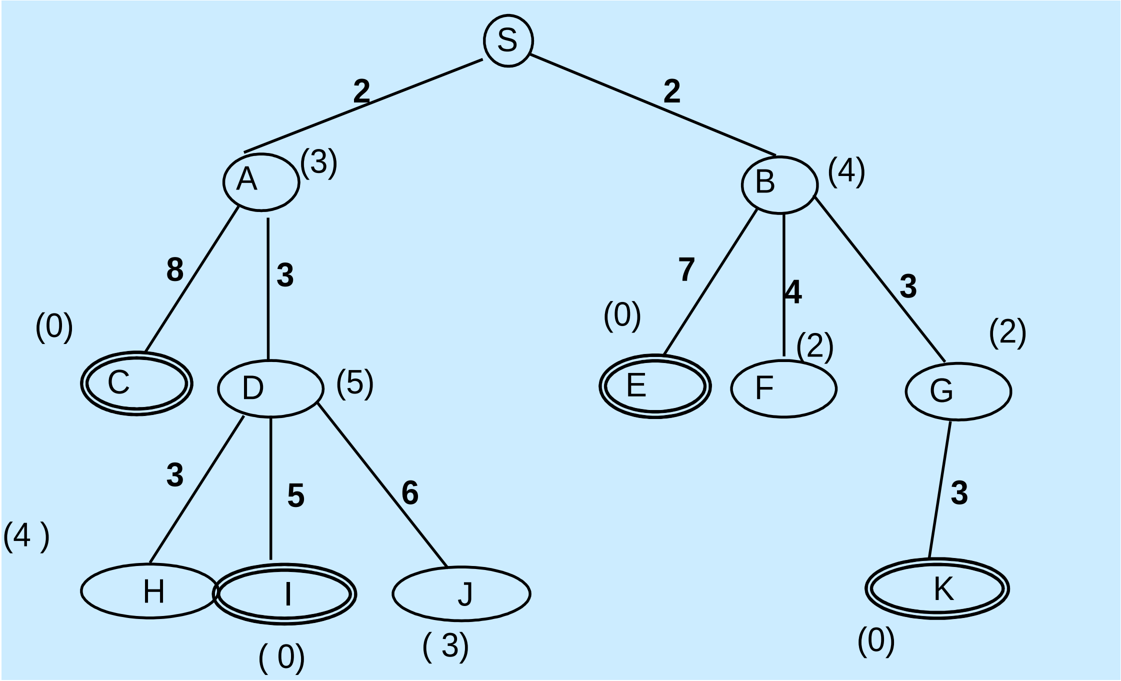


Solution

- **K is selected; Goal node? Yes => stop**

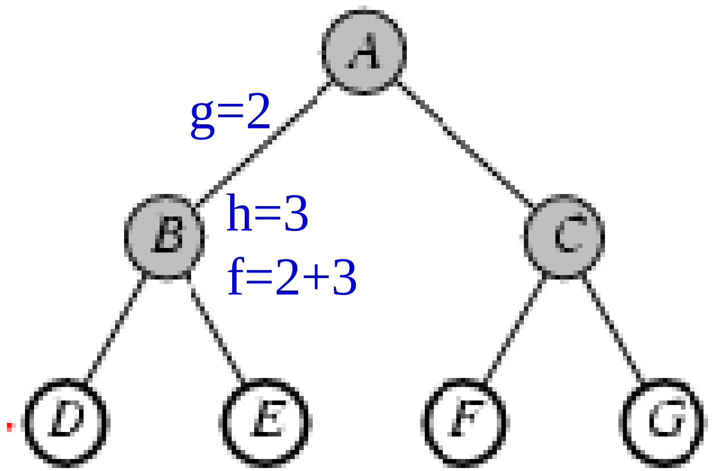
- **Expanded: S, A, B, G, K**
- **Solution path: SBGK, cost=8**

- **Is this the optimal solution=?**



A* and UCS

- **UCS is a special case of A* when *h(n)* =?**
- **In other words, when will A* behave as UCS?**

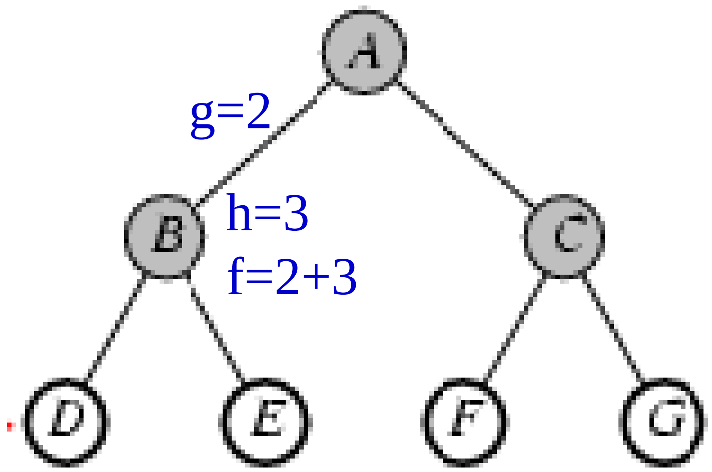


Hint:

- **UCS uses which cost?**
- **A* uses which cost?**
- **Relation between the 2 costs =?**

A* and UCS (Answer)

- **UCS is a special case of A* when *h(n)* =?**
- **In other words, when will A* behave as UCS?**

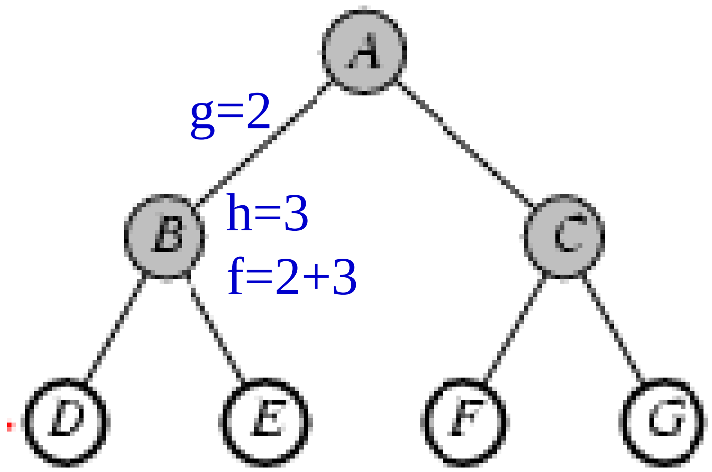


- **UCS uses which cost?**
- **A* uses which cost?**

- **UCS: $g(n)$**
- **A*: $f(n)=g(n)+h(n)$**
- **if $h(n)=0$ => $f(n)=g(n)$, i.e. A* becomes UCS**

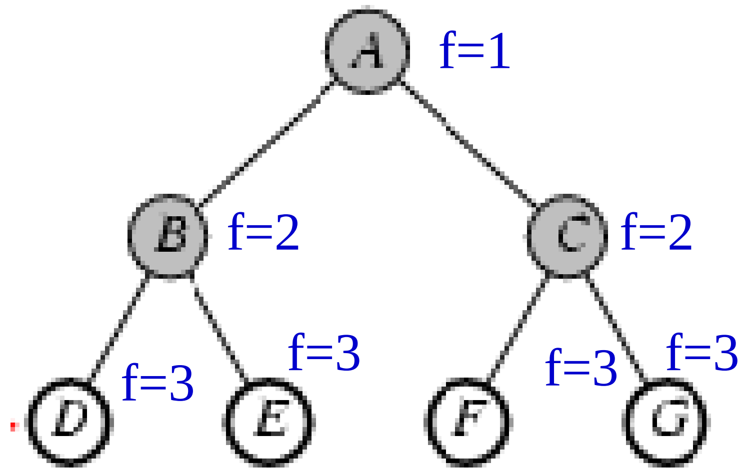
A* and BFS

- **BFS is a special case of A* when *f(n)* =?**
- **When will A* behave as BFS?**



A* and BFS (Answer)

- **BFS is a special case of A* when *f(n)* =?**

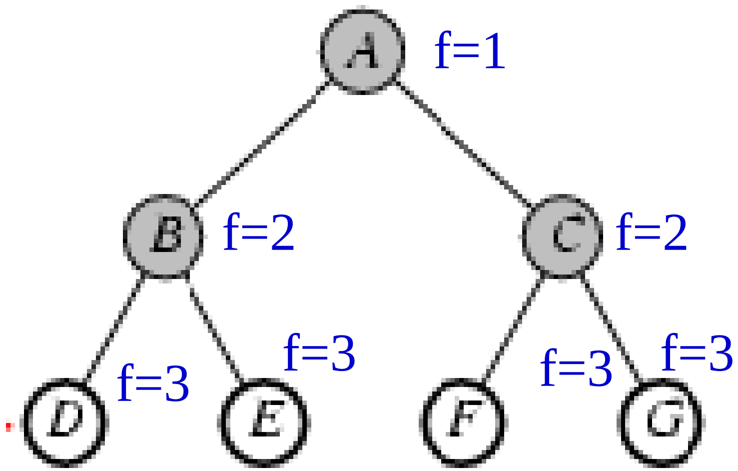


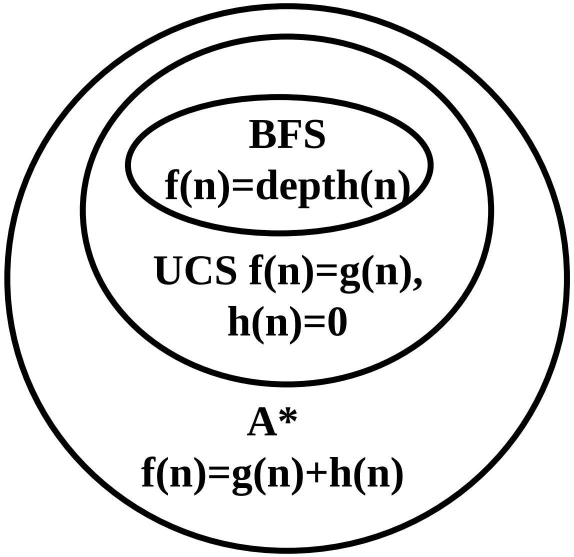
when *f(n)=depth(n)*

- **And also when this assumption for resolving ties is true: among nodes with the same priority, the left most is expanded first**

