

## Lecture 8: dynamic programming

- Knapsack
- Dynamic programming approach to knapsack
- A practical example for knapsack
- Dijkstra's algorithm revisited
- Dynamic programming idea behind Dijkstra's algorithm
- How to construct dynamic programming algorithms
- Landing scheduling via dynamic programming
- Travelling salesman

## Knapsack problem

How to pack as much value with a weight constraint  $W$ ?

$$\begin{array}{ll} \mathbf{max:} & \sum_{i=1}^p u_i x_i \\ \mathbf{s.t.} & \sum_{i=1}^p w_i x_i \leq W \\ & x_i \in \{0, 1\} \quad \text{for all } i \end{array}$$

$p$  number of items  
 $i$  index of items  
 $W$  maximum weight to be carried  
 $w_i$  weight of object  $i$   
 $u_i$  value of object  $i$   
 $x_i$  decision variable to take ( $x_i = 1$ ) or not to take ( $x_i = 0$ ) object  $i$

## Dynamic programming solution of knapsack

Let us index by  $i$  the items.  
 Let us index by  $j$  the weight restriction.

Question (to be answered by induction)

- If I can take objects 1, 2, 3, ...  $i$ ,
- How much value can I take away
- Given that I am restricted to take a maximum weight of  $j$

This question is to be answered by induction, on  $i$  AND  $j$

## Dynamic programming induction relation

Introduce a quantity  $d(i,j)$ , indexed by

-  $i$ , the number of items to be taken away

→  $i = 0, 1, 2, \dots, p$

-  $j$ , the weight restriction

→  $j = 1, 2, \dots, W$

Note that all quantities of this problem have to be integer

Introduce  $d(i,j)$ , the maximum value of the selected items, if we are allowed to take items 1 to  $i$ , and we have a weight restriction of  $j$

We will thus compute  $d(i,j)$  recursively, in an array of dimension  $p \times W$ .

## Dynamic programming induction relation

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j - w_i)\}$$

maximum value of the selected items, if we are allowed to take items 1 to  $i$ , and we have a weight restriction of  $j$

## Dynamic programming induction relation

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j - w_i)\}$$

maximum value of the selected items, if we are allowed to take items 1 to  $i$ , and we have a weight restriction of  $j$

value of for  $i-1$  items, with restriction  $j$  minus weight of object  $i$

Dynamic programming induction relation

$$d(i, j) = \max \{ d(i-1, j), u_i + d(i-1, j-w_i) \}$$

maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j  
 value of for i-1 items, with restriction j minus weight of object i  
 +  
 value of item i

Dynamic programming induction relation

$$d(i, j) = \max \{ d(i-1, j), u_i + d(i-1, j-w_i) \}$$

maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j  
 keep i-1 items (do nothing)  
 value of for i-1 items, with restriction j minus weight of object i  
 +  
 value of item i

Dynamic programming induction relation

$$d(i, j) = \max \{ d(i-1, j), u_i + d(i-1, j-w_i) \}$$

keep i-1 items (do nothing)  
 value of for i-1 items, with restriction j minus weight of object i  
 +  
 value of item i  
 maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

Dynamic programming induction relation

Best option: - keep the previous selection ?  
- add the new object ?

$$d(i, j) = \max \{ d(i-1, j), u_i + d(i-1, j-w_i) \}$$

keep i-1 items (do nothing)  
 value of for i-1 items, with restriction j minus weight of object i  
 +  
 value of item i  
 maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

Dynamic programming induction relation

Best option: - keep the previous selection ?  
- add the new object ?

$$d(i, j) = \max \{ d(i-1, j), u_i + d(i-1, j-w_i) \}$$

keep i-1 items (do nothing)  
 value of for i-1 items, with restriction j minus weight of object i  
 +  
 value of item i  
 maximum value of the selected items, if we are allowed to take items 1 to i, and we have a weight restriction of j

What do we do next?

