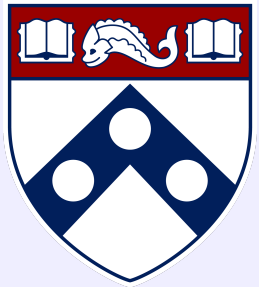
All-Pairs Shortest Paths



Goal

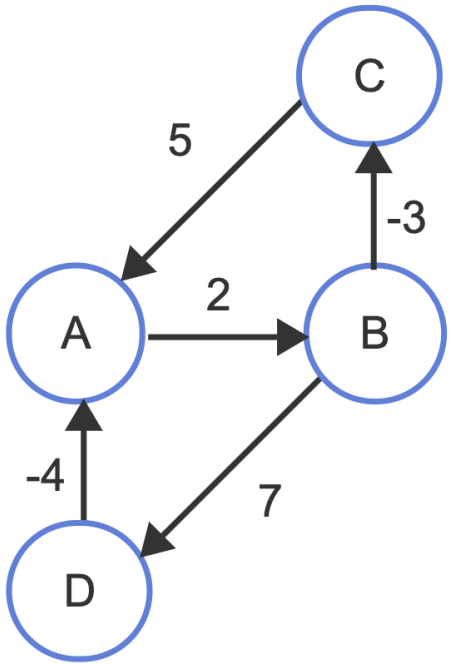
- Calculate the shortest path distance between any two pairs of vertices in the graphs.
- Constraints:
  - Cycles?
  - Negative Edge Weights?
  - Negative Weight Cycles?

Goal

- Calculate the shortest path distance between any two pairs of vertices in the graphs.
- Constraints:
  - Cycles? ✅
  - Negative Edge Weights? ✅
  - Negative Weight Cycles 🔴

An Example Instance

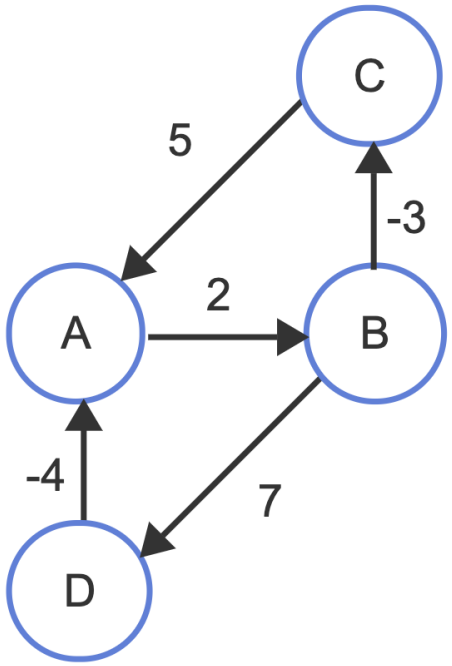
- The output for APSP problems is a $|V| \times |V|$ matrix
- Rows are "from", columns are "to"



		to			
		A	B	C	D
from	A	0	2	-1	9
	B	2	0	-3	7
	C	5	7	0	14
	D	-4	-2	-5	0

An Observation...

The matrix diagonal is all zeroes.



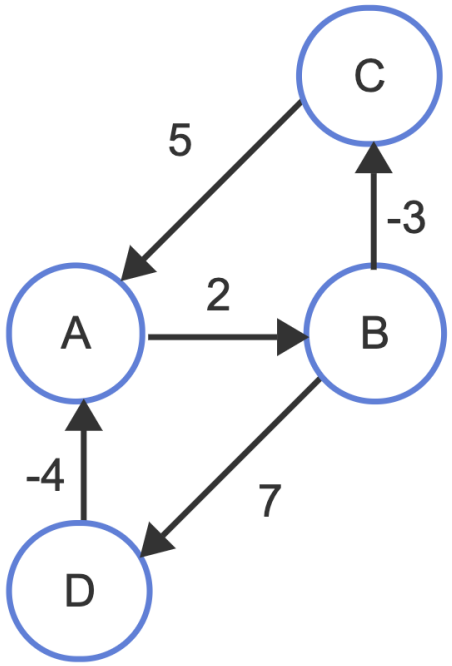
		to			
		A	B	C	D
from	A	0	2	-1	9
	B	2	0	-3	7
	C	5	7	0	14
	D	-4	-2	-5	0

A Conclusion!

Provided that there are no negative weight cycles,

$\text{shortestPath}(i, i) = 0$

🤔

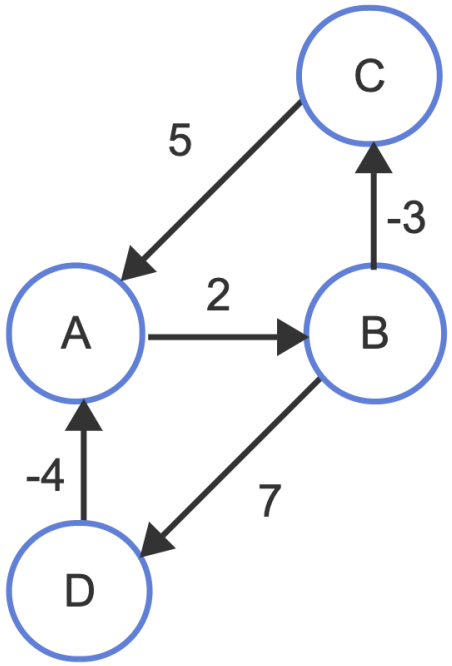


from \ to	A	B	C	D
A	0	2	-1	9
B	2	0	-3	7
C	5	7	0	14
D	-4	-2	-5	0

A Corollary.

