Game Engineering

CS420-2011F-12

Artificial Intelligence

David Galles

Department of Computer Science
University of San Francisco

12-0: Artifical Intelligence

- AI in games is a huge field
  - Creating a believable world
    - Characters with their own appearnt goals and desires, especially in RPGs and open world games
    - Opponents that seem to think and plan
  - Simulating human players
    - Chess players, FPS “bots”, strategy game opponents, etc

12-1: Most AI is Faked ...

- ... which in unsurprising, since most *everything* is faked, if possible
- Don’t need to have intelligent enemies, just need to *appear* intelligent
- Surprisingly large quantity is done with Finite State Machines

12-2: Finite state machines

- Each entity has a number of states, that represent behaviors
  - Patrolling, advancing to a position, searching, running away, finding cover, etc
- Each behavior can be relatively simple
- Transitions between behaviors can be triggered by timers, scripting, “sensing” by entities, etc

12-3: Case Study: Stealth shooter

- Creating a stealth-based action game (Thief, Splinter Cell, Metal Gear Solid, etc)
  - Patrol state (traversing between waypoints)
  - Alerted state (simple search pattern)
  - Attacking state (advance towards player, attack)
- Each behavior is relatively simple, well-managed transitions between them (especially scripted transitions) can lead to very intelligent-seeming enemies. Add in some random audio cues, and the enemies can seem quite smart ...

12-4: Pathfinding

- One aspect of tradional AI that is commonly used in games is pathfinding
  - RTS units getting from home base to place they are attacking
  - Enemies attacking player in a maze-style game
  - Bots finding shortest route to powerups / other players / etc in FPSs
- First step: Simplifiying the problem

12-5: Pathfinding

- Navigating a real-life (or even complex simulated) enviornment is tricky
- Vastly simplify the search space, make it a standard CS-style graph
  - Waypoint System
  - Navigation Mesh
- 2D games (RTS, etc), can be easier – just use a grid

12-6: Pathfinding

- OK, so we've simplified the problem to searching for a path in a (potentially very complicated) graph
  - Verticies (places AI can go)
  - Edges (links between verticies, cost – often just a distance, can be mor complicated)
- How do we efficiently search the graph?

12-7: Breadth-First Search

- Examine all nodes that are 1 unit away
- Examine all nodes that are 2 units away
- . . .
- Examine all nodes that are n units away

(Examples)

12-8: Breadth-First Search

- A few more wrinkes:
  - Searching a graph instead of a tree
  - Get to the same node in more than one way
  - Once we’ve found shortest path to a path to a node, don’t need to consider any other paths

12-9: Breadth-First Search

- Maintain two data structures
  - “Open List” – search horizon
  - “Closed list” – nodes we’ve already found the shortest path to, don’t need to examine again

12-10: Breadth-First Search

```
void BFS(Graph G, Vertex v) {

  Queue Q = new Queue();
  Closed = new ClosedList();

  Q.enquque(v);
  while (!Q.empty()) {
    nextV = Q.dequeue()
    if (v not in Closed)
     {
          Closed.Add(v);
          forach (Vertex neighbor adjacent to v in G)
              Q.enqueue(neighbor);
      }
    }
  }
}
```

12-11: Breadth-First Search

- Problem #1 with BFS:
  - Assumes uniform edge cost
  - Not actually true with most graphs we will be searching
- Solution?

12-12: Best-first Search

- Uniform-cost search
  - Store node *and cost to get to node* in queue
  - Use a priority queue instead of a standard queue
  - Always choose the cheapeast node to expand
    - “Expand” means examine children of node

12-13: Uniform-Cost Search

- Uniform-Cost Pseudocode

```
enqueue(initialState)
do
  node = prioroty-dequeue()
  if (node not in closed list)
     add node to closed list
     if goalTest(node)
        return node (potenially path as well)
     else
        children = successors(node)
        for child in children
           prioroty-enqueue(child, dist(child))
```

- $dist$ is the cost of the path from the initial state to the child node

(EXAMPLES!)

Uniform-Cost Search

- Problem with Uniform cost search
  - To find a goal that is 100 units away from the start, we examine *all* nodes that are 100 units away from the start
  - RTS example on board
- Make a minor change to Uniform cost serach, make it much more general

12-15: Best-First Search

```
enqueue(initialState)
do
  node = prioroty-dequeue()
  if (node not in closed list)
     add node to closed list
     if goalTest(node)
        return node (potenially path as well)
     else
        children = successors(node)
        for child in children
           prioroty-enqueue(child, f(child))
```

- $f(n)$ is a function that describes how "good" a node is

12-16: Best-first Search

- (Almost) all searches are instances of best-first, with different evaluation functions f
- What functions f would yield the following searches:
  - Depth-First Search
  - Breadth-First Search
  - Uniform Cost Search

12-17: Best-first Search

- (Almost) all searches are instances of best-first, with different evaluation functions f
- What functions f would yield the following searches:
  - Breadth-First Search $f(n)$ = depth(n)
  - Depth-First Search $f(n)$ = -depth(n)
  - Uniform Cost Search $f(n) = g(n)$ (actual cost to get to n)

12-18: Heuristic Function

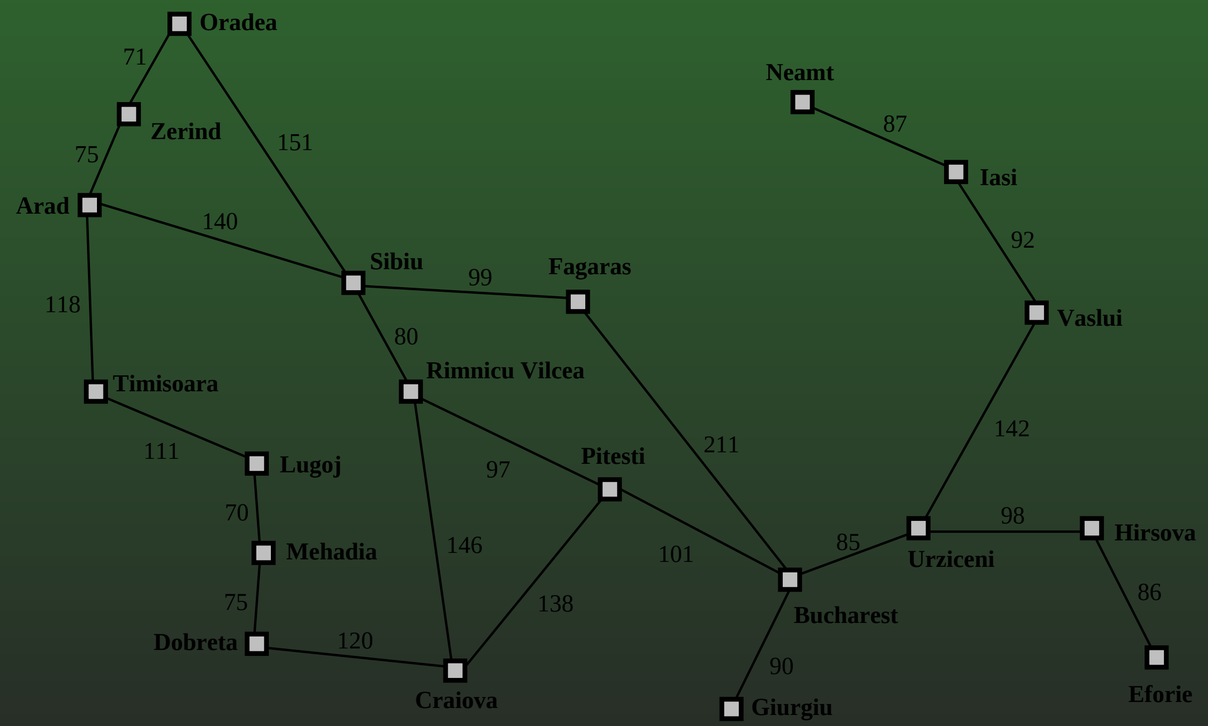
- A Heuristic Function $h(n)$ is an estimate of how much it would cost to get to the solution from node n
- $h(n)$ is not perfect
  - What could we do if h was perfect?
- Example heuristic: Route planning: straight-line distance to the goal
- How could we use a heuristic function as part of best-first search to find a goal quickly?

12-19: Greedy Search

- Best-First search with $f(n) = h(n)$
- Route-planning example: Always travel to the city that looks like it is closest to out destination

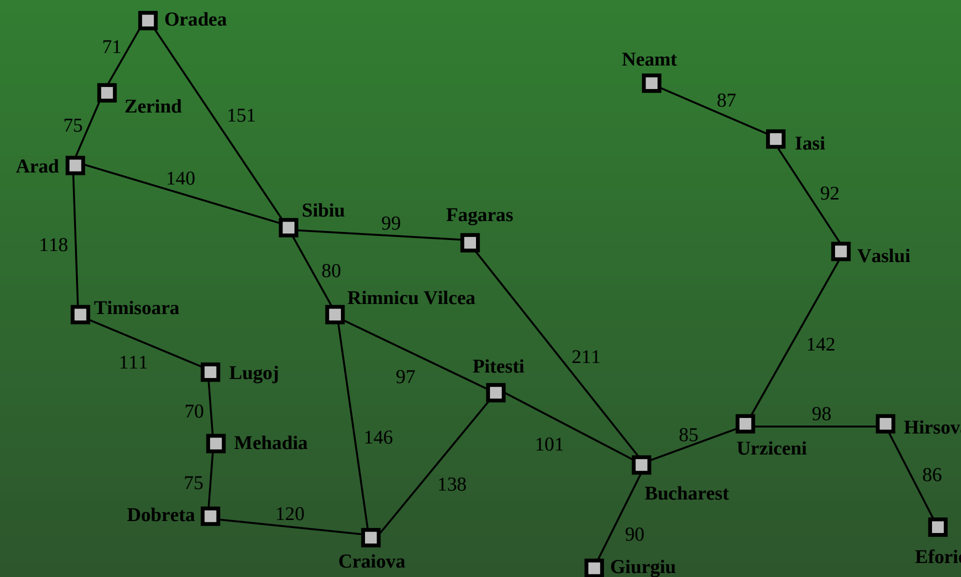
12-20: Greedy Search Example

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374



12-21: Greedy Search Example

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374



(A, 336)

(S,253), (T,329), (Z,374)

(F,176), (RV,193), (T,329), (A,336), (Z,374), (O,380)

(B,0), (RV,193), (S,253), (T,329), (A,336), (Z,374), (O,380)

Solution: A $\longrightarrow$ S $\longrightarrow$ F $\longrightarrow$ B

Optimal: A $\longrightarrow$ S $\longrightarrow$ RV $\longrightarrow$ P $\longrightarrow$ B

12-22: Greedy Search Problems

- Optimal solution can involve moving ’away’ from goal
  - Sliding tile puzzle: “undo” a partial solution
  - Rubic’s cube: “Mess up” part of cube to solve
- Not really moving away from goal – as a measure of the number of moves to a solution, you are actually getting closer to the goal. Only relative to your heuristic function are you going backwards
  - Perfect h == no need to search

12-23: Greedy Search Problems

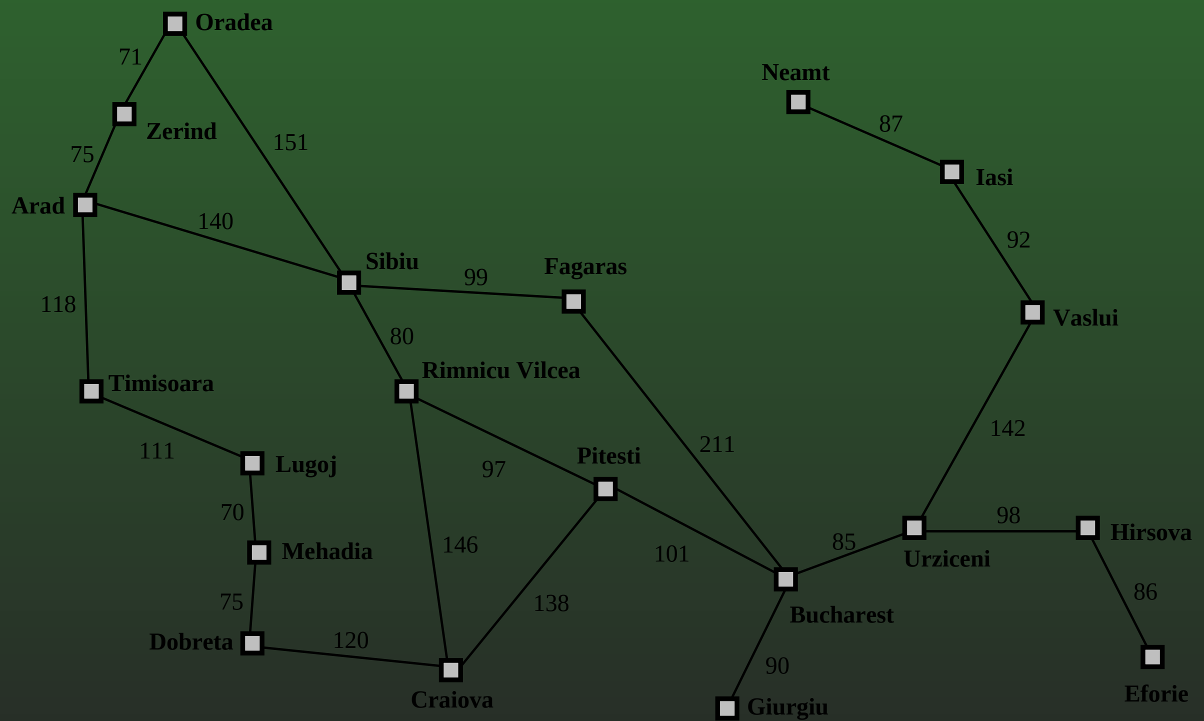
- Greedy search has similar strengths / weaknesses to DFS
  - Expands a linear number of nodes
  - Not optimal
  - May not even necessarily find goal (depending upon the heuristic function)
- What are the flaws of greedy search?
- How could we fix them?

