Slides for a Course on the Analysis and Design of Algorithms

Chapter 4: Dynamic Programming and Optimization

Stephen J. Hegner
Department of Computing Science
Umeå University
Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

4. Dynamic Programming and Optimization

4.1 Basic Shortest-Path Problems on Graphs

4.1.1 General definitions for shortest-path problems for graphs

- Let $G = (V, E.g)$ be a directed graph, and let

  $$p : E \to \mathbb{R}^{>0}$$

  be an associated cost function.

- Given a path $P = \langle e_1, e_2, \ldots, e_k \rangle$, the *length* (or *profit*, or *cost*) of P is

  $$p(P) = \sum_{i=1}^{k} p(e_i)$$

- P is a *shortest path* from v to w if it is a path from v to w such that, for any other path Q from v to w, $p(P) \leq p(Q)$.

- Three distinct variations of this problem will be investigated.

  Single source shortest path: Given a vertex v, find a shortest path from v to w for each vertex w.

  All-source shortest path: For each pair (v, w) of vertices, find a shortest path from v to w.

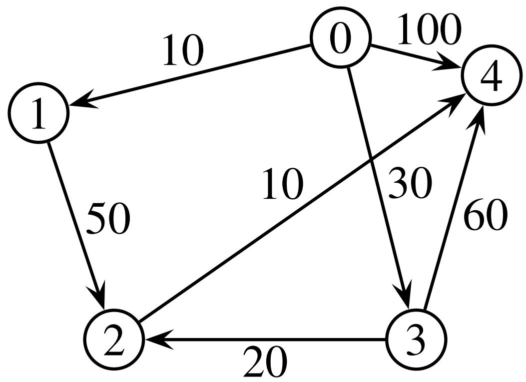
  Multistage graph optimization: For a given pair of vertices (the *source* and *sink*, respectively) in a special kind of graph known as a multistage graph, determine a shortest path from v to w.

4.1.2 The principle of optimality for shortest-path problems

- Roughly stated, the principle of optimality asserts that, in an optimal solution, any partial solution embedded in it must be an optimal solution for the corresponding subproblem.
- Relative to the single-source shortest-path problem, this translates as follows.
- If $\langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $v = v_0$ to $w = v_k$, then for any pair (i, j) with $0 \le i \le j \le k$, $\langle v_i, \ldots, v_j \rangle$ is a shortest path from v_i to v_j.

4.1.3 Dijkstra's algorithm for single-source shortest path

- Dijkstra's single-source shortest-path algorithm combines the principle of optimality with a "greedy style" selection process.



Source vertex = 0

Step	Other vertices allowed in path	To vertex 1	2	3	4	Nearest
1	{0}	10	∞	30	100	1
2	{0,1}	-	60	30	100	3
3	{0,1,3}	-	50	-	90	2
4	{0,1,2,3}	-	-	-	60	4
5	{0,1,2,3,4}	-	-	-	-	-

4.1.4 Implementation of Dijkstra's algorithm

Given: type *vertex* $= \{0, 1, \ldots, n-1\}$; /∗ 0 = source vertex ∗/
cost : array[*vertex*, *vertex*] of integer;
/∗ $cost[i,j] =$ cost of edge from i to j ∗/
/∗ $cost[i,j] = \infty$ if no such edge exists ∗/
/∗ $cost[i,j] = 0$ if $i = j$ ∗/
/∗ All costs must be nonnegative. ∗/

Build: *dist* : array[*vertex*] of integer;
path : array[*vertex*] of *vertex*;
/∗ $dist[i] =$ cost of a minimal path from 0 to i ∗/
/∗ $path[i] =$ vertex preceding i in the least-cost path from 0 to i ∗/

```
pool ← {1, 2, ..., n − 1};
for i ∈ vertex do
    ⟨ dist[i] ← cost[0, i];
      path[i] ← 0;
    ⟩;
while (pool ≠ ∅) do
      ⟨ i ← member of pool with dist[i] minimal;
        pool ← pool \ {i};
        for j ∈ pool do
            if dist[i] + cost[i, j] < dist[j]
               then ⟨ dist[j] ← dist[i] + cost[i, j];
                      path[j] ← i;
                    ⟩
      ⟩
```

- Dijkstra's algorithm is not formally a greedy algorithm; therefore a more direct proof of its completeness must be provided. The correctness follows from the following lemma.

4.1.5 Lemma *In the algorithm of 4.1.4, for each* $i \in \{1,2,\ldots,n-1\}$*,* $dist[i]$ *is the cost of a minimal path from* 0 *to* i *as soon as vertex* i *is deleted from pool.*

PROOF: The proof is by induction on the size of $\{0,1,2,\ldots,n-1\} \setminus pool$.

Basis: For $\mathsf{Card}(\{0,1,\ldots,n-1\} \setminus pool) = 1$, the assertion is obvious.

Step: Let k be such that $2 \leq k \leq n-1$ and suppose that the assertion is true whenever $\mathsf{Card}(\{0,1,\ldots,n-1\} \setminus pool) < k$. Let $i \in pool$ be the element selected at line 7 of the program, and let $\langle 0,\ldots,\ell,i \rangle$ be an optimal path from 0 to i. Then $\ell \notin pool$, (else the algorithm would have picked ℓ before i). Now by the inductive hypothesis, $dist[\ell]$ is the cost of a minimal path from 0 to ℓ. Hence, after execution of the if statement beginning on line 10, $dist[i] = dist[\ell] + cost[\ell,i]$; thus $dist[i]$ records the cost of a minimal path from 0 to i. □

4.1.6 Improved implementation and complexity of Dijkstra's algorithm

- If an adjacency list is used to represent the graph, the running time will clearly be $\Theta(n^2)$ in the average and worst case, in the doubly-nested loop at lines 6-14.
- A better approach is to mimic the implementation of Prim's algorithm which employs an adjustable priority queue.

