AI2615 算法设计与分析 2020-2021 春季学期

Lecture 8 – Dynamic Programming

2021 年 4 月 16 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

Starting from this lecture, we will learn a new class of algorithms, *dynamic programming*, that is applicable to a very board class of problems. In general, a problem for which dynamic programming is applicable can be divided into multiple sub-problems, where those sub-problems are vertices of a directed acyclic graph such that those sources are sub-problems that are simple enough to be solved directly and each of those non-source vertices is a sub-problem whose solution depends on the solutions of the sub-problems represented by the in-neighbors of this vertex. After breaking down the problem into sub-problems with a directed acyclic graph structure, dynamic programming can be implemented in two different ways: *topological sorting* and *memoization*.

In the first approach, we find a total order of the vertices in a directed acyclic graph (e.g., using the method at the end of Lecture 5). Then, following the order, we solve all the sub-problems, where each sub-problem is solved by relating it to the previously solved sub-problems it depends on.

In the second approach, the problem is solved by directly throwing recursive calls to the sub-problems it depends on. Whenever a solution to an intermediate sub-problem is obtained, it is stored, so that the solution can be directly used if this sub-problem appears again in another branch of the recursion tree.

1 Finding Shortest Paths on Directed Acyclic Graphs

We will revisit the problem of finding the shortest path in a directed acyclic graph in Lecture 5. This problem is a typical problem that can be solved by dynamic programming. It can also be viewed as a canonical form of dynamic programming: as we mentioned earlier, dynamic programming solves those problem that can be broken down to sub-problems forming a directed acyclic graph structure.

Problem 1. Given a weighted directed acyclic graph $G = (V, E, w)$ (where edges may have negative weights) and two vertices s and t, find the distance from s to t.

By the method we have learned in Lecture 5, we can sort the vertices in G by a total order that agrees with the partial order defined by G. This enables us to label the vertices using $1, 2, \ldots, n$ such that $(i, j) \in E$ implies $i < j$. An example is shown in Figure 1. We assume without loss of generality that s is the vertex with label 1. If this is not the case, we can just delete all the vertices having index less than s, as we know that there is no path from s to each of these vertices.

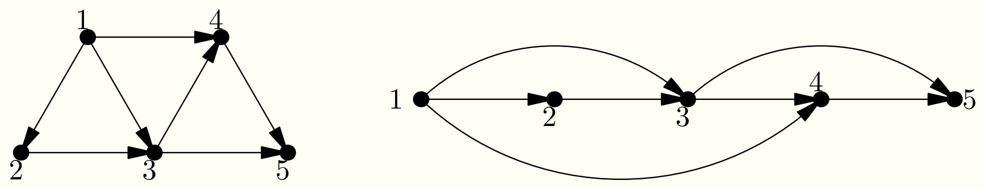

Figure 1: An example of topological sorting.

Coming back to our problem, a simple yet important observation is that, for any $j > i$, the shortest path from s to i cannot contain j, for otherwise there is a path from j to i and $j > i$ should not have been true. Let $\mathsf{dist}(i)$ be the distance from s to i. The observations implies that the sub-problem $\mathsf{dist}(i)$ can only depend on $\mathsf{dist}(1),\ldots,\mathsf{dist}(i-1)$. In fact, we only need to look at each of i's in-neighbors, as a path from s to i must visit one of i's in-neighbors before reaching i. We have the following recurrence relation:

$$\mathsf{dist}(i) = \min_{j:j<i}\{\mathsf{dist}(j) + w(j,i)\}, \tag{1}$$

where we set $w(j,i) = \infty$ if $(j,i) \notin E$.
A recurrence relation like (1) is the key for a dynamic programming algorithm. Next, we will see two different implementations for dynamic programming: *topological sorting* and *memoization*.

1.1 Topological Sorting

In this approach, all sub-problems are topologically sorted by a total order that agrees with the partial order defined by the directed acyclic graph. Then, all the sub-problems are solved one-by-one following the order. In our shortest path problem, we solve $\mathsf{dist}(1),\mathsf{dist}(2),\ldots,\mathsf{dist}(n)$ one-by-one. The algorithm is shown in Algorithm 1.

