Informed Search Methods

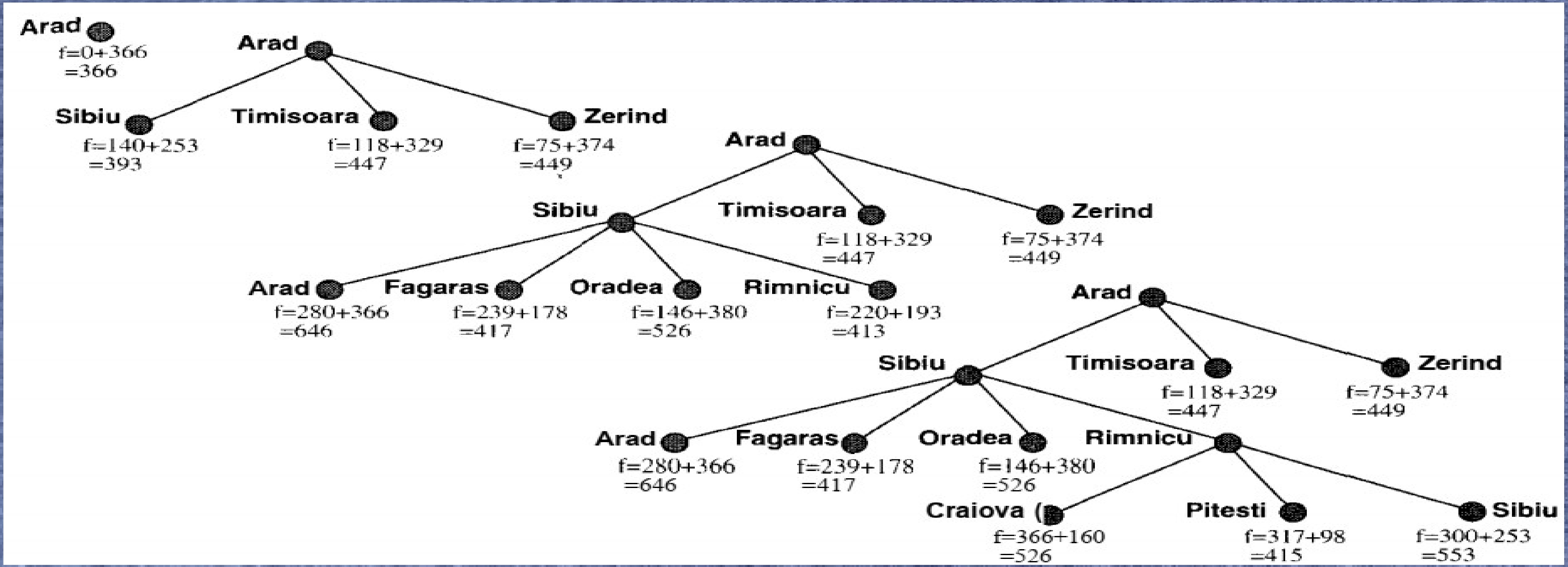
Sunil Surve
Fr. Conceicao Rodrigues College of Engineering

A* Algorithm

- Greedy search minimizes the estimated cost to the goal, *h(n),* and thereby cuts the search cost considerably
- Uniform-cost search, on the other hand, minimizes the cost of the path so far, *g(n);* it is optimal and complete, but can be very inefficient
- Combining the two evaluation functions simply by summing them:
  f(n) = g(n) + h(n)
  where *g(n)* is the path cost from the start node to node *n*, *h(n) is the estimated cost of the cheapest path from n to the goal* *and* f(n) is estimated cost of the cheapest solution through *n*
- **Admissible heuristic**
- ***If h is admissible, f (n) never overestimates the actual cost of* the best solution through n**