BFS, UCS and A*

- **BFS is a special case of A* when *f(n)=depth(n)***
- **BFS is also a special case of UCS when *g(n)=depth(n)***
- **UCS is a special case of A* when *h(n)=0***





Admissible Heuristic

- **A heuristic *h(n)* is admissible if for every node *n*:**
  - ***$h(n) \leq h^*(n)$* where *$h^*(n)$* is the <u>true</u> cost to reach a goal from *n***
  - **i.e. the estimate to reach a goal is smaller than (or equal to) the true cost to reach a goal**

- **Admissible heuristics are *optimistic* – they think that the cost of solving the problem is less than it actually is!**
  - **e.g. the straight line distance heuristic $h_{SLD}(n)$ never overestimates the actual road distance (cost from *n* to goal) => it is admissible**

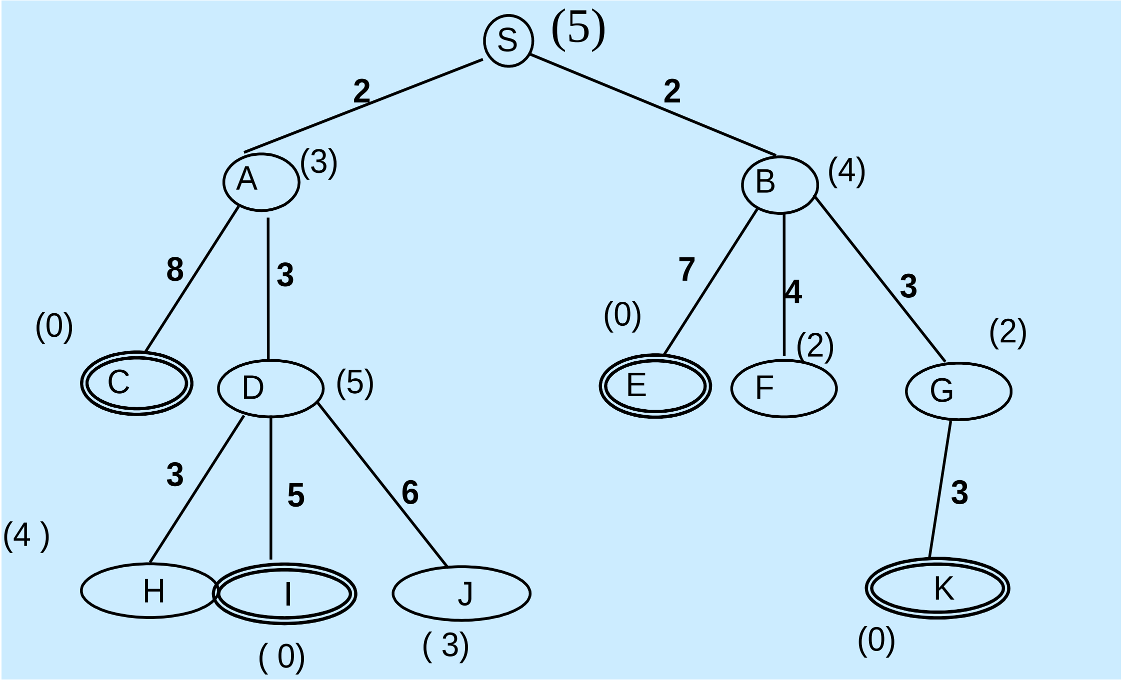
- **Theorem: If *h* is an *admissible heuristic*, then A* is complete and optimal**

Is *h* Admissible for Our Example?

- **No need to check goal nodes (*h=0* for them) and nodes that are not on a goal path**

- ***h*(S)=5<=8 (shortest path from S to a goal, i.e. to goal K)**
- ***h*(B)=4<=6**
- ***h*(G)=2<=3**
- ***h*(A)=3<=8**
- ***h*(D)=5<=5**

- **=> *h* is admissible**



Optimality of A* - Proof

Optimal solution = the shortest (lowest cost) path to a goal node

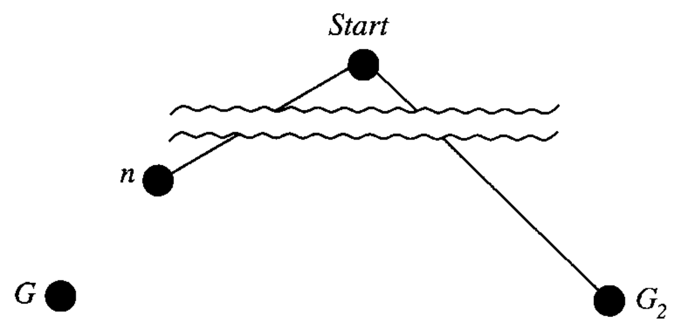
Idea: Suppose that some sub-optimal goal G_2 has been generated and it is in the fringe. We will show that G_2 can not be selected from the fringe.

<u>Given:</u>

***G* - the optimal goal**

G_2 – a sub-optimal goal

***h* is admissible**



<u>To prove:</u> G_2 can not be selected from the fringe for expansion

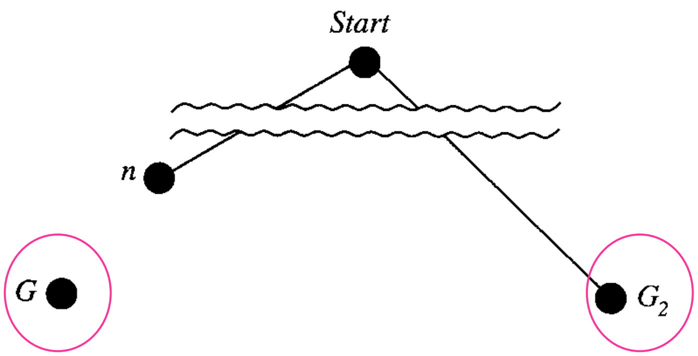
<u>Proof:</u>

Let *n* be an unexpanded node in the fringe such that *n* is <u>on</u> the optimal (shortest) path to *G* (there must be such a node). We will show that $f(n)<f(G2)$, i.e. *n* will be expanded, not G2

Optimality of A* - Proof (2)

Compare *f(G2)* and *f(G)*

1) ***f(G2)=g(G2)+h(G2) (by definition) = g(G2)* as *h(G2)=0, G2* is a goal**
2) ***f(G)=g(G)+h(G) (by definition) = g(G)* as *h(G)=0, G* is a goal**
3) ***g(G2)>g(G)* as *G2* is suboptimal**
4) ***=> f(G2)>f(G)* by substituting 1) and 2) into 3)**



Optimality of A* - Proof (3)

Compare $f(n)$ and $f(G)$

5) $f(n)=g(n)+h(n)$ (by definition)

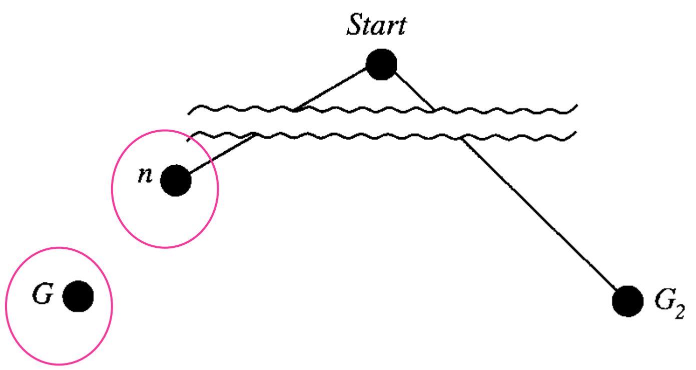
6) $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost from n to G (as h is admissible)

7) => $f(n)<= \underbrace{g(n) + h^*(n)}$ (5 & 6)

8) = g(G) path cost from S to G via n

9) $g(G) = f(G)$ as $f(G)=g(G)+h(G)=g(G)+0$ as h(G)=0, G is a goal

10) => $f(n)<=f(G)$ (7,8,9)



Thus $f(G) < f(G2)$ (4)

