Introduction to Dynamic Programming

Basic Concepts and Selected Examples

Deep Dive Coding

30 November, 2018

Inception

- The term “dynamic programming” was invented by Richard Bellman at The RAND Corporation in the early 1950s to describe a problem-solving approach for multistage decision problems.
- In this context, “programming” does not necessarily refer to writing software, but to the solution of logistics or scheduling problems by finding optimal policies, plans, or programs.
- “Dynamic” was used not only to evoke the multistage (and often time-based) decisions involved, but also to sound impressive—without appearing too mathematical in nature.

Bellman's *Principle of Optimality*

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

- Richard Bellman, 1957

In other words, if a system moves from state $\boldsymbol{A}$ to state $\boldsymbol{C}$, and the optimal sequence of transitions to from $\boldsymbol{A}$ to $\boldsymbol{C}$ includes state $\boldsymbol{B}$, then that optimal sequence must include the optimal subsequence of transitions from $\boldsymbol{B}$ to $\boldsymbol{C}$.

Optimal Substructure

- A computational problem has optimal substructure if an optimal solution can be constructed from the optimal solutions of its subproblems.
- This is simply the computer science-oriented re-statement of Bellman's *principle of optimality*.
- Many computational problems (and problem-solving approaches), not limited to dynamic programming, take advantage of optimal substructure. Take quick sort, for example: The sequence $\{s_1, \ldots, s_n\}$ is in sorted order if (and only if) the sequence $\{s_1, \ldots, s_m\}$ is in sorted order and the sequence $\{s_m, \ldots, s_n\}$ is also in sorted order; this is the logic behind the partitioning of a sequence in quick sort.

Example: Fibonacci sequence

- The Fibonacci numbers are the population sizes after each generation in the “Fibonacci’s rabbits” thought experiment:
  - Each newborn pair (male and female) of rabbits takes one generation to reach sexual maturity—i.e. to be capable of breeding.
  - In each generation, each breeding pair produces 1 new pair of rabbits, and no rabbits die.
  - Start with 1 newborn pair.
  - f_n = number of pairs after n generations = ?

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Immature	1	0	1	1	2	3	5	8	13	21
Mature	0	1	1	2	3	5	8	13	21	34
Total	*1*	*1*	*2*	*3*	*5*	*8*	*13*	*21*	*34*	*55*

Overlapping Subproblems

- A computing problem has *overlapping subproblems* if it can be broken down into subproblems that must (at least conceptually) be re-solved multiple times.
- For example, in the definition of the Fibonacci sequence as the mathematical recurrence $f_n = f_{n-1} + f_{n-2}$, computing f_n requires that we compute both f_{n-1} and f_{n-2}; however, computing f_{n-1} also requires that we compute f_{n-2} and f_{n-3} Thus, computing f_{n-2} (as well as $f_{n-3,}$, f_{n-4}, etc.) is an overlapping subproblem.
- How can we avoid having to repeat these computations?

Example: Fibonacci in JavaScript

Naïve approach

```
function fibonacci(n) {
  if (n <= 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return fibonacci(n - 1) +
      fibonacci(n – 2);
}
```

Memoized approach

```
var memo = [];

function fibonacci(n) {
  if (n <= 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  if (memo[n] === undefined) {
    memo[n] = fibonacci(n - 1) +
        fibonacci(n – 2);
  }
  return memo[n];
}
```

Overlapping Subproblems (cont.)

- A naïve recursive approach to a problem with overlapping subproblems will often have exponential time complexity, caused by computing the overlapping subproblems repeatedly. Dynamic programming incorporates techniques to address this, e.g.
  - Bottom-up solution
  - Memoization
- If a computational problem can be expressed and implemented with a recursive solution, but the approach involves combining the solutions of non-overlapping subproblems, the strategy is called *divide-and-conquer*, rather than dynamic programming. For example, merge sort and quick sort are divide-and-conquer algorithms, not dynamic programming algorithms.

Example: Finding the shortest path

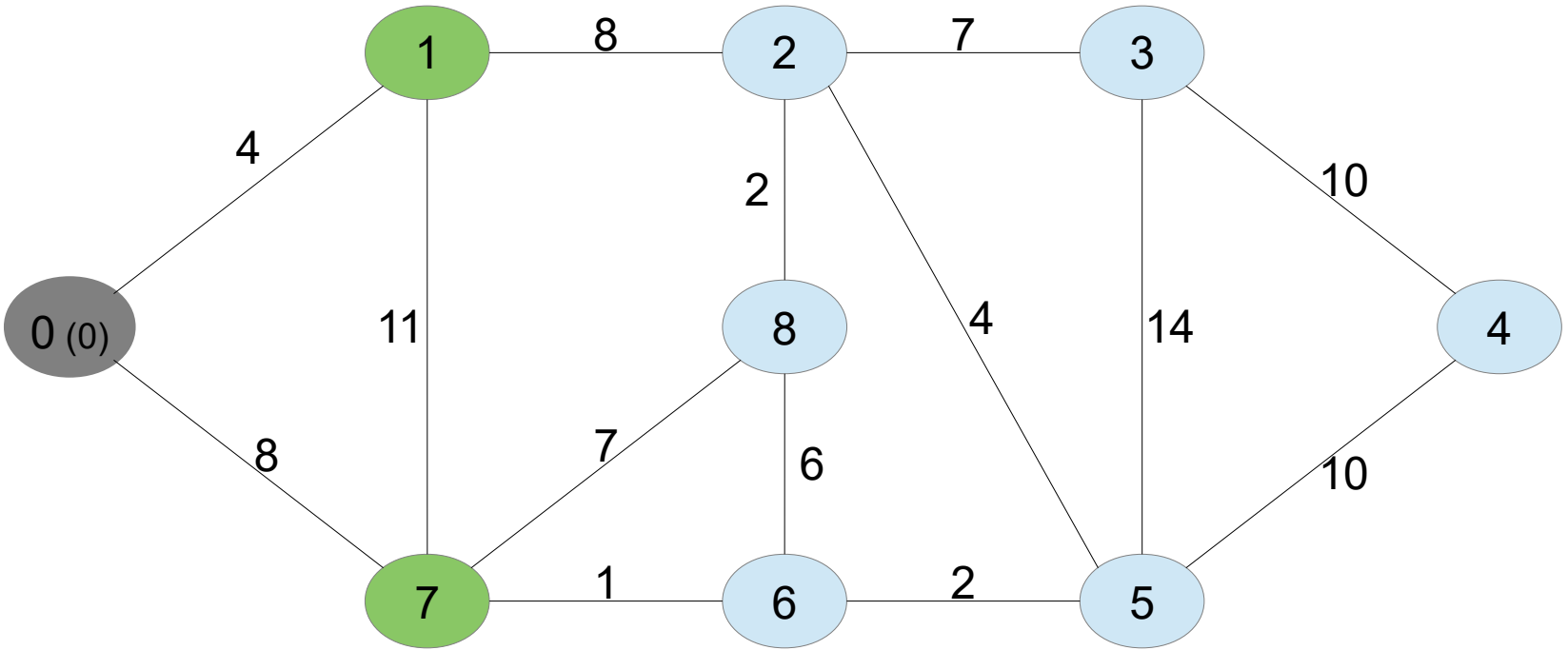
- Finding the shortest (or minimum cost/weight) path between two nodes in a graph can easily be seen to have optimal substructure: If the shortest path from A to C passes through B, then the portion of that path that connects A to B must be the shortest path between those 2 nodes; similarly, the portion of the full path that connects B to C must be the shortest path between those 2.
- It is also the case (though less immediately obvious) that finding the shortest path involves overlapping subproblems, as well.
- Even the brute-force approach of enumerating all possible paths, and selecting the shortest, has overlapping subproblems: When enumerating all paths from A to C that pass through B we must—repeatedly, for each path from A to B—enumerate all paths from B to C (unless, of course, we implement some kind of memoization).