If i is on a negative weight cycle,

$\text{shortestPath}(i, i) = -\infty$



from \ to	A	B	C	D
A	0	2	-1	9
B	2	0	-3	7
C	5	7	0	14
D	-4	-2	-5	0

$\mathrm{shortestPath}()$ as a Function

- $\mathrm{shortestPath}(i, j)$ represents the cost of the shortest path starting at vertex i and ending at vertex j
- We can "overload" $\mathrm{shortestPath}()$ with a third argument: $\mathrm{shortestPath}(i, j, k)$
  - We label all vertices in V with indices $1, 2, 3, \ldots, |V|$
  - The shortest path starting at vertex i and ending at vertex j containing only intermediate vertices with indices $\leq k$
- Therefore, $\mathrm{shortestPath}(i, j) = \mathrm{shortestPath}(i, j, |V|)$

Calculating $\mathrm{shortestPath}()$

- This is the hard part of the APSP problem: find $\mathrm{shortestPath}(i, j, |V|)$ for all values $i, j \in [1, |V|]$.
- Some of these are easy, though

Base Cases to the Rescue

For which values of i, j, k is it trivial to compute $\text{shortestPath}(i, j, k)$?

Base Cases to the Rescue

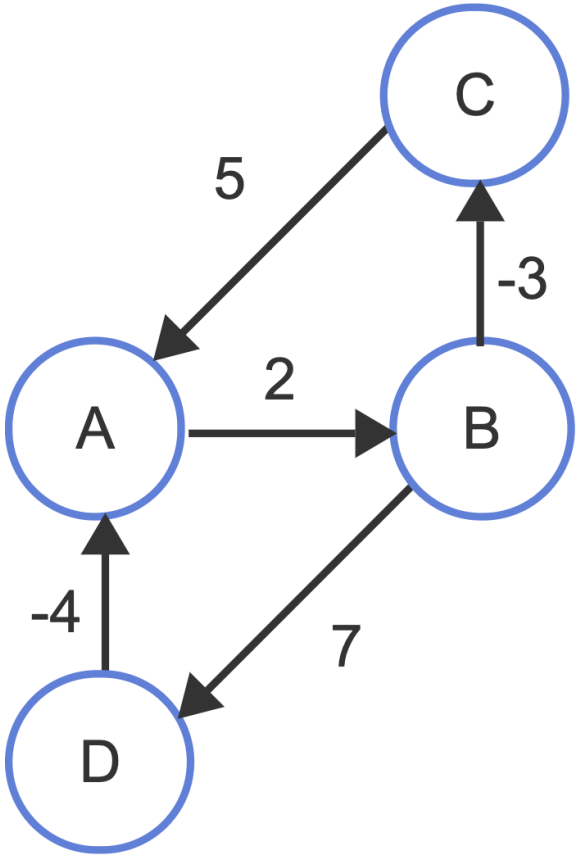
For which values of i, j, k is it trivial to compute $\text{shortestPath}(i, j, k)$?

- When $i = j, k = 0, \text{shortestPath}(i, j, k) = 0$.
- When $i \neq j, k = 0$, and $e_{i,j} \in E, \text{shortestPath}(i, j, k) = w_{i,j}$
- When $i \neq j, k = 0$, and $e_{i,j} \notin E, \text{shortestPath}(i, j, k) = \infty$

Starting Out

$k = 0$	A	B	C	D
A	0	2	∞	∞
B	∞	0	-3	7
C	5	∞	0	∞
D	-4	∞	∞	0

This is the full set of results of $\text{shortestPath}(i, j, 0)$!



Inductive Steps: Bringing More Vertices into the Fold

- For any i, j, k, we can infer that $\text{shortestPath}(i, j, k) \geq \text{shortestPath}(i, j, k+1)$
  - There are at least as many paths from i to j that include the first $k+1$ vertices than those that can be built with just the first k
  - If there's a shorter path, then we take that one!
  - If there isn't, then we stick with the previous shortest.

Inductive Steps: Bringing More Vertices into the Fold

- If $\text{shortestPath}(i, j, k+1) < \text{shortestPath}(i, j, k)$...
  - That means we found a shorter path using vertex $k+1$! (Otherwise, we would have had the path using just $[1, k]$)
  - This new path consists of a path from $i \to k+1$ and a path from $k+1 \to j$
  - Both of these subpaths use only vertices $[1, k]$ (🤔), and they are the **shortest paths** using these vertices.

