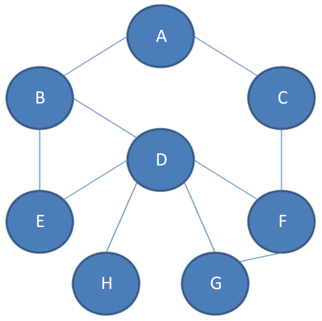
Searching Algorithms

Er. Rudra Nepal

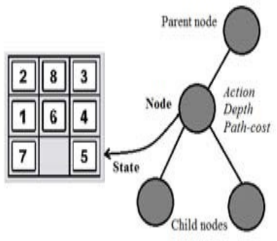
Searching

- Step in Problem Solving
- Searching is Performed through the State Space
- Searching accomplished by constructing a search tree
- The root of the search tree is search node



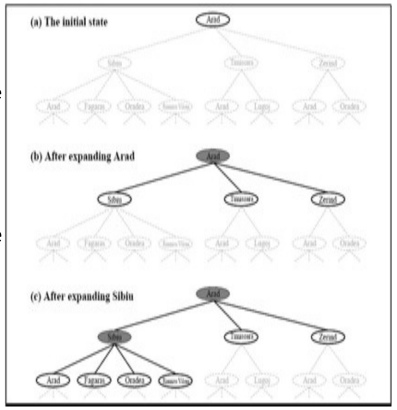
Searching

- There are many ways to represent nodes, but we will assume that a node is a data structure with five components:
  - State:the state in the state space to which the node corresponds
  - Parent-Node: the node in the search tree that generated this node.
  - Action:the action that was applied to the parent to generate the node.
  - Path-cost:the cost of the path from initial state to the node
  - Depth:the number of steps along the path from initial state.



Searching: Steps

- Check whether the current state is the goal state or not
- Expand the current state to generate the new sets of states
  - The collection of nodes that have been generated but not yet expanded is called fringe.
  - Each element of fringe is a leaf node-a node with no successor in the tree.
- Choose one of the new states generated for search which entire depend on the selected search strategy
- Repeat the above steps until the goal state is reached or there are no more states to be expanded



Searching: Criteria to Measure Performance

- The output of a search algorithm is either failure or a solution.
- Some algorithm get stuck in an infinite loop and never return an output.
- We can evaluate the algorithm's performance in four ways:
  - Completeness:Is the algorithm guaranteed to find a solution when there is one?
  - Optimality:Does the strategy find the optimal solution?
  - Time Complexity: How long does it take to find a solution?
  - Space Complexity: How much memory is needed to perform the search?
- We can express the algorithm's complexity in terms of three quantities:
  - b, the branching factor or maximum number of successor of any node
  - d, the depthof the shallowest goal node
  - m, the maximum length of any path in the state space

Searching: Types

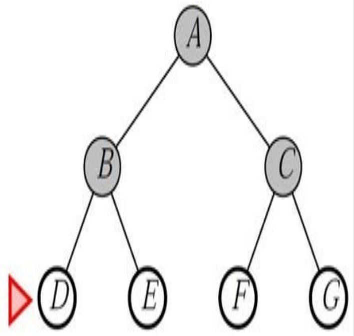
- Uninformed Search or Blind Search
- Informed Search or Heuristic Search

Uninformed Search

- Search provided with problem definition only and no additional information about the state space
- Expansion of current state to new set of states is possible
- It can only distinguish between goal state and non-goal state
- Less effective compared to Informed search
- The uninformed search strategies are distinguished by the order in which nodes are expanded
- Various uninformed search techniques/strategies are:
  - Breadth-first Search
  - Uniform-cost Search
  - Depth-first Search
  - Depth-limited Search
  - Iterative deepening depth-first search

Breadth-first Search

- Root node is expanded first
- Then all the successors of the root node are expanded
- Then their successors are expanded and so on.
- Nodes, which are visited first will be expanded first (FIFO)
- All the nodes of depth 'd' are expanded before expanding any node of depth 'd+1'