Dijkstra's Algorithm

- In the late 1950s, Edsgar W. Dijkstra invented and published an algorithm for finding the shortest (minimum weight) path between nodes in a weighted graph.
- In Dijsktra's description, the algorithm didn't explicitly use dynamic programming terminology, but it can be seen as a dynamic programming approach. In fact, Dijkstra includes the following equivalent to Bellman's principle of optimality:

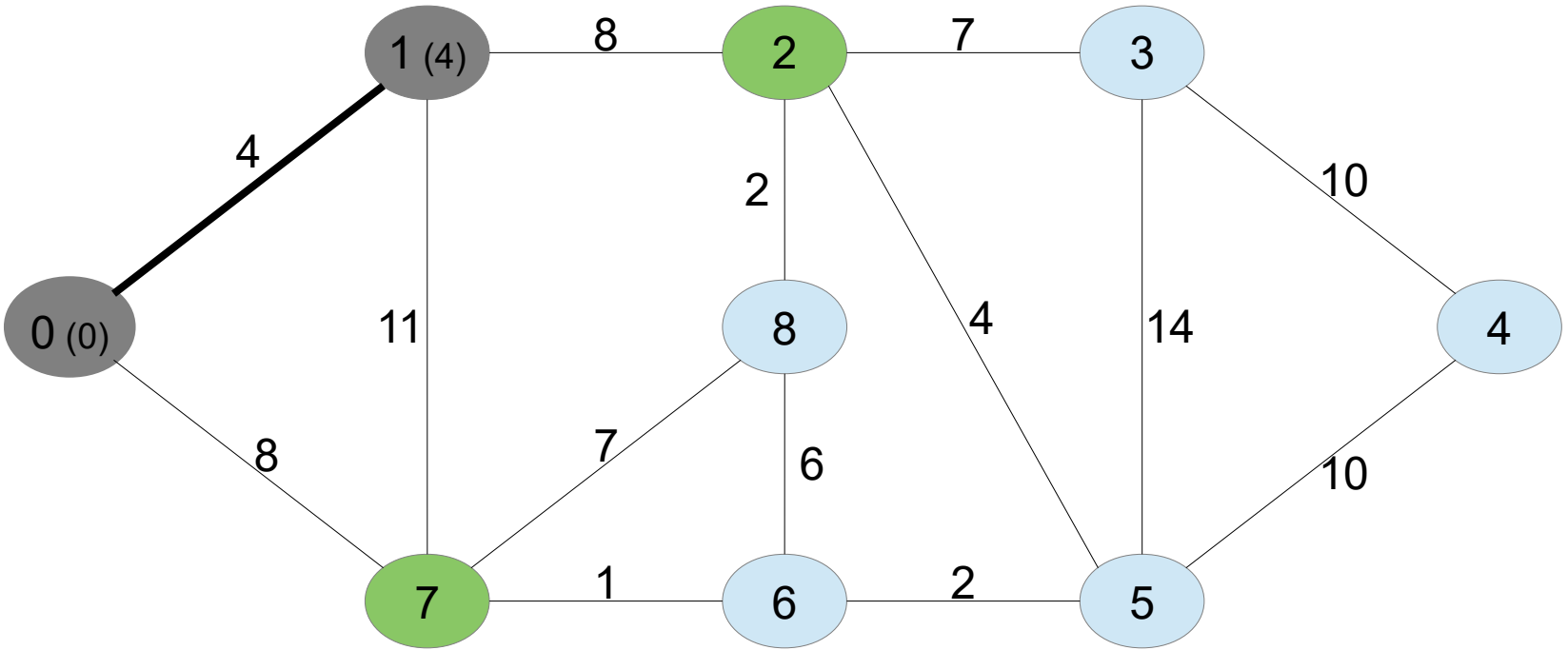
  "We use the fact that, if $\boldsymbol{R}$ is a node on the minimal path from $\boldsymbol{P}$ to $\boldsymbol{Q}$, knowledge of the latter implies the knowledge of the minimal path from $\boldsymbol{P}$ to $\boldsymbol{R}$."
- The algorithm proceeds from the starting node, adding 1 new node to a growing tree of nodes at each iteration. The node added is the one which has a direct connection to at least one node already in the tree, and which has the minimum total path length or cost from the starting node. When the node added is the desired destination node, the process is complete.

Dijkstra's Algorithm: Example

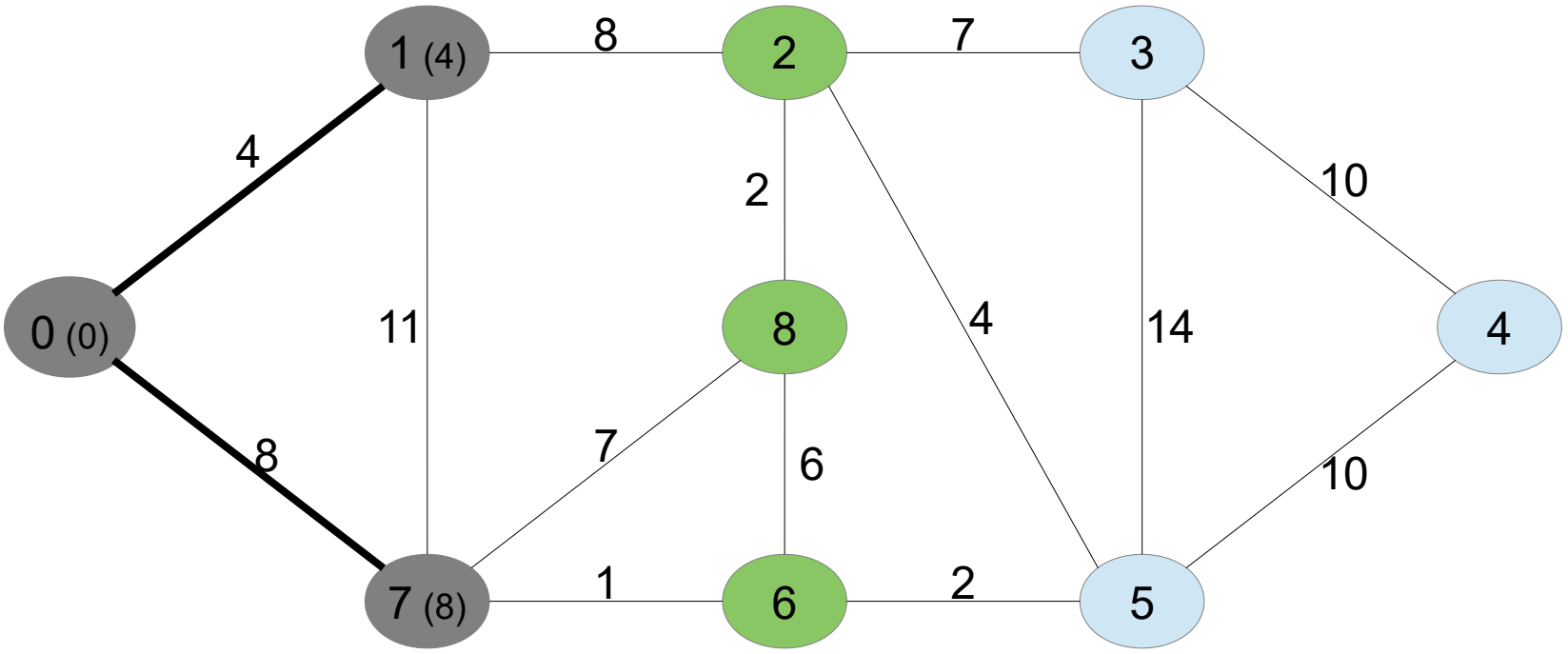


Note: As a node is added to the tree, the total weight of the path from the starting node to the given node is shown in parentheses.

