Problem Solving State-Space Search and Control Strategies

Chapter 2(c)

Heuristic Search

- Heuristics are criteria for deciding which among several alternatives be the most effective in order to achieve some goal.
- Heuristic is a technique that
  - improves the efficiency of a search process possibly by sacrificing claims of systematicity and completeness.
  - It no longer guarantees to find the best answer but almost always finds a very good answer.

Heuristic Search – Contd..

- Using good heuristics, we can hope to get good solution to hard problems (such as travelling salesman) in less than exponential time.
- There are **general-purpose** heuristics that are useful in a wide variety of problem domains.
- We can also construct **special purpose** heuristics, which are domain specific.

General Purpose Heuristics

- A general-purpose heuristics for combinatorial problem is
  - **Nearest neighbor algorithms** which works by selecting the locally superior alternative.
  - For such algorithms, it is often possible to prove an upper bound on the error which provide reassurance that one is not paying too high a price in accuracy for speed.
- In many AI problems
  - It is often hard to measure precisely the goodness of a particular solution.
  - But still it is important to keep performance question in mind while designing algorithm.

Contd…

- For real world problems
  - It is often useful to introduce heuristics based on relatively unstructured knowledge.
  - It is impossible to define this knowledge in such a way that mathematical analysis can be performed.
- In AI approaches
  - Behavior of algorithms are analyzed by running them on computer as contrast to analyzing algorithm mathematically.

Contd..

- There are at least two reasons for the adhoc approaches in AI.
  - It is a lot more fun to see a program do something intelligent than to prove it.
  - AI problem domains are usually sufficiently complex, so generally not possible to produce analytical proof that a procedure will work.
  - It is even not possible to describe the range of problems well enough to make statistical analysisof program behavior meaningful.

Contd..

- One of the most important analysis of the search process is straightforward i.e.,
  - Number of nodes in a complete search tree of depth D and branching factor F is F*D .
- This simple analysis motivates to
  - look for improvements on the exhaustive search.
  - find an upper bound on the search time which can be compared with exhaustive search procedures.

Informed Search Strategies- Branch & Bound Search

- It expands the least-cost partial path. Sometimes, it is called **uniform cost search**.
- **Function g(X)** assigns some cumulative expense to the path from *Start* node to *X* by applying the sequence of operators .
  - *For example*, in salesman traveling problem, g(X) may be the actual distance from *Start* to current node *X*.
- During search process there are many incomplete paths contending for further consideration.

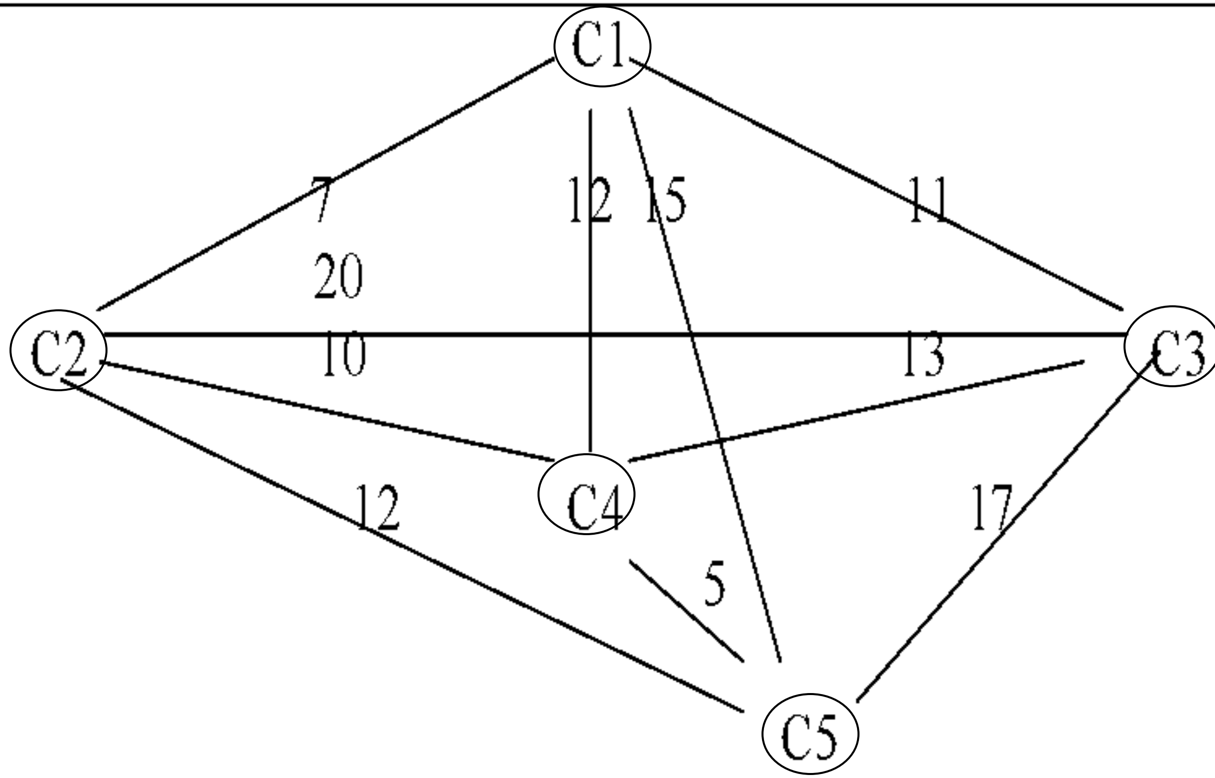
Contd..