Inductive Steps: Bringing More Vertices into the Fold

Therefore, assuming we have $\text{shortestPath}(i, j, k)$ for all i, j and a fixed k, then we know that

$$\text{shortestPath}(i, j, k+1) = \min(\text{shortestPath}(i, j, k),$$
$$\text{shortestPath}(i, k+1, k) + \text{shortestPath}(k+1, j, k))$$

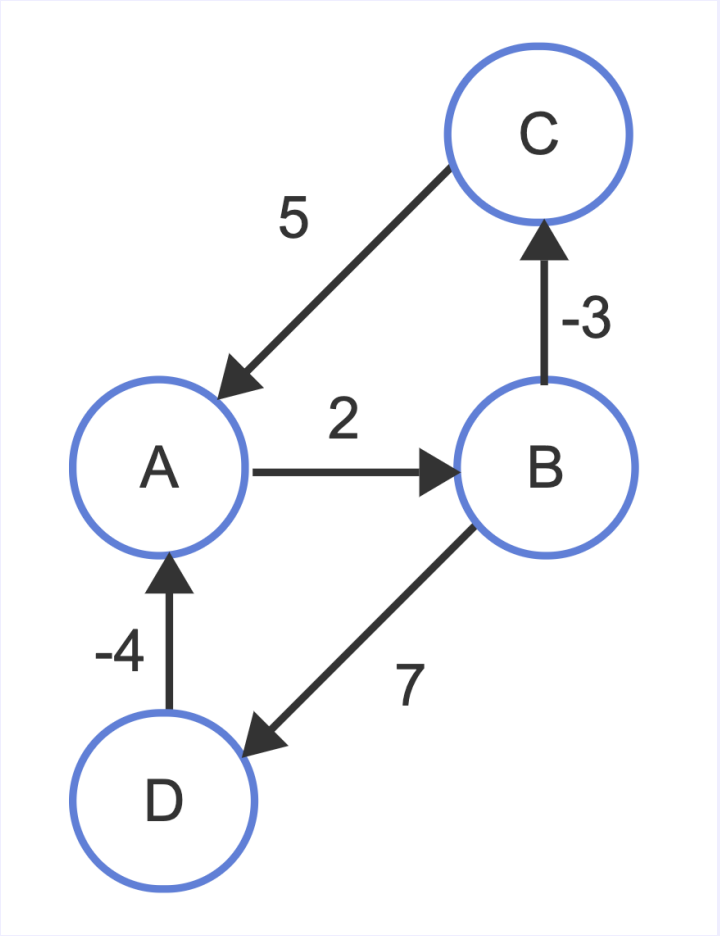
Floyd-Warshall

(pseudo-java)

```
floydWarshall(Graph g) {
    distances = new double[|V|][|V|];
    distances.setAllValues(Double.POSITIVE_INFINITY)
    for (Edge e : g.edges) {
        distances[e.from][e.to] = e.weight
    }
    for (Vertex v : g.vertices) {
        distances[v][v] = 0
    }
    for (int k = 1; k <= |V|; k++) {
        for i --> |V| and for j --> |V|:
            if (distances[i][j] > distances[i][k] + distances[k][j]) {
                distances[i][j] = distances[i][k] + distances[k][j]
            }
    }
}
```

Running the Algorithm

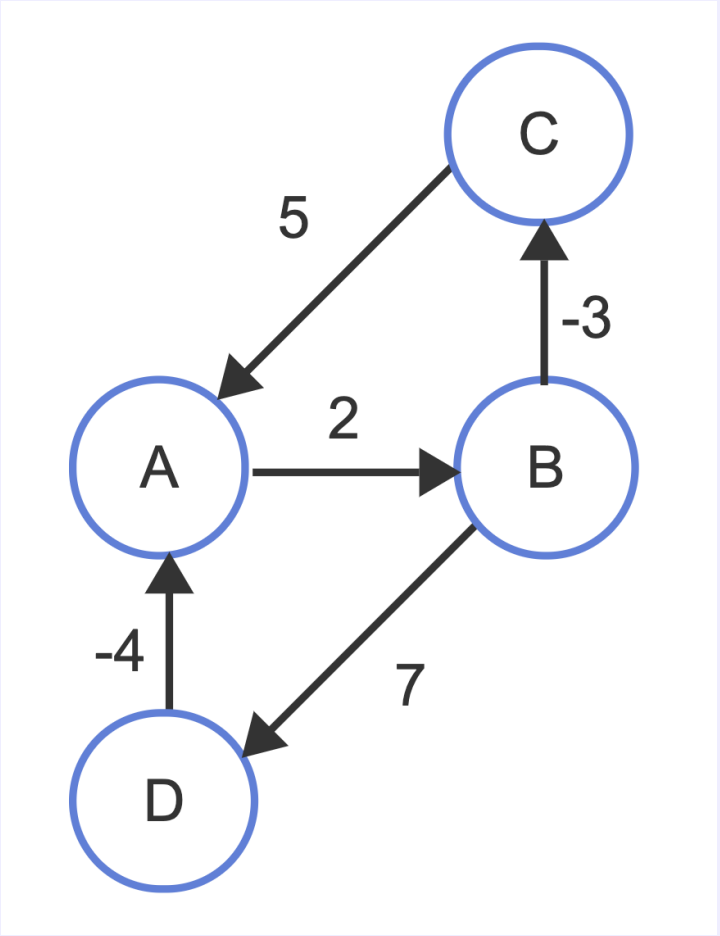
$k = 0$	A	B	C	D
A	0	2	∞	∞
B	∞	0	-3	7
C	5	∞	0	∞
D	-4	∞	∞	0



Running the Algorithm

(let the labeling be
$A = 1, B = 2, C = 3, D = 4$)

