AI Programming

CS662-2013S-06

Heuristic Search

David Galles

Department of Computer Science
University of San Francisco

06-0: Overview

- Heuristic Search - exploiting knowledge about the problem
- Heuristic Search Algorithms
  - “Best-first” search
  - Greedy Search
  - A* Search
  - Extensions to A*
- Constructing Heuristics

06-1: Informing Search

- Uninformed search was able to find solutions, but were very inefficient.
  - Exponential number of nodes expanded.
- By taking advantage of knowledge about the problem structure, we can improve performance.
- Two caveats:
  - We have to get knowledge about the problem from somewhere.
  - This knowledge has to be correct.

06-2: Best-first Search

- Uniform-cost search
  - Nodes were expanded based on their total path cost
  - Implemented using a priority queue
- Path cost is an example of an *evaluation function*.
  - We'll use the notation $f(n)$ to refer to an evaluation function.
- An evaluation function tells us how promising a node is.
- Indicates the quality of the solution that node leads to.

06-3: Best-first Search

- Best-first Pseudocode

```
enqueue(initialState)
do
  node = prioroty-dequeue()
  if goalTest(node)
     return node
  else
     children = successors(node)
     for child in children
         prioroty-enqueue(child, f(child))
```

- where insert-with orders our priority queue accordingly.

06-4: 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

06-5: 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

06-6: Heuristic Function

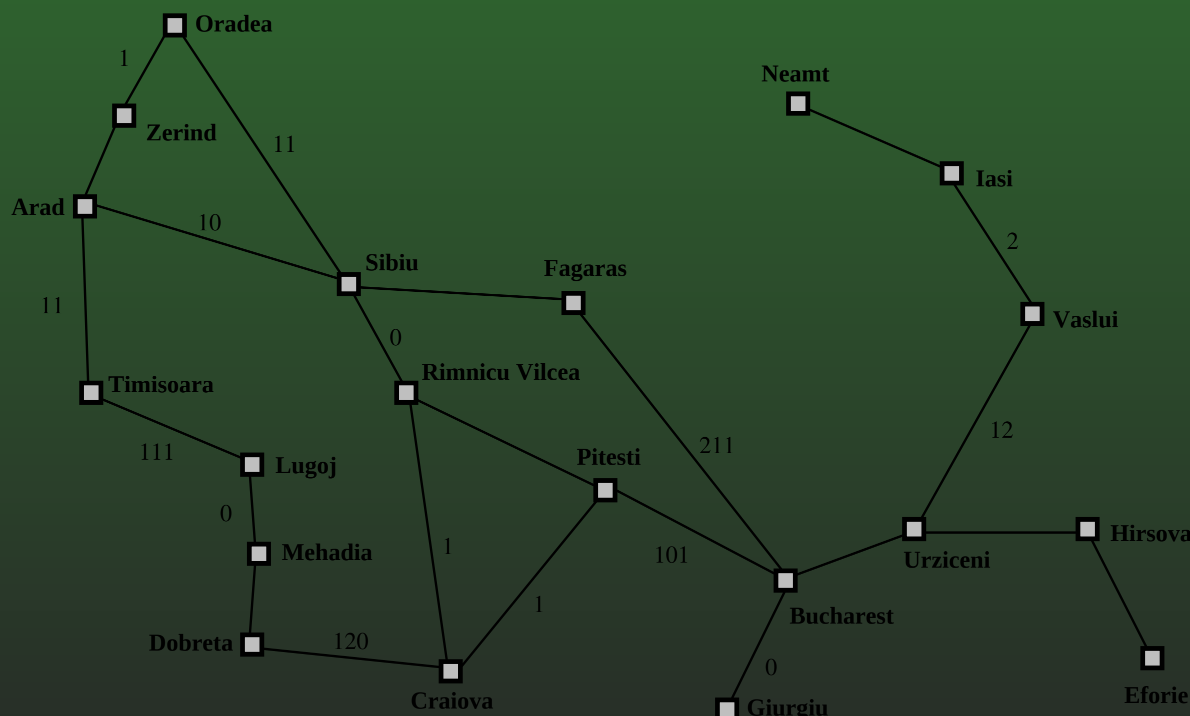
- A Heuristic Function $h(n)$ is an estimate of how much it would cost to get to the solution from node n
- $h(n)$ is not perfect
  - What could we do if h was perfect
- Example heuristic: Route planning: straight-line distance to the goal
- How could we use a heuristic function as part of best-first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

06-7: 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

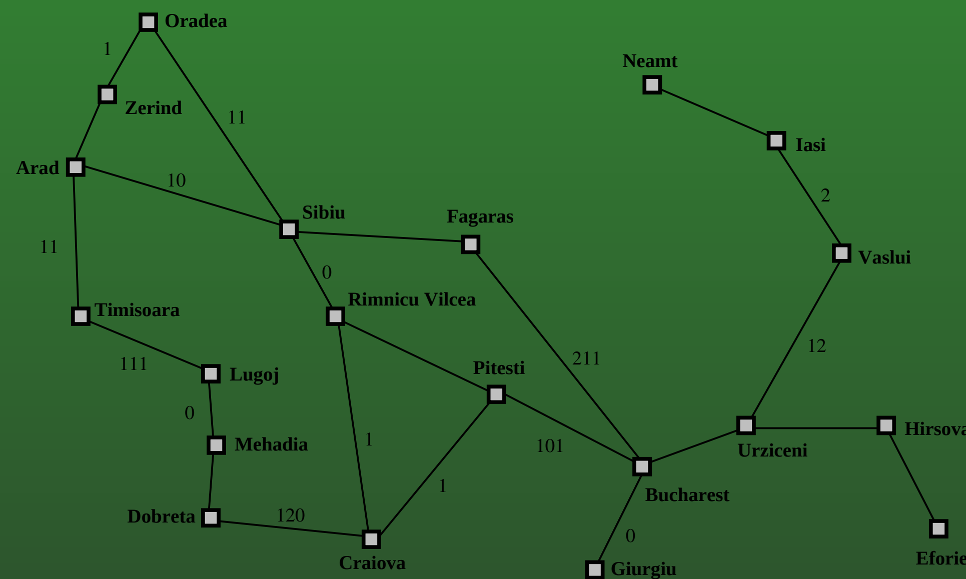
06-8: Greedy Search Example

Arad		**Mehadia**	21
Bucharest	0	**Neamt**	2
Craiova	10	**Oradea**	0
Dobreta	22	**Pitesti**	100
Eforie	11	**Rimnicu Vilcea**	1
Fagaras	1	**Sibiu**	2
Giurgiu		**Timisoara**	2
Hirsova	11	**Urziceni**	0
Iasi	22	**Vaslui**	1
Lugoj	2	**Zerind**	



06-9: Greedy Search Example

Arad		**Mehadia**	21
Bucharest	0	**Neamt**	2
Craiova	10	**Oradea**	0
Dobreta	22	**Pitesti**	100
Eforie	11	**Rimnicu Vilcea**	1
Fagaras	1	**Sibiu**	2
Giurgiu		**Timisoara**	2
Hirsova	11	**Urziceni**	0
Iasi	22	**Vaslui**	1
Lugoj	2	**Zerind**	