12-24: A* search

- A* search is a combination of uniform cost search and greedy search.
- $f(n) = g(n) + h(n)$
  - $g(n)$ = current path cost
  - $h(n)$ = heuristic estimate of distance to goal.
- Favors nodes with best estimated total cost to goal
- If $h(n)$ satisfies certain conditions, A* is both complete (always finds a solution) and optimal (always finds the best solution).

12-25: A* Search Example

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374



12-26: A* Search Example

- Arad = 0 + 366 = 366
- (dequeue A: g = 0) S = 140 + 253 = 393, T = 118 + 329 = 447, Z = 75 + 374 = 449
- (dequeue S: g = 140) RV = 220 + 193 = 413, F = 239 + 176 = 415, T = 118 + 329 = 447, Z = 374 + 75 = 449, A = 280 + 336 = 616, O = 291 + 380 = 671,
- (dequeue RV: g = 220) F = 239 + 176 = 415, P = 317 + 100 = 417, T = 118 + 329 = 447, Z = 374 + 75 = 449, C = 366 + 160 = 526, S = 300 + 253 = 553, A = 280 + 336 = 616, O = 291 + 380 = 671
- (dequeue F: g = 239) P = 317 + 100 = 417, T = 118 + 329 = 447, Z = 374 + 75 = 449, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, S = 338 + 253 = 591,A = 280 + 336 = 616, O = 291 + 380 = 671

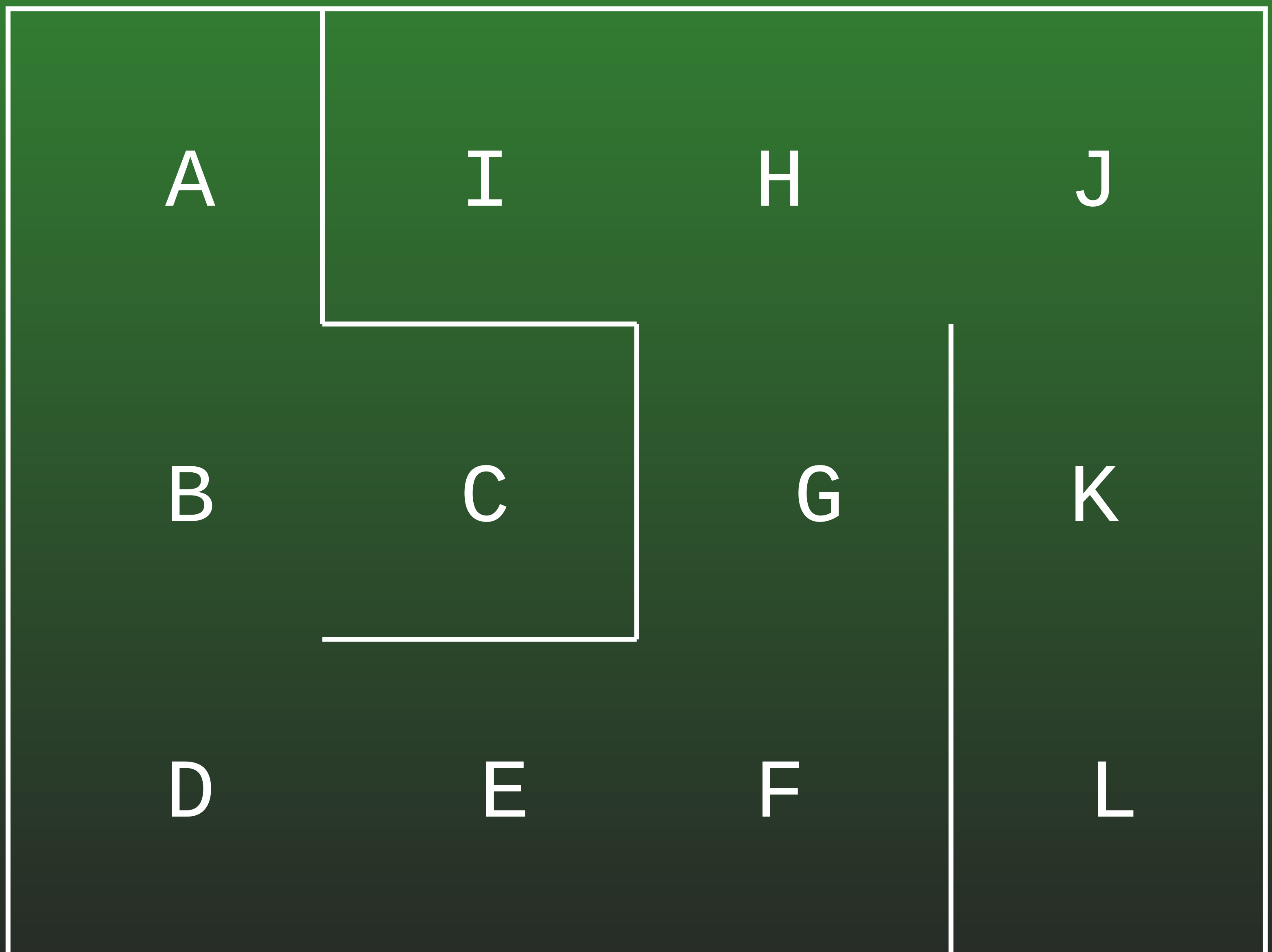
12-27: A* Search Example

- (dequeue P: g = 317) T = 118 + 329 = 447, Z = 374 + 75 = 449, B = 518 + 0 = 518, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, S = 338 + 253 = 591, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671
- (dequeue T: g = 118) Z = 374 + 75 = 449, L = 229 + 244 = 473, B = 518 + 0 = 518, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 338 + 253 = 591, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671
- (dequeue Z: g = 75) L = 229 + 244 = 473, A = 150 + 336 = 486, B = 518 + 0 = 518, O = 146 + 380 = 526, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 338 + 253 = 591, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671

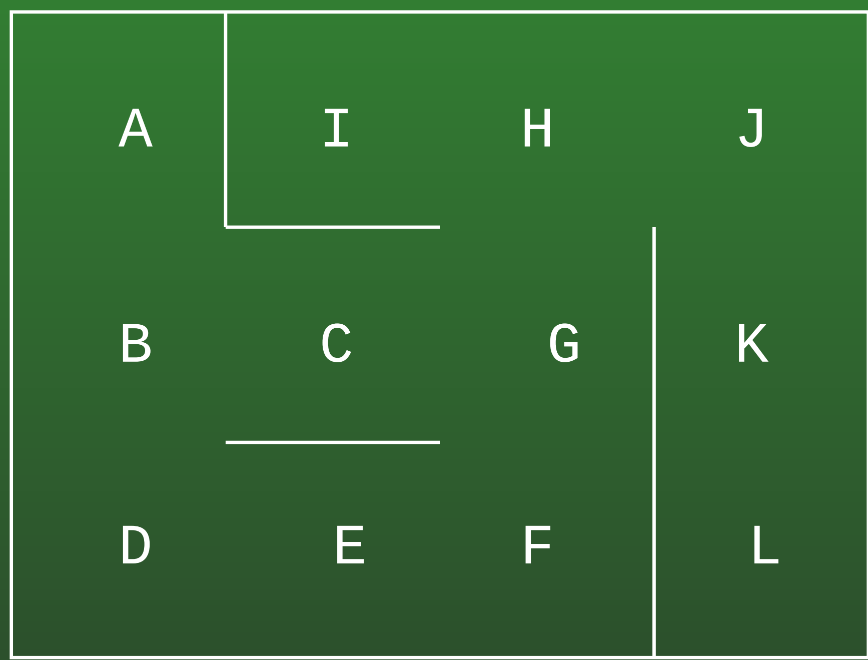
12-28: A* Search Example

- (dequeue L: g = 229) A = 150 + 336 = 486, B = 518 + 0 = 518, O = 146 + 380 = 526, C = 366 + 160 = 526, M = 299 + 241 = 540, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 338 + 253 = 591, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, T = 340 + 329 = 669, O = 291 + 380 = 671
- (dequeue A: g = 150) B = 518 + 0 = 518, O = 146 + 380 = 526, C = 366 + 160 = 526, M = 299 + 241 = 540, S = 290 + 253 = 543, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 338 + 253 = 591, T = 268 + 329 = 597, Z = 225 + 374 = 599, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, T = 340 + 329 = 669, O = 291 + 380 = 671
- (dequeue B: g = 518) solution. A -> S -> RV -> P -> B