Breadth First Search: Four Criteria

- Completeness
  - This search strategy finds the shallowest goal first
  - Complete, if the shallowest goal is at some finite depth
- Optimality
  - The shallowest goal node is not necessarily the optimal one
  - Optimal, if the path cost is a non-decreasing function of the path of the node (For example: when all the actions have the same cost)

Breadth First Search: Four Criteria

Time Complexity

- For a search tree a branching factor 'b'expanding the root yields 'b'nodes at the first level.
- Expanding 'b'nodes at first level yields b2nodes at the second level.
- Similarly, expanding the nodes at dthlevel yields bd+1 node at (d+1)thlevel
- If the goal is in dthlevel, in the worst case, the goal node would be the last node in the dth level

Hence, We should expand 1) nodes in the (d+1) level (Except the goal node itself which doesn't need to be expanded)

- So, number of nodes generated at (d+1)thlevel = b(bd-1) =bd+1-b
- Again, Total number of nodes generated = 1+b+b2+...+bd+1-b =O(bd+1)
- Hence, time complexity is O(bd+1)

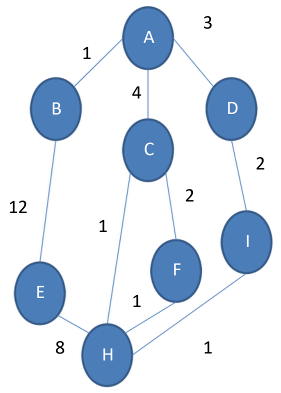
where, b= branching factor and d=level of goal node in the search table

Breadth First Search: Four Criteria

- Space Complexity
  - Same as time complexity
  - i.e. O(bd+1)
  - Since each node has to be kept in the memory
- Disadvantages
  - Memory Wastage
  - Irrelevant Operations
  - Time Intensive It
  - doesn't assure the optimal cost solution

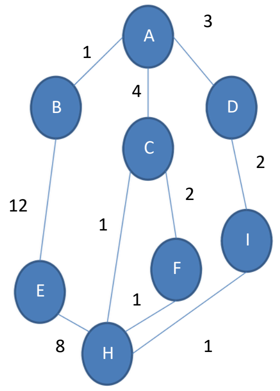
Uniform Cost Search

- It expands the lowest cost node on the fringe
- The first solution is guaranteed to be the cheapest one because a cheaper one would have expanded earlier and so would have been found first
- If all step costs are equal, this is identical to breadth first search
- Required Condition: A to H
  - ABEH=21, ACH=5, ACFH=7, ADIH=6



Uniform Cost Search

- Solution: Required Operation
  Expand A⍰Yield B, C, D
  With AB=1, AC=4, AD=3
  Expand B⍰Yield E with
  ABE=13
  As ABE>AC and ABE>AD
  –Expand D⍰Yield I with
  ADI=5
  As ADI>AC
  –Expand C⍰Yield H and F
  with ACH=5 and ACF=6
  –Solution Achieved
- If all step costs are equal, it is identical breadth first search



Uniform Cost Search

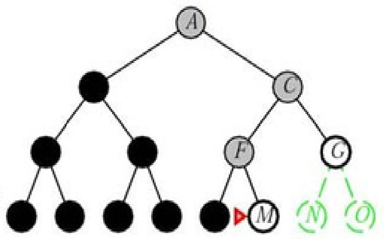
- Disadvantages
  - Doesn’t care about the number of steps a path has but only about their cost
  - It might get stuck in an infinite loop if it expands a node that has a zero cost action leading back to same state

Uniform Cost Search: Four Criteria

- Completeness
  - Complete, if the cost of every step is greater than or equal to some small positive constant ϵ
- Optimality
  - The same ensures optimality
- Time Complexity
  - O(b C*/ϵ)
  - Where C*⍰cost of optimal path and ϵ ⍰small positive constant
  - –This complexity is much greater than that of Breadth first search
- Space Complexity
  - –O(b C*/ϵ)

Depth-first Search

- Always expands the deepest node in the current fringe of the search tree
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors (dead end)
- When a dead end is reached, the search backup to the next shallowest node that still has unexplored successors
- This strategy can be implemented by tree search with a last-in-first-out (LIFO) queue, also known as stack.