- The shortest one is extended one level, creating as many new incomplete paths as there are branches.
- These new paths along with old ones are sorted on the values of function **g**.
- The shortest path is chosen and extended.
- Since the shortest path is always chosen for extension, the path first reaching to the destination is certain to be nearly optimal.

Contd..

- Termination Condition:
  - Instead of terminating when a path is found, terminate when the shortest incomplete path is longer than the shortest complete path.
- If $g(X) = 1$, for all operators, then it degenerates to simple Breadth-First search.
- It is as bad as depth first and breadth first, from AI point of view,.
- This can be improved if we augment it by dynamic programming i.e. delete those paths which are redundant.

Traveling Salesman Problem (Example) – Start city is C1



D(C1,C2) = 7; D(C1,C3) = 11; D(C1,C4) = 12; D(C1,C5) = 15; D(C2,C3) =20;

D(C2,C4) = 10; D(C2,C5) =12; D(C3,C4) =13; D(C3,C5) = 17; D(C4,C5) =5;

Paths explored. Assume C1 to be the start city		Distance
1. C1 → C2 → C3 → C4 → C5 → C1 7 20 13 5 15 27 40 45 60	current best path	60 √ ×
2. C1 → C2 → C3 → C5 → C4 → C1 7 20 17 5 12 27 44 49 61		61 ×
3. C1 → C2 → C4 → C3 → C5 → C1 7 10 13 17 15 17 40 57 72		72 ×
4. C1 → C2 → C4 → C5 → C3 → C1 7 10 5 17 11 17 22 39 50	current best path, cross path at S.No 1.	50 √ ×
5. C1 → C2 → C5 → C3 → C4 → C1 7 12 17 13 12 19 36 49 61		61 ×
6. C1 → C2 → C5 → C4 → C3 → C1 7 12 5 13 11 19 24 37 48	current best path, cross path at S.No. 4.	48 √
7. C1 → C3 → C2 → C4 → C5 → C1 11 20 10 5 15 31 41 46 63		63 ×

Paths explored. Assume C1 to be the start city	Distance
8. C1 → C3 → C2 → C5 → C4 (not to be expanded further) 11 20 12 5 37 49 54	54 ×
9. C1 → C3 → C4 → C2 → C5 → C1 11 13 10 12 15 24 34 46 61	61 ×
10. C1 → C3 → C4 → C5 → C2 → C1 same as current best path at S. No. 6. 11 13 5 12 7 24 29 41 48	48 √
11. C1 → C3 → C5 → C2 (not to be expanded further) 11 17 12 38 50	50 ×
12. C1 → C3 → C5 → C4 → C2 (not to be expanded further) 11 17 5 10 38 43 53	53 ×
13. C1 → C4 → C2 → C3 → C5 (not to be expanded further) 12 10 20 17 22 42 55	59 ×
Continue like this	

Hill Climbing- (Quality Measurement turns DFS into Hill climbing (Variant of generate and test strategy)

- Search efficiency may be improved if there is some way of ordering the choices so that the most promising node is explored first.
- Moving through a tree of paths, hill climbing proceeds in
  - depth-first order but the choices are ordered according to some heuristic value (i.e, measure of remaining cost from current to goal state).

Hill Climbing- Algorithm

Generate and Test *Algorithm*

Start

- Generate a possible solution
- Test to see, if it is goal.
- If not go to start else quit

End

Example of heuristic function

- Straight line (as the crow flies) distance between two cities may be a heuristic measure of remaining distance in traveling salesman problem .

Simple Hill climbing : Algorithm

- Store initially, the root node in a OPEN list (maintained as stack) ; Found = false;
- While (OPEN ≠ empty AND Found = false) Do

{

  - Remove the top element from OPEN list and call it NODE;
  - If NODE is the goal node, then **Found = true** else find SUCCs, of NODE, if any, and **sort SUCCs** by estimated cost from NODE to goal state and add them to the front of OPEN list.