A* Example II

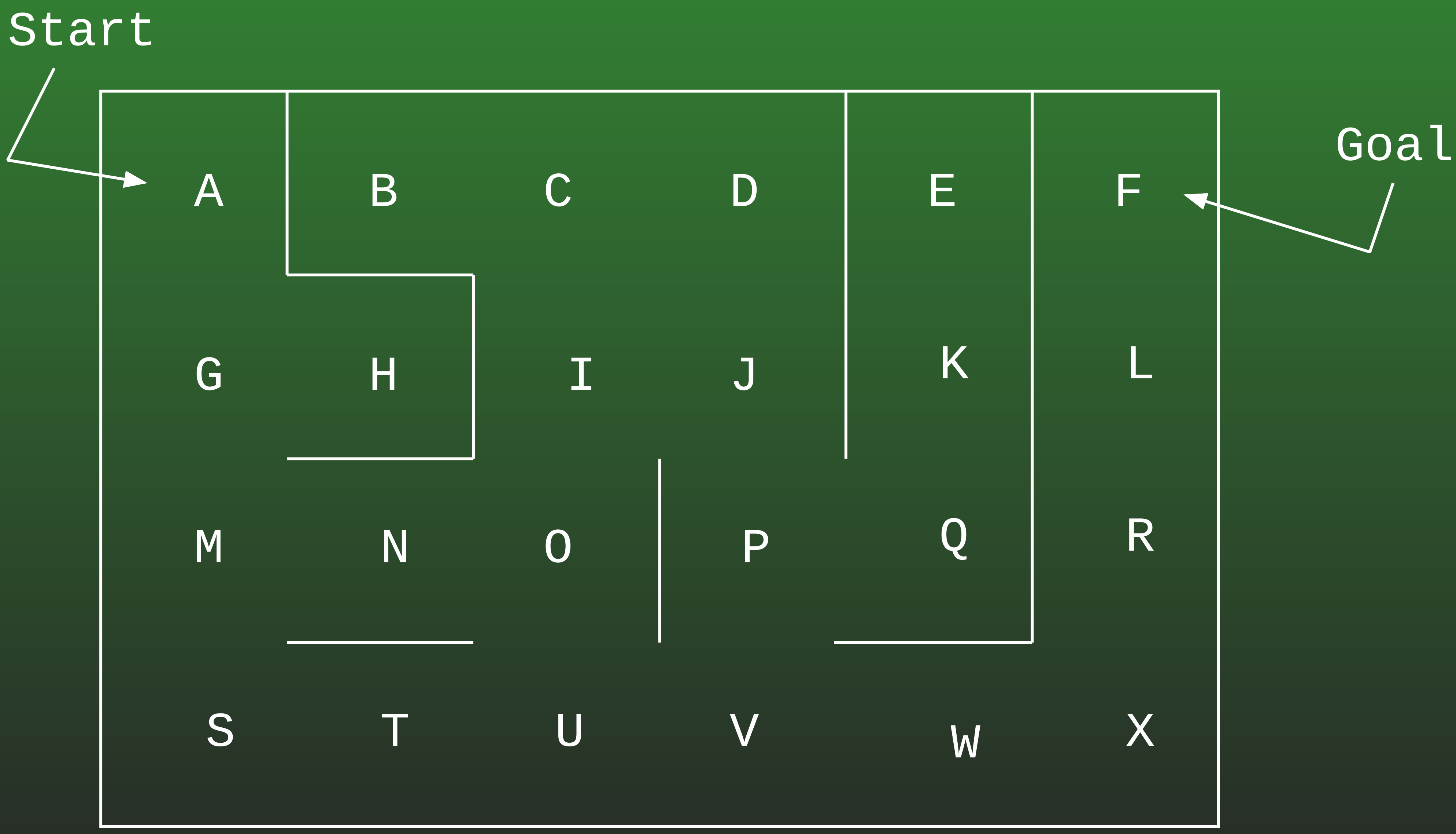


12-30: A* Example II

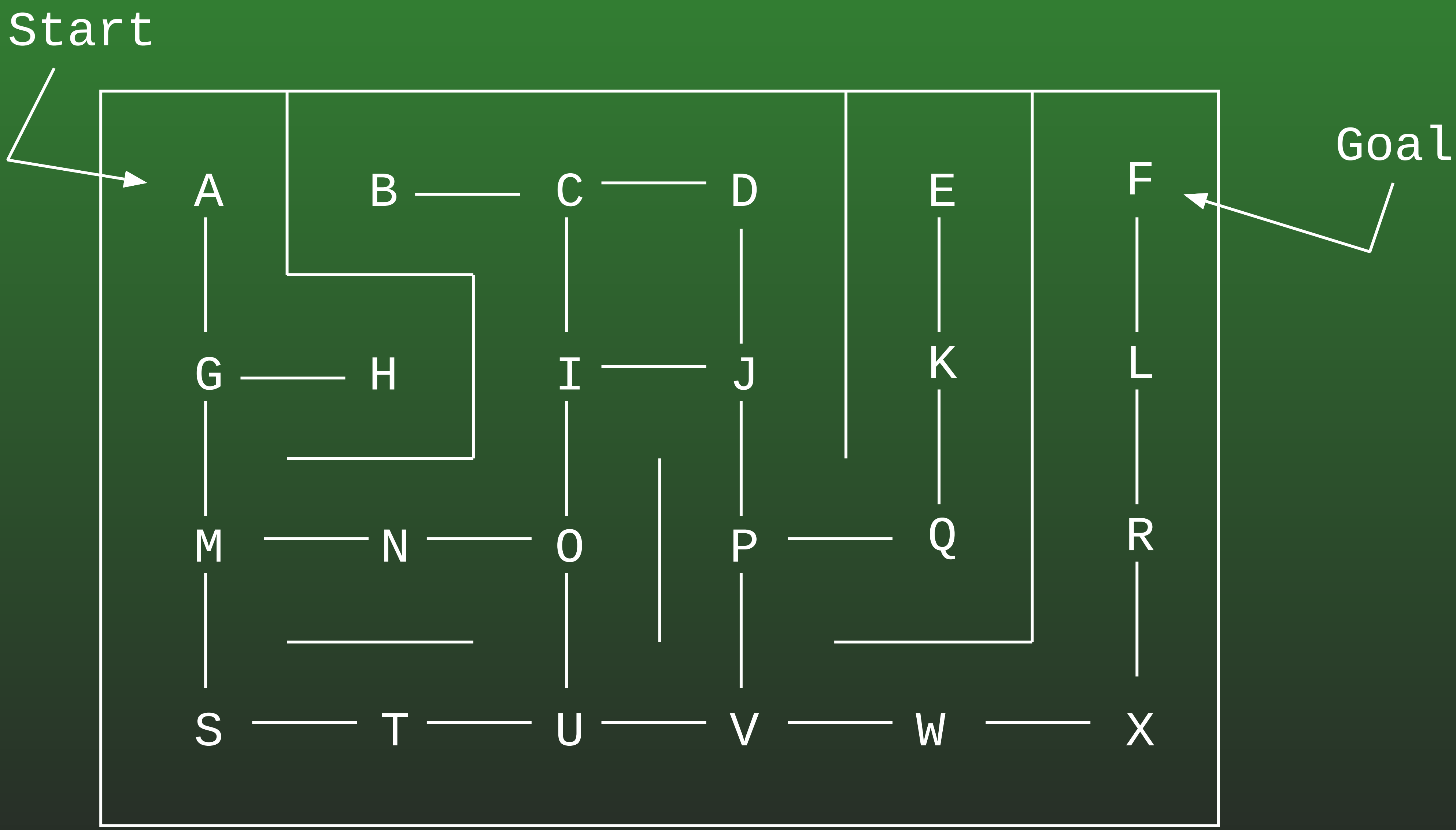


12-31: A* Example II

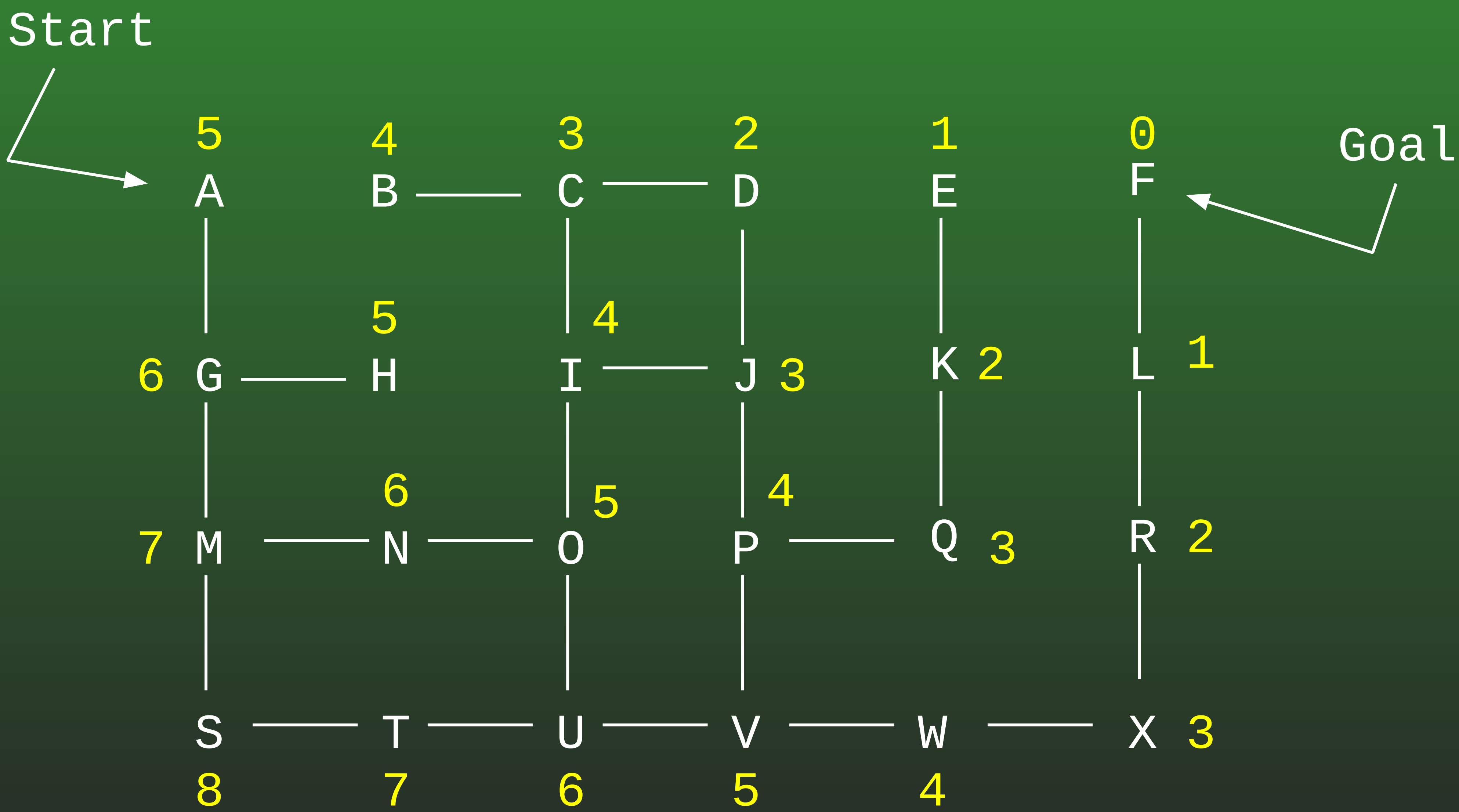
12-32: A* Example III



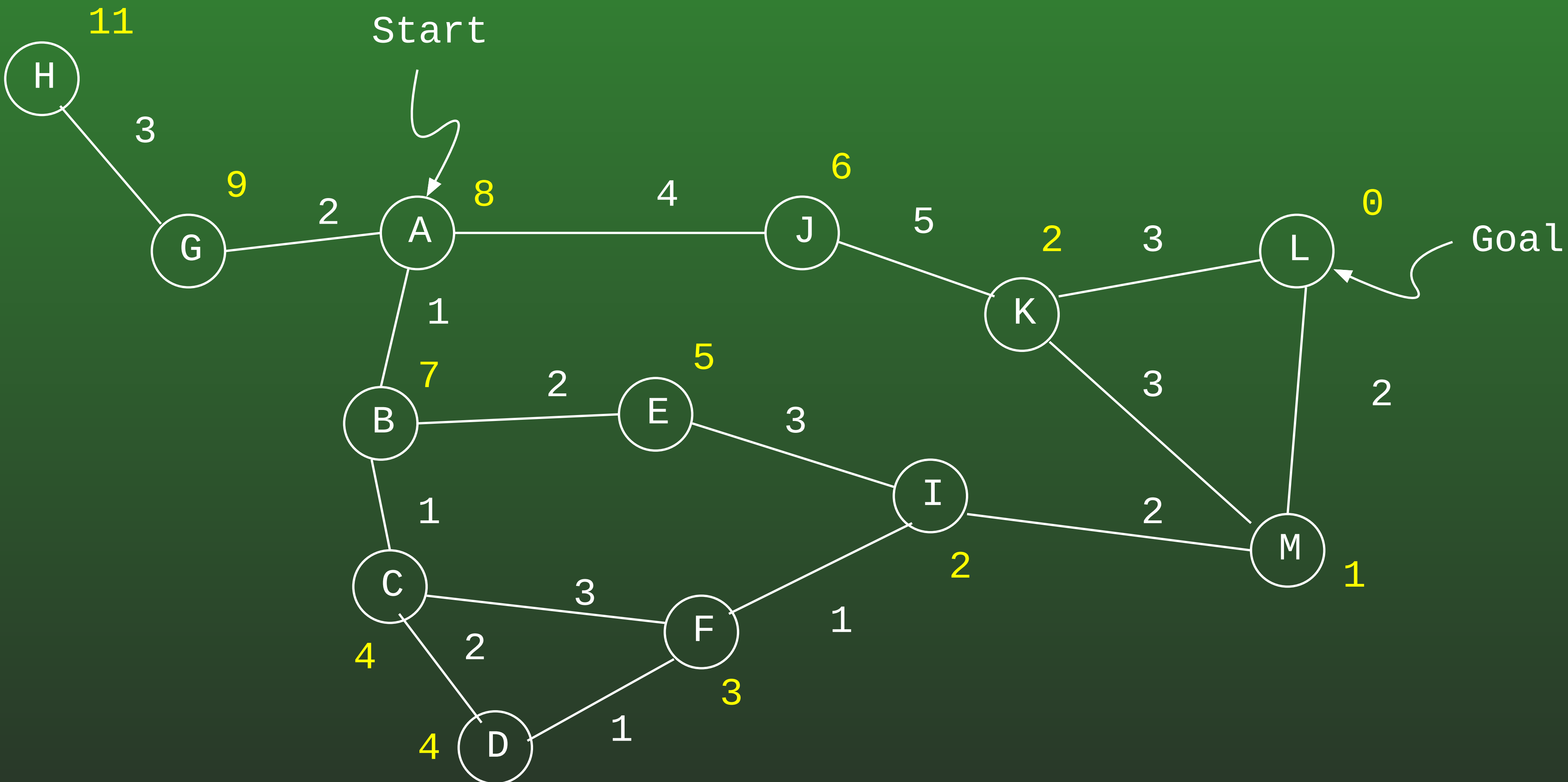
12-33: A* Example III



12-34: A* Example III



12-35: A* Example IV



h() values in yellow
edge costs in white

Node expsnsion order for
BFS, Uniform Cost, Greedy, A*

12-36: A* Example IV

- BFS:
  - AGBJHCEKDIML (goal found)
  - (Other orderings are possible)