(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 → S → F → B

Optimal: A → S → RV → P → B

06-10: 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

06-11: Greedy Search Problems

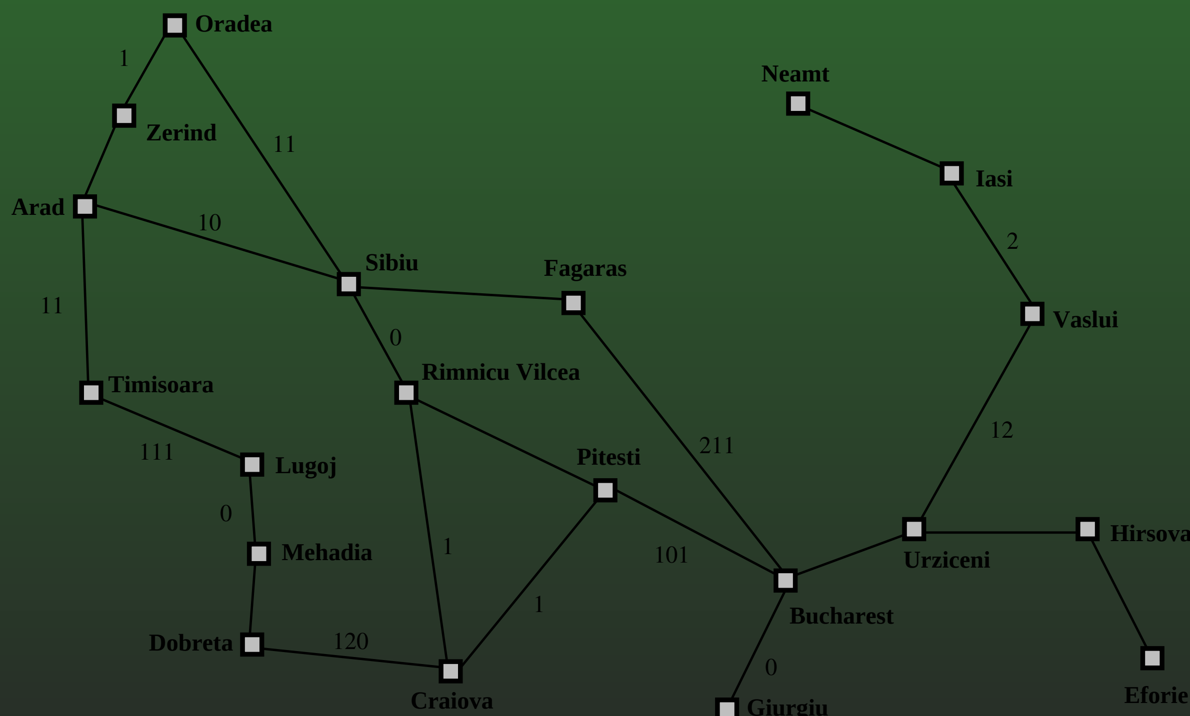
- Greedy search has similar strengths / weaknesses to DFS
  - Expands a linear number of nodes
  - Not optimal
  - Not even necessarily complete (depending upon the heuristic function)
- What are the flaws of greedy search?
- How could we fix them?

06-12: 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).

06-13: A* Search Example

Arad		**Mehadia**	21
Bucharest	0	**Neamt**	2
Craiova	10	**Oradea**	0
Dobreta	22	**Pitesti**	100
Eforie	11	**Rimnicu Vilcea**	1
Fagaras	1	**Sibiu**	2
Giurgiu		**Timisoara**	2
Hirsova	11	**Urziceni**	0
Iasi	22	**Vaslui**	1
Lugoj	2	**Zerind**	



06-14: 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

06-15: 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

06-16: 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

06-17: 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.

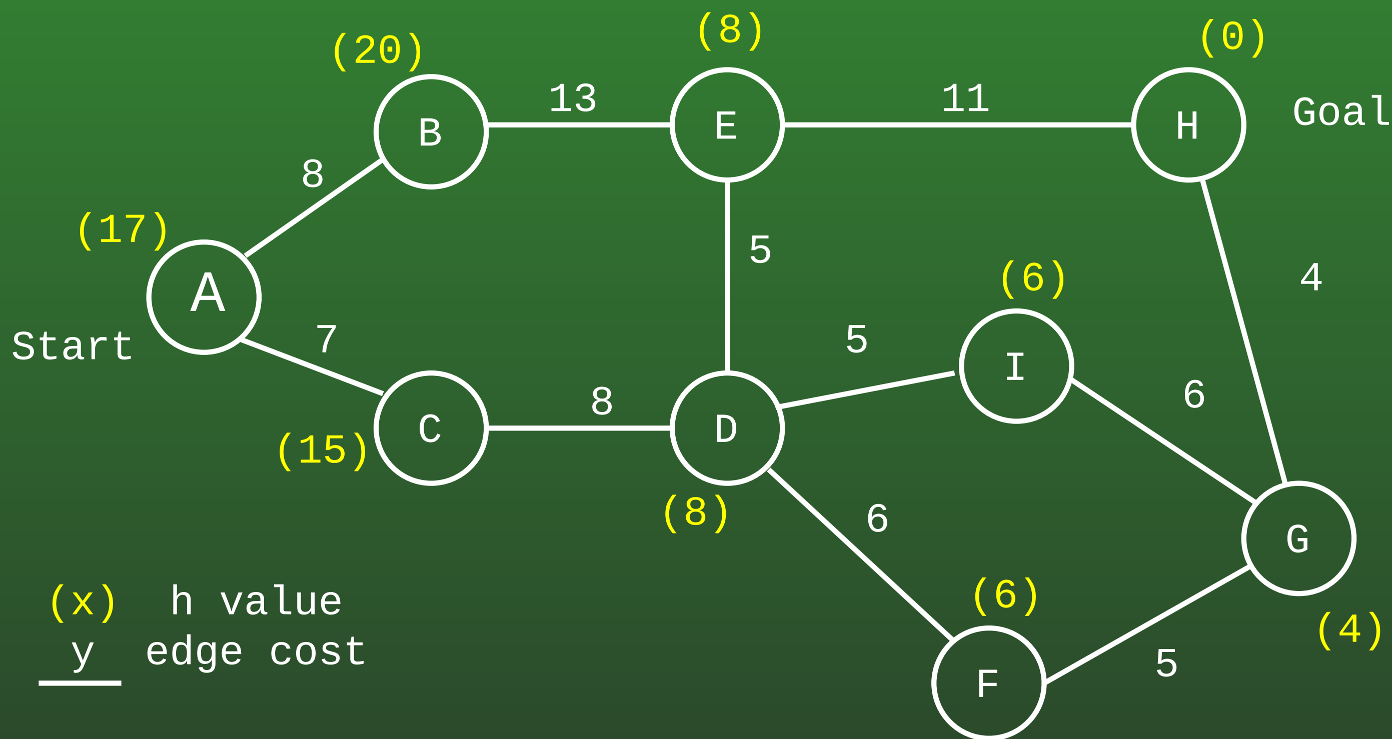
06-18: Optimality of A*

- 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.