} /* end while */

- If **Found = true** then return **Yes** otherwise return **No**
- Stop

Problems in hill climbing

- There might be a position that is not a solution but from there no move improves situations?
- This will happen if we have reached a *Local maximum*, a *plateau* or a *ridge.*
  - **Local maximum:** It is a state that is better than all its neighbors but is not better than some other states farther away. All moves appear to be worse.
    - *Solution to this is to backtrack to some earlier state and try going in different direction.*

Contd…

- **Plateau:** It is a flat area of the search space in which, a whole set of neighboring states have the same value. It is not possible to determine the best direction.
  - *Here make a big jump to some direction and try to get to new section of the search space.*
- **Ridge:** It is an area of search space that is higher than surrounding areas, but that can not be traversed by single moves in any one direction. (Special kind of local maxima).
  - *Here apply two or more rules before doing the test i.e., moving in several directions at once.*

Beam Search

- Beam Search progresses level by level.
- It moves downward from the best **W** nodes only at each level. Other nodes are ignored.
  - W is called **width** of beam search.
- It is like a BFS where also expansion is level wise.
- Best nodes are decided on the heuristic cost associated with the node.
- If B is the **branching factor**, then there will be only **W*B** nodes under consideration at any depth but only W nodes will be selected.

Algorithm – Beam search

- Found = false;
- NODE = Root_node;
- If NODE is the goal node, then *Found* = *true* else find SUCCs of NODE, if any with its estimated cost and store in OPEN list;
- While (Found = false AND not able to proceed further)

  {

  - Sort OPEN list;
  - Select top W elements from OPEN list and put it in W_OPEN list and empty OPEN list;

Algorithm – Contd…

- While (W_OPEN ≠ φ AND Found = false)

  {
  - Get NODE from W_OPEN;
  - If NODE = Goal state then Found = true else

    {
    - Find SUCCs of NODE, if any with its estimated cost
    - store in OPEN list;