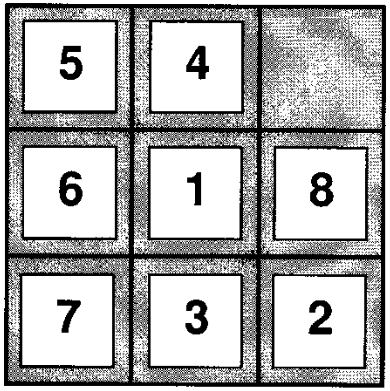
$f(n)<=f(G)$ (10)

11) $f(n)<=f(G)<f(G2)$ (10, 4)

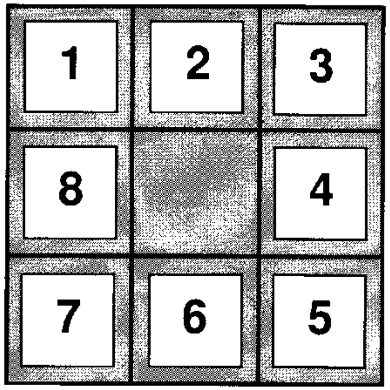
12) $f(n)<f(G2)$ => *n* will be expanded not *G2*; A* will not select G2 for expansion

Admissible Heuristics for 8-puzzle – h1

- $h_1(n)$ = number of misplaced tiles
- $h_1(Start)$ = ?
  - 7 (7 of 8 tiles are out of position)
- Why is h_1 admissible?
  - recall: admissible heuristics are optimistic – they never overestimate the number of steps to the goal
  - h_1: any tile that is out of place must be moved once
  - true cost: higher; any tile that is out of place must be moved ***at least*** once



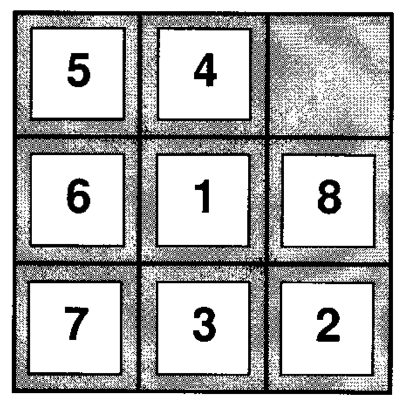
Start State



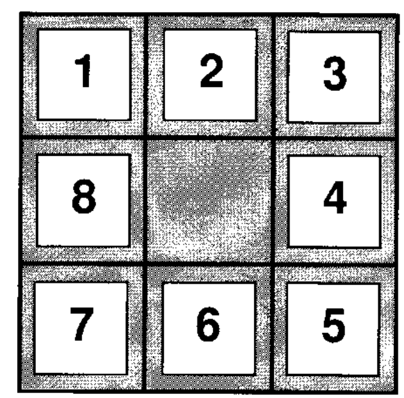
Goal State

Admissible Heuristics for 8-puzzle – h2

- ***$h_2(n)$* = the sum of the distances of the tiles from their goal positions (Manhattan distance)**
  - **note: tiles can move only horizontally and vertically**
- ***$h_2(Start)$* = ?**
  - **18 (2+3+3+2+4+2+0+2)**
- **Why is *h_2* admissible?**
  - ***h2:* at each step move a tile to an adjacent position so that it is 1 step closer to its goal position and you will reach the solution in h2 steps, e.g. move tile 1 up, then left**
  - **True cost: higher as moving a tile to an adjacent position is not always possible; depends on the position of the blank tile**



Start State



Goal State

Dominance

- **Definition of a *dominant heuristic*:**
  - **Given 2 admissible heuristics h_1 and h_2 ,**
  - **h_2 *dominates* h_1 if for all nodes n $h_2(n) \geq h_1(n)$**
- **Theorem: A* using h_2 will expand fewer nodes than A* using h_1 (i.e. h_2 is better for search)**
  - **$\forall$ n with f(n) < f* will be expanded (f*=cost of optimal solution path)**
  - **=> $\forall$ n with h(n) < f*- g(n) will be expanded**
  - **but $h_2(n) \geq h_1(n)$**
  - **=> $\forall$ n expanded by A*using h_2 will also be expanded by h_1 and h_1 may also expand other nodes**

- **Typical search costs for 8-puzzle with d=14:**

  IDS = 3 473 941 nodes, A*(h1) = 539 nodes, A*(h2) = 113 nodes
- **Dominant heuristics give a better estimate of the true cost to a goal G**

Question

- **Suppose that *h1* and *h2* are two admissible heuristics for a given problem. We define two other heuristics:**
  - ***h3=min(h1, h2)***
  - ***h4= max(h1, h2)***
- **Q1. Is *h3* admissible?**
- **Q2. Is *h4* admissible?**
- **Q3. Which one is a better heuristic - *h3* or *h4*?**

Answer

- **Suppose that *hl* and *h2* are two admissible heuristics for a given problem. We define two other heuristics:**
  - ***h3=min(hl, h2)***
  - ***h4= max(hl, h2)***
- **Q1. Is *h3* admissible?**
- **Q2. Is *h4* admissible?**
- **Q2. Which one is a better heuristic - *h3* or *h4*?**

Answer:

- **Q1 and Q2: Both *h3* and *h4* are admissible as their values are never greater than an admissible value *h1* or *h2***
- **Q3: *h4* is a better heuristic since it is closer to the real cost, i.e. *h4* is a dominant heuristic since $h4(n) \geq h3(n)$**

How to Invent Admissible Heuristics?

- **By formulating a *relaxed* version of the problem and finding the *exact* solution. This solution is an admissible heuristic.**
- **Relaxed problem – a problem with fewer restrictions on the actions**
- **8-puzzle relaxed formulation 1:**
  - **a tile can move *anywhere***
  - **How many steps do we need to reach the goal state from the initial state? (=solution)**
  - **solution = the number of misplaced tiles = $h1(n)$**
- **8-puzzle relaxed formulation 2:**
  - **a tile can move to *any adjacent square***
  - **solution = Manhattan distance = $h_2(n)$**

Admissible Heuristics from Relaxed Problems

- **<u>Theorem:</u> The optimal solution to a relaxed problem is an admissible heuristic for the original problem**
- **Intuitively, this is true because:**

  The optimal solution to the original problem is also a solution to the relaxed version (by definition) => it must be at least as expensive as the optimal solution to the relaxed version => the solution to the relaxed version is less or equally expensive than the solution to the original problem => it is an admissible heuristic for the original problem

Constructing Relaxed Problems Automatically

- **Relaxed problems can be constructed automatically if the problem definition is written in a formal language**
  - **Problem:**

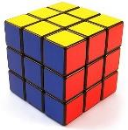
    A tile can move from square A to square B if

    A is adjacent to B and B is blank
  - **3 relaxed problems generated by removing 1 or both conditions:**

    1) A tile can move from square A to square B if A is adjacent to B

    2) A tile can move from square A to square B if B is blank

    ***3) A tile can move from square A to square B* (always, no conditions)**
- **ABSOLVER (1993) is a program that can generate heuristics automatically using the "relaxed problem" method and other methods**
  - **Generated a new heuristic for the 8-puzzle that was better than any existing heuristic**
  - **Found the first useful heuristic for the Rubik's cube puzzle**



No Single Clearly Best Heuristic?

- **Often we can't find a single heuristic that is clearly the best (i.e. dominant)**
- **We have a set of heuristics *h1*, *h2*, …, *hm* but none of them dominates any of the others**
- **Which should we choose?**

- **Solution: define a composite heuristic:**

  h(n)=max{h1(n), h2(n), …, hm(n)}

  At a given node, it uses whichever heuristic is most accurate (dominant)
- **Is *h(n)* admissible?**

  Yes, because the individual heuristics are admissible

Learning Heuristics from Experience

- **Example: 8-puzzle**
- **Experience = many 8-puzzle solutions (paths from A to B)**
- **Each previous solution provides a set of examples to learn *h***
- **Each example is a pair (state, associated *h*)**
  - ***h* is known for each state, i.e. we have a *labelled* dataset**
- **The state is suitably represented as a set of useful features, e.g.**
  - ***f1* = number of misplaced tiles**
  - ***f2* = number of adjacent tiles that should not be adjacent**
  - ***h* is a function of the features but we don't know how exactly it depends on them, we will learn this relationship from the data**
- **We can generate e.g. 100 random 8-puzzle configurations and record the values of *f1*, *f2* and *h* to form a *training set* of examples. Using this training set, we build a classifier.**

training data

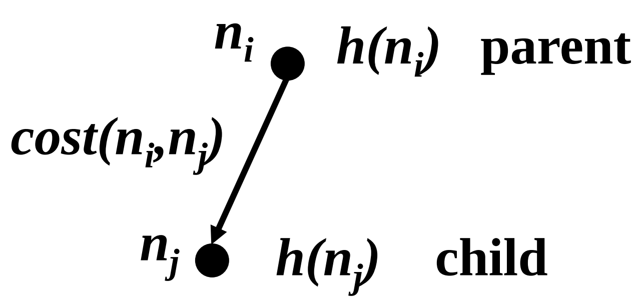
Ex.#	f1	f2	h
Ex1	7	8	14
…			
Ex100	5	2	5

- **We use this classifier on new data, i.e. given *f1* and *f2*, to predict *h* which is unknown. No guarantee that the learned heuristic is admissible or consistent.**

