Introduction to

Algorithm Design and Analysis

[15] Path in Graph

Jingwei Xu
https://cs.nju.edu.cn/ics/people/jingweixu/index.html
Institute of Computer Software
Nanjing University

In the last class…

- **Optimization Problem**
  - Greedy strategy
- **MST Problem**
  - Prim algorithm
  - Kruskal algorithm
- **Single-Source Shortest Path Problem**
  - Dijkstra algorithm

Path in Graph

- **Single-source shortest paths (SSSP)**
  - Dijkstra algorithm by example
  - Priority queue-based implementation
  - Proof of correctness
- **All-pairs shortest paths (APSP)**
  - Shortest path and transitive closure
  - Warshall algorithm for transitive closure
    - BF1, BF2, BF3 => Warshall algorithm
    - Floyd-Warshall algorithm for shortest paths

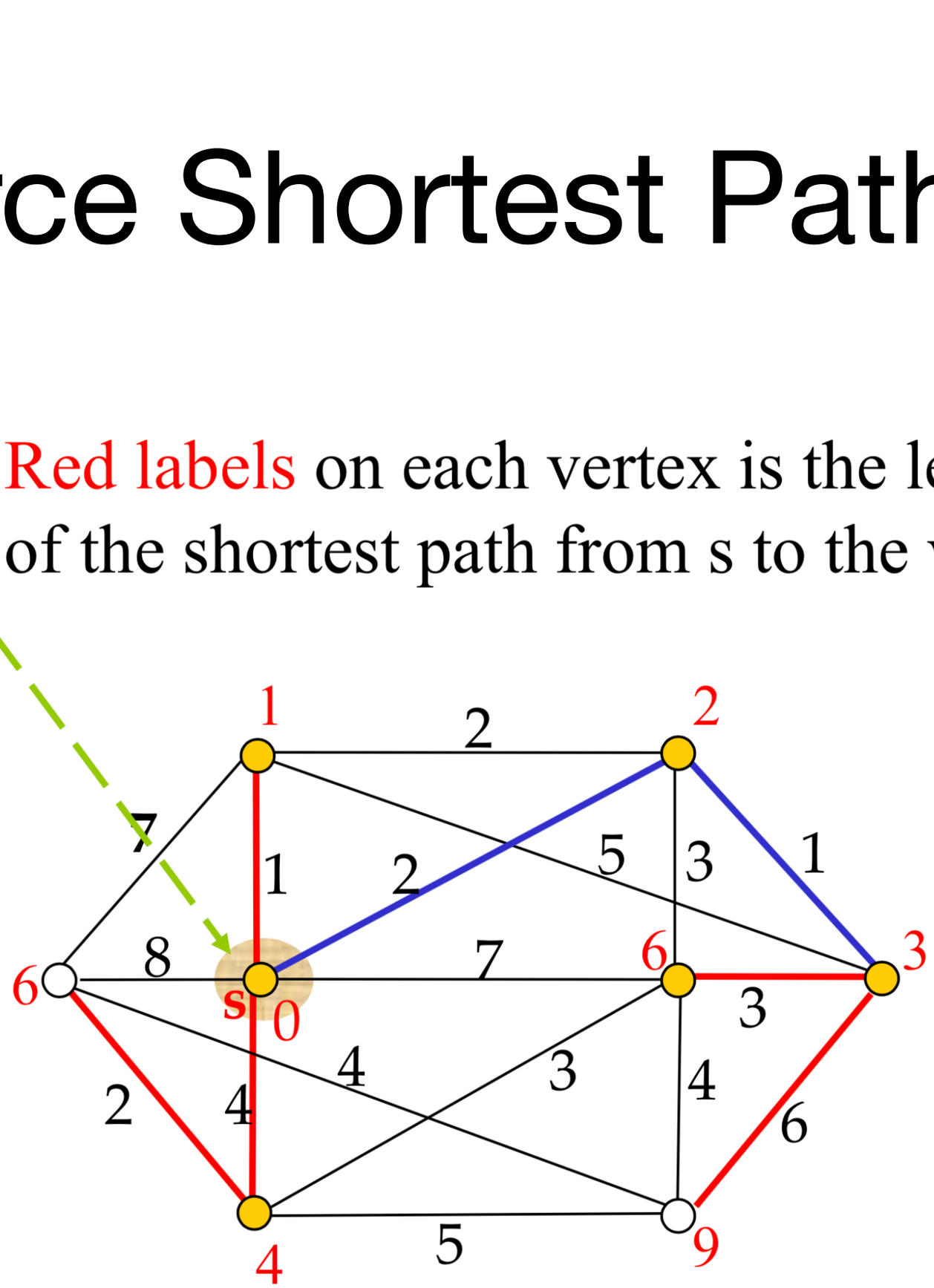
Single Source Shortest Paths

The single source

Red labels on each vertex is the length of the shortest path from s to the vertex.

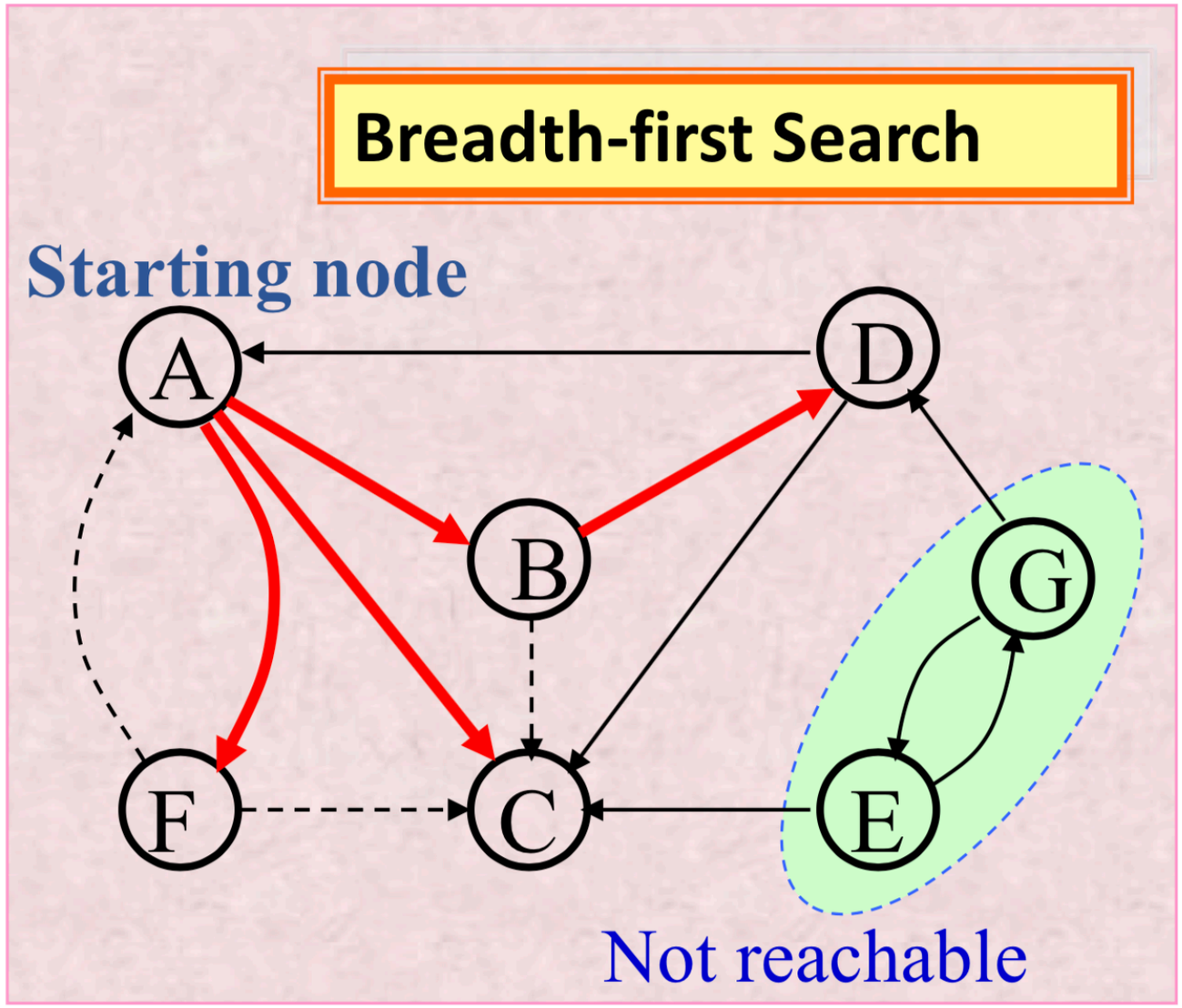
Note:

The shortest [0, 3]-path doesn't contain the shortest edge leaving s, the edge [0,1]

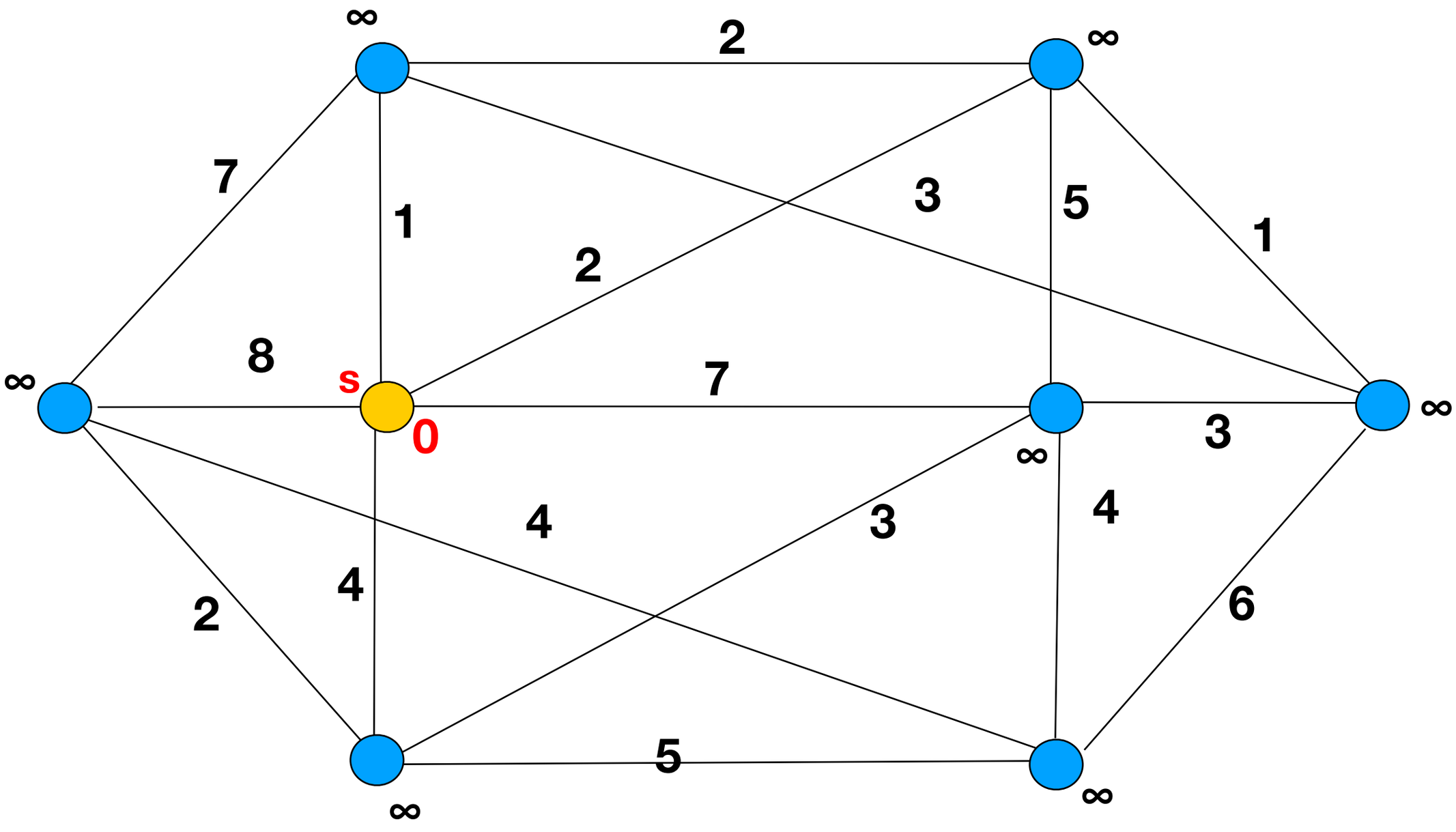


Warm Up

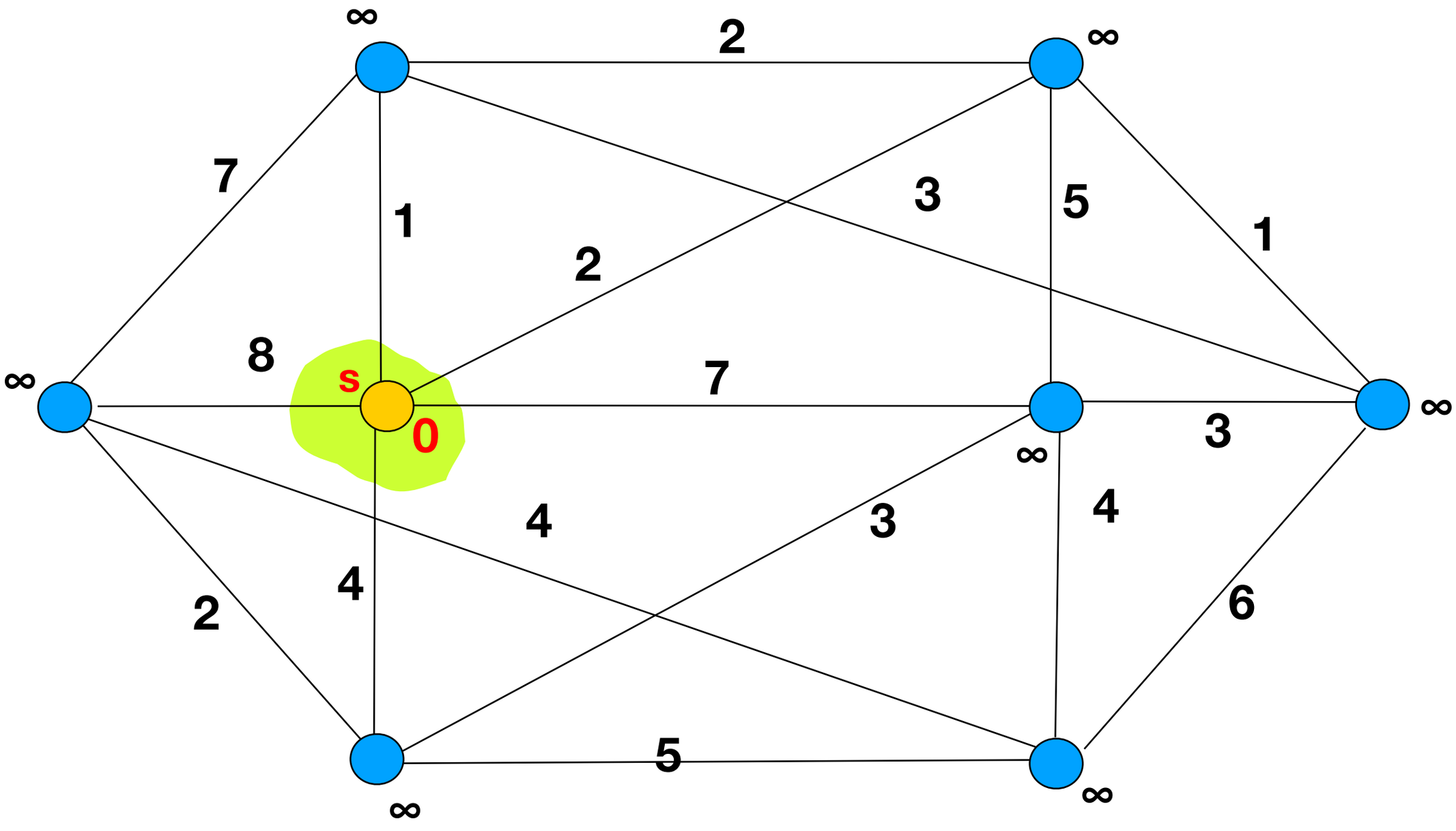
- **Single-source shortest path over uniformly weighted graph**
  - Just BFS