- The pseudocode below shows such an implementation.
- It is very similar to the implementation of Prim's algorithm described in 3.5.26.
- Upon completion, for each vertex v aside from the source, the array *previous* will contain the identity of the vertex just before v in the path from the source to v.

```
foreach v ∈ vertex_set do
   ⟨ cost_to_source[v] ← ∞;
     in_queue[v] ← true;
   ⟩
cost_to_source[source_vertex] ← 0;
decrease_elt(M, source_vertex, 0);
while ( not (is_empty(M))) do
        ⟨ next_vertex ← retrieve_min(M);
          in_queue[next_vertex] ← false;
          foreach x ∈ adj_set[next_vertex] do
             if ((in_queue[x.id] = true)
                    and (x.dist + cost_to_source[next_vertex]
                           < cost_to_source[x.id]))
                then ⟨ cost_to_source[x.id] ←
                          x.dist + cost_to_source[next_vertex];
                       previous[x.id] ← next_vertex;
                       decrease_elt(M, x.id,
                                    cost_to_source[x.dist]);
                     ⟩
        ⟩
```

4.1.7 The complexity of the improved version of Dijsktra's algorithm *Dijkstra's algorithm may be realized with a worst-case running time of* $\Theta(n_E \cdot \log(n_V))$*, an average-case running time of* $\Theta(n_V \cdot \log(n_V))$*, and a best-case time of* $\Theta(n_E)$*, with* n_E *and* n_V *denoting the number denoting the number of edges and vertices in the graph, respectively.*

PROOF: Similar to that of 3.5.27. □

4.1.8 Floyd's algorithm for the all-source shortest path problem

- Assume that the graph has n vertices, is stored in an array *cost*, as described in 4.1.4.
- For each k, $0 \leq k \leq n$, define the array $A_k[0..n-1, 0..n-1]$ as follows:

$$A_k[i, j] = \text{cost of a minimal path from } i \text{ to } j$$
$$\text{with intermediate vertices lying in the set } [0..k-1].$$

- Note the following:

  1. $cost[i, j] = A_0[i, j]$.
  2. $A_{k+1}[i, j] = \min\{A_k[i, j], A_k[i,k] + A_k[k, j]\}$.
  3. The least-cost path from i to j is $A_n[i, j]$.

- The declarations and pseudocode:

```
/* Data types: */
 type vertex :      {0, ..., n−1};
 type ext_vertex :  {−1, 0, ..., n−1};
 /* Constants and variables */
 cost :  array[vertex, vertex] of real; /* Given */
 A :     array[vertex, vertex] of real; /* To be computed */
 path :  array[ext_vertex, ext_vertex] of vertex; /* To be computed */
/* Program Body: */
⟨ A ← cost;
  foreach i ∈ vertex do path[i] ← −1;
  for k ← 0 to n do
      for i ← 0 to n−1 do
          for j ← 0 to n−1 do
              if A[i,k] + A[k,j] < A[i,j]
                 then ⟨ A[i,j] ← A[i,k] + A[k,j];
                        path[i,j] ← k;
                      ⟩
⟩
/* To extract the least-cost path from i to j: */
procedure getpath(i,j : vertex) : string of vertex;
  ⟨ if path[i,j] < 0
      then return nil;
       else return getpath(i, path[i,j]) · path[i,j] · getpath(path[i,j], j)
  ⟩
```

4.1.9 The complexity of Floyd's algorithm *Floyd's algorithm for the all-source shortest path problem has time complexity* $\Theta(n^3)$ *in all cases, with n the number of vertices in the graph.* □

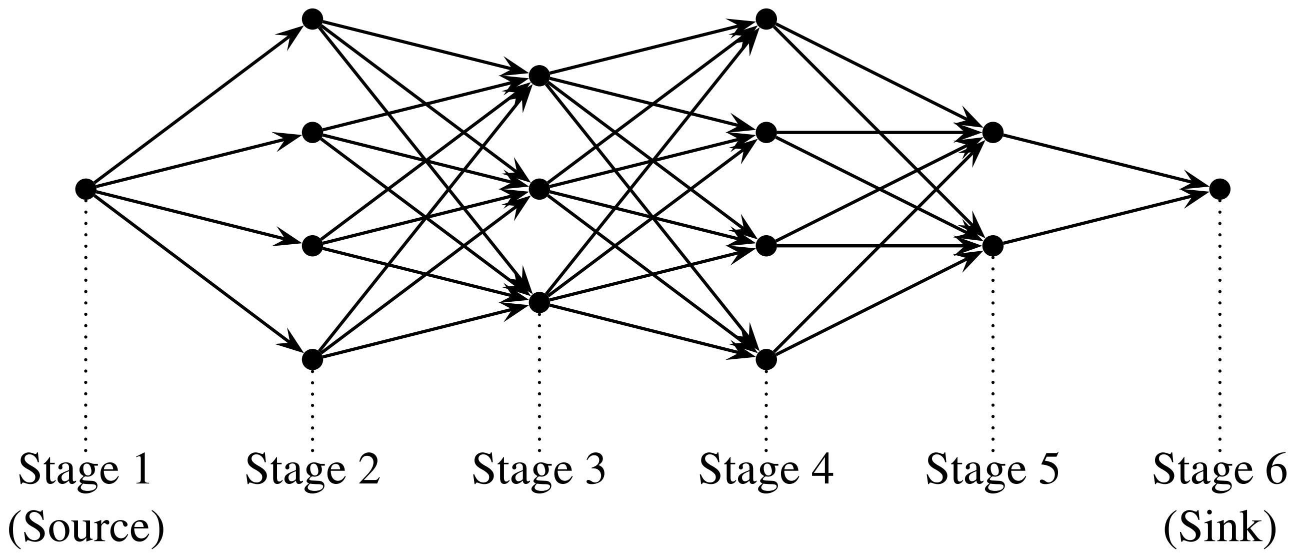
4.1.10 The principle of optimality and dynamic programming

- The *principle of optimality* states:
  - Any partial solution to a problem must be an optimal solution for the subproblem which it solves.
- Roughly, *dynamic programming* is a technique for solving optimization problems which makes explicit use of the principle of optimality.
- In contrast to the greedy method, there need not be a simple predictive strategy for determining which subproblem to solve.
- In Floyd's algorithm, the subproblem which is solved optimally is that of determining an optimal path $i \to j$ which only passes through the vertices in $\{0, 1, \ldots, k-1\}$.
- The solution through $\{0, 1, \ldots, k\}$ is built upon this previous solution.
- The problem of multistage graph optimization, which makes the idea of dynamic programming transparent, is considered next.

4.2 Multistage Graph Optimization

4.2.1 The idea of a weighted multistage graph

- The idea of a multistage graph is embodied in the picture below.



- Each edge has a nonnegative *cost* or *profit* associated with it.

Problem: Find a minimum cost (or maximum profit) path from the source to the sink.

4.2.2 A motivating application of multistage graph optimization

Given:

- r projects, numbered 1,2, …, r;
- m units of resource to be allocated;
- $p(i, j)$ = profit realized when j units of resource are applied to project i;
- Assume that $p(i,0) = 0$; $p(i, j) \geq 0$ always.

