Algorithm Design IX

Dynamic Programming I

Guoqiang Li
School of Software

SHANGHAI JIAO TONG UNIVERSITY

Shortest Paths in DAGs, Revisited

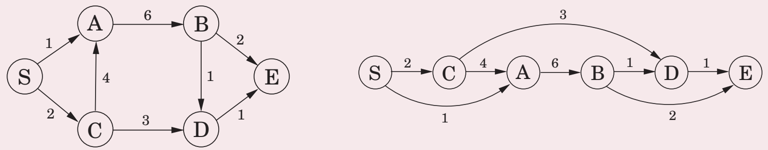
Shortest Paths in Dags, Revisited

The special distinguishing feature of a DAG is that its nodes can be linearized.

Shortest Paths in Dags, Revisited

The special distinguishing feature of a DAG is that its nodes can be linearized.

If compute `dist` values in the left-to-right order, by the time get to a node v, we already have all the information to compute $\texttt{dist}(v)$.



Shortest Paths in Dags, Revisited

Initialize all $dist(.)$ value to ∞;
$dist(s)$=0;
for *each* $v \in V \backslash \{s\}$, *in linearized order* **do**
  $dist(v) = min_{(u,v) \in E}\{dist(u) + l(u,v)\}$;
end

Shortest Paths in Dags, Revisited

Initialize all $dist(.)$ value to ∞;
$dist(s)$=0;
for *each $v \in V \backslash \{s\}$, in linearized order* **do**
    $dist(v) = min_{(u,v) \in E}\{dist(u) + l(u,v)\}$;
end

This algorithm is solving a collection of subproblems, $\{dist(u) \mid u \in V\}$

Dynamic Programming

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Dynamic Programming

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.

Dynamic Programming

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.Then proceed with progressively "larger" subproblems, where a subproblem is large if a lot of other subproblems is solved before get to it.

Dynamic Programming

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.Then proceed with progressively “larger” subproblems, where a subproblem is large if a lot of other subproblems is solved before get to it.

Dynamic programming is a powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling smallest first, using the answers to small problems to figure out larger ones, until the whole lot of them is solved.

Dynamic Programming

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.Then proceed with progressively “larger” subproblems, where a subproblem is large if a lot of other subproblems is solved before get to it.

Dynamic programming is a powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling smallest first, using the answers to small problems to figure out larger ones, until the whole lot of them is solved.

In dynamic programming we are not given a DAG; the DAG is implicit.

DP本身应该是基于DAG的，即可以将所谓"子问题"理解成当前这个子问题的前序点的集合，与"最短路径"问题类似的，每一个点的"最优解"应该是是在这个点之前所有的子问题已解出来的情况下得到的，因而两者等价。

Longest Increasing Subsequences

The Problem

SHANGHAI JIAO TONG UNIVERSITY

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

The Problem

SHANGHAI JIAO TONG UNIVERSITY

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \le i_1 < i_2 < \ldots < i_k \le n$.

The Problem

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

An increasing subsequence is one in which the numbers are getting strictly larger.

The Problem

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$ 子序列：可以不连续，但是需要维持原有的顺序

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

An increasing subsequence is one in which the numbers are getting strictly larger.

The task is to find the increasing subsequence of greatest length.

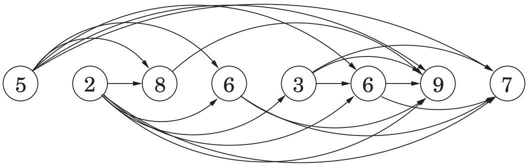
$$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

从DAG角度看这个问题，就是等价于原始序列中每一个点都是一个DAG中的vertex，而这个点和其之后的满足递增关系的点之间都有一条有向边，因而问题就转化为了求解一个DAG中的最长路径。

Graph Reformulation

Create a graph of all permissible transitions:

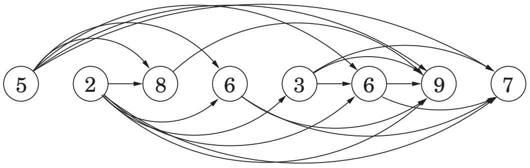
- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.



