CSCI 4830 / 5722
Computer Vision

University of Colorado **Boulder**

CSCI 4830 / 5722
Computer Vision

Dr. Ioana Fleming
Spring 2019
Lecture 23

University of Colorado **Boulder**

Reminders

Submissions:

- Homework 4: Wed 3/20 at 11 pm

Readings:

- Szeliski:
  - chapter 11.5.1 (Dynamic Programming)
- P&F:
  - chapter 7.5 (Dynamic Programming)

Dynamic Programming

- A technique for designing (optimizing) algorithms
- It can be applied to problems that can be decomposed in subproblems, but these subproblems overlap.
- Instead of solving the same subproblems repeatedly, applying dynamic programming techniques helps to solve each subproblem just once.

Dynamic Programming

- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

Dynamic Programming Examples

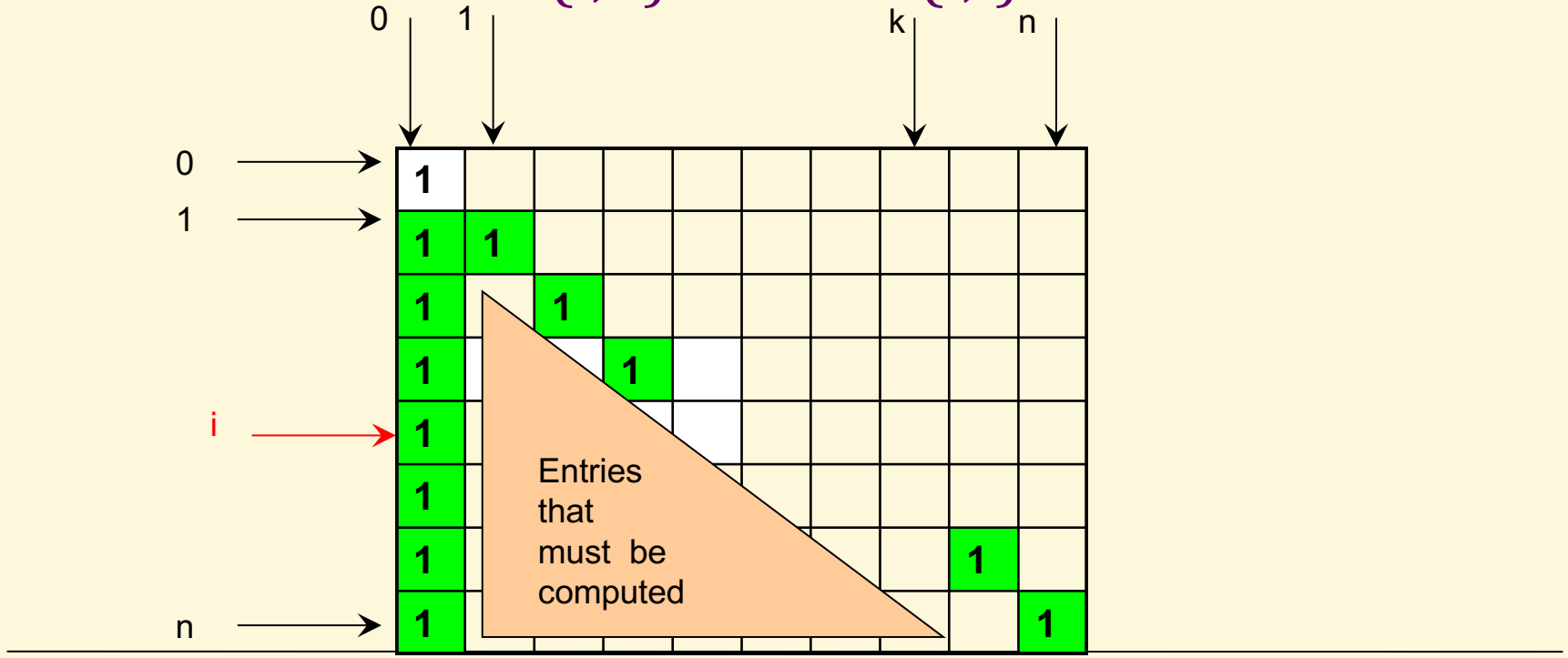
- String Alignment
- Binomial Coefficients
- The Integer Exact Knapsack problem
- Fibonacci numbers
- Longest Common Subsequence
- Commuters – intersection delays

Optimization level 1: Memoization

- We can speed up the ***recursive*** algorithm by writing down the results of the recursive calls and looking them up again if we need them later.
- In this way we do not compute again a recursive call that was already computed before, just take the result from a ***table***
- This process was called ***memoization***
  - ***Memoization*** (not memorization!): the term comes from *memo (memorandum)*, since the technique consists of recording a value so that we can look it up later.

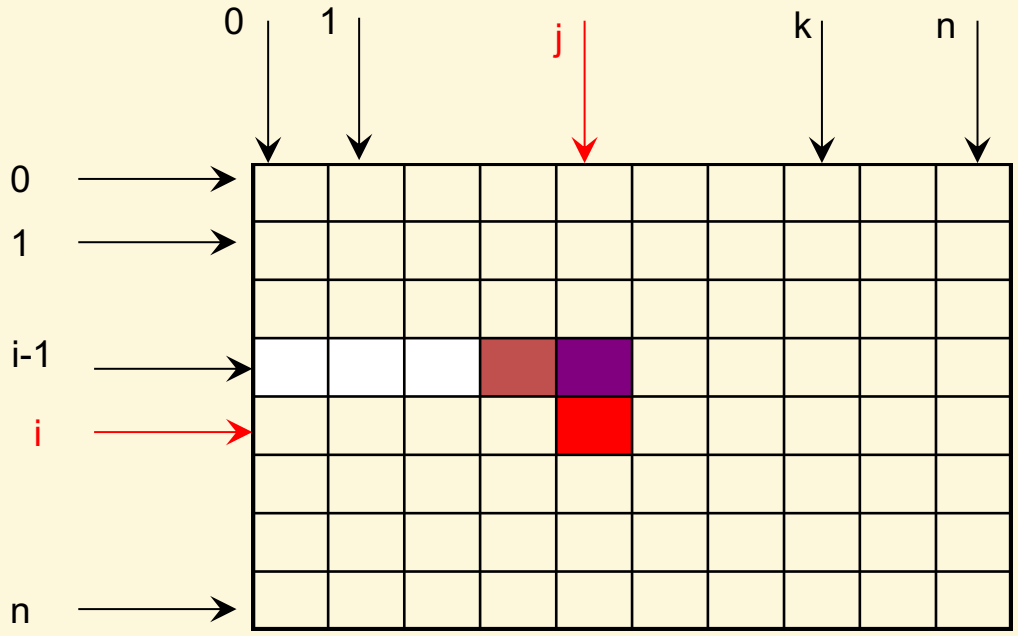
Binomial Coefficients – Table Filling Order

- result[i][j] stores the value of C(i,j)
- Table has n+1 rows and k+1 columns, k<=n
- Initialization: C(i,0)=1 and C(i,i)=1 for i=1 to n



Binomial Coefficients – Order (cont)

- result[i][j] stores the value of C(i,j)
- Rest of entries (i,j), for i=2 to n and j = 1 to i-1 are computed using entry (i-1, j-1) and (i-1, j)

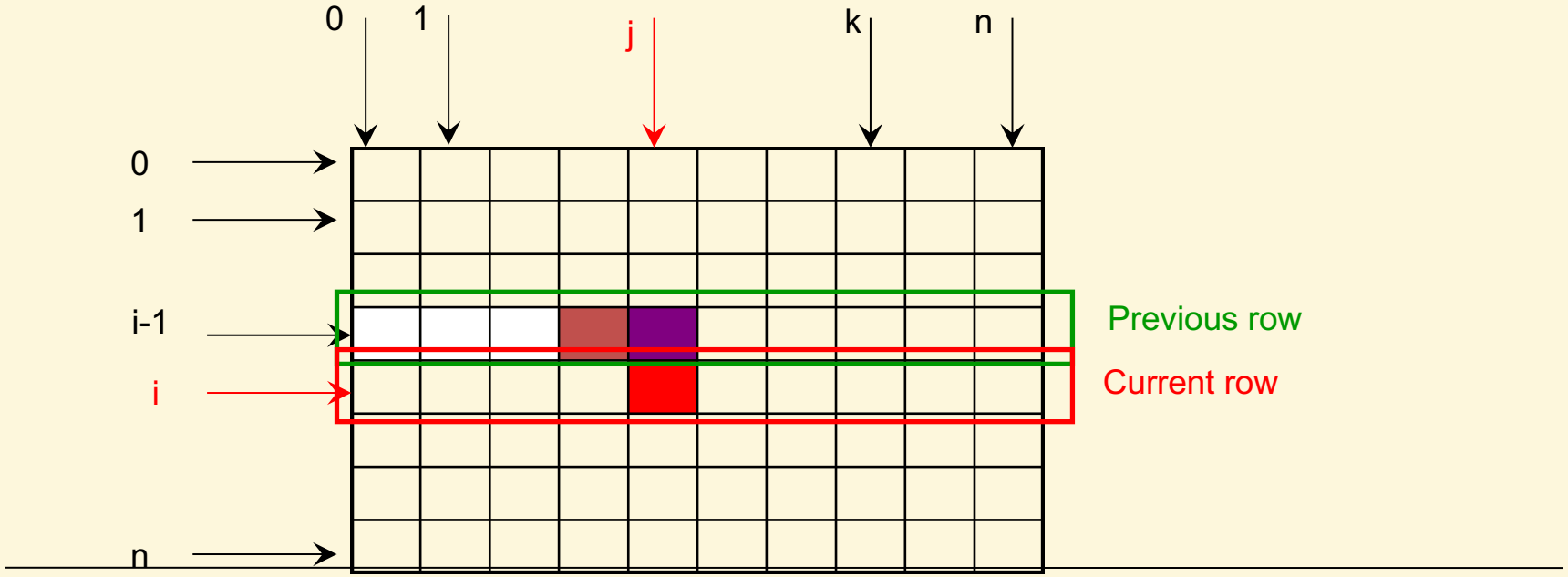


Optimization level 3:
Memory Efficient Dynamic Programming

- In many dynamic programming algorithms, it may be not necessary to retain all intermediate results through the entire computation.
- Every step (every subproblem) depends usually on a reduced set of subproblems, not all other subproblems
- We replace the big table storing the results of all subproblems by some smaller buffers that are reused during the computation

Binomial Coefficients – Reduce Memory Complexity

- At every iteration for i, we compute the values of a row using the values of the row before it
- Two buffers of the length of a row are enough
- The buffers are *reused* after each iteration



Binomial Coefficients –

Memory Efficient Dynamic Programming

```
long C(int n, int k) {

long[] result1 = new long[n + 1];
long[] result2 = new long[n + 1];
result1[0] = 1;
result1[1] = 1;

for (int i = 2; i <= n; i++) {
    result2[0] = 1;
    for (int j = 1; j < i; j++)
        result2[j] = result1[j - 1] + result1[j];
    result2[i] = 1;
   long[] auxi = result1;
   result1 = result2;
   result2 = auxi;
   }
return result1[k];
}
```

Time: O(n*n) (or O(n*k))
Memory: O(n) (or O(k))

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

Example – Fibonacci Numbers (DP)

```
int fibDP(int n)
{
  /* Declare an array to store Fibonacci numbers. */
  int f[n+1];
  int i;

  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }

  return f[n];
}
```

Longest Common Subsequence

- ***Problem:*** Given 2 sequences, $X = \langle x_1,...,x_m \rangle$ and $Y = \langle y_1,...,y_n \rangle$, find the maximum common subsequence.

springtime ncaa tournament basketball

printing north carolina snoeyink

Subsequence need not be consecutive, but must be in order.