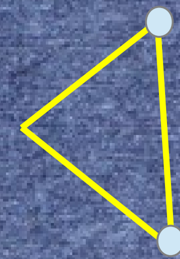
A* Algorithm

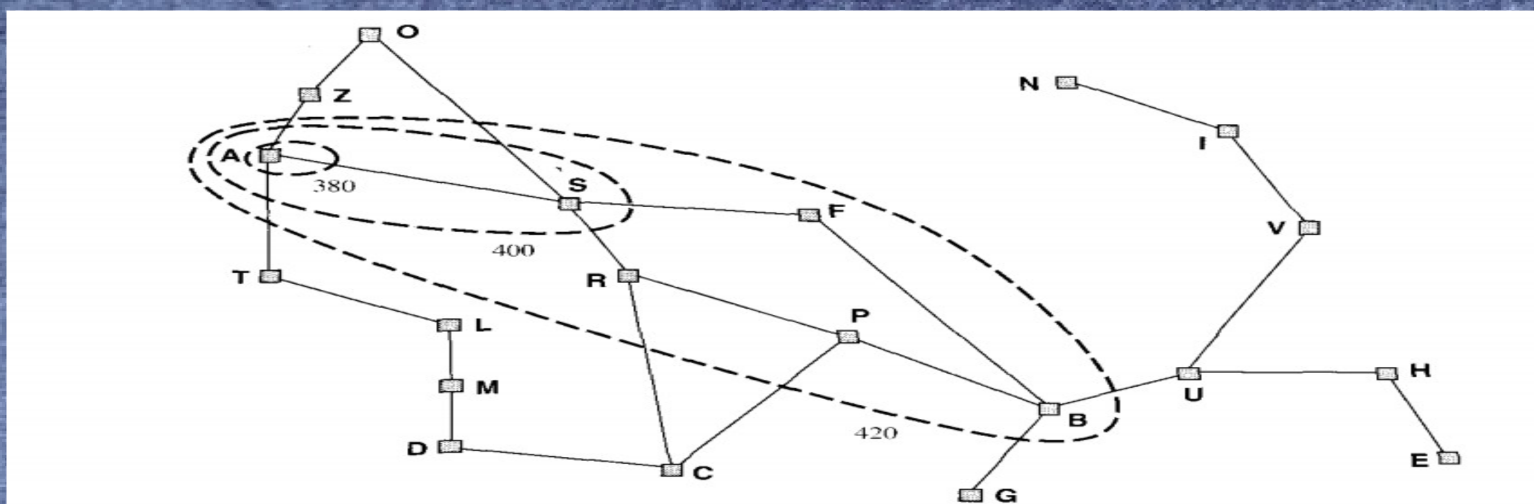
- Along any path from the root, the f-cost never decreases.
- It holds true for almost all admissible heuristics.
- A heuristic for which it holds is said to exhibit **monotonicity**



A* Algorithm

- If f-cost of child is less than parent's f-cost, then heuristic used is nonmonotonic.
- Pathmax equation f(n') = max(f(n), g(n') + h(n'))
- If/ never decreases along any path out from the root, we can conceptually draw contours in the state space





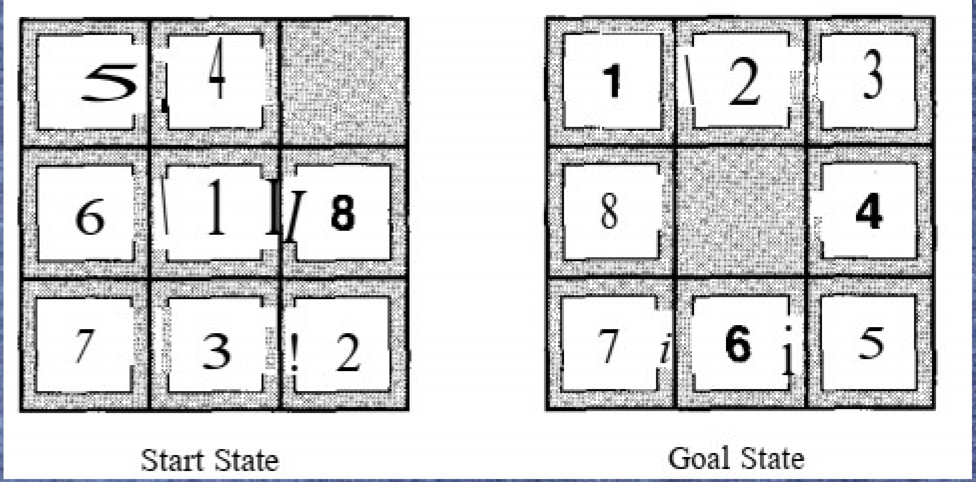
A* Algorithm

- If we define/* to be the cost of the optimal solution path, then we can say the following:
    - A* expands all nodes with f(n) < f*.
    - A* may then expand some of the nodes right on the "goal contour," for which f(n) = f*, before selecting a goal node
- First solution found must be the optimal one, because nodes in all subsequent contours will have higher/-cost, and thus higher g-cost
- A* is **optimally efficient** for any given heuristic function

function A*-SEARCH(*problem)* **returns** a solution or failure
return BEST-FIRST-SEARCH(problem,g + *h)*

Heuristic Functions

- 8-puzzle - the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the initial configuration matches the goal configuration
- The branching factor is about 3, and 320 = 3.5 x 109 states and By keeping track of repeated states, there are only 9! = 362,880 states



Heuristic Functions

- h_1 = the number of tiles that are in the wrong position. For Figure 4.7, none of the 8 tiles is in the goal position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic
- h_2 = the sum of the distances of the tiles from their goal positions. This is sometimes called the **city block distance** or **Manhattan distance.** *h2* is also admissible $h_2 = 2 + 3 + 2 + 1+2 + 2+ 1+2 =15$
- Which heuristic function is better? h_1 or h_2?
- h_2 will expand lesser nodes than h_1.