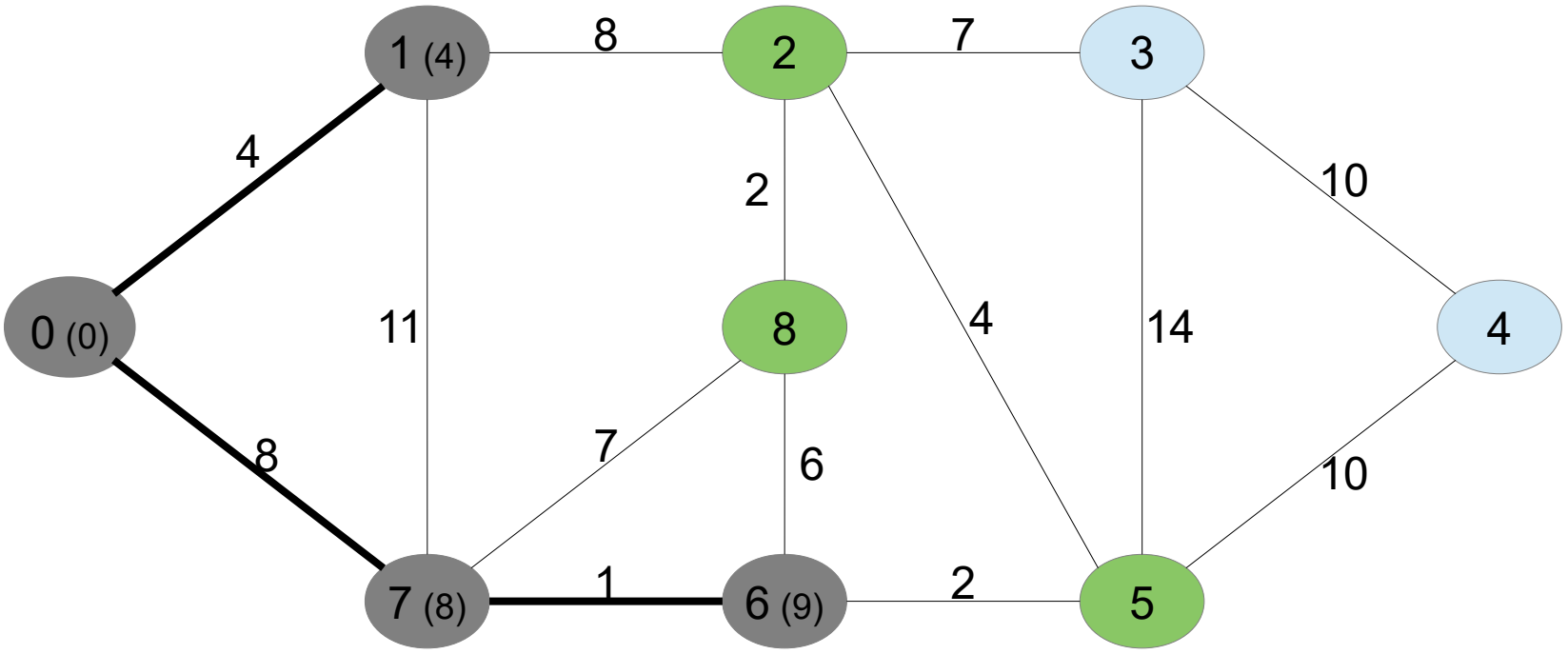
Dijkstra's Algorithm: Example



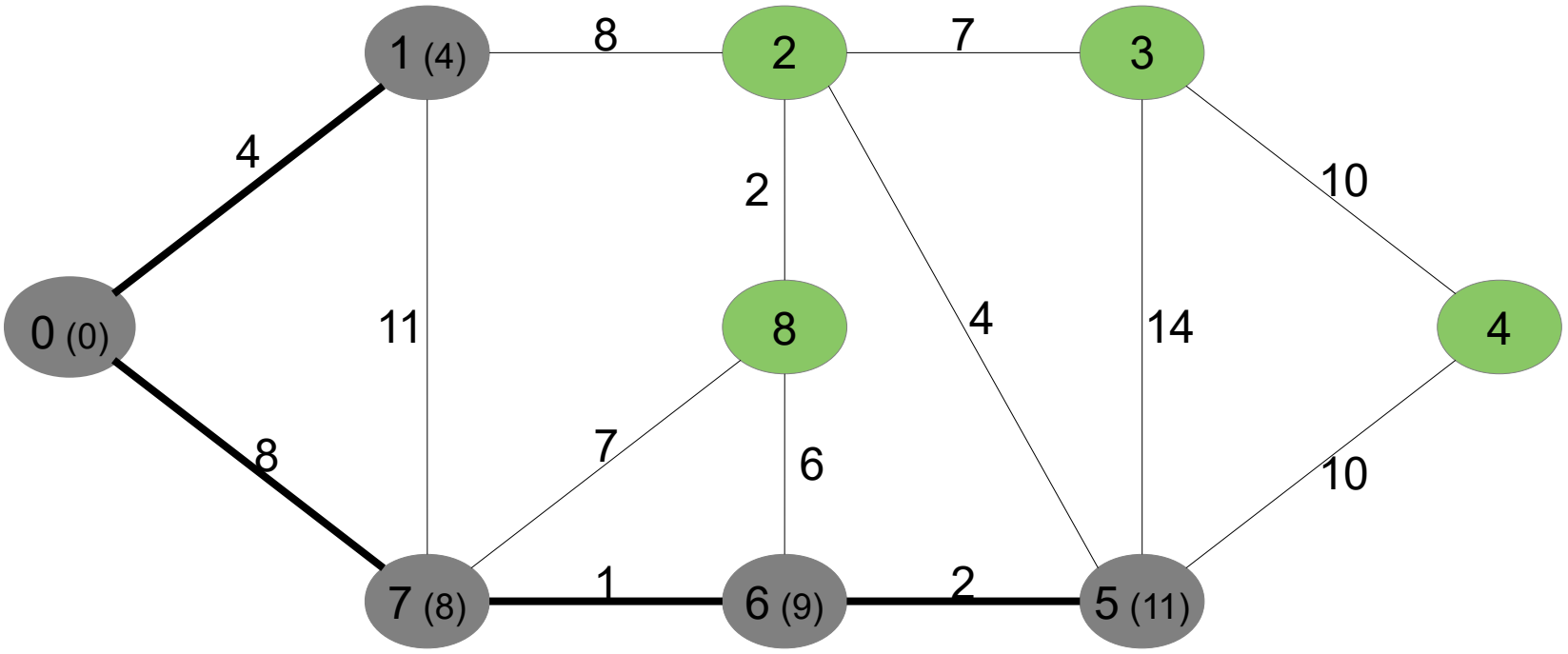
Dijkstra's Algorithm: Example



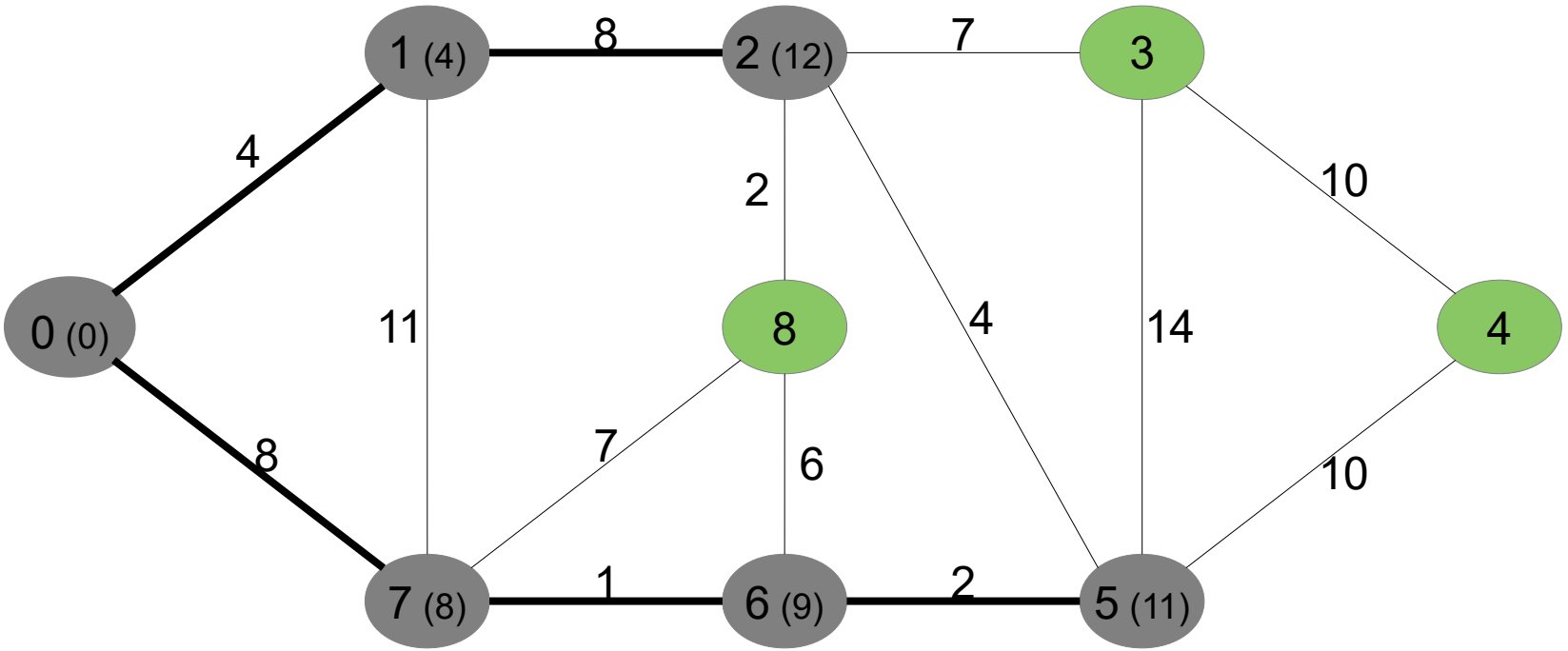
Dijkstra's Algorithm: Example



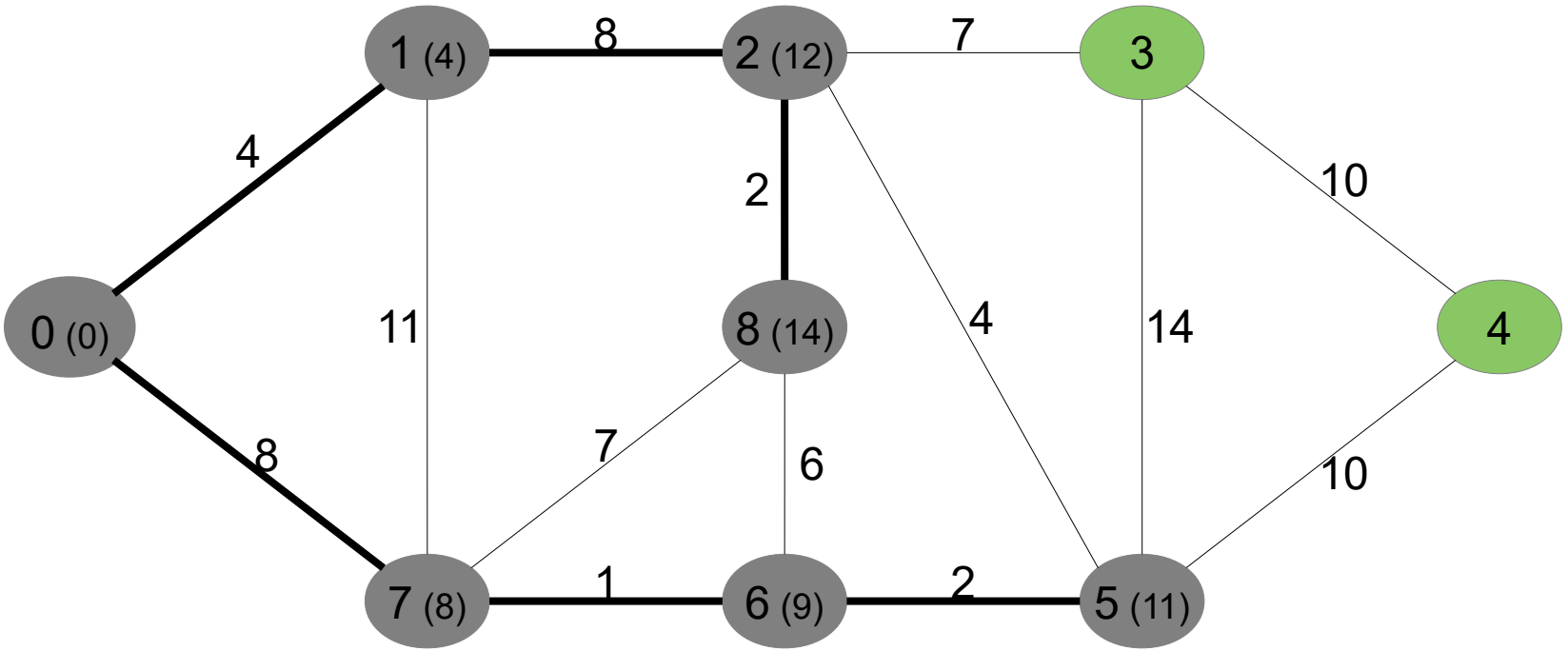
Dijkstra's Algorithm: Example



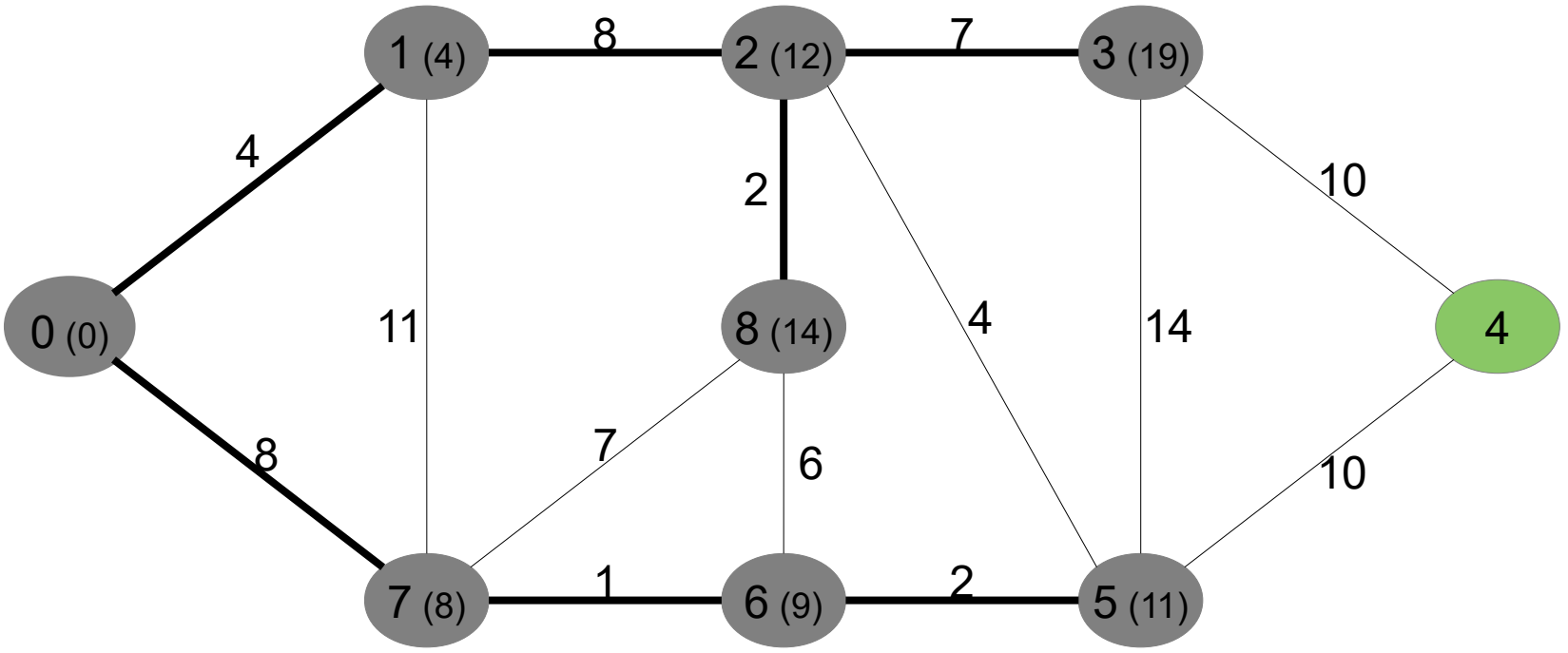
Dijkstra's Algorithm: Example



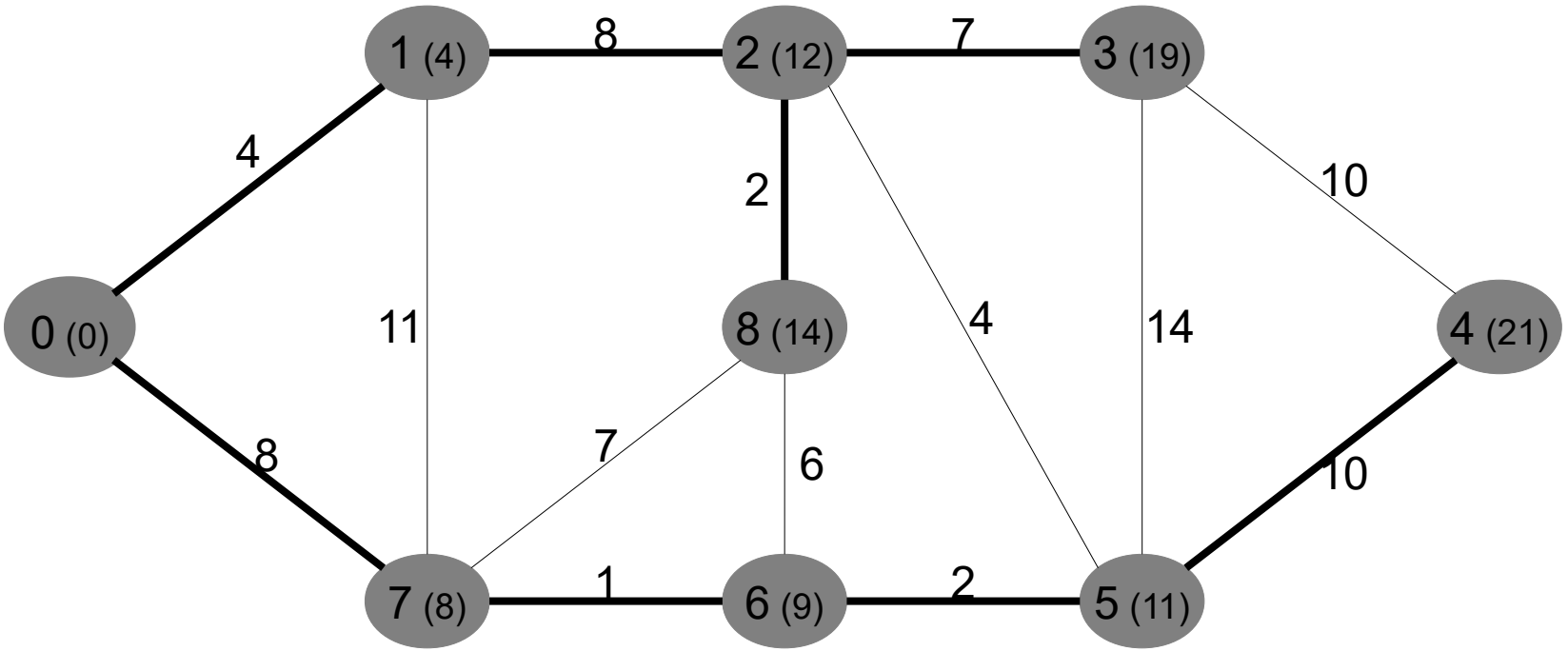
Dijkstra's Algorithm: Example