Recursive Solution

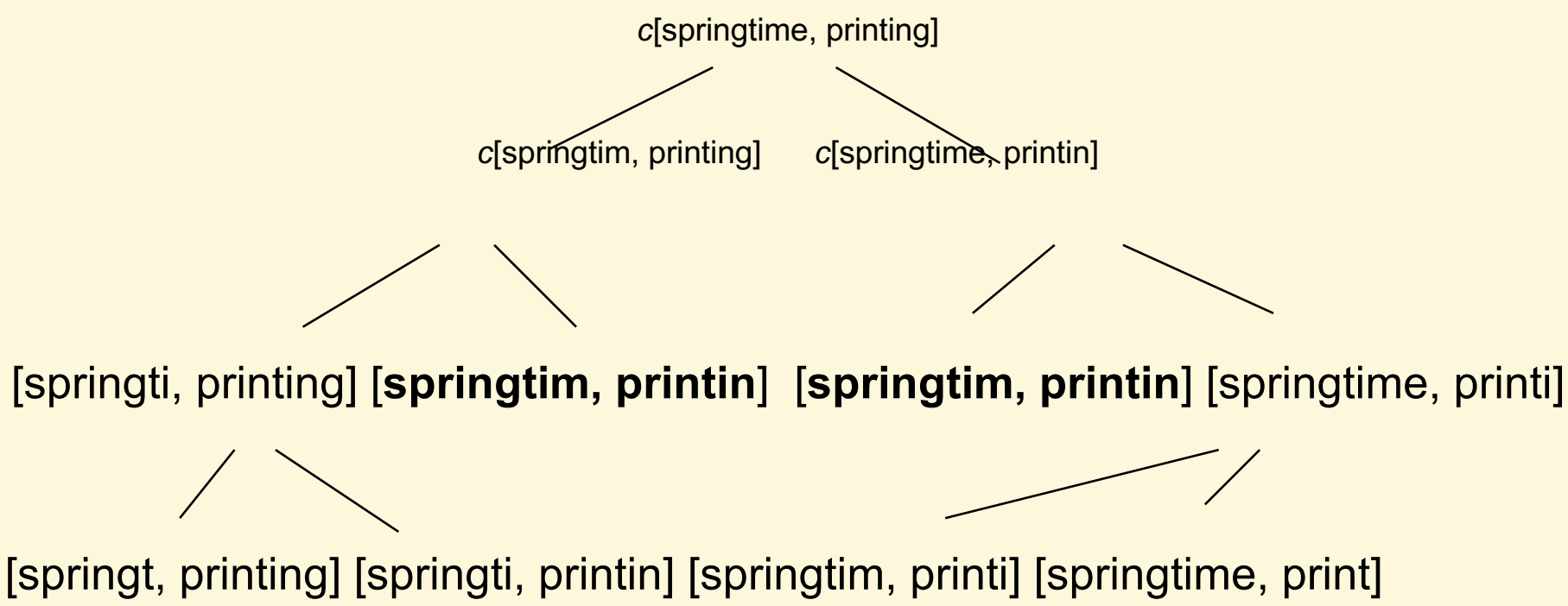
- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

Recursive Solution

$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\alpha, prefix\beta]+1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(c[prefix\alpha,\beta], c[\alpha, prefix\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$



The LCS Problem

- The LCS problem has 2 versions:
  - The *Simple* version, requesting only to find out the length of the longest common subsequence
  - The *Complete* version, requesting to find out the sequence itself

- We discuss first the Simple version

LCS

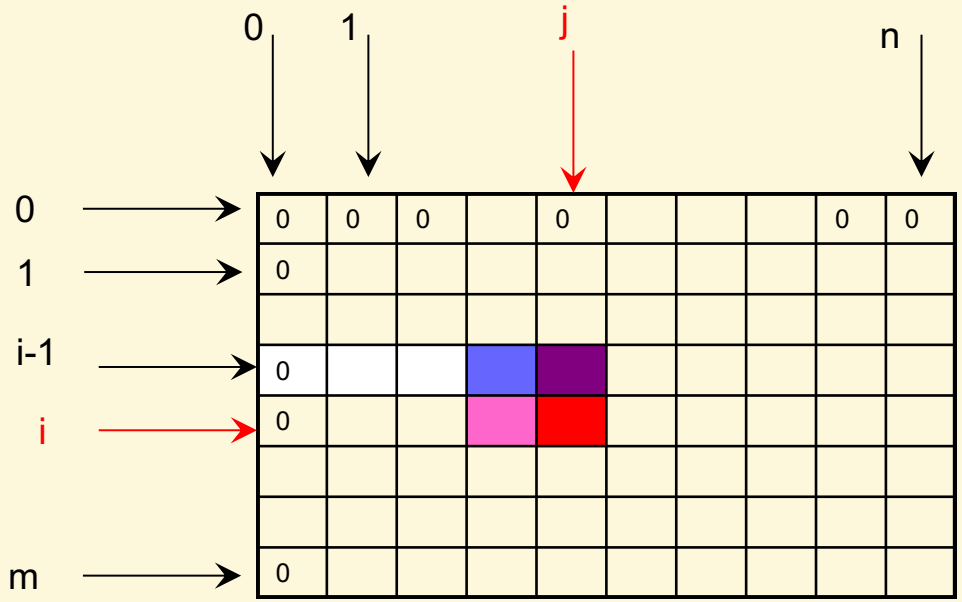
- X = $\{x_1, \ldots x_m\}$
- Y = $\{y_1, \ldots, y_n\}$
- X_i = the prefix subsequence $\{x_1, \ldots x_i\}$
- Y_i = the prefix subsequence $\{y_1, \ldots y_i\}$
- Z =$\{z_1, \ldots z_k\}$ is the LCS of X and Y .
- LCS(i,j) = LCS of X_i and Y_j

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ LCS(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(LCS(i-1, j), LCS(i, j-1)) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

See [CLRS] – chap 15.4

LCS – Dynamic programming

- Entries of row i=0 and column j=0 are initialized to 0
- Entry (i,j) is computed from (i-1, j-1), (i-1, j) and (i, j-1)



A valid order is:

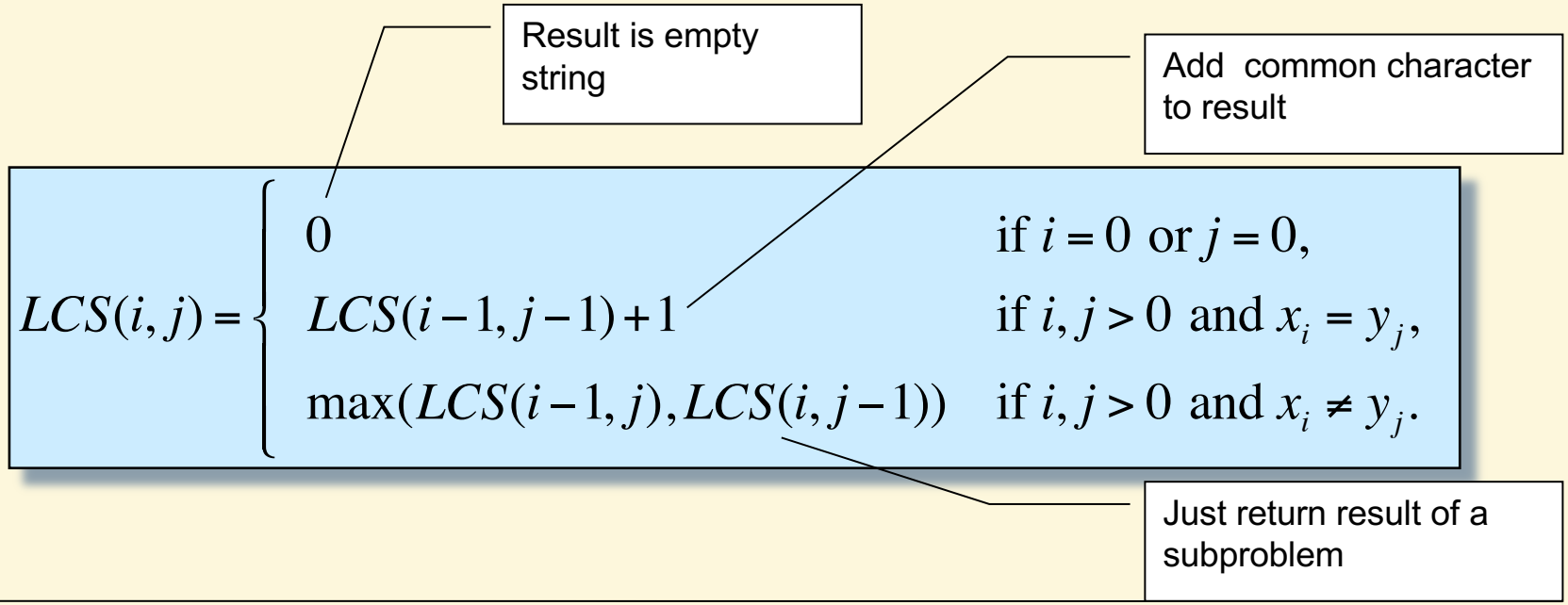
```
For i:=1 to m do
     For j:=1 to n do
         … compute lcs[i,j]
```

Time complexity: O(n*m)

Memory complexity: n*m

LCS – The Complete Version

- The *Complete* version of the problem: we are also interested in finding the characters of the longest common subset