Back to A* and another property of the heuristics…

Consistent (Monotonic) Heuristic

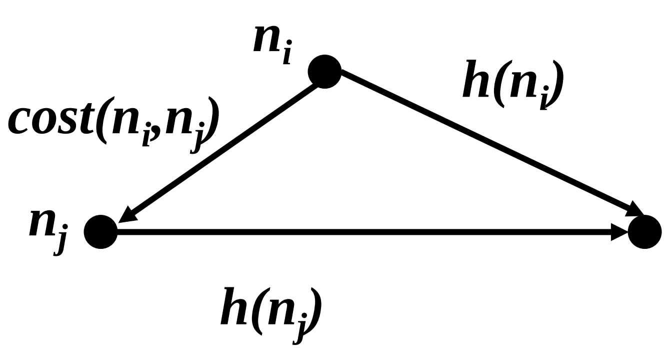
- **Consider a pair of nodes *ni* and *nj*, where *ni* is the parent of *nj***

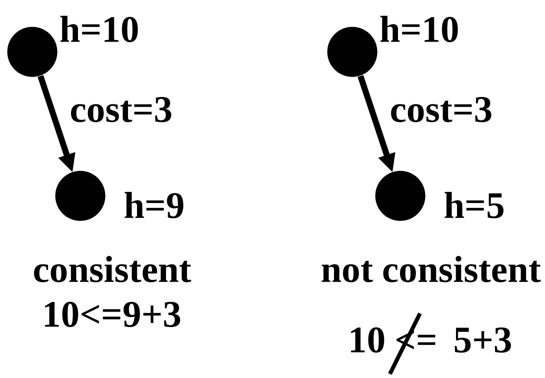


- ***h* is a *consistent (monotonic) heuristic*, if for all such pairs in the search graph the following triangle inequality is satisfied:**

$h(ni) \leq cost(ni,nj) + h(nj)$ **for all *n***

parent **child**



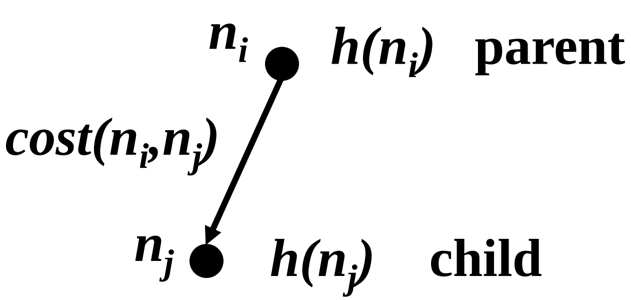


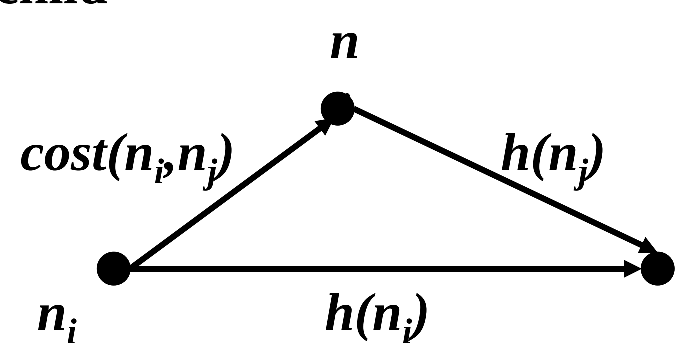
Another Interpretation of the Triangle Inequality

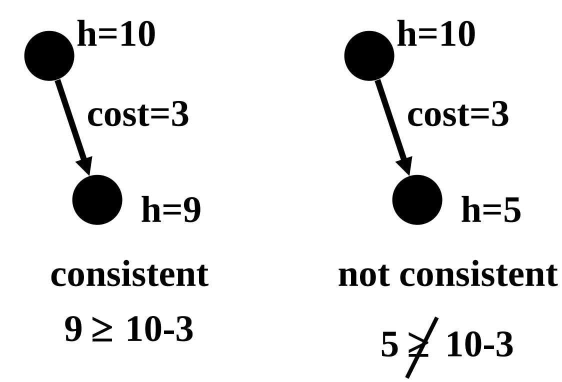
$h(ni) \le cost(ni,nj) + h(nj)$ **for all** n

parent **child**

- **=>** $h(n_j) \ge h(n_i) - cost(n_i,n_j)$ **, i.e. along any path our estimate of the remaining cost to the goal cannot decrease by more than the arc cost**







Consistency Theorems

- **Theorem 1: If *h(n)* is consistent, then $\underset{\text{child}}{f(nj)} \geq \underset{\text{parent}}{f(ni)}$, i.e. *f* is non-decreasing along any path**

Given: $h(ni) \leq c(ni, nj)+h(nj)$

To prove: $f(nj) \geq f(ni)$

Proof: $f(nj)=\underbrace{g(nj)}+h(hj)=$

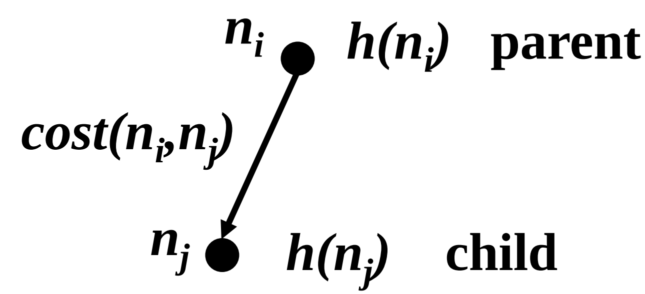
$=g(ni)+\underbrace{c(ni,nj)+h(nj)}=$

$\geq g(ni) + h(ni) =$

$=f(ni)$

$\Rightarrow f(nj) \geq f(ni)$

deff. h(n) consistent



- **Theorem 2: If $f(nj) \geq f(ni)$, i.e. *f* is non-decreasing along any path, then *h(n)* is consistent**

Admissibility and Consistency

- **Consistency is the stronger condition**
- **Theorems:**
  - **If a heuristic is consistent, it is also admissible**

    consistent => admissible

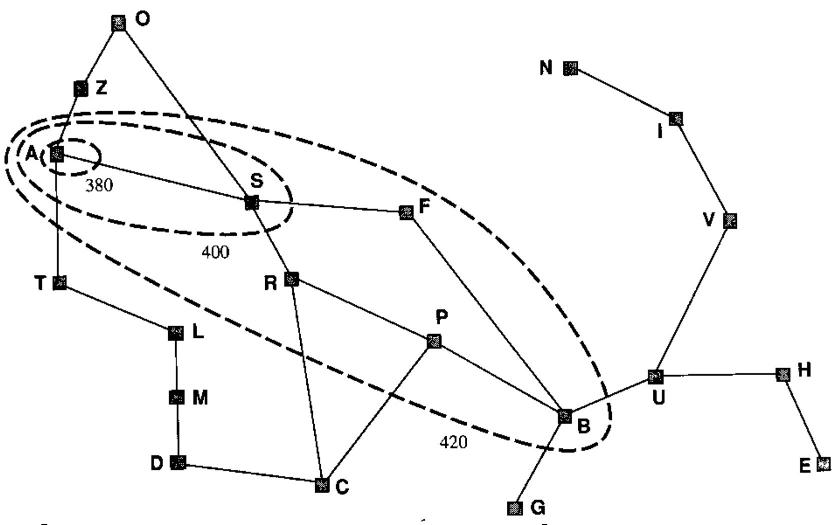
  - **If a heuristic is admissible, there is no guarantee that it is consistent**

    admissible ≠> consistent

Completeness of A* with Consistent Heuristic – Intuitive Idea

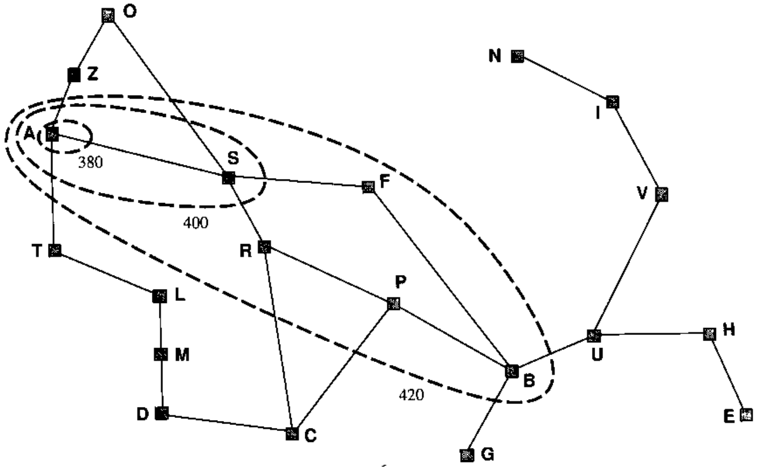
- A* uses the f-cost to select nodes for expansion
- If *h* is consistent, the f-costs are non-decreasing => we can draw f-contours in the state space
- A* expands nodes in order of increasing f-values, i.e.
  - It gradually adds f-contours of nodes
  - Nodes inside a contour have f-cost less than or equal to the contour value

- Completeness – as we add bands of increasing f, we must eventually reach a band where f=h(G)+g(G)=h(G)