Inventing heuristic functions

- Relaxed problem
- If a collection of admissible heuristics $h_1,\ldots,h_m$ is available for a problem, and none of them dominates any of the others, which should we choose?
  $h(n) = max(h_1(n),\ldots,h_m(n))$.
- *Another way to invent a good heuristic is to use statistical information*

Iterative deepening A* search (IDA*)

- Use an f-cost limit
- Each iteration expands all nodes inside the contour for the current f-cost
- Once the search inside a given contour has been completed, a new iteration is started using a newf-cost for the next contour
- IDA* is complete and optimal with the same caveats as A* search IDA* is complete and optimal with the same caveats as A* search
- Requires space proportional to the longest path that it explores.
- If *b* is the smallest operator cost and f* the optimal solution cost, then in the worst case, IDA* will require *bf*/8* nodes of storage.
- In most cases, *bd* is a good estimate of the storage requirements

Iterative deepening A* search (IDA*)

- IDA* has difficulty in more complex domains
- In the travelling salesperson problem, for example, the heuristic value is different for every state. This means that each contour only includes one more state than the previous contour.
- If A* expands *N* nodes, IDA* will have to go through *N* iterations and will therefore expand $1 + 2 + \bullet \bullet - + N = O(N^2)$ nodes
- One way around this is to increase the f-cost limit by a fixed amount ε on each iteration, so that the total number of iterations is proportional to $1/\varepsilon$.
- This can reduce the search cost, at the expense of returning solutions that can be worse than optimal by at most e. Such an algorithm is called **ε-admissible**