06-19: Optimality of A*

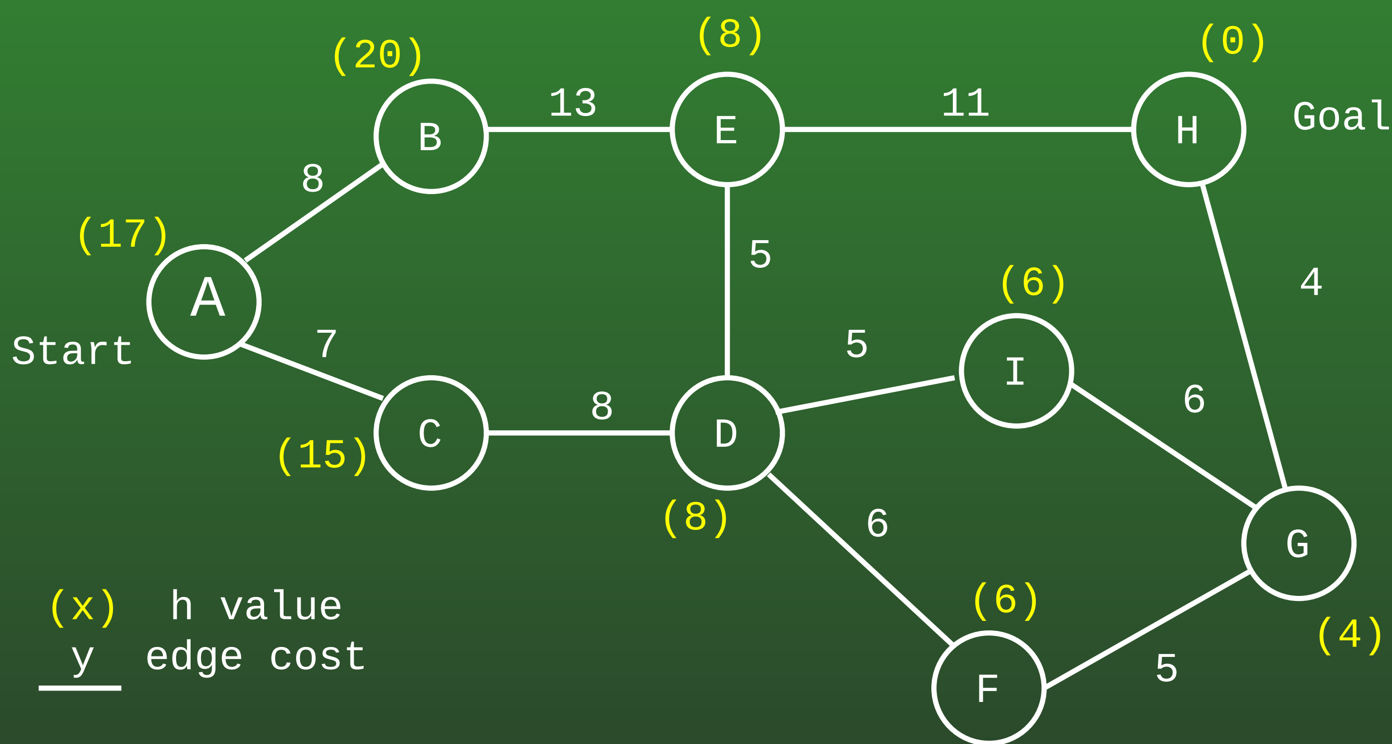
- SLD is monotonic. (In general, it's hard to find realistic heuristics that are admissible but not monotonic).
- Corollary: If h is monotonic, then f is nondecreasing as we expand the search tree.
- Alternative proof of optimality.
- Notice also that UCS is A* with $h(n) = 0$
- A* is also *optimally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

06-20: A* Example II



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

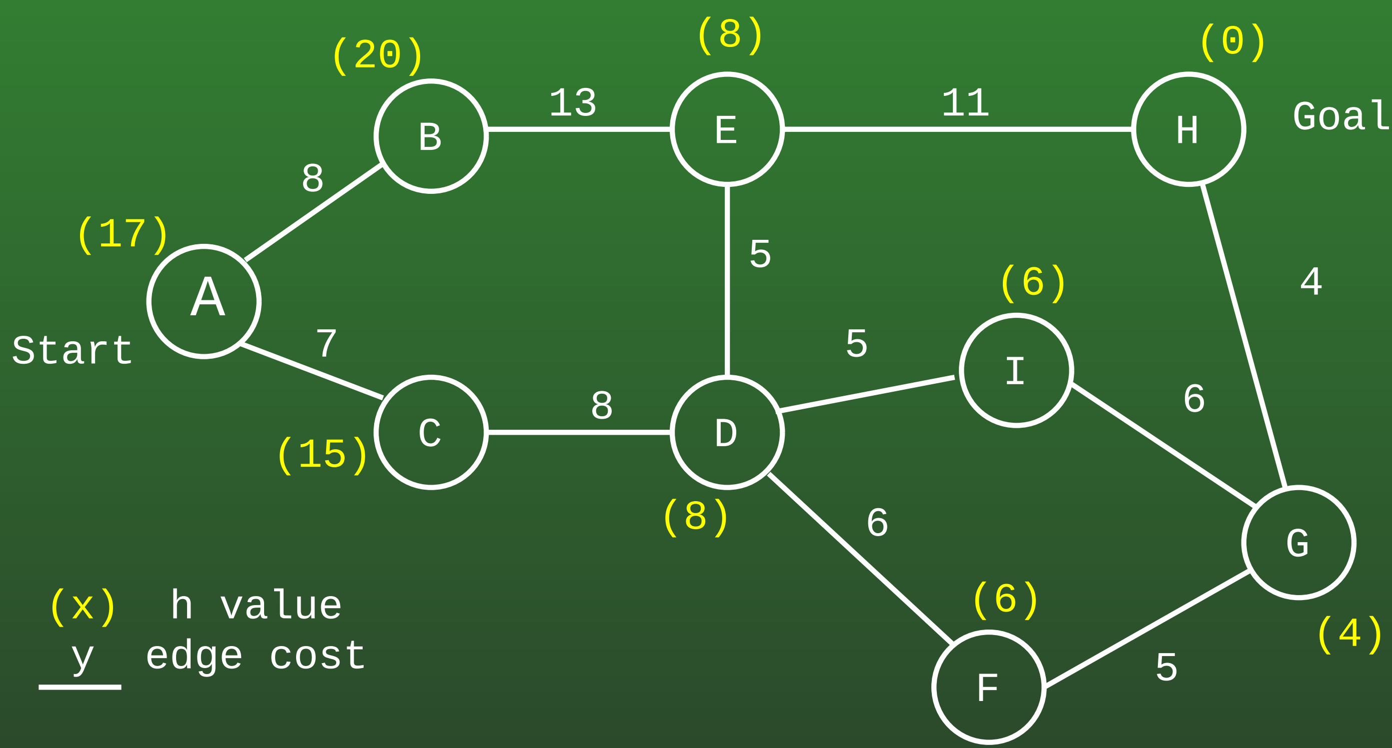
06-21: A* Example II



Node: Queue :

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

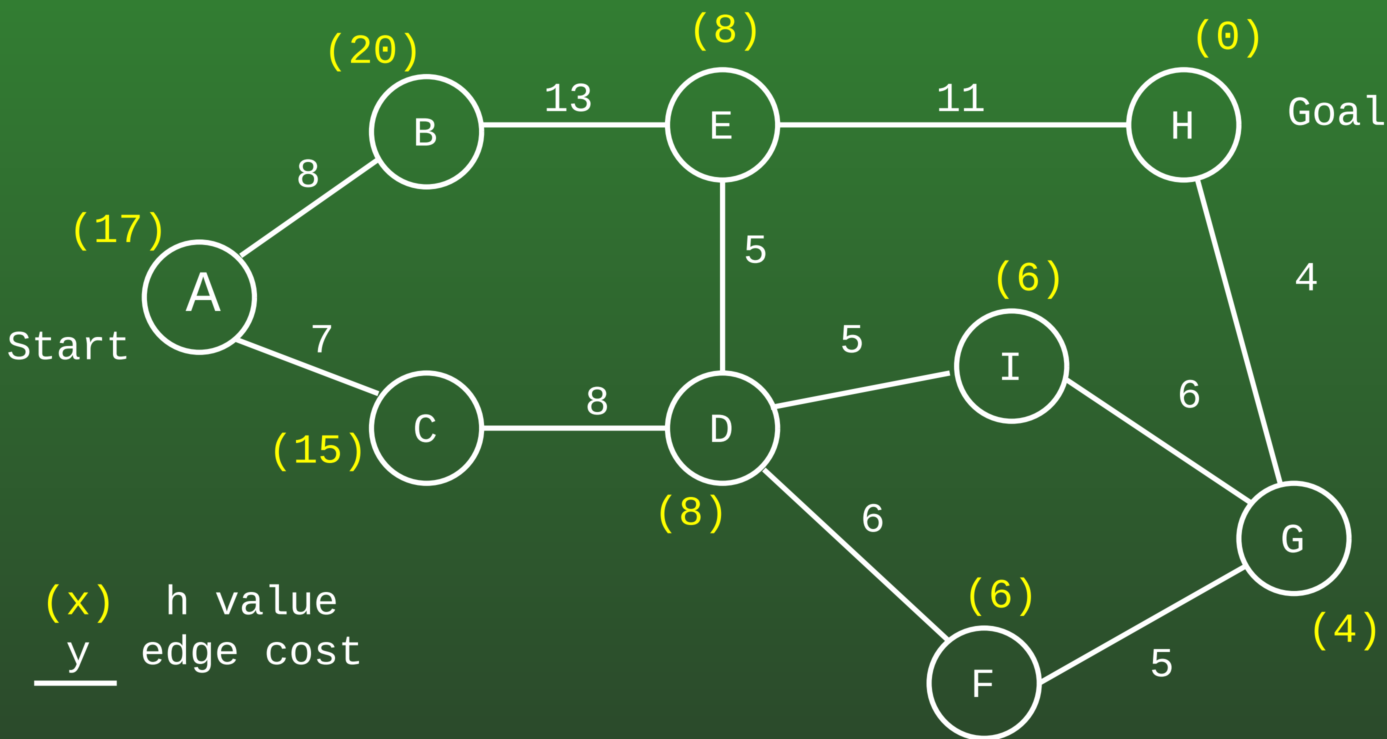
06-22: A* Example II



Node: Queue :