Algorithm 1 A dynamic programming algorithm for the shortest path problem by topological sorting

Input: $G = (V,E,w)$ and $s,t \in V$

Output: distance from s to t

```
1: find a total order on $G$, and label the vertices with $1,\ldots,n$ such that $s = 1$
2: $\mathsf{dist}(1) \leftarrow 0$, and for each $i = 2,\ldots,n$: $\mathsf{dist}(i) \leftarrow \infty$
3: for $i = 1,\ldots,n$:
4:     $\mathsf{dist}(i) = \min_{j:j<i}\{\mathsf{dist}(j) + w(j,i)\}$
5: endfor
6: return $\mathsf{dist}(t)$
```

The time complexity for Algorithm 1 is $O(m+n)$. Notice that, for Step 4, we do not need to check for all vertices that are less than i. Instead, we only need to check for i's in-neighbors. Therefore, each edge is checked exactly once.

1.2 Memoization

The recurrence relation (1) straightforwardly gives us the recursive algorithm in Algorithm 2, and DIST(t) returns the distance from s to t.

Algorithm 2 A straightforward recursive algorithm for the shortest path problem

```
DIST($i$)
1: if $i = 1$: return 0
2: return $\min_{j:j<i}\{\mathsf{dist}(j) + w(j,i)\}$
```

Using the graph in Figure 1 as an example, if we call DIST(5), the recursion tree is shown in Figure 2. From the figure, it is easy to see that Algorithm 2 merely enumerates all possible paths from 1 to 5. In general, this requires exponential time.

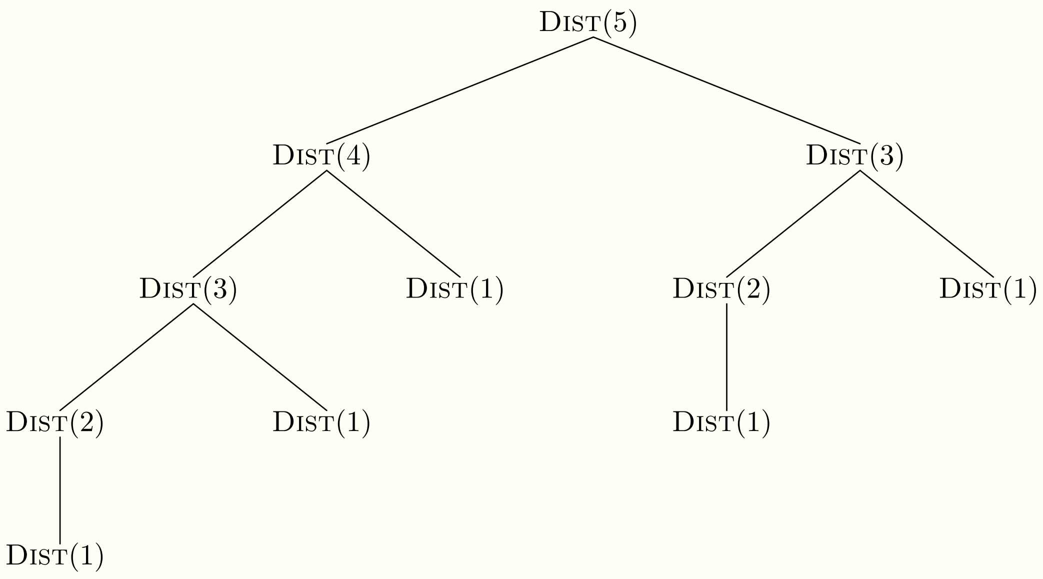

Figure 2: Recursion tree for DIST(5) for the graph in Figure 1.

However, there is a natural way to reduce the time complexity. From Figure 2, we can see that DIST(3) and DIST(2) are computed for more than once. This is clearly unnecessary: once the value of DIST(2) (or DIST(3)) is obtained, we can store it, and directly use it later. This is exactly the other implementation of dynamic programming: the *memoization*. The algorithm is shown in Algorithm 3. To find the distance from s to t, we call DIST(t).

When implementing a dynamic programming algorithm with memoization, it is of crucial importance to store the solutions of sub-problems once obtained. Otherwise, it is often the case that the implementation becomes an exponential time brute-force search.

The algorithm runs in $O(m+n)$ time. Again, each edge is checked once.