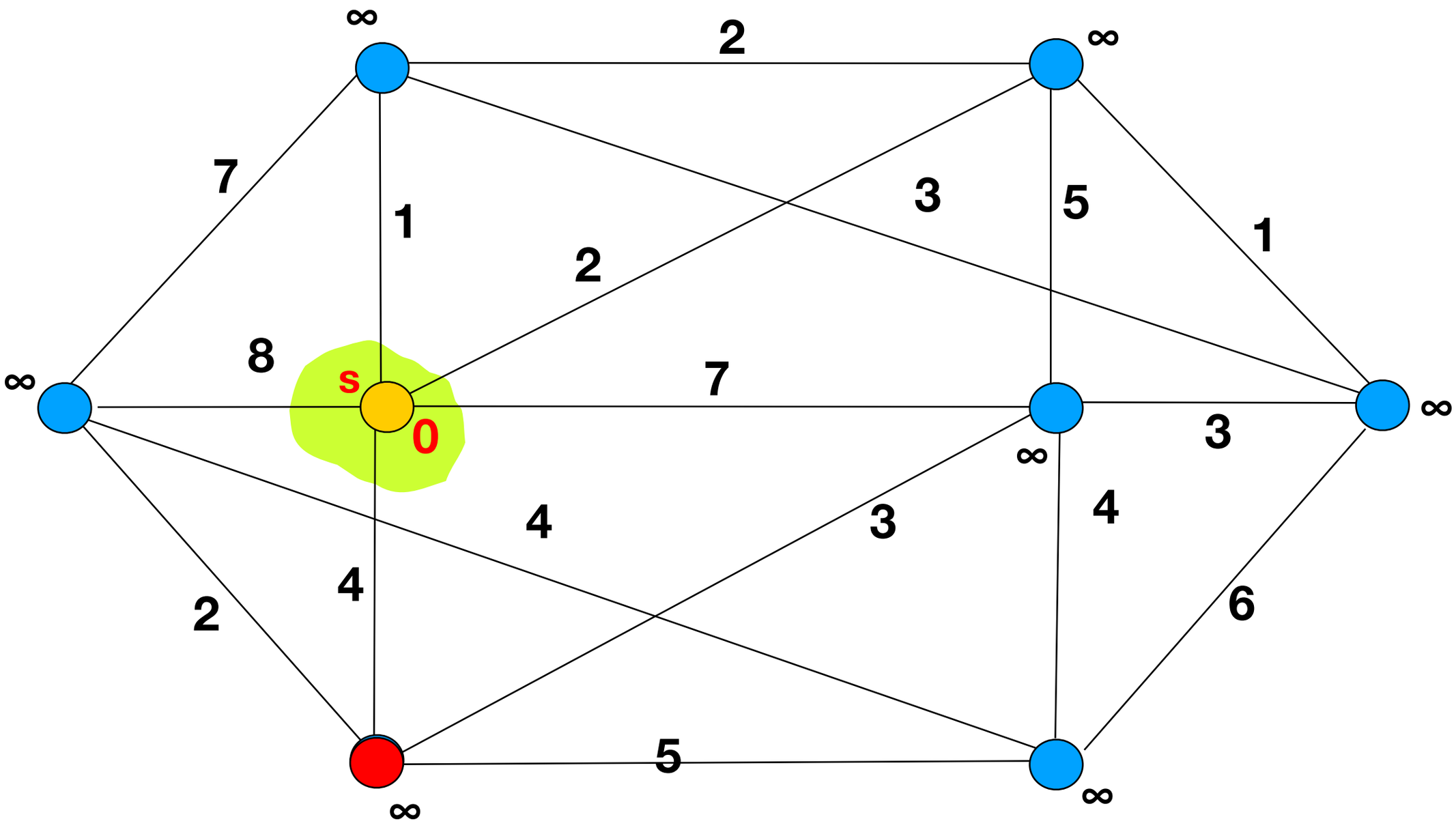
Dijkstra's Algorithm



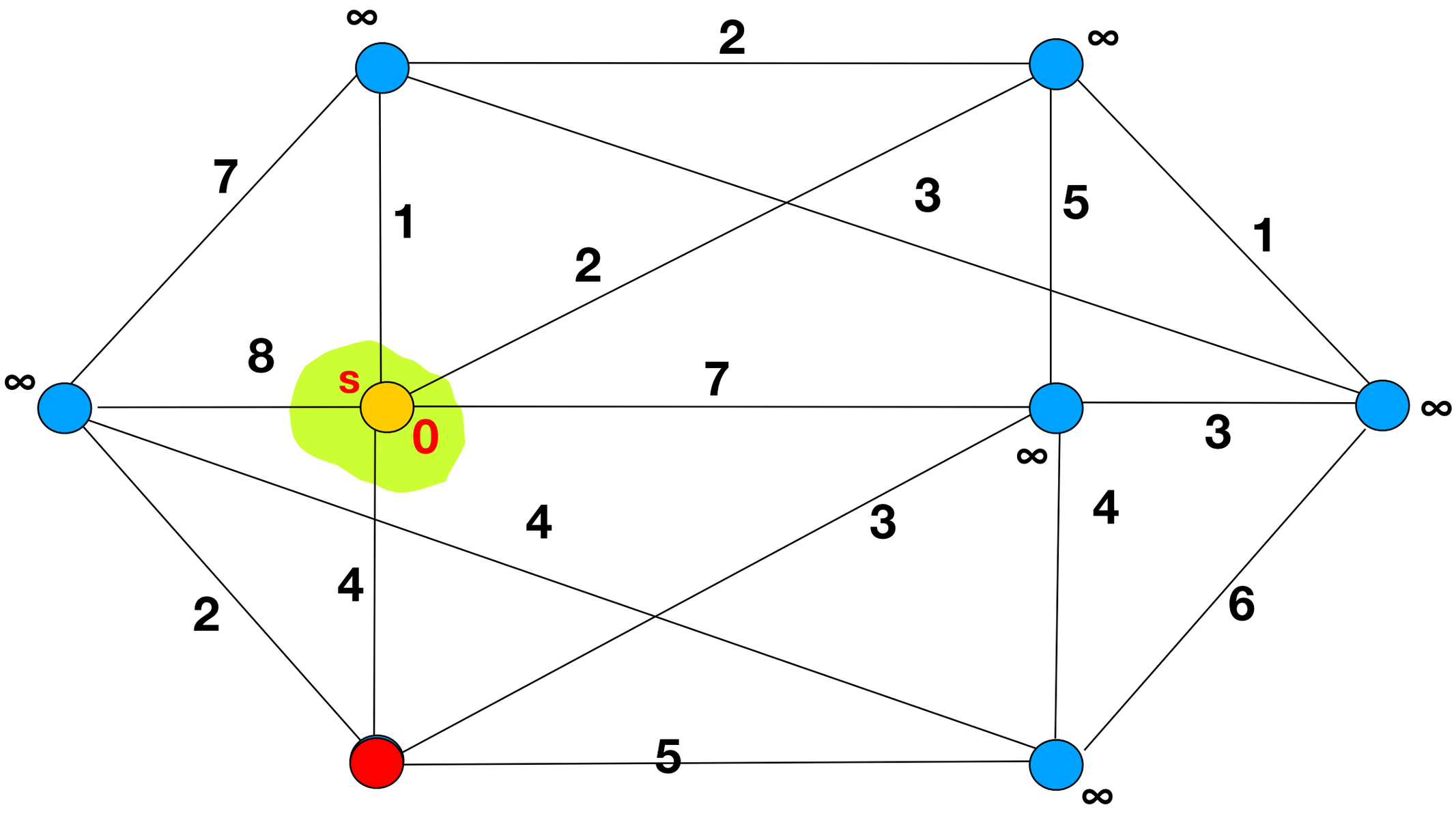