12-37: A* Example IV

- Uniform Cost Search:
  - ABCGEJDHFIMKL
  - (Other orderings are possible)

12-38: A* Example IV

- Greedy
  - AJKL
  - (Other orderings are possible)

12-39: A* Example IV

- A*
  - ABCFIEMJL
  - (Other orderings are possible)

12-40: Optimality of A*

- A* is optimal (finds the shortest solution) as long as our h function is *admissible*.
  - Admissible: always underestimates the cost to the goal.
- Proof: When we dequeue a goal state, we see $g(n)$, the actual cost to reach the goal. If h underestimates, then a more optimal solution would have had a smaller $g + h$ than the current goal, and thus have already been dequeued.
- Or: If h overestimates the cost to the goal, it's possible for a good solution to "look bad" and get buried further back in the queue.

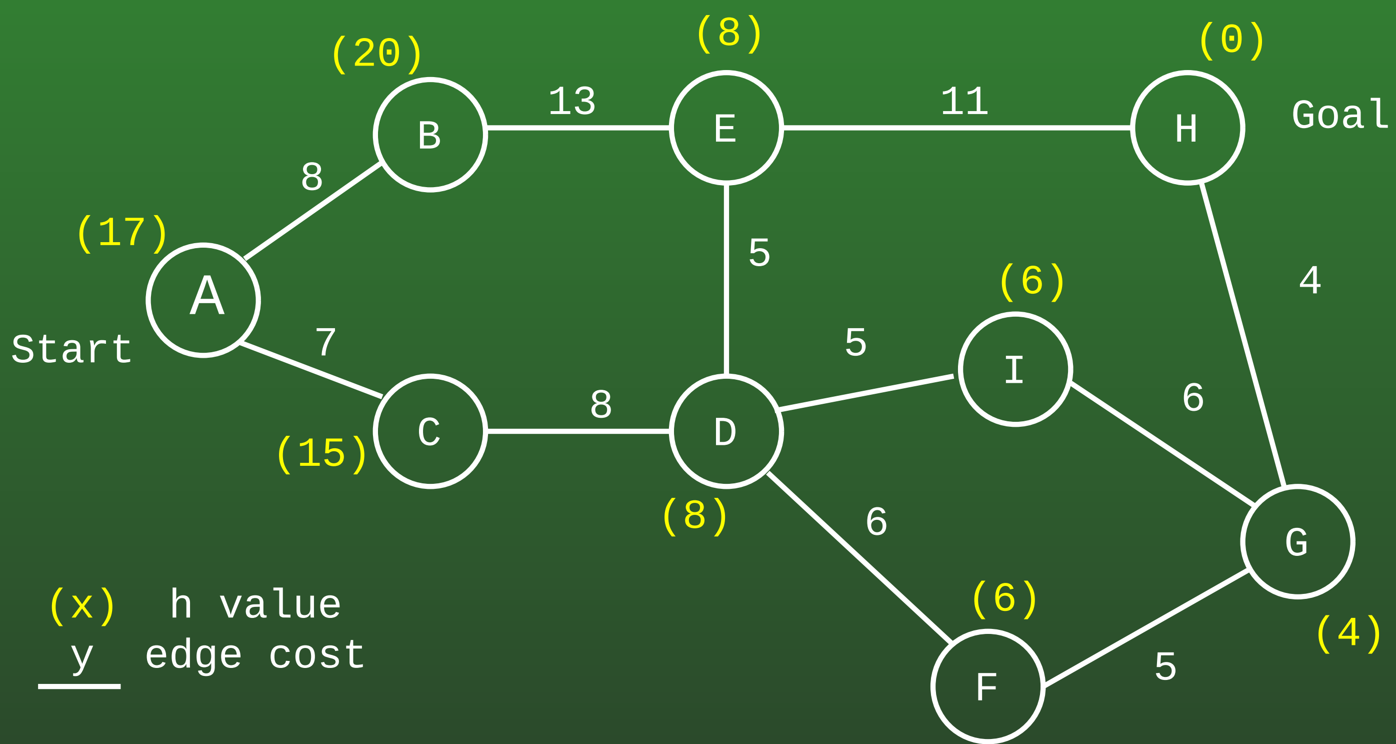
Optimality of A*

- Notice that we can't discard repeated states.
  - We could always keep the version of the state with the lowest g
- More simply, we can also ensure that we always traverse the best path to a node first.
- a *monotonic* heuristic guarantees this.
- A heuristic is monotonic if, for every node n and each of its successors (n'), $h(n)$ is less than or equal to $stepCost(n, n') + h(n')$.
  - In geometry, this is called the triangle inequality.

12-42: Optimality of A*

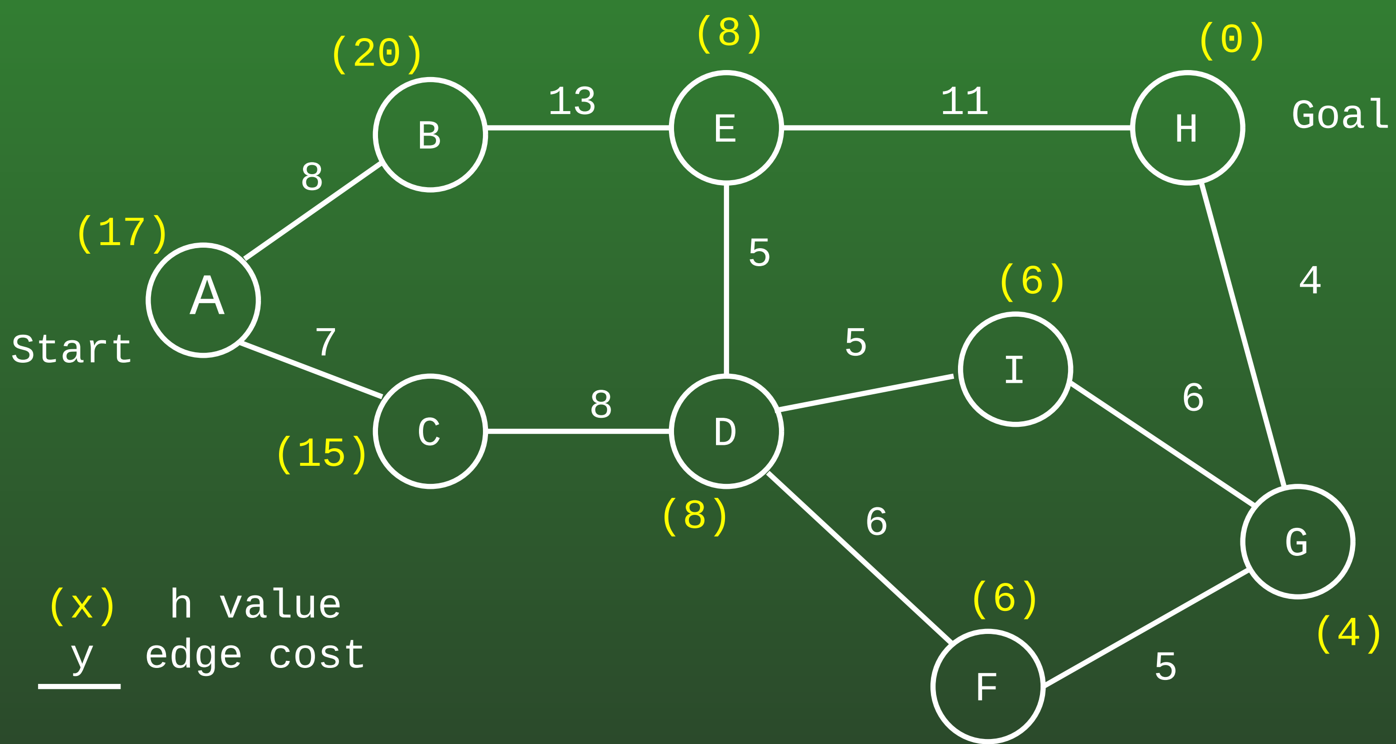
- SLD is monotonic. (In general, it's hard to find realistic heuristics that are admissible but not monotonic).
- Corollary: If h is monotonic, then f is nondecreasing as we expand the search tree.
- Alternative proof of optimality.
- Notice also that UCS is A* with $h(n) = 0$
- A* is also *optimally efficient*
  - No other complete and optimal algorithm is guaranteed to expand fewer nodes.

12-43: A* Example II



- Is h() admissible?
- Is h() monotonic?

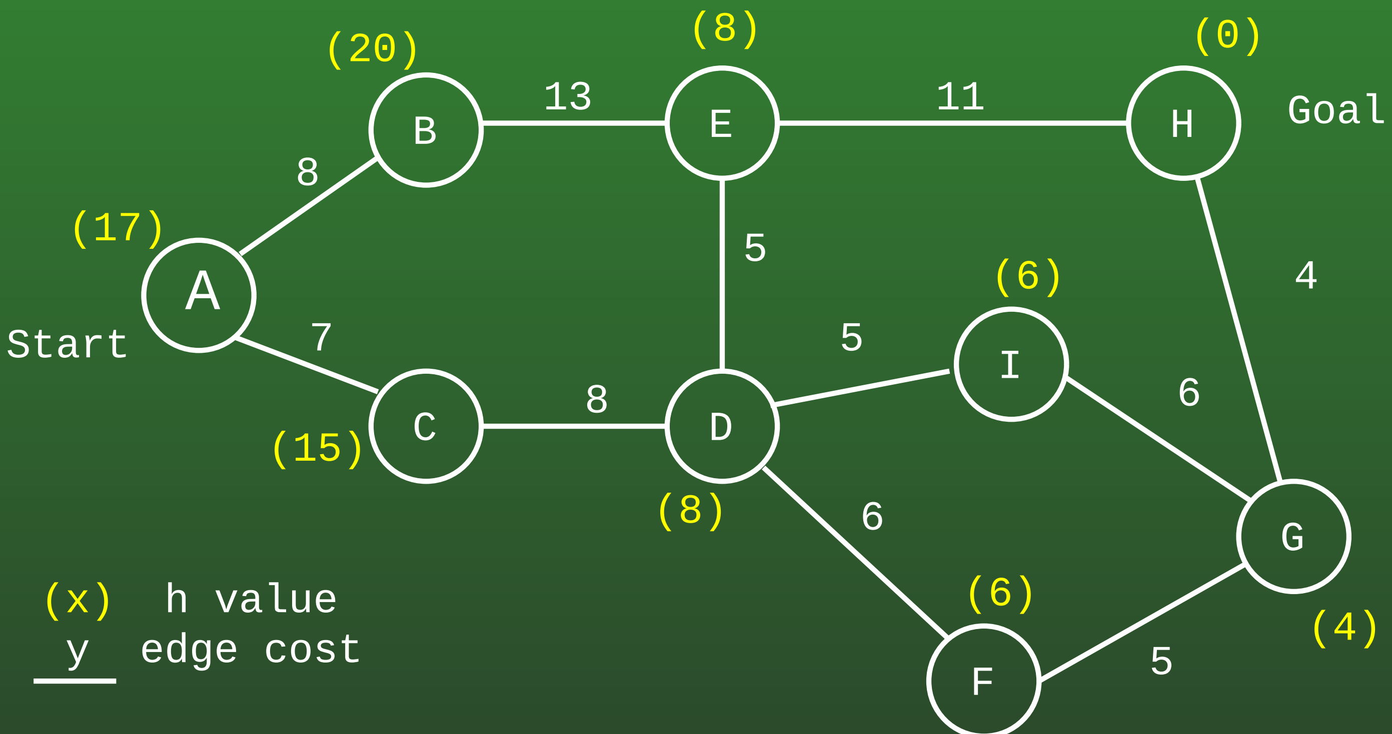
12-44: A* Example II



Node: Queue :