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

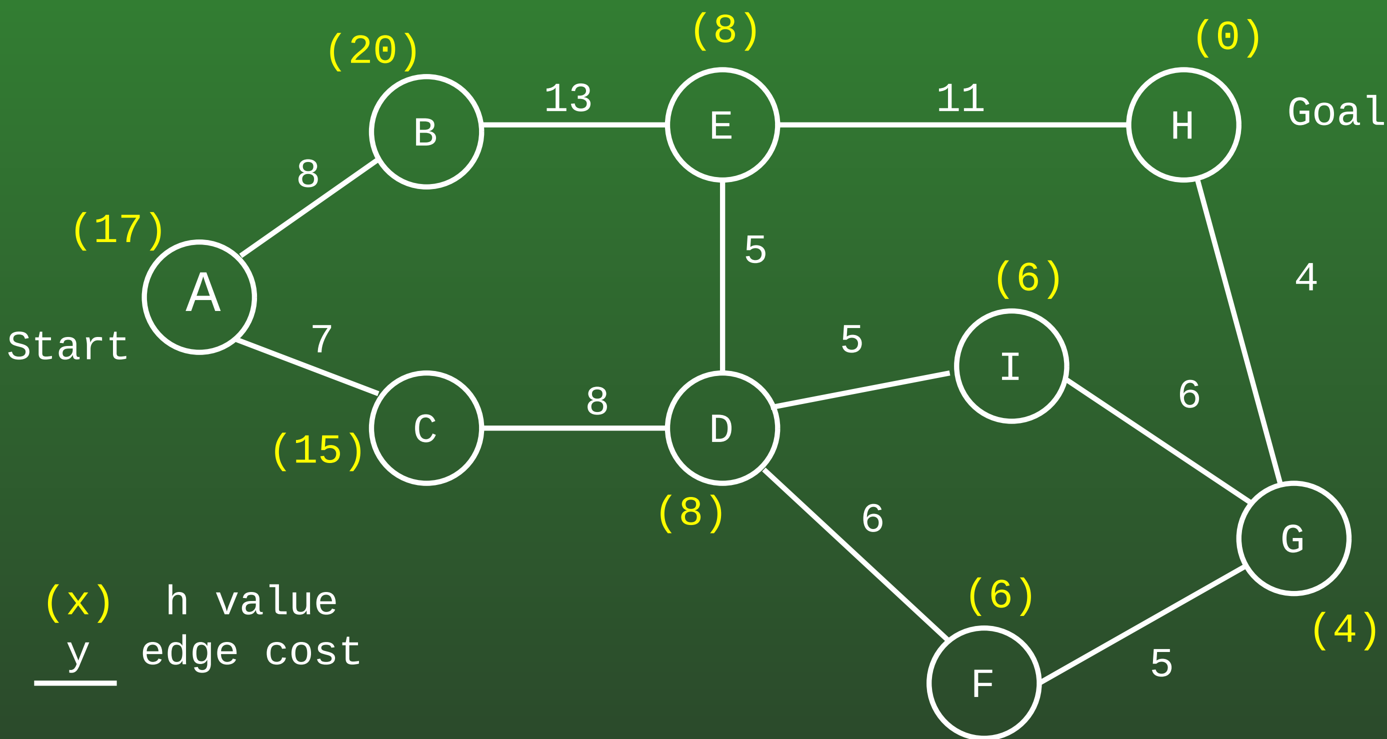
06-23: A* Example II



Node: Queue :

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

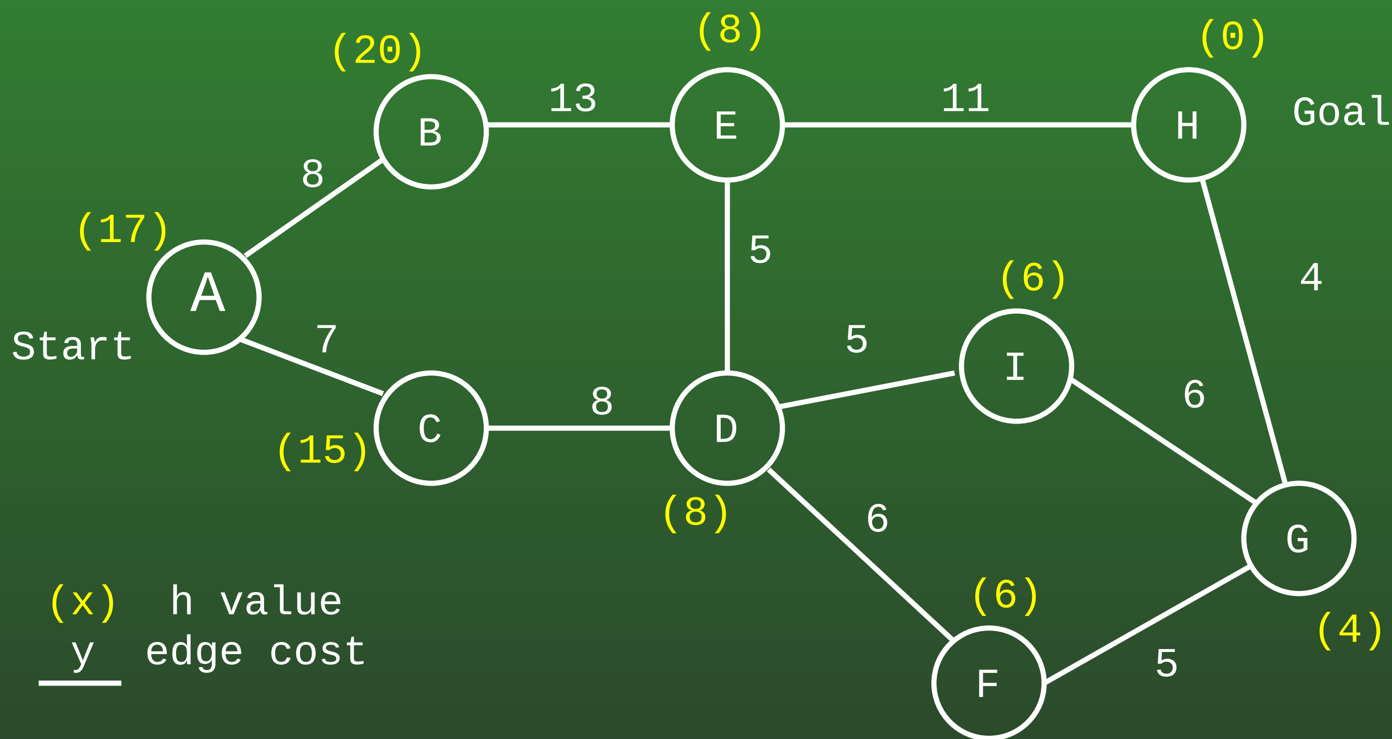
A* Example II



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)]