Dijkstra's Algorithm



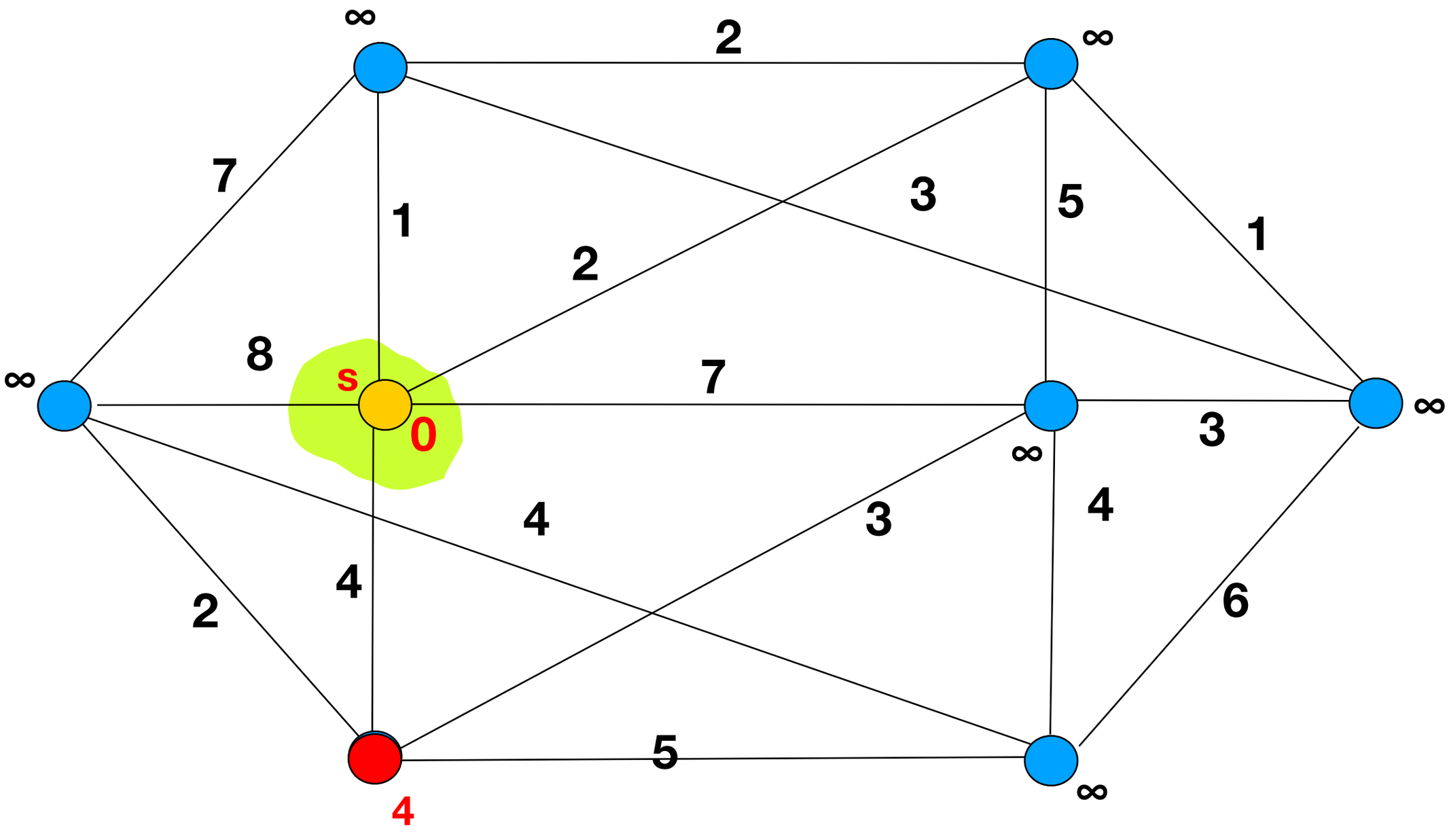
Dijkstra's Algorithm



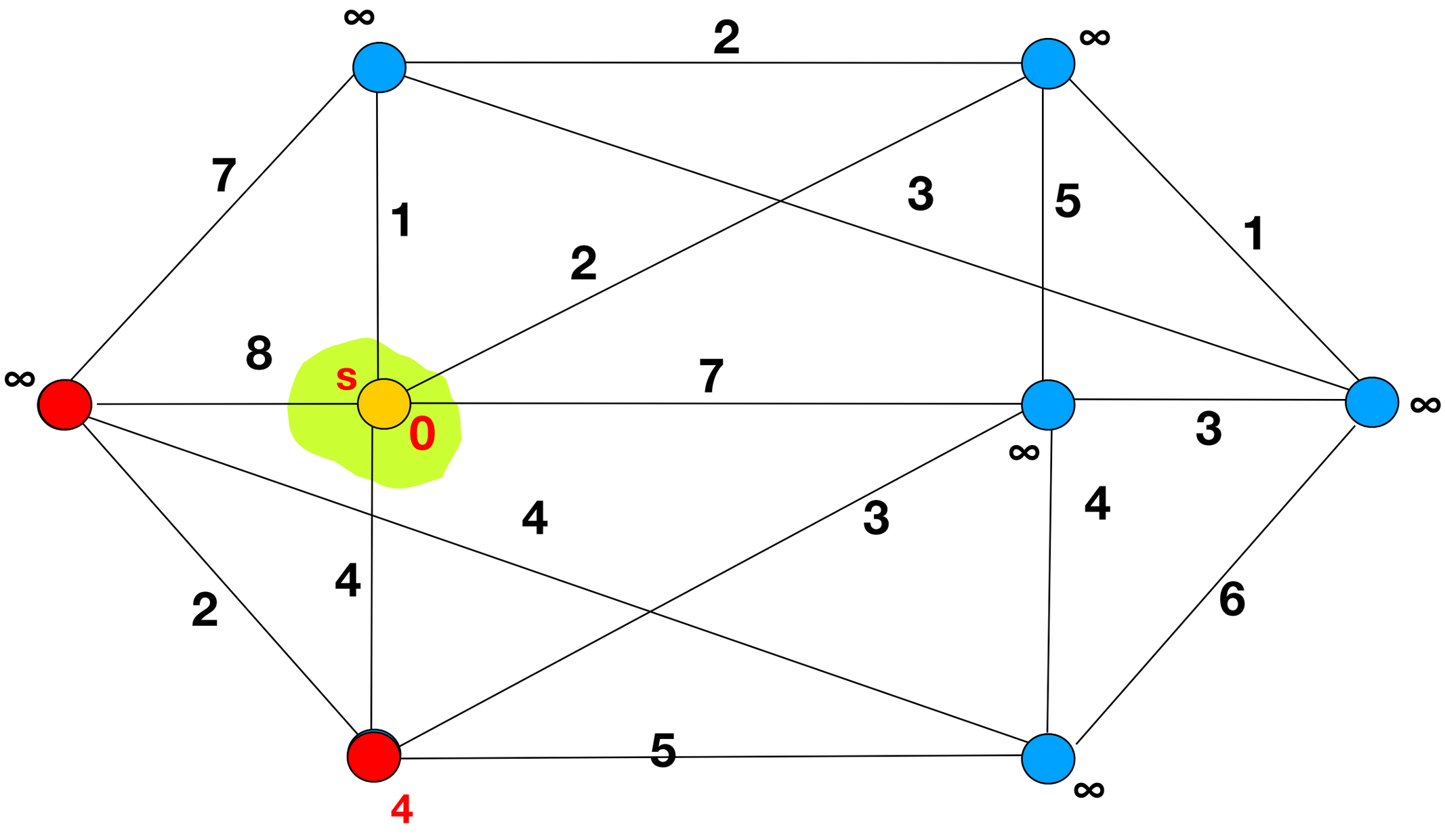
