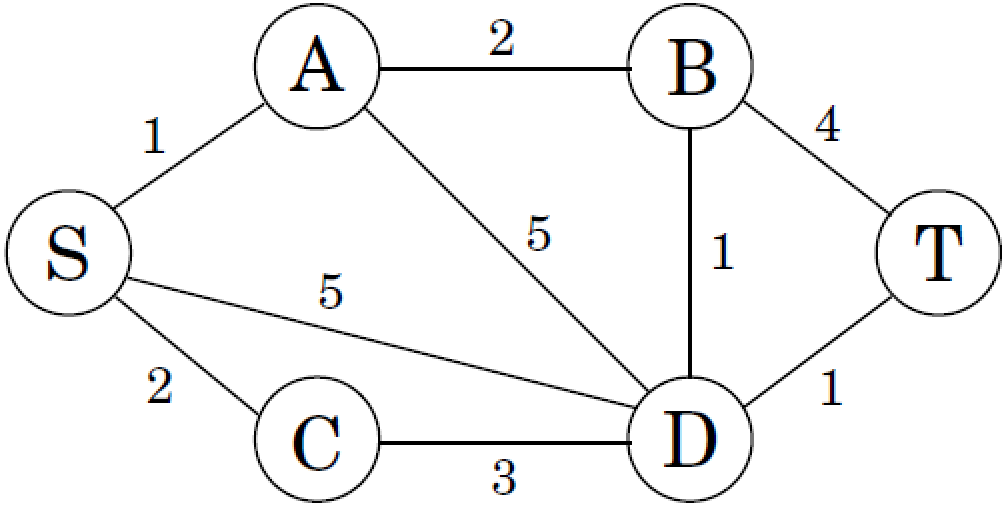
Dynamic programming

Dynamic programming

- Divide the problem into subproblems.
- Define subproblem recursively. (Express larger subproblem in terms of smaller ones.)
- Find the right order to solve the subproblems.

Shortest reliable paths

- We want a path from *s* to *t* that is both short *and has few edges*.



Shortest reliable paths

- Given a graph G with edge lengths, two nodes s and t and an integer k, we want the shortest path from s to t *that uses at most k edges*.
- Dijkstra's algorithm does not care the number of hops.
- Choose subproblems so that all vital information is remembered!
- For each vertex v and each integer $i \leq k$, define dist(v, i) = the length of the shortest path from s to v that uses i edges.
- Base case : dist(s, 0) = 0, dist(v, 0) = ∞ for all vertices except s.
- $\texttt{dist}(v, i) = \min_{(u,v)\in E} \{\texttt{dist}(u, i-1) + \ell(u, v)\}$

All-pairs shortest paths

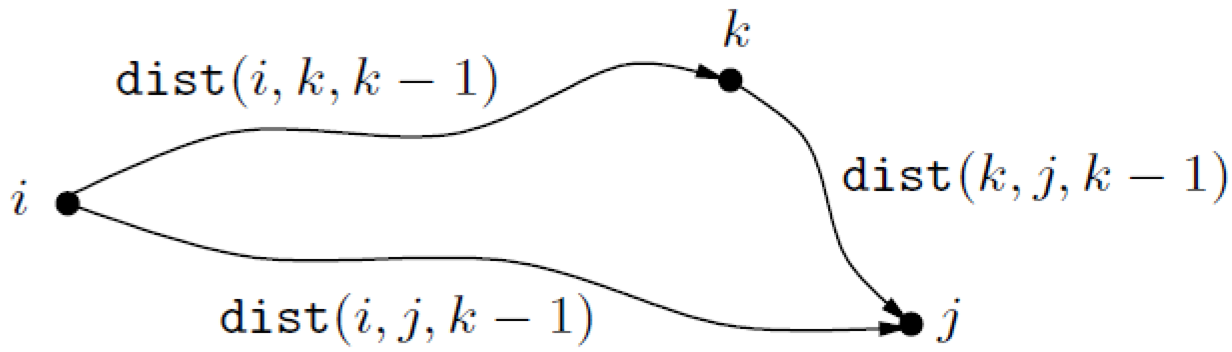
- How to find the shortest path between *all pairs* of vertices?
- Run single-source shortest path algorithm $|V|$ times, once for each starting node.
  - $|V| \times$ Bellman-Ford = $O(V^2E)$.
- Can we do better?
- What is a good *subproblem*?
- Consider the set of *intermediate* nodes.
- Initially, allow no intermediate node and gradually expand the *set of permissible intermediate nodes.*

