BIOINFORMATICS

How do we compare biological sequences?

Marco Beccuti

Università degli Studi di Torino

Dipartimento di Informatica

March 2019

Outline

1. Introduction to Sequence Alignment
2. Hamming distance for similarity between sequences
3. Alignment Game and the Longest Common Subsequence;
4. The Manhattan Tourist Problem;
5. The Change Problem;
6. Dynamic programming and backtracking pointers;
7. From Manhattan to Alignment Graph;
8. From Global to Local Alignment;
9. Penalizing Insertions and Deletions in Sequence Alignment;
10. Space-Efficient Sequence Alignment;
11. Multiple Sequence Alignment.

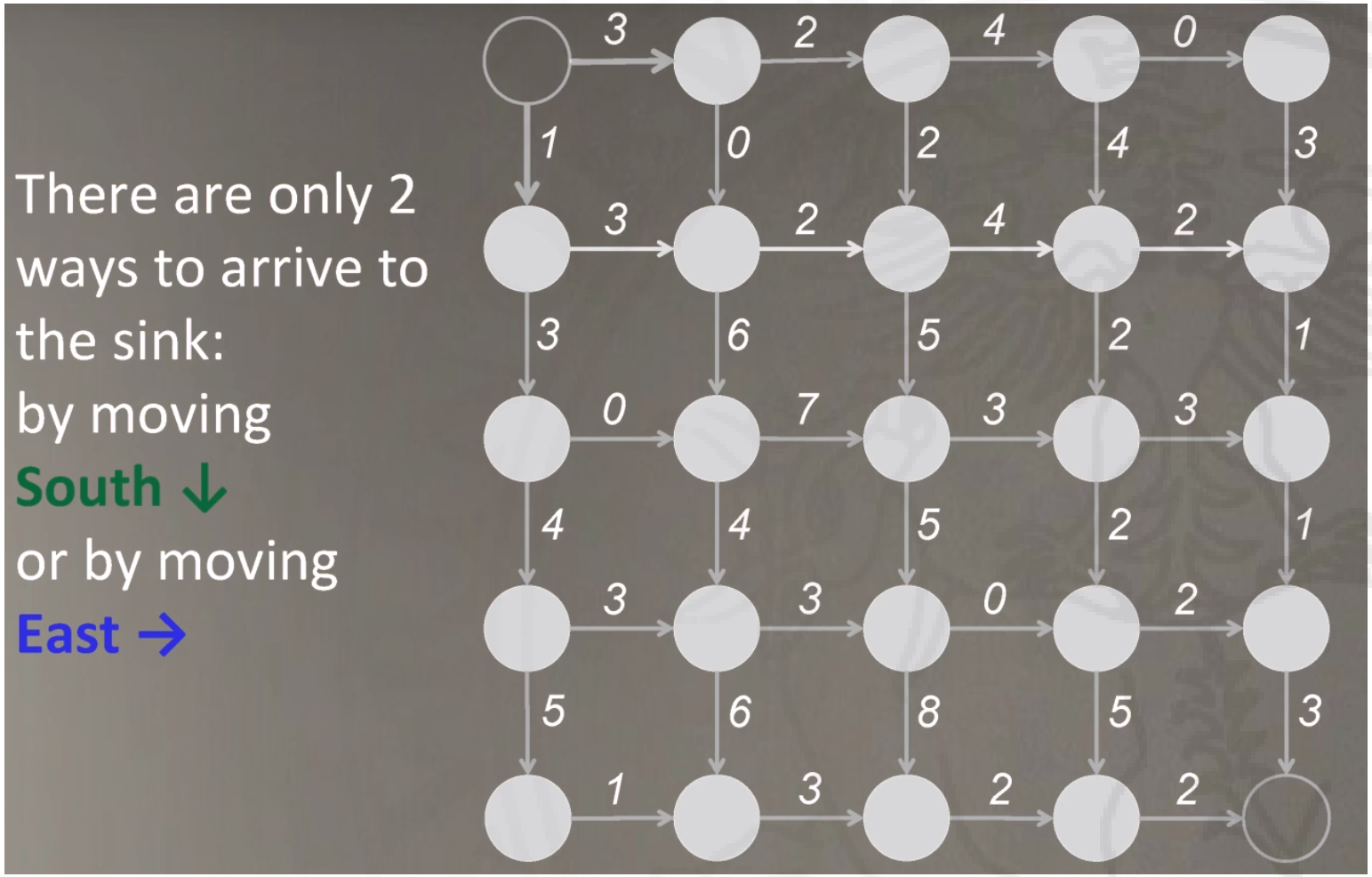
Chapter 5 in ***Bioinformatics Algorithms: An active Learning Approach (Vol.1).***

Part 2

Dynamic Programming and Backtracking Pointers

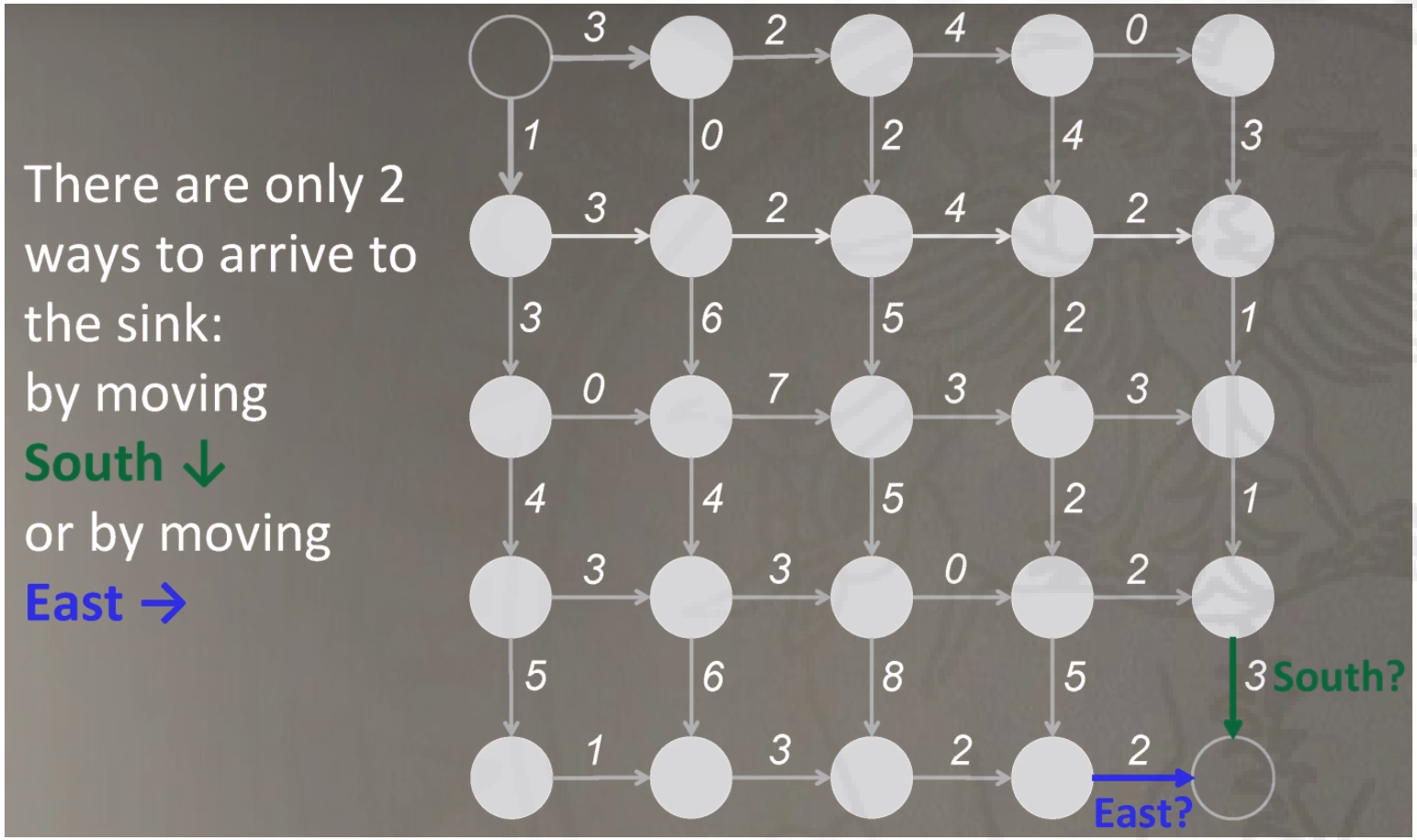
Dynamic Programming and Backtracking Pointers

- We come back to Longest Path Problem in a grid;



Dynamic Programming and Backtracking Pointers

- We come back to Longest Path Problem in a grid;

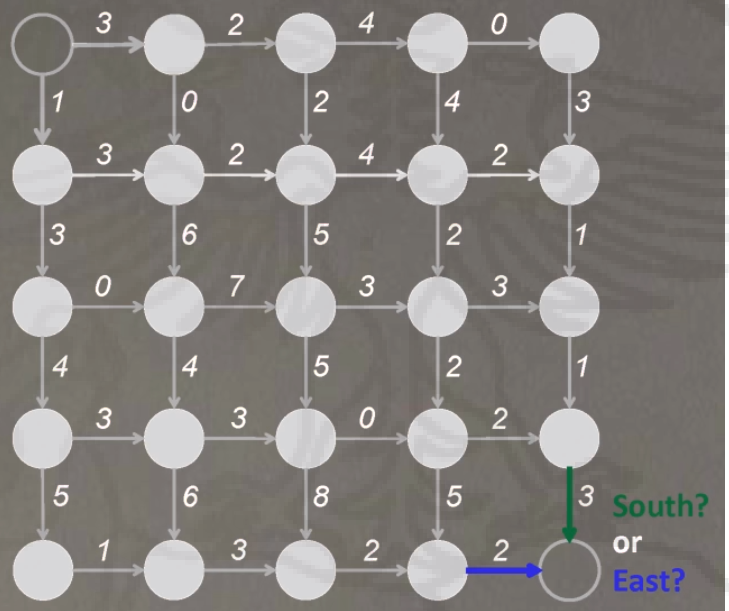


Dynamic Programming and Backtracking Pointers

- We come back to Longest Path Problem in a grid;

South or East?

SouthOrEast(*i,j*):
the length of the longest path from (0,0) to (*i,j*)



SouthOrEast(*n,m*)=

MAX { **SouthOrEast**(*n-1,m*)+weight of edge "↓"into (*n,m*)
SouthOrEast(*n,m-1*)+ weight of edge "→"into (*n,m*)

Dynamic Programming and Backtracking Pointers

- A recursive algorithm can easily define to compute the Longest Path Problem in a grid;

```
SouthOrEast(i,j)
If i=0  and j=0
  return 0
x= - infinity, y= -infinity
If i>0
  x ← SouthOrEast(i-1,j) + weight of the vertical edge into (I,j)
If j>0
  y ← SouthOrEast(I,j-1) + weight of the horizontal  edge into (I,j)
return max{x,y}
```

Dynamic Programming and Backtracking Pointers

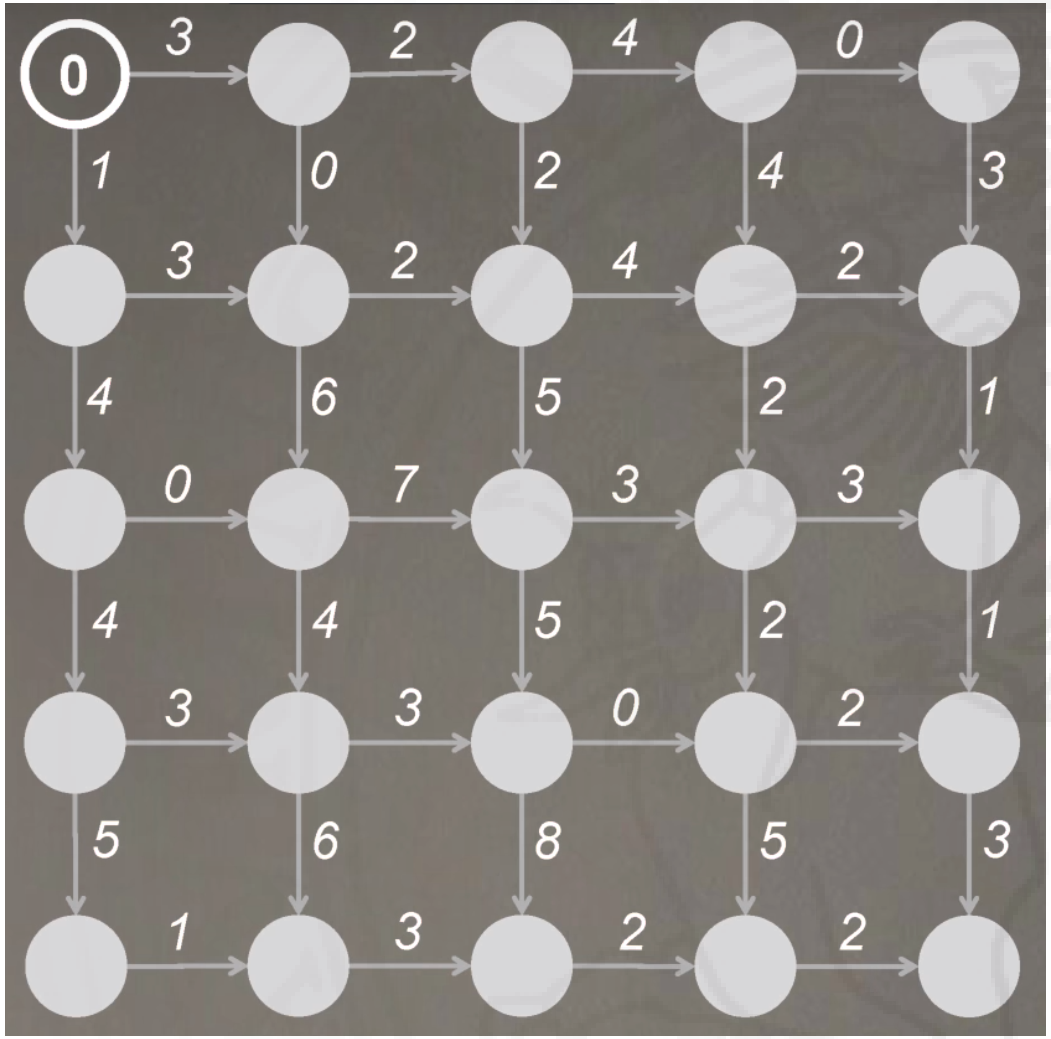
- A recursive algorithm can easily define to compute the Longest Path Problem in a grid;

```
SouthOrEast(i,j)
If i=0  and j=0
  return 0
x= - infinity, y= -infinity
If i>0
  x ← SouthOrEast(i-1,j) + weight of the vertical edge into (I,j)
If j>0
  y ← SouthOrEast(I,j-1) + weight of the horizontal  edge into (I,j)
return max{x,y}
```