Graph Reformulation

Create a graph of all permissible transitions:

- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.

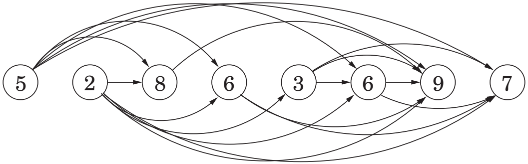


This graph $G = (V, E)$ is a DAG, since all edges (i, j) have $i < j$.

Graph Reformulation

Create a graph of all permissible transitions:

- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.



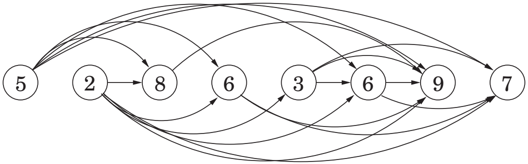
This graph $G = (V, E)$ is a DAG, since all edges (i, j) have $i < j$.

There is a one-to-one correspondence between increasing subsequences and paths in this DAG.

Graph Reformulation

Create a graph of all permissible transitions:

- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.



This graph $G = (V, E)$ is a DAG, since all edges (i, j) have $i < j$.

There is a one-to-one correspondence between increasing subsequences and paths in this DAG.

Therefore, the goal is to find the longest path in the DAG!

The Algorithm


```
for j = 1 to n do
   L(j) = 0(初始化)
```

for $j = 1$ *to* n **do**

$L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$;

end

`return(`$\max_j L(j)$`);`

The Algorithm

for $j = 1$ *to* n **do**

$\quad L(j) = 1 + \max\{L(i) \mid (i,j) \in E\}$;

end

`return` $(\max_j L(j))$;

$L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1).

The Algorithm

for $j = 1$ *to* n **do**

  $L(j) = 1 + \max\{L(i) \mid (i,j) \in E\}$;

end

`return` $(\max_j L(j))$;

$L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1).

If there are no edges into j , we take zero.

The Algorithm

需要注意的是DAG中的"最短路径"指的是基于边的最短，即希望所有边的权重和最小；
而这个DP问题中的"最短路径"指的是基于点的路径，即假定边的权重均为1，而去寻找
通过这些边连接起来的点的个数最多的序列，因而需要对结果**+1**处理。

for $j = 1$ *to* n **do**

  $L(j) = 1 + \max\{L(i) \mid (i, j) \in E\};$

end

return ($\max_j L(j)$);

$L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1).

If there are no edges into j , we take zero.

The final answer is the largest $L(j)$, since any ending position is allowed.

The Analysis

Q: How long does this step take?

The Analysis

SHANGHAI JIAO TONG UNIVERSITY

Q: How long does this step take?

It requires the predecessors of j to be known;

The Analysis

Q: How long does this step take?

It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time is handy.

The Analysis

Q: How long does this step take?

It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time is handy.

The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$.

The Analysis

Q: How long does this step take?

It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time is handy.

The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$.

This is at most $O(n^2)$, the maximum being when the input array is sorted in increasing order.

This Is Dynamic Programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

This Is Dynamic Programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

This Is Dynamic Programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

In our case, each subproblem is solved using the relation

$$L(j) = 1 + \max\{L(i)|(i, j) \in E\}$$

需要注意的是，DP在寻找"子问题"的时候需要保证子问题是父问题的一个子集。

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

Why Not Recursion

SHANGHAI JIAO TONG UNIVERSITY

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

- e.g. (i, j) for all $i < j$. The formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

- e.g. (i, j) for all $i < j$. The formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Why did recursion work so well with divide-and-conquer? In divide-and-conquer, a problem is expressed in terms of subproblems that are substantially smaller, say, half the size.

problem：为何分治法使用递归的性能就很好，但是DP就很差？
1.DP的子问题与父问题差距比较小，以一维表为例，一般只是N与N-1的区别；而分治法一般是每次划分都是划分为父问题的1/n，(一般为1/2)，所以子问题划分的更小，递归的时候树的深度更小，因而更好
2.分治法在递归的时候一般是直接划分为原来的一半，所以划分之后的同一层次的运算发生overlap的情况几乎没有，但是DP只是N与N-1的区别，所以很大概率会发生重复运算的问题，所以性能更差

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

- e.g. (i, j) for all $i < j$. The formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Why did recursion work so well with divide-and-conquer? In divide-and-conquer, a problem is expressed in terms of subproblems that are substantially smaller, say, half the size.