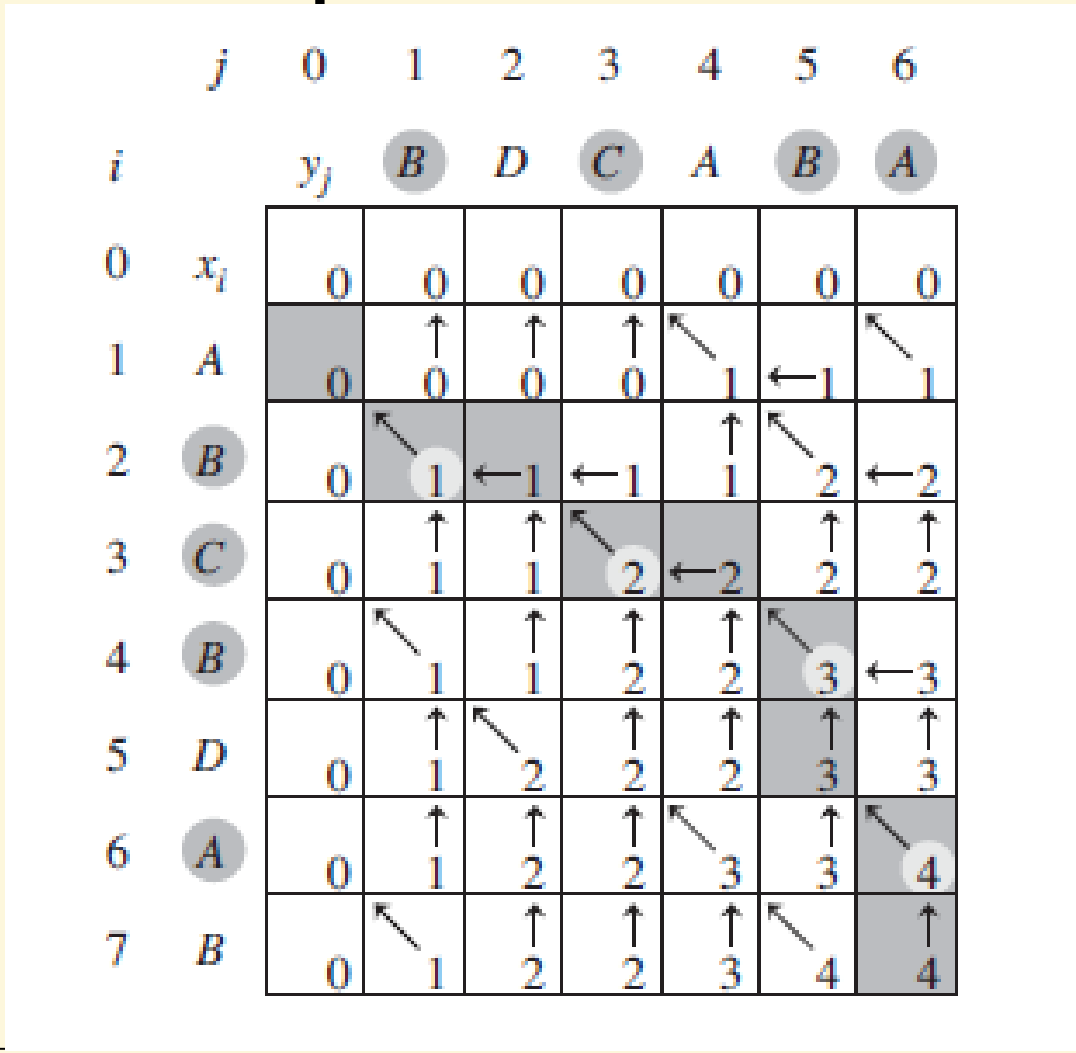
```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

LCS – Restoring the common sequence

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

[CLRS – chap 15.4, page 395]

LCS - Example



i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

[CLRS – Fig. 15.8]

The principle of optimality, I

- Dynamic programming is a technique for finding an *optimal* solution
- The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all subproblems
- Example: Consider the problem of making N¢ with the fewest number of coins
  - Either there is an N¢ coin, or
  - The set of coins making up an optimal solution for N¢ can be divided into two nonempty subsets, n_1¢ and n_2¢
    - ***If either subset, n_1¢ or n_2¢, can be made with fewer coins, then clearly N¢ can be made with fewer coins, hence solution was not optimal***

The principle of optimality, II

- The principle of optimality holds if
  - Every optimal solution to a problem contains...
  - ...optimal solutions to all subproblems
- The principle of optimality does *not* say
  - If you have optimal solutions to all subproblems...
  - ...then you can combine them to get an optimal solution
- Example: In US coinage,
  - The optimal solution to 7¢ is 5¢ + 1¢ + 1¢, *and*
  - The optimal solution to 6¢ is 5¢ + 1¢, *but*
  - The optimal solution to 13¢ is *not* 5¢ + 1¢ + 1¢ + 5¢ + 1¢
- But there is *some* way of dividing up 13¢ into subsets with optimal solutions (say, 11¢ + 2¢) that will give an optimal solution for 13¢
  - Hence, the principle of optimality holds for this problem

The coins problem

Given a value N, if we want to make change for N cents, and we have infinite supply of each of C = { C1, C2, .. , Cm} valued coins, what is the minimum number of coins to make the change?

Examples:

```
Input: coins[] = {25, 10, 5}, N = 30
Output: Minimum 2 coins required
We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, N = 11
Output: Minimum 2 coins required
We can use one coin of 6 cents and 1 coin of 5 cents
```

The coins problem

*Given a value **n**, if we want to make change for **n** cents, and we have infinite supply of each of C = { C1, C2, .. , Cm} valued coins, what is the minimum number of coins to make the change?*

Greedy Approach: how do you do it in everyday life?

The coins problem – recursive solution

*Given a value **n**, if we want to make change for **n** cents, and we have infinite supply of each of C = { C1, C2, .. , Cm} valued coins, what is the minimum number of coins to make the change?*

$coins[\,] = \{1, 5, 10, 25\}$, *s = 4 (size)*

$$Min_{cins}(coins[\,], n) = \begin{cases} 0, & if\ n = 0 \\ \min\{1 + Min_{coins}(n - coins[i])\}, & i = 0, \dots, s-1, \\ . & n > 0\ and\ coins[i] < n \end{cases}$$

The coins problem – recursive solution

```
// s is size of coins array (number of different coins)
int minCoins(int coins[], int s, int n)
{
   if(n == 0)return0; // base case
   int res = INT_MAX; // Initialize result

   // Try every coin that has smaller value than n
   for(int i=0; i<s; i++){
     if(coins[i]<= n){
         int sub_res = minCoins(coins, s, n-coins[i]);

 // Check for INT_MAX to avoid overflow and see if result can minimized
         if(sub_res != INT_MAX && sub_res + 1 < res)
            res = sub_res + 1;
     }
   return res;
}
```

The coins problem – recursive solution

Recursive solution has exponential runtime. **Many problems are solved again and again.**

Example: n = 11, $coins[\,] = \{1, 5, 10, 25\}$, *s = 4 (size)*

min(coins, 4, 11) = min (min(coins, 4, 11-10), min(coins, 4, 11-5), min(coins, 4, 11-1))

min(coins, 4, 6) = min (min(coins,4,6-5), min (coins, 4, 6-1)) = min (min(coins,4,1), **min (coins, 4, 5))**

min(coins, 4, 10) = min (min(coins,4,10-5), min (coins, 4, 10-1)) = min (**min(coins,4,5),** min (coins, 4, 9))

… and so on.

DP solution: constructing a temporary array table in bottom up manner

The coins problem – DP solution

DP solution: constructing a temporary array table[] in bottom up manner.

$coins[\,] = \{1, 5, 10, 25\}$, s = 4 (size)

table[i] will be storing the minimum number of coins required for i value. table[n] will have result.

Base case: for n = 0, table[0] = 0

i = 1. Look for **all** the values in coins[] <= i.

Look in the table the value for i – coins[j], j<s

Only option is coins[0] = 1

Look at the value in table[i-coins[j]] = table[0] = 0

If that value + 1 is < value at table[i],

table[i] gets replaced by that value

The coins problem – DP solution

```
int minCoins(int coins[],int s,int n) // s is size of coins array
{
    // table[i] will be storing the minimum number of coins required for i
value. table[n] will have result
    int table[n+1];

    table[0] = 0; // Base case (If given value n is 0)

    for(int i=1; i<=n; i++) // Initialize all table values as Infinite
        table[i] = INT_MAX;