- It visits all the possible paths: ***it is correct. but it is too expensive!!***

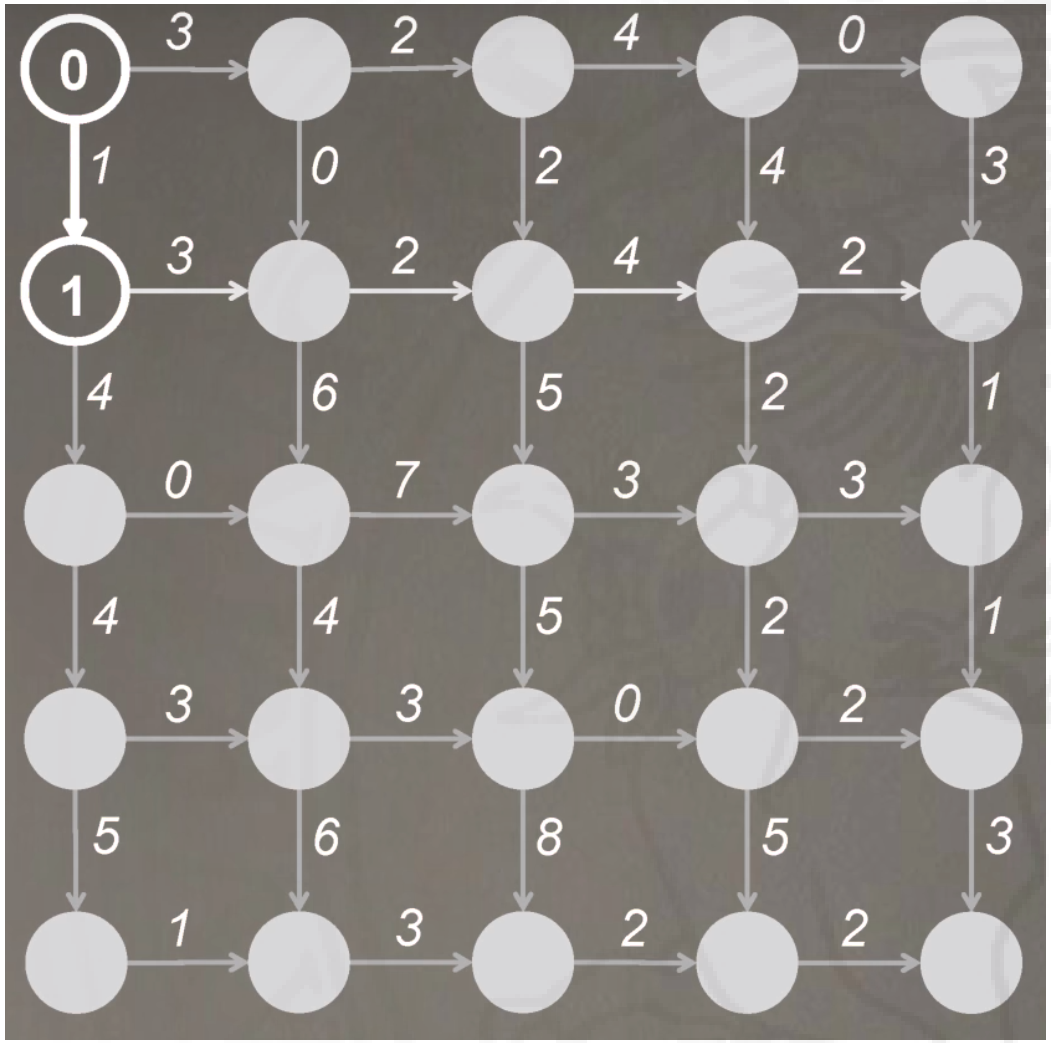
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



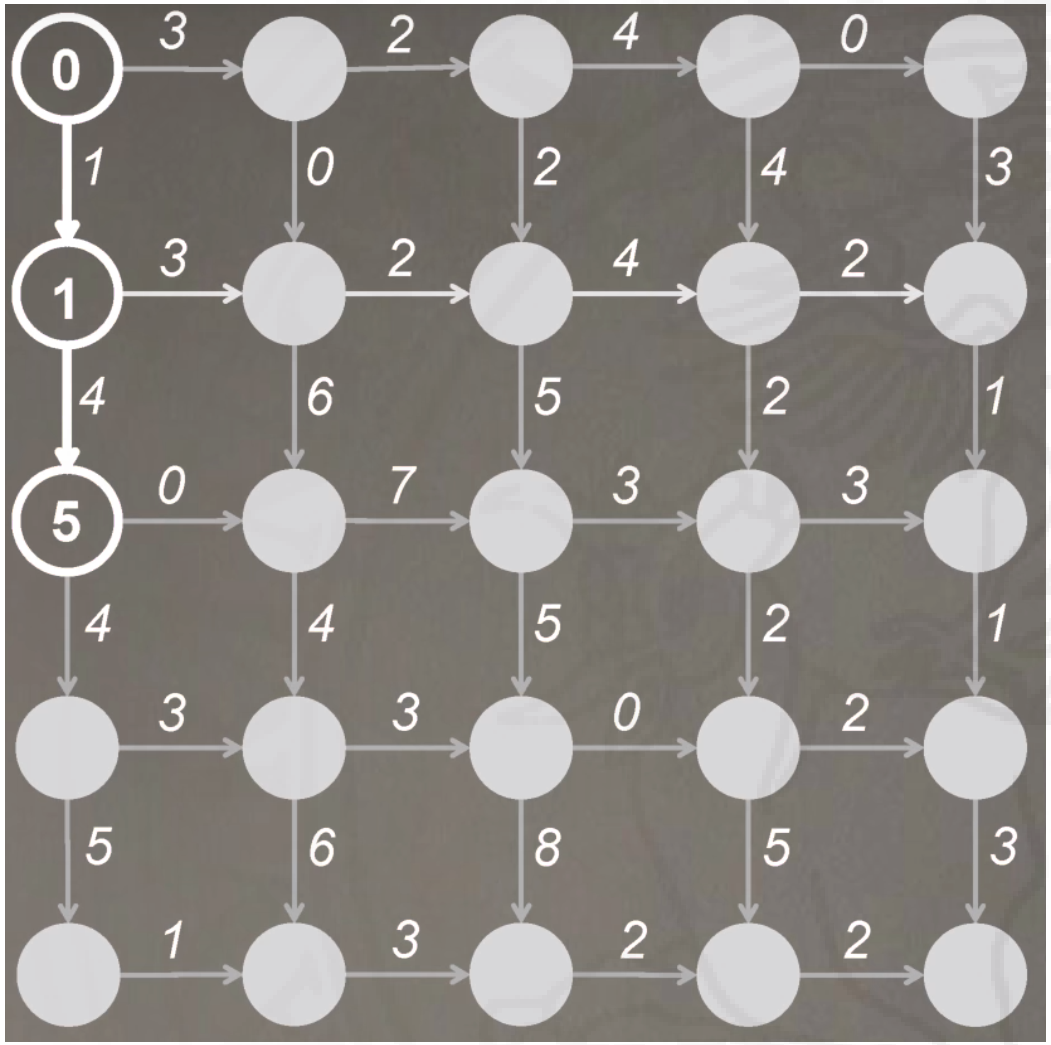
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



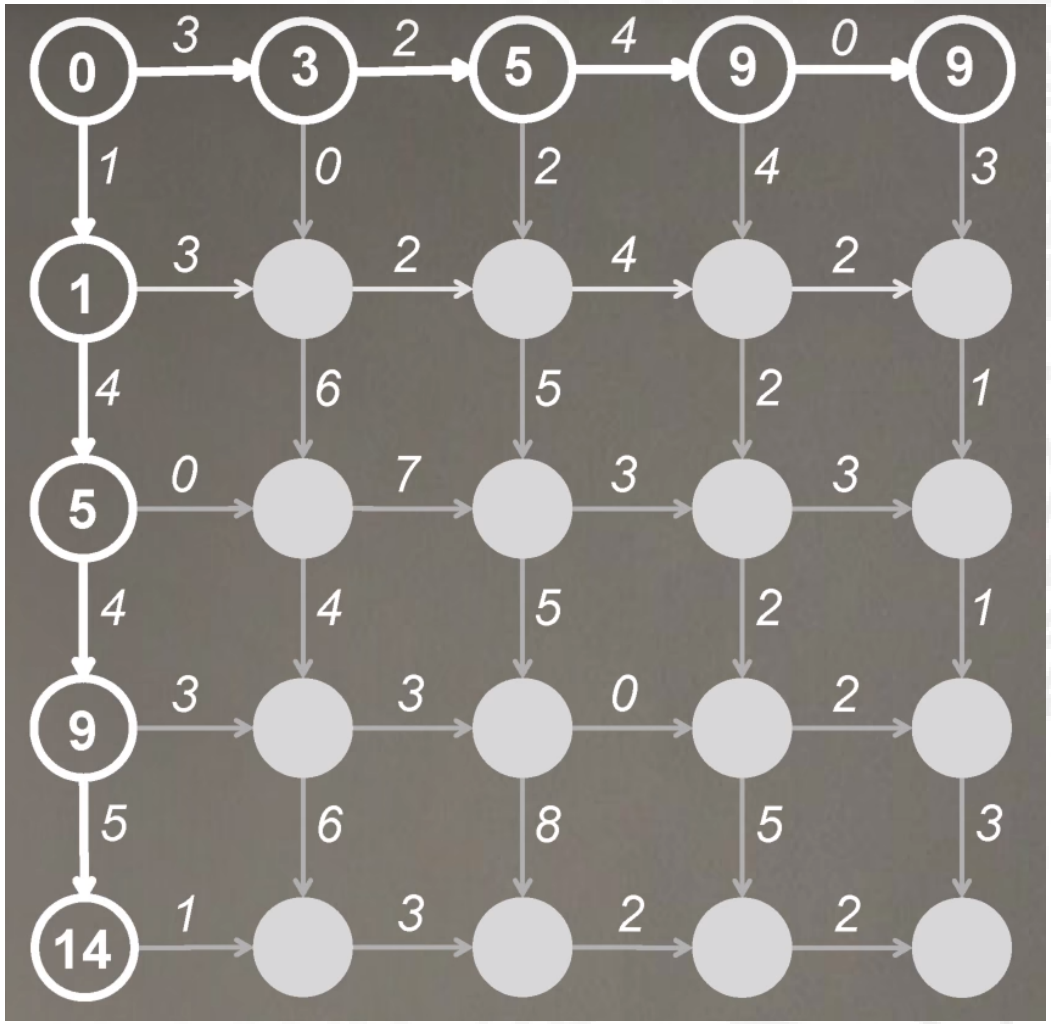
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