Dijkstra's Algorithm: Example



Dijkstra's Algorithm: Example



Bellman-Ford Algorithm

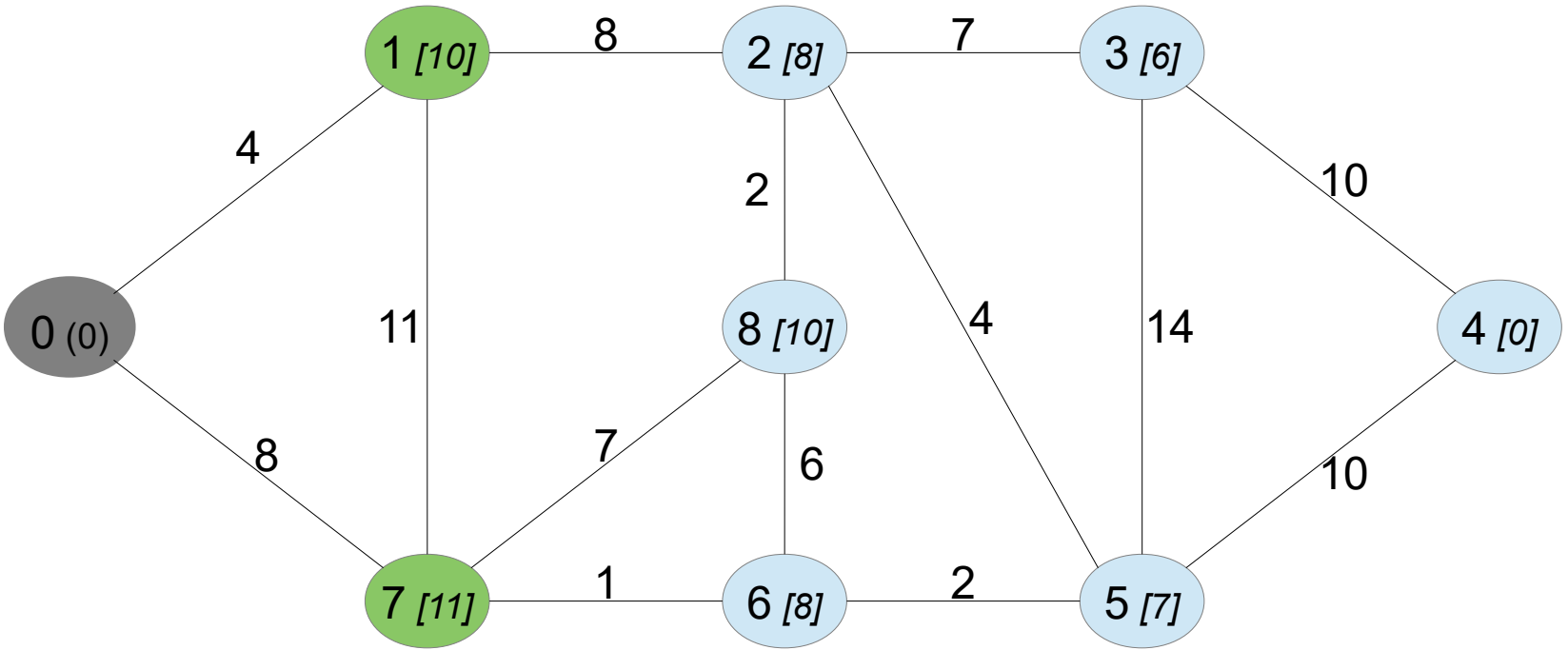
- Dijkstra's algorithm has a subtle but significant weakness: If assumes that adding a node to the tree will never result in a lower cost of the path from the start to that node, vs. the path from the start to the node's predecessor. In other words, it fails if any edge costs are negative.
- In the 1950s, an algorithm that handled negative edge costs was proposed or published independently by 4 researchers: Alfonso Shimbel, Lester Ford Jr., Edward F. Moore, and Richard Bellman. This algorithm has the capability not only to handle negative-cost edges, but also to detect negative-cost cycles.
- The Bellman-Ford algorithm (as it is most commonly known) is less efficient than Dijkstra's algorithm*, but more flexible—and more amenable to parallel or distributed computation.