Optimality of A* with Consistent Heuristic – Intuitive Idea

- **A* finds the optimal solution, i.e. the one with smallest path cost g(n) among all solutions**
- **The first solution must be the optimal one, as subsequent contours will have higher f-cost, and thus higher g-cost (h(n)=0 for goal nodes):**
  - **Bands f1<f2<f3…**
  - **Compare 2 solutions at band 2 and 3: G2 and G3 (G2 will be found first)**
  - **f(G2)=g(G2)+h(G2), f(G3)=g(G3)+h(G3)**
  - **But f(G2)<f(G3) and h(G2)=h(G3)=0 => g(G2)<g(G3), i.e. the first solution found is the optimal**



A* with Consistent Heuristic is Optimally Efficient

- **Theorem: If *h* is a consistent heuristic, then A* is *optimally efficient* among all optimal search algorithms using *h***
  - **no other optimal algorithm using *h* is guaranteed to expand fewer nodes than A***

- **Which are the optimal algorithms we have studied so far?**
- **Which are the optimal heuristic algorithms we have studied so far?**

Properties of A*

- **<u>Complete?</u> Yes, unless there are infinitely many nodes with f ≤ f(G), G – optimal goal state**
- **<u>Optimal?</u> Yes, with admissible heuristic**
- **<u>Time?</u> Exponential $O(b^d)$**
- **<u>Space?</u> Exponential, keeps all nodes in memory**

- **For most problems, the number of nodes which have to be expanded is exponential**
- **Both time and space are problems for A* but space is the bigger problem - A* runs out of space long before it runs out of time; solution: Iterative Deepening A* (IDA*) or Simplified Memory-Bounded A* (SMA*)**

Summary of A*

- **An admissible heuristic never overestimates the true distance to a goal**
- **A consistent (monotonic) heuristic satisfies the triangle equation**

- ***h(n)* satisfies the triangle equation <=> *f(n)* does not decrease along any path**

- **Admissible ≠> consistent**
- **Consistent => admissible**

- **Dominant heuristic**
  - **given 2 admissible heuristics *h*1 and *h2*, *h2* is dominant if it gives a better estimate of the true cost to a goal node**
  - **A* with a dominant heuristic will expand fewer nodes**

Summary of A* (2)

- If *h(n)* is admissible, A* is optimal
- If *h(n)* is consistent, A* is optimally efficient - A* will expand less or equal number of nodes than any other optimal algorithm using *h(n)*

- However, theoretical completeness and optimality do not mean practical completeness and optimality if it takes too long to get the solution (time and space are exponential)
- => If we can't design an accurate admissible or consistent heuristic, it may be better to settle for a <u>non-admissible</u> heuristic that works well in practice or for a local search algorithm (next lecture) even though completeness and optimality are no longer guaranteed.
- => Also, although dominant (i.e. good) heuristics are better, they may need a lot of time to compute; it may be better to use a simpler heuristic - more nodes will be expanded but overall the search may be faster.

COMP3308/COMP3608, Lecture 3b

ARTIFICIAL INTELLIGENCE

Local Search Algorithms

Reference: Russell and Norvig, ch. 4

Outline

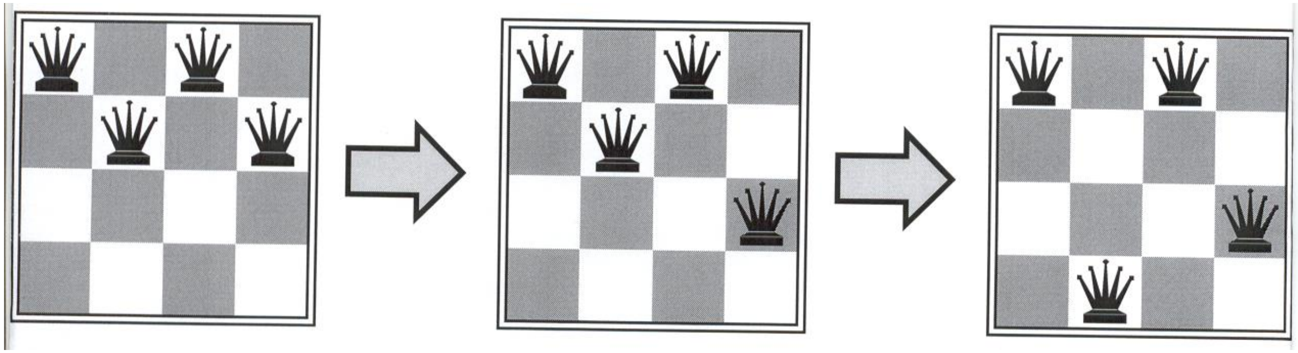
- **Optimisation problems**
- **Local search algorithms**
  - **Hill-climbing**
  - **Beam search**
  - **Simulated annealing**
  - **Genetic algorithms**

Optimisation Problems

- **Problem setting so far: path finding**
  - **Goal: find a path from S to G**
  - **Solution: the path**
  - **Optimal solution: least cost path**
  - **Search algorithms:**
    - **Uninformed: BFS, UCS, DFS, IDS**
    - **Informed: greedy, A***
- **Now a new problem setting: optimisation problem**
  - **Each state has a value *v***
  - **Goal: find the optimal state**
    - **= the state with the highest or lowest *v* score (depending on what is desirable, maximum or minimum)**
  - **Solution: the state; the path is not important**
- **A large number of states => can't be enumerated**
  - **=> We can't apply the previous algorithms – too expensive**

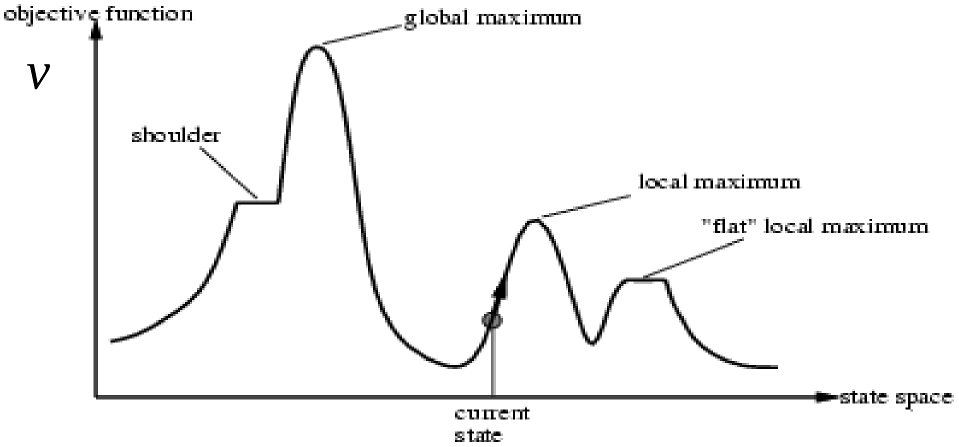
Optimisation Problems - Example

- **n-queens problem**
  - **The solution is the goal configuration, not the path to it**
- **Non-incremental formulation**
  - ***Initial state:*** **n-queens on the board (given or randomly chosen)**
  - ***States:*** **any configuration with n-queens on the board**
  - ***Goal:*** **no queen is attacking each other**
  - **Operators: "move a queen" or "move a queen to reduce the number of attacks"**



V-value Landscape

- **Each state has a value *v* that we can compute**
- **This value is defined by a heuristic *evaluation function* (also called *objective function*)**
- **Goal - 2 variations depending on the task:**
  - **find the state with the highest value (global maximum) or**
  - **find the state with the lowest value (global minimum)**
- **Complete local search – finds a goal state if one exists**
- **Optimal local search – finds the best goal state – the state associated with the global maximum/minimum**



Hill-Climbing Algorithm - Idea

- **Also called *iterative improvement* algorithm**
- **Idea: Keep only a single state in memory, try to improve it**

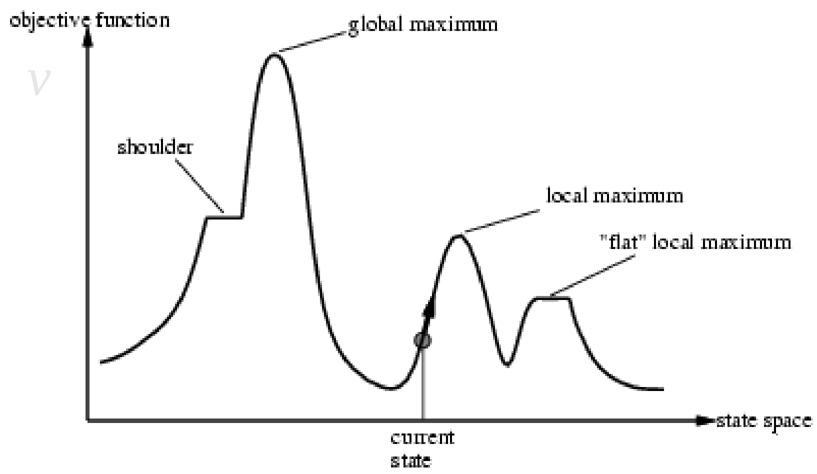
- **Two variations:**
  - **Steepest *ascent* – the goal is the *maximum* value**
  - **Steepest *descent* – the goal is the *minimum* value**