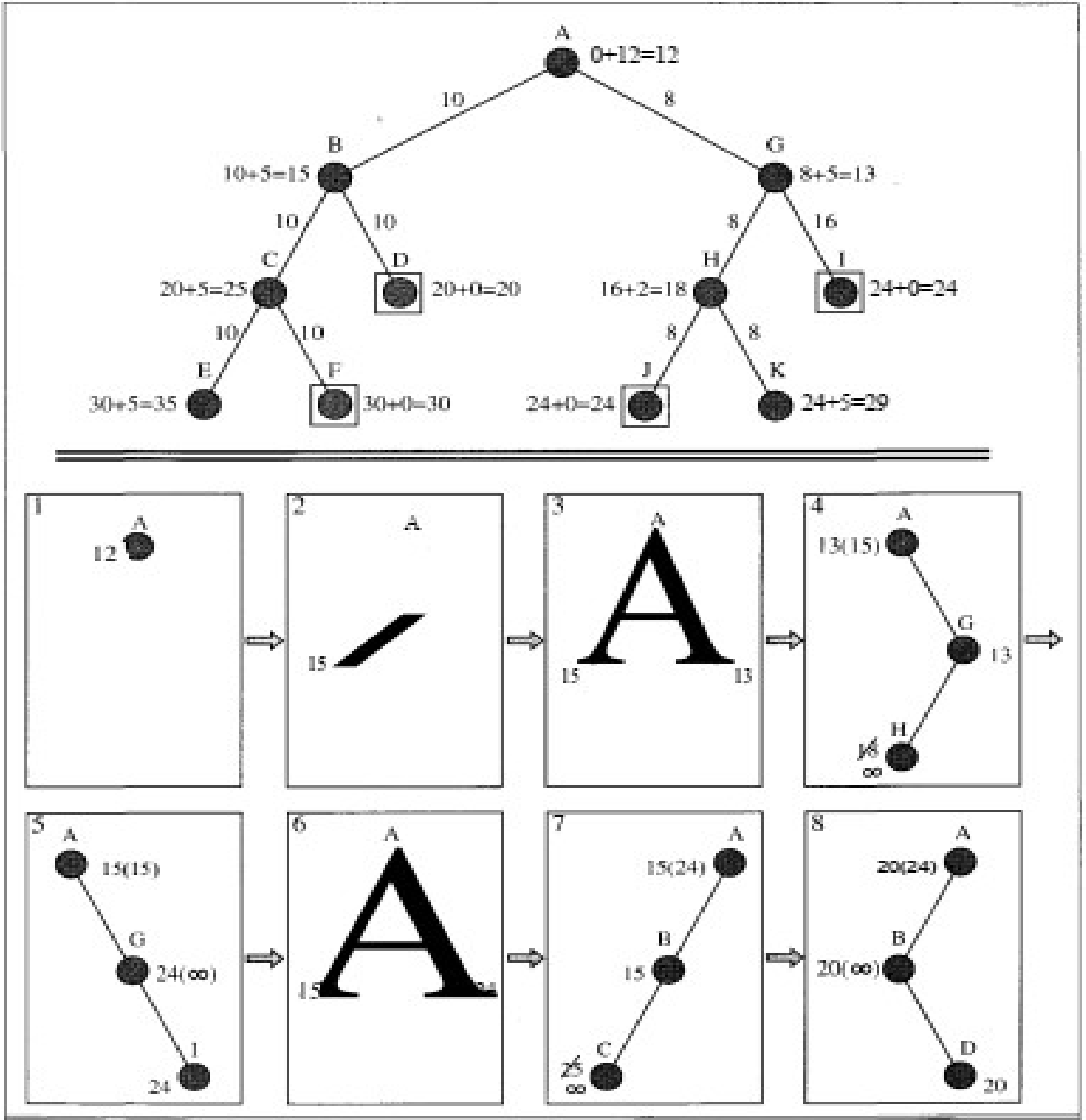
Iterative deepening A* search (IDA*)

```
function IDA*(problem) returns a solution sequence
   inputs: problem, a problem
   static: f-limit, the current f-COST limit
           mot, a node

   root ← MAKE-NODE(INITIAL-STATE[problem])
   f-limit ← f-COST(root)
   loop do
       solution, f-limit — DFS-CONTOUR(root,f-limit)
       if solution is non-null then return solution
       itf-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node,f-limit) returns a solution sequence and a new f-COST limit
   inputs: node, a node
           f-limit, the current f-COST limit
   static: next-f, the f-COST limit for the next contour, initially ∞

   if f-COST[node] > f-limit then return null, f-COST[node]
   if GOAL-TEST[problem](STATE[node]) then return node, f-limit
   for each node s in SUCCESSORS(node) do
       solution, new-f ← DFS-CONTOUR(s,f-limit)
       if solution is non-null then return solution, f-limit
       next-f ← MIN(next-f, new-f); end
   return null, next-f
```

Simplified Memory-Bounded A*

- SMA* has the following properties:
  - It will utilize whatever memory is made available to it.
  - It avoids repeated states as far as its memory allows.
  - It is complete if the available memory is sufficient to store the shallowest solution path.
  - It is optimal if enough memory is available to store the shallowest optimal solution path.
  - Otherwise, it returns the best solution that can be reached with the available memory.
  - When enough memory is available for the entire search tree, the search is optimally efficien


A
0+12=12
10
8
B
10+5=15
G
8+5=13
10
10
8
16
C
20+5=25
D
20+0=20
H
16+2=18
I
24+0=24
10
10
8
8
E
30+5=35
F
30+0=30
J
24+0=24
K
24+5=29
1
A
12
2
A
15
3
A
15
13
4
A
13(15)
G
13
H
∞
5
A
15(15)
G
24(∞)
I
24
6
A
15
7
A
15(24)
B
15
C
∞
8
A
20(24)
B
20(∞)
D
20

Simplified Memory-Bounded A*

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
      if Queue is empty then return failure
      n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
      if s is not a goal and is at maximum depth then
          f(s) ← ∞
      else
          f(s) ← MAX(f(n), g(s)+h(s))
      if all of n's successors have been generated then
          update n's f-cost and those of its ancestors if necessary
      if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
          delete shallowest, highest-f-cost node in Queue
          remove it from its parent's successor list
          insert its parent on Queue if necessary
      insert s on Queue
  end
```

Hill-climbing search

- **A loop that continually moves in the direction of increasing value.**
- **The algorithm does not maintain a search tree**
- **Local maxima: a local maximum is a peak that is lower than the highest peak in the state space. Once on a local maximum, the algorithm will halt even though the solution may be far from satisfactory**
- **Plateaux: a plateau is an area of the state space where the evaluation function is essentially flat.**
- **Ridges:** a ridge may have steeply sloping sides, so that the search reaches the top of the ridge with ease, but the top may slope only very gently toward a peak