* Depending on the data structures used, Dijkstra's algorithm is $O(E \log V)$ or $O(V \log V)$; Bellman-Ford is $O(EV)$, or for a very dense graph, $O(V^3)$.

A* Search Algorithm

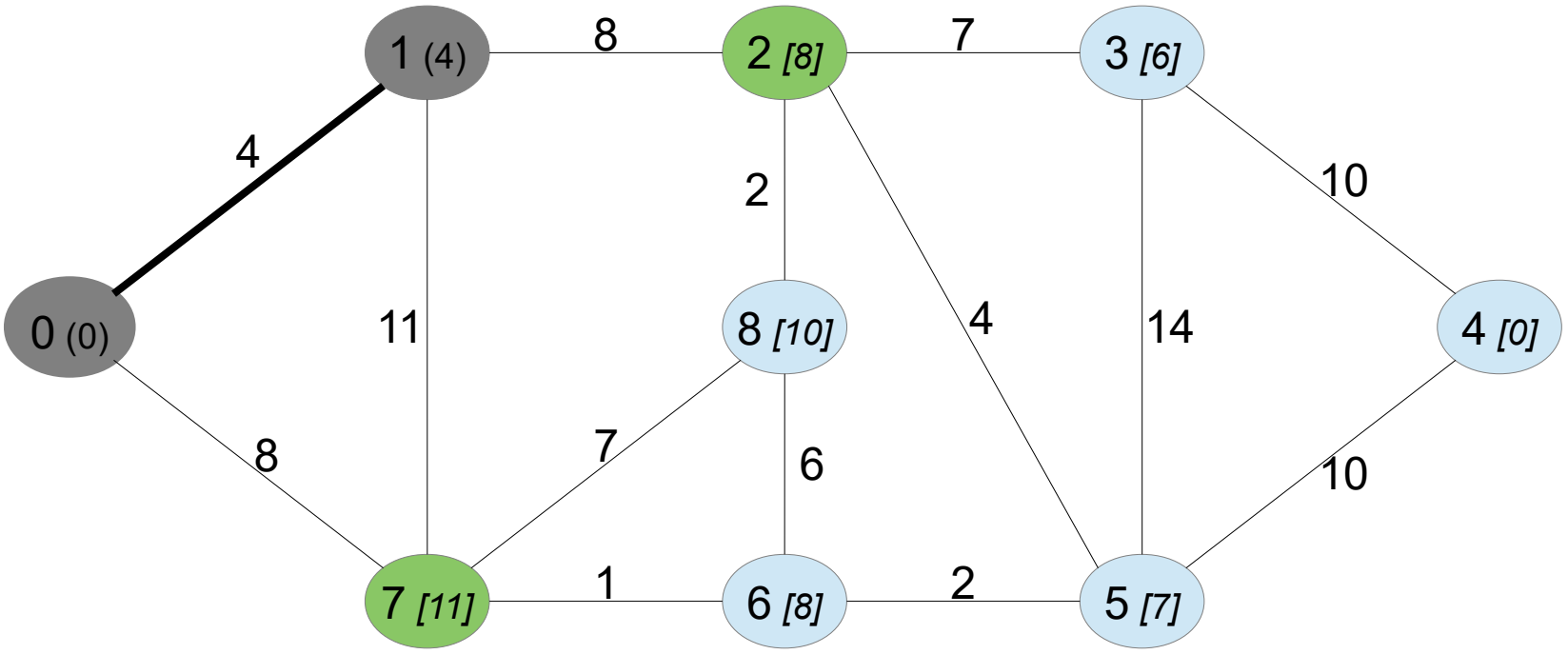
- A* is an algorithm for solving the shortest path problem (and other related graph traversal problems), first published in 1968 by Peter Hart, Hans Nilsson, and Bertram Raphael, of SRI International.
- A* extends Dijkstra's algorithm by incorporating additional information into the process by which new nodes are added to the tree. In practice, this *can* exclude from consideration significant sections of the overall graph, and more quickly identify nodes along the shortest path.
- Instead of adding to the tree the node that has the minimum total path length from the starting node, the node that has the minimum sum of the total path length and an *estimated* distance to the destination node is chosen for addition to the tree.
- Where might we obtain these estimated distances, in practice?

A* Search: Example

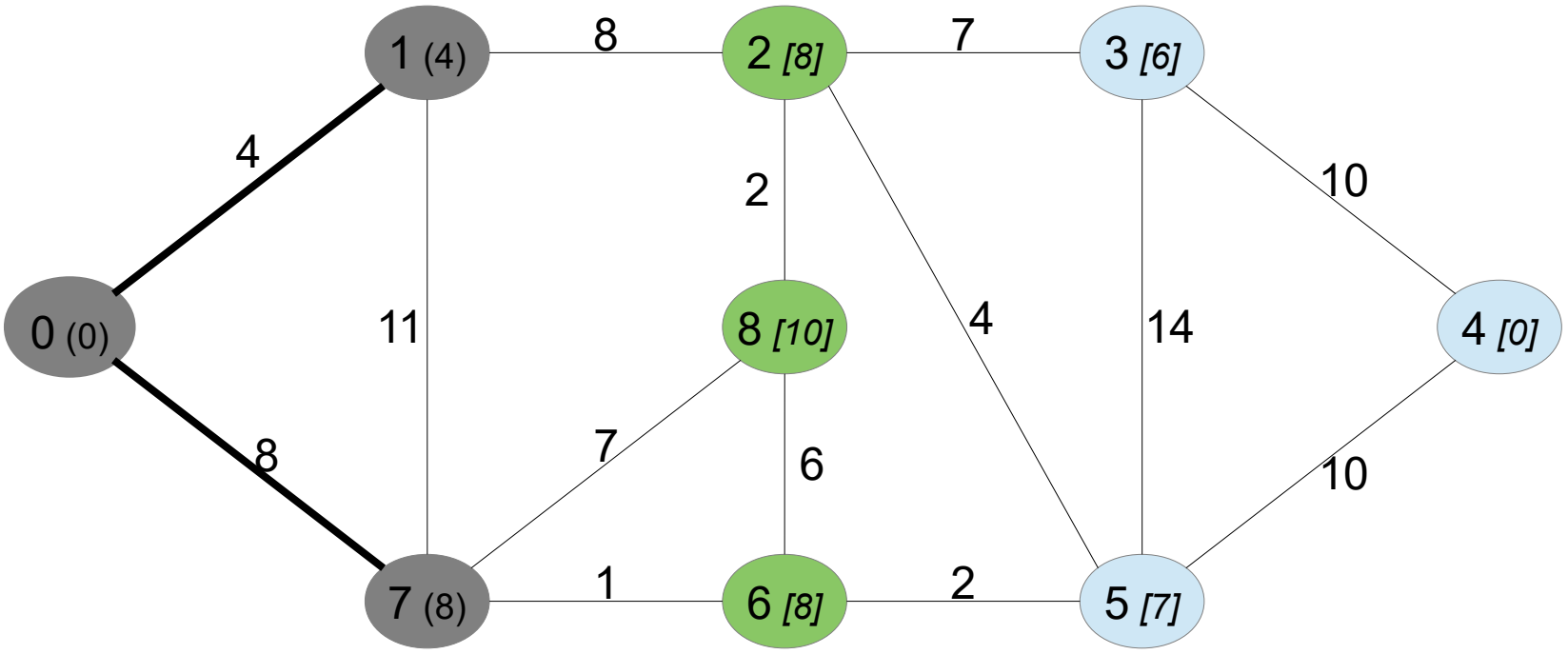


Note: Before a node is added to the tree, the estimated distance from that node to the ending node is show italicized, in brackets. After it is added, the actual distance from the starting node is shown.