We fill an array of size p x W

$$\begin{matrix} w_1 = 1 \\ w_2 = 7 \\ w_3 = 4 \\ w_4 = 2 \end{matrix}$$

Weight = value  
 p=4  
 W=11 (arbitrary, but less than 14 obviously)

$$d(i, j) = \max \{ d(i-1, j), u_i + d(i-1, j-w_i) \}$$

### What do we do next?

We fill an array of size  $p \times W$

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j-w_i)\}$$

Weight restriction  $j (j=1, 2, \dots, W)$

Items picked (i=1, 2, 3, 4, ...)		1	2	3	4	5	6	7	8	9	10	11
1												
2												
3												
4												

### We fill an array of size $p \times W$

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j-w_i)\}$$

Weight restriction  $j (j=1, 2, \dots, W)$

Items picked (i=1, 2, 3, 4, ...)		1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1	1
2		{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}
3		{1}	{1}	{1}	{1}	{1}	{1}	{2}	{2,1}	{2,1}	{2,1}	{2,1}
4		{1}	{1}	{1}	{3}	{3,1}	{3,1}	{2}	{2,1}	{2,1}	{2,1}	{3,2}

### We fill an array of size $p \times W$

$$w_1 = 1, w_2 = 7, w_3 = 4, w_4 = 2$$

$$d(i, j) = \max \{ \text{[red box]}, u_i + d(i-1, j-w_i) \}$$

Weight restriction  $j (j=1, 2, \dots, W)$

Items picked (i=1, 2, 3, 4, ...)		1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1	1
2		{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}
3		{1}	{1}	{1}	{1}	{1}	{1}	{2,1}	{2,1}	{2,1}	{2,1}	{2,1}
4		{1}	{1}	{1}	{3}	{3,1}	{3,1}	{2,1}	{2,1}	{2,1}	{3,2}	{3,2}

### We fill an array of size $p \times W$

$$w_1 = 1, w_2 = 7, w_3 = 4, w_4 = 2$$

$$d(i, j) = \max \{ \text{[red box]}, u_i + d(i-1, j-w_i) \}$$

Weight restriction  $j (j=1, 2, \dots, W)$

Items picked (i=1, 2, 3, 4, ...)		1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1	1
2		{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}
3		{1}	{1}	{1}	{1}	{1}	{1}	{1}	{2,1}	{2,1}	{2,1}	{2,1}
4		{1}	{1}	{1}	{3}	{3,1}	{3,1}	{2,1}	{2,1}	{2,1}	{3,2}	{3,2}

### We fill an array of size $p \times W$

$$w_1 = 1, w_2 = 7, w_3 = 4, w_4 = 2$$

$$d(i, j) = \max \{d(i-1, j), \text{[red box]}\}$$

Weight restriction  $j (j=1, 2, \dots, W)$

Items picked (i=1, 2, 3, 4, ...)		1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1	1
2		{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}
3		{1}	{1}	{1}	{1}	{1}	{1}	{2}	{2,1}	{2,1}	{2,1}	{2,1}
4		{1}	{1}	{1}	{3}	{3,1}	{3,1}	{2,1}	{2,1}	{2,1}	{3,2}	{3,2}

### We fill an array of size $p \times W$

$$w_1 = 1, w_2 = 7, w_3 = 4, w_4 = 2$$

$$d(i, j) = \max \{d(i-1, j), \text{[red box]}\}$$

Weight restriction  $j (j=1, 2, \dots, W)$

Items picked (i=1, 2, 3, 4, ...)		1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1	1
2		{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}
3		{1}	{1}	{1}	{1}	{1}	{1}	{2}	{2,1}	{2,1}	{2,1}	{2,1}
4		{1}	{1}	{1}	{3}	{3,1}	{3,1}	{2,1}	{2,1}	{2,1}	{3,2}	{3,2}

We fill an array of size  $p \times W$

$$w_1 = 1, w_2 = 7, w_3 = 4, w_4 = 2$$

No uniqueness: could pick also item 2

Weight restriction  $j$  ( $j=1,2, \dots, W$ )

Items picked ( $i=1, 2, 3, 4, \dots$ )

	1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1
2	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}
3	1	1	1	4	5	5	7	8	8	8	8
4	{1}	{1}	{1}	{3}	{3,1}	{3,1}	{2}	{2,1}	{2,1}	{2,1}	{3,2}
5	1	2	3	4	5	6	7	8	9	10	11
6	{1}	{4}	{4,1}	{3}	{3,1}	{4,3}	{2,1}	{4,2}	{4,2,1}	{3,2}	

What do we do next?