-- [(A f = 17, g = 0, h = 17)]

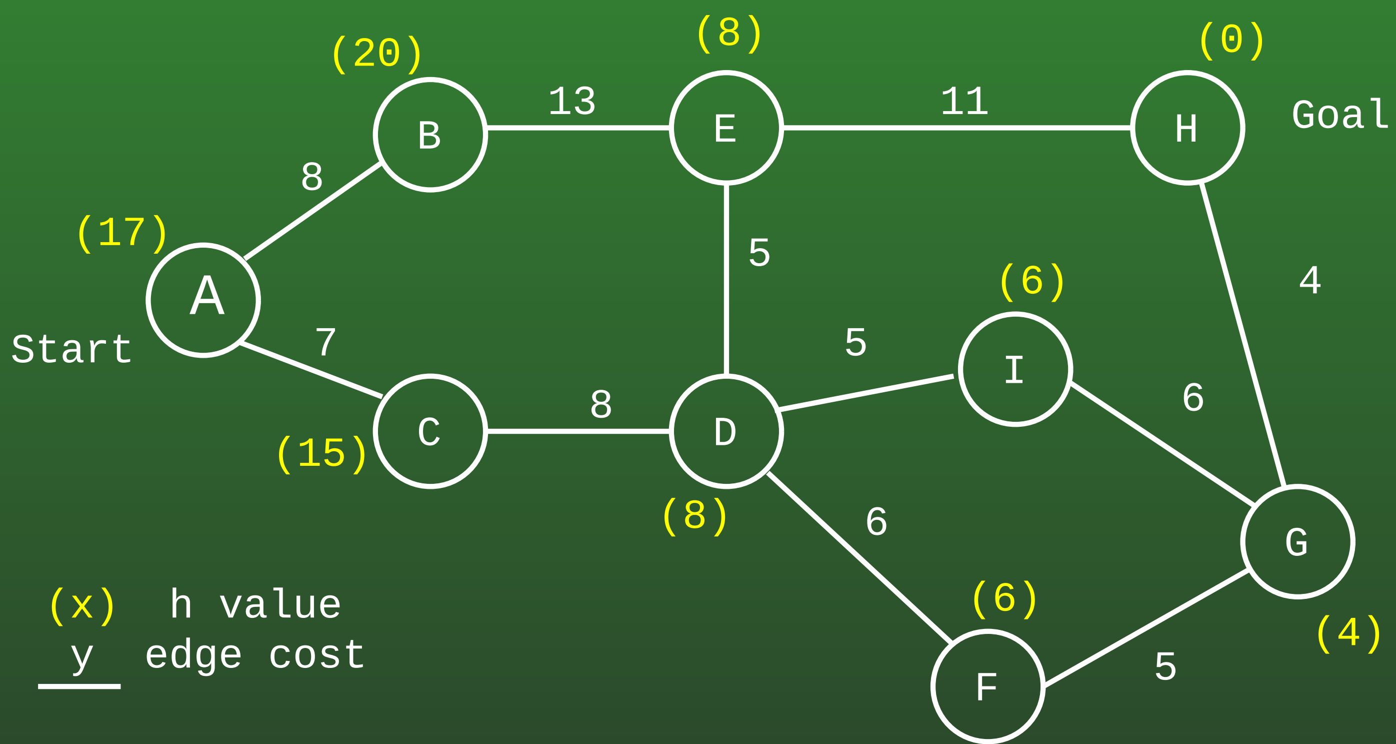
12-45: A* Example II



Node: Queue :

A [(C f = 22, g = 7, h = 15), (B f = 28, g = 8, h = 20)]

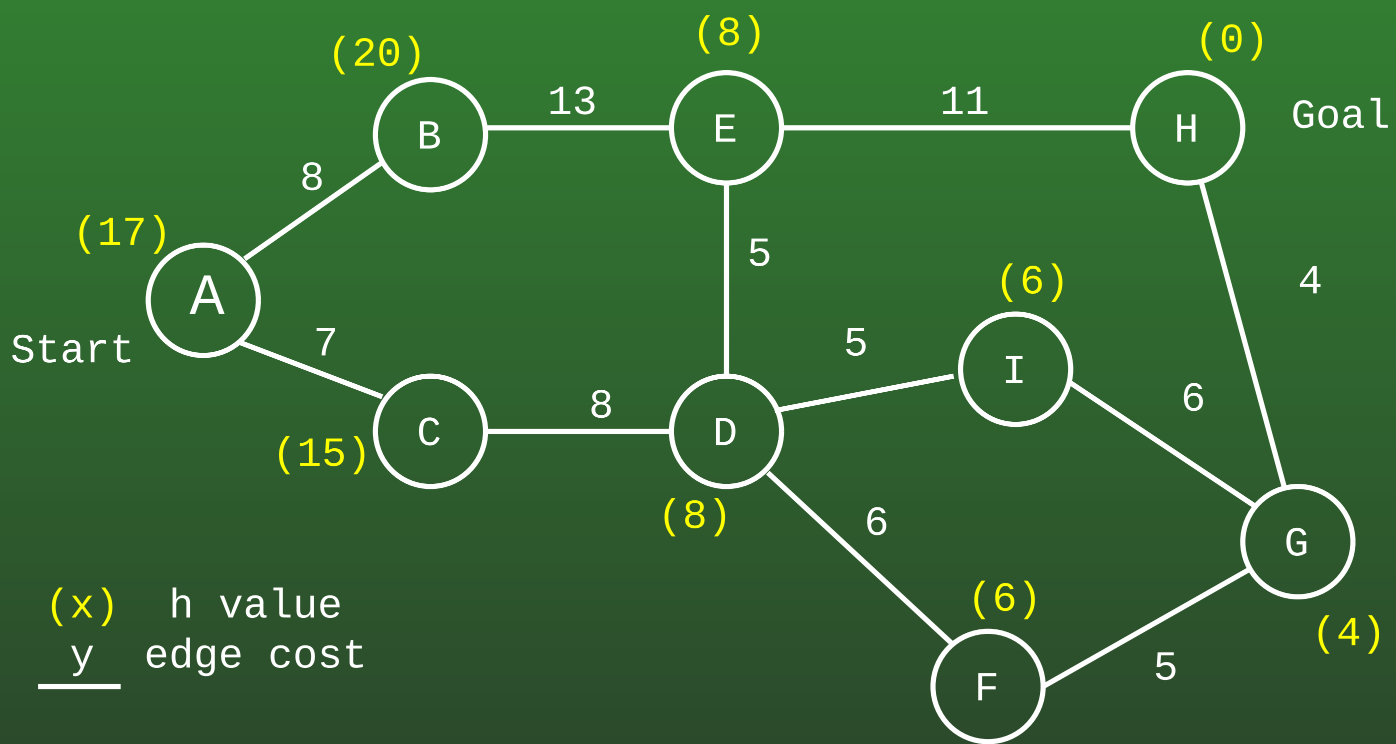
12-46: A* Example II



Node: Queue :

C [(D f = 23, g = 15, h = 8), (B f = 28, g = 8, h = 20)]

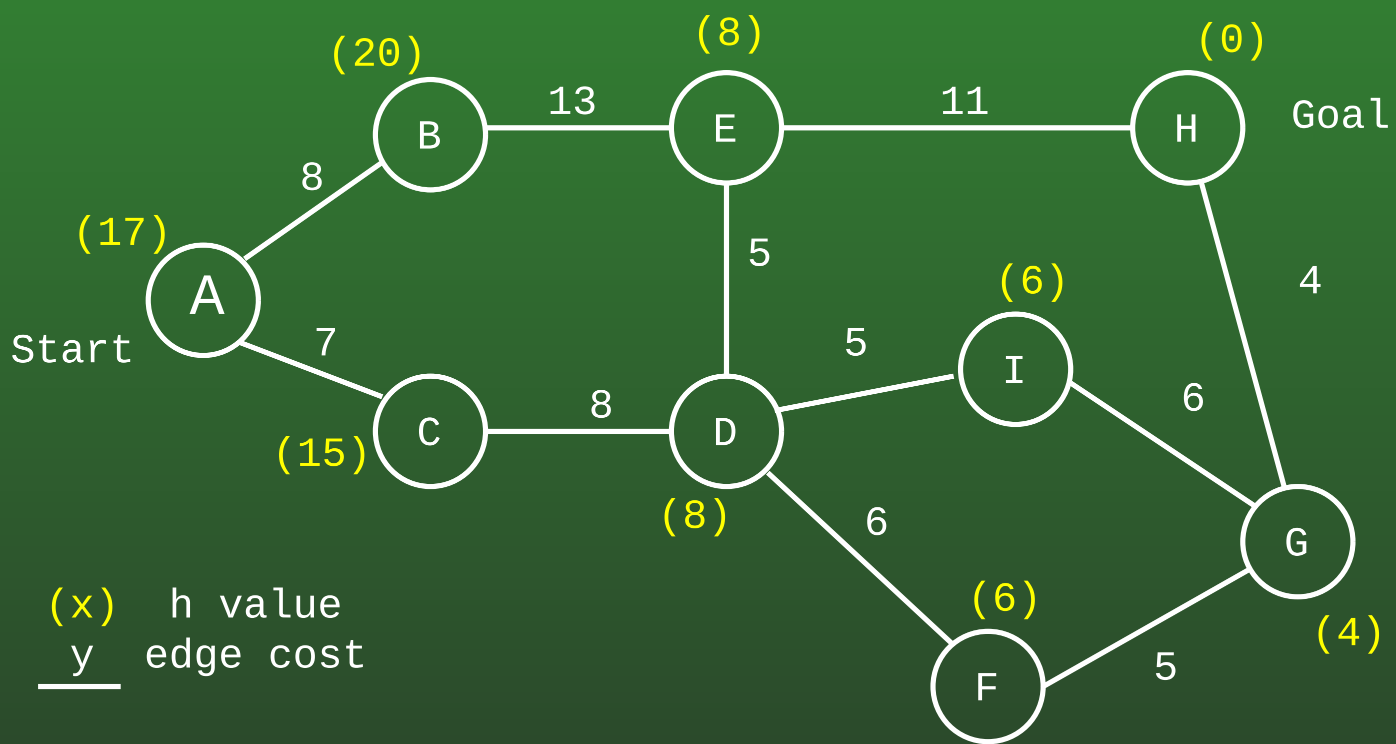
12-47: A* Example II



Node: Queue :

D [(I f = 26, g = 20, h = 6), (F f = 27, g = 21, h = 6),
(B f = 28, g = 8, h = 20), (E f = 28, g = 20, h = 8)]