Dijkstra's Algorithm



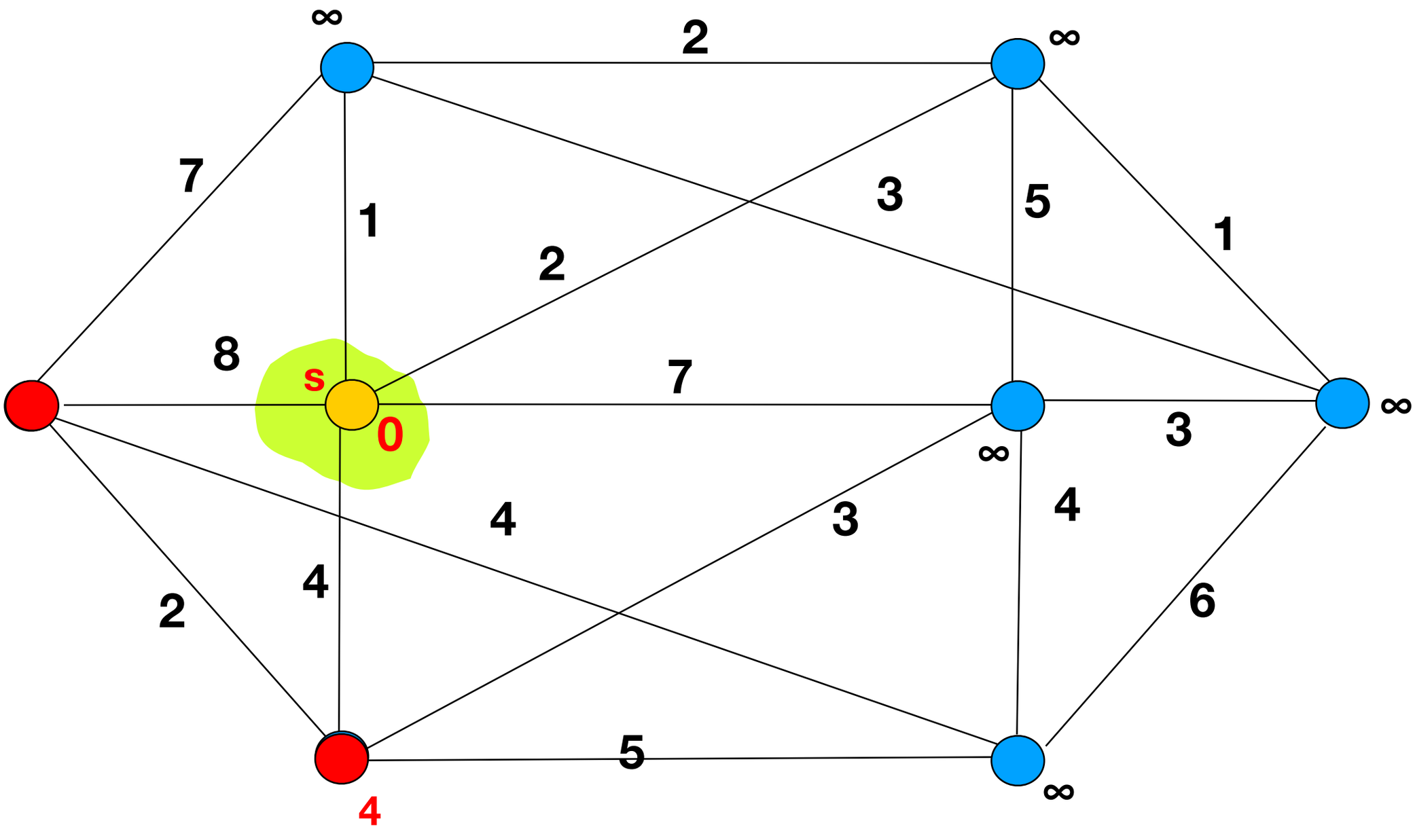
Dijkstra's Algorithm