Dynamic programming

- $V = \{ 1, 2, \ldots, n \}$
- dist(i, j, k) = the length of the shortest path from i to j in which only nodes { 1, 2, …, k } can be used as intermediate nodes.
- Initially, dist(i, j, 0) = l(i, j) if (i, j) $\in$ E,

  ∞ otherwise.

- How to expand the intermediate set to include k ?



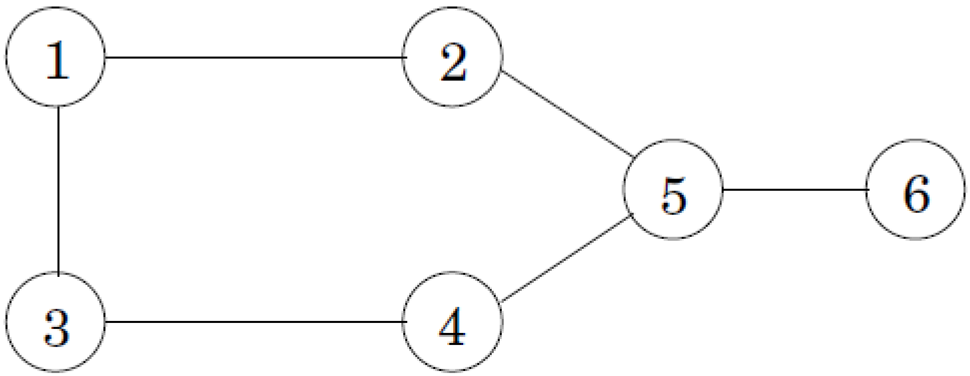
- dist(i, j, k) = min {dist(i, k, k - 1) + dist(k, j, k -1) , dist(i, j, k -1) }

Floyd-Warshall algorithm

for $i = 1$ to n:
    for $j = 1$ to n:
        $\texttt{dist}(i, j, 0) = \infty$
for all $(i, j) \in E$:
    $\texttt{dist}(i, j, 0) = \ell(i, j)$
for $k = 1$ to n:
    for $i = 1$ to n:
        for $j = 1$ to n:
            $\texttt{dist}(i, j, k) = \min\{\texttt{dist}(i, k, k-1) + \texttt{dist}(k, j, k-1),\ \texttt{dist}(i, j, k-1)\}$

Independent set

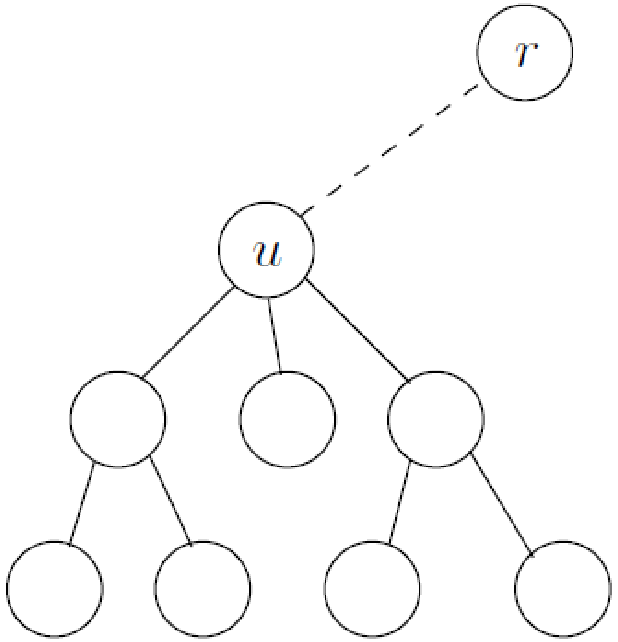
- A subset of nodes $S \subset V$ is an *independent set* of graph $G = (V,E)$ if there are no edges between them.
- Finding the largest independent set in a graph is believed to be intractable.