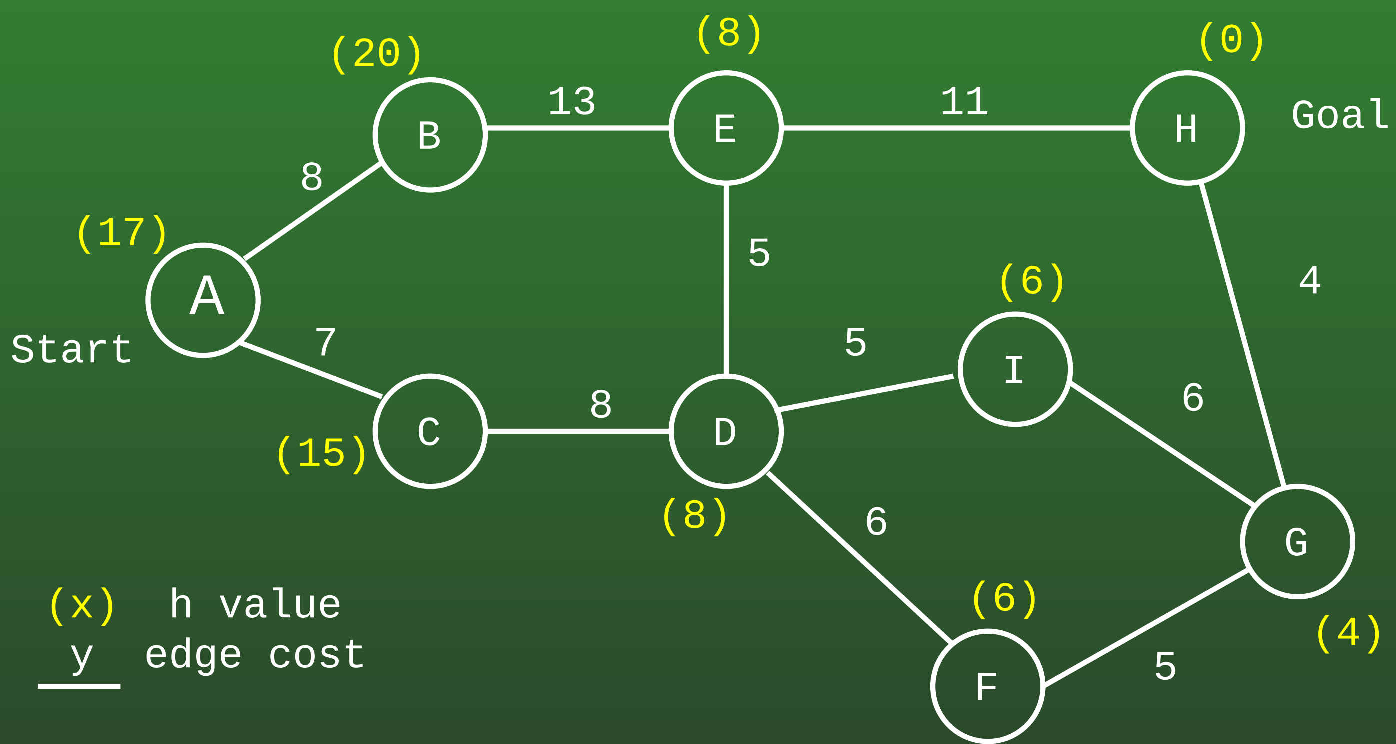
12-48: A* Example II



Node: Queue :

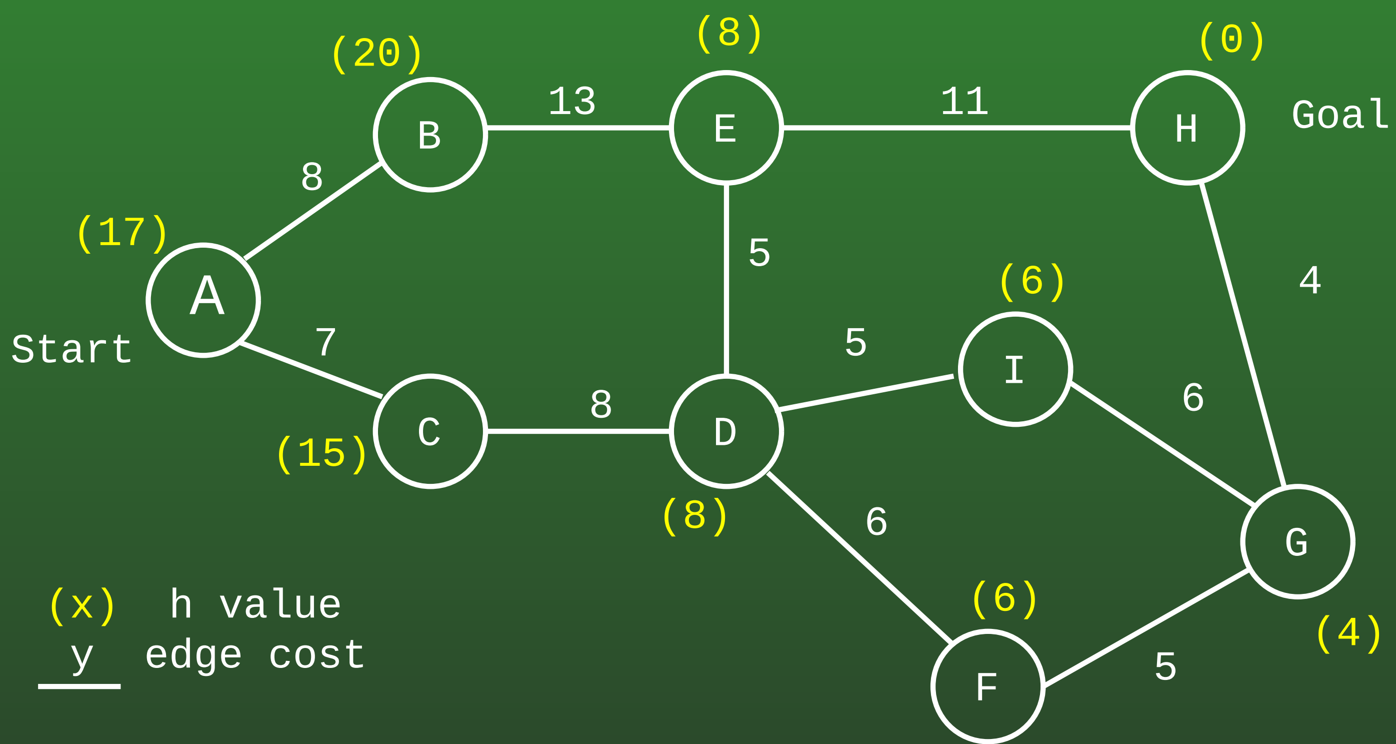
I [(F f = 27, g = 21, h = 6), (B f = 28, g = 8, h = 20),
(E f = 28, g = 20, h = 8), (G f = 30 g = 26, h = 4)]

12-49: A* Example II




```
Node:   Queue :
 F      [(B f = 28, g = 8, h = 20), (E f = 28, g = 20, h = 8),
         (G f = 30  g = 26, h = 4),(G f = 30  g = 26   h = 4)]
```

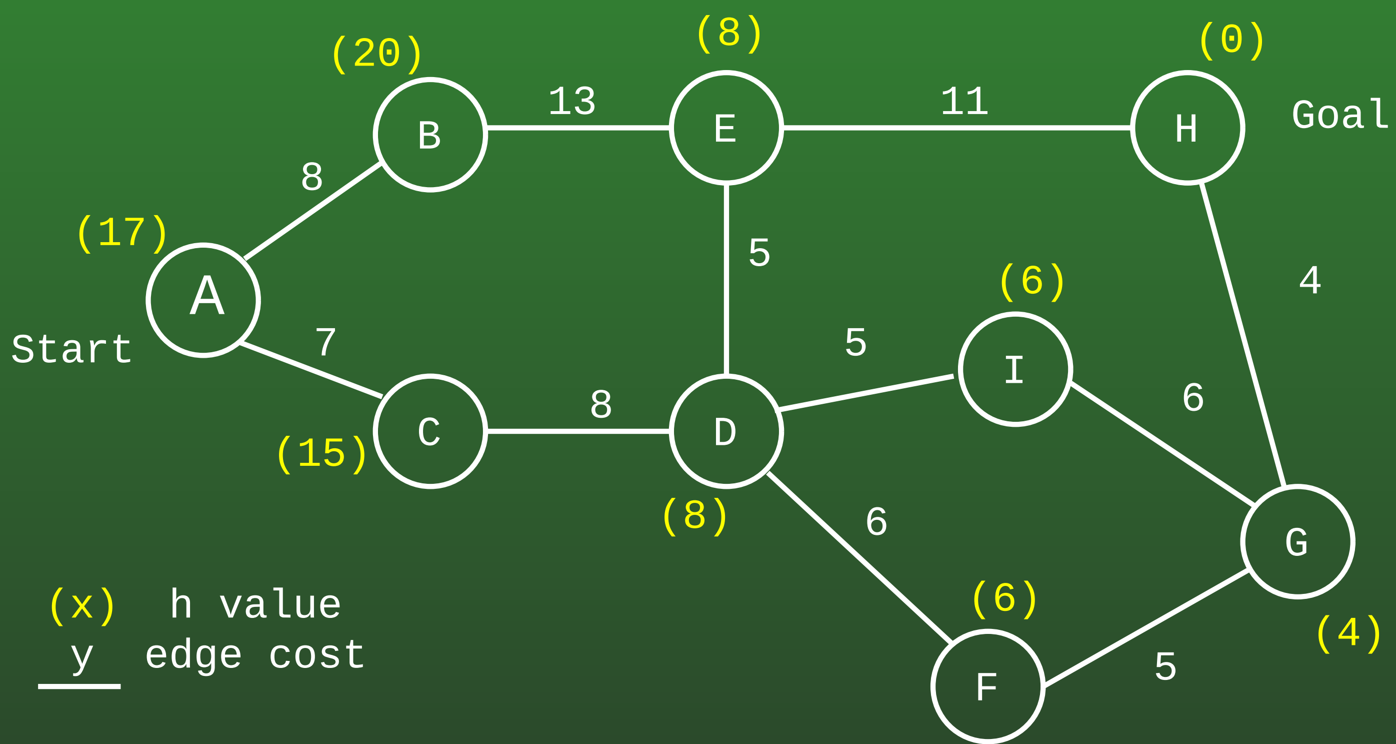
12-50: A* Example II



Node: Queue :

B [(E f = 28, g = 20, h = 8), (E f = 29, g = 21, h = 8),
(G f = 30, g = 26, h = 4),(G f = 30, g = 26, h = 4)]

12-51: A* Example II



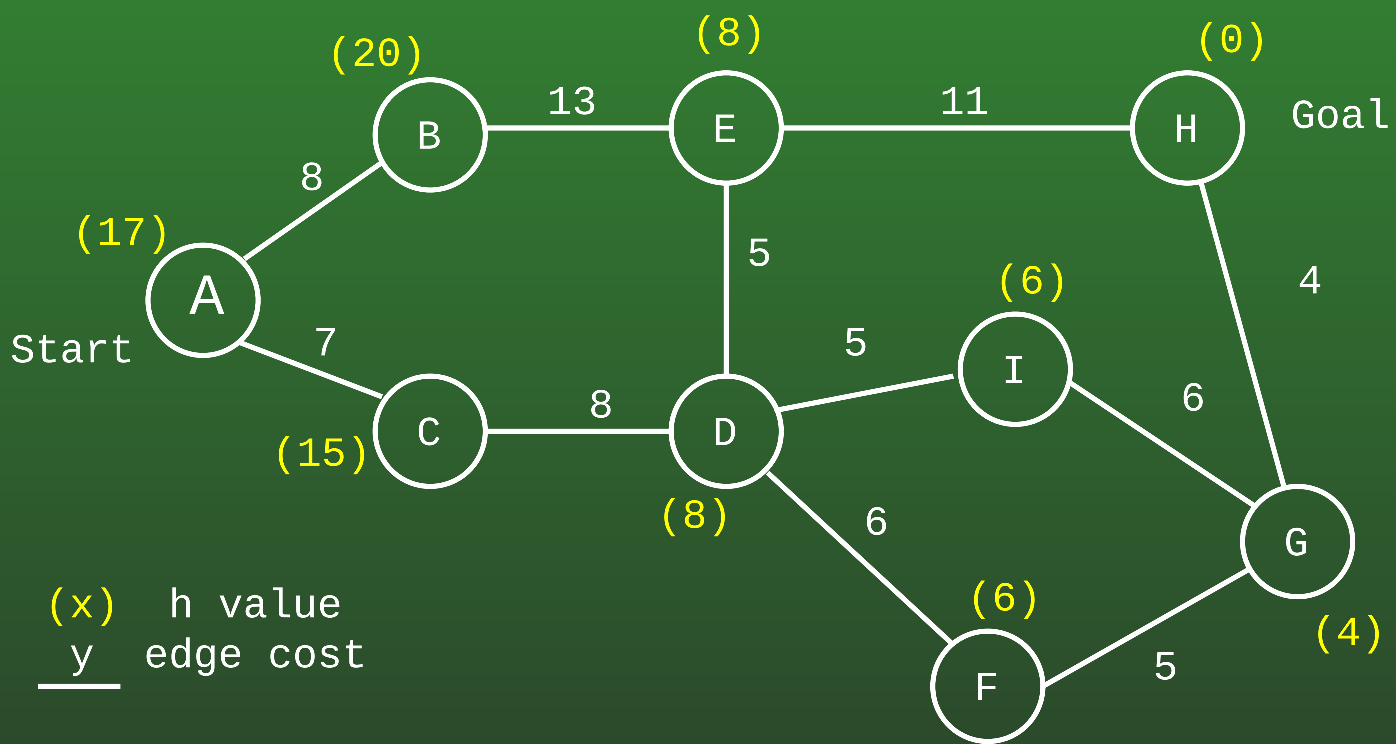
Node: Queue :