Depth First Search: Four Criteria

- Completeness
  Can get stuck going down the wrong path when a different choice would lead to a solution near the root of the search tree
  –Not complete
- Optimality
  – The strategy might return a solution path that is longer than the optimal solution, if it starts with an unlucky path
  – Not optimal

Depth First Search: Four Criteria

- Space Complexity
  – It needs to store a single path from root to a leaf node and the remaining unexpanded sibling nodes for each node in the path
  – Once a node has been expanded, it can be removed from memory as soon as all its decedents have been fully explored
  – For a search tree of branching factor 'b' and maximum tree depth 'm', only the storage of *bm+1*node is required
  – Hence,
    Space Complexity = O(bm+1)
    = O(bm)
- Time Complexity
  – in the worst case all the bmnodes of the search tree would be generated
  – Hence,
    Time Complexity= O(bm)

Depth-limited Search

- Modification of depth first search
- Depth first search with predetermined limit *'l'*
- After the nodes at the level *'l'*are explored, the search backtracks without going further deep
- Hence, it solves the infinite path problem of the depth first search strategy
- Completeness:Complete except at additional source of incompleteness if *l<d,*i.esallowest goal is beyond the depth limit
- Optimality:Optimal except at *l>d*
- Time Complexity=*O(bl)*
- Space Complexity= *O(bl)*

Iterative deepening depth first Search

- Finds the best limit by gradually increasing depth limit first to 0, then to 1, 2 and so on-until the goal is found
- Combines the benefits of the depth first and breadth first search
- In depth-limited search, the complex part is to choose good depth limit
- This strategy addresses the issue of good depth limit by trying all possible depth limits
- The process is repeated until goal is found at depth limit *'d'* which is the depth of shallowest goal
- Completeness:as of Breadth First Search i.e. Complete if branching factor is finite
- Optimality: as of Breadth First Search i.e. optimal if the path cost is non decreasing function of depth
- Space Complexity = *O(bd)*
- Time Complexity = *O(bd)*

Iterative deepening depth first Search

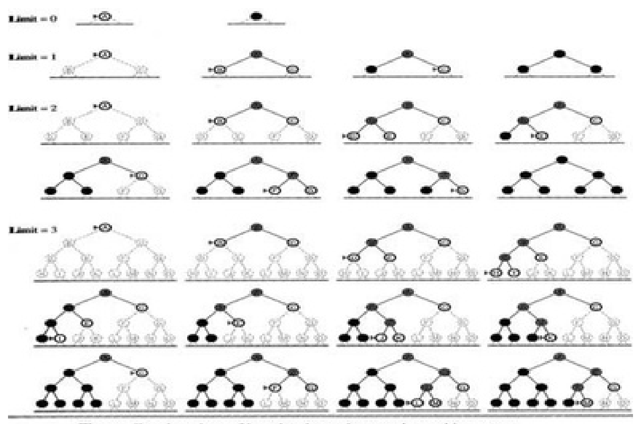


Figure: Four iterations of iterative deepening search on a binary tree.

Comparing uninformed search strategies

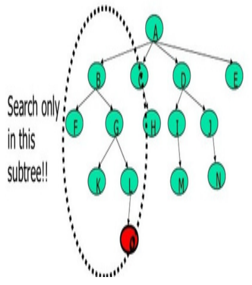
Criterion	Breadth-first Search	Uniform-cost Search	Depth-first Search	Depth-limited Search	Iterative deepening Search
Completeness	Yes, *if b is finite*	Yes, if step cost ≥ *ϵfor positive ϵ*	No	No	Yes, *if b is finite*
Optimality	Yes, *if step cost are identical*	Yes	No	No	Yes, *if step cost are identical*
Time Complexity	*O(bd+1)*	*O(bC*/ϵ)*	*O(bm)*	*O(bl)*	*O(bd)*
Space Complexity	*O(bd+1)*	*O(bC*/ϵ)*	*O(bm)*	*O(bl)*	*O(bd)*

*b*is branching factor

*d*is the depth of the shallowest solution
*m*is the maximum depth of search tree
*l*is the depth limit