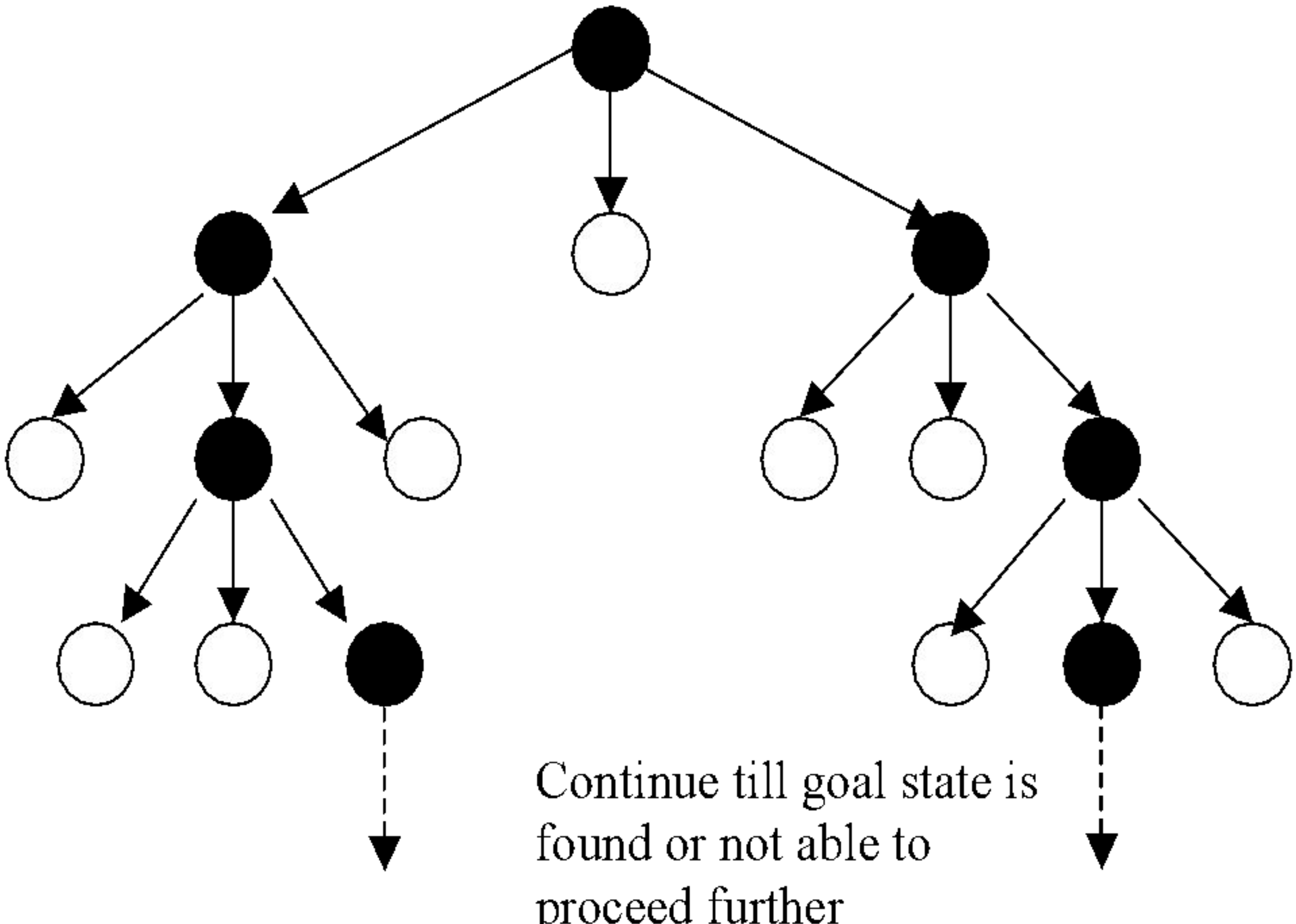
  }

  } // end while

} // end while

- If *Found* = *true* then return *Yes* otherwise return *No* and Stop

W = 2


Continue till goal state is found or not able to proceed further

Best First Search

- Expand the best partial path.
- Here forward motion is carried out from the best open node so far in the entire partially developed tree.

Algorithm (Best First Search)

- Initialize OPEN list by root node; CLOSED = φ;
- Found = false;
- While (OPEN ≠ φ AND Found = false) Do

  {
  - If the first element is the goal node, then **Found = true** else remove it from OPEN list and put it in CLOSED list.
  - Add its successor, if any, in OPEN list.
  - Sort the **entire list** by the value of some heuristic function that assigns to each node, the estimate to reach to the goal node

  } /* end while */
- If the **Found = true**, then announce the success else announce failure.
- Stop.

Observations

- In hill climbing, sorting is done on the successors nodes whereas in the best first search sorting is done on the entire list.
- It is not guaranteed to find an optimal solution, but normally it finds some solution faster than any other methods.
- The performance varies directly with the accuracy of the heuristic evaluation function.

Termination Condition

- Instead of terminating when a path is found, terminate when the shortest incomplete path is longer than the shortest complete path.

A* Method

- A* ("Aystar") (Hart, 1972) method is a combination of **branch & bound** and **best search**, combined with the **dynamic programming principle.**
- The heuristic function (or Evaluation function) for a node N is defined as **f(N) = g(N) + h(N)**
- The function **g** is a measure of the cost of getting from the **Start** node (initial state) to the **current** node.
  - It is sum of costs of applying the rules that were applied along the best path to the current node.
- The function **h** is an estimate of additional cost of getting from the **current** node to the **Goal** node (final state).
  - Here knowledge about the problem domain is exploited.
- A* algorithm is called OR graph / tree search algorithm.

Algorithm (A*)

- Initialization OPEN list with initial node; CLOSED= φ; g = 0, f = h, **Found = false**;
- While (OPEN ≠ φ and **Found = false**)

  {1

  - Remove the node with the lowest value of **f** from OPEN to CLOSED and call it as a **Best_Node**.
  - If Best_Node = Goal state then **Found = true** else

    {2

    - Generate the **Succ** of **Best_Node**
    - For each **Succ** do

      {3

      - Compute g(Succ) = g(Best_Node) + cost of getting from Best_Node to Succ.

A* - Contd..

- If Succ $\in$ OPEN then /* already being generated but not processed */

{[4]

  - Call the matched node as OLD and add it in the list of Best_Node successors.
  - Ignore the **Succ** node and change the parent of OLD, if required.

    - If g(Succ) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD
    - If g(Succ) >= g(OLD) then ignore

}[4]

A* - Contd..

- If Succ $\in$ CLOSED then /* already processed */

{5

  - Call the matched node as OLD and add it in the list of Best_Node successors.
  - Ignore the **Succ** node and change the parent of OLD, if required

    - If g(Succ) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD.

    - Propogate the change to OLD's children using depth first search

    - If g(Succ) >= g(OLD) then do nothing

}5

A* - Contd..

- If Succ $\notin$ OPEN or CLOSED
  {6
  - Add it to the list of Best_Node's successors
  - Compute f(Succ) = g(Succ) + h(Succ)
  - Put **Succ** on OPEN list with its f value