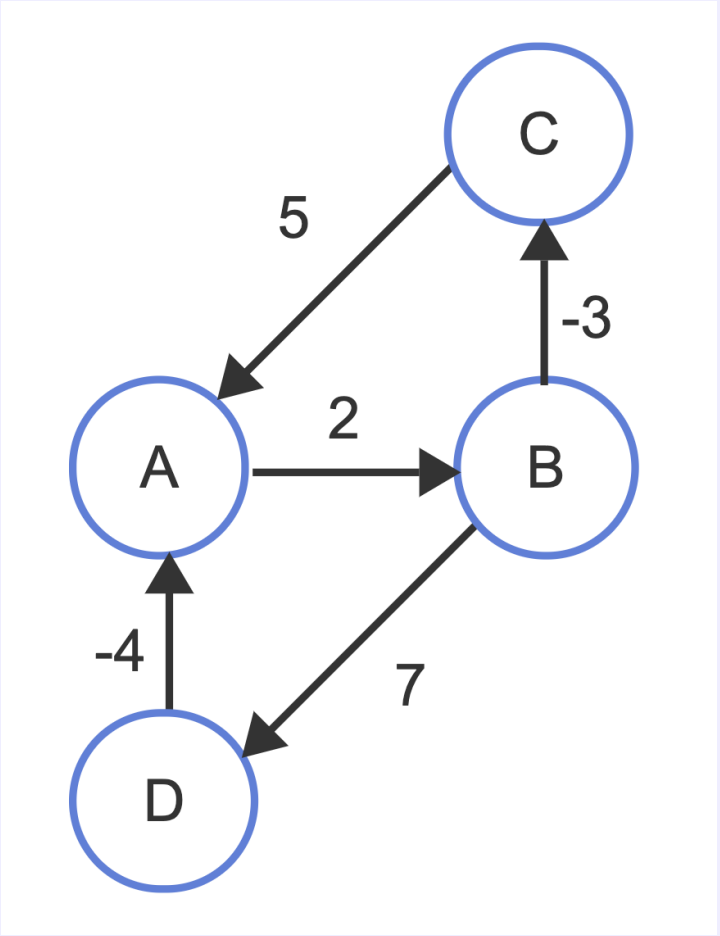
$k = 1$	A	B	C	D
A	0	2	∞	∞
B	∞	0	-3	7
C	5	7	0	∞
D	-4	-2	∞	0



Running the Algorithm

(let the labeling be
$A = 1, B = 2, C = 3, D = 4$)

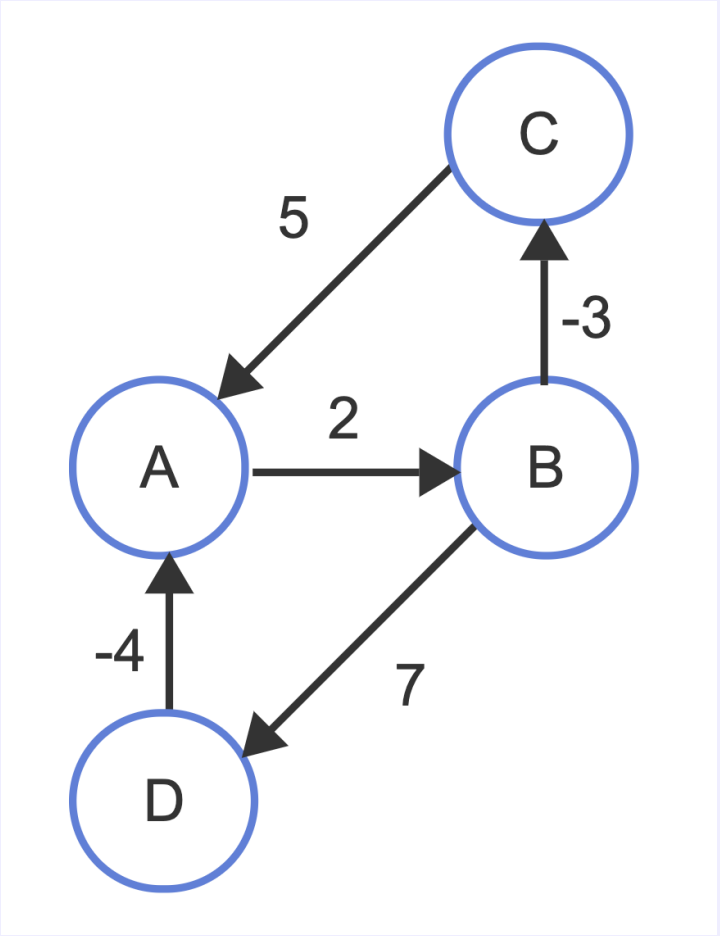
$k = 1$	A	B	C	D
A	0	2	-1	-9
B	∞	0	-3	7
C	5	7	0	14
D	-4	-2	-5	0



Running the Algorithm

(let the labeling be
$A = 1, B = 2, C = 3, D = 4$)