In a dynamic programming, a problem is reduced to subproblems that are slightly smaller. Thus the full recursion tree has polynomial depth and an exponential number of nodes.

Edit Distance

（编辑距离）

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

Q: What is the appropriate notion of closeness in this case?

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

Q: What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be aligned, or matched up.

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

Q: What is the appropriate notion of closeness in this case? 需要修改的次数

A natural measure of the distance between two strings is the extent to which they can be aligned, or matched up.

Technically, an alignment is simply a way of writing the strings one above the other.

S	—	N	O	W	Y
S	U	N	N	—	Y

Cost: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

Cost: 5

The problem

S	—	N	O	W	Y
S	U	N	N	—	Y

Cost: 3

—	S	N	O	W	—	Y
S	U	N	—	—	N	Y

Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The problem

S	—	N	O	W	Y		—	S	N	O	W	—	Y
S	U	N	N	—	Y		S	U	N	—	—	N	Y

Cost: 3   Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The edit distance between two strings is the cost of their best possible alignment.

The problem

S	—	N	O	W	Y		—	S	N	O	W	—	Y
S	U	N	N	—	Y		S	U	N	—	—	N	Y

Cost: 3   Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The edit distance between two strings is the cost of their best possible alignment.

Edit distance is so named because it can also be thought of as the minimum number of edits.

The problem

S	—	N	O	W	Y		—	S	N	O	W	—	Y
S	U	N	N	—	Y		S	U	N	—	—	N	Y

Cost: 3   Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The edit distance between two strings is the cost of their best possible alignment.

Edit distance is so named because it can also be thought of as the minimum number of edits.

The number of insertions, deletions, and substitutions of characters needed to transform the first string into the second.

A Dynamic Programming Solution

SHANGHAI JIAO TONG UNIVERSITY

What are the subproblems?

A Dynamic Programming Solution

SHANGHAI JIAO TONG UNIVERSITY

What are the subproblems?

The goal is to find the edit distance between two strings, $x[1,\ldots,m]$ and $y[1,\ldots,n]$.

A Dynamic Programming Solution

What are the subproblems?

The goal is to find the edit distance between two strings, $x[1, \ldots, m]$ and $y[1, \ldots, n]$.

For every i, j with $1 \leq i \leq m$ and $1 \leq j \leq n$, let $E(i, j)$: the edit distance between some prefix of the first string, $x[1, \ldots, i]$, and some prefix of the second, $y[1, \ldots, j]$.

A Dynamic Programming Solution

What are the subproblems?

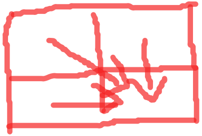
The goal is to find the edit distance between two strings, $x[1, \ldots, m]$ and $y[1, \ldots, n]$.

For every i, j with $1 \leq i \leq m$ and $1 \leq j \leq n$, let $E(i, j)$: the edit distance between some prefix of the first string, $x[1, \ldots, i]$, and some prefix of the second, $y[1, \ldots, j]$.

$$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$$

where $\texttt{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

E(i-1, j-1)



E(i-1, j),+1是因为j方向长度没变，i方向长度多一个，所以只能多添加一次

E(i, j - 1)

An Example

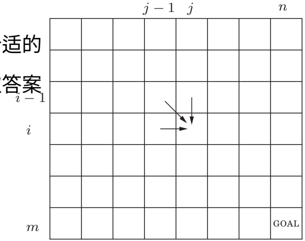
Edit distance between EXPONENTIAL and POLYNOMIAL, subproblem $E(4,3)$ corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:

$$\begin{array}{c} \texttt{O} \\ - \end{array} \quad \texttt{or} \quad \begin{array}{c} - \\ \texttt{L} \end{array} \quad \texttt{or} \quad \begin{array}{c} \texttt{O} \\ \texttt{L} \end{array}$$

Thus, $E(4,3) = \min\{1 + E(3,3), 1 + E(4,2); 1 + E(3,2)\}$.