Algorithm 3 A dynamic programming algorithm for the shortest path problem by memoization

```
DIST($i$) // initialize array $a[1\cdots n]$ with $a[1]=0$ and $a[i]=\infty$ for $i=2,\dots,n$
1: if $a[i]<\infty$: return $a[i]$
2: $a[i]=\min_{j:j<i}\{\mathsf{dist}(j)+w(j,i)\}$
3: return $a[i]$
```

1.3 Topological Sorting versus Memoization

Which of the two implementations is better? This depends on the problem we are dealing with. Topological sorting solves all the sub-problems "from left to right" in the topological order. Memoization starts from the problem on the "right", and goes to the "left" based on the recurrence relation. In particular, memoization does not necessarily solve all the sub-problems. For example, if the graph in Figure 1 is changed by deleting the edge $(4,5)$, Dist(4) is not called when we call Dist(5). Therefore, memoization has the advantage for skipping some of the unuseful sub-problems. On the other hand, throwing a recursive call to a sub-problem is more time consuming than one iteration of the for-loop. In conclusion, memoization should be used when we know that significantly many of the sub-problems will not be useful to solve the original problem. Otherwise, topological sorting is a better idea.
Theoretically, the asymptotic time complexities for the two implementations are usually the same.

2 Longest Increasing Sequence

Problem 2. Given a sequence of n numbers $a[1\cdots n]$, find the length of the longest (strictly) increasing sub-sequence.

For example, in the sequence $5,2,8,6,3,6,9,7$, there are two longest increasing sub-sequence of length 4: $2,3,6,7$ and $2,3,6,9$.
To apply dynamic programming, we need to define sub-problems. A natural way for this is to define the sub-problem $F(i)$ being the length of the longest increasing sub-sequence from $a[1],\dots,a[i]$, and $F(n)$ is what we want to return. Then, we need to find a recurrence relation of $F(i)$. Now, we are facing the trouble of relating $F(i)$ to $F(i-1),\dots,F(1)$. The recurrence relation is hard to find. In particular, we do not know if the i-th number should be in the longest sub-sequence. If the i-th number is in the sub-sequence, it places a restriction to the solution of $F(i-1)$, as the end of the longest sequence in the first $i-1$ numbers now needs to be less than $a[i]$. Therefore, it is unclear how to break down $F(i)$.
The trick here is to define the sub-problem differently so that a recurrence relation is easy to find. In this problem, we can let $F(i)$ be *the length of the longest increasing sub-sequence in* $a[1],\dots,a[n]$ *ending at* $a[i]$. In this case, we have the following natural recurrence relation:

$$F(i)=1+\max_{j:(j<i)\wedge(a[j]<a[i])}\{F(j)\}. \tag{2}$$

To find the longest increasing sub-sequence in $a[1], \ldots, a[n]$, we just need to return the largest value from $F(1), \ldots, F(n)$.
We leave it as an exercise to write the pseudo-code of the algorithm. The time complexity of the algorithm is clearly $O(n^2)$.
The example in this section shows that sometimes we need to cleverly break down the problem into sub-problems in order to make the dynamic programming efficient.

3 Edit Distance

Definition 3. Given two strings, the *edit distance* between them is the minimum number of operations required to modify one string to the other, where each operation can be one of the following three:

- Deletion: deleting a character at a given position.
- Insertion: insert a character after a given position.
- Replacement: replace a character by a given character at a given position.

For example, the edit distance between the string SNOWY and the string SUNNY is 3: from the string SNOWY, we can either change the middle three characters to UNN by three replacements, or insert an U after the first character S, change the character O to N, and then delete the character W, each of which requires 3 operations.

Problem 4. Given two strings $x[1 \cdots n]$ and $y = [1 \cdots m]$, find the edit distance between them.

A given sequence of operations that changes x to y can be viewed in the following way. We first expand x and y by inserting multiple *place-holder character* "_", so that x and y are of the same length, then we align the two expanded strings by the positions of the characters. If the i-th positions of the two strings are equal, then no operation was taken. If the i-th position of x is not a place-holder character while the i-th position of y is, then a delete operation was taken. If the i-th position of x is a place-holder character while the i-th position of y is not, then a insert operation was taken. If both the i-th position of x and the i-th position of y are not the place-holder character and they are not equal, then a replace operation was taken. Two examples are given in Figure 3.

S_NOWY SUNN_Y	SNOWY_____ _____SUNNY
an insertion of U, an replacement of O to N, and an deletion of W	five deletions and five insertions

Figure 3: Two Examples of operations with place-hold interpretation.