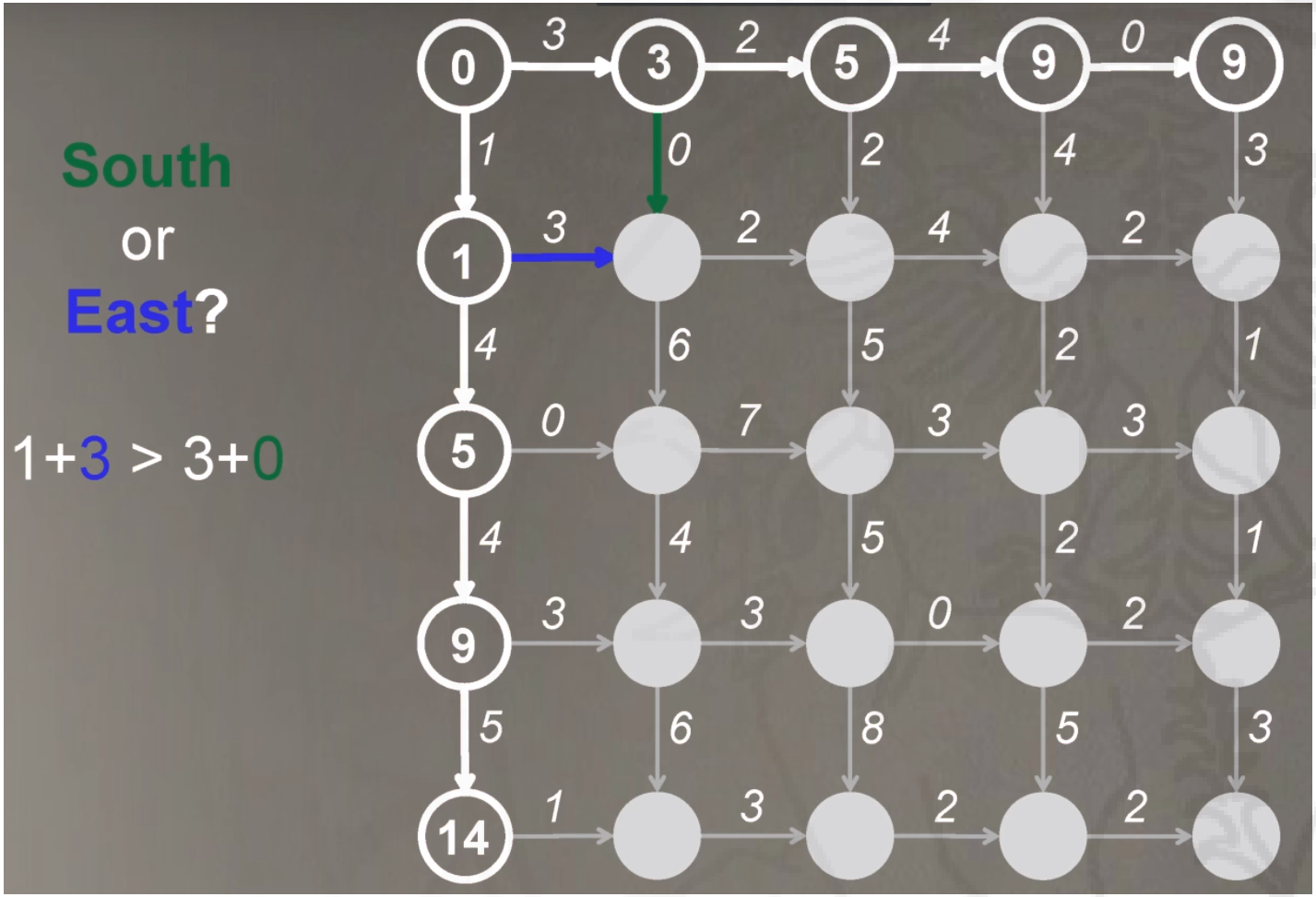
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



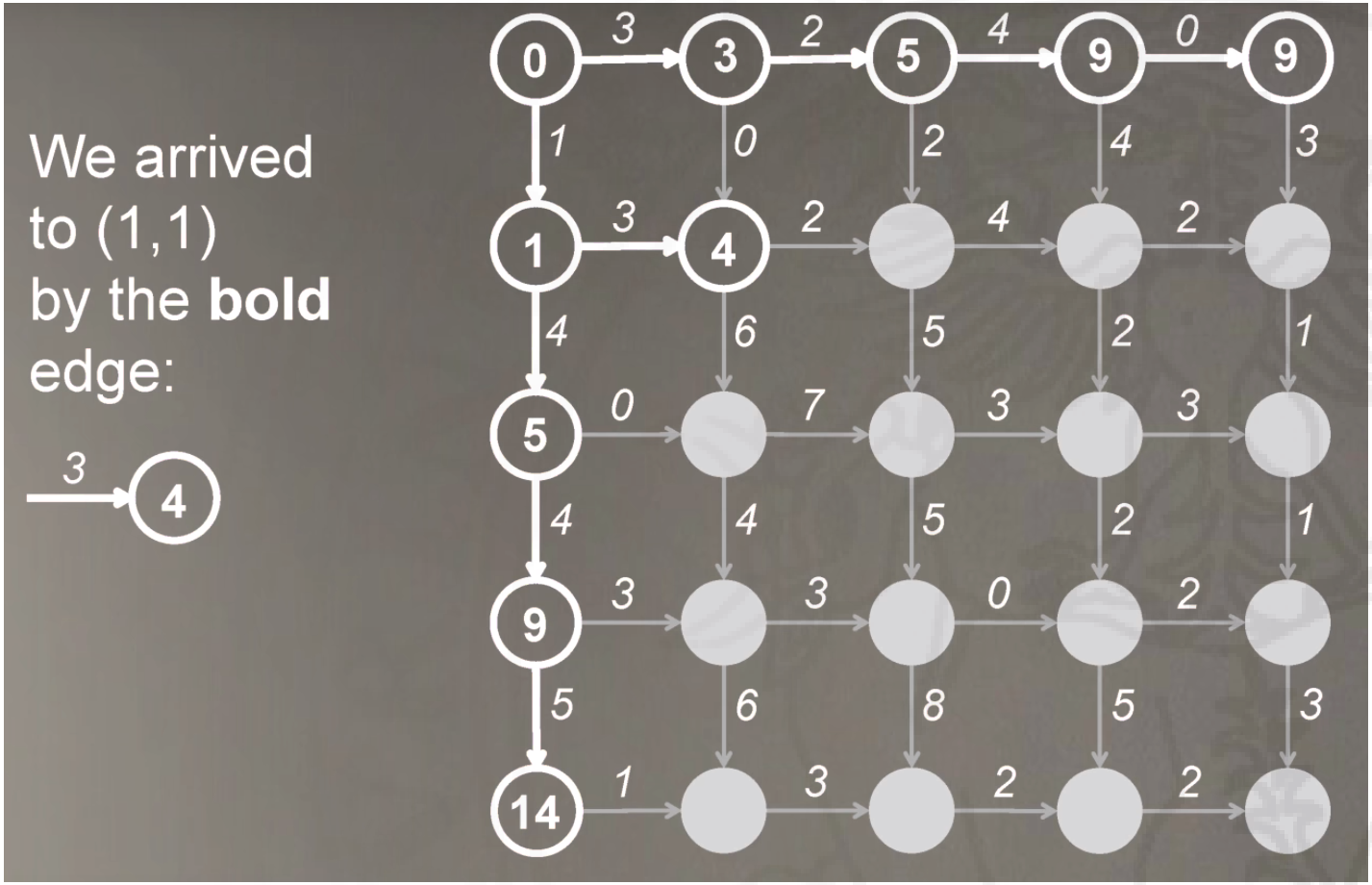
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



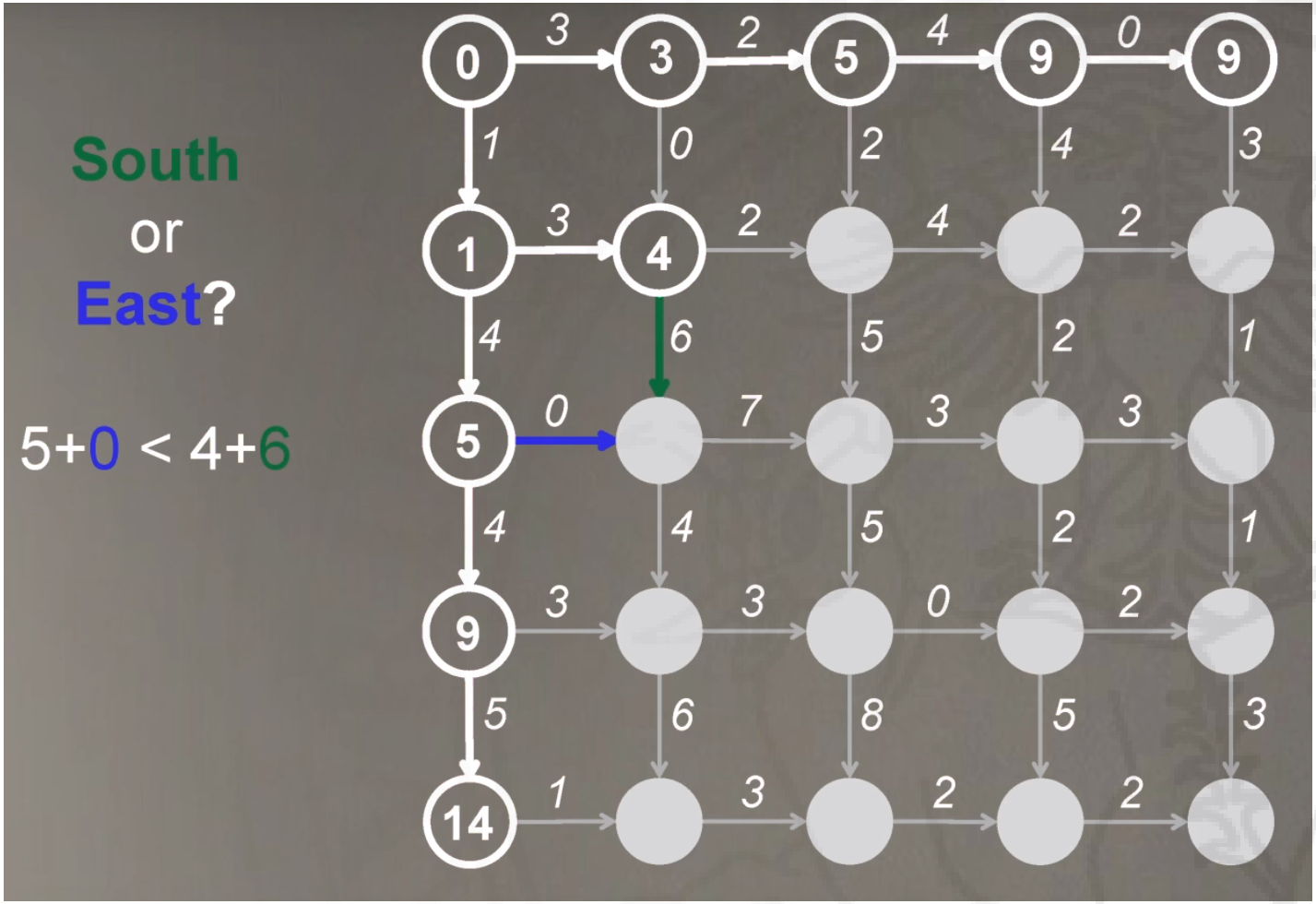
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



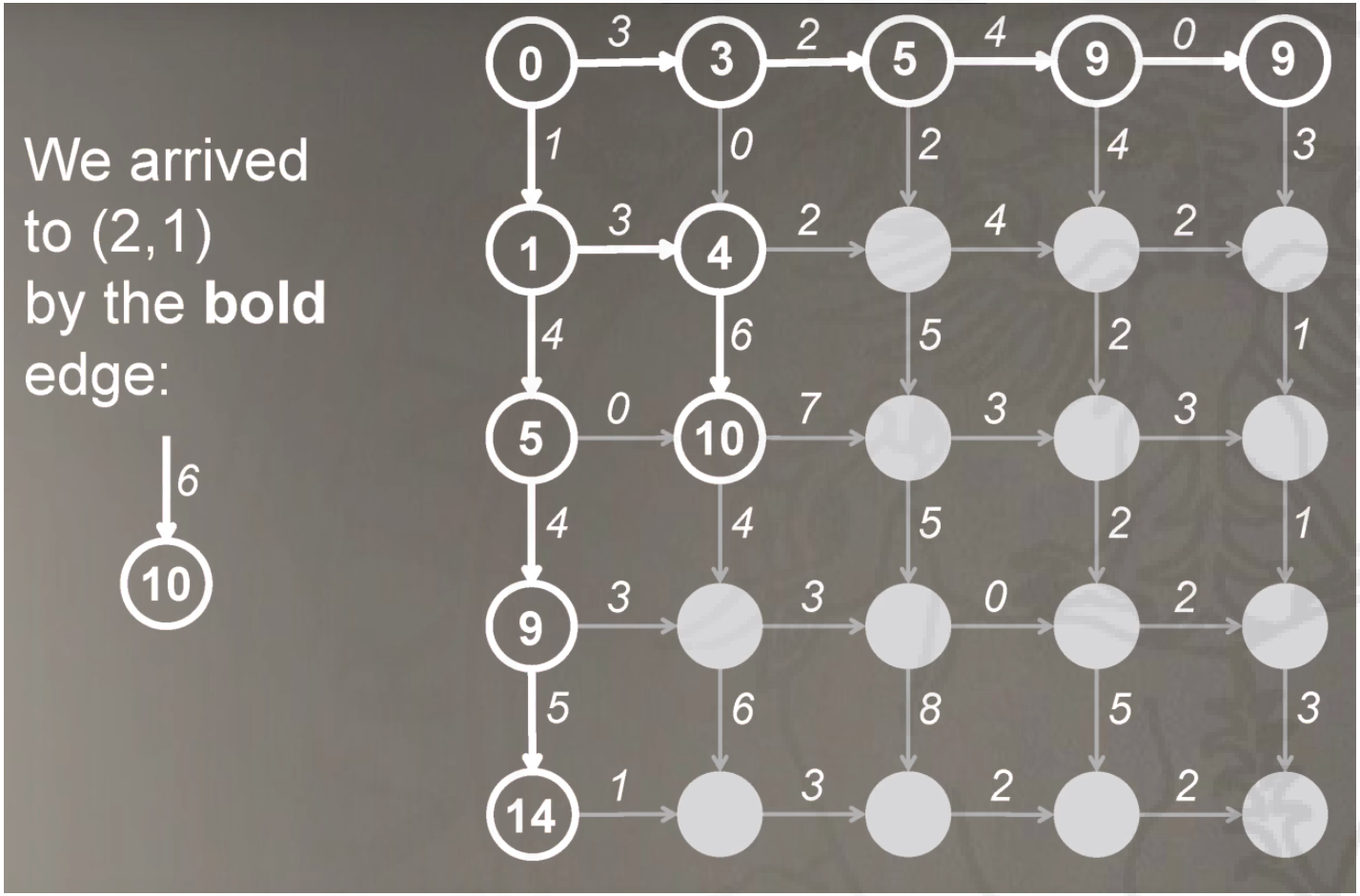
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