Goal: Allocate resources so as to maximize profit; *i.e.*, find an r-tuple $(x_1, x_2, \ldots, x_r) \in \mathbb{N}^r$ such that:

$$\sum_{i=1}^{r} p(i, x_i) \quad \text{is maximized, subject to:}$$

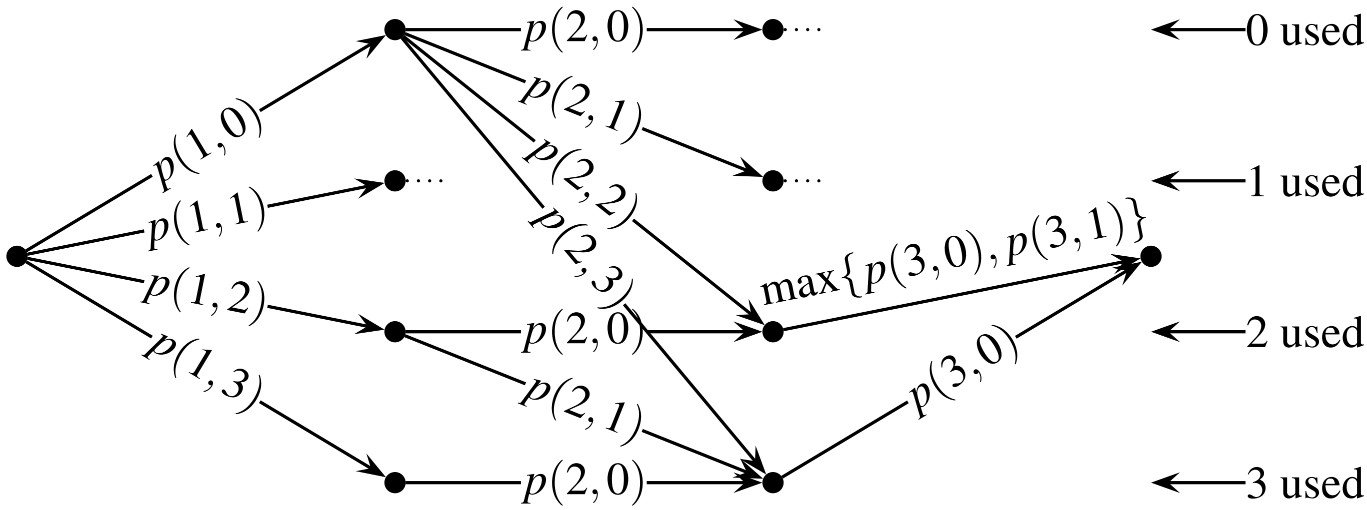
$$\sum_{i=1}^{r} x_i \leq m$$

Note: $p(i, j)$ is not assumed to be either:

- linear in j, or
- monotonic in j.

- This may easily be converted to a minimization problem, if so desired.

- The design of the corresponding multistage graph is as follows:
  - r projects $\Rightarrow r+1$ stages
  - Edges from stage i to stage $i+1$ correspond to resource allocation to project i.
  - m units of resource $\Rightarrow m+1$ vertices at stage i, $2 \leq i \leq r$.
  - There is one vertex at each stage for each possible quantity of resource used.
  - The edge weights are the profits $p(i, j)$.
- Shown below is an example for $r = 3$ and $m = 3$.



- Notes:
  - Only edges which use an amount of resource not exceeding that which is still available are included.
  - In the last step (into the sink), the optimal amount of resource is included for completion of the path.

4.2.3 The formal definition of a multistage graph

- A *multistage graph* is a pair $M = (G, \Pi)$ in which:

  (a) $G = (V, E, g)$ is a directed graph with the property that there is at most one edge connecting any two vertices.

  - (b) Π is an ordered partition $\langle V_1, V_2, \ldots, V_k \rangle$ of V with $k \geq 2$ such that:

    (i) $\mathsf{Card}(V_1) = \mathsf{Card}(V_k) = 1$.

    (ii) $\mathsf{Card}(V_i) \geq 1$ for $1 \leq i \leq k$.

    (iii) For each $e \in E$, $g(e) \in V_i \times V_{i+1}$ for some i, $1 \leq i \leq k-1$.

    (iv) For $v \in V_1$, $\mathsf{InDegree}(v) = 0$; $\mathsf{OutDegree}(v) = \mathsf{Card}(V_2)$.

    (v) For $v \in V_k$, $\mathsf{InDegree}(v) = \mathsf{Card}(V_{k-1})$; $\mathsf{OutDegree}(v) = 0$.

    (vi) For $v \in \{V_2, \ldots, V_{k-1}\}$, $\mathsf{InDegree}(v) \geq 1$; $\mathsf{OutDegree}(v) \geq 1$.

  - A *weighted multistage graph* is a multistage graph with non-negative integers (or possibly nonnegative real numbers) as weights on its edges.

Note: $\mathsf{InDegree}(v)$ (resp. $\mathsf{OutDegree}(v)$) denotes the number of edges which terminate (resp. begin) at vertex v.

4.2.4 The dynamic-programming solution to multistage graph optimization

- A path from the source to the sink is specified as a sequence $\langle v_1, v_2, \ldots, v_k \rangle$ of vertices with:
  - v_1 = source vertex;
  - v_k = sink vertex;
  - v_i is at stage i of the graph.
- The rôle of the principle of optimality in solving this problem is embodied in the following:
  - If $\langle v_1, v_2, \ldots, v_k \rangle$ is an optimal path (*i.e.*, yields maximum profit) from source to sink, then for any subpath

    $$\langle v_i, v_{i+1}, \ldots, v_{j-1}, v_j \rangle$$

    the profit along that path is maximal over all paths from v_i to v_j.
- It is assumed initially that the graph is represented by an $n \times n$ weight matrix, with n the number of vertices:

  $$\mathit{weight}: \ \text{array}[n,n] \text{ of } \text{integer};$$

- It is also assumed that the vertices are ordered by stage; *e.g.*:

Stage 1 = source	{1}
Stage 2	{2,3,4}
Stage 3	{5,6,7,8}
⋮	⋮
Stage k = sink	{n}

```
/* Data types and structures: */
type vertex = {1, 2, ..., n};
type stage = {1, 2, ..., k};
path :       array[stage] of vertex;
profit :     array[vertex] of  integer;
decision :   array[vertex] of  integer;
/* path records the optimal path as a sequence of vertices. */
/* profit[i]  =  profit along the optimal path from vertex i
                 to the sink. */
/* decision[i]  =  the vertex following vertex i in the optimal path
                   to the sink. */
/* Main procedure: */
profit ← 0;
for cur_vertex ← n − 1 downto 1 do
    ⟨ next_vertex ← vertex with
          weight[cur_vertex, next_vertex] + profit[next_vertex]
            maximized;
        decision[cur_vertex] ← next_veretex;
        profit[cur_vertex] ←
          weight[cur_vertex, next_vertex] + profit[next_vertex];
    ⟩
path[1] ← 1;
path[k] ← n;
for stage ← 2 to k − 1 do
    path[stage] ← decision[path[stage − 1]];
```