  }6

}3 /* for loop*/

}2 /* else if */

}1 /* End while */

- If **Found = true** then report the best path else report failure
- Stop

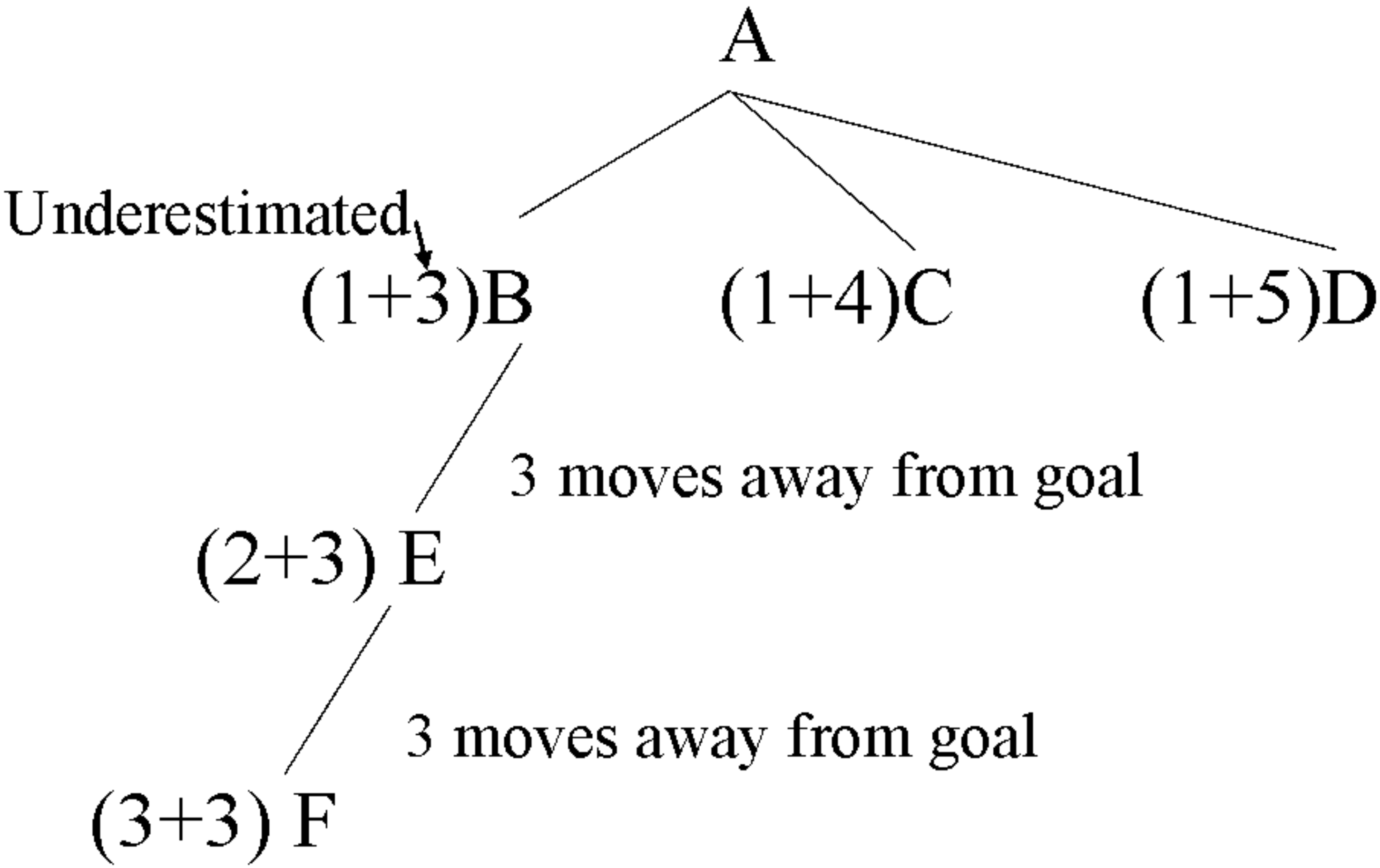
Behavior of A* Algorithm

Underestimation

- If we can guarantee that **h** never over estimates actual value from current to goal, then A* algorithm is guaranteed to find an optimal path to a goal, if one exists.