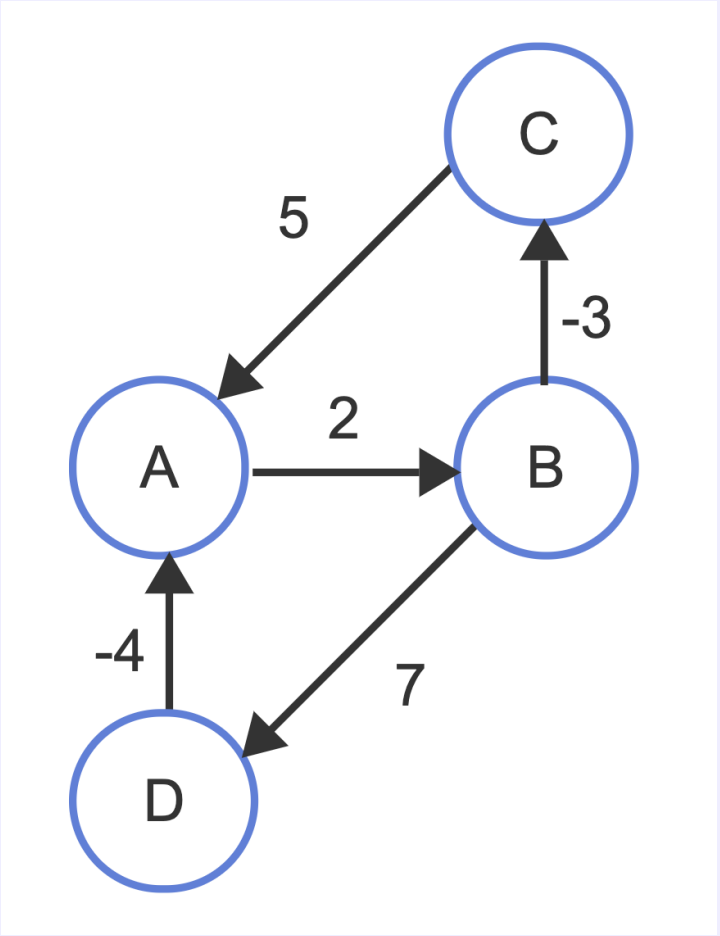
$k = 2$	A	B	C	D
A	0	2	-1	-9
B	2	0	-3	7
C	5	7	0	14
D	-4	-2	-5	0



Running the Algorithm

(let the labeling be
$A = 1, B = 2, C = 3, D = 4$)

$k = 3$	A	B	C	D
A	0	2	-1	-9
B	2	0	-3	7
C	5	7	0	14
D	-4	-2	-5	0



Done!

Runtime Analysis

```
floydWarshall(Graph g) {
    distances = new double[|V|][|V|];
    distances.setAllValues(Double.POSITIVE_INFINITY)
    for (Edge e : g.edges) {
        distances[e.from][e.to] = e.weight
    }
    for (Vertex v : g.vertices) {
        distances[v][v] = 0
    }
    for (int k = 1; k <= |V|; k++) {
        for i --> |V| and for j --> |V|:
            if (distances[i][j] > distances[i][k] + distances[k][j]) {
                distances[i][j] = distances[i][k] + distances[k][j]
            }
    }
}
```

Runtime Analysis

We have a triply nested for-loop ranging from $1 \rightarrow |V|$ each time.

Therefore, our runtime is a cut-and-dry $O(|V|^3)$!

- Gut check: Dijkstra's runs in $O(|V|^2)$ for an unsorted vertex ordering
- Good that Floyd-Warshall is no worse than running Dijkstra's $|V|$ more times!

Java time!

Reconstructing the Path

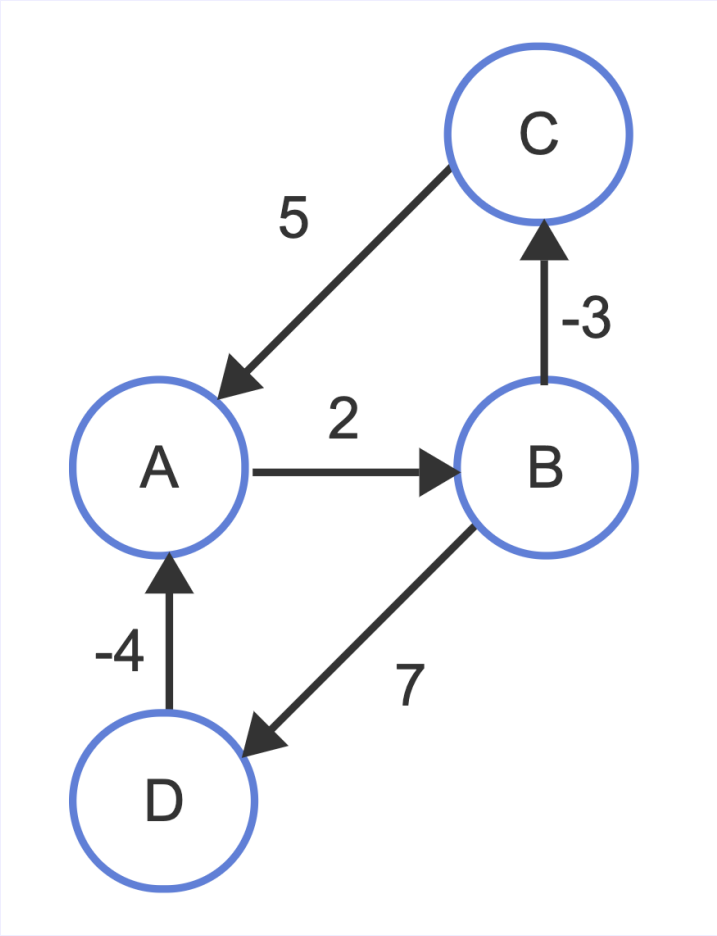
Floyd-Warshall is a dynamic programming algorithm, and these algorithms typical have a step where we have to read a solution from the resulting table.

$k = 3$	A	B	C	D
A	0	2	-1	-9
B	2	0	-3	7
C	5	7	0	14
D	-4	-2	-5	0

Reconstructing the Path

How do we read the shortest path from B to A?