Informed (Heuristic) Search Strategies

- Strategy of problem solving where problem specific knowledge is known along with problem definition
- These search find solutions more efficiently by the use of heuristics
- Heuristic is a search technique that improves the efficiency of the search process
- By eliminating the unpromising states and their descendants from consideration, heuristic algorithms can find acceptable solutions



Informed (Heuristic) Search Strategies

- Heuristics are fallible i.e. they are likely to make mistakes as well
- It is the approach following an informed guess of next step to be taken
- It is often based on experience or intuition
- Heuristic have limited information and hence can lead to suboptimal solution or even fail to find any solution at all

Best First Search

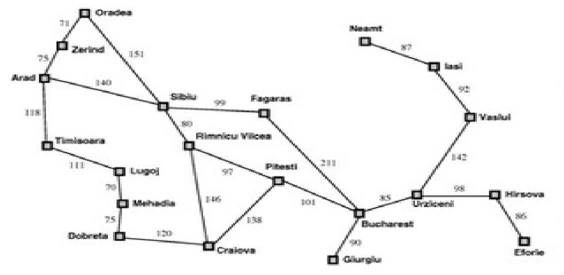
- A node is selected for expansion based on evaluation function *f(n)*
- A node with lowest evaluation function is expanded first
- The measure i.e. evaluation function must incorporate some estimate of the cost of the path from a state to the closest goal state
- The algorithm may have different evaluation function, one of such important function is the heuristic function *h(n)*
  - *h(n)*= the estimated cost of the cheapest path from node *n* to the goal
  - *h(n)*= 0 , if n is goal node
- Types of best first search
  - Greedy Best First Search
  - A* Search (pronounced “A-star search”)

Greedy Best First Search

- The node whose state is judged to be the closest to the goal state is expanded first
- At each step it tries to be as close to the goal as it can
- It evaluates the nodes by using heuristic function hence, *f(n)=h(n)* where, *h(n)=0*, for the goal node
- This search resembles depth first search in the way that it prefers to follow a single path all the way to the goal or if not found till the dead end and returns back up

Greedy Best First Search: Example

- Let us see how this works for route-finding problem, using the straight-line distanceheuristic
- If goal is Bucharest, we will need to know the straight-line distance to Bucharest as shown in figure

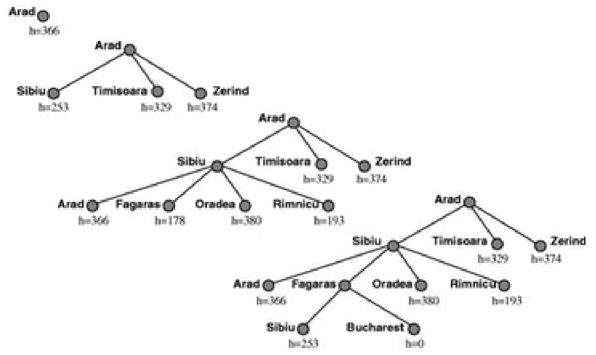


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best First Search: Example

- Figure below shows the progress of greedy best first search using straight-line distance to find the path from Arad to Bucharest



Greedy Best First Search:FourCriteria

- Completeness
  - Can start down an infinite path and never return to any possibilities
  - Not complete
- Optimality
  - Looks for immediate best choice and doesn’t make careful analysis of long term options
  - May give longer solution even if shorter solution exists
  - Not optimal
- Space Complexity
  - O(bm) where, *m* is the maximum depth of search space, since all nodes have to be kept in memory
- Time Complexity
  - O(bm)

A* Search

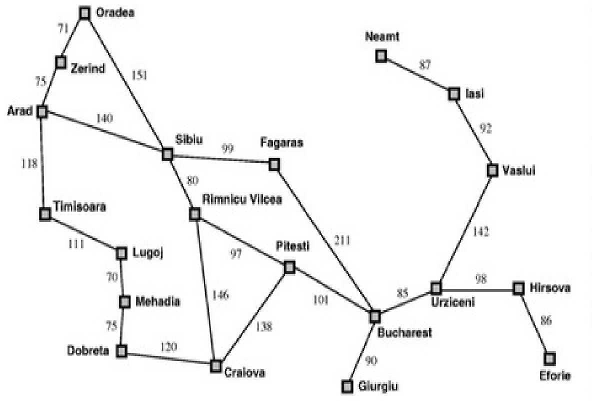
- It evaluates nodes by combining *g(n)*, the cost to reach the node, and *h(n)*, the cost to get from node to the goal