Example – Underestimation – f=g+h

Here h is underestimated

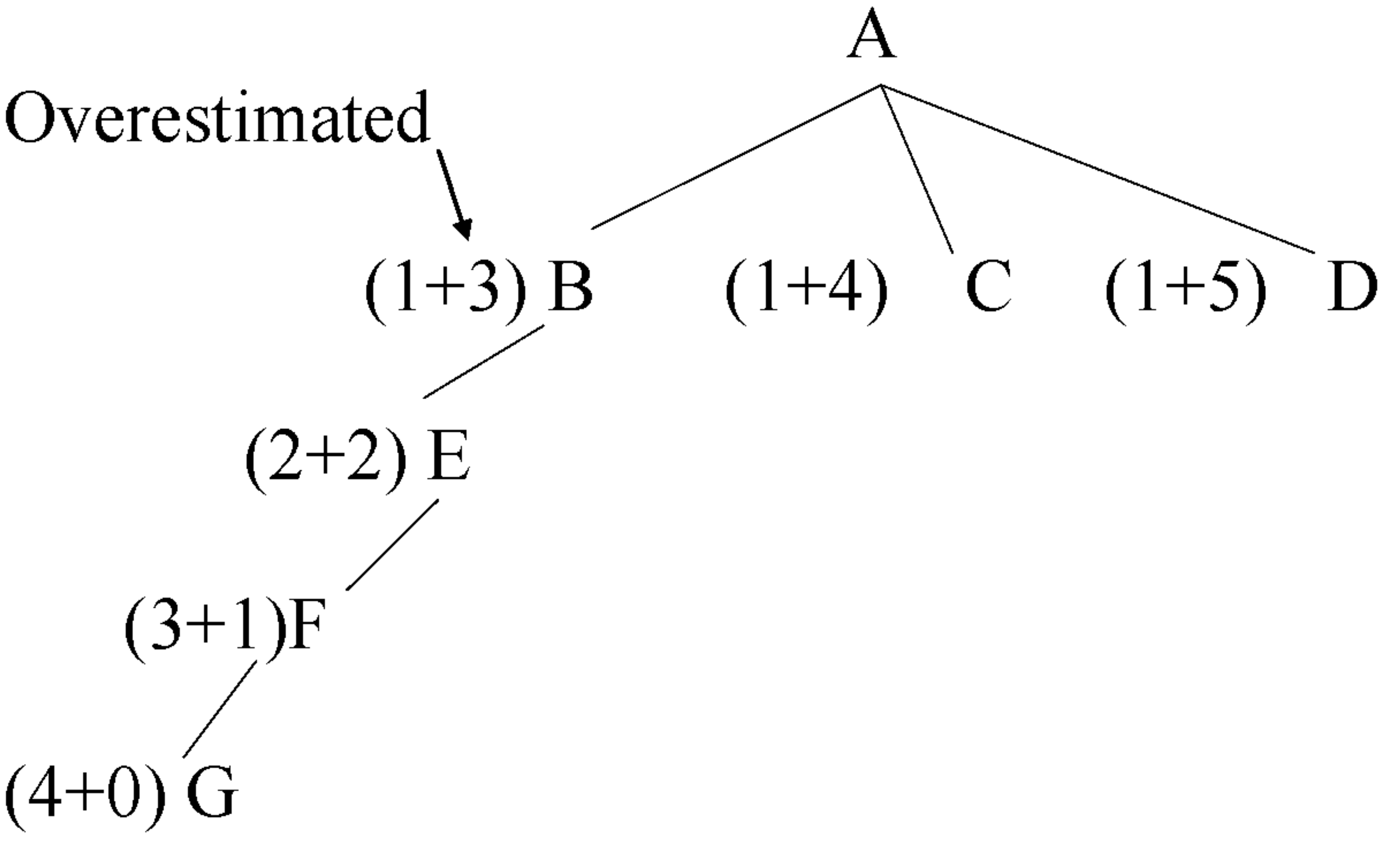


Explanation -Example of Underestimation

- Assume the cost of all arcs to be 1. A is expanded to B, C and D.
- ‘f’ values for each node is computed.
- B is chosen to be expanded to E.
- We notice that f(E) = f(C) = 5
- Suppose we resolve in favor of E, the path currently we are expanding. E is expanded to F.
- Clearly expansion of a node F is stopped as f(F)=6 and so we will now expand C.
- Thus we see that by underestimating h(B), we have wasted some effort but eventually discovered that B was farther away than we thought.
- Then we go back and try another path, and will find optimal path.

Example – Overestimation

Here h is overestimated



Explanation –Example of Overestimation

- A is expanded to B, C and D.
- Now B is expanded to E, E to F and F to G for a solution path of length 4.
- Consider a scenario when there a direct path from D to G with a solution giving a path of length 2.
- We will never find it because of overestimating h(D).
- Thus, we may find some other worse solution without ever expanding D.
- So by overestimating h, we can not be guaranteed to find the cheaper path solution.

Admissibility of A*

- A search algorithm is **admissible,** if
  - for any graph, it always terminates in an optimal path from initial state to goal state, if path exists.
- If heuristic function **h** is **underestimate** of actual value from current state to goal state, then the it is called **admissible function**.
- Alternatively we can say that A* always terminates with the optimal path in case
  - h(x) is an **admissible heuristic function**.

Monotonicity

- A heuristic function **h** is monotone if
  - $\forall$ states Xi and Xj such that Xj is successor of Xi

    $h(Xi) - h(Xj) \leq cost\ (Xi, Xj)$

    where, cost (Xi, Xj) actual cost of going from Xi to Xj
  - h (goal) = 0

- In this case, heuristic is locally admissible i.e., consistently finds the minimal path to each state they encounter in the search.

Contd..

- Alternatively, the monotone property:
  - that search space which is every where locally consistent with heuristic function employed i.e., reaching each state along the shortest path from its ancestors.
- With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter.
- Each monotonic heuristic is admissible
  - A **cost function** $f(n)$ is monotone. if $f(n) \leq f(succ(n))$, $\forall n$.
- For any admissible cost function f, we can construct a monotone admissible function.

Example: Solve Eight puzzle problem using A* algorithm

Start state

3	**7**	**6**
5	**1**	**2**
4	□	**8**

Goal state

5	**3**	**6**
7	□	**2**
4	**1**	**8**

- Evaluation function f (X) = g (X) + h(X)

  h (X) = the number of tiles not in their goal position in a given state X

  g(X) = depth of node X in the search tree
- Initial node has **f(initial_node)** = 4
- Apply A* algorithm to solve it.
- The choice of evaluation function critically determines search results.