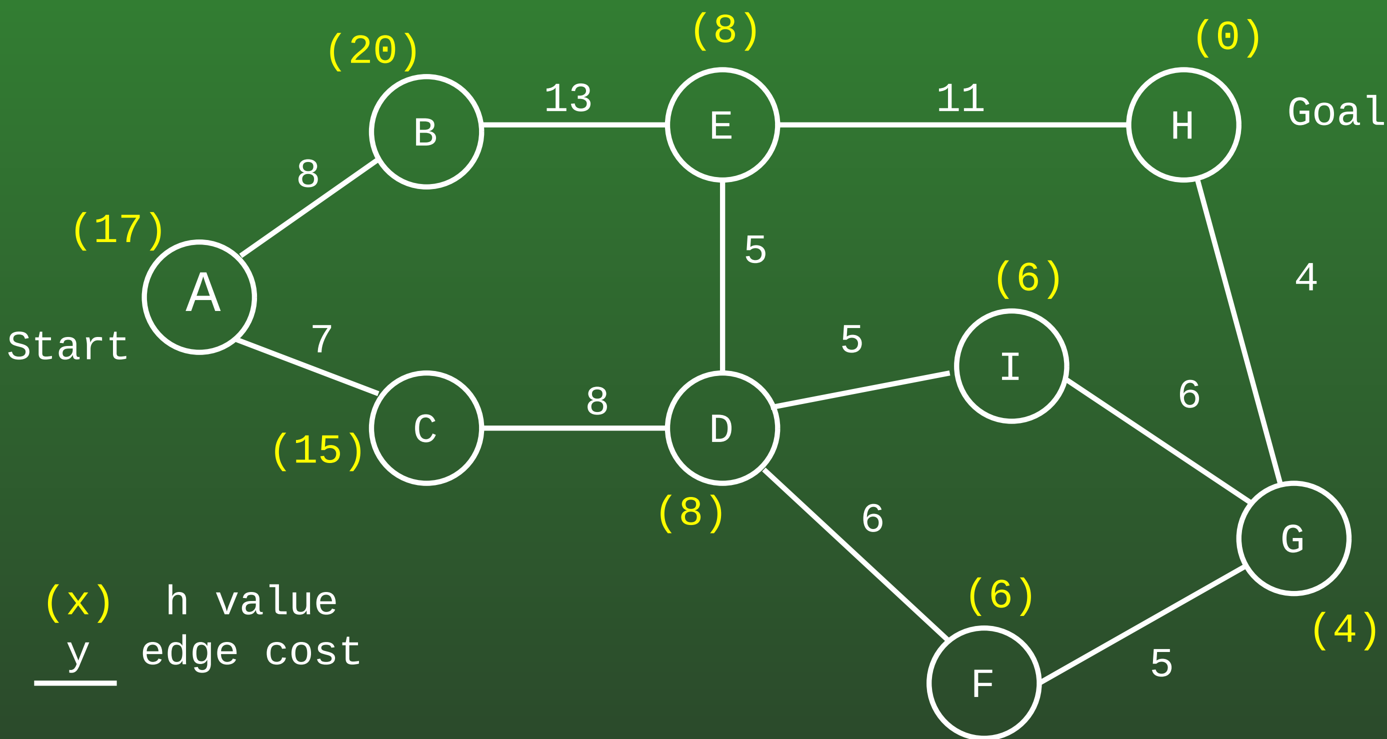
06-25: A* Example II



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)]

06-26: A* Example II

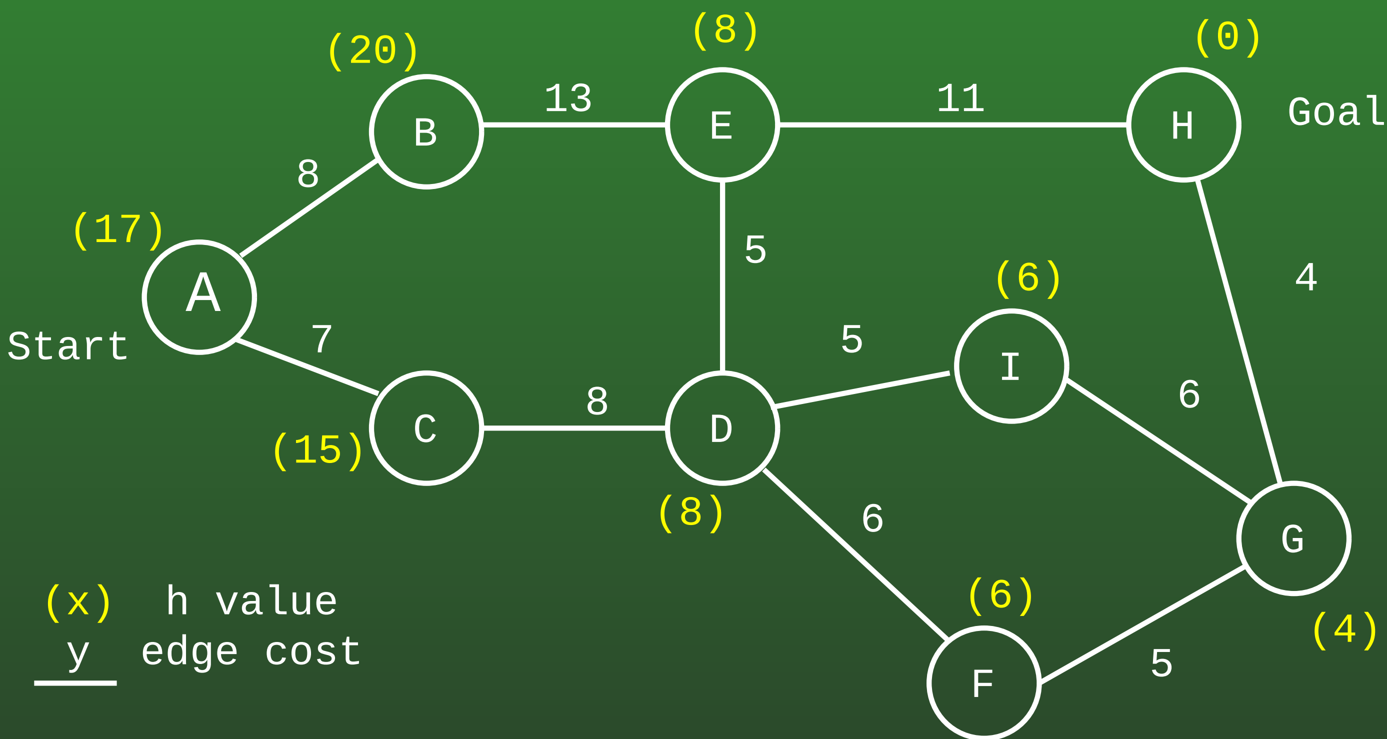


Node: Queue :

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)]