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

  Since *g(n)* gives the path cost from the start node to node n, and *h(n)* is the estimated cost of the cheapest path from n to goal node, we have

  f(n)= estimated cost of the cheapest solution through *n*
- Admissible Heuristic: *h(n)* is admissible if it never overestimates the cost to reach the solution
  - example: h_{SLD}(straight line distance)
  - SLD is admissible because the shortest path between any two point is straight line

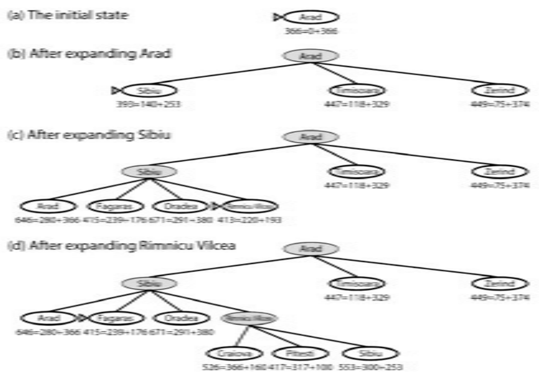
A* Search: Example



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search: Example



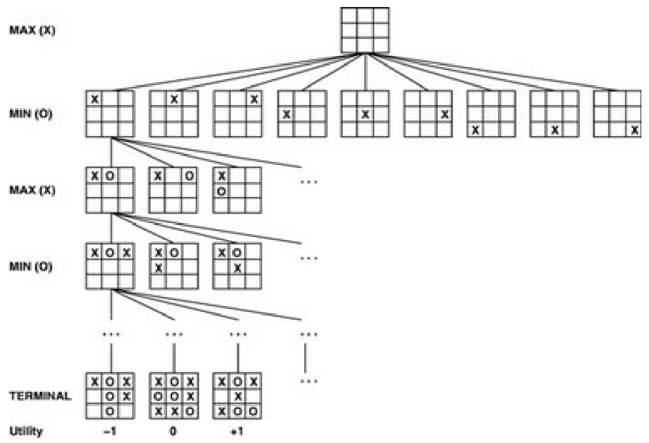
A* Search: Four Criteria

- Optimality
  - Optimal if *h(n)* is admissible
- Completeness
  - Complete if *h(n)* is admissible
- Space Complexity
  - **$O(b)^d$** if *h(n)* is admissible
- Time Complexity
  - **$O(b)^d$** if *h(n)* is admissible

Adversarial Search Techniques

- A game can be defined as a kind of search problem (game tree) with the following components:
  - Initial State identifying the initial position in the game and identification of the first player
  - Actionsreturns the set of legal moves in a state
  - Successor Function returning a list of (move, state) pairs
  - Terminal Test which is true if game is over and false otherwise. States where the game has ended are called terminal states
  - Utility function which gives a numeric value for the terminal states. Example: in TTT Lose, draw and win with -1, 0 and +1

Game Tree Example: Tic-Tac-Toe



Adversarial Search Techniques

- Minimax Algorithm
- Alpha-Beta Purning

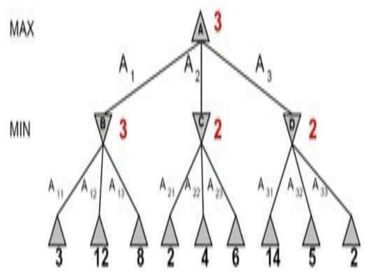
Minimax Algorithm

- Max is considered as the first player in the game and Min as the second player
- This algorithm computes the minimaxdecision from the current state
- At each MAX node, pick the move with maximum utility.
- At each MIN node, pick the move with minimum utility
- It uses a recursive computation of minimaxvalues (minimaxvalue of a node is the utility of being in the corresponding state)of each successor state directly implementing some defined function
- The recursion proceeds from the initial node to all the leaf nodes
- Then the minimaxvalues are backed up through the tree as the recursion unwinds
- It performs the depth first exploration of a game tree in a complete way
- If the maximum depth of ▯ the tree is m and there are b legal moves at each point then for minimaxalgorithm:
  - Time Complexity=
  - Space Complexity=