Let $F(i, j)$ be the sub-problem for the edit distance between the substring $x[0 \cdots i]$ and the substring

$y[0 \cdots j]$. We would like to find $F(n,m)$. To find a recurrence relation, we only need to enumerate the three possible cases regarding the last characters in the expansions of $x[0 \cdots i]$ and $y[0 \cdots j]$:

- Case 1: the last characters of the two expanded strings are $x[i]$ and the place-holder character respectively (a deletion was performed);
- Case 2: the last characters of the two expanded strings are the place-holder character and $y[j]$ respectively (an insertion was performed);
- Case 3: the last characters of the two expanded strings are $x[i]$ and $y[j]$ respectively (if $x[i] \neq y[j]$, a replacement was performed; otherwise, nothing was done);

This gives the following recurrence relation:

$$F(i,j) = \min\left\{F(i,j-1)+1, F(i-1,j)+1, F(i-1,j-1)+\mathbb{I}(x[i] \neq y[j])\right\}, \tag{3}$$

where $\mathbb{I}(\cdot)$ is the indicator function:

$$\mathbb{I}(p) = \begin{cases} 1 & \text{if } p \text{ is true} \\ 0 & \text{otherwise} \end{cases}.$$

For the initial condition, we have $F(0,j) = j$ for each $j = 1, \ldots, m$ and $F(i,0) = i$ for each $i = 1, \ldots, n$.
In the dynamic programming algorithm, we have broken down the problem into nm sub-problems. The underlying directed acyclic graph is shown in Figure 4 with $n = 3$ and $m = 5$.

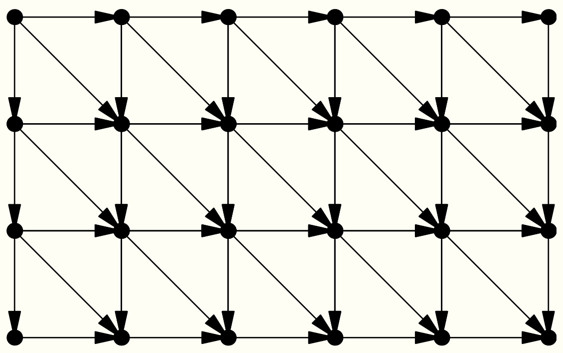

Figure 4: Underlying directed acyclic graph for the dynamic programming algorithm for the edit distance, with $n = 3$ and $m = 5$.

In this problem, if we use memoization, no sub-problem can be skipped. Therefore, topological sorting is a better implementation. To find a total order that agree with the directed acyclic graph, we can either use the row-by-row order or column-by-column order. In terms of coding, we can nest the two for-loops in both ways: either put the for-loop for i inside the for-loop for j, or put the for-loop for j inside the for-loop for i. The time complexity is given by $O(nm)$.

Exercise 5. Suppose each of the three operations, deletion, insertion and replacement, require different costs. Adapt the dynamic programming algorithm to find the edit distance for this setting.

4 Knapsack

Problem 6 (Knapsack). Given a set of n items each of which has a weight $w_i \in \mathbb{Z}^+$ and a value $v_i \in \mathbb{R}^+$, and given a capacity $W \in \mathbb{Z}^+$, find a subset of items with maximum total value such that the total weight is at most W.

We consider two different settings for the Knapsack problem:

- Setting 1: each item has infinitely many copies, so that we can choose the same item for more than once;
- Setting 2: each item can only be chosen once.

For example, we have four items with weights $(w_1, w_2, w_3, w_4) = (6, 3, 4, 2)$ and values $(v_1, v_2, v_3, v_4) = (30, 14, 16, 9)$, and we have the capacity $W = 10$. Under Setting 1, the optimal solution is to take the first item once and the fourth item twice. The total weight is 10 and the total value is 48. Under Setting 2, the optimal solution is to take the first item and the third item. The total weight is 10 and the total value is 46. There is a natural greedy algorithm for this: iteratively select an item with the largest value-to-weight ratio v_i/w_i while possible. However, this algorithm does not always output the optimal solution. In the above-mentioned example, this greedy algorithm will select the first and the second item in both settings, which has total value only 44.

Exercise 7. Consider a different setting where items are divisible. That is, we are allow to choose, for example, a 0.5 fraction of an item. In this setting, each item has only one copy. Adapt the greedy algorithm mentioned above so that it always outputs an optimal solution under this setting.