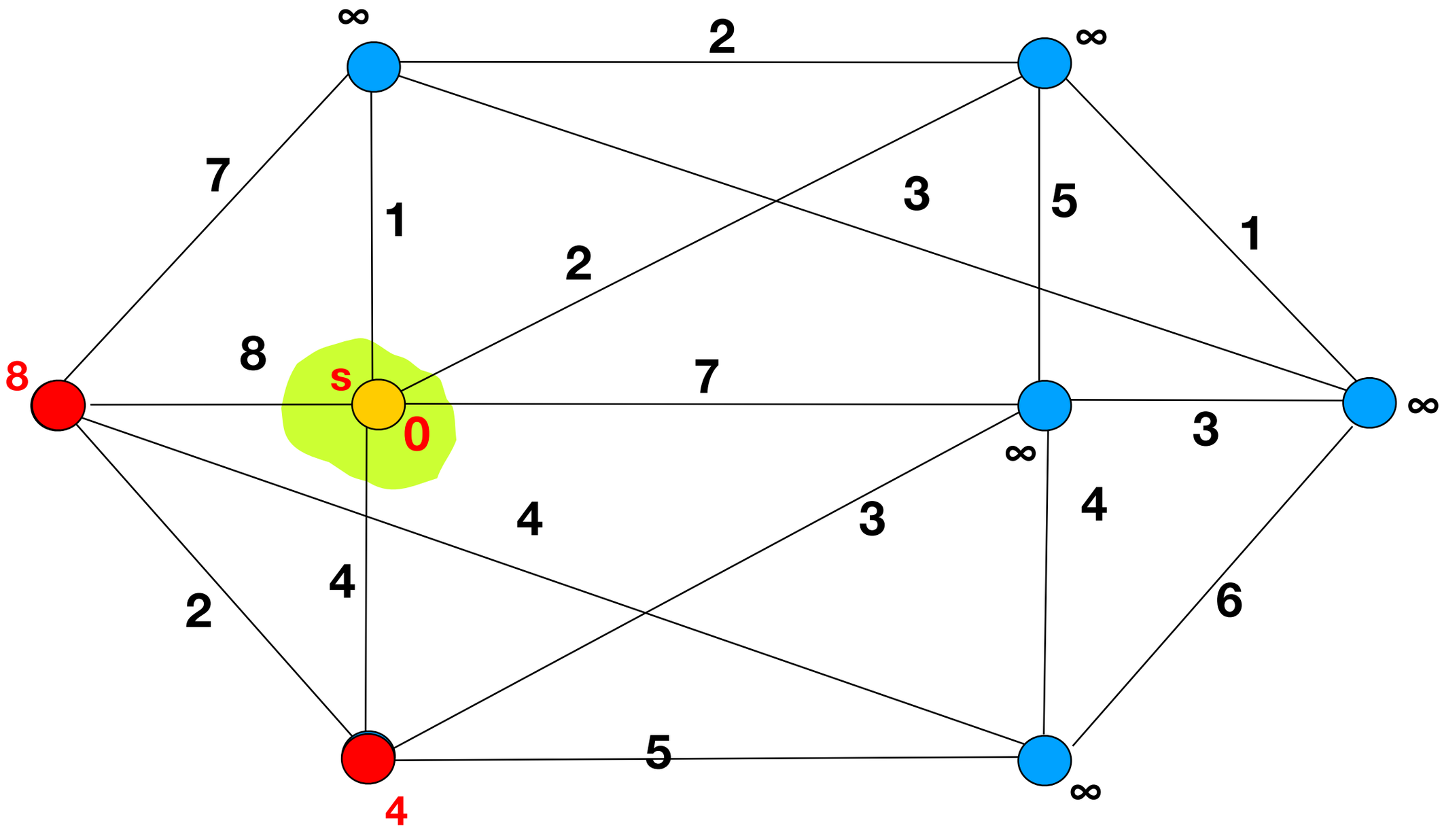
Dijkstra's Algorithm



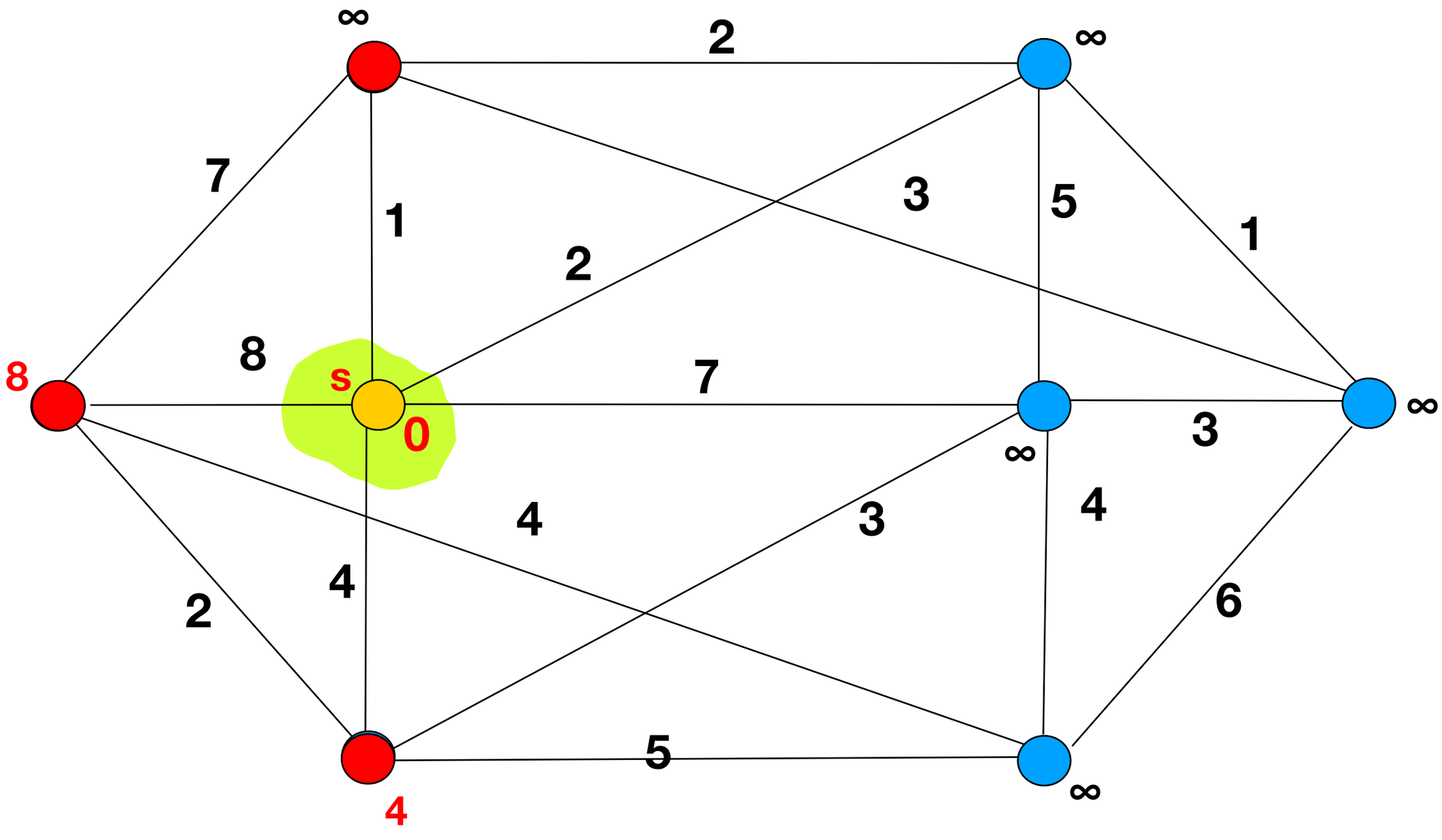