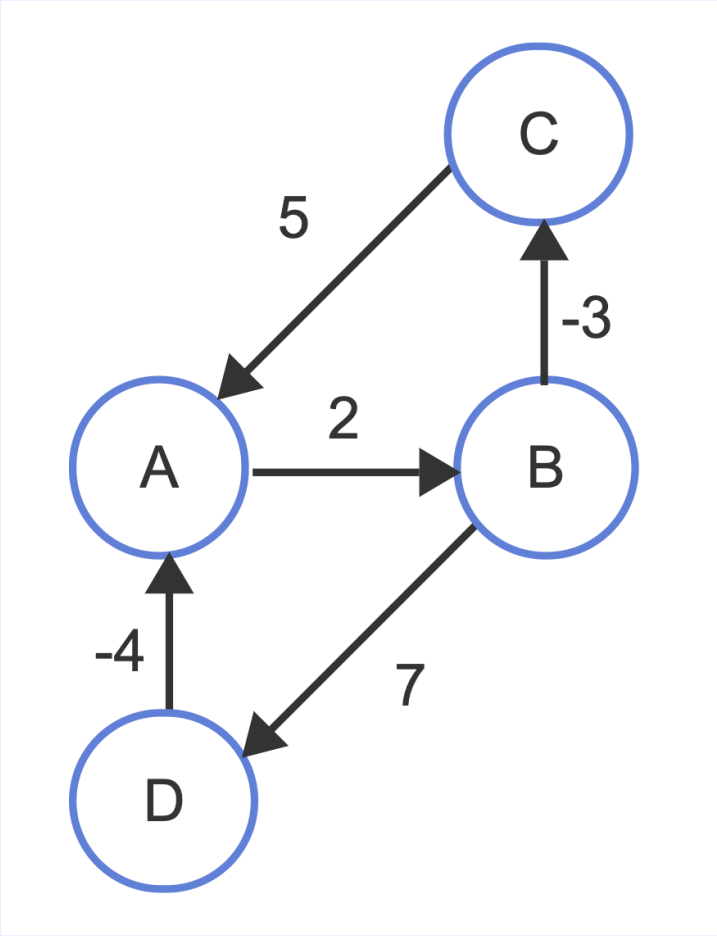
- We know the path goes through A.
- Of all possible ways to get to A, we should choose the shortest.
  - Perhaps the path comes from C?
  - Alternatively, maybe from D?



Reconstructing the Path

How do we read the shortest path from B to A?

- For an edge to be part of the shortest path, we need that $sp(B, A) - w_{X,A} = sp(B, X)$
- If we come from C, $sp(B, A) - w_{C,A} = -3$ and $sp(B, C) = -3$
- If we come from D, $sp(B, A) - w_{D,A} = 6$ and $sp(B, D) = 7$

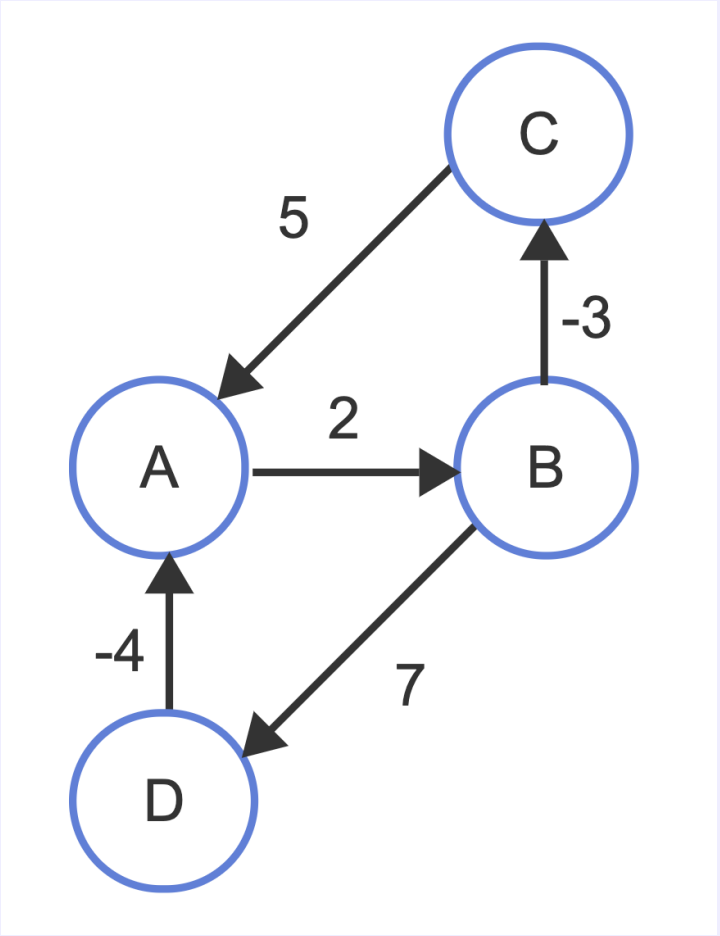


Reconstructing the Path

How do we read the shortest path from B to A?

Apparently, the shortest path goes from D to A. Now, we need to repeat, calculating the last step of the path from B to D.

Eventually, we get that the shortest path is $B \to D \to A$.