DP问题的几个步骤：
1.创建1/2维表
2.确定表的入口，选择合适的顺序填表
3.确定表的出口，即获取答案

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

The Algorithm

for $i = 0$ *to* m **do**
  $E(i, 0) = i$;
end
for $j = 1$ *to* n **do**
  $E(0, j) = j$;
end
for $i = 1$ *to* m **do**
  for $j = 1$ *to* m **do**
    $E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$;
  end
end
`return(`$E(m, n)$`);`

The Algorithm

for $i = 0$ *to* m **do**
  $E(i, 0) = i$;
end
for $j = 1$ *to* n **do**
  $E(0, j) = j$;
end
for $i = 1$ *to* m **do**
  for $j = 1$ *to* m **do**
    $E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$;
  end
end
`return(`$E(m, n)$`);`

The over running time is $O(m \cdot n)$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$. The number of subproblems is $O(mn)$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \dots x_n$ and a subproblem is $x_1, x_2, \dots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \dots x_n$, and $y_1, \dots, y_m$. A subproblem is $x_1, \dots, x_i$ and $y_1, \dots y_j$. The number of subproblems is $O(mn)$.
- The input is $x_1, \dots, x_n$ and a subproblem is $x_i, x_{i+1}, \dots, x_j$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \dots x_n$ and a subproblem is $x_1, x_2, \dots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \dots x_n$, and $y_1, \dots, y_m$. A subproblem is $x_1, \dots, x_i$ and $y_1, \dots y_j$. The number of subproblems is $O(mn)$.
- The input is $x_1, \dots, x_n$ and a subproblem is $x_i, x_{i+1}, \dots, x_j$. The number of subproblems is $O(n^2)$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$. The number of subproblems is $O(mn)$.
- The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$. The number of subproblems is $O(n^2)$.
- The input is a rooted tree. A subproblem is a rooted subtree.

Knapsack

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or “knapsack”) will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or “knapsack”) will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What’s the most valuable combination of items he can put into his bag?

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \dots, w_n$ and dollar value $v_1, \dots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

- there are unlimited quantities of each item available;

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

- there are unlimited quantities of each item available;
- there is one of each item.

Knapsack with Repetition

SHANGHAI JIAO TONG UNIVERSITY

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

Knapsack with Repetition

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

We express this in terms of smaller subproblems:

$$K(w) = \max_{i : w_i \leq w} \{K(w - w_i) + v_i\}$$

Knapsack with Repetition

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

We express this in terms of smaller subproblems:

$$K(w) = \max_{i : w_i \leq w} \{K(w - w_i) + v_i\}$$

$K(0) = 0$;
for $w = 1$ *to* W **do**
  $K(w) = \max_{i : w_i \leq w} \{K(w - w_i) + v_i\}$;
end
`return(`$K(W)$`);`

Knapsack with Repetition

SHANGHAI JIAO TONG UNIVERSITY

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

We express this in terms of smaller subproblems:

这里的限制"w_i<=w"是考虑w的合法范围应该是>=0.

w-w_i实际上就是回到还没有装入这个item i的位置去计算装入之后的重量(前提当然是w足够的话)

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

$K(0) = 0;$
for $w = 1$ *to* W **do**
  $K(w) = \max_{i:w_i \leq w}\{K(w - w_i) + v_i\};$
end
return($K(W)$);

此问题由于不需要关注每个物品的数量，所以直接使用一维表。而一维表的入口一般都是最左面的那个item，即从左向右填表。

The over running time is $O(n \cdot W)$.

W维的一维表，每一个格子需要计算n个数值(商品数量)

Example: Knapsack with Repetition

Take $W = 10$, and

Item	Weight	Value
1	6	$\$30$
2	3	$\$14$
3	4	$\$16$
4	2	$\$9$

and there are unlimited quantities of each item available.

W	0	1	2	3	4	5	...
V	0	0	9	14	18	23	...