Example: **Eight puzzle problem (EPP)**

Start state

3	7	6
5	1	2
4	□	8

Goal state

5	3	6
7	□	2
4	1	8

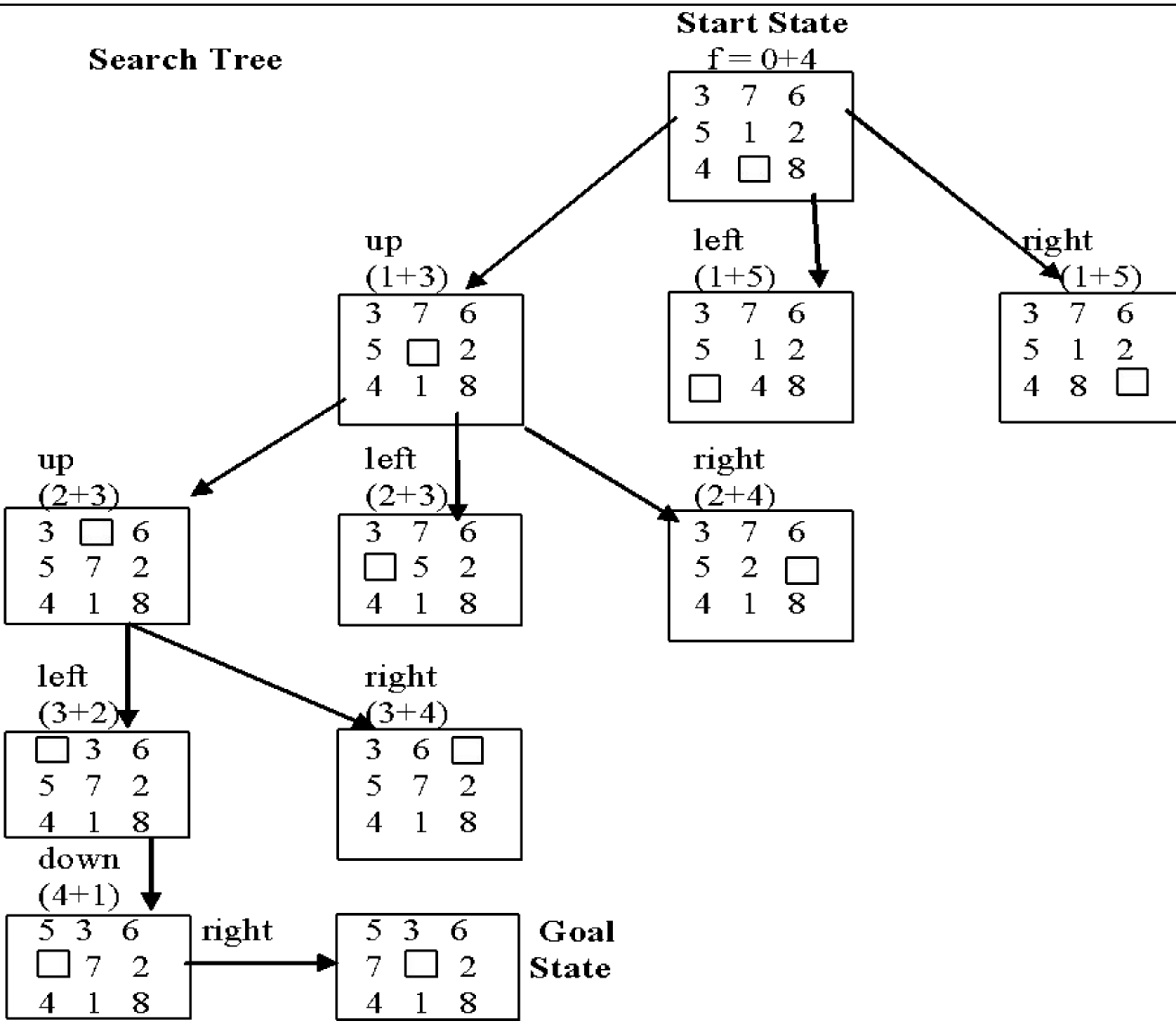
Evaluation function - f for EPP

- The choice of evaluation function critically determines search results.
- Consider Evaluation function

  $f(X) = g(X) + h(X)$

  - h (X) = the number of tiles not in their goal position in a given state X
  - g(X) = depth of node X in the search tree
- For Initial node
  - **f(initial_node)** = 4
- Apply A* algorithm to solve it.

Search Tree


Start State
f = 0+4
3 7 6
5 1 2
4 8
up
(1+3)
3 7 6
5 2
4 1 8
left
(1+5)
3 7 6
5 1 2
4 8
right
(1+5)
3 7 6
5 1 2
4 8
up
(2+3)
3 6
5 7 2
4 1 8
left
(2+3)
3 7 6
5 2
4 1 8
right
(2+4)
3 7 6
5 2
4 1 8
left
(3+2)
3 6
5 7 2
4 1 8
right
(3+4)
3 6
5 7 2
4 1 8
down
(4+1)
5 3 6
7 2
4 1 8
right
5 3 6
7 2
4 1 8
Goal State

Harder Problem

- Harder problems (8 puzzle) can't be solved by heuristic function defined earlier.