4.2.5 The complexity of multistage graph optimization *In all cases, the complexity of the multistage graph optimization algorithm described in 4.2.4 above is* $\Theta(n^2)$*, with n denoting the total number of vertices in the graph.*

PROOF: The process of selecting *next_vertex* at lines 13-14 requires a search of the list of vertices, which takes $\Theta(n)$ time. Thus, the for loop which encompasses lines 12-19 takes time $\Theta(n^2)$. The rest of the program runs in linear time. □

4.2.6 The complexity of resource allocation *Using the algorithm of 4.2.4, the problem of allocating m units of resource over r projects, as described in 4.2.2, requires time* $\Theta((mr)^2)$. □

4.2.7 Improving the performance of multistage graph optimization

- The performance may be improved substantially via the use of an adjacency list, similar to that employed in the improved implementations of Prim's algorithm 3.5.26 and of Dijkstra's algorithm 4.1.6.
- The amortized complexity over all executions of the assignment of lines 13-14 is $\Theta(E)$, with E denoting the total number of edges in the graph.
- Since $E \geq k - 1$, it follows that the overall complexity of this improved algorithm is $\Theta(E)$.
- The details are not presented here.

4.3 Dynamic-Programming Solution of the Discrete Knapsack Problem

4.3.1 Review of the Problem

Given:

- A knapsack with weight capacity M.
- n objects $\{\mathsf{obj}_1, \mathsf{obj}_2, \ldots, \mathsf{obj}_n\}$, each with a weight w_i and a value v_i.
- M, the w_i's, and the v_i's are all taken to be positive real numbers.

Find:

- $(x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ such that:
  (a) $\sum_{i=1}^{n} x_i \cdot \mathsf{v}_i$ is a maximum, subject to the constraint that
  (b) $\sum_{i=1}^{n} x_i \cdot \mathsf{w}_i \leq M$.

Example application:

- Knapsack = computer.
- capacity M = total time available.
- objects = potential jobs.
- w_i = time required to execute job_i.
- v_i = income earned by running job_i.
- The goal is to maximize the profit.

4.3.2 The idea of the dynamic programming solution

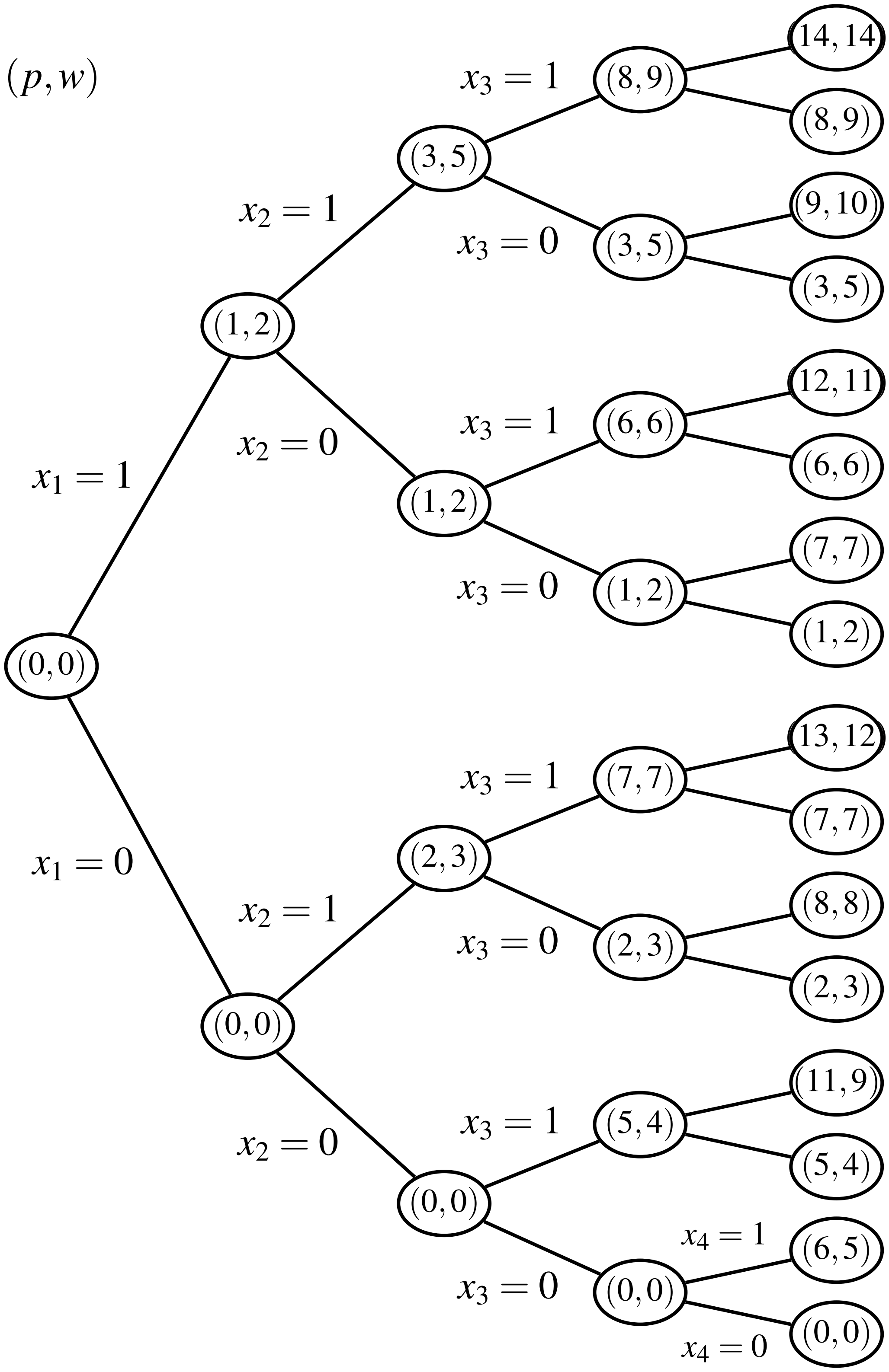
- For $Y \leq M$ and $1 \leq \ell \leq j \leq n$, let $\mathsf{Knap}(\ell, j, Y)$ denote the subproblem of the above knapsack problem with
  (i) knapsack capacity = Y;
  (ii) $j - \ell + 1$ objects $\{\mathsf{obj}_\ell, \ldots, \mathsf{obj}_j\}$.
- The weights and profits of the objects are unaltered.
- The problem is thus to find
  - $(x_\ell, x_{\ell+1}, \ldots, x_j) \in \{0,1\}^n$ such that:
    (a) $\sum_{i=\ell}^{j} x_i \cdot \mathsf{v}_i$ is a maximum, subject to the constraint that
    (b) $\sum_{i=\ell}^{j} x_i \cdot \mathsf{w}_i \leq Y$.
  - Note that $\mathsf{Knap}(1, n, M)$ is the original problem.
- Let $(y_1, y_2, \ldots, y_n) \in \{0,1\}^n$ be an optimal solution to the original problem. Note that:
  (a) If $y_n = 0$, then $(y_1, y_2, \ldots, y_{n-1})$ is an optimal solution for $\mathsf{Knap}(1, n-1, M)$.
  (b) If $y_n = 1$, then $(y_1, y_2, \ldots, y_{n-1})$ is an optimal solution for $\mathsf{Knap}(1, n-1, M - \mathsf{w}_n)$.
- This idea may be continued via induction to obtain the following:
  (c) For any k with $1 \leq k \leq n$, $(y_1, y_2, \ldots, y_k)$ is an optimal solution for $\mathsf{Knap}(1, k, M - \sum_{i=k+1}^{n} y_i \cdot \mathsf{w}_i)$.
- In the dynamic programming approach, instead of computing the cost of each partial solution, attention is restricted to those whose whose lead sequence (*e.g.* $(y_1, y_2, \ldots, y_k)$) is optimal for some "tail" $(y_{k+1}, y_{k+2} \ldots, y_n)$.