Dijkstra's Algorithm



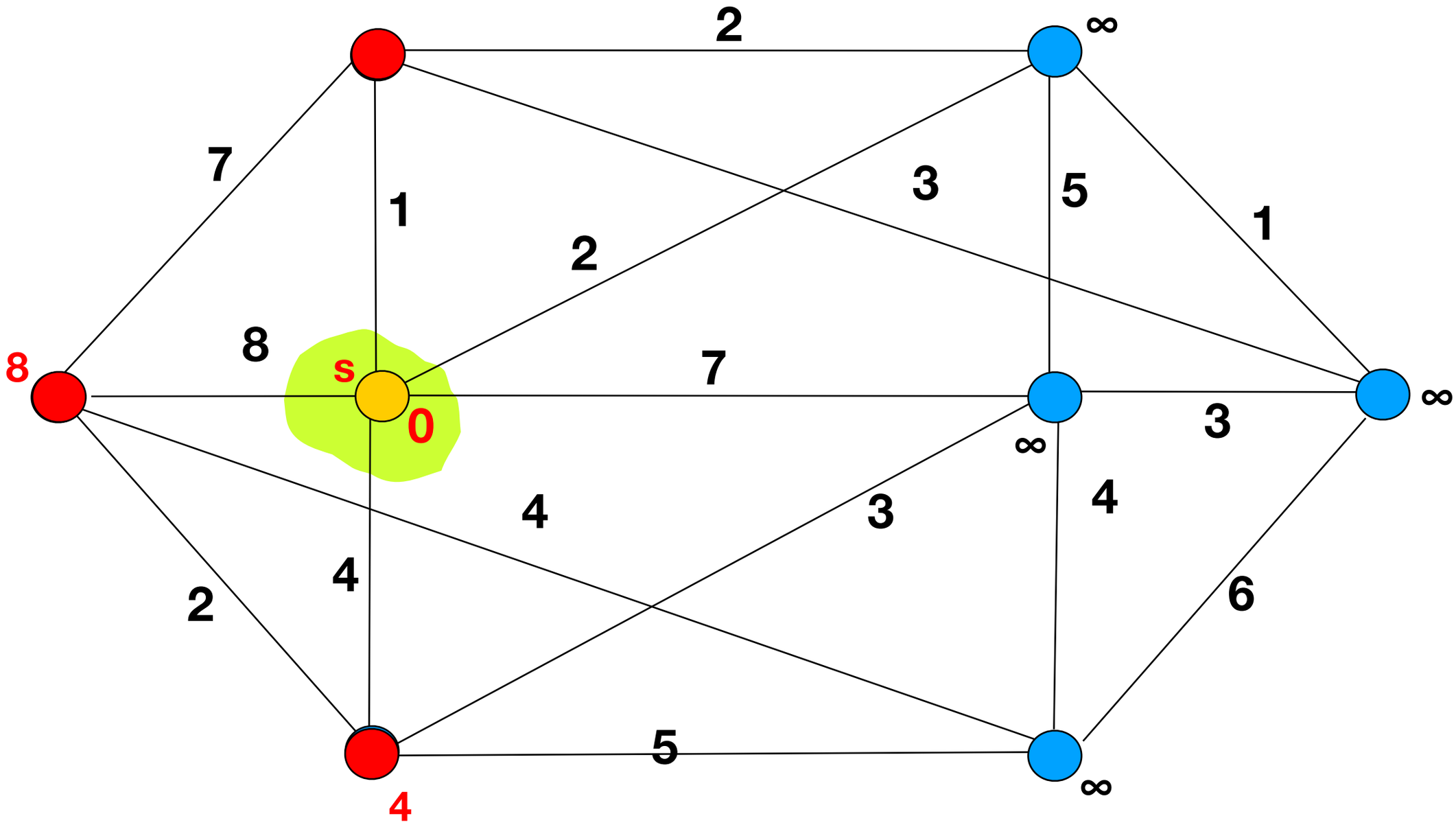
Dijkstra's Algorithm



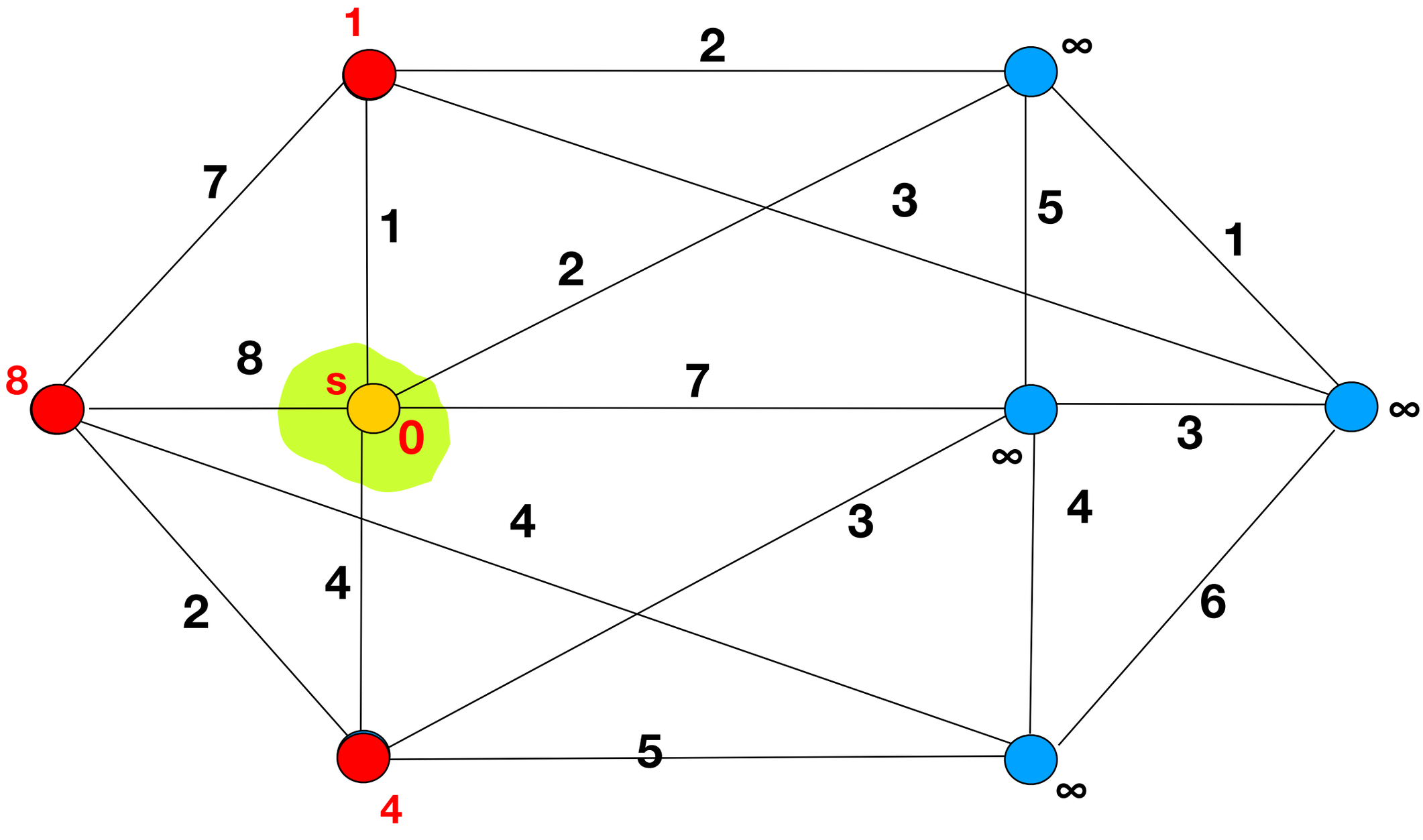