4.1 Setting 1

For each $w = 0, 1, \ldots, W$, let $F(w)$ be the maximum total value if the capacity is w. We need to find $F(W)$. By enumerating all the possibilities for the previous selected item, we have the following recurrence relation:

$$F(w) = \max_{i:1\leq i\leq n; w_i \leq w} \{v_i + F(w - w_i)\}, \tag{4}$$

with initial condition $F(0) = 0$. Figure 5 presents the incoming edges for vertex $F(10)$ in the underlying directed acyclic graph for the dynamic programming, with the example $(w_1, w_2, w_3, w_4) = (6, 3, 4, 2)$, values $(v_1, v_2, v_3, v_4) = (30, 14, 16, 9)$, and $W = 10$.

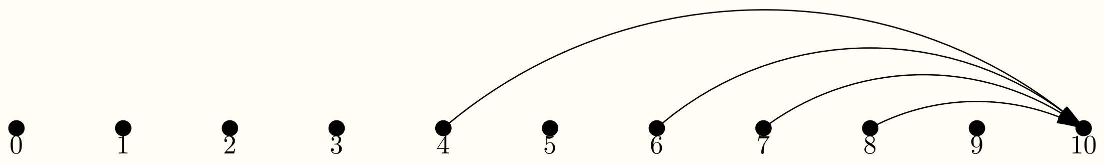

Figure 5: The incoming edges for vertex $F(10)$ in the underlying directed acyclic graph.

A dynamic programming algorithm can be designed based on (4), with time complexity $O(Wn)$. In this problem, memoization is usually a better implementation, especially when the weights of the items are generally large. This is because we can skip many sub-problems. Notice that we are not talking about the worst-case scenario. In the worst-case scenario where there is an item with weight 1, we have to come across every sub-problem.

4.2 Setting 2

For each $w = 0, 1, \dots, W$ and each $k = 0, 1, \dots, n$, let $F(w, k)$ be the maximum total value if the capacity is w and we are only allowed to select from the first k items. We need to find $F(W, n)$. To find a recurrence relation for $F(w, k)$, we consider the k-th item. If the k-th item has weight more than w, we know that the k-th item cannot be added. Otherwise, there are two options: include the k-th item, or not. By taking these possibilities into consideration, we have the following recurrence relation:

$$F(w,k) = \begin{cases} F(w, k-1) & \text{if } w_k > w \\ \max\{F(w, k-1), F(w - w_k, k-1) + v_k\} & \text{if } w_k \le w \end{cases}. \tag{5}$$

We leave it as an exercise to fill in the remaining details for this dynamic programming algorithm.

5 Chain Matrix Multiplication

Suppose A, B, C, D are four matrices with dimensions given in Table 1, and we would like to compute $A \times B \times C \times D$.

matrix	A	B	C	D
dimension	50×20	20×1	1×10	10×100

Table 1: Dimensions of the four matrices

We know that matrix multiplication satisfies associative law. The multiplication $A \times B \times C \times D$ can be computed in different orders. In particular, different orders of computation require different numbers of operations. For example, if the multiplication is done by the order $A \times ((B \times C) \times D)$, the total number of operations is

$$20 \times 1 \times 10 + 20 \times 10 \times 100 + 50 \times 20 \times 100 = 120200.$$

If the multiplication is done by the order $(A \times B) \times (C \times D)$, the total number of operations is

$$50 \times 20 \times 1 + 1 \times 10 \times 100 + 50 \times 100 \times 1 = 7000.$$

We can see that a better order of computation can save a significant number of operations. This motivates the following problem.