4.3.3 Solution of an example

- The example problem introduced in 3.1.3 is solved here using dynamic programming.
- For completeness, the data of the example are restated.
- Let $M = 8$; $n = 4$, and let v_i and w_i be as shown in the table below.

i	1	2	3	4
v_i	1	2	5	6
w_i	2	3	4	5

- The solution space is conveniently viewed as a *decision tree*, as illustrated on the next slide.


(p,w)
(0,0)
x1 = 1
x1 = 0
(1,2)
(0,0)
x2 = 1
x2 = 0
x2 = 1
x2 = 0
(3,5)
(1,2)
(2,3)
(0,0)
x3 = 1
x3 = 0
x3 = 1
x3 = 0
x3 = 1
x3 = 0
x3 = 1
x3 = 0
(8,9)
(3,5)
(6,6)
(1,2)
(7,7)
(2,3)
(5,4)
(0,0)
(14,14)
(8,9)
(9,10)
(3,5)
(12,11)
(6,6)
(7,7)
(1,2)
(13,12)
(7,7)
(8,8)
(2,3)
(11,9)
(5,4)
x4 = 1
(6,5)
x4 = 0
(0,0)

- The process builds partial solution vectors S_0, S_1, S_2, S_3, ..., with S_i corresponding to the i^{th} level in the decision tree (with the root at level 0).
- More specifically:
  - The notation $\binom{p}{w}$ is used to denote a profit-weight pair.
  - $S_0 = \binom{0}{0}$.
  - S_{i+1} = "merge" of S_i with S'_{i+1}, with
  - S'_{i+1} = S_i with $\binom{p_i}{w_i}$ added to each pair.
  - The merge operation removes suboptimal pairs.
- The following documents, in detail, the solution of the example.

<u>Step 0</u>: Fix $S_0 = \binom{0}{0}$.

<u>Step 1</u>: Find S_1.

  - First candidate = $\binom{0}{0} + \binom{1}{2} = \binom{1}{2}$.
  - Fill in the values from S_0 with lesser weight, yielding $S_1 = \binom{0}{0}$.
  - Include the candidate, if admissible, yielding $S_1 = \begin{pmatrix} 0 & 1 \\ 0 & 2 \end{pmatrix}$.

Step 2: Find S_2.

- First candidate = $\binom{0}{0} + \binom{2}{3} = \binom{2}{3}$.
- Fill in the values from S_1 of lesser weight, yielding $S_2 = \begin{pmatrix} 0 & 1 \\ 0 & 2 \end{pmatrix}$.
- Include the candidate, if admissible, yielding $S_2 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 2 & 3 \end{pmatrix}$.
- Second candidate = $\binom{1}{2} + \binom{2}{3} = \binom{3}{5}$.
- Fill in the values from S_1 of lesser weight (none new), yielding $S_2 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 2 & 3 \end{pmatrix}$.
- Include the candidate, if admissible, yielding $S_2 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 2 & 3 & 5 \end{pmatrix}$.

Step 3: Find S_3.

- First candidate = $\binom{0}{0} + \binom{5}{4} = \binom{5}{4}$.
- Fill in the values from S_2 of lesser weight, yielding $S_3 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 2 & 3 \end{pmatrix}$.
- Include the candidate, if admissible, yielding $S_3 = \begin{pmatrix} 0 & 1 & 2 & 5 \\ 0 & 2 & 3 & 4 \end{pmatrix}$.
- The suboptimal pair $\binom{3}{5}$ from S_2 is purged, since $\binom{5}{4}$ yields more profit with less cost. (A pair $\binom{p}{w}$ is suboptimal if there is another pair $\binom{p'}{w'}$ with either ($p < p'$ and $w' \leq w$) or ($p \leq p'$ and $w' < w$.))
- Second candidate = $\binom{1}{2} + \binom{5}{4} = \binom{6}{6}$.
- Fill in the values from S_2 of lesser weight (none new), yielding $S_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 5 \\ 0 & 2 & 3 & 5 & 4 \end{pmatrix}$.
- Include the candidate, if admissible, yielding $S_3 = \begin{pmatrix} 0 & 1 & 2 & 5 & 6 \\ 0 & 2 & 3 & 4 & 6 \end{pmatrix}$.
- Third candidate = $\binom{2}{3} + \binom{5}{4} = \binom{7}{7}$.
- Fill in the values from S_2 of lesser weight (none new), yielding $S_3 = \begin{pmatrix} 0 & 1 & 2 & 5 & 6 \\ 0 & 2 & 3 & 4 & 6 \end{pmatrix}$.
- Include the candidate, if admissible, yielding $S_3 = \begin{pmatrix} 0 & 1 & 2 & 5 & 6 & 7 \\ 0 & 2 & 3 & 4 & 6 & 7 \end{pmatrix}$.

- Fourth candidate = $\binom{3}{5} + \binom{5}{4} = \binom{8}{9}$.
- Note that $\binom{3}{5}$ was purged from S_3, but it remains in S_2, and must be used to construct candidates for S_3.
- Fill in the values from S_2 of lesser weight (none new), yielding $S_3 = \begin{pmatrix} 0 & 1 & 2 & 5 & 6 & 7 \\ 0 & 2 & 3 & 4 & 6 & 7 \end{pmatrix}$.
- Include the candidate, if admissible; however, it is not admissible, so the value remains $S_3 = \begin{pmatrix} 0 & 1 & 2 & 5 & 6 & 7 \\ 0 & 2 & 3 & 4 & 6 & 7 \end{pmatrix}$.