E [(E f = 29, g = 21, h = 8), (G f = 30 g = 26, h = 4),

(G f = 30 g = 26 h = 4), (H f = 31, g = 31, h = 0)]

(next E can be discarded)

12-52: A* Example II

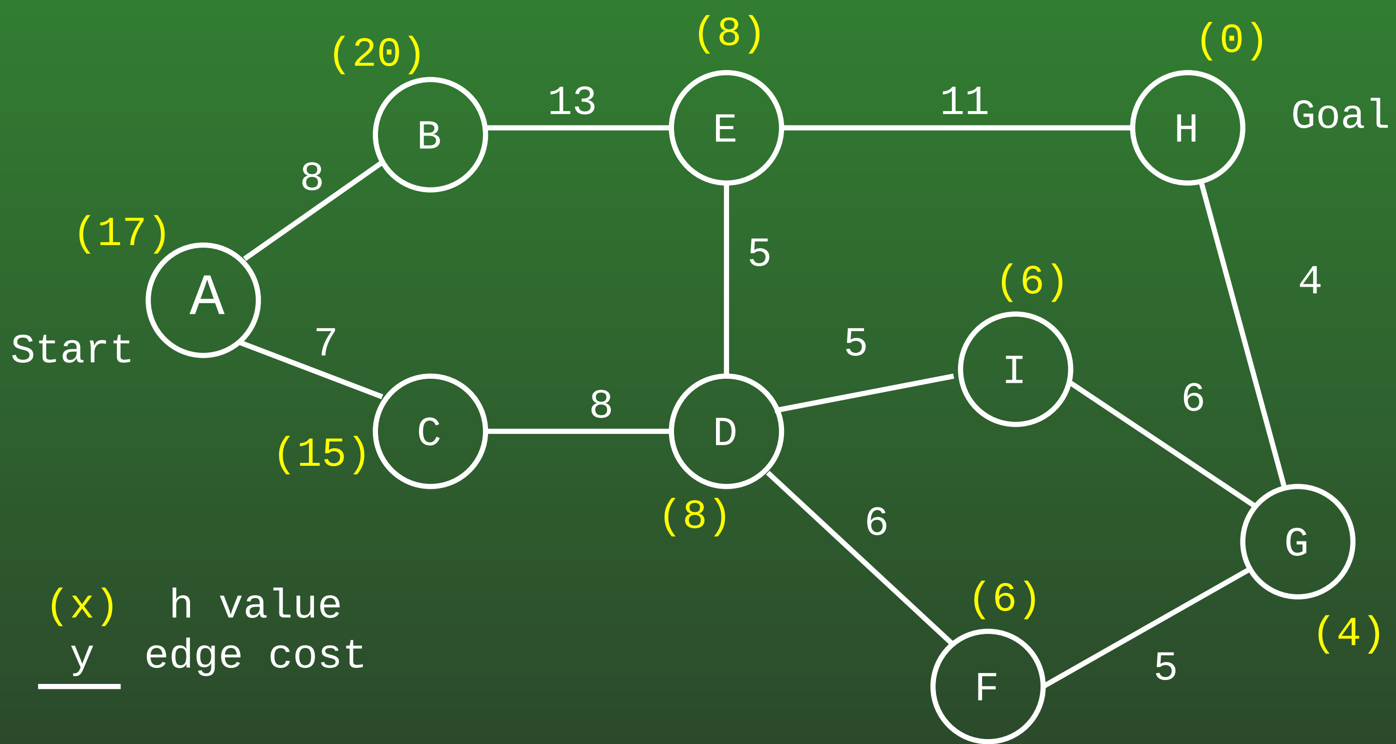


Node: Queue :

G [(G f = 30 g = 26 h = 4), (H f = 30, g = 30, h = 0),
(H f = 31, g = 31, h = 0)]

(next G can be discarded)

A* Example II

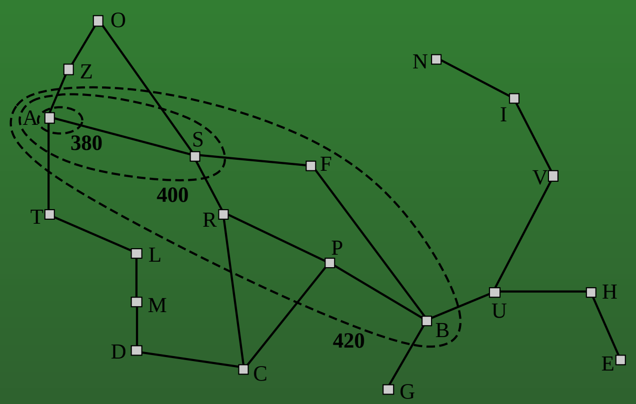


Node: Queue :

H. Goal. [(H f = 31, g = 31, h = 0)]

Solution: A,C,D,I,G,H (or A,C,D,F,G,H)

12-54: Pruning and Contours



- Topologically, we can imagine A* creating a set of contours corresponding to f values over the search space.
- A* will search all nodes within a contour before expanding.
- This allows us to *prune* the search space.
  - We can chop off the portion of the search tree corresponding to Zerind without searching it.

12-55: IDA*

- A* has one big weakness - Like BFS, it potentially keeps an exponential number of nodes in memory at once.
- Iterative deepening A* is a workaround
  - IDS was depth-limited search – IDA* is f-limited search
  - Each iteration, increase bound to smallest value that allows search to continue

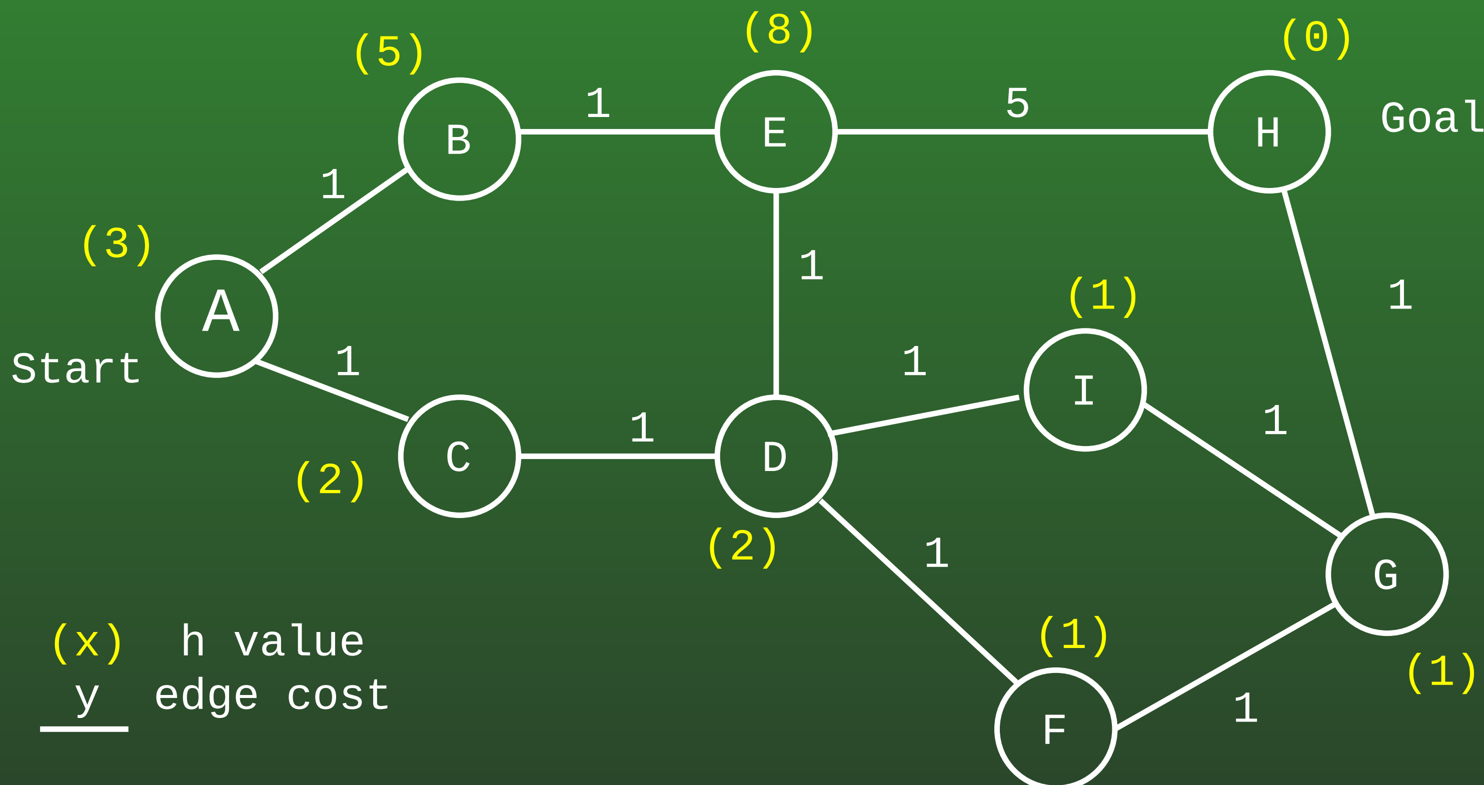
12-56: Iterative Deepening A* (IDA*)

```
f-limited-DFS(node, limit)
   if g(n) + h(n) > limit
      return fail, g(node) + h(node)
   if goalTest(node)
      return node, g(node)
   children = successor(node)
   smallestFail = MAX_VALUE
   for child in children
      sol, cost = depth-limited-DFS(child, limit)
      if sol != fail
         return sol, cost
      smallestFail = min(cost, smallestFail)
   return smalestFail, fail
```

12-57: Iterative Deepening A* (IDA*)

```
ida-star(node)
   limit = h(node)
   while true
      sol, limit = f-limited-DFS(node, limit)
      if (sol != fail)
         return sol
```

12-58: IDA* Example



12-59: IDA*

- Works well in works with discrete-valued step costs
  - Prefereably with steps having the same cost
- Each iteration brings in a large section of nodes
- What is the worst case performance for IDA*?
- When does the worst case occur?