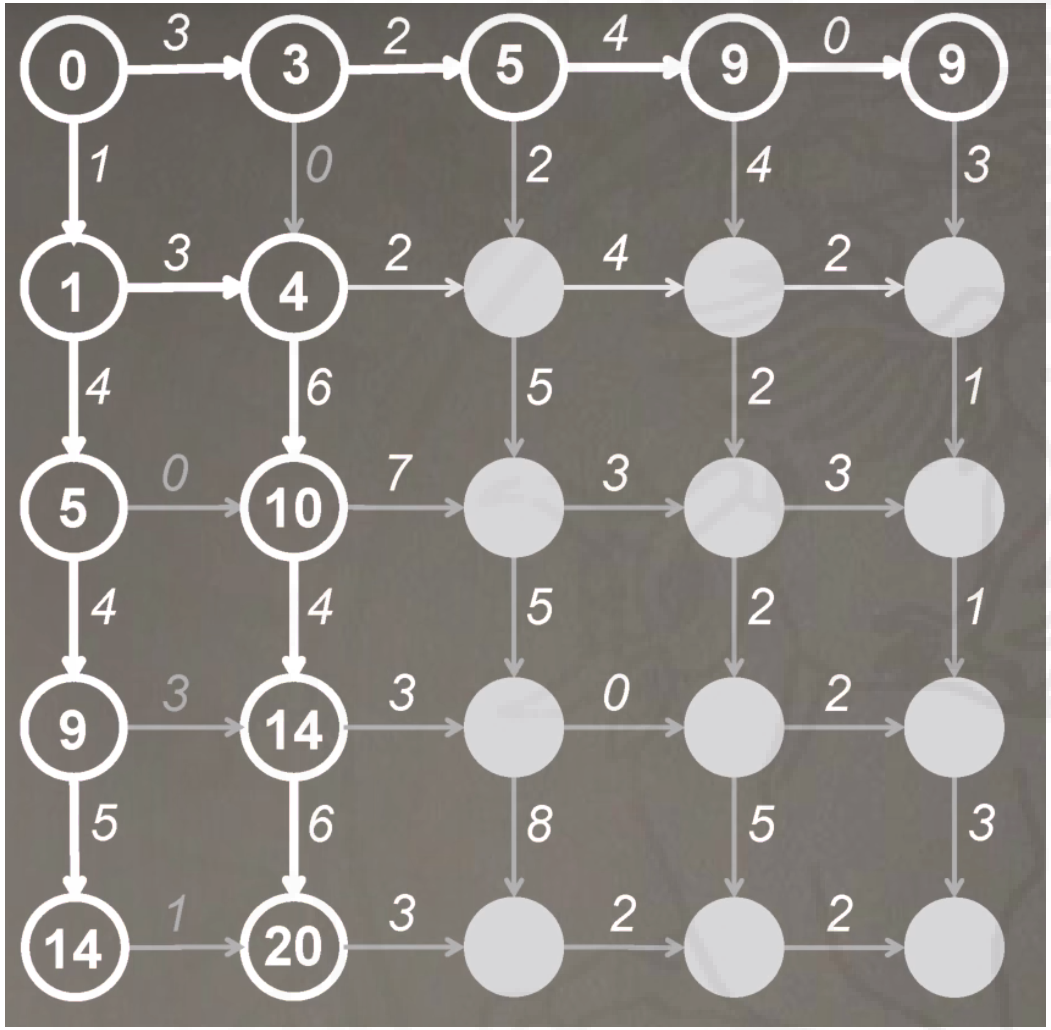
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



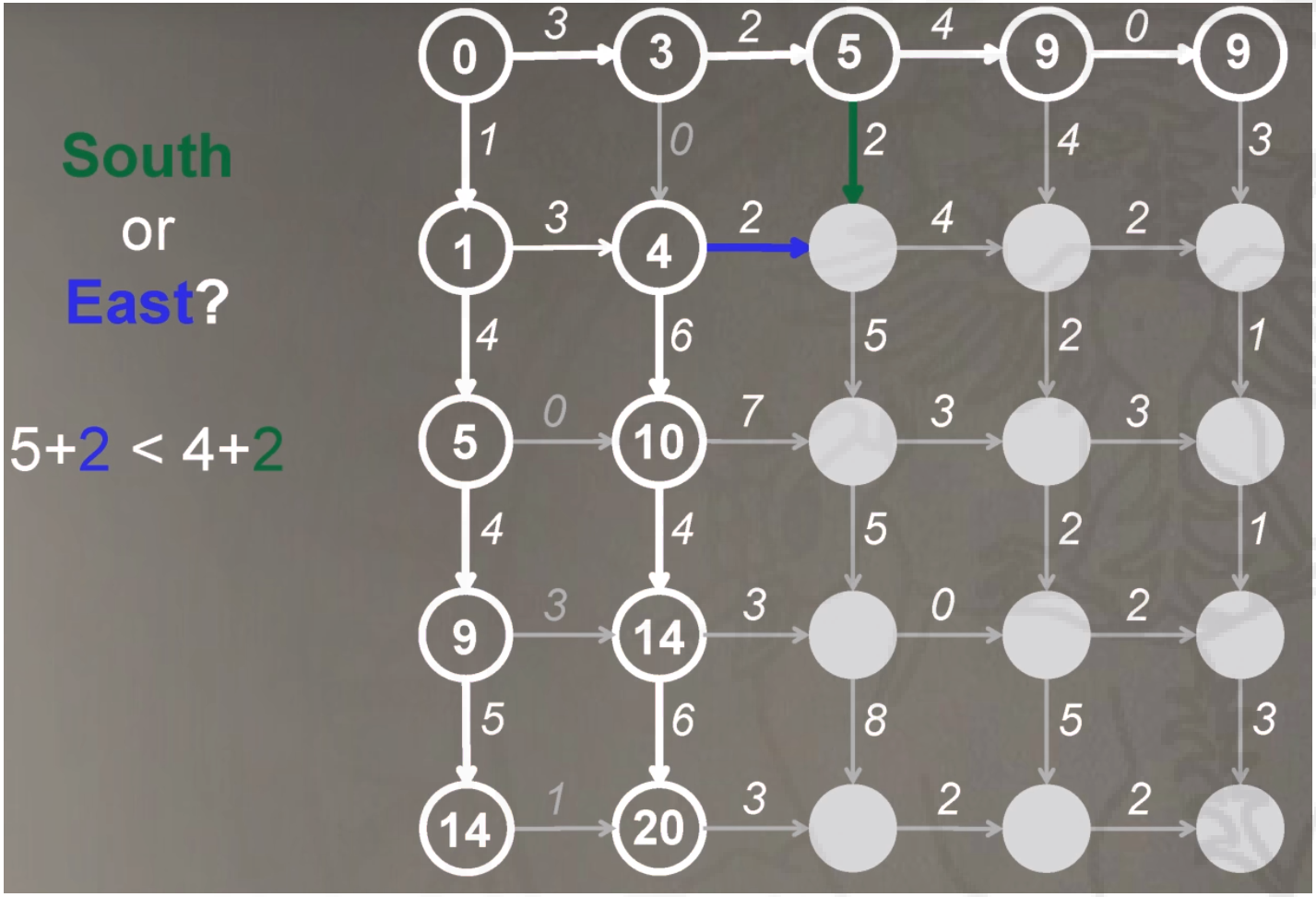
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



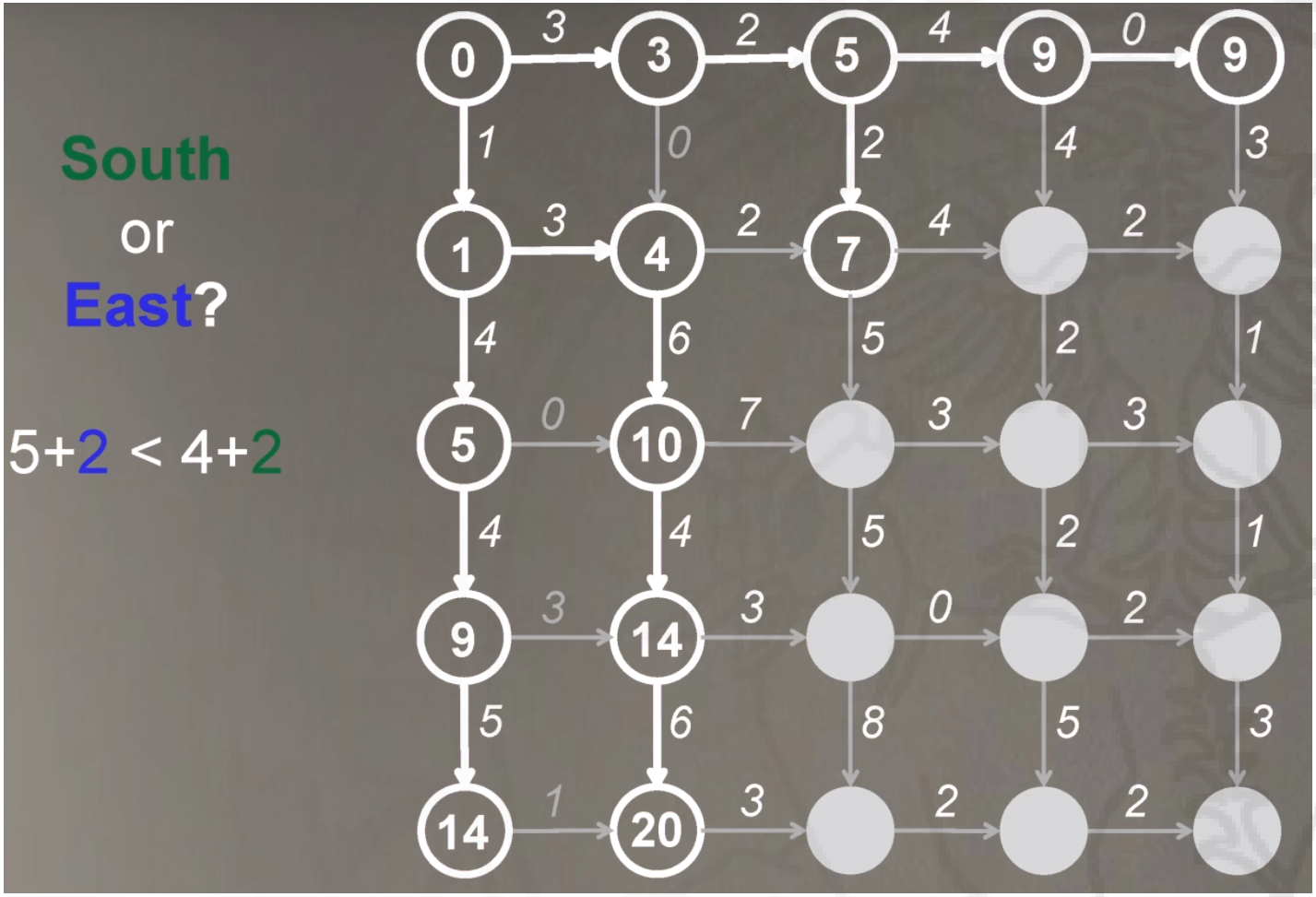
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