Reconstructing the Path

```
reconstructPath(graph, startVertex, endVertex, distances) {
   path = new, empty path

   currentV = endVertex
   while (currentV != startVertex) {
      incomingEdges = graph.getEdgesTo(currentV)
      for (Edge e : incomingEdges) {
         expected = distances[startVertex][currentV] - e.weight
         actual = distances[startVertex][e.from]
         if (expected == actual) {
            currentV = e.from
            path.append(e)
            break
         }
      }
   }

   return path
}
```

Easier: Just Store the Parents

Why bother with that fussy algorithm? Just store a table of parent pointers!

- Same space complexity
- Update the parents each time a "relaxation" is performed
- Lookup is straightforward

Easier: Just Store the Parents

```
floydWarshall(Graph g) {
    distances = new double[|V|][|V|];
    parents = new Vertex[|V|][|V|];
    distances.setAllValues(Double.POSITIVE_INFINITY)
    for (Edge e : g.edges) {
        distances[e.from][e.to] = e.weight
        parents[e.from][e.to] = e.from
    }
    for (Vertex v : g.vertices) {
        distances[v][v] = 0
        parents[v][v] = v;
    }
    for (int k = 1; k <= |V|; k++) {
        for i --> |V| and for j --> |V|:
            if (distances[i][j] > distances[i][k] + distances[k][j]) {
                distances[i][j] = distances[i][k] + distances[k][j]
                parents[i][j] = parents[k][j]
            }
    }
}
```

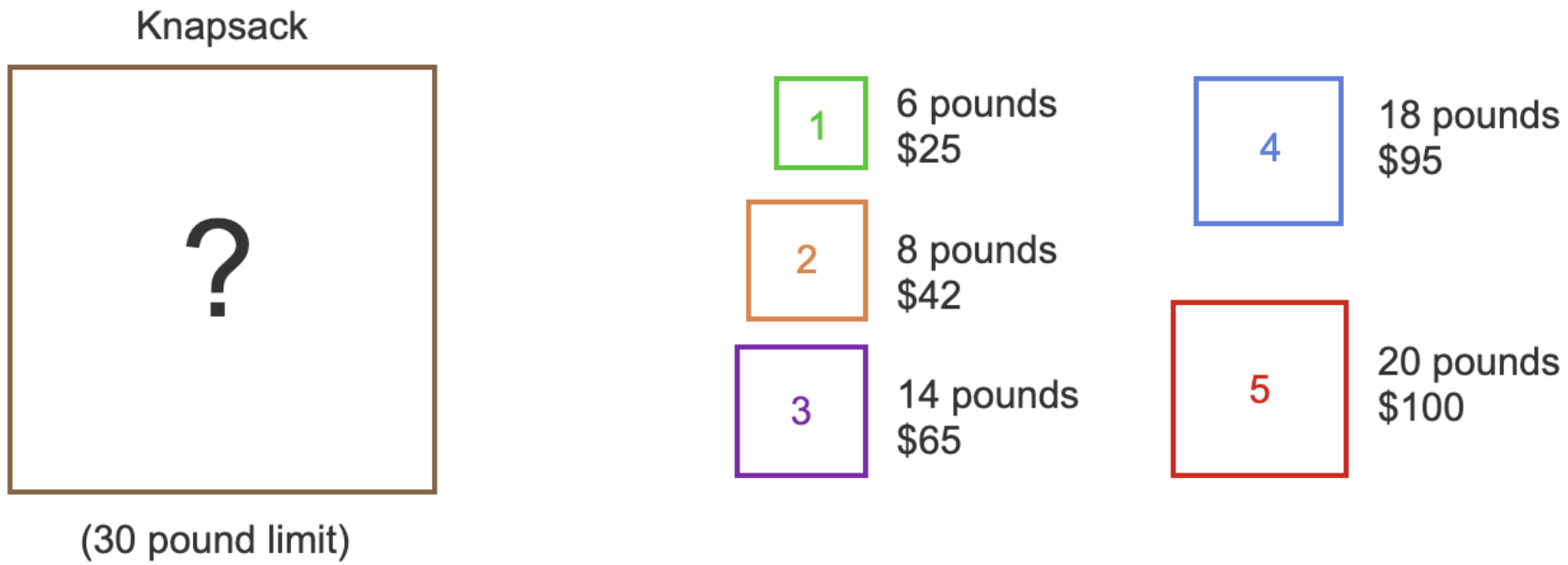
More Dynamic Programming & Heuristics

Knapsack Problem

- Knapsack:
  - Given a set of items with **weights** (kgs) and **values** ($$), choose the correct number of each item in the set to maximize total value without exceeding a maximum weight
- 0-1 Knapsack:
  - Same as above, but each item can be chosen 0 or 1 times.