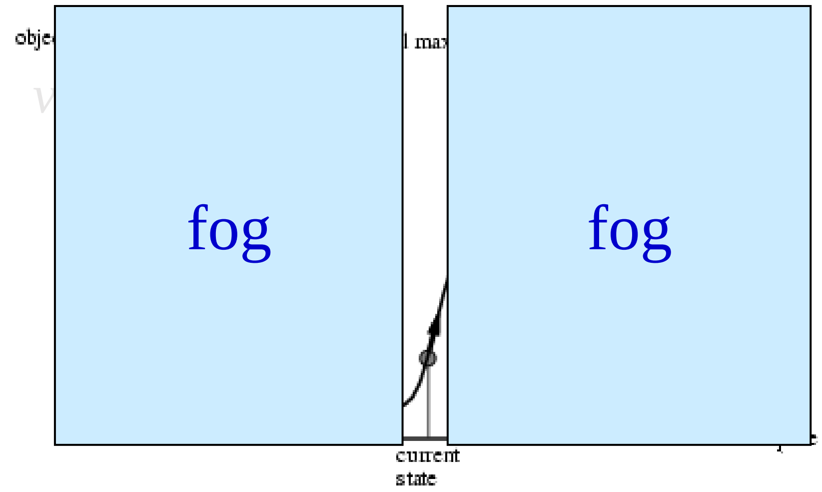
Hill-climbing Search

- **Assume that we are looking for a maximum value (i.e. hill-climbing <u>ascend</u>)**
- **Idea: move around trying to find the highest peak**
  - **Store only the current state**
  - **Do not look ahead beyond the immediate neighbors of the current state**
  - **If a neighboring state is better, move to it and continue, otherwise stop**
  - **"Like climbing Everest in <u>thick fog</u> <u>with amnesia</u>"**

we can't see the whole landscape, only the neighboring states

keeps only 1 state in memory, no backtracking to previous states





Hill-climbing Algorithm

Hill-climbing <u>descent</u>

1) **Set current node *n* to the initial state *s***

(The initial state can be given or can be randomly selected)

2) **Generate the successors of *n*. Select the best successor n_{best} ; *it is* the successor with the best *v* score, *v(best)* (i.e. the <u>lowest</u> score for <u>descent</u>)**

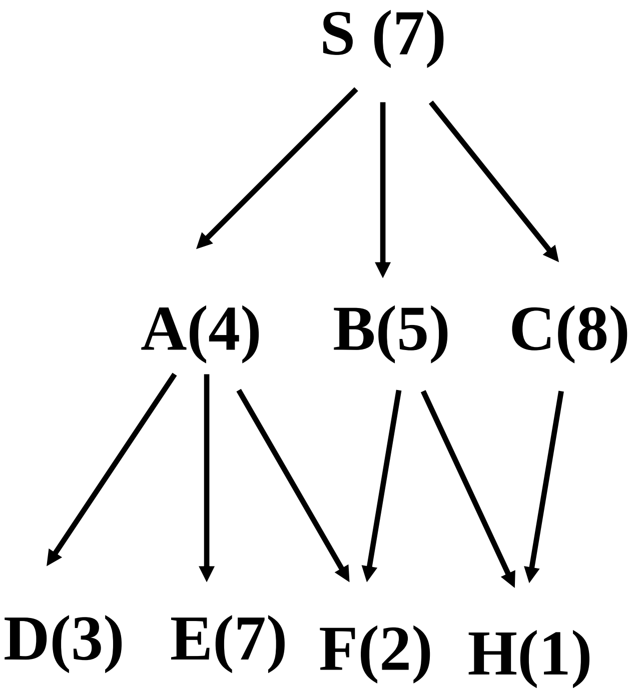
3) **If *v(best)* > *v(n)*, return *n*** **//compare the child with the parent; if child not //better – stop; local or global minimum found**

Else set *n* to n_{best} . Go to step 2 **//if better - accept the child and keep // searching**

- **Summary: Always expands the best successor, no backtracking**

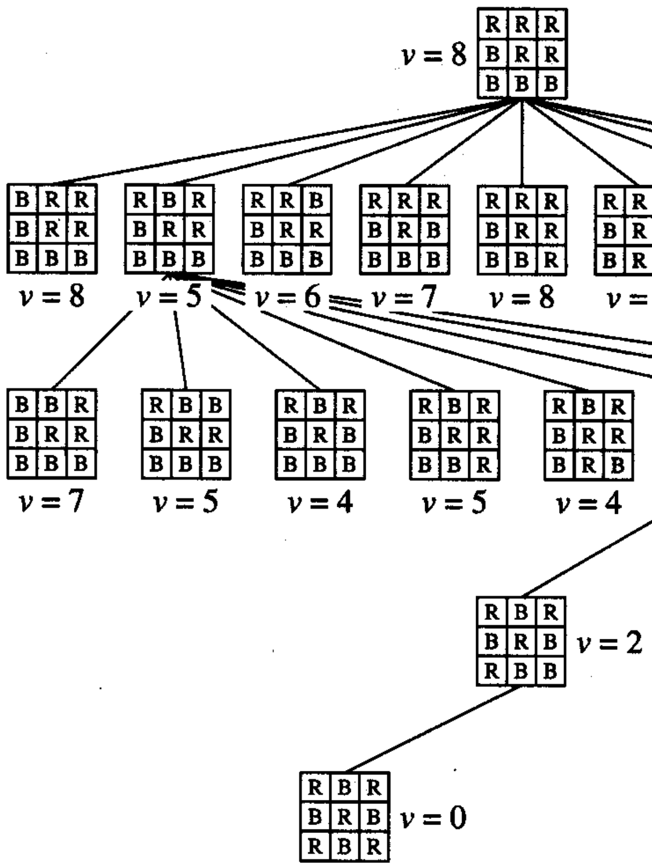
Hill-climbing – Example 1

- ***v*** **- value is in brackets; the lower, the better (i.e. descent)**
- **Expanded nodes: SAF**



Hill-climbing – Example 2

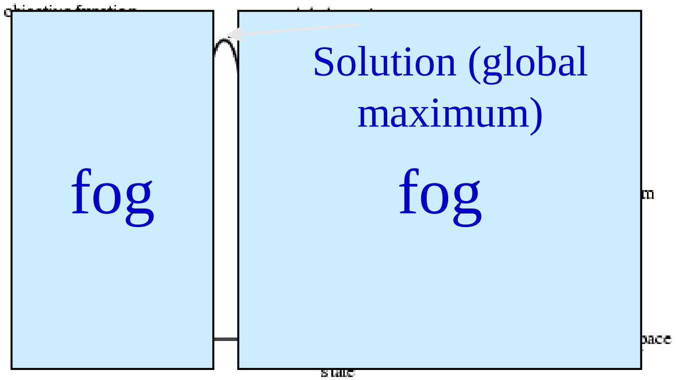
- **Given: 3x3 grid, each cell is colored either red (R) or blue (B)**
- **Aim: find the coloring with minimum number of pairs of adjacent cells with the same color**
- **Ascending or descending?**

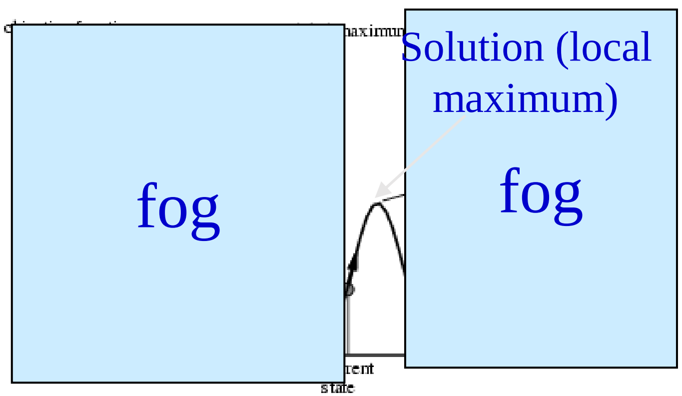


v **– # pairs of adjacent cells with the same color**

Picture from N. Nielsen, AI, 1998

Hill-climbing Search





Weaknesses:

- Not a very clever algorithm – can easily get stuck in a local optimum (maximum/minimum)
- However, not all local maxima/minima are bad – some may be reasonably good even though not optimal

Advantages: good choice for hard, practical problems

- Uses very little memory
- Finds reasonable solutions in large spaces where systematic algorithms are not useful

Not complete, not optimal but keeps just one node in memory!