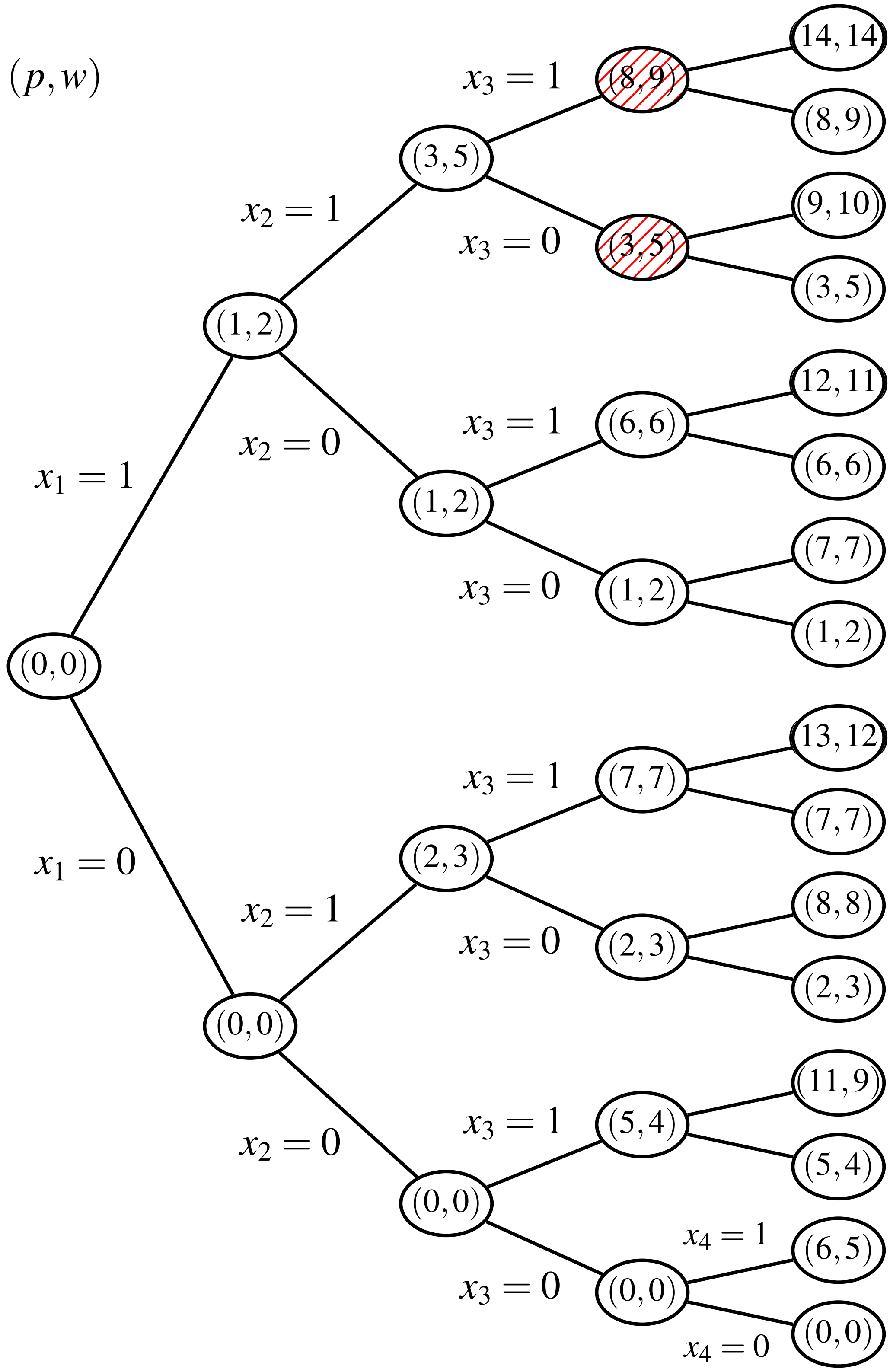
Step 4: Find the value of x_4.

- Although S_4 could be compute in a manner similar to that above, it is possible to find it directly.
- For $x_4 = 0$, the best pair is $\binom{7}{7}$.
- For $x_4 = 1$, find the $\binom{p}{w} \in S_3$ with the maximum p and with $\mathsf{w}_4 + w = 5 + w \leq M = 8$.
- The choice is $\binom{p}{w} = \binom{2}{3}$, which yields $\binom{8}{8}$. This is better than $\binom{7}{7}$, so $x_4 = 1$.

Step 5: Find the solution vector (x_1, x_2, x_3, x_4).

- It is already known that $x_4 = 1$ with $\binom{8}{8}$ representing the total profit and weight.
- Thus, since $\mathsf{v}_4 = 6$ and $\mathsf{w}_4 = 5$, $\binom{8}{8} - \binom{6}{5} = \binom{2}{3}$ must be matched by (x_1, x_2, x_3).
- Since $\mathsf{w}_3 = 4$, $x_3 = 0$.
- Since $\binom{\mathsf{v}_2}{\mathsf{w}_2} = \binom{2}{3}$, $x_2 = 1$, whence $x_1 = 0$.
- Thus, $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$.

- The final, pruned decision tree is shown on the next page.


(p,w)
$x_1=1$
$x_1=0$
$x_2=1$
$x_2=0$
$x_2=1$
$x_2=0$
$x_3=1$
$x_3=0$
$x_3=1$
$x_3=0$
$x_3=1$
$x_3=0$
$x_3=1$
$x_3=0$
$x_4=1$
$x_4=0$
(0,0)
(1,2)
(0,0)
(3,5)
(1,2)
(2,3)
(0,0)
(8,9)
(3,5)
(6,6)
(1,2)
(7,7)
(2,3)
(5,4)
(0,0)
(14,14)
(8,9)
(9,10)
(3,5)
(12,11)
(6,6)
(7,7)
(1,2)
(13,12)
(7,7)
(8,8)
(2,3)
(11,9)
(5,4)
(6,5)
(0,0)

4.3.4 Skeletal representation of the algorithm

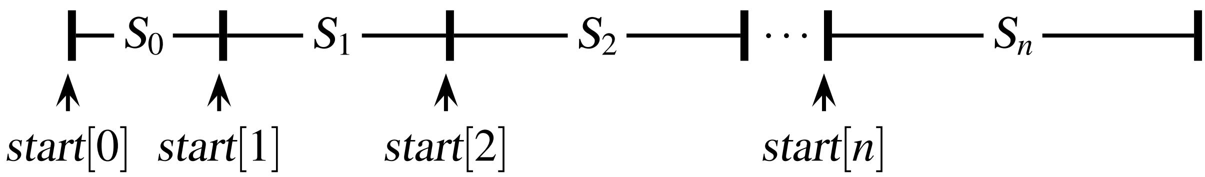
- Assume that there are n objects.
- The skeletal algorithm is as follows.