We fill an array of size  $p \times W$

$$d(i, j) = \max \{d(i-1, j), u_i + d(i-1, j-w_i)\}$$

Weight restriction  $j$  ( $j=1,2, \dots, W$ )

Items picked ( $i=1, 2, 3, 4, \dots$ )

	1	2	3	4	5	6	7	8	9	10	11
1											
2											
3											
4											

Final result

Weight restriction  $j$  ( $j=1,2, \dots, W$ )

Items picked ( $i=1, 2, 3, 4, \dots$ )

	1	2	3	4	5	6	7	8	9	10	11
1	1	1	1	1	1	1	1	1	1	1	1
2	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}	{1}
3	1	1	1	4	5	5	7	8	8	8	8
4	{1}	{1}	{1}	{3}	{3,1}	{3,1}	{2}	{2,1}	{2,1}	{2,1}	{3,2}
5	1	2	3	4	5	6	7	8	9	10	11
6	{1}	{4}	{4,1}	{3}	{3,1}	{4,3}	{4,3,1}	{2,1}	{4,2}	{4,2,1}	{3,2}

Dynamic programming: main idea

Main idea: when solving an integer program, in order to avoid enumeration (to expensive computationally), cut the problem in two and use induction:

At current step: assume you know the best solution at previous steps

Compute the best solution for the current step, and pair it with the solution at the previous steps

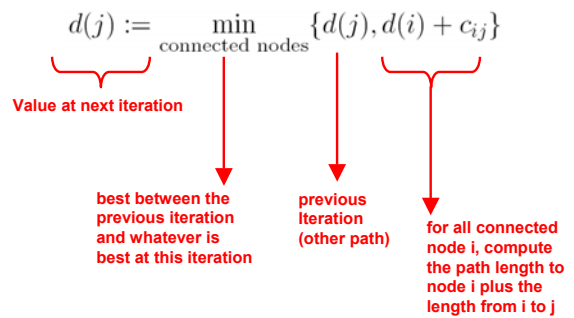
A more precise definition of dynamic programming will be given later in class.

Dijkstra's algorithm revisited

```

begin
  S := ∅
  d(i) := +∞ for each node i
  d(s) := 0 and pred(s) = 0
  while |S| < n do
    begin
      let i in S* for which d(i) = min{d(j), j in S*}
      S = S ∪ {i}
      S* = S* \ {i}
    end
  end
  d(j) := min_{connected nodes} {d(j), d(i) + c_{ij}}
end
  
```

Core of Dijkstra's algorithm



This is the main idea under dynamic programming algorithms

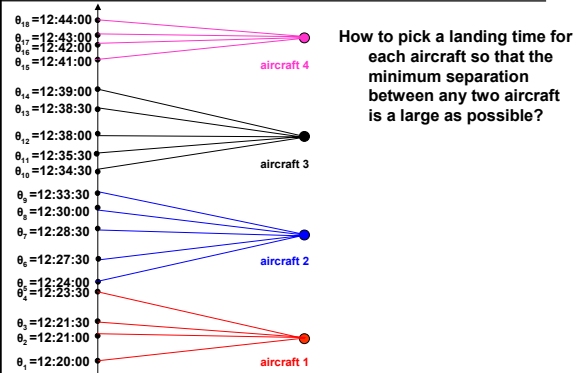
## How to construct dynamic programming algorithms

- 1) View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.
- 2) Define the state as a summary of all relevant decisions
- 3) Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.
- 4) Write a recursion on the optimal cost from the state to a state

square indicates that time is meant for the algorithm (more about this in the lab).

[Introduction to linear optimization, Bertsimas, Tsitsiklis, 1997]

## Landing scheduling through dynamic programming



## Landing scheduling through dynamic programming

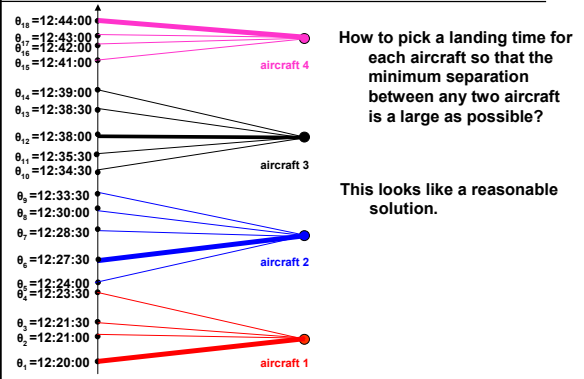
Example of data used for this type of problem:

```
ATA001 1295 1305 1310 1311 1316 1320 1325 1327 1335
UAL002 1413 1423 1429 1439 1447 1458 1468 1478 1489
DAL003 1522 1532 1541 1551 1557 1567 1581 1587 1601
UAL004 1606 1613 1619 1629 1638 1648 1659 1673 1682
COA005 1693 1700 1705 1710 1710 1715 1720 1725 1730
SWA006 1787 1794 1799 1799 1804 1809 1814 1819 1824
```



[TCSim, T. Callantine, NASA Ames]

## Landing scheduling through dynamic programming



## Landing scheduling through dynamic programming

$t_{ij}$  j-th possible landing time of aircraft i  
 $n_i$  number of possible landing times for aircraft i  
 $\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