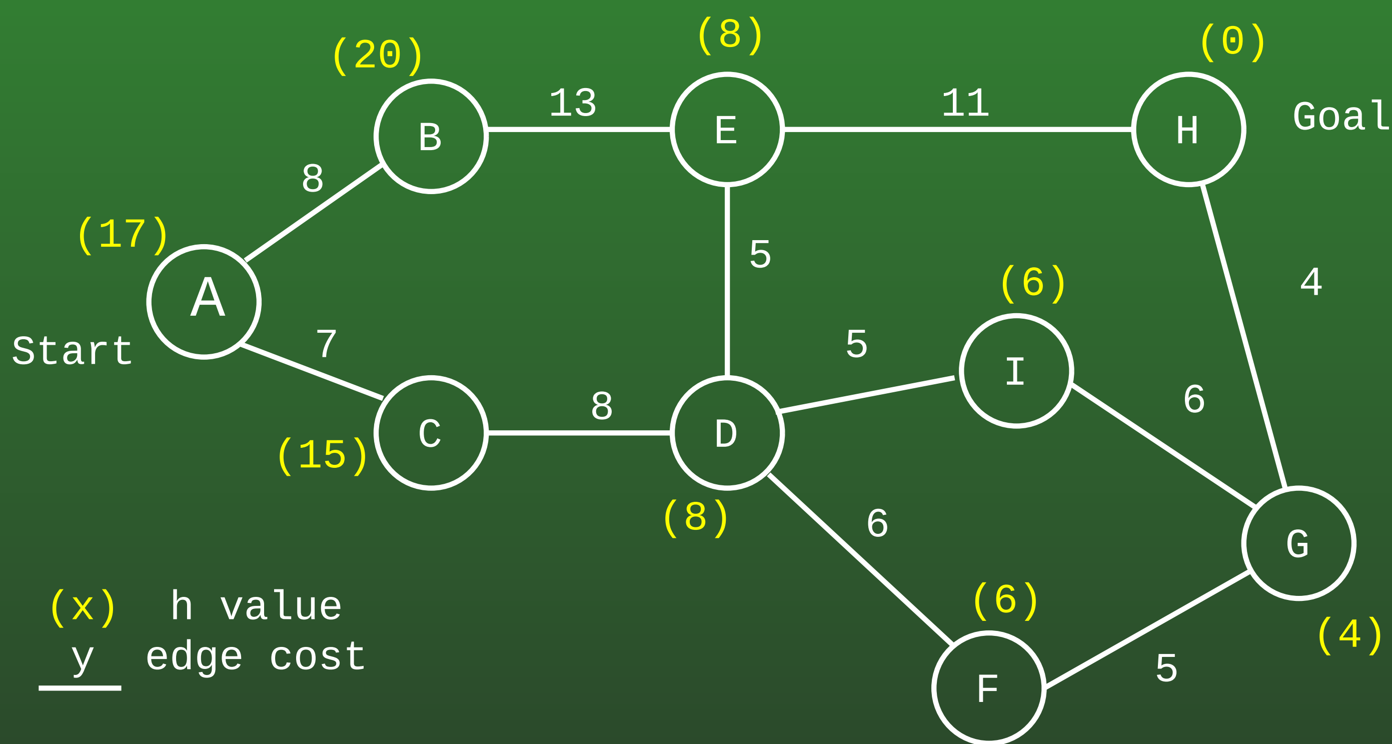
06-27: A* Example II



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)]

06-28: A* Example II



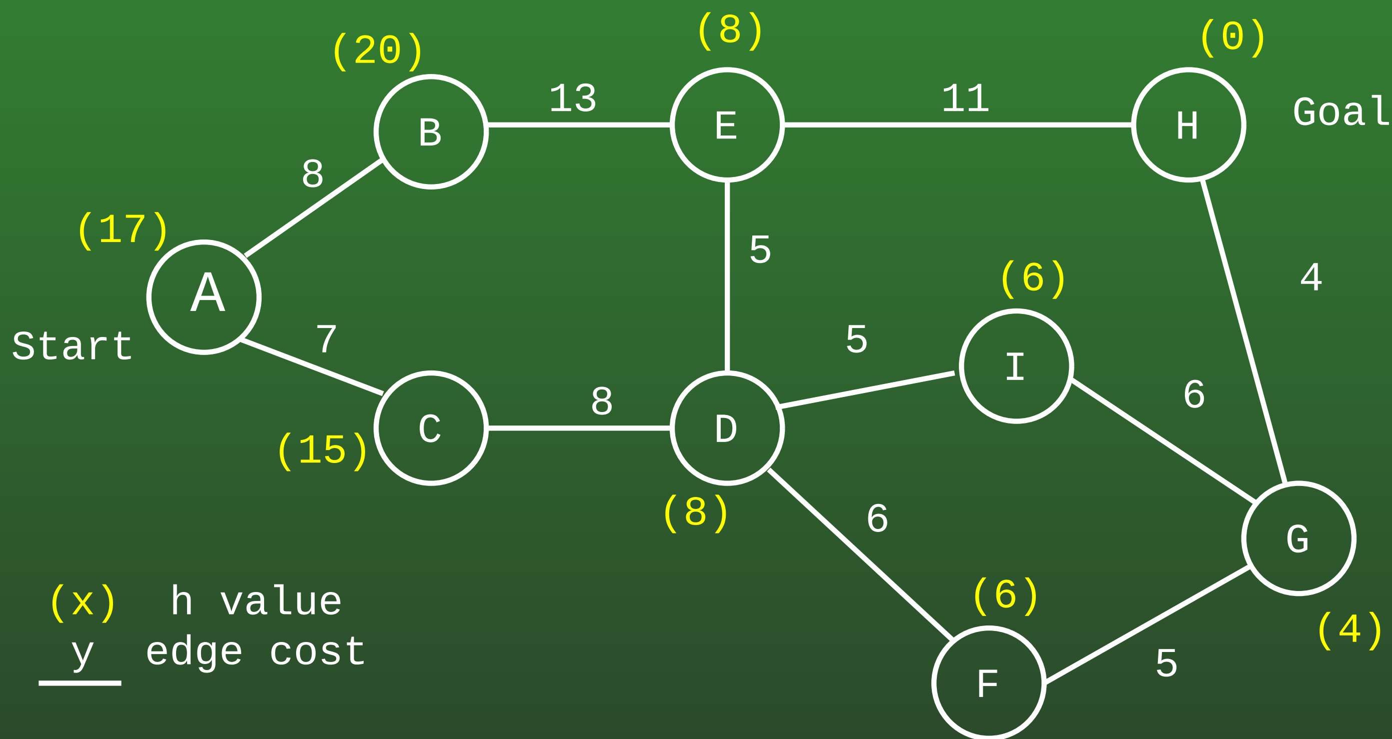
Node: Queue :

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)

06-29: A* Example II



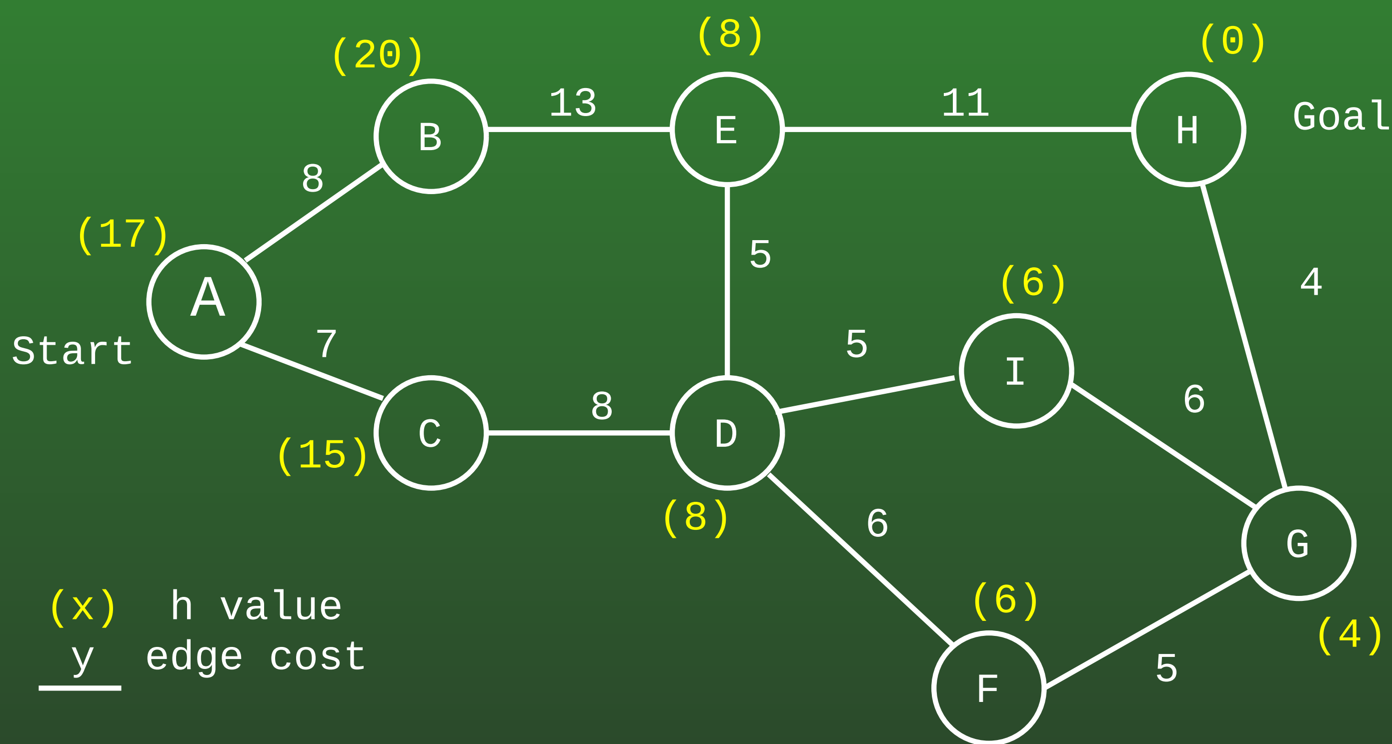
Node: Queue :