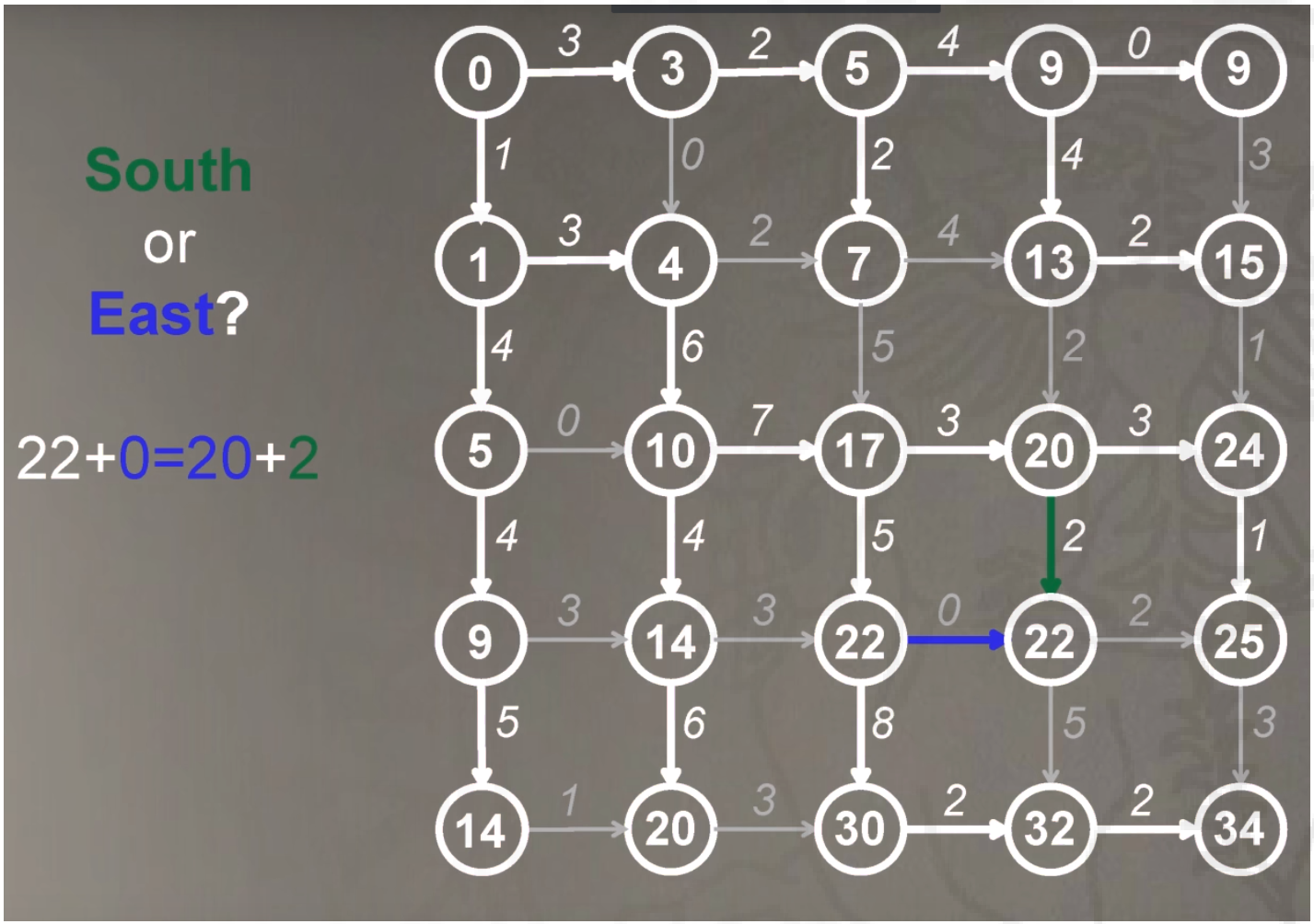
Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



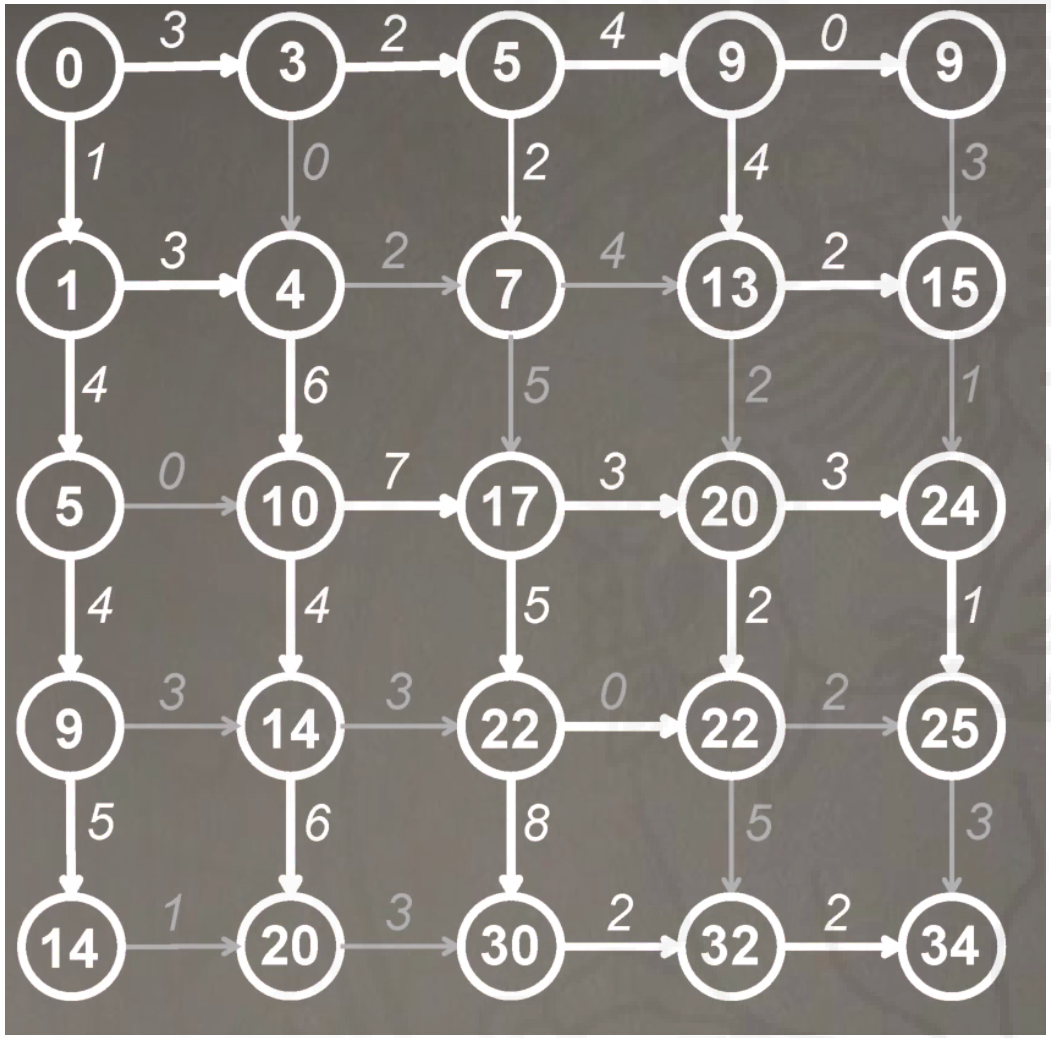
Dynamic Programming and Backtracking Pointers

- We can have cases in which both choices provide a similar results;



Dynamic Programming and Backtracking Pointers

- Now, we exploit Dynamic Programming to compute Longest Path in grid;



Dynamic Programming and Backtracking Pointers

- Now, we can define the Dynamic Programming Recurrence as follows:

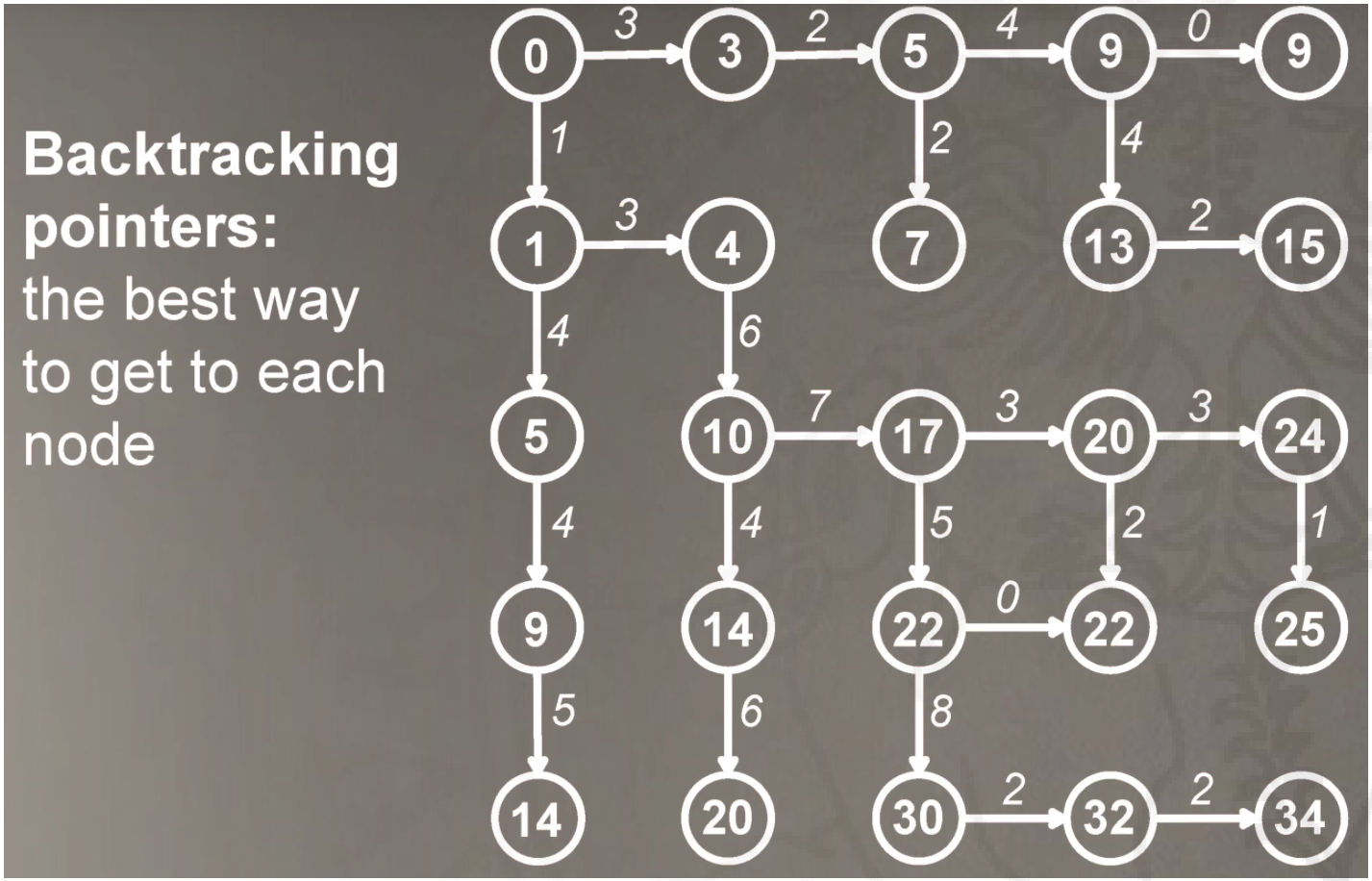
$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \textit{weight of edge } \text{“}\downarrow\text{“ } \textit{into } (i,j) \\ s_{i,j-1} + \textit{weight of edge } \text{“}\rightarrow\text{“ } \textit{into } (i,j) \end{cases}$$

where $s_{i,j}$ is the length of the longest path from $(0,0)$ to (i,j).