    for(int i=1; i<=n; i++){ // Compute minimum coins required for all values from
1 to n
        for(int j=0; j<s; j++) // Go through all coins smaller than i
          if(coins[j]<= i){
              int sub_res = table[i-coins[j]];
              if(sub_res != INT_MAX&&sub_res + 1<table[i])
                  table[i] = sub_res + 1;
          }
    }
    return table[n];
}
```

Optimal path - Intersection delays

In order to introduce the dynamic-programming approach to solving multistage problems, in this section we analyze a simple example. Figure 11.1 represents a street map connecting homes and downtown parking lots for a group of commuters in a model city. The arcs correspond to streets and the nodes correspond to intersections. The network has been designed in a diamond pattern so that every commuter must traverse five streets in driving from home to downtown. The design characteristics and traffic pattern are such that the total time spent by any commuter between intersections is independent of the route taken. However, substantial delays, are experienced by the commuters in the intersections. The lengths of these delays in minutes, are indicated by the numbers within the nodes. We would like to minimize the total delay any commuter can incur in the intersections while driving from his home to downtown.

From Bradley, S. P., A. C. Hax, and T. L. Magnanti. *Applied Mathematical Programming*. Addison-Wesley, 1977. Chapter 11 - DP

Optimal path - Intersection delays

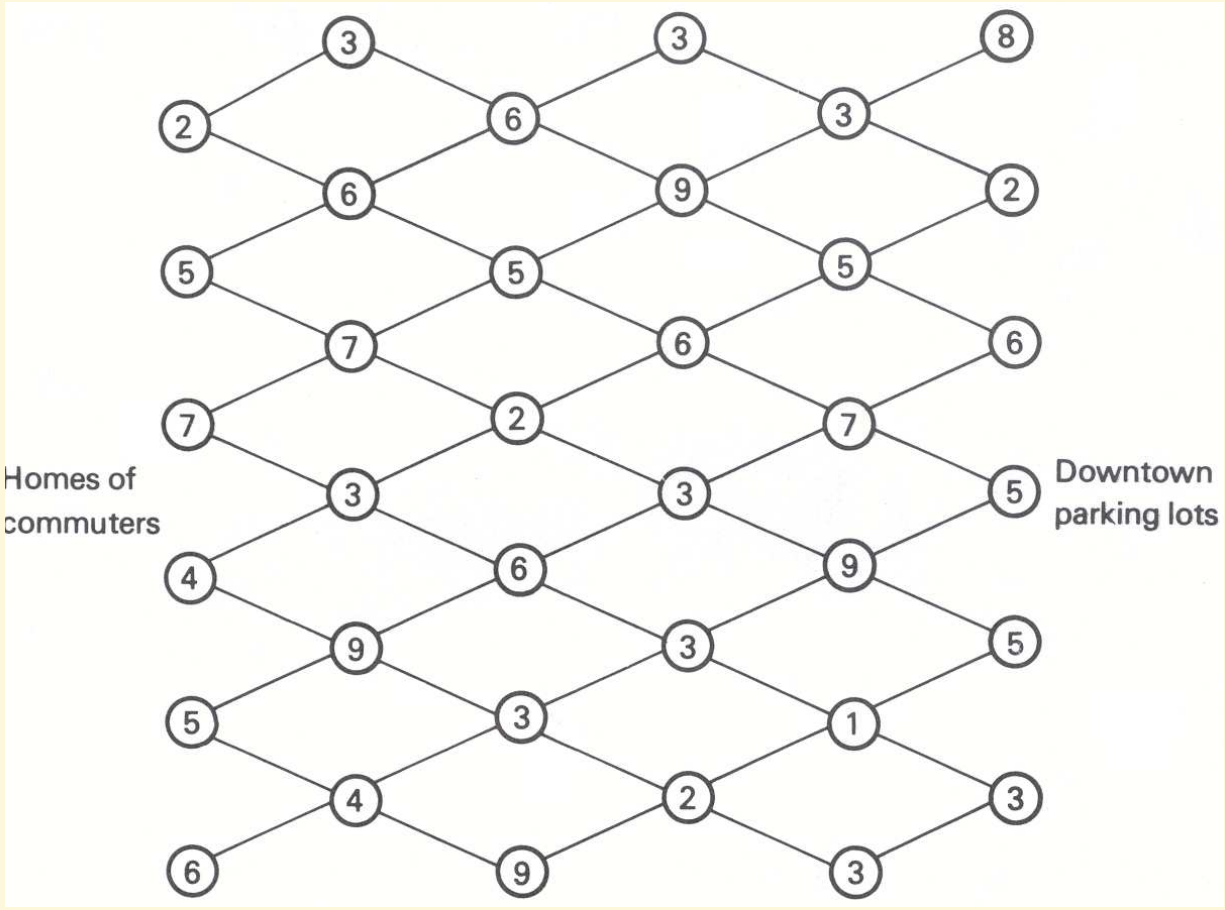


Figure 11.1 Street map with intersection delays.

Optimal path - Intersection delays

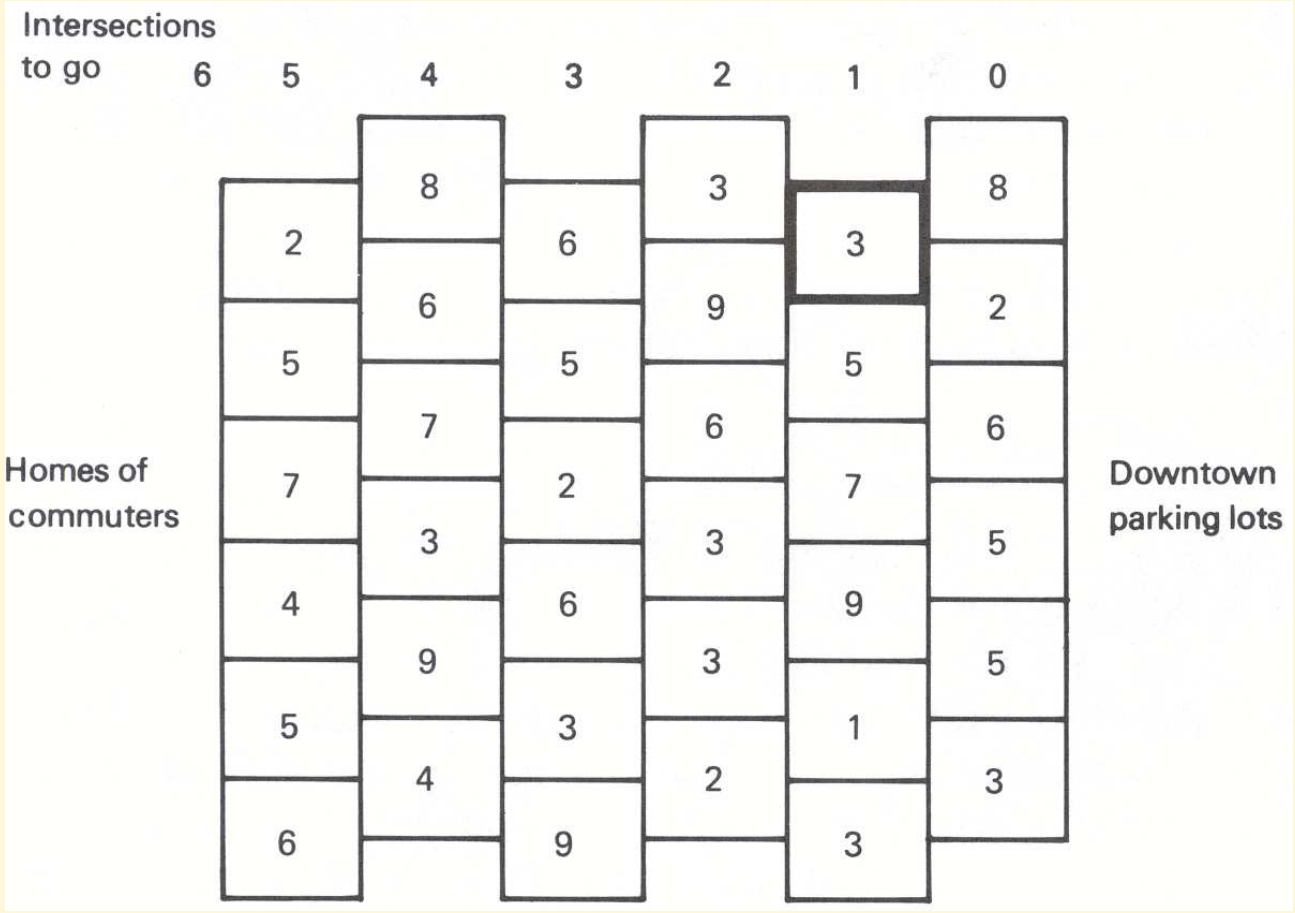


Figure 11.2 Compact representation of the network.

Optimal path - Intersection delays

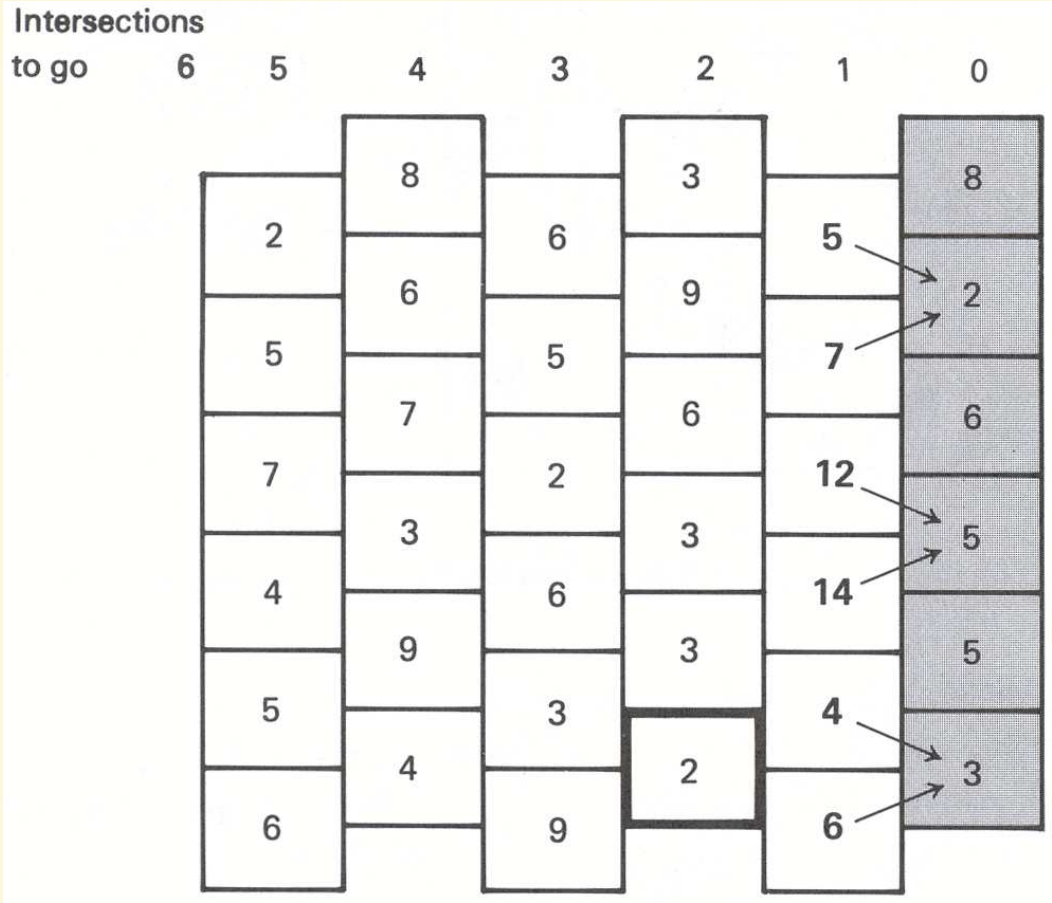


Figure 11.3 Decisions and delays with one intersection to go.

Optimal path - Intersection delays

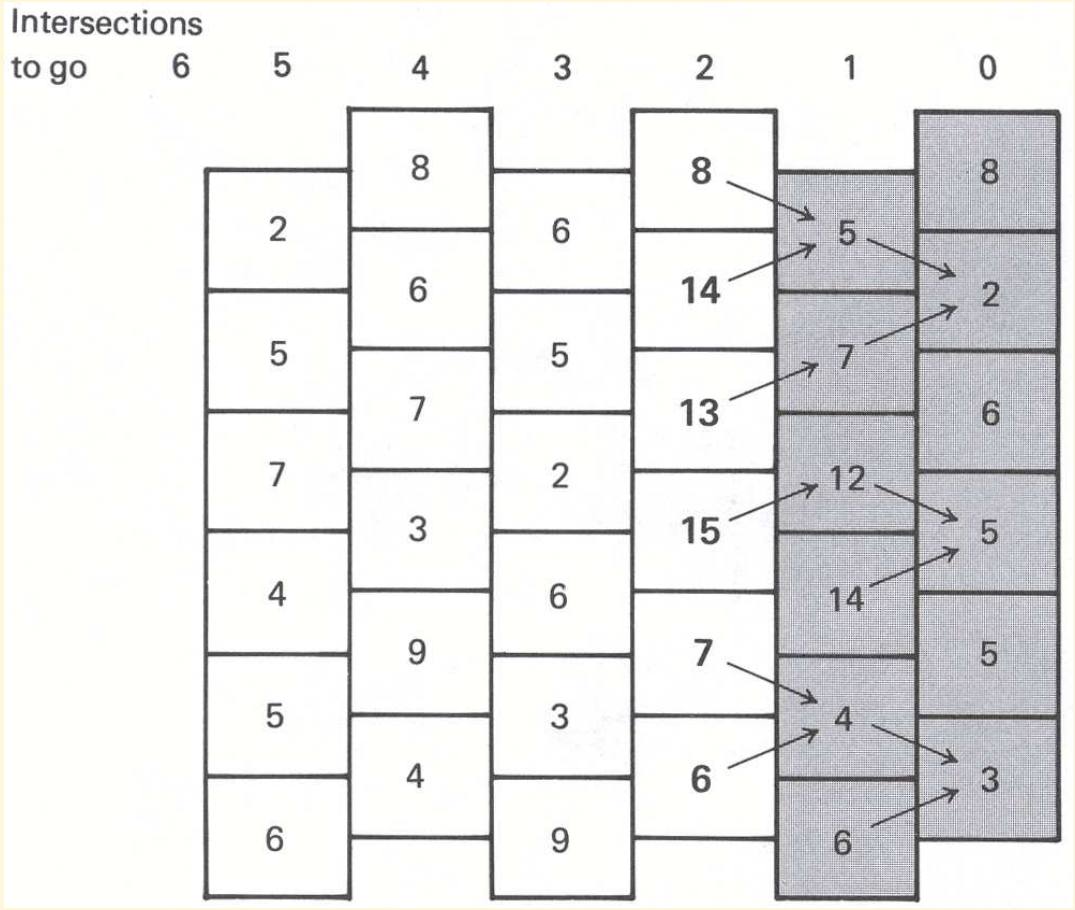
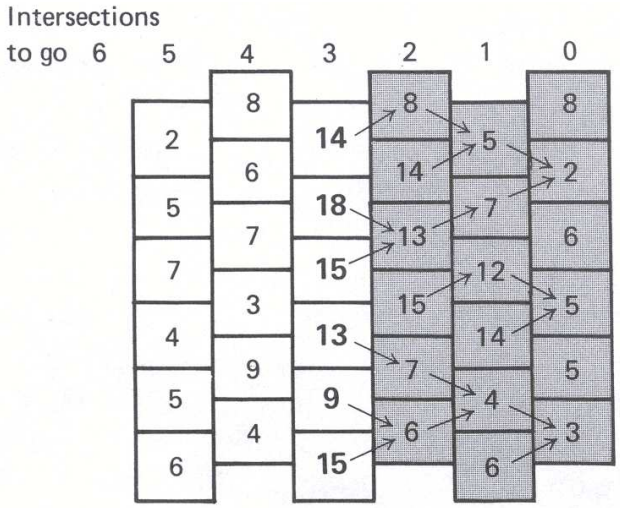
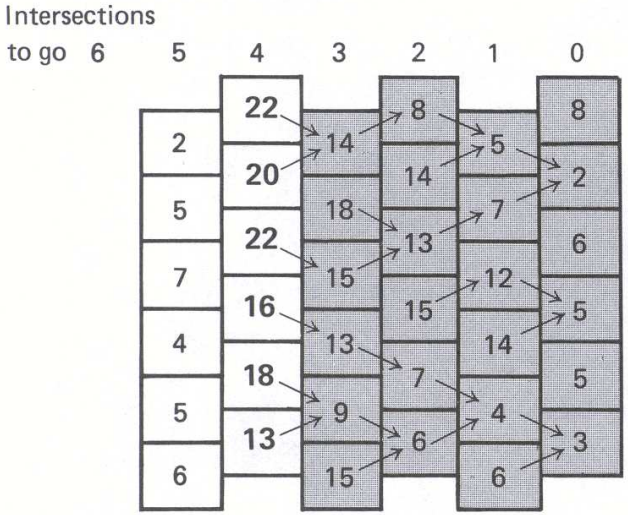


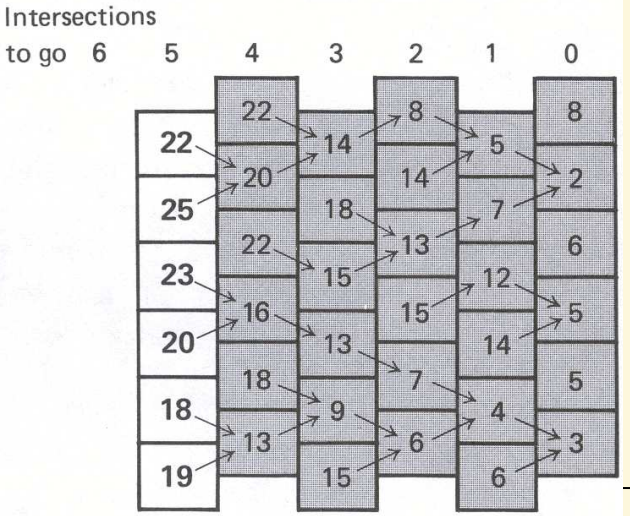
Figure 11.4 Decisions and delays with two intersections to go.



(a) Three intersections to go



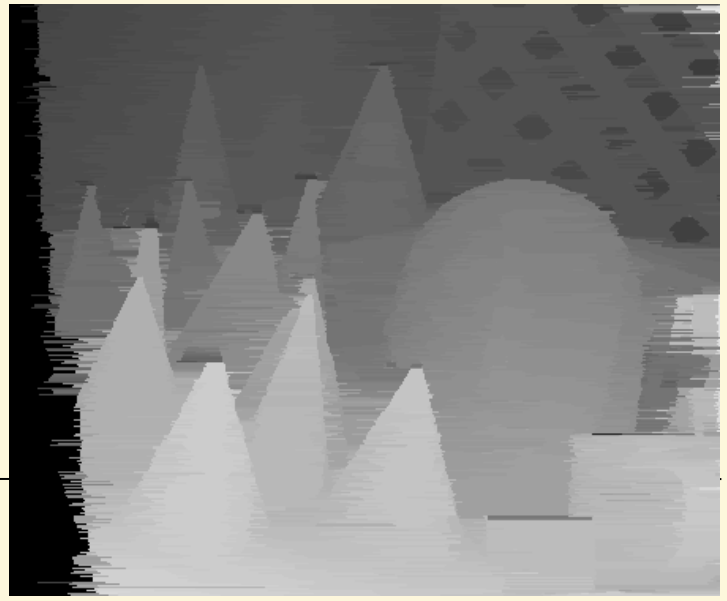
(b) Four intersections to go



(c) Five intersections to go

Figure 11.5 Charts of optimal delays and decisions.

Dynamic Programming Results