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)

06-30: A* Example II

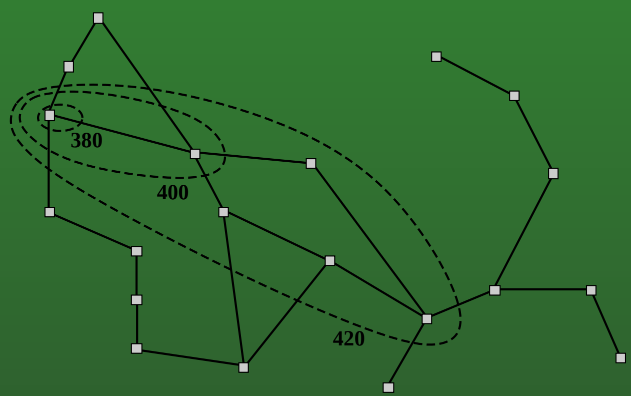


Node: Queue :

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

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

06-31: 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.

06-32: 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

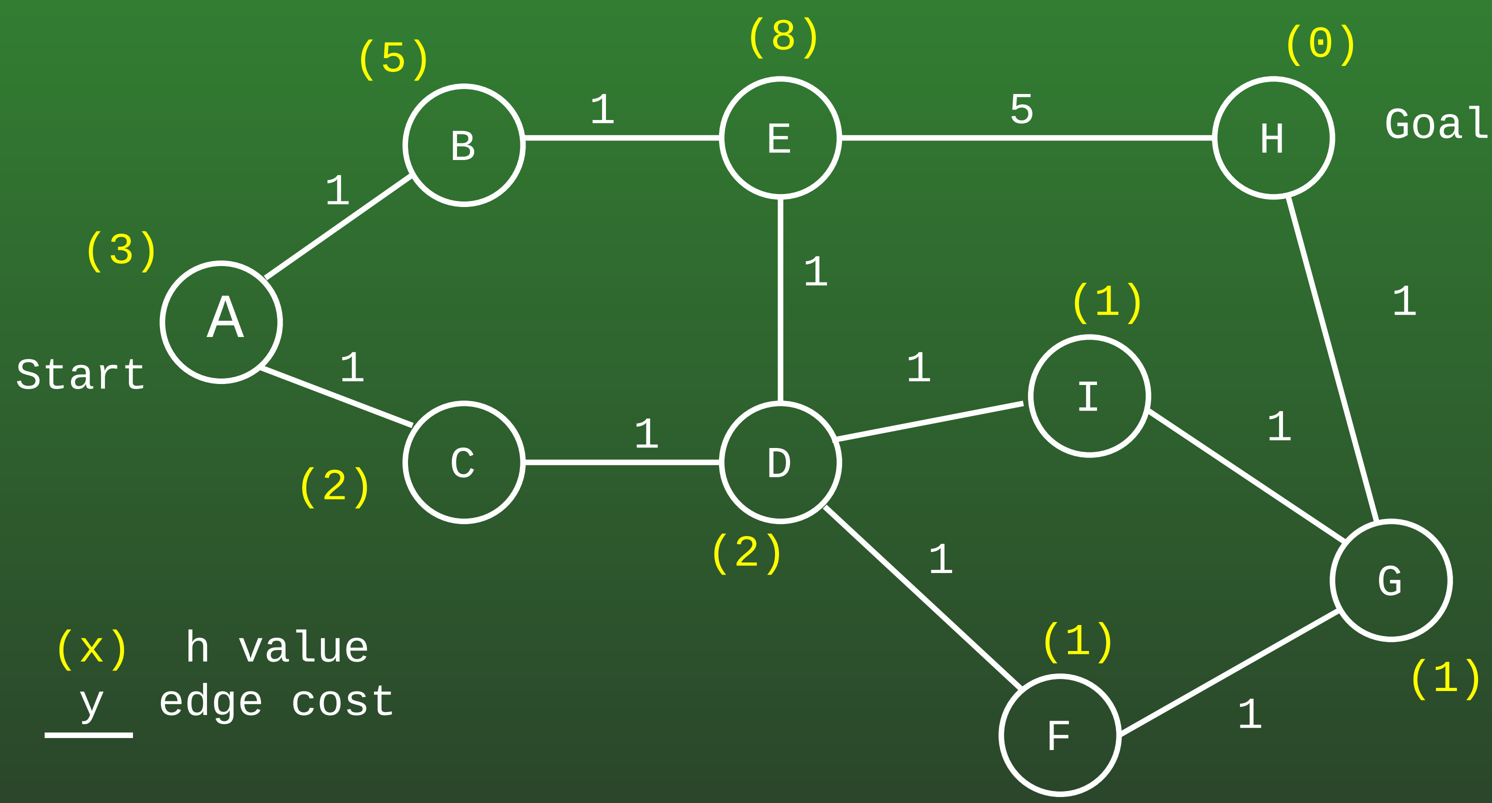
06-33: Iterative Deepening A* (IDA*)

```
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
```

06-34: Iterative Deepening A* (IDA*)

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

06-35: IDA* Example



06-36: 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?