Dynamic Programming and Backtracking Pointers

- Backtracking pointer can be defined as:

the best way to get to each node



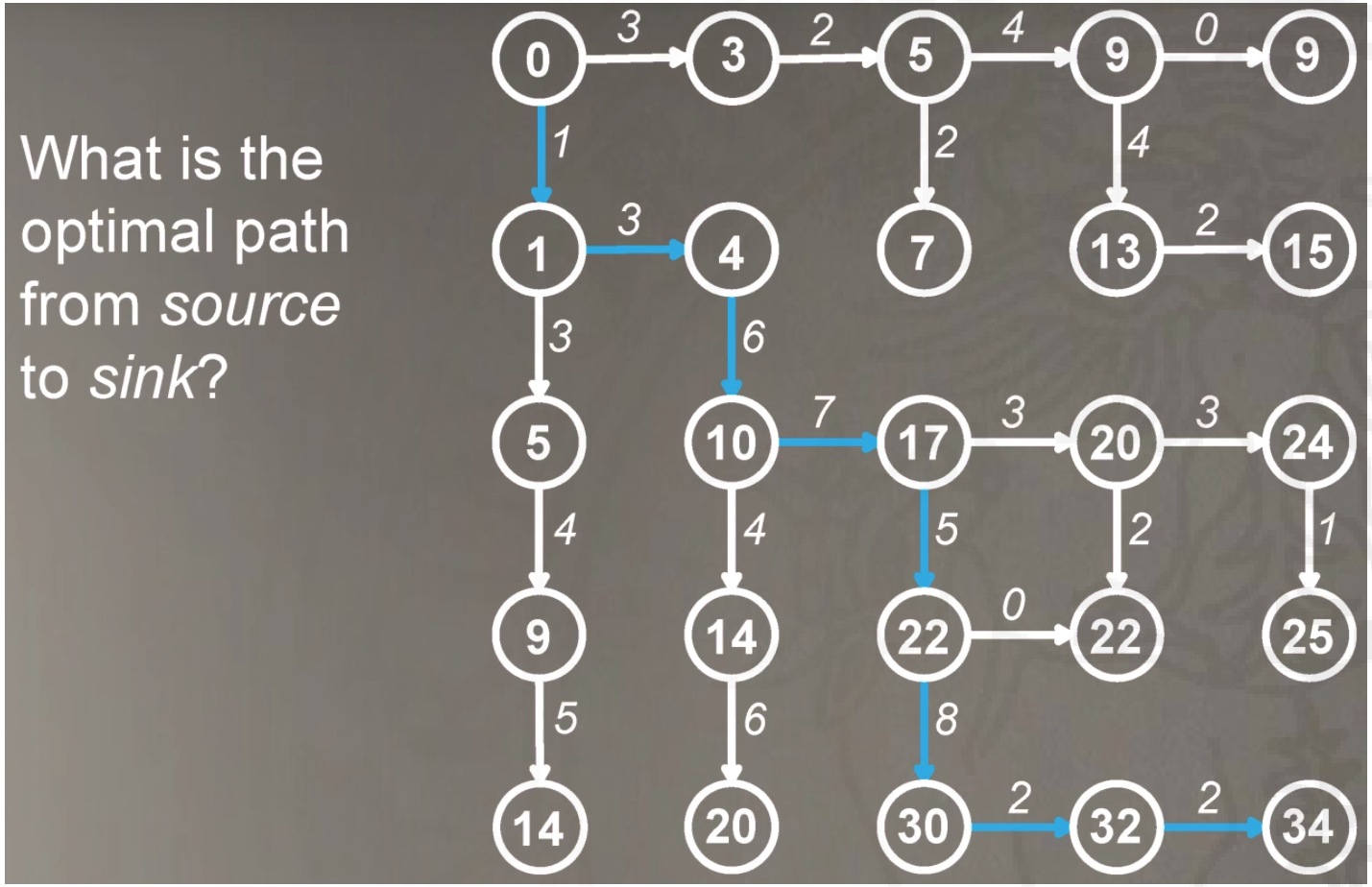
- the value in the sink node is the ***value of the longest path***.

Dynamic Programming and Backtracking Pointers

How can we derive the optimal path from source to sink?

Dynamic Programming and Backtracking Pointers

How can we derive the optimal path from source to sink?

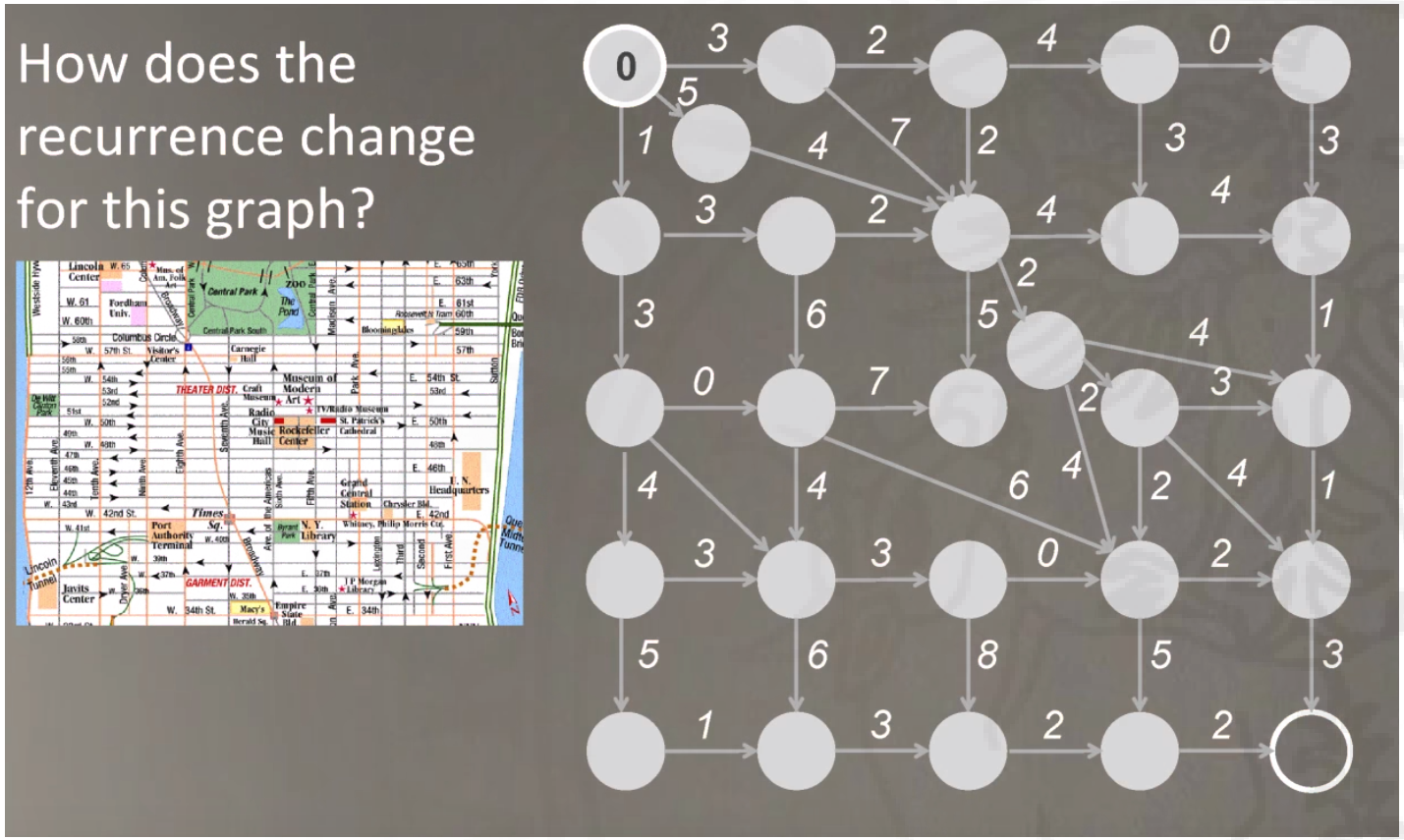


Part 2

From Manhattan to Alignment Graph

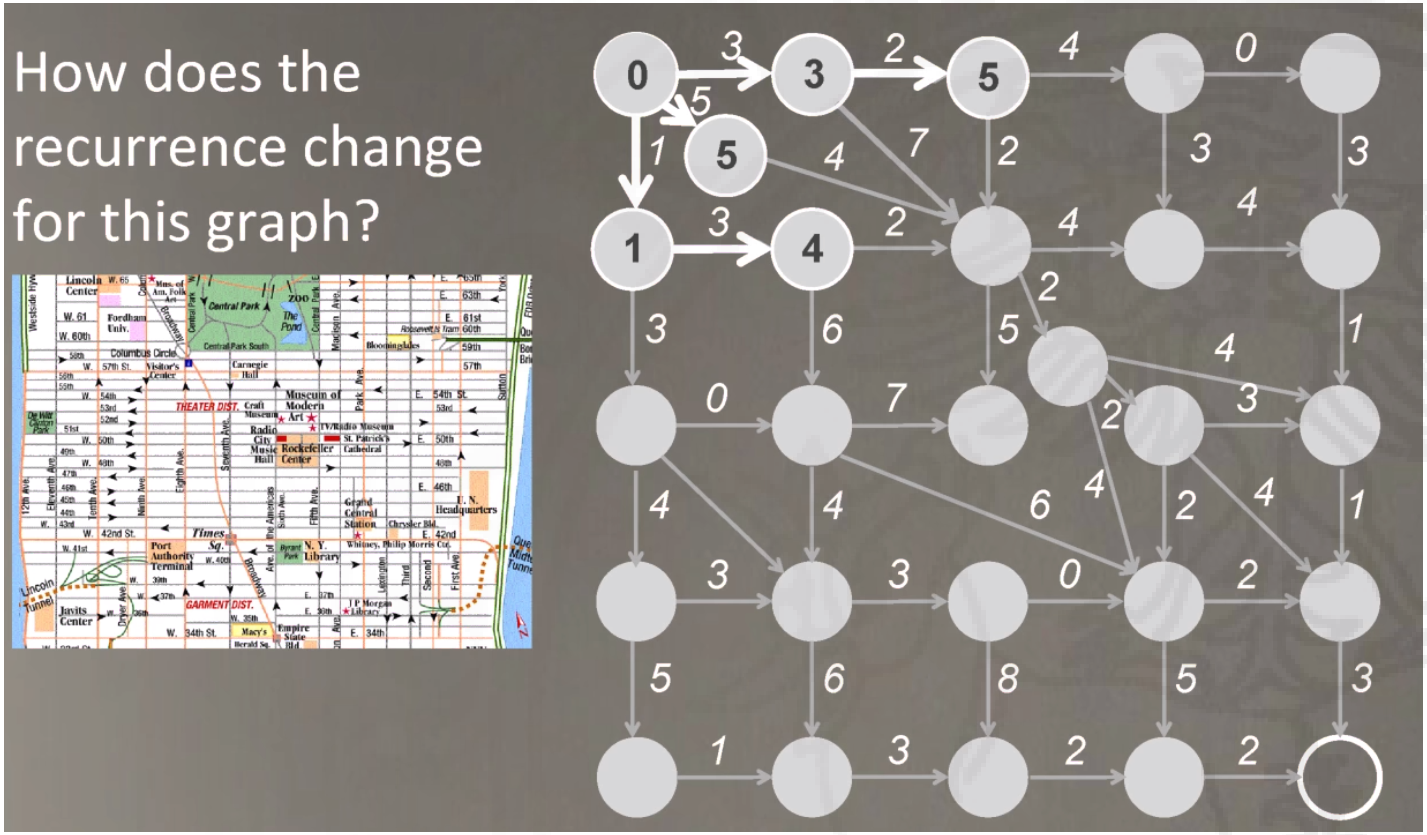
From Manhattan to Alignment Graph

- Extending the approach for DAGs;



From Manhattan to Alignment Graph

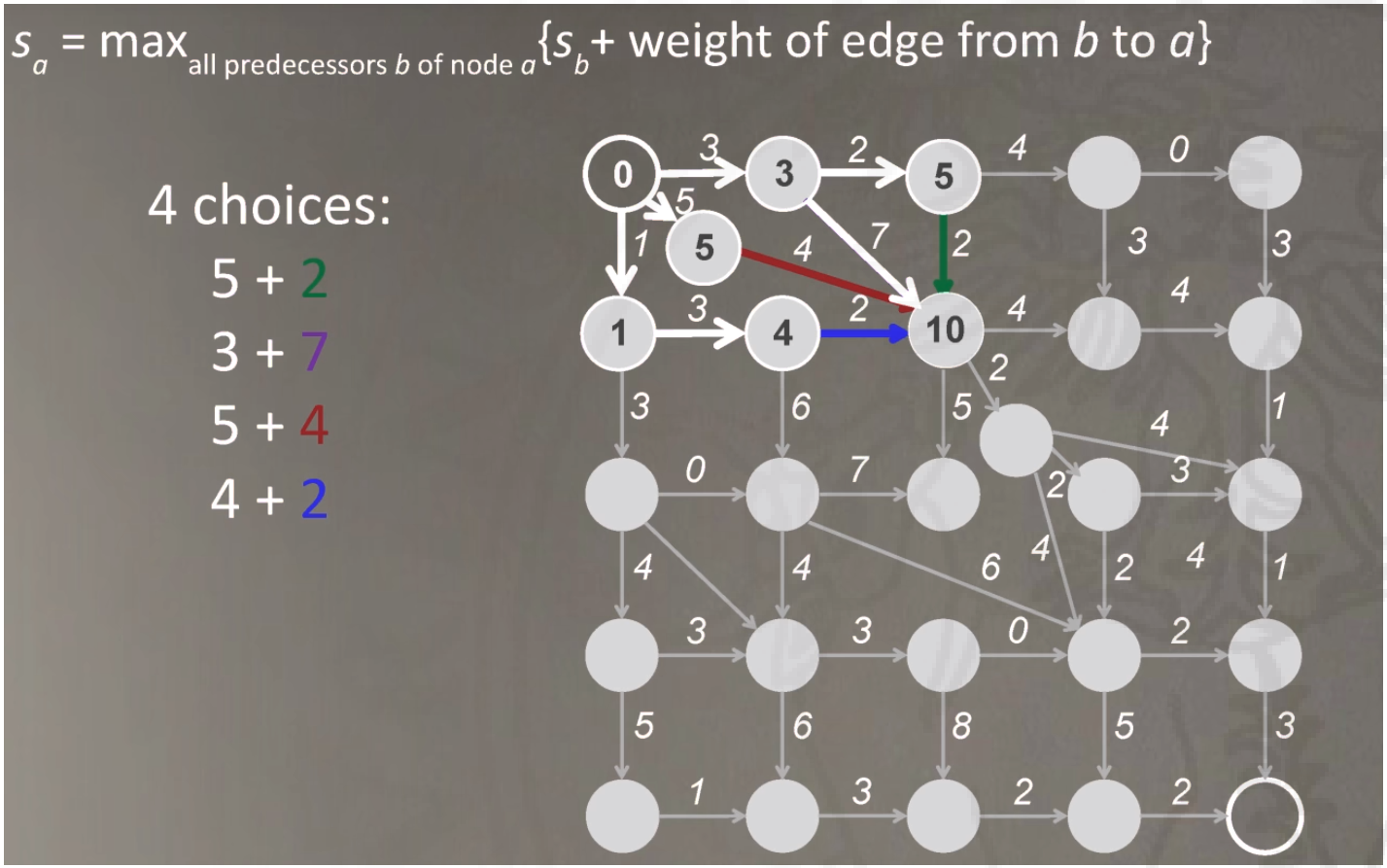
- Extending the approach for DAGs;



- all the highlighted nodes can be reached in only one-way.