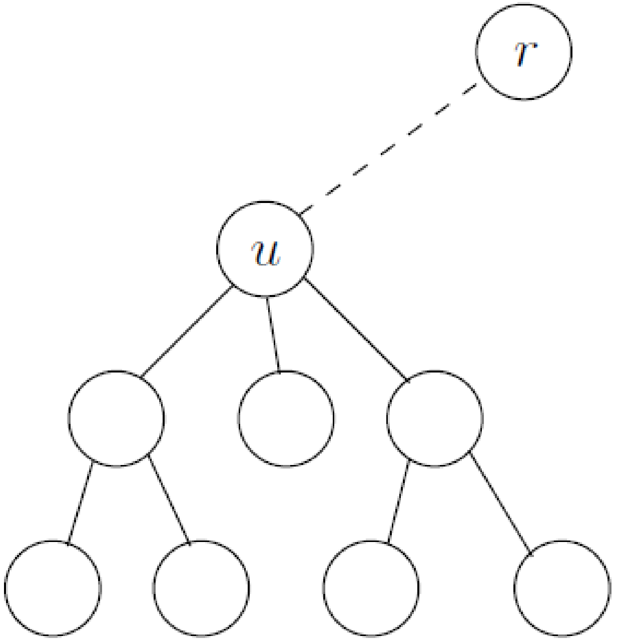
Independent set in trees

- When the graph is a *tree*, we can solve it in linear time!
- What is the subproblem?
- Start by rooting the tree at any node r. Now, each node defines a subtree - the one hanging from it.
- This immediately suggests subproblems:

  $I(u)$ = size of largest independent set of subtree hanging from u
- Goal : $I(r)$
- Suppose we know $I(w)$ for all descendants w of u.
- How can we compute $I(u)$?
- 2 cases: any independent set either includes u or it doesn't



- Case 1 If the independent set includes *u* :
  we get one point for it, but we cannot include the children of *u*. Move on to the grandchildren.



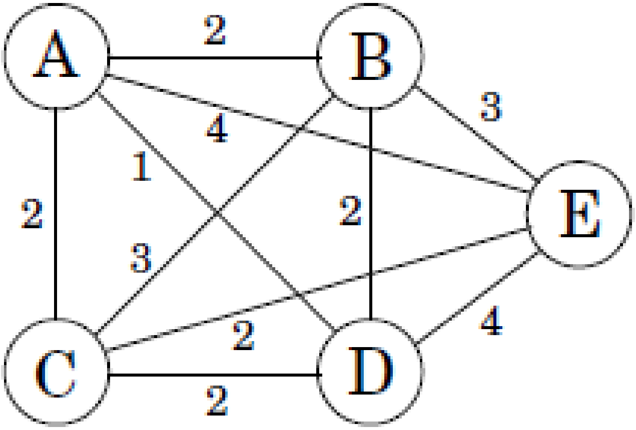
- Case 2 If we don't include *u* :
  we don't get a point for it, but we can move on to its children.

$$I(u) = \max\left\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w)\right\}$$

- The number of subproblems is exactly the number of vertices.
- Running time : O(|*V*| + |*E*|).

Traveling salesman problem (TSP)

- Given n cities and the matrix of intercity distances $D = (d_{ij})$, find a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length.



- What is the optimal traveling salesman tour?
- Brute-force : try all possible tour -> O(n!) time.

Dynamic programming

- What is the appropriate subproblem for TSP?
  - Consider initial portion of tour.

  - For a subset of cities $S \subseteq \{ 1, 2, \ldots, n \}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

  - When $|S|>1$, define $C(S, 1) = \infty$ since the path cannot both start and end at 1.

  - Express $C(S, j)$ in terms of smaller subproblems!