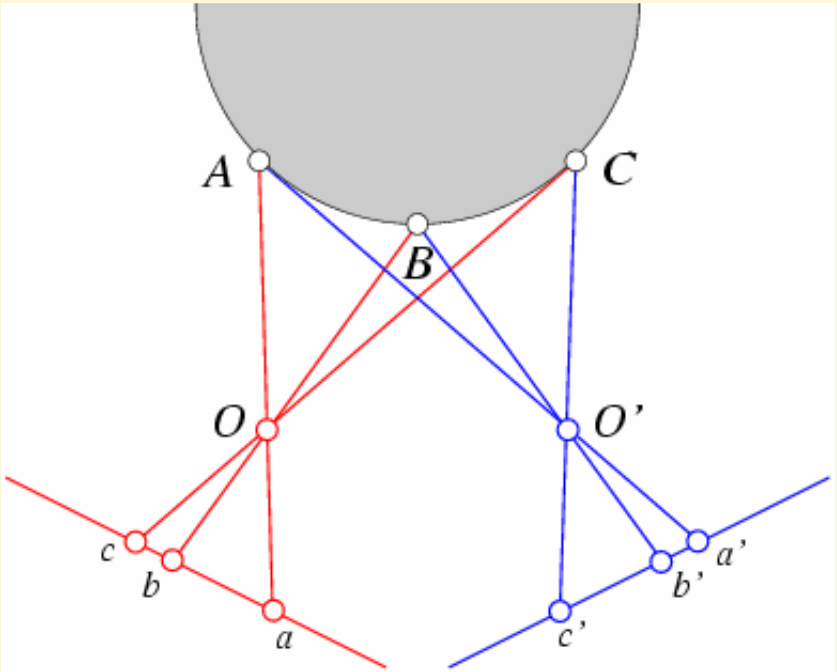


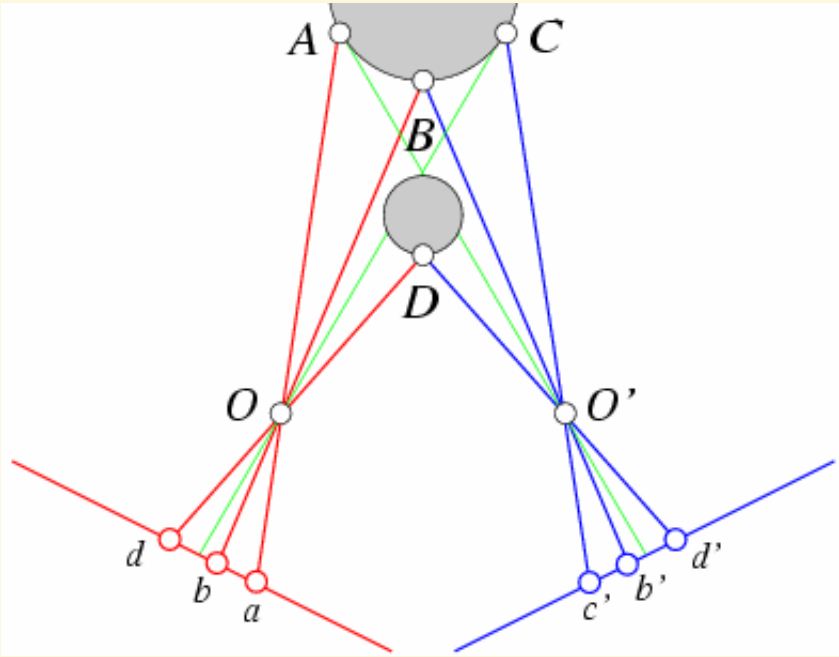
DP for stereo matching

Constraints:

- epipolar
- ordering
- uniqueness

Ordering constraint in stereo

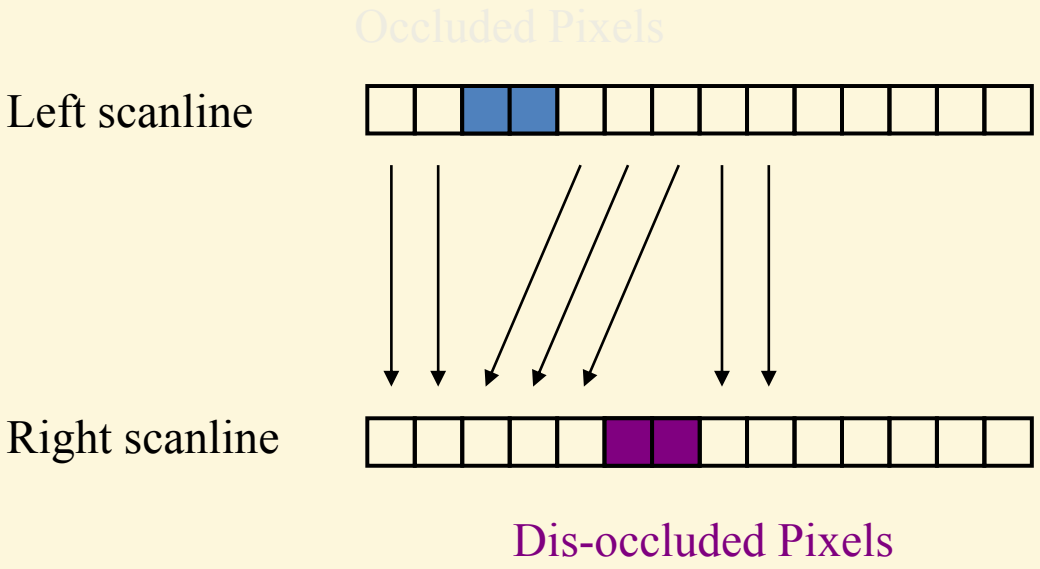




In general the points are in the same order on both epipolar lines.

But it is not always the case..

Search Over Correspondences



Three cases:

- Sequential – cost of match
- Occluded – cost of no match
- Disoccluded – cost of no match

Search Over Correspondences

$I_l(i)$ and $I_r(j)$, $1 \le i, j \le N$, where N is the number of pixels in each line.

Let d_{ij} be the cost associated with matching pixel $I_l(i)$ with pixel $I_r(j)$.

$$d_{ij} = (I_l(i) - I_r(j))^2$$

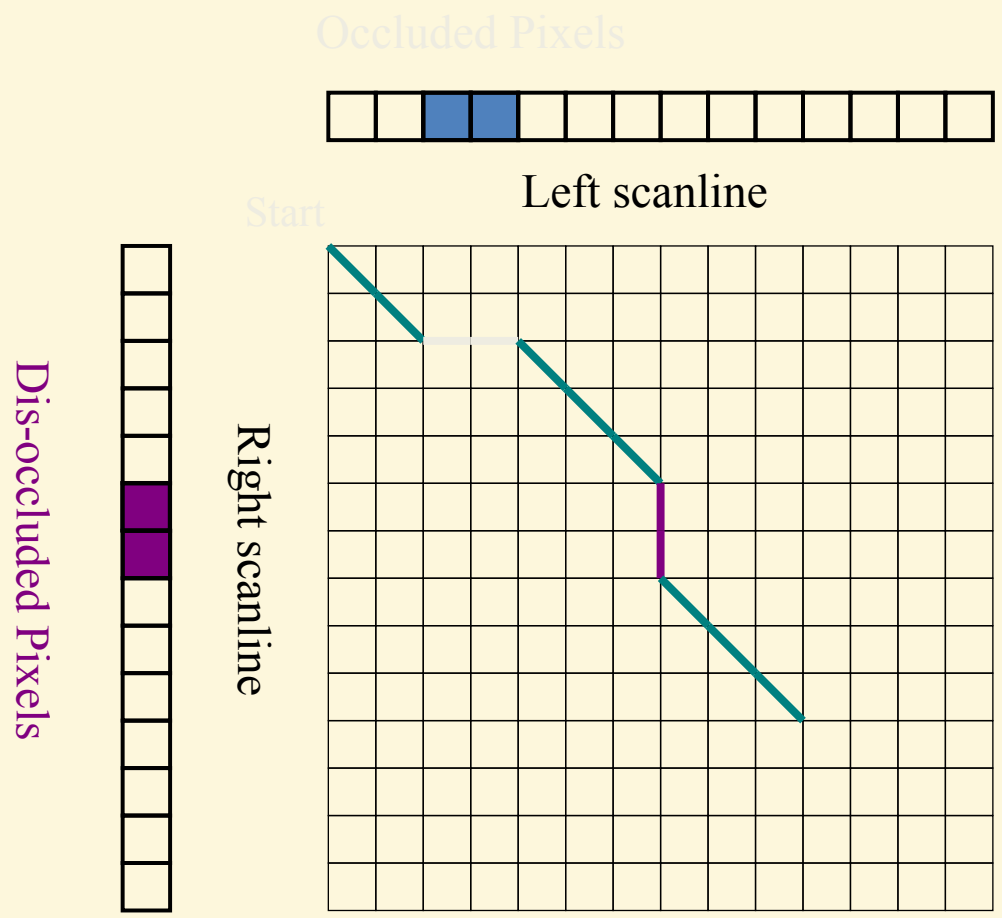
The cost of skipping a pixel (in either scanline) is given by a constant `occ`.

Optimal (minimal cost) alignment of two scalines (recursively):

1. `D(0,j) = j * occ`
2. `D(i,0) = i * occ`
2. $D(1, 1) = d_{11}$,
3. `D(i, j) =`

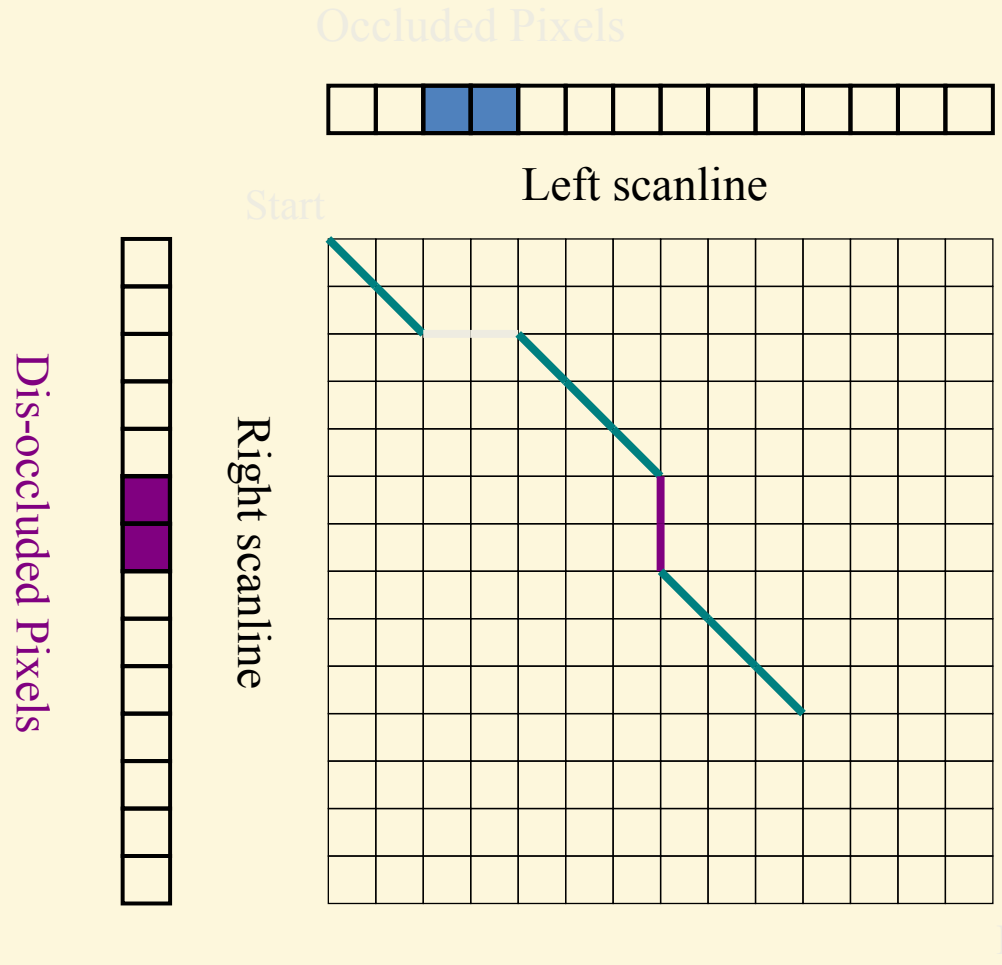
 $= \min(D(i-1,j-1) + d_{ij},\ D(i-1,j) + occ,\ D(i,j-1) + occ)$

Stereo Matching with DP



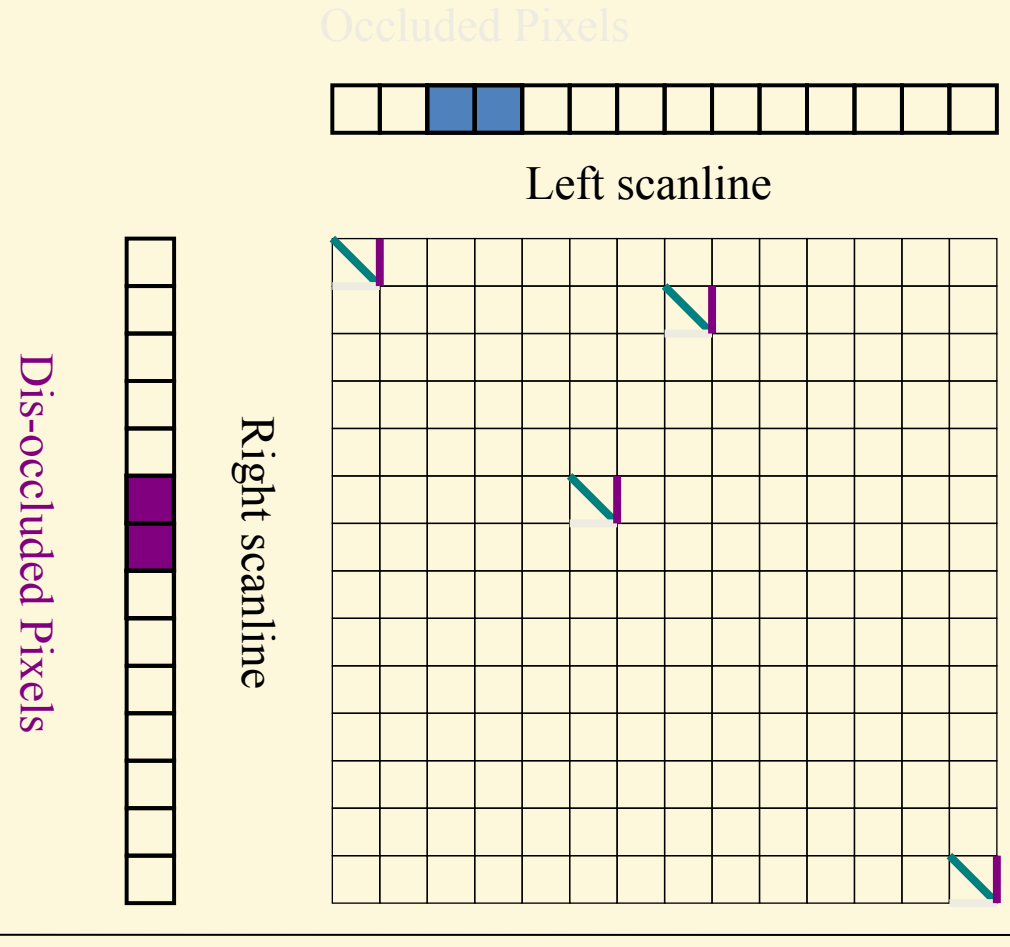