  $S_0 = \{(0,0)\}$;
  for $i \leftarrow 1$ **to** $n-1$ **do**
  $\langle$ $T \leftarrow$new admissible pairs (p,w)
  found by adding $(\mathsf{v}_i, \mathsf{w}_i)$ to S_{i-1};
  $S_i \leftarrow$ *merge-purge*(S_{i-1}, T);
  $\rangle$
  Select optimal pairs from S_n;
  Trace back to find $(x_1, x_2, \ldots, x_n)$;

- The major data structures are as follows:

  type *solution_pair* =**record**
  profit : $\{0, 1, \ldots, max\_profit\}$;
  weight : $\{0, 1, \ldots, max\_weight\}$;
  end record;
  constant *max_size* $= 2^{n-1} - 1$; /* Max vertices in decision tree */
  s : array$[1..max_size]$ of *sol_pair*;
  start : *array*$[0..n]$ of $\{0, 1, \ldots, max\_size\}$;

- *start*$[i]$ identifies the starting point of $|S|_i$ in the array s:



- The full algorithm will not be presented here.

- A complexity analysis is nonetheless possible.

4.3.5 Complexity of dynamic programming applied to the discrete knapsack problem

- The following apply in the worst case:
  - The time required to produce S_i is $\Theta(\mathsf{Card}(S_{i-1}))$.
  - In the worst case, all vertices of the decision tree are retained, so $\mathsf{Card}(S_i) = 2 \cdot \mathsf{Card}(S_{i-1})$.
  - Thus, the worst-case time to produce all of the S_i's, $0 \leq i \leq n-1$ is
$$\Theta(\sum_{i=0}^{n-1} S_{i-1}) = \Theta(\mathbf{2}^n)$$
  - In a typical case, however, many pairs are purged, and so the performance may be much better.
  - The space complexity is also $\Theta(\mathbf{2}^n)$.

4.3.6 Some heuristics for speedup

- There are a number of heuristics which may be employed to speed up the solution of many instances of the discrete knapsack problem.

- Suppose that a lower bound L is given on the profit of an optimal solution; *i.e.*,

$$(y_1, y_2, \ldots, y_n) \text{ optimal } \Rightarrow \sum_{i=1}^{n} y_i \cdot \mathsf{v}_i \geq L$$

- For each k, $1 \leq k \leq n$, define

$$\mathsf{PLeft}(k) = \sum_{j=k+1}^{n} \mathsf{v}_k$$

- The following heuristic may then be employed:

$$\text{If } \binom{p}{w} \in S_i \text{ and } p + \mathsf{PLeft}(i) < L, \text{ then purge } \binom{p}{w}$$

- There are a number of ways to obtain such an L:

  - Use $\max\{p \mid \binom{p}{w} \in S_i\}$ as the bound L for the i^{th} stage.
  - Obtain a feasible solution using a greedy method, and use the resulting profit as the bound L.

4.4 The Travelling Salesman Problem

4.4.1 Problem description

- The *travelling-salesman problem*, often abbreviated *TSP*, may be described as follows.

Given: A directed graph $G = (V, E, g)$, together with a weighting function $d : E \to \mathbb{N}$.

- Think of d as giving a distance between vertices.

Define: A *tour* of G is a simple cycle of G which passes through each vertex of G. The *cost* of a tour is the sum of the distances of its edges.

Find: A tour of minimum cost.

- Note: By definition, a tour passes through each vertex exactly once.

4.4.2 The combinatorics of the travelling salesman problem

Given: A directed graph $G = (V, E, g)$, together with a weighting function $d : E \to \mathbb{N}$.

Question: How many distinct tours of G are there?

Answer:

- First, assume that the graph is complete; *i.e.*, that there is an edge between any two vertices.
- Since a tour must pass through all vertices, the start vertex may be chosen arbitrarily.

- The second vertex may be chosen in any of $n_V - 1$ ways, with n_V denoting the number of vertices in the graph.
- The third vertex may be chosen in any of $n_V - 2$ ways.
- The k^{th} vertex may be chosen in any of $n_v - (k+1)$ ways.
- Thus, there are

  $$(n_V - 1) \cdot (n_V - 2) \cdot \ldots \cdot 2 \cdot 1 = (n_V - 1)!$$

  possible tours.
- Since $n!$ is the number of permutations of n elements, the TSP is often called a *permutation problem*.
- On the other hand, problems whose solution space is on the order of 2^n, such as the discrete knapsack problem, are often called *subset problems*.
- Permutation problems often have worst-case complexity which is even greater than that of subset problems, since

  $$\Theta(2^n) \subsetneq \Theta(n!)$$

- This is easily seen by comparing the following sequences:

  $$\begin{array}{rcccccccccccc} 2^n & = & 2 & \cdot & 2 & \cdot & 2\cdot & \ldots & \cdot & 2 & \cdot & 2 \\ n! & = & 1 & \cdot & 2 & \cdot & 3\cdot & \ldots & \cdot & n-1 & \cdot & n \end{array}$$

- Note, however, that a graph has $(n_V - 1)!$ possible tours iff it is complete.
- In practice, the number of possible tours may be far less.

4.4.3 The principle of optimality applied to the travelling salesman problem

- Let $\langle v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)}, v_{\sigma(1)} \rangle$ be the sequence of vertices followed in an optimal tour.
- Then $\langle v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)} \rangle$ must be a shortest path from $v_{\sigma(1)}$ to $v_{\sigma(n)}$ which passes through each vertex exactly once.
- Invoking the principle of optimality, for any i, j, with $1 \leq i \leq j \leq n$, the path $\langle v_{\sigma(i)}, v_{\sigma(i+1)}, \ldots, v_{\sigma(j)} \rangle$ must be optimal for all paths beginning at $v_{\sigma(i)}$, ending at $v_{\sigma(j)}$, and passing through exactly the intermediate vertices $\{v_{\sigma(i+1)}, \ldots, v_{\sigma(j-1)}\}$.
- In general, for $v, w \in V$ and $S \subseteq V \setminus \{v, w\}$, define

  $$\mathsf{TSP}(v, S, w)$$

  to be the shortest path from v to w which passes through each vertex in S exactly once, and through no other intermediate vertices.
- Define