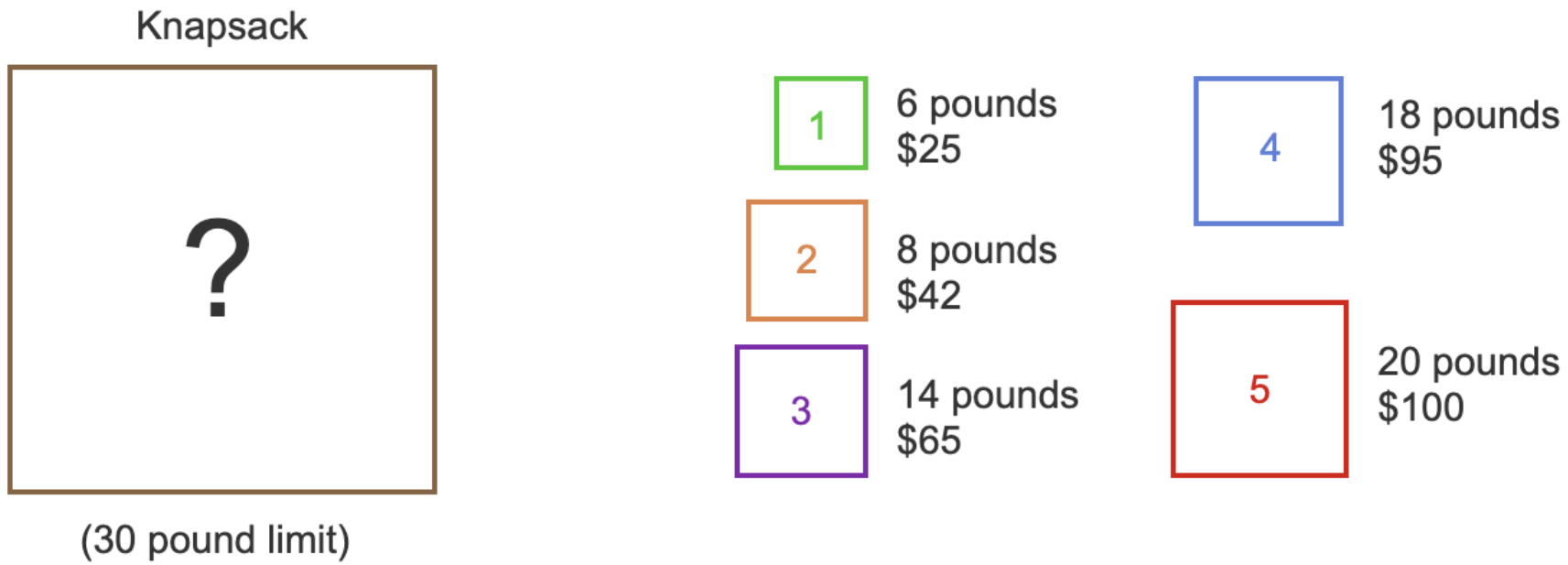
Knapsack Example

Can you find the optimal solution to this Knapsack problem?



Knapsack Example

Optimal: 2 of **item 1** and 1 of **item 4** for a total of $145.



0-1 Knapsack Formulation

- Each of our n items have labels $1, 2, 3, \ldots, n$.
- The maximum weight of the knapsack is W.
- The weight of each individual item i is represented by w_i.
- The value of each individual item i is represented by v_i.

We want to choose the subset S of $[1, n]$ such that the sum of weights is less than W but the sum of values is maximized.

Objective Function

We will use an objective function $\mathrm{maxValue}(i, w)$, meaning the maximum value attainable using only the first i elements with a maximum weight of w.

Therefore, the overall problem we want to solve is $\mathrm{maxValue}(n, W)$.

Base Cases

- $\mathrm{maxValue}(0, w) = 0$
  - Can't get any value using no items!
- $\mathrm{maxValue}(i, w) = \mathrm{maxValue}(i-1, w)$ when $w_i > w$
  - Can't add the new item if it's too heavy.

Recursive Cases

- If $w_i \leq w$, then we have the choice of whether or not to include item i.
- When we include an item i, we are effectively lowering the maximum available weight by w_i.
- If we don't include an item i, then we're using the maximum values using only items $[1, i-1]$.

Recursive Cases

If $w_i \leq w$, then we have the choice of whether or not to include item i.

In this case,

$$\text{maxValue}(i, w) = max(\text{maxValue}(i-1, w), \text{maxValue}(i, w - w_i) + v_i)$$

(either we include the element, or we don't!)

Dynamic Programming to the Rescue!

Without Dynamic Programming, the recursive $max()$ operation will incur an $O(2^n)$ runtime.

Heuristic

A technique that willingly accepts a non-optimal or less accurate solution in order to improve execution speed.

Heuristics & Knapsack

- Knapsack is very hard (read: slow) to solve!
  - Already NP-Complete to just identify if a certain value can be acheived for a given Knapsack instance
  - Even harder still to calculate an optimal solution

Heuristics & Knapsack

- If we're willing to trade **optimality** for **speed**, we can use a heuristic
  - The solution will not necessarily be the best, but at least we'll come up with a solution!

0-1 Knapsack Heuristic

- Sort the list of available items in descending order of value
- Traverse through the list in descending order and add each item if there is space remaining.

0-1 Knapsack Heuristic

```
public static Knapsack knapsack01(Item[] availableItems, double maxWeight) {
    // Sort the items in descending order based on value
    Arrays.sort(availableItems, new ItemValueComparator());

    // Initialize a new knapsack to hold items
    Knapsack knapsack = new Knapsack(maxWeight);

    double remaining = maxWeight;
    for (Item item : availableItems) {
        if (item.weight <= remaining) {
            knapsack.getItems().add(item);
            remaining -= item.weight;
        }
    }

    return knapsack;
}
```

Soon: Heuristics as Pathways to Optimal Solutions

- In A*, we use a heuristic function to help guide a shortest path search
- Even though the heuristic is always an optimistic (read: wrong) guess, we can use the intuition to pick better candidate paths