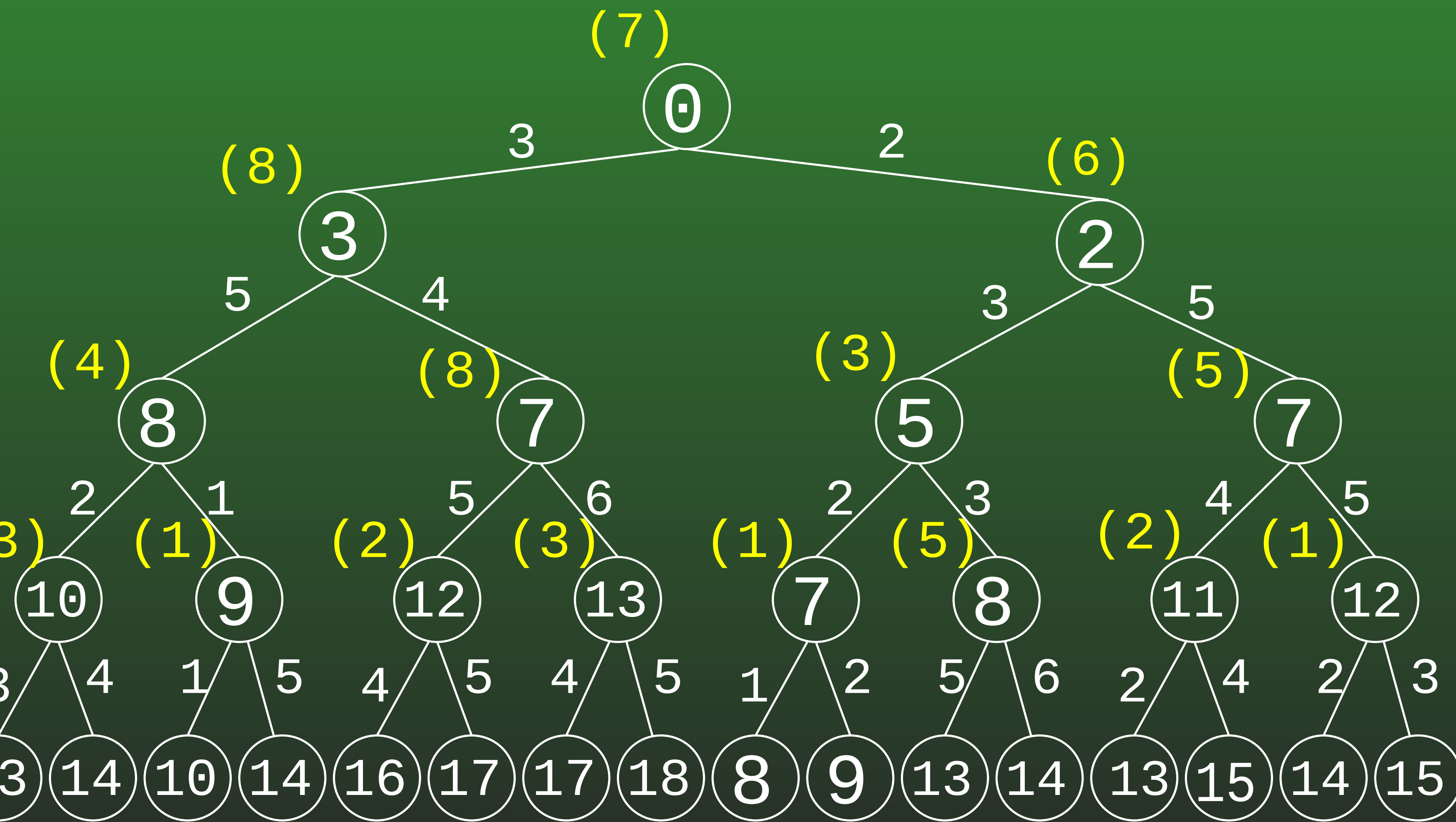
06-37: SMA*

- 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

06-38: DFB&B

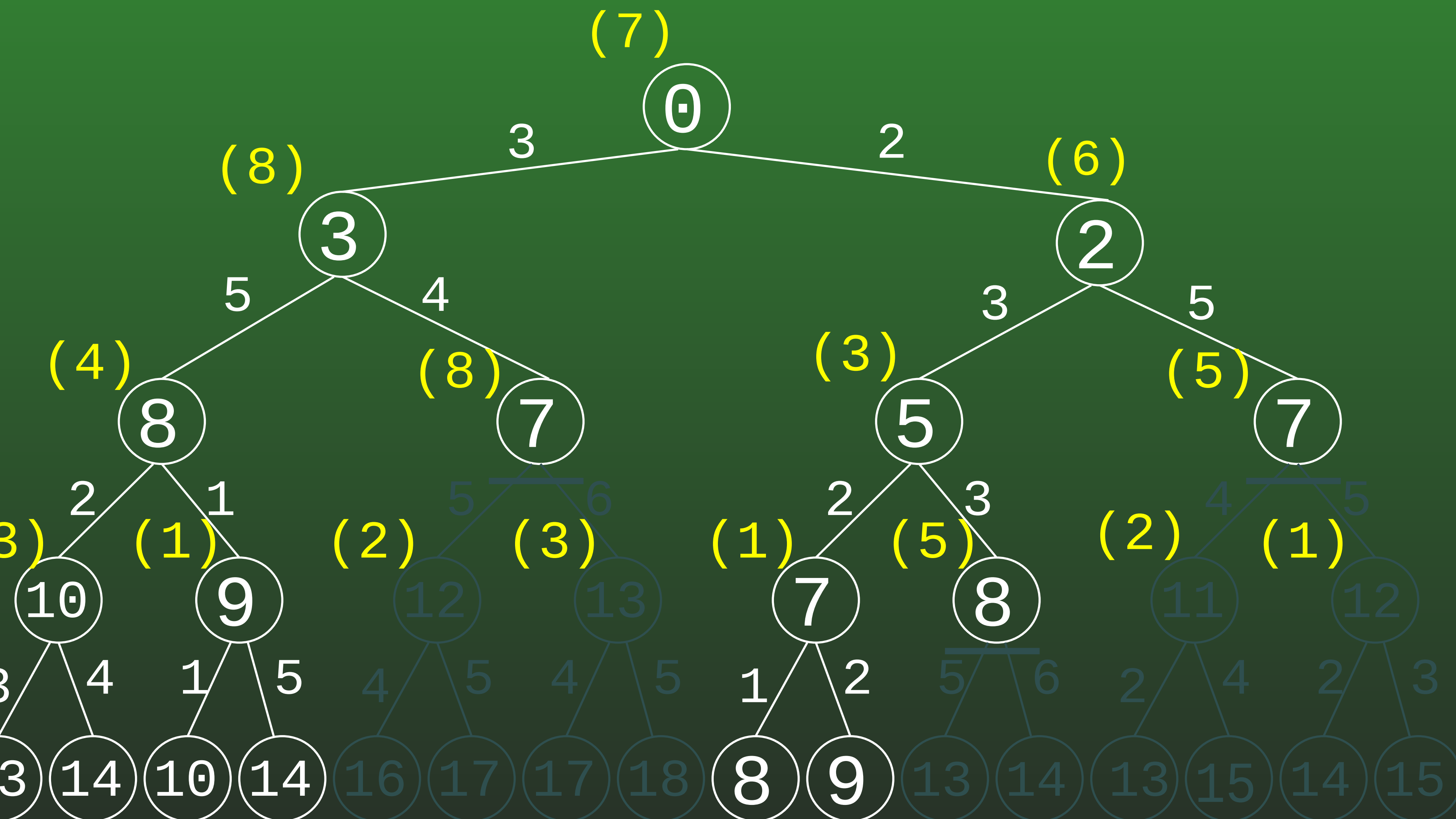
- Depth-First Branch and Bound
  - Run f-limited DFS, with limit set to infinity
  - When a goal is found, don't stop – record it, and set limit to the goal depth
  - Keep going until all branches are searched or pruned.
- We will use something similar in 2-player games
- (DFB&B not in the text)

06-39: 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
(1)
(2)
(3)
(1)
(5)
(2)
(1)
10
9
12
13
7
8
11
12
4
1
5
4
5
4
5
1
2
5
6
2
4
2
3
14
10
14
16
17
17
18
8
9
13
14
13
15
14
15

06-40: 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
4
1
5
4
5
4
5
1
2
5
6
2
4
2
3
14
10
14
16
17
17
18
8
9
13
14
13
15
14
15

06-41: 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?

06-42: 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

06-43: 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.

06-45: 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?

06-46: 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.

06-47: 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

06-48: 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

06-49: Finding a heuristic

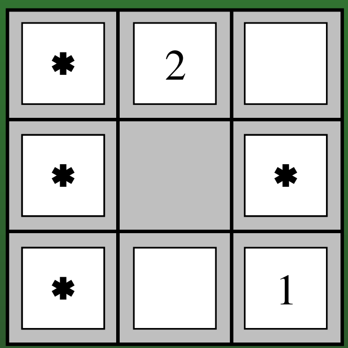
- So how do we find a good heuristic?
- Solve a relaxed version of the problem.
  - Traveling Salesman
    - Connected graph
    - Each node has 2 neighbors
  - Minimum Cost Spanning Tree Heuristic

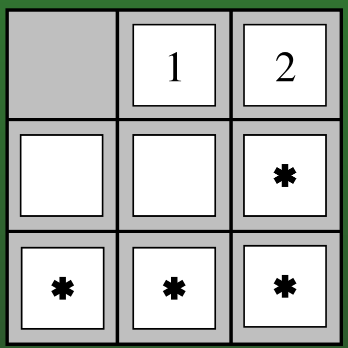
Finding a heuristic

- 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)

06-51: Finding a heuristic

- Using subproblems





06-52: 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?

06-53: 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?
  - $h(n) = \max(h_1(n), h_2(n), \ldots h_k(n))$

06-54: Summary

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