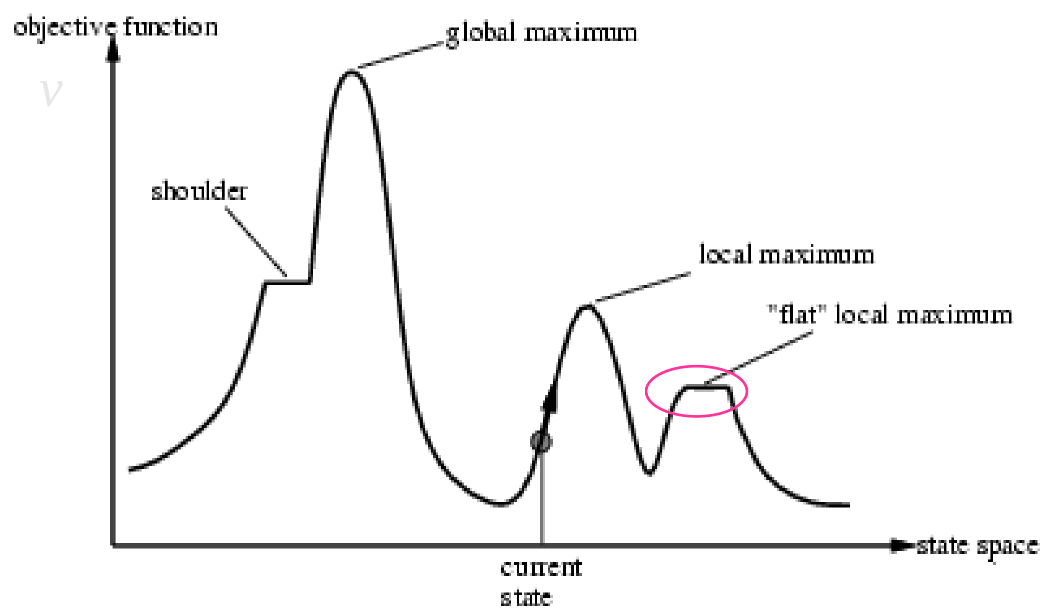
Hill-climbing – Escaping Bad Local Optima

- **Hill climbing finds the closest local optimum (minimum or maximum, depending on the version – descent or ascent)**
  - **Which may or may not be the global optimum**

- **The solution that is found depends on the initial state**
- **When the solution found is not good enough - random restart:**
  - **run the algorithm several times starting from different points; select the best solution found (i.e. the best local optimum)**
  - ***If at first you don't succeed, try, try again!***
  - **This is applicable for tasks without a fixed initial state**

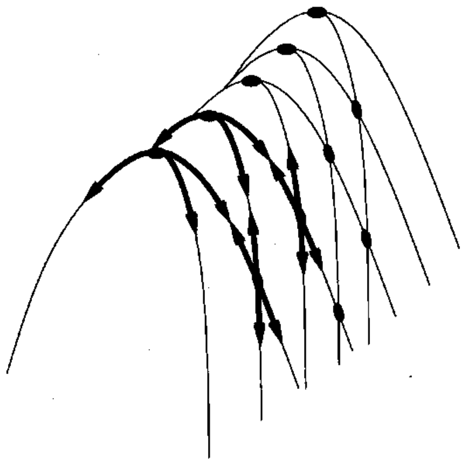
Hill-climbing – Escaping Bad Local Optima (2)

- **Plateaus (flat areas): no change or very small change in *v***
- **Our version of hill-climbing does not allow visiting states with the same *v* as it terminates if the best child's *v* is the same as the parent's**
- **But other versions keep searching if the values are the same and this may result in visiting the same state more than once and walking endlessly**
  - **Solution: keep track of the number of times *v* is the same and do not allow revisiting of nodes with the same *v***



Hill-climbing – Escaping Bad Local Optima (3)

- **Ridges – the current local maximum is not good enough; we would like to move up but all moves go down**
  - **Example:**



- Dark circles = states

- **A sequence of local maxima that are <u>not</u> connected with each other**
- **- From each of them all available actions point downhill.**

  - **Possible solutions: combine 2 or more moves in a macro move that can increase the height or allow a limited number of look-ahead search**

Beam Search

- **It keeps track of *k* states rather than just 1**

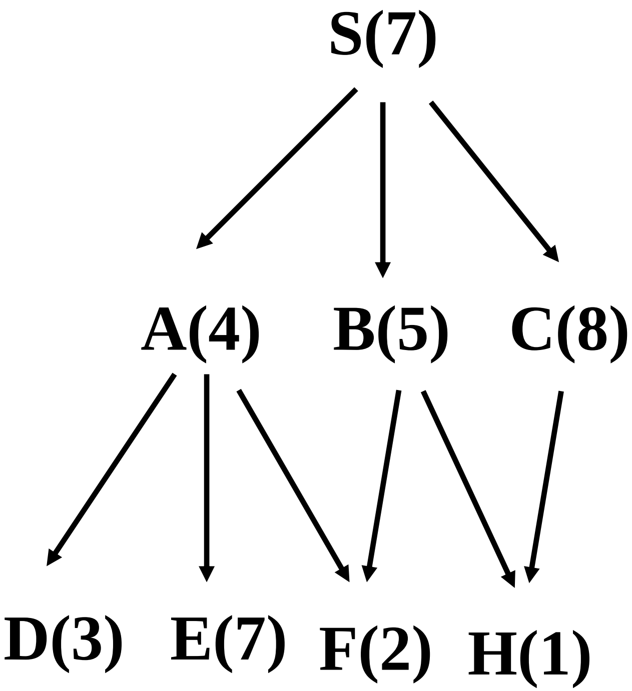
- **Version 1: Starts with <u>1 given state</u>**
- **At each level: generate all successors of the given state**
- **If any one is a goal state, stop; else select the *k* best successors from the list and go to the next level**

- **Version 2: Starts with <u>*k* randomly generated states</u>**
- **At each level: generate all successors of all *k* states**
- **If any one is a goal state, stop; else select the *k* best successors from the list and go to the next level**

- **In nutshell: keeps only *k* best states**

Beam Search - Example

- **Consider the version that starts with 1 given state**
- **Starting from S, run beam search with k=2 using the values in brackets as evaluation function (the smaller, the better)**
- **Expanded nodes = ?**



- S
- generate ABC
- select AB (the best 2 children)
- generate DEFH
- select FH (the best 2 children)
- expanded nodes: SABFH

Beam Search and Hill-Climbing Search

- **Compare beam search with 1 initial state and hill climbing with 1 initial state**
  - **Beam – 1 start node, at each step keeps *k* best nodes**
  - **Hill climbing – 1 start node, at each step keeps 1 best node**

- **Compare beam search with *k* random initial states and hill-climbing with *k* random initial states**
  - **Beam – *k* starting positions, *k* threads run in parallel, passing of useful information among them as at each step the best *k* children are selected**
  - **Hill climbing – *k* starting positions, *k* threads run individually, no passing of information among them**

Beam Search with A*

- **Recall that memory was a big problem for A***
- **Idea: keep only the best *k* nodes in the fringe, i.e. use a priority queue of size *k***

- **Advantage: memory efficient**
- **Disadvantage: neither complete, nor optimal**

Simulated Annealing

- **What is annealing in metallurgy?**
  - **a material's temperature is gradually decreased (very slowly) allowing its crystal structure to reach a minimum energy state**
- **Similar to hill-climbing but selects a random successor instead of the best successor (step 2 below)**

1) Set current node *n* to the initial state *s*.

Randomly select *m*, one of *n*'s successors

2) If *v(m)* is better than *v(n)*, *n=m* //accept the child *m*

Else *n=m* with a probability *p* //accept the child *m* with probability *p*

3) Go to step 2 until a predefined number of iterations is reached or the state reached (i.e. the solution found) is good enough

The Probability *p*

parent ***n***

↓

child ***m***

- **Assume that we are looking for a minimum**
- **There are different ways to define *p*, e.g.**

$$p = e^{\frac{v(n)-v(m)}{T}}$$

- **Main ideas:**
  - **1) *p* decreases exponentially with the badness of the child (move) and**
  - **2) bad children (moves) are more likely to be allowed at the beginning than at the end**

- **nominator – shows how good the child *m* is**
  - **Bad move (the child is worse than the parent): v(n)<v(m), e.g.**
  - **case1: v(n)=10, v(m)=20, p1=e^-10/T**
  - **case2: v(n)=10, v(m)=11, p2=e^-1/T**
  - **m (the child) in case1 is worse than in case2**
  - **p1<p2 as T is positive => p exponentially decreases with the badness of the move**

The Probability *p* (2)

$$p = e^{\frac{v(n) - v(m)}{T}}$$

- **denominator: parameter *T* that decreases (anneals) over time based on a *schedule*, e.g. *T=T*0.8***
  - **high *T* – bad moves are more likely to be allowed**
  - **low *T* – more unlikely; becomes more like hill-climbing**
  - ***T* decreases with time and depends on the number of iterations completed, i.e. until "bored"**
- **Some versions have an additional step (Metropolis criterion for accepting the child):**
  - ***p* is compared with *p'*, a randomly generated number [0,1]**
  - **If *p > p'*, accept the child, otherwise reject it**
- **In summary, simulated annealing combines a hill-climbing step (accepting the best child) with a *random walk step* (accepting bad children with some probability). The random walk step can help escape bad local minima.**

Simulated Annealing - Theorem

- **What is the correspondence?**
  - ***v*** **– total energy of the atoms in the material**
  - ***T*** **– temperature**
  - ***schedule*** **– the rate at which *T* is lowered**