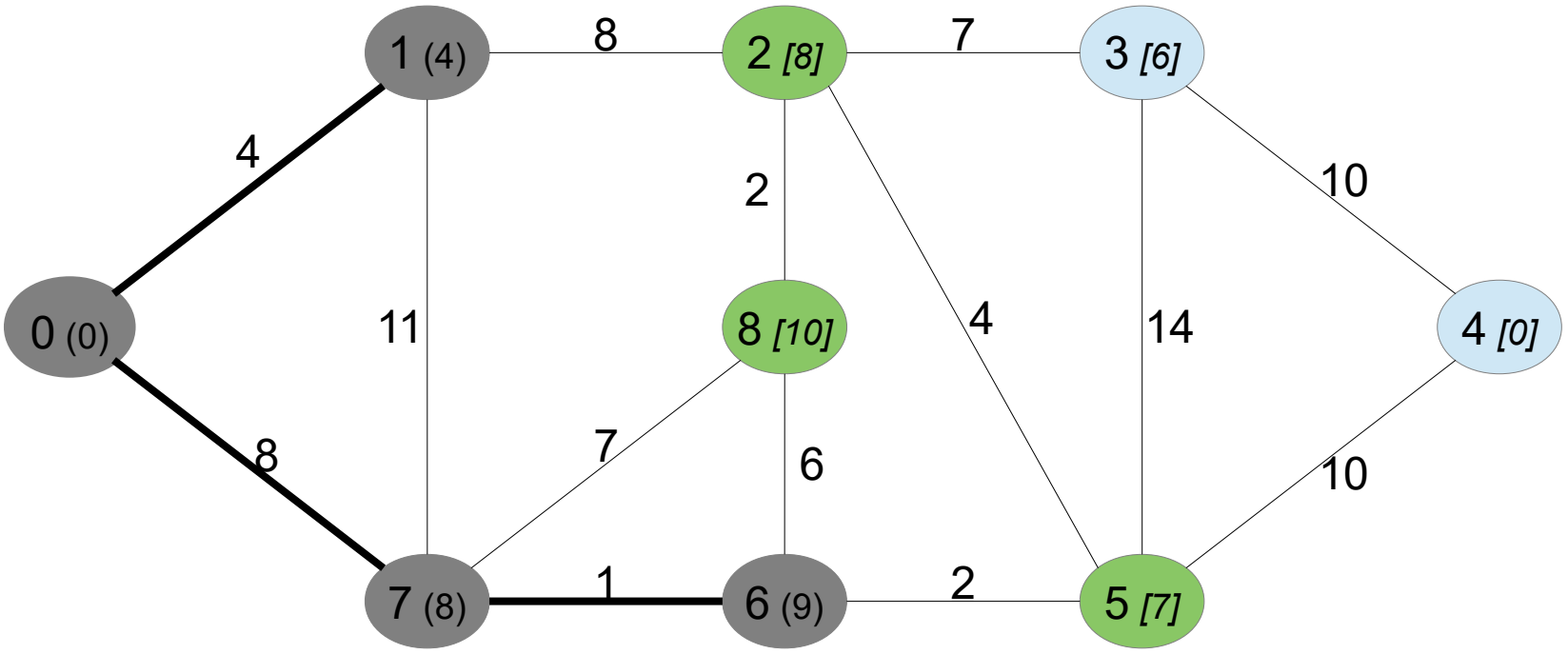
A* Search: Example



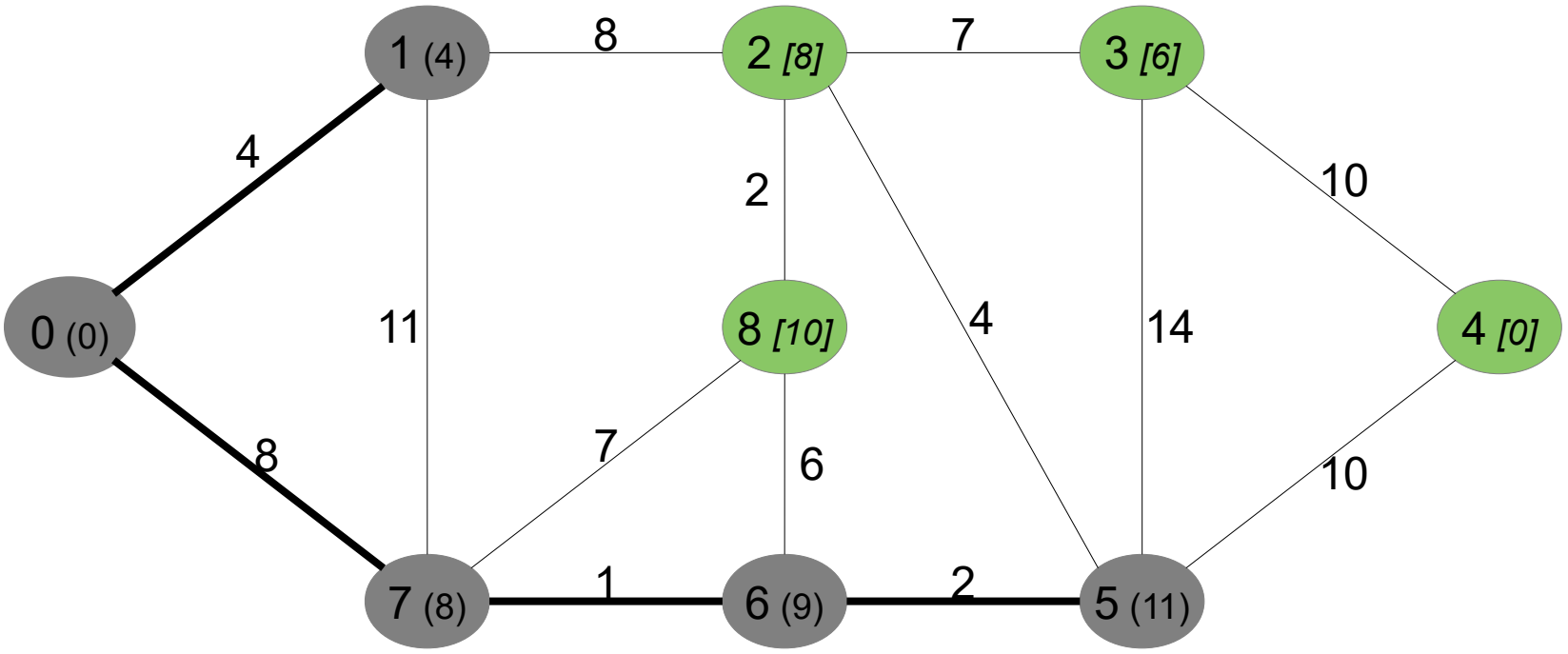
A* Search: Example



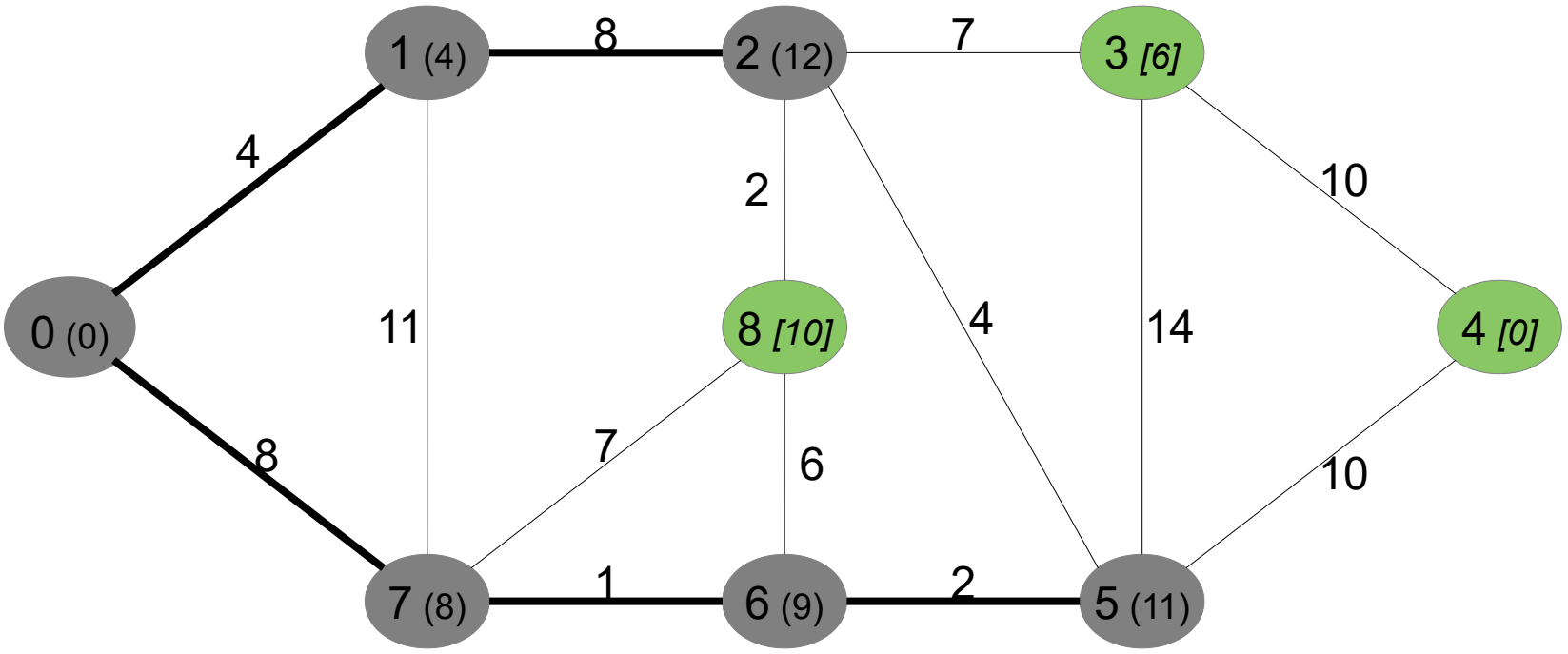
A* Search: Example



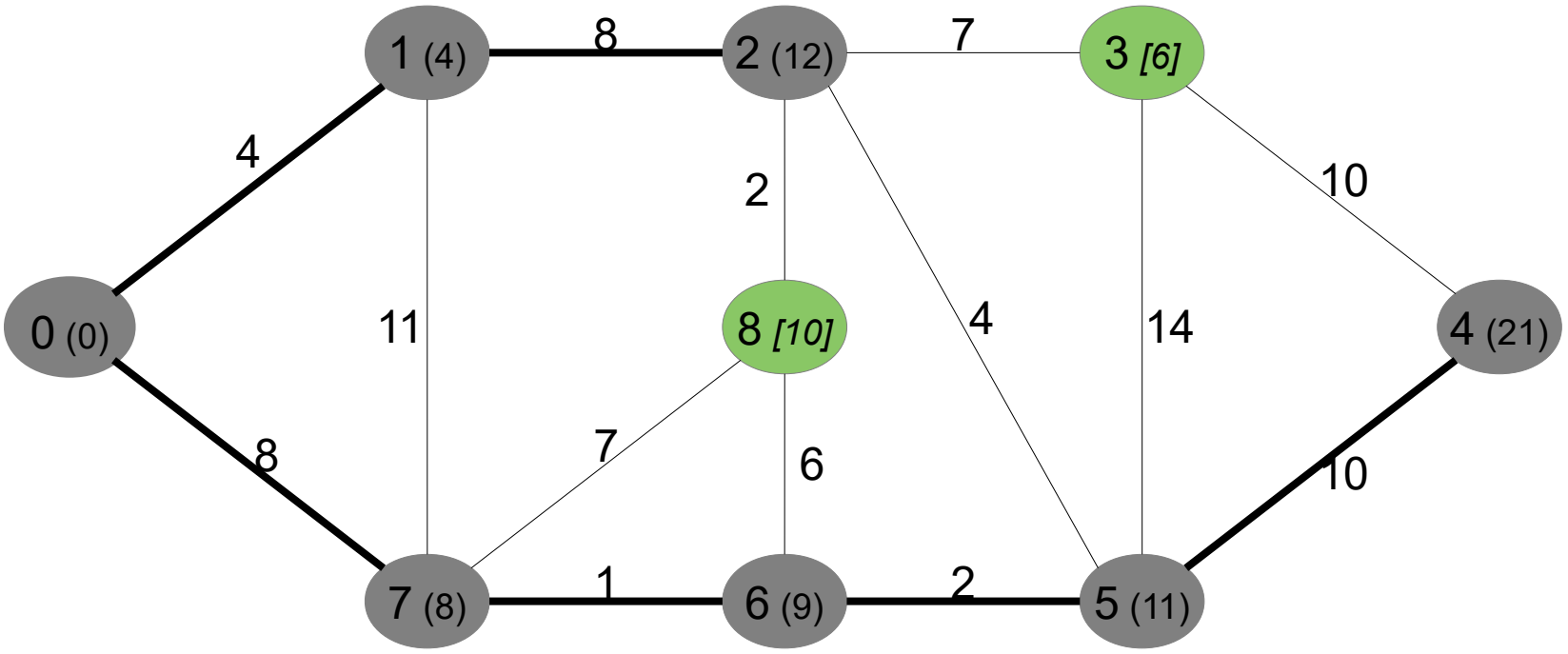
A* Search: Example



A* Search: Example



A* Search: Example



A* Performance

- The performance of A* depends greatly on the heuristic estimates of the distance from each node to the ending node.
- If the heuristic overestimates the actual path length or cost from any node to the ending node, A* is not guaranteed to find the shortest path. This is called an *inadmissible* heuristic. (An *admissible* heuristic is one that never overestimates the path length or cost.)
- If the heuristic gives an estimate of 0 for the path length from any node to the ending node (this is an admissible heuristic, but not informative), the A* search algorithm reduces to Dijkstra's algorithm.
- If the heuristic gives the actual correct path length, then A* will find the shortest path without adding any unused nodes to the tree.

Applications

- Route planning & travel time estimation (e.g. Google Maps).
- Internet routing via *Open Shortest Path First* (OSPF).
- Puzzle solutions, where each state of the puzzle can be represented as a node on a graph, directly reachable via moves from adjacent states (e.g. sliding tiles, Rubik's cube).
- Protein sequencing.
- …

Dijkstra's Algorithm vs. A* (et al)

- As noted previously, A* can outperform Dijkstra's algorithm, as long as the heuristic estimate of distance from each intermediate node to the terminal node is both admissible and informative.
- A* is not well suited to finding the shortest paths from a given starting node to *all* other nodes; this is easily done with Dijkstra's algorithm (or Bellman-Ford, for graphs with negative weights).
- For an unweighted graph—i.e. one where each "hop" has a uniform cost—Dijkstra's algorithm becomes, essentially, the simpler breadth-first search (BFS).