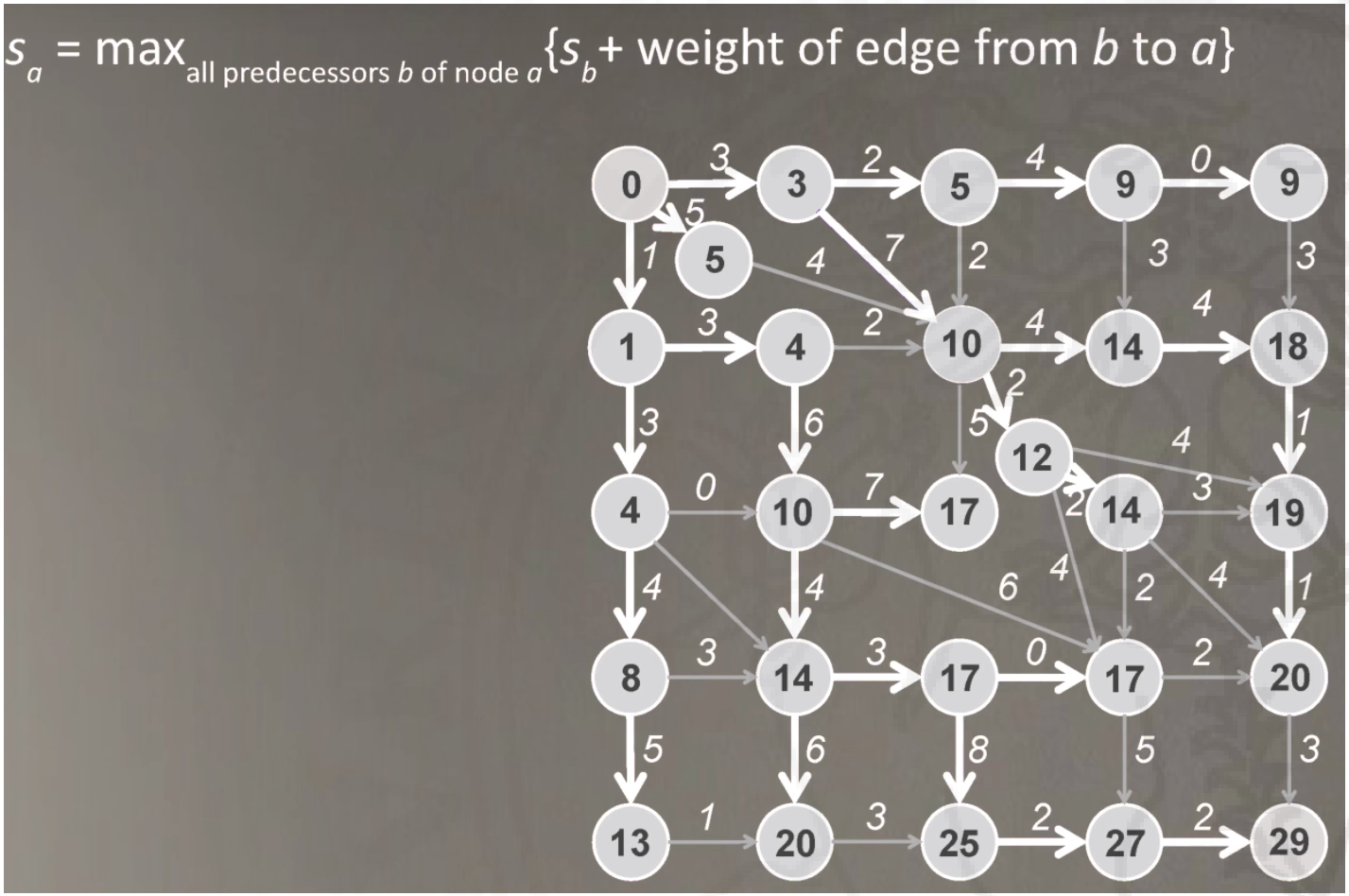
From Manhattan to Alignment Graph

- Extending the approach for DAGs;



From Manhattan to Alignment Graph

- Extending the approach for DAGs;

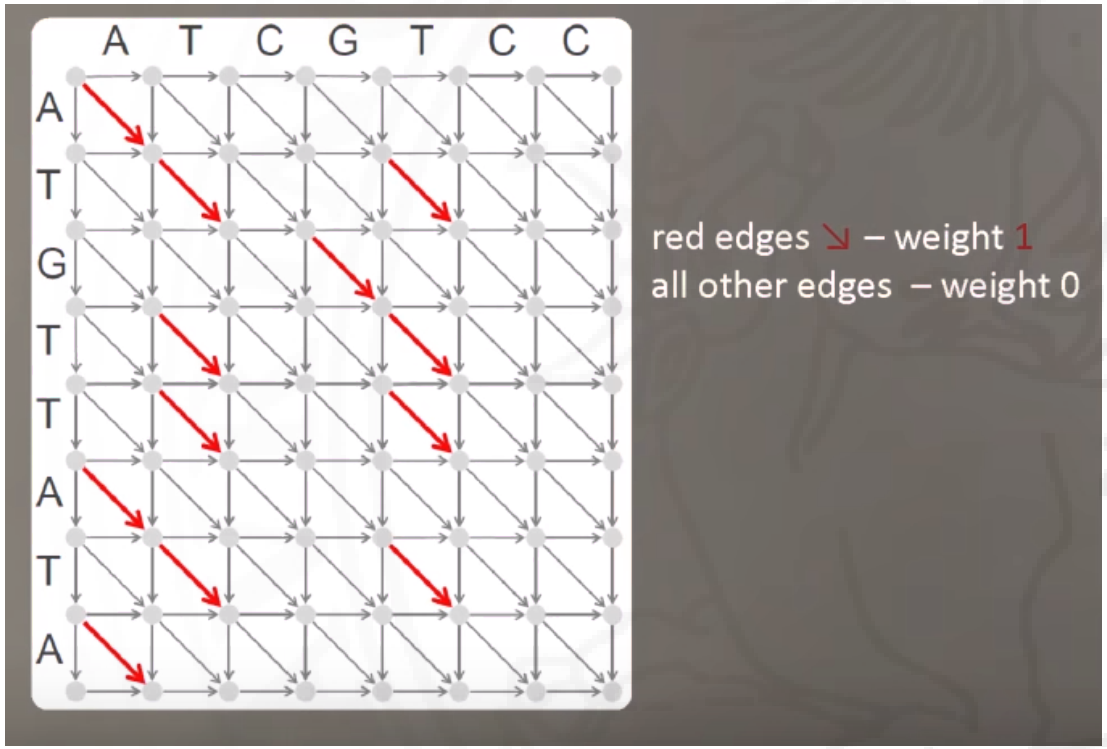


From Manhattan to Alignment Graph

- Dynamic Programming Recurrence for Alignment Graph

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \textit{weight of edge } \text{“}\downarrow\text{“ } \textit{into } (i,j) \\ s_{i,j-1} + \textit{weight of edge } \text{“}\rightarrow\text{“ } \textit{into } (i,j) \\ s_{i-1,j-1} + \textit{weight of edge } \text{“}\searrow\text{“ } \textit{into } (i,j) \end{cases}$$

where $s_{i,j}$ is the length of the longest path from $(0,0)$ to (i,j).