(()

MinimaxAlgorithm: Computation

- In the figure, the algorithm first recursesdown to the three bottom-leaf nodes and uses Utility function to discover that their values are 3, 12 and 8 respectively.
- Then it takes the minimum of these values, 3, and return it as backed-up value of node B.
- A similar process gives backed-up value of 2 for C and 2 for D.
- Finally, we take maximum of 3, 2, 2 to get backed-up value of 3 for the root node A.



Alpha-Beta Pruning

- The main disadvantage of the minimaxalgorithm is that all the nodes in the game tree cutoff to a certain depth are examined.

- Alpha-beta pruning helps reduce the number of nodes explored.
- When applied to a standard minimaxtree, alpha beta pruning returns the same move as minimax would, but prunes away the branches which couldn’t possibly influence the final decision.

Alpha-Beta Pruning

- In the figure alongside, if *m* is better than *n* for Player, we will never get *n* in play
- Alpha-Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values:
  - α = the value of the best (i.e highest value) choice we have found so far at any choice point along the path for MAX
  - β = the value of the best (i.e lowest value) choice that we have found so far at any choice point along the path for MIN

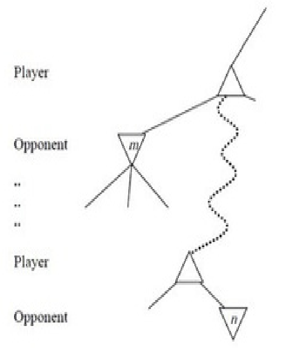
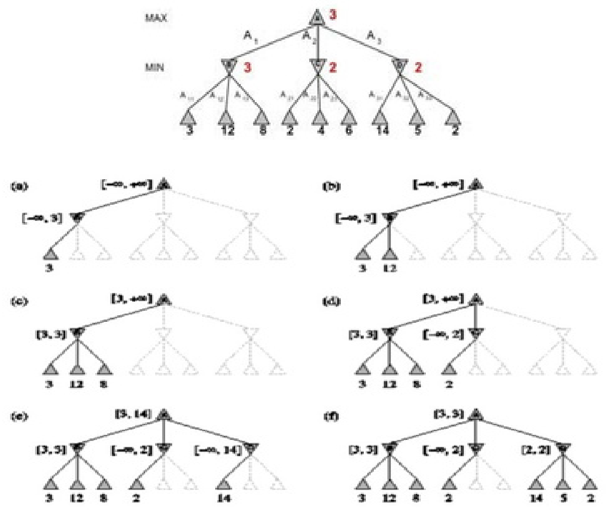


Figure: General case for alpha-beta pruning

Alpha-Beta Pruning: Example


MAX
MIN
3
2
2
3 12 8 2 4 6 14 5 2
(a) [−∞, +∞] [−∞, 3]
(b) [−∞, +∞] [−∞, 3]
(c) [3, +∞] [3, 3]
(d) [3, +∞] [3, 3] [−∞, 2]
(e) [3, 14] [3, 3] [−∞, 2] [−∞, 14]
(f) [3, 3] [3, 3] [−∞, 2] [2, 2]

Alpha-Beta Pruning: Example

- *a:*The first leaf below B has the value 3. Hence, B, which is a MIN node has a value of at most 3.
- *b:*The second leaf below B has a value 12; MIN would avoid this move, so the value of B is still at most 3.
- *c:*The third leaf below B has a value of 8; we have seen all B’s successor states, so the value of B is exactly 3. Now we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the node
- *d:*The first leaf below C has the value 2. hence C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. therefore, there is no point in looking at the other successor states of C. This is an example of Alpha-Beta Pruning.
- *e*: The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX’s best alternative (i.e3), so we need to keep exploring D’s successors states.
- *f:* The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX’s decision at the root is to move to B, giving a value of 3.