Knapsack without Repetition

SHANGHAI JIAO TONG UNIVERSITY

For every $w \leq W$ and $0 \leq j \leq n$, let

$$K(w, j) = \text{ maximum value achievable with a knapsack of capacity } w \text{ and items } 1, \ldots, j$$

Knapsack without Repetition

Shanghai Jiao Tong University

For every $w \leq W$ and $0 \leq j \leq n$, let

$$K(w, j) = \text{ maximum value achievable with a knapsack of capacity } w \text{ and items } 1, \ldots, j$$

We express this in terms of smaller subproblems:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Knapsack without Repetition

Initialize all $K(0, j) = 0$ and all $K(w, 0) = 0$;
for $j = 1$ *to* n **do**
    for $w = 1$ *to* n **do**
        if $w_j > w$ **then** $K(w, j) = K(w, j - 1)$;
        $K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$;
    end
end
`return` $(K(W, n))$;

Knapsack without Repetition

Initialize all $K(0, j) = 0$ and all $K(w, 0) = 0$;
for $j = 1$ *to* n **do**
    for $w = 1$ *to* n **do**
        if $w_j > w$ **then** $K(w, j) = K(w, j - 1)$;
        $K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$;
    end
end
`return(`$K(W, n)$`);`

The over running time is $O(n \cdot W)$.

Knapsack without Repetition

Shanghai Jiao Tong University

由于需要保证子问题是父问题的子集，所以只能是[1],[1,2],[1,2,3],[1,2,3,4]这样的序列，
而这时候，单单从序列上来看会发现缺少了[2,3]这样的集合，但是其实比如对于[1,2,3]，
在用下方的迭代公式进行计算K(w,j)的时候，如果添加了w_1之后发现更小，则实际上就会舍弃
item1，这时候在对于[1,2,3]这种情况进行讨论时就有可能选取到[2，3]

Take $W = 9$, and

Item	Weight	Value
1	2	$\$3$
2	3	$\$4$
3	4	$\$5$
4	5	$\$7$

and there is only one of each item available.

	0	1	2	3	4	
0	0	0	0	0	0	...
1	0					...
1,2	0					...

需要注意子问题需要是父问题的一个子集，所以
每一行表示的是1-j个物品被装入的时候的情况
所以K(w,j)可以使用K(--，j - 1)来表示

Chain Matrix Multiplication

The Problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

The Problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $m \cdot n \cdot p$ multiplications.

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

The Problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $m \cdot n \cdot p$ multiplications.

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

Q: How do we determine the optimal order, if we want to compute $A_1 \times A_2 \times \ldots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, respectively?

Binary Tree

A particular parenthesization can be represented by a binary tree in which

- the individual matrices correspond to the leaves,
- the root is the final product, and
- interior nodes are intermediate products.

Binary Tree

A particular parenthesization can be represented by a binary tree in which

- the individual matrices correspond to the leaves,
- the root is the final product, and
- interior nodes are intermediate products.

The possible orders in which to do the multiplication correspond to the various full binary trees with n leaves.

Subprompblems

For $1 \leq i \leq j \leq n$, let

$$C(i,j) = \text{ minimum cost of multiplying } A_i \times A_{i+1} \times \ldots \times A_j$$

Subproblems

For $1 \leq i \leq j \leq n$, let

$$C(i,j) = \text{ minimum cost of multiplying } A_i \times A_{i+1} \times \ldots \times A_j$$

$$C(i,j) = \min_{i \leq k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

The Program

The Program

for $i = 1$ *to* n **do**
  $C(i,i) = 0;$
end
for $s = 1$ *to* $n - 1$ **do**
  for $i = 1$ *to* $n - s$ **do**
    $j = i + s;$
    $C(i,j) = \min_{i \le k < j}\{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\};$
  end
end
`return` $(C(1,n))$`;`

The Program

for $i = 1$ *to* n **do**
  $C(i,i) = 0;$
end
for $s = 1$ *to* $n - 1$ **do**
  for $i = 1$ *to* $n - s$ **do**
    $j = i + s;$
    $C(i,j) = \min_{i \le k < j}\{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\};$
  end
end
`return` $(C(1,n))$`;`

The over running time is $O(n^3)$.

The Example

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.