  $$\mathsf{Cost}(v, S, w)$$

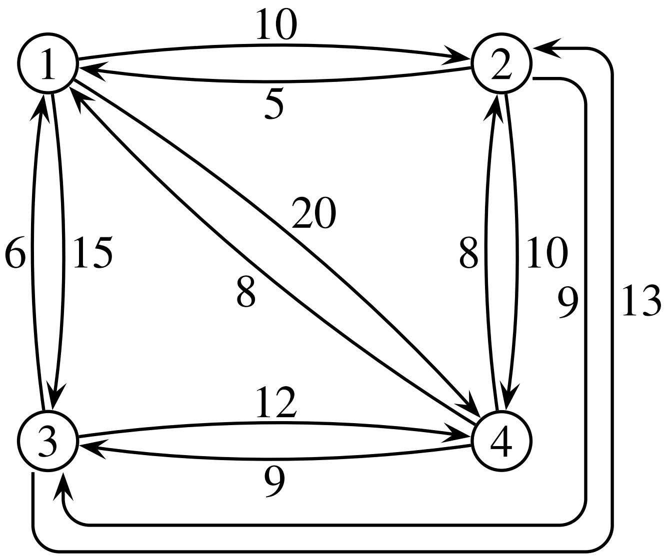
  to be the cost of such a path.
- For $u, v \in V$, let $d(u, v)$ denote the distance of the minimal edge between u and v. Thus,

  $$d(u, v) = \mathsf{Cost}(u, \emptyset, v)$$

- By the principle of optimality, for $u \in S$,

  $$\mathsf{Cost}(v, S, w) = \min(\{d(v, u) + \mathsf{Cost}(u, S \setminus \{u\}, w) \mid u \in S\}) \quad (*)$$

4.4.4 Example



Distance Matrix					
To:		1	2	3	4
From:	1	0	10	15	20
	2	5	0	9	10
	3	6	13	0	12
	4	8	8	9	0

- Choose vertex 1 as the terminal point (arbitrary choice).
- Process intermediate sets in order of increasing size.
- For intermediate set $S = \emptyset$:

$$\begin{aligned}\mathsf{Cost}(2,\emptyset,1) &= d(2,1) = 5\\ \mathsf{Cost}(3,\emptyset,1) &= d(3,1) = 6\\ \mathsf{Cost}(4,\emptyset,1) &= d(4,1) = 8\end{aligned}$$

- For $\mathsf{Card}(S) = 1$:

$$\begin{aligned}\mathsf{Cost}(2,\{3\},1) &= d(2,3) + \mathsf{Cost}(3,\emptyset,1) = 15\\ \mathsf{Cost}(2,\{4\},1) &= d(2,4) + \mathsf{Cost}(4,\emptyset,1) = 18\\ \mathsf{Cost}(3,\{2\},1) &= d(3,2) + \mathsf{Cost}(2,\emptyset,1) = 18\\ \mathsf{Cost}(3,\{4\},1) &= d(3,4) + \mathsf{Cost}(4,\emptyset,1) = 20\\ \mathsf{Cost}(4,\{2\},1) &= d(4,2) + \mathsf{Cost}(2,\emptyset,1) = 13\\ \mathsf{Cost}(4,\{3\},1) &= d(4,3) + \mathsf{Cost}(3,\emptyset,1) = 15\end{aligned}$$

- For $\mathsf{Card}(S) = 2$:

$$\begin{aligned}
&\mathsf{Cost}(2,\{3,4\},1)\\
&\quad = \min(\{d(2,3)+\mathsf{Cost}(3,\{4\},1),\ d(2,4)+\mathsf{Cost}(4,\{3\},1)\}\\
&\quad = \min(\{\{9+20\},\{10+15\}\}) = 25\\
&\mathsf{Cost}(3,\{2,4\},1)\\
&\quad = \min(\{d(3,2)+\mathsf{Cost}(2,\{4\},1),\ d(3,4)+\mathsf{Cost}(4,\{2\},1)\}\\
&\quad = \min(\{\{13+18\},\{12+13\}\}) = 25\\
&\mathsf{Cost}(4,\{2,3\},1)\\
&\quad = \min(\{d(4,2)+\mathsf{Cost}(2,\{3\},1),\ d(4,3)+\mathsf{Cost}(3,\{2\},1)\}\\
&\quad = \min(\{\{8+15\},\{9+18\}\}) = 23
\end{aligned}$$

- For $\mathsf{Card}(S) = 3$, attention may be restricted to paths starting with vertex 1, since the cycle will be completed at this point.

$$\begin{aligned}
&\mathsf{Cost}(1,\{2,3,4\},1)\\
&\quad = \min(\{d(1,2)+\mathsf{Cost}(2,\{3,4\},1),\\
&\qquad\qquad d(1,3)+\mathsf{Cost}(3,\{2,4\},1),\\
&\qquad\qquad d(1,4)+\mathsf{Cost}(4,\{2,3\},1)\})\\
&\quad = \min(10+25,\ 15+25,\ 20+23) = 35
\end{aligned}$$

- In general, the rule (*) of 4.4.3 is applied repeatedly to subproblems with increasing size of S.

- To see the size of this computation, proceed as follows.
- Recall that for $k \leq n$, the *binomial coefficient*

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

  gives the number of distinct k-element subsets of a set of n elements.

- Thus, the total number of values of the form $\mathsf{Cost}(v, S, v_t)$, which must be computed with this method, with v_t the terminal vertex in the tour, is:

$$(n-1) \cdot \sum_{k=0}^{n-2} \binom{n-2}{k} \quad + \quad 1$$

- However, by the *binomial theorem*:

$$\sum_{k=0}^{n-2} \binom{n-2}{k} = \sum_{k=0}^{n-2} \left(\binom{n-2}{k} \cdot 1^k \cdot 1^{n-2k} \right) = (1+1)^{n-2} = 2^{n-2}$$

- Thus, the total number of computations of the form $\mathsf{Cost}(v, S, v_t)$ is $(n-1) \cdot 2^{n-2} + 1$.
- These require worst-case time $\Theta(n)$ to compute, hence the total running time will be

$$\Theta(n^2 \cdot 2^n)$$

  in the worst case.

- This is better than $\Theta((n-1)!)$, but it shall soon be shown that there are better algorithms.

- Note also that this approach requires $\Theta(n \cdot 2^n)$ space, since all values of the form $\mathsf{Cost}(v, S, v_t)$ must be saved for a given cardinality of S, in order to compute the paths for $\mathsf{Card}(S) + 1$.
- The associated path must also be saved.
- This is prohibitively expensive.