- Aircraft 1 should obviously arrive as early as possible
- If aircraft 2 is assigned the j-th arrival time, the spacing between aircraft i and j is obviously

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

## Landing scheduling through dynamic programming

$t_{ij}$  j-th possible landing time of aircraft i  
 $n_i$  number of possible landing times for aircraft i  
 $\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

Recursion

$$\delta(i, j) = \max_{j'=1}^{n_i-1} \{ \min \{ t_{i,j} - t_{i-1,j'}, \delta(i-1, j') \} \}$$

### Landing scheduling through dynamic programming

$$\delta(i, j) = \max_{j'=1}^{n_i-1} \{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1, j')\} \}$$

Recursion is done with variable i, i.e. with the number of aircraft. For a number i of aircraft, this represents the largest smallest spacing if aircraft i arrives at time j

$t_{ij}$  j-th possible landing time of aircraft i

$n_i$  number of possible landing times for aircraft i

$\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

### Landing scheduling through dynamic programming

$$\delta(i, j) = \max_{j'=1}^{n_{i-1}} \{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1, j')\} \}$$

Maximal minimum separation between any two aircraft within the first i-1 aircraft, if aircraft i-1 is assigned arrival time j'

$t_{ij}$  j-th possible landing time of aircraft i

$n_i$  number of possible landing times for aircraft i

$\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

### Landing scheduling through dynamic programming

$$\delta(i, j) = \max_{j'=1}^{n_{i-1}} \{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1, j')\} \}$$

Time separation resulting from assigning aircraft i to its j-th arrival time and aircraft i-1 to its j'-th arrival time.

$t_{ij}$  j-th possible landing time of aircraft i

$n_i$  number of possible landing times for aircraft i

$\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

### Landing scheduling through dynamic programming

$$\delta(i, j) = \max_{j'=1}^{n_{i-1}} \{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1, j')\} \}$$

For a given j' (aircraft i-1's arrival time), the overall resulting minimum separation is the worst between  
- aircraft i and aircraft i-1  
- any two other aircraft within 1, 2, ..., i-1

$t_{ij}$  j-th possible landing time of aircraft i

$n_i$  number of possible landing times for aircraft i

$\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

### Landing scheduling through dynamic programming

$$\delta(i, j) = \max_{j'=1}^{n_{i-1}} \{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1, j')\} \}$$

For aircraft i, we have to compute the maximum spacing so far when trying all possible assignments for the previous i-1 aircraft

$t_{ij}$  j-th possible landing time of aircraft i

$n_i$  number of possible landing times for aircraft i

$\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

### Landing scheduling through dynamic programming

$$\delta(i, j) = \max_{j'=1}^{n_{i-1}} \{ \min\{t_{i,j} - t_{i-1,j'}, \delta(i-1, j')\} \}$$

For aircraft i, we have to compute the maximum spacing so far when trying all possible assignments for the previous i-1 aircraft

$t_{ij}$  j-th possible landing time of aircraft i

$n_i$  number of possible landing times for aircraft i

$\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{ij}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

### Landing scheduling through dynamic programming

$$\delta(i, j) = \max_{j'=1}^{n_{i-1}} \{ \min \{ t_{i,j} - t_{i-1,j'}, \delta(i-1, j') \} \}$$

The goal of the dynamic algorithm is to compute the largest spacing, by trying the best among all possible assignments of aircraft i-1 (since this is the induction step between i-1 and i).

$t_{i,j}$  j-th possible landing time of aircraft i

$n_i$  number of possible landing times for aircraft i

$\delta(i, j)$  Maximal minimum spacing between any two aircraft, for the subset of aircraft 1, 2, ..., i, if the aircraft number i is assigned the arrival time  $t_{i,j}$

Initialization of the recursion:

$$\delta(2, j) = t_{2,j} - t_{1,1}$$

### How to construct dynamic programming algorithms

- 1) View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.
- 2) Define the state as a summary of all relevant decisions
- 3) Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.
- 4) Write a recursion on the optimal cost from the state to a state

square indicates that time is meant for the algorithm (more about this in the lab).

[Introduction to linear optimization, Bertsimas, Tsitsiklis, 1997]

### Traveling salesman

What is the shortest path to loop through N cities?



[<http://www.informatik.uni-leipzig.de/~meiler/>]

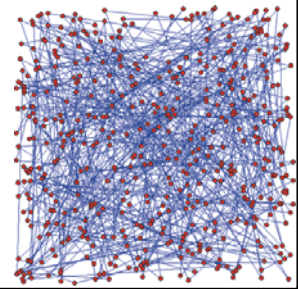


[<http://www.superbasescientific.com/>]

### Traveling salesman: engineering applications

What is the shortest path to loop through N cities?