Dijkstra's Algorithm



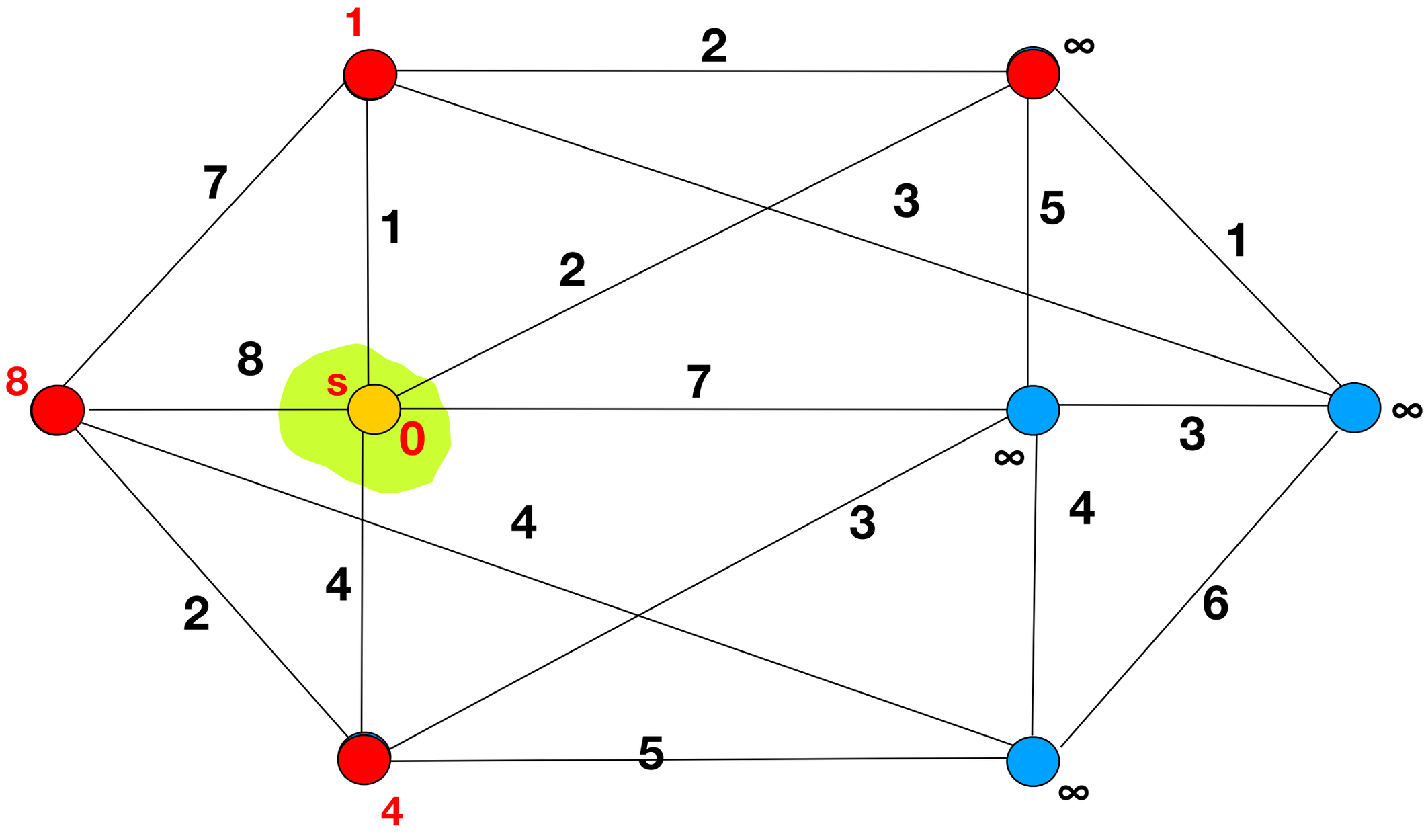
Dijkstra's Algorithm