    The second-to-last city should be some $i \in$ S.

$$C(S, j) = \min_{i \in S : i \neq j} C(S - \{j\}, i) + d_{ij}.$$

- The subproblems are ordered by | *S* |.

```
C({1},1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1,2,...,n} of size s and containing 1:
        C(S,1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S,j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1,...,n}, j) + d_j1
```

- There are at most $2^n \cdot n$ subproblems.
- Each subproblem takes $O(n)$ time.
- Total running time : $O(n^2 2^n)$

Coin change problem

- Given a set of denominations $D = \{d_1, d_2, \ldots, d_k\}$, find the minimum number of coins for the given amount of cents, n.
- Assume each d_i is an integer and $d_1 > d_2 > \ldots > d_k$ and $d_k = 1$ so that there is always a solution.
- Greedy algorithm repeatedly chooses the largest coin less than or equal to the remaining sum, until the desired sum is obtained.
- For $D = \{ 25, 10, 5, 1 \}$, greedy algorithm works. (Prove it!)
- For $D = \{ 25, 10, 1 \}$, greedy does not work.

Dynamic programming

- Define $C[j]$ to be the minimum number of coins we need to make change for j cents.
- If an optimal solution used a coin of denomination d_i, we would have $C[j] = 1 + C[j - d_i]$.
- Recursively define $C[j]$.

$$C[j] = \begin{cases} \infty & \text{if } j < 0, \\ 0 & \text{if } j = 0, \\ 1 + \min\limits_{1 \le i \le k} \{C[j - d_i]\} & \text{if } j \ge 1 \end{cases}$$

Example $D = \{ 50, 25, 10, 1 \}$

- $C[0] = 0$

$$C[1] = \min \begin{cases} 1 + C[1 - 50] & = \infty \\ 1 + C[1 - 25] & = \infty \\ 1 + C[1 - 10] & = \infty \\ 1 + C[1 - 1] & = 1 \end{cases}$$

$$C[2] = \min \begin{cases} 1 + C[2 - 50] & = \infty \\ 1 + C[2 - 25] & = \infty \\ 1 + C[2 - 10] & = \infty \\ 1 + C[2 - 1] & = 2 \end{cases}$$

- Similarly, $C[3] = 3$, $C[4] = 4$, …, $C[9] = 9$, $C[10] = 1$

$$C[11] = \min \begin{cases} 1 + C[11-50] & = \infty \\ 1 + C[11-25] & = \infty \\ 1 + C[11-10] & = 2 \quad \{\ 1¢,\ 10¢\ \} \\ 1 + C[11-1] & = 2 \quad \{\ 10¢,\ 1¢\ \} \end{cases}$$

C[20] = 2; ..., C[29] = 5;

$$C[30] = \min \begin{cases} 1 + C[30-50] & = \infty \\ 1 + C[30-25] & = 1 + C[5] = 6 \\ 1 + C[30-10] & = 1 + C[20] = 3; \\ 1 + C[30-1] & = 1 + C[29] = 6; \end{cases}$$

Dynamic programming

- Avoid examining C[j] for $j < 0$ by ensuring that $j \geq d_i$ before looking up $C[j - d_i]$.
- *denom*[1..n] : *denom*[j] is the denomination of a coin used for making change for j

```
COMPUTE-CHANGE(n, d, k)
    C[0] := 0
    for j := 1 to n do
        C[j] := ∞
        for i := 1 to k do
            if j ≥ d_i and 1 + C[j − d_i] < C[j] then
                C[j] := 1 + C[j − d_i]
                denom[j] := d_i
    return c
```

- Running time : $\Theta(nk)$

Greedy vs. dynamic programming

- The knapsack problem is a good example of the difference.
- 0-1 knapsack problem :
  - n items.
  - Item i is worth \$$v_i$, weighs w_i pounds.
  - Find a most valuable subset of items with total weight $\leq W$.
  - Have to either take an item or not take it – can't take part of it.
- Fractional knapsack problem : Like the 0-1 knapsack problem, but can take fraction of an item.
- Greedy algorithm works for fractional knapsack problem. (Prove it!)
- Greedy algorithm does not work for 0-1 knapsack.

Knapsack problem example

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50$.

Greedy solution:

- Take items 1 and 2.
- value $= 160$, weight $= 30$.

Have 20 pounds of capacity left over.

Optimal solution:

- Take items 2 and 3.
- value $= 220$, weight $= 50$.

No leftover capacity.