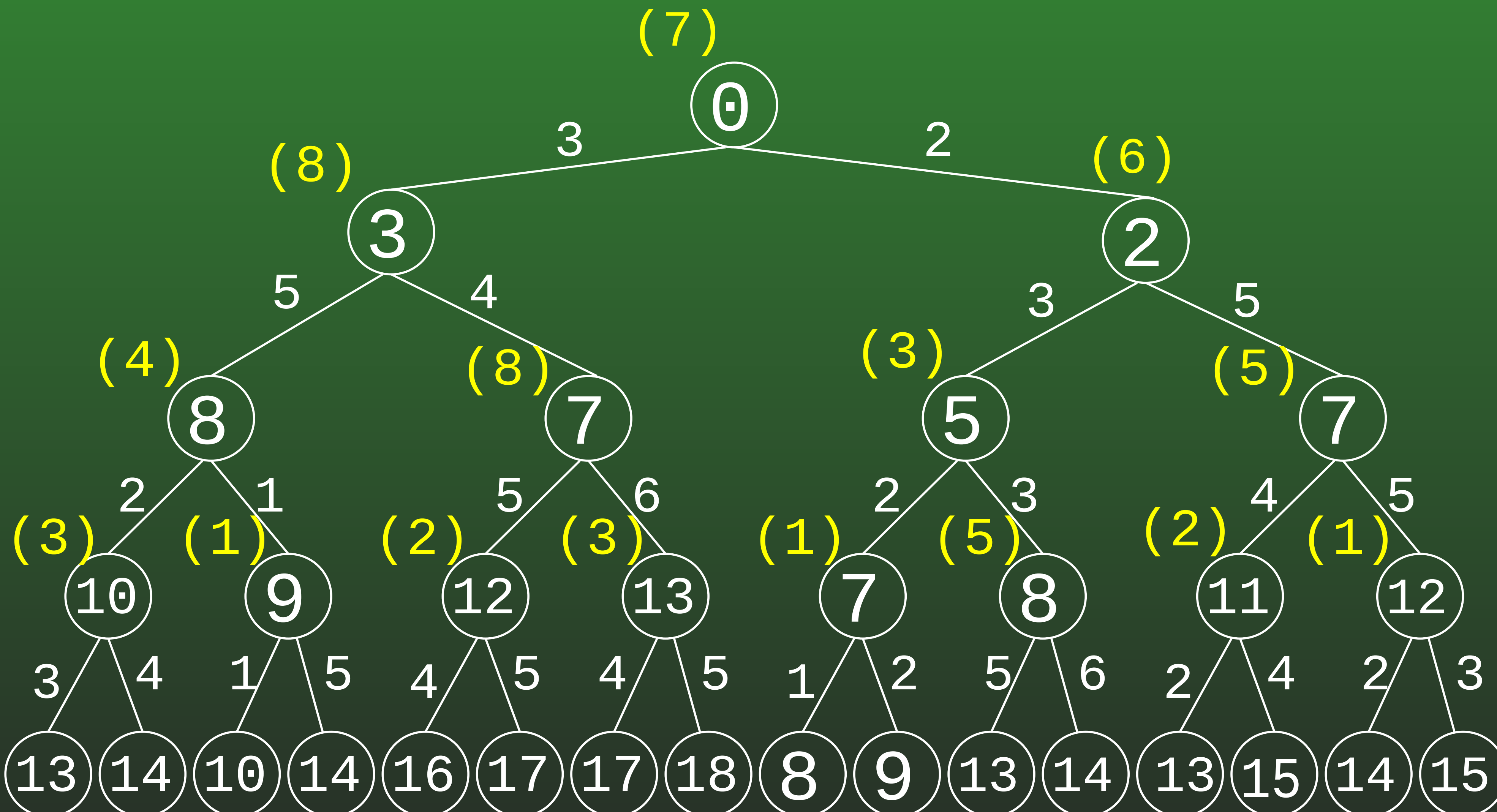
SMA*

- Run regular A*, with a fixed memory limit
- When limit is reached, discard node with highest f
- Value of discarded node is assigned to the parent
  - Use the discarded node to get a better f value for parent
  - ’remember’ the value of that branch
  - If all other branches get higher f value, regenerate
- SMA* is complete and optimal
- Very hard problems can case SMA* to thrash, repeatedly regenerating branches

12-61: DFB&B

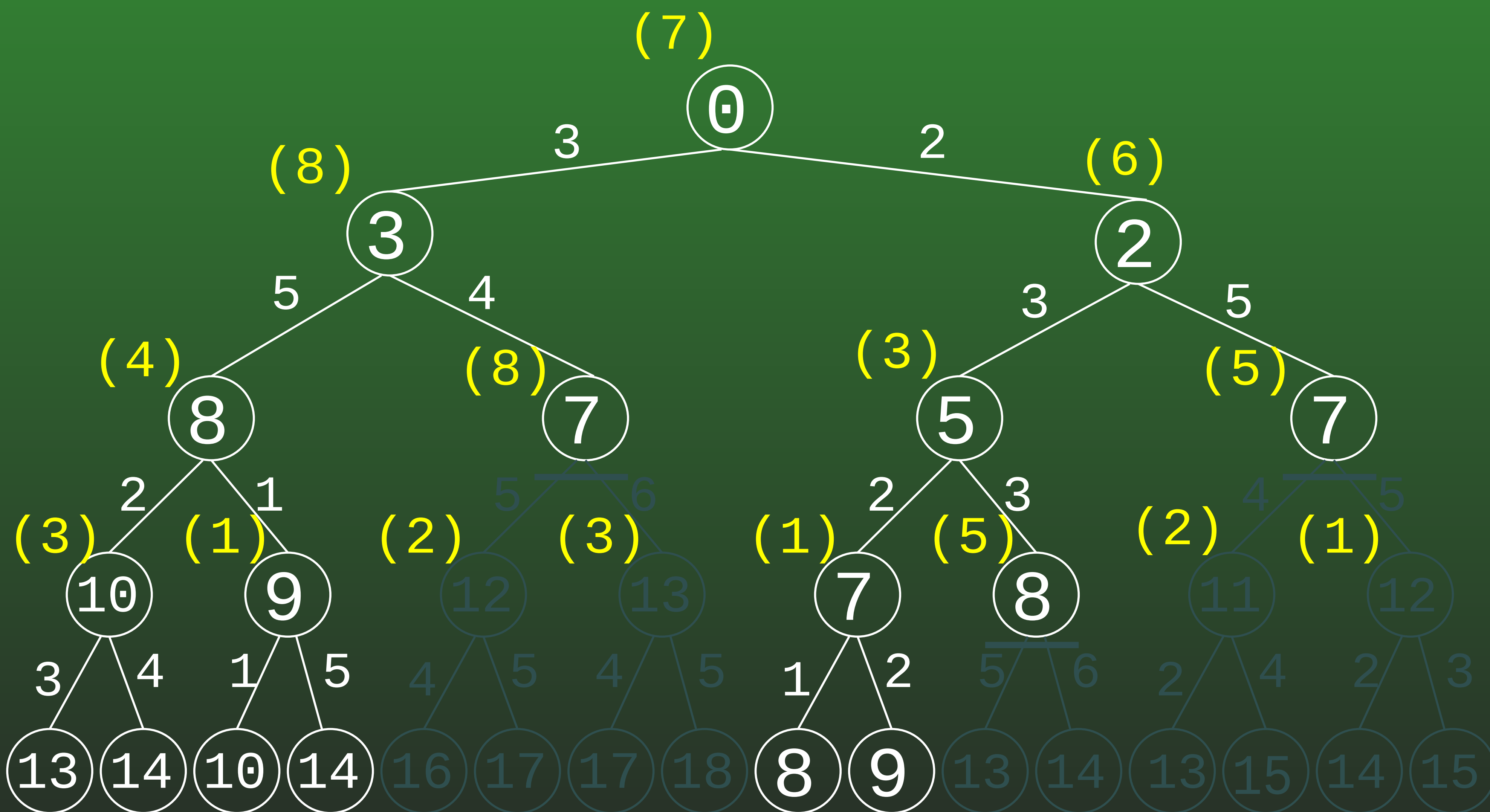
- Depth-First Branch and Bound
  - Run f-limited DFS, with limit set to infinity
  - When a goal is found, don’t stop – record it, and set limit to the goal depth
  - Keep going until all branches are searched or pruned.
- We will use something similar in 2-player games
- (DFB&B not in the text)

12-62: DFB&B



(7)
0
3
2
(8)
(6)
3
2
5
4
3
5
(4)
(8)
(3)
(5)
8
7
5
7
2
1
5
6
2
3
4
5
(3)
(1)
(2)
(3)
(1)
(5)
(2)
(1)
10
9
12
13
7
8
11
12
3
4
1
5
4
5
4
5
1
2
5
6
2
4
2
3
13
14
10
14
16
17
17
18
8
9
13
14
13
15
14
15

DFB&B


(7)
0
3
2
(8)
3
(6)
2
5
4
3
5
(4)
8
(8)
7
(3)
5
(5)
7
2
1
5
6
2
3
4
5
(3)
(1)
(2)
(3)
(1)
(5)
(2)
(1)
10
9
12
13
7
8
11
12
3
4
1
5
4
5
4
5
1
2
5
6
2
4
2
3
13
14
10
14
16
17
17
18
8
9
13
14
13
15
14
15

12-64: DFB&B

- What kinds of problems might Depth-First Branch and Bound work well for?
- Is DFB&B Complete? Optimal?
- How could we improve performance?

DFB&B

- What kinds of problems might Depth-First Branch and Bound work well for?
  - Optimization: Finding a solution is easy, finding the best is hard (TSP)
- Is DFB&B Complete? Optimal?
  - If we can find *a* solution easily, it is complete and optimal
- How could we improve performance?
  - Examine children in increasing g() value

12-66: DFB&B

- Some nice features:
  - Quickly find a solution
  - Best solution so far gradually gets better
  - Run DFB&B until it finishes (we have an optimal solution), or we run out of time (use the best so far)

Building Effective Heuristics

- While A* is optimally efficient, actual performance depends on developing accurate heuristics.
- Ideally, h is as close to the actual cost to the goal (h^*) as possible while remaining admissible.
- Developing an effective heuristic requires some understanding of the problem domain.

12-68: Effective Heuristics - 8-puzzle

- h_1 - number of misplaced tiles.
  - This is clearly admissible, since each tile will have to be moved at least once.
- h_2 - *Manhattan distance* between each tile's current position and goal position.
  - Also admissible - best case, we'll move each tile directly to where it should go.
- Which heuristic is better?

12-69: Effective Heuristics - 8-puzzle

- h_2 is better.
  - We want h to be as close to h^* as possible.
- If $h_2(n) > h_1(n)$ for all n, we say that h_2 *dominates* h_1.
- We would prefer a heuristic that dominates other known heuristics.

12-70: Finding a heuristic

- So how do we find a good heuristic?
- Solve a relaxed version of the problem.
  - 8-puzzle:
    - Tile can be moved from A to B if:
      - A is adjacent to B
      - B is blank
    - Remove restriction that A is adjacent to B
      - Misplaced tiles
    - Remove restriction that B is blank
      - Manhattan distance

12-71: Finding a heuristic

- So how do we find a good heuristic?
- Solve a relaxed version of the problem.
  - Romania path-finding
    - Add an extra road from each city directly to goal
    - (Decreases restrictions on where you can move)
  - Straight-line distance heuristic

12-72: Finding a heuristic

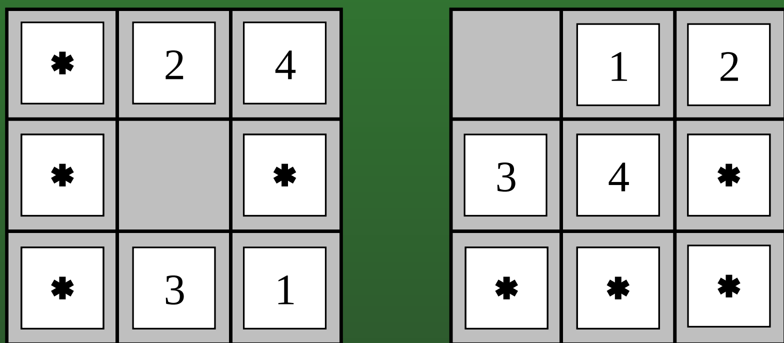
- So how do we find a good heuristic?
- Solve a relaxed version of the problem.
  - Traveling Salesman
    - Connected graph
    - Each node has 2 neighbors
  - Minimum Cost Spanning Tree Heuristic

12-73: Finding a heuristic

- Solve subproblems
  - Cost of getting a subset of the tiles in place (ignoring the cost of moving other tiles)
- Save these subproblems in a database (could get large, depending upon the problem)

12-74: Finding a heuristic

- Using subproblems



Start State Goal State

12-75: Finding a heuristic

- Number of heurisitcs $h_1, h_2, \ldots h_k$
- No one heuristic dominates any other
  - Different heuristics have different performances with different states
- What can you do?

Finding a heuristic

- Number of heurisitcs $h_1, h_2, \ldots h_k$
- No one heuristic dominates any other
  - Different heuristics have different performances with different states
- What can you do?
  - $h(n) = \text{max}(h_1(n), h_2(n), \ldots h_k(n))$

Summary

- Problem-specific heuristics can improve search.
- Greedy search
- A*
- Memory limited search (IDA*, SMA*)
- Developing heuristics
  - Admissibility, monotonicity, dominance