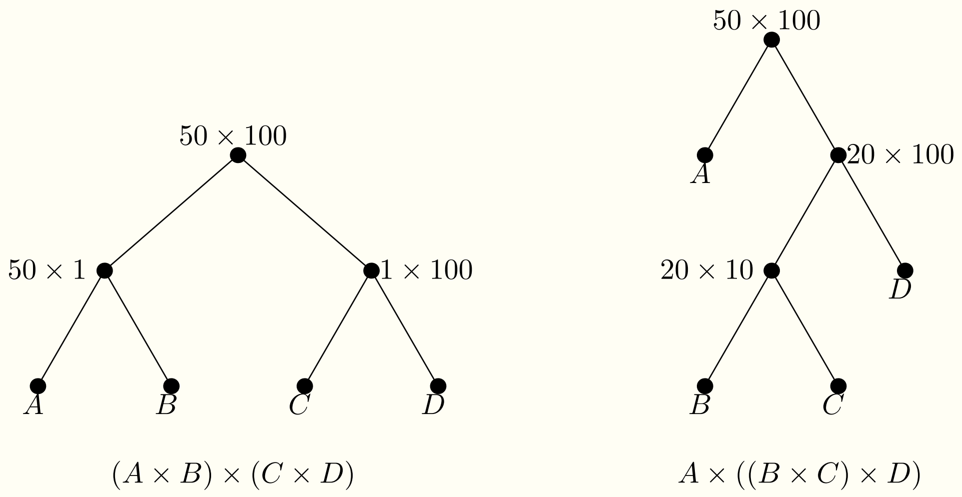

Figure 6: Full binary tree representation for the order of multiplication. Leafs are labeled with matrix names, and internal nodes are labeled with the dimensions of the matrices.

Problem 8. Given n matrices $M_1, \ldots, M_n$ with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$ respectively, find an order of the multiplication $M_1 \times M_2 \times \cdots \times M_n$ with minimum number of operations.

An order of multiplication for $M_1 \times M_2 \times \cdots \times M_n$ can be described by a full binary tree with n leafs, such that the two children for every internal node are multiplied and the parent of the two children represents the product of this multiplication. The full binary tree representations for the two examples $(A \times B) \times (C \times D)$ and $A \times ((B \times C) \times D)$ are shown in Figure 6.

Consider an internal node i with two children j_1 and j_2. The dimension of the matrix represented by i is determined by the dimensions of the two matrices represented by j_1 and j_2 respectively. In particular, the number of the operations to compute the product of the two matrices represented by j_1 and j_2 is independent to how j_1 and j_2 further break down. Suppose we are at an internal node representing $M_i \times M_{i+1} \times \cdots \times M_j$ for certain i, j with $j > i$. Its two children must represent $M_i \times \cdots M_k$ and $M_{k+1} \times \cdots \times M_j$ for certain k with $i \le k < j$. If k is determined, the number of operations for computing the product of the two matrices $(M_i \times \cdots M_k)$ and $(M_{k+1} \times \cdots \times M_j)$ is determined, which is $m_{i-1} \times m_k \times m_j$.

Let $F(i, j)$ be the optimal number of operations to compute $M_i \times M_{i+1} \times \cdots \times M_j$. We have the recurrence relation

$$F(i, j) = \min_{k: i \le k < j} \left\{ F(i, k) + F(k+1, j) + m_{i-1} \times m_k \times m_j \right\}. \tag{6}$$

We need to compute $F(1, n)$. We have the initial condition $F(i, i) = 0$ for any $i = 1, \ldots, n$. Figure 7 presents all the incoming edges for the vertex $F(2, 5)$ in the directed acyclic graph under the dynamic programming for an example with $n = 6$.

Exercise 9. Shall we use topological sorting or memoization here?

Exercise 10. Give a topological order for the underlying directed acyclic graph.

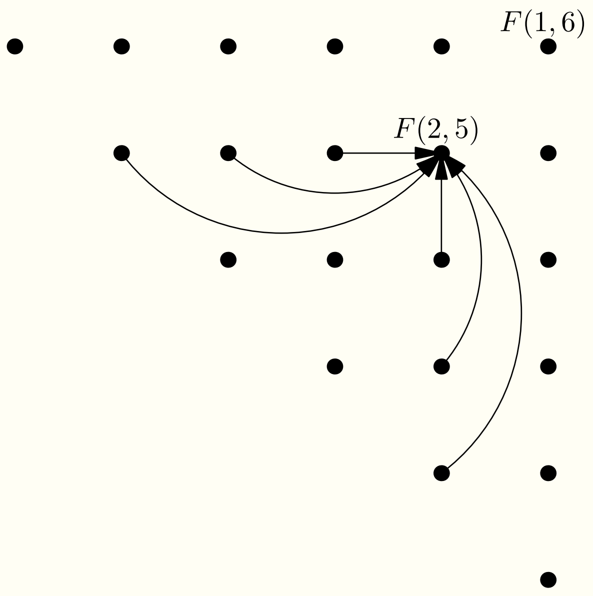

Figure 7: The incoming edges for vertex $F(2,5)$ in the underlying directed acyclic graph.