nodes : 0, 1, 2, 3, 4

best path: 0 → 4 → 3 → 1 → 2 → 0 ...

path = [4, 2, 0, 1, 3]

Lettura soluzione:

Start: 0 — next = path[0] = 4

↓ 4 — next = path[4] = 3

↓ 3 — next = path[3] = 1

↓ 1 — next = path[1] = 2

↓ 2 — next = path[2] = 0

↓ 0

Trovo soluzione: path = [-1, -1, -1, -1, -1]

→ start: 0 — nodo + vicino: 4

↓ path = [4, -1, -1, -1, -1]

→ node = 4 — nodo + vicino: 3

↓ path = [4, -1, -1, -1, 3]

→ node = 3 — nodo + vicino: 1

↓ path = [4, -1, -1, 1, 3]

⋮

Vantaggi: se path[i] == -1, allora il nodo i non è stato visitato

⇓

risparmio memoria rispetto ad avere una lista che conta se il nodo i è stato visitato

Trovo nodo + vicino:

Costruisco la matrice min_edges:

↳ Precomputed

0	
1	
⋮	
i	k_1 k_2 k_3 ....
⋮	
nnodes - 1	

→ $k_1 = \text{argmin}_j \{c_{ij}\}$, k_2 il 2° più piccolo, ecc ...

$k_1 = i \ \forall i$ visto che il nodo + vicino a 'i' è 'i' ⇒ posso eliminare la prima riga

Vantaggi: find_min() trova il nodo più vicino a i, pertanto mi basta leggere il primo elemento della riga i $\Rightarrow O(1)$

Se il primo elemento è già nel path guardo il successivo, e così via.

Non dovrò mai leggere tutta la riga data che spostandomi i nodi più vicini saranno diversi

Sono costretto a leggere tutta la riga solo all'ultima iterazione di un caso limite.

Con questo metodo esploro pochi nodi, contro a $\Omega(n)$ se devo trovare il minimo ogni volta

Test: su un'istanza da 500 nodi c'è un <u>guadagno di 6 secondi</u> (da 53 sec a 47 sec $\Rightarrow$ miglioramento dell'11%)