500 cities, random solution!

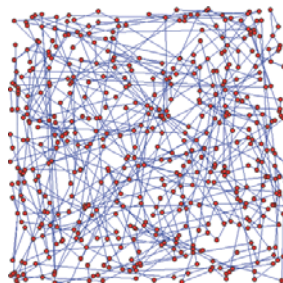


[<http://www.logicalgenetics.com/>]

### Traveling salesman: engineering applications

What is the shortest path to loop through N cities?

500 cities, a better solution!

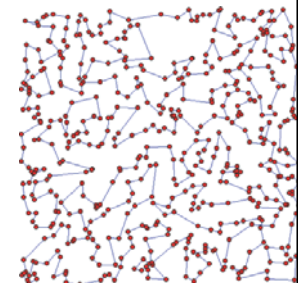


[<http://www.logicalgenetics.com/>]

### Traveling salesman: engineering applications

What is the shortest path to loop through N cities?

500 cities, a much better solution!



[<http://www.logicalgenetics.com/>]

Traveling salesman: dynamic programming solution

$$C(S, k) = \min_{m \in S \setminus \{k\}} (C(S \setminus \{k\}, m) + c_{mk})$$

$S$  subset of cities including city 1 (departure city)

City 1

Traveling salesman: dynamic programming solution

$$C(S, k) = \min_{m \in S \setminus \{k\}} (C(S \setminus \{k\}, m) + c_{mk})$$

$S$  subset of cities including city 1 (departure city)  
 $k$  a city in  $S$

City 1  
a city in  $S$

Induction is done on  $S$

Traveling salesman: dynamic programming solution

$$C(S, k) = \min_{m \in S \setminus \{k\}} (C(S \setminus \{k\}, m) + c_{mk})$$

$S$  subset of cities including city 1 (departure city)  
 $k$  a city in  $S$

City 1  
a city in  $S$

$C(S, k)$  shortest path from city 1 to city  $k$  that visits all nodes in  $S$

Induction is done on  $S$

Traveling salesman: dynamic programming solution

$$C(S, k) = \min_{m \in S \setminus \{k\}} \underbrace{(C(S \setminus \{k\}, m))}_{\text{Shortest way to go to } m \text{ without going through } k} + \underbrace{c_{mk}}_{\text{Cost to go from } m \text{ to } k}$$

$C(S, k)$  shortest path from city 1 to city  $k$  that visits all nodes in  $S$  |  $c_{mk}$  cost to go from city  $m$  to city  $k$

$S$  subset of cities including city 1 (departure city) | Where  $m$  is a node in  $S$ , but not  $k$

$k$  a city in  $S$

Induction is done on  $S$

Traveling salesman: dynamic programming solution

$$C(S, k) = \min_{m \in S \setminus \{k\}} \underbrace{(C(S \setminus \{k\}, m) + c_{mk})}_{\text{Best over all possible } m \text{ (not equal to } k)}$$

$C(S, k)$  shortest path from city 1 to city  $k$  that visits all nodes in  $S$  |  $c_{mk}$  cost to go from city  $m$  to city  $k$

$S$  subset of cities including city 1 (departure city) | Where  $m$  is a node in  $S$ , but not  $k$

$k$  a city in  $S$

Induction is done on  $S$

Traveling salesman: dynamic programming solution

$$C(S, k) = \min_{m \in S \setminus \{k\}} (C(S \setminus \{k\}, m) + c_{mk})$$

$C(\{1\}, 1) = 0$  Cost to go from city 1 to city 1: zero

$C(S, k)$  shortest path from city 1 to city  $k$  that visits all nodes in  $S$  |  $c_{mk}$  cost to go from city  $m$  to city  $k$

$S$  subset of cities including city 1 (departure city) | Where  $m$  is a node in  $S$ , but not  $k$

$k$  a city in  $S$

Induction is done on  $S$



## How to construct dynamic programming algorithms

- 1) View the choice of a feasible solution as a sequence of decisions occurring in stages, and so that the total cost is the sum of the costs of individual decisions.
- 2) Define the state as a summary of all relevant    decisions
- 3) Determine which state transitions are possible. Let the cost of each state transition be the cost of the corresponding decision.
- 4) Write a recursion on the optimal cost from the    state to a    state

   square indicates that time is meant for the algorithm (more about this in the lab).

[\[Introduction to linear optimization, Bertsimas, Tsitsiklis, 1997\]](#)