Dynamic programming yields the optimal path through grid. This is the best set of matches that satisfy the ordering constraint

Stereo Matching with DP



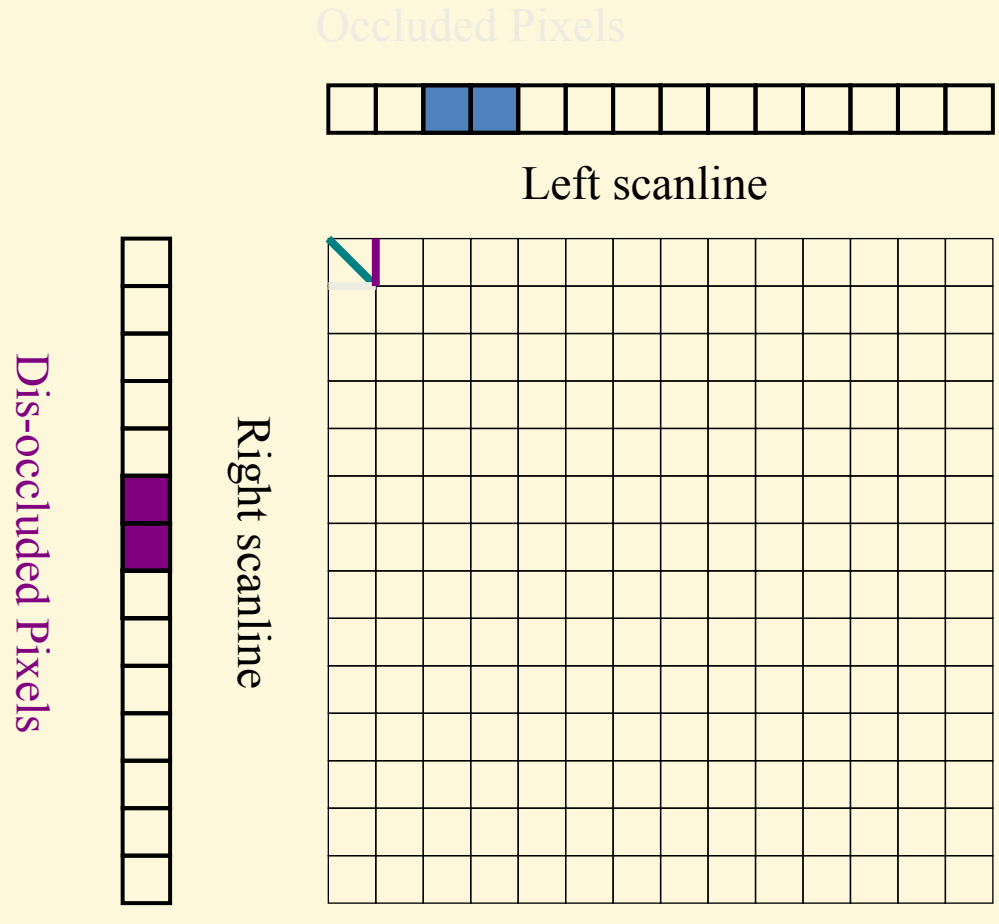
- **Enforces ordering constraint.**
- **Given appropriate cost functions, solves for best path (matches, occlusions, disocclusions).**

Stereo Matching with Dynamic Programming



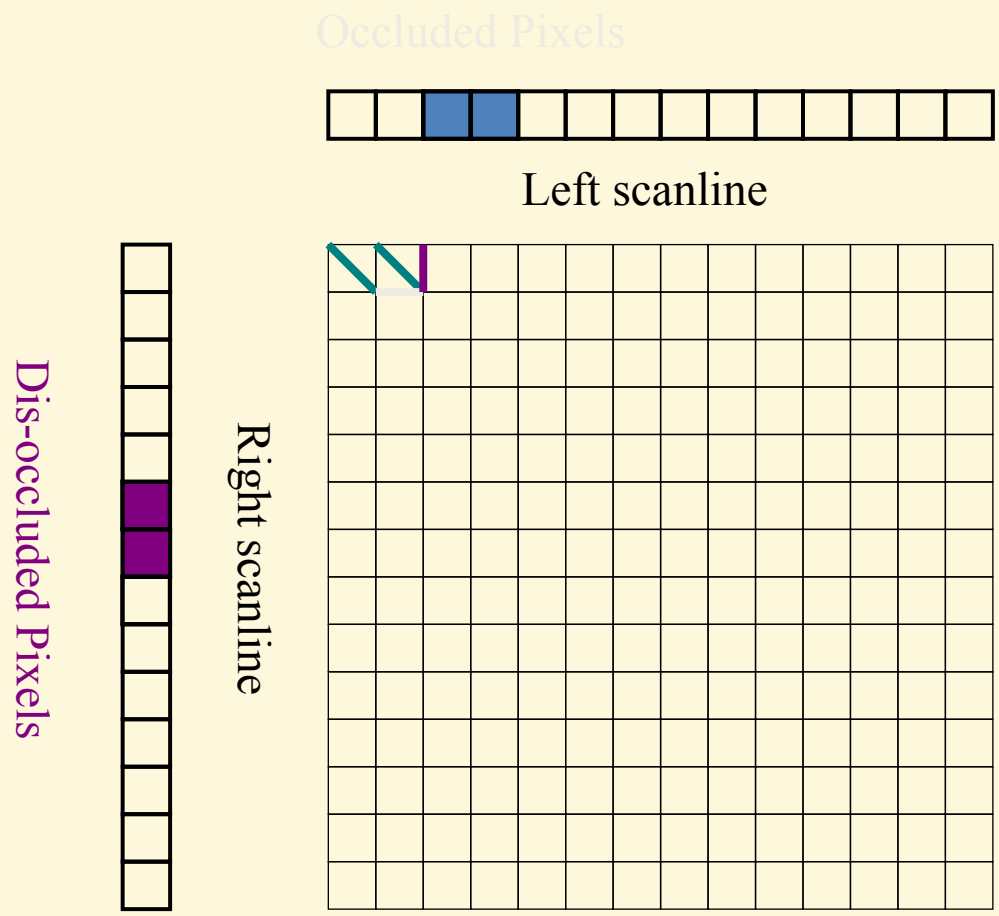
Scan across grid computing optimal cost for each node given its upper-left neighbors. Backtrack from the terminal to get the optimal path.

Stereo Matching with Dynamic Programming



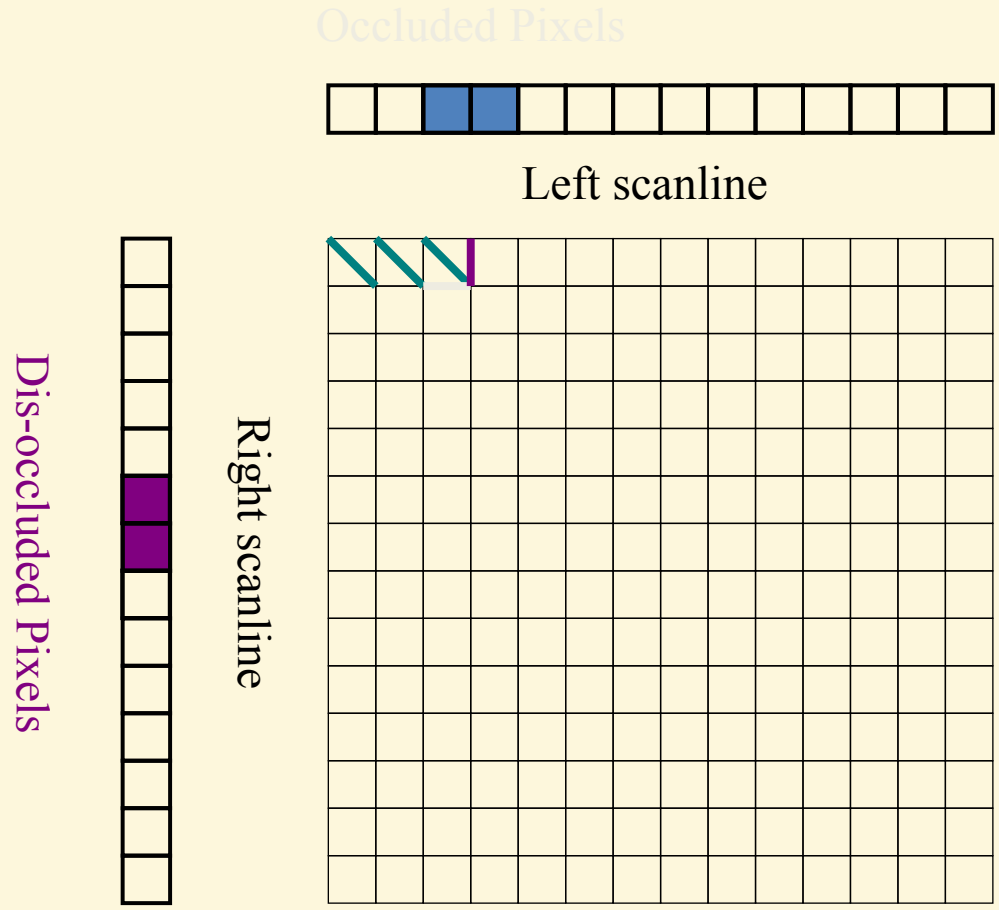
Scan across grid computing optimal cost for each node given its upper-left neighbors. Backtrack from the terminal to get the optimal path.

Stereo Matching with Dynamic Programming



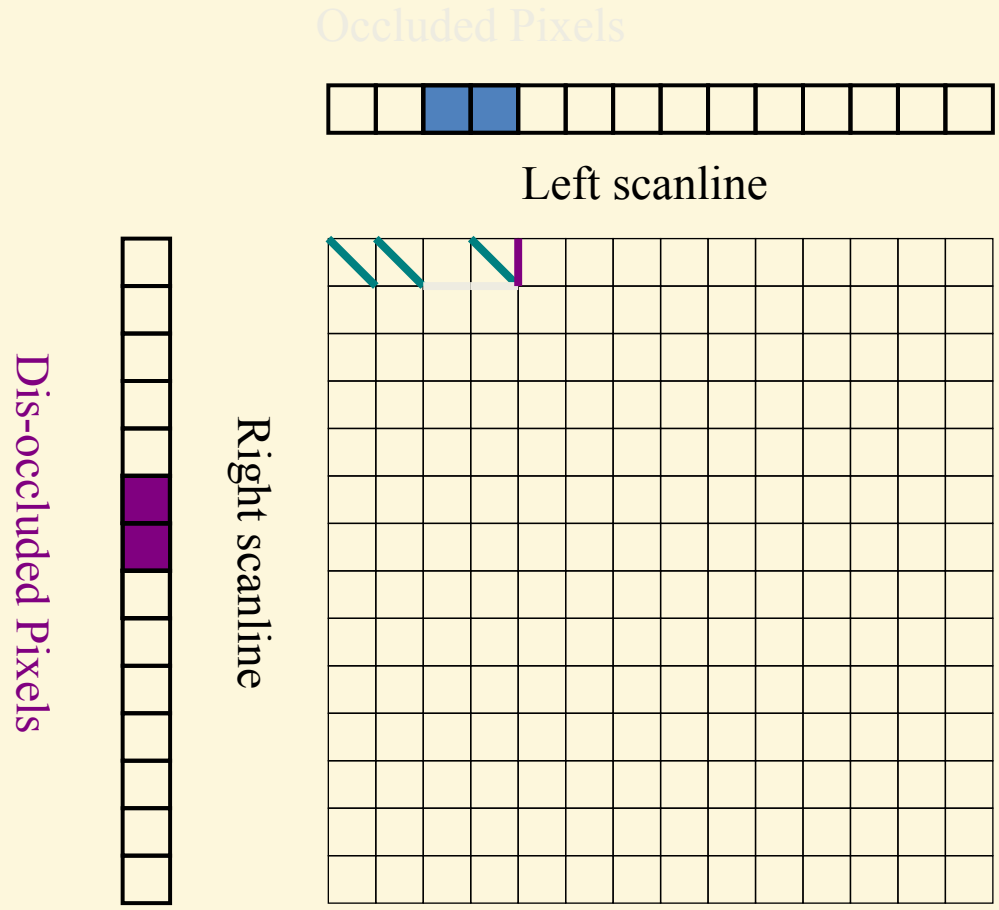
Scan across grid computing optimal cost for each node given its upper-left neighbors. Backtrack from the terminal to get the optimal path.

Stereo Matching with Dynamic Programming



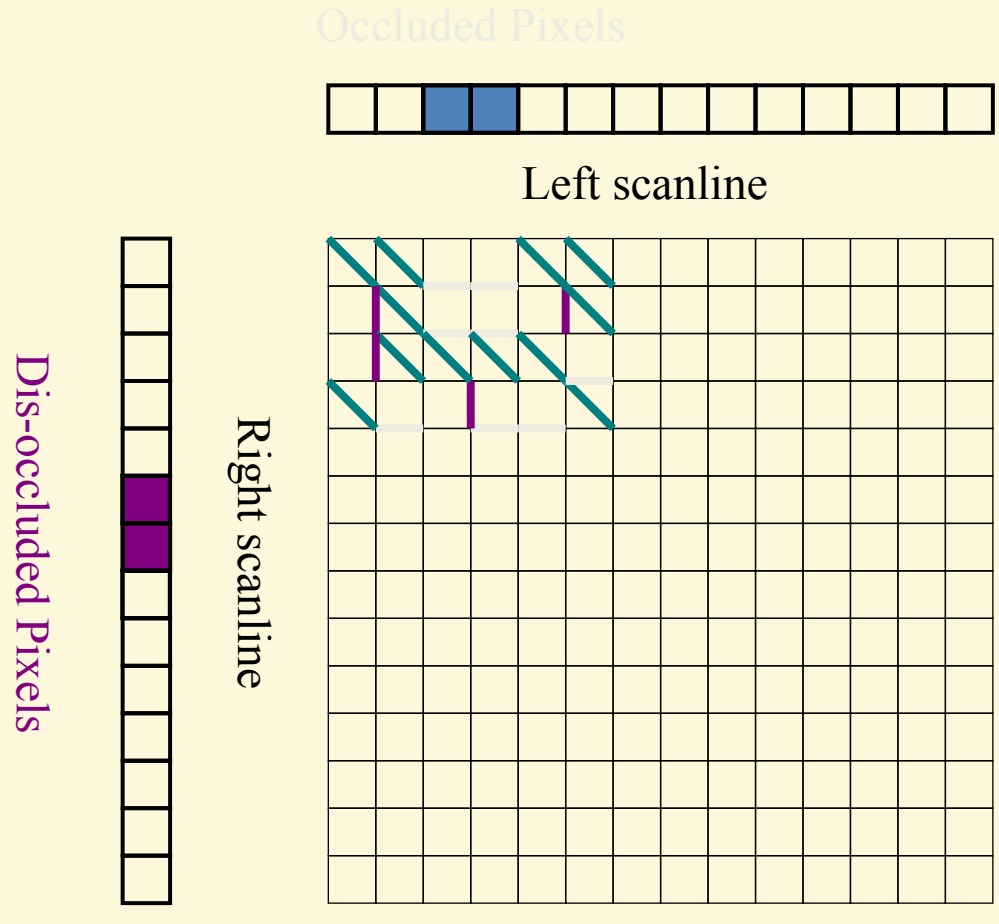
Scan across grid computing optimal cost for each node given its upper-left neighbors. Backtrack from the terminal to get the optimal path.

Stereo Matching with Dynamic Programming



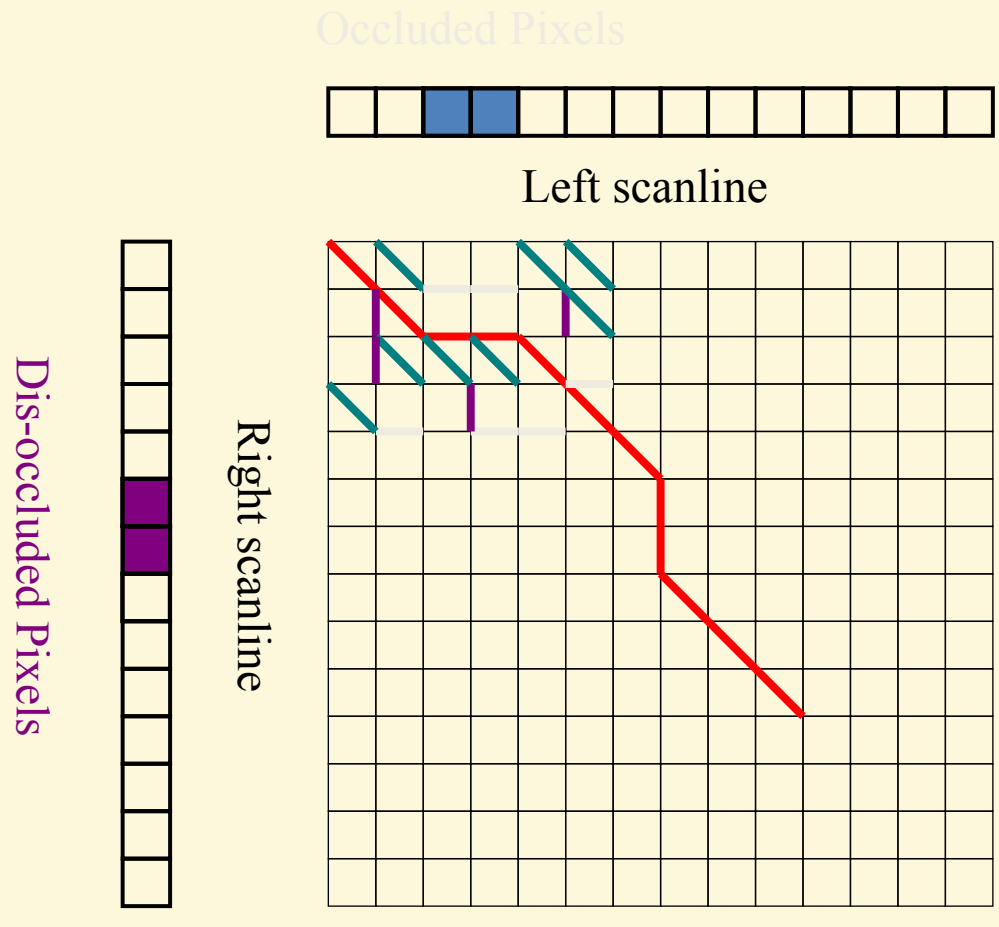
Scan across grid computing optimal cost for each node given its upper-left neighbors. Backtrack from the terminal to get the optimal path.

Stereo Matching with Dynamic Programming



Scan across grid computing optimal cost for each node given its upper-left neighbors. Backtrack from the terminal to get the optimal path.

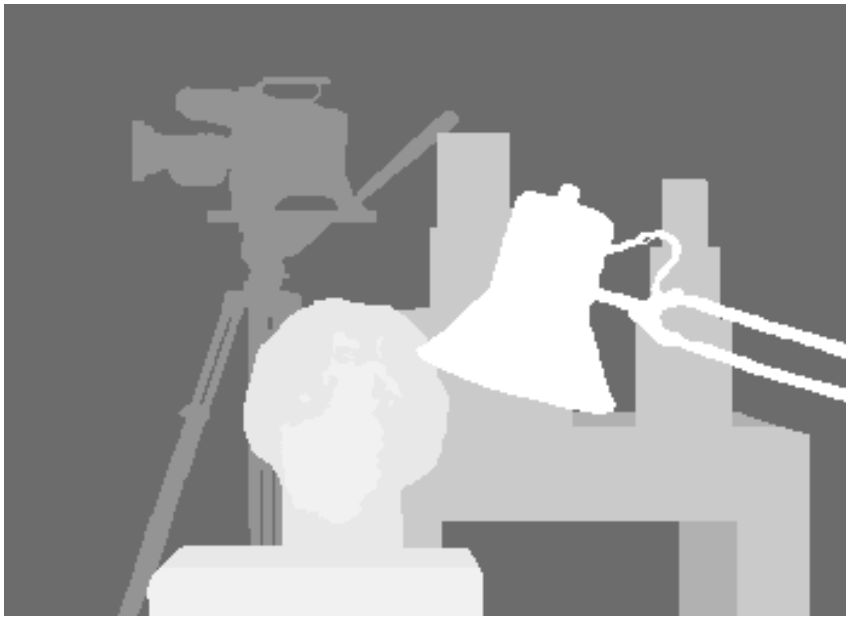
Stereo Matching with Dynamic Programming



Scan across grid computing optimal cost for each node given its upper-left neighbors. Backtrack from the terminal to get the optimal path.

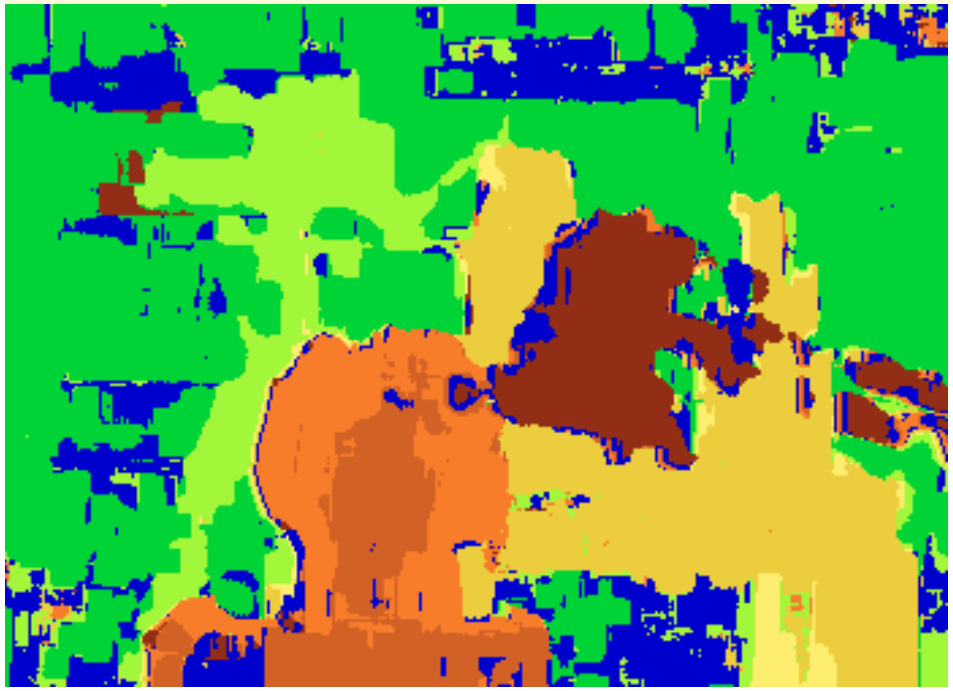
Data from University of Tsukuba

Scene



Ground truth

Results with window correlation



Window-based matching
(best window size)

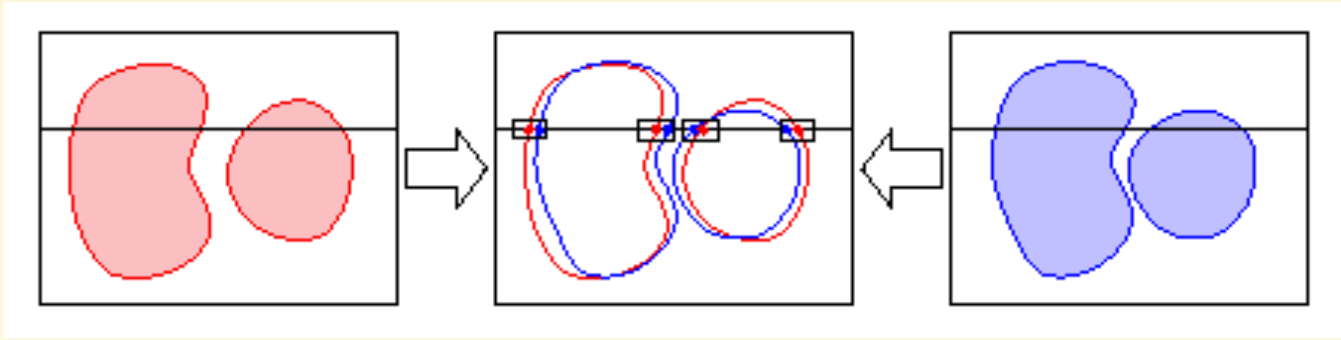
Ground truth

Results with DP

- Much better results already!
- But no consistency between scanlines!

Computing Correspondence

- Another approach is to match *edges* rather than windows of pixels:



- Which method is better?
  - Edges tend to fail in dense texture (outdoors)
  - Correlation tends to fail in smooth featureless areas