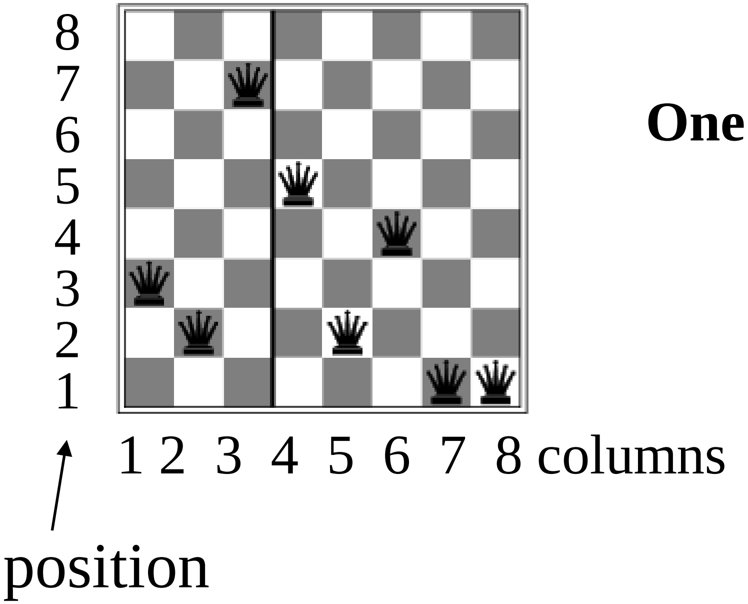
- **Theorem:** **If the schedule lowers *T* slowly enough, the algorithm will find global optimum**
  - **i.e. is complete and optimal given a long enough cooling schedule => annealing schedule is very important**
- **Simulated annealing has been widely used to solve VLSI layout problems, factory scheduling and other large-scale optimizations**
- **It is easy to implement but a "slow enough" schedule may be difficult to set**

Genetic Algorithms

- **Inspired by mechanisms used in evolutionary biology, e.g. selection, crossover, mutation**
- **Similar to beam search, in fact a variant of stochastic beam search**

- **Each state is called an *individual*. It is coded as a string.**
- **Each state *n* has a fitness score *f(n)* (evaluation function). The higher the value, the better the state.**
- **Goal: starting with *k* randomly generated individuals, find the optimal state**
- **Successors are produced by selection, crossover and mutation**
- **At any time keep a fixed number of states (the population)**

Example – 8-queens Problem



One possible encoding is (3 2 7 5 2 4 1 1)

column 1: a queen at position 3

column 2: a queen at position 2

…

Example – 8-queens Problem (2)

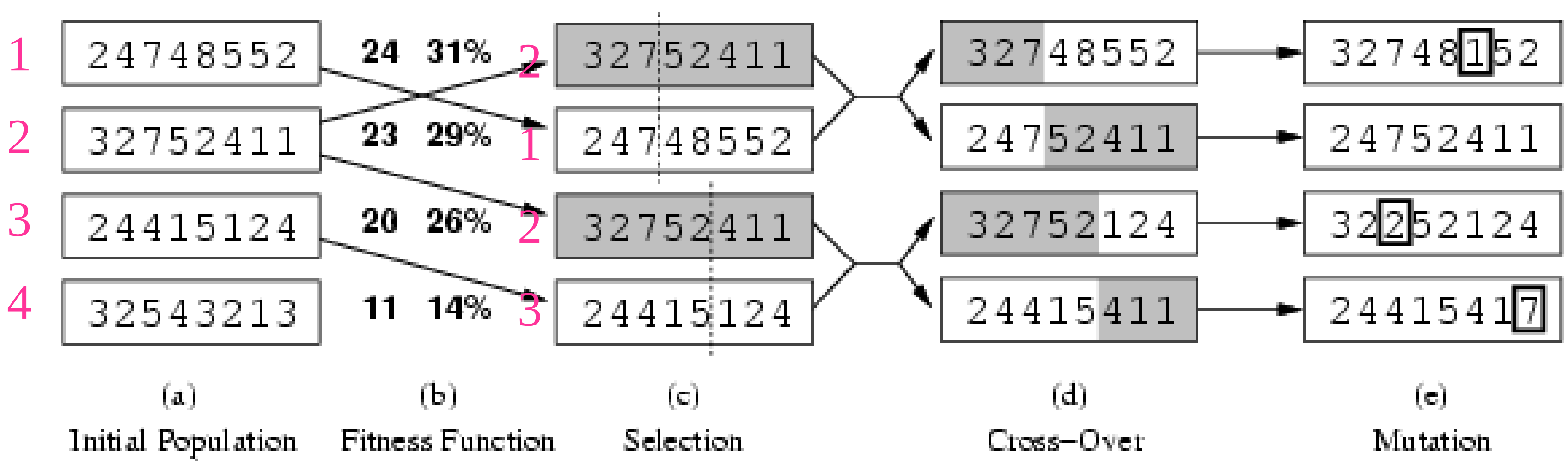
- Suppose that we are given 4 individuals (initial population) with their fitness values
  - Fitness values = number of non-attacking pairs of queens (given 8 queens, how many different pairs of queens are there? 28 => max value is 28)
  - Let the probability for reproduction is proportional to the fitness (expressed is %)

24748552	24	31%
32752411	23	29%
24415124	20	26%
32543213	11	14%

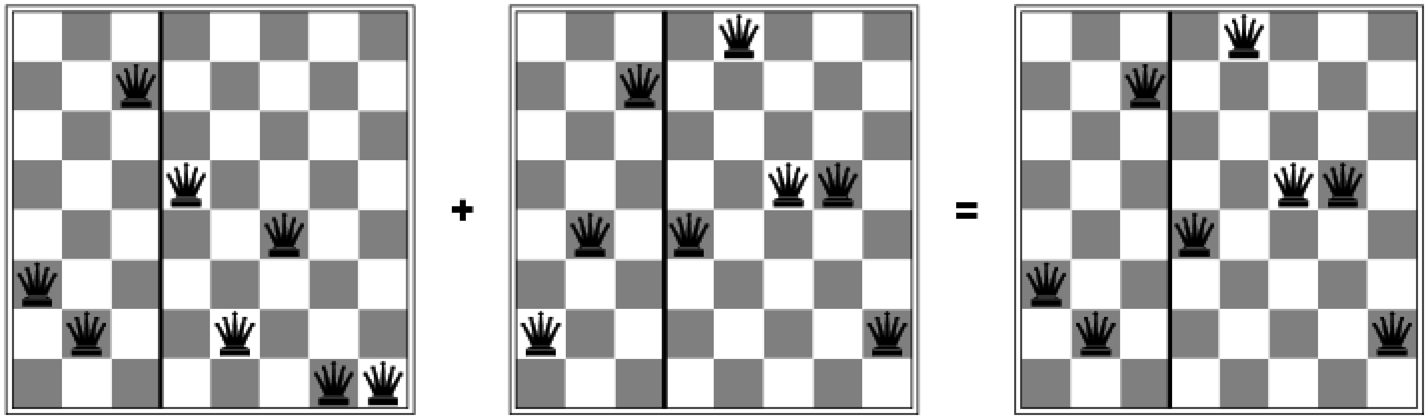
(a)
Initial Population

Example – 8-queens Problem (3)

- *Select* 4 individuals for reproduction based on the fitness function
  - The higher the fitness function, the higher the probability to be selected
  - Let individuals 2, 1, 2 and 3 be selected, i.e. individual 2 is selected twice while 4 is not selected
- *Crossover* – random selection of crossover point; crossing over the parents strings
- *Mutation* – random change of bits (in this case 1 bit was changed in each individual)



A Closer Look at the Crossover



(3 2 7 5 2 4 1 1) + (2 4 7 4 8 5 5 2) = (3 2 7 4 8 5 5 2)

- **When the 2 states are different, crossover produces a state which is a long way from either parents**
- **Given that the population is diverse at the beginning of the search, crossover takes big steps in the state space early in the process and smaller later, when more individuals are similar**

Genetic Algorithm – Pseudo Code (1 variant)

from http://pages.cs.wisc.edu/~jerryzhu/cs540.html

1. Let $s_1, \ldots, s_N$ be the current population
2. Let $p_i = f(s_i) / \Sigma_j f(s_j)$ be the reproduction probs
3. FOR $k = 1$; $k<N$; k+=2
   - parent1 = randomly pick s with probs p
   - parent2 = randomly pick another s with probs p
   - randomly select a crossover point, swap strings of parents 1, 2 to generate children $t[k]$, $t[k+1]$
4. FOR $k = 1$; $k<=N$; k++
   - Randomly mutate each position in $t[k]$ with a small probability
5. The new generation replaces the old: $\{ s \} \leftarrow \{ t \}$.

   Repeat **until some individual is fit enough or a predefined maximum number of iterations has been reached**

Genetic Algorithms - Discussion

- **Combine:**
  - **uphill tendency (based on the fitness function)**
  - **random exploration (based on crossover and mutation)**
- **Exchange information among parallel threads - the population consists of several individuals**
- **The main advantage comes from crossover**
- **Success depends on the representation (encoding)**
- **Easy to implement**
- **Not complete, not optimal**

Links

Simulated annealing as a training algorithm for backpropagation neural networks:

- **R.S. Sexton, R.E. Dorsey, J.D. Johnson, *Beyond backpropagation: using simulated annealing for training neural networks,* people.missouristate.edu/randallsexton/sabp.pdf**