Initial State

2	1	6
4	□	8
7	5	3

Goal State

1	2	3		
8			4	□
7	6	5		

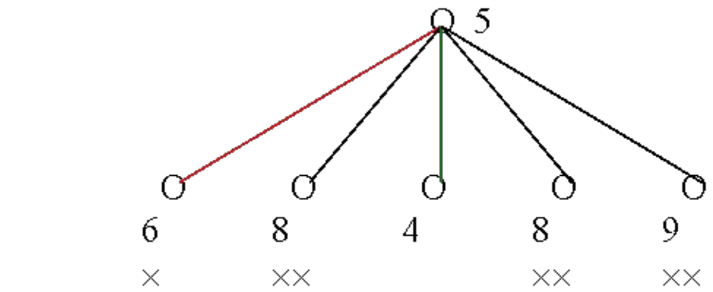
- A better estimate function is to be thought.

  h (X) = the sum of the distances of the tiles from their goal position in a given state X
- Initial node has **h(initial_node)** = 1+1+2+2+1+3+0+2=12

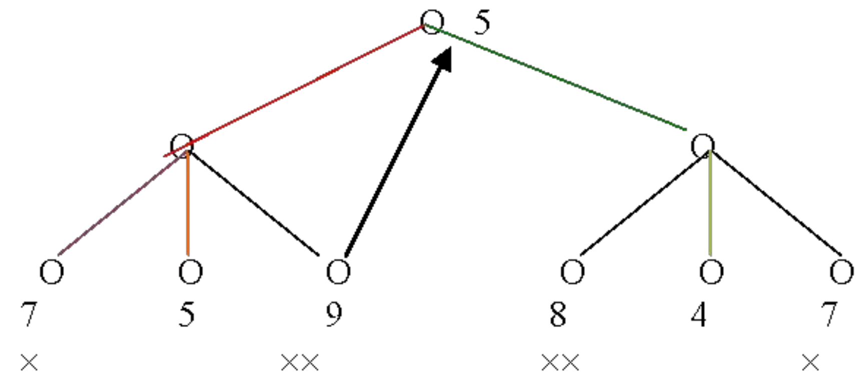
IDA* Algorithm

- At each iteration, perform a DFS cutting off a branch when its total cost (g+h) exceeds a given threshold.
- This threshold starts at the estimate of the cost of the initial state, and increases for each iteration of the algorithm.
- At each iteration, the threshold used for the next iteration is the minimum cost of all values exceeded the current threshold.

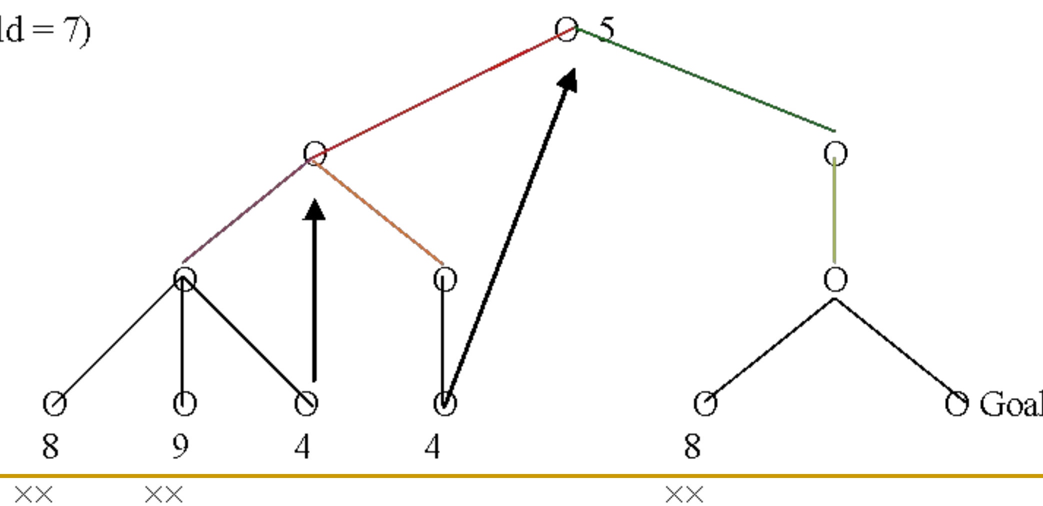
Ist iteration (Threshhold = 5)



IInd iteration (Threshhold = 6)



IIIrd iteration (Threshhold = 7)



Contd..

- Given an admissible monotone cost function, IDA* will find a solution of least cost or optimal solution if one exists.
- IDA* not only finds cheapest path to a solution but uses far less space than A* and it expands approximately the same number of nodes as A* in a tree search.
- An additional benefit of IDA* over A* is that it is simpler to implement, as there are no open and closed lists to be maintained.
- A simple recursion performs DFS inside an outer loop to handle iterations.