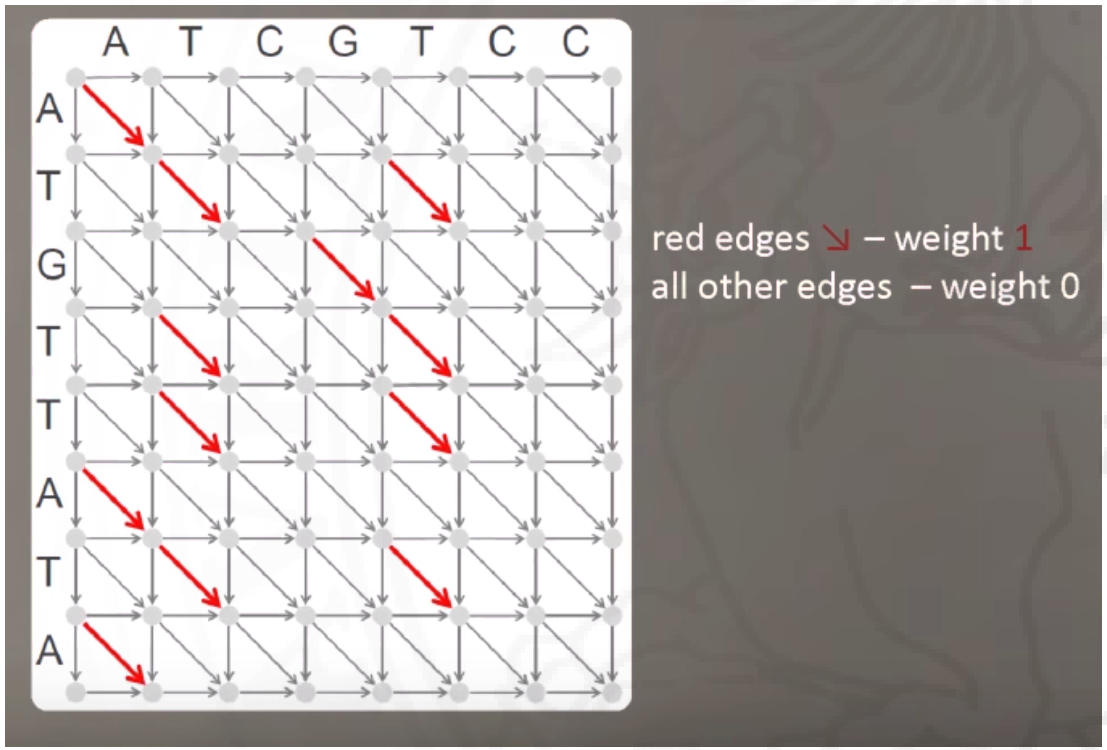


From Manhattan to Alignment Graph

- Dynamic Programming Recurrence for Longest Subsequence Problem

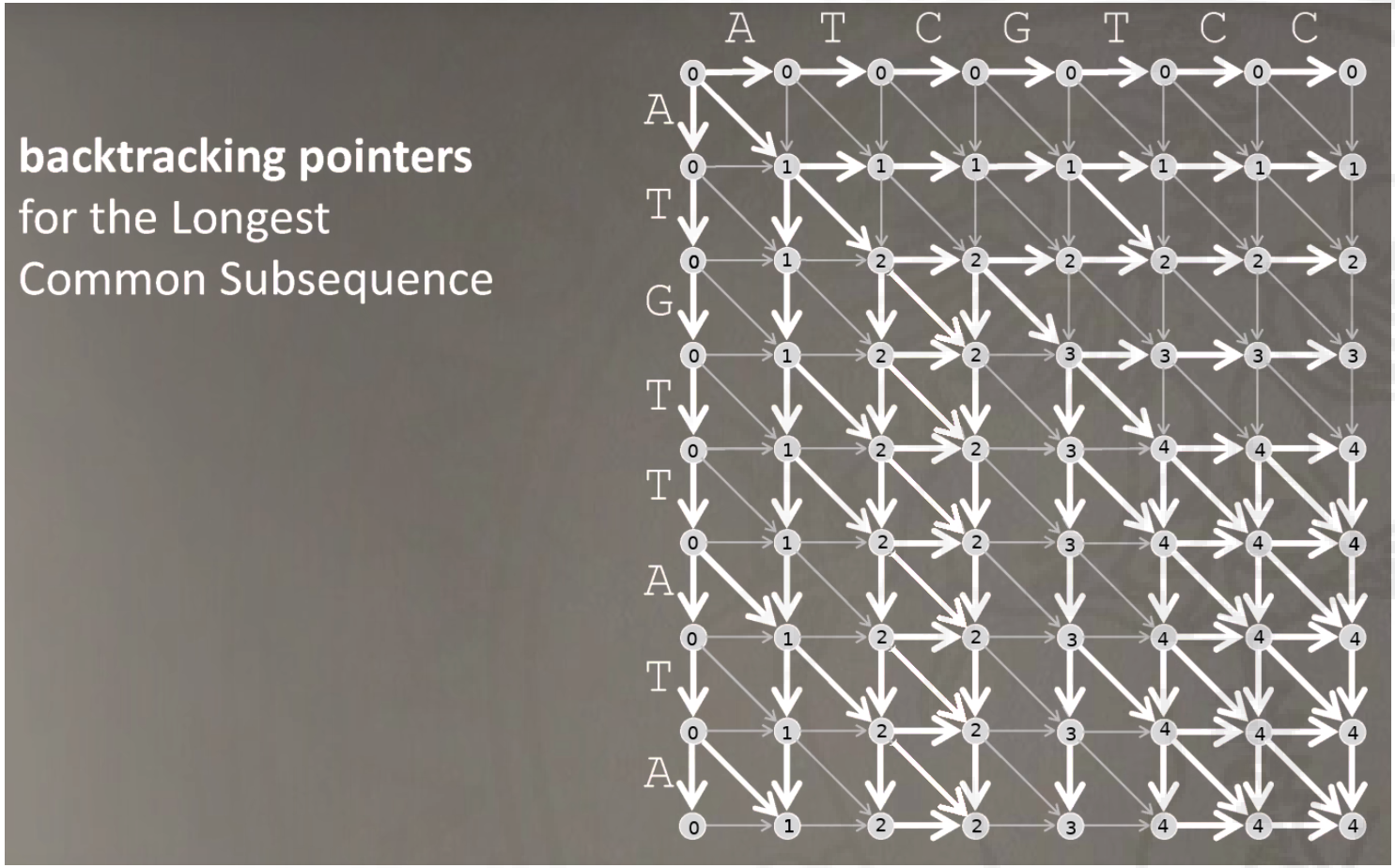
$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 & \\ s_{i,j-1} + 0 & \\ s_{i-1,j-1} + 0 & iff\ V_i \neq W_i \\ s_{i-1,j-1} + 1 & iff\ V_i = W_i \end{cases}$$

where $s_{i,j}$ is the length of the longest path from $(0,0)$ to (i,j).



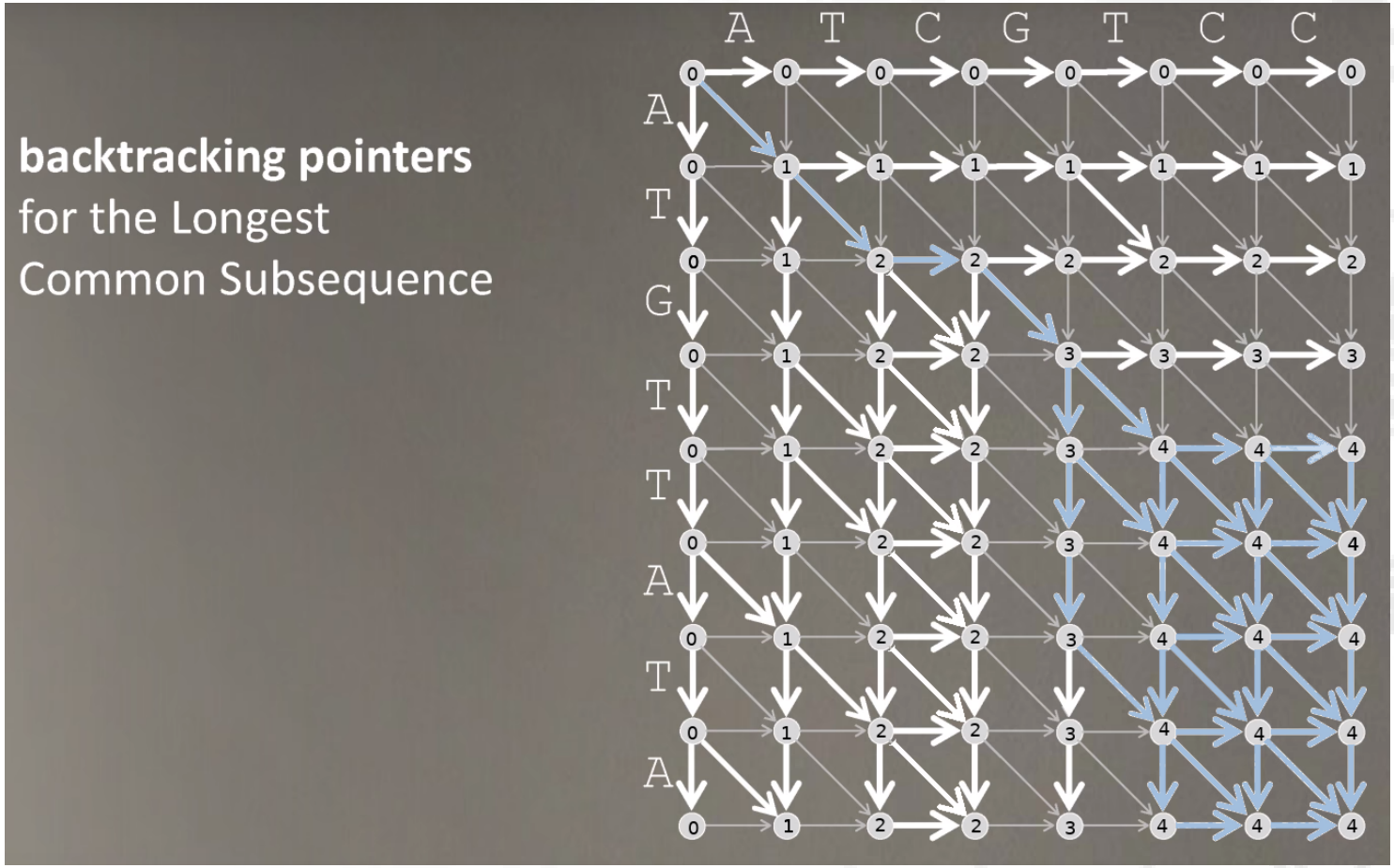
From Manhattan to Alignment Graph

- Dynamic Programming Recurrence for Longest Subsequence Problem



From Manhattan to Alignment Graph

- Dynamic Programming Recurrence for Longest Subsequence Problem



- Blue arrows show the set of optimal alignments

Part 2
Penalizing Insertions and Deletions in Sequence Alignment

Penalizing Insertions and Deletions in Sequence Alignment

- it is not difficult to construct an alignment having a lot of matches at the expense of introducing more indels, ...
- but more indels we add, then less biologically relevant the alignment becomes;

How can we cope with this point?

Penalizing Insertions and Deletions in Sequence Alignment

How can we cope with this point?

- we can extend our score function

$$\|matches\|$$

into

$$\|matches\| - \mu\|mismatches\| - \sigma\|indels\|$$

so that we take into account penalties associated with mismatches and indels

```
A T - G T T A T A
A T C G T - C - C
+1+1-2+1+1-2-3-2-3=-7
```

Penalizing Insertions and Deletions in Sequence Alignment

- we can define a scoring matrix as follows:

	A	C	G	T	–
A	$+1$	$-\mu$	$-\mu$	$-\mu$	$-\sigma$
C	$-\mu$	$+1$	$-\mu$	$-\mu$	$-\sigma$
G	$-\mu$	$-\mu$	$+1$	$-\mu$	$-\sigma$
T	$-\mu$	$-\mu$	$-\mu$	$+1$	$-\sigma$
–	$-\sigma$	$-\sigma$	$-\sigma$	$-\sigma$	

scoring matrix

	A	C	G	T	–
A	+1	−3	−5	−1	−3
C	−4	+1	−3	−2	−3
G	−9	−7	+1	−1	−3
T	−3	−5	−8	+1	−4
–	−4	−2	−2	−1	

scoring matrix with arbitrary values

Penalizing Insertions and Deletions in Sequence Alignment

- a strong effort was spent to define the most appropriate scores;
- for instance ....

Scoring Matrices for Amino Acid Sequences

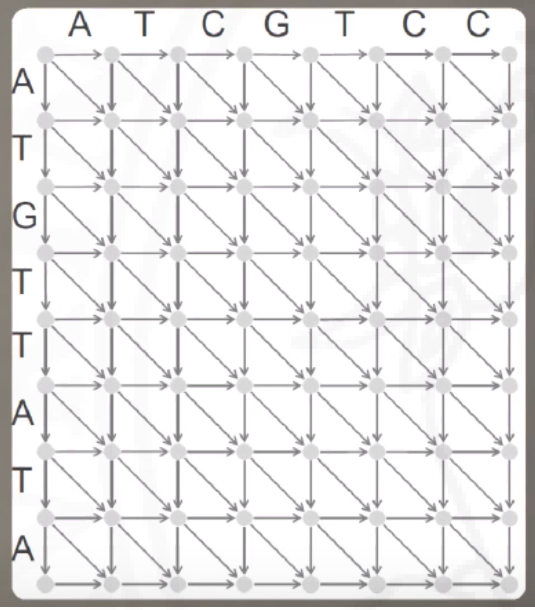
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C Cys	12																			
S Ser	0	2																		
T Thr	-2	1	3																	
P Pro	-3	1	0	6																
A Ala	-2	1	1	1	2															
G Gly	-3	1	0	-1	1	5														
N Asn	-4	1	0	-1	0	0	2													
D Asp	-5	0	0	-1	0	1	2	4												
E Glu	-5	0	0	-1	0	0	1	3	4											
Q Gln	-5	-1	-1	0	0	-1	1	2	2	4										
H His	-3	-1	-1	0	-1	-2	2	1	1	3	6									
R Arg	-4	0	-1	0	-2	-3	0	-1	-1	1	2	6								
K Lys	-5	0	0	-1	-1	-2	1	0	0	1	0	3	5							
M Met	-5	-2	-1	-2	-1	-3	-2	-3	-2	-1	-2	0	0	6						
I Ile	-2	-1	0	-2	-1	-3	-2	-2	-2	-2	-2	-2	-2	2	5					
L Leu	-6	[illegible]	[illegible]	-3	-2	-4	-3	-4	-3	-2	-2	-3	-3	4	2	6				
V Val	-2	-1	0	-1	0	-1	-2	-2	-2	-2	-2	-2	-2	2	4	2	4			
F Phe	-4	-3	-3	-5	-5	-5	-4	-6	-5	-5	-2	-4	-5	0	1	2	-1	9		
Y Tyr	0	-3	-3	-5	-3	-5	-2	-4	-4	-4	0	-4	-4	-2	-1	-1	-2	7	10	
W Trp	-8	-2	-5	-6	-6	-7	-4	-7	-7	-5	-3	2	-3	-4	-5	-2	-6	0	0	17

Y often mutates into F (score +7)
but rarely mutates into P (score -5)

Penalizing Insertions and Deletions in Sequence Alignment

- How to change the dynamic programming recurrence:

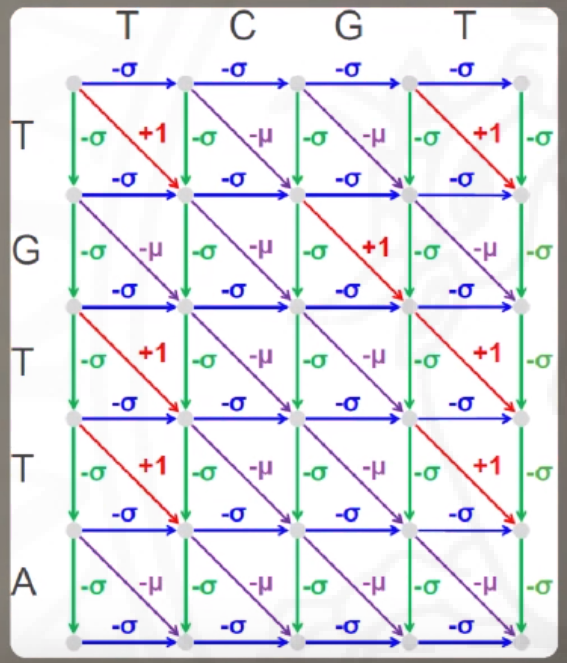
$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge "}\downarrow\text{" into } (i,j) \\ s_{i,j-1} + \text{weight of edge "}\rightarrow\text{" into } (i,j) \\ s_{i-1,j-1} + \text{weight of edge "}\searrow\text{" into } (i,j) \end{cases}$$



Penalizing Insertions and Deletions in Sequence Alignment

- How to change the dynamic programming recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \quad \text{(match)} \\ s_{i-1,j-1} - \mu, \text{ if } v_i \neq w_j \quad \text{(mismatch)} \end{cases}$$



Penalizing Insertions and Deletions in Sequence Alignment

- How to change the dynamic programming recurrence in general way:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + score(v_i, -) \\ s_{i,j-1} + score(-, w_j) \\ s_{i-1,j-1} + score(v_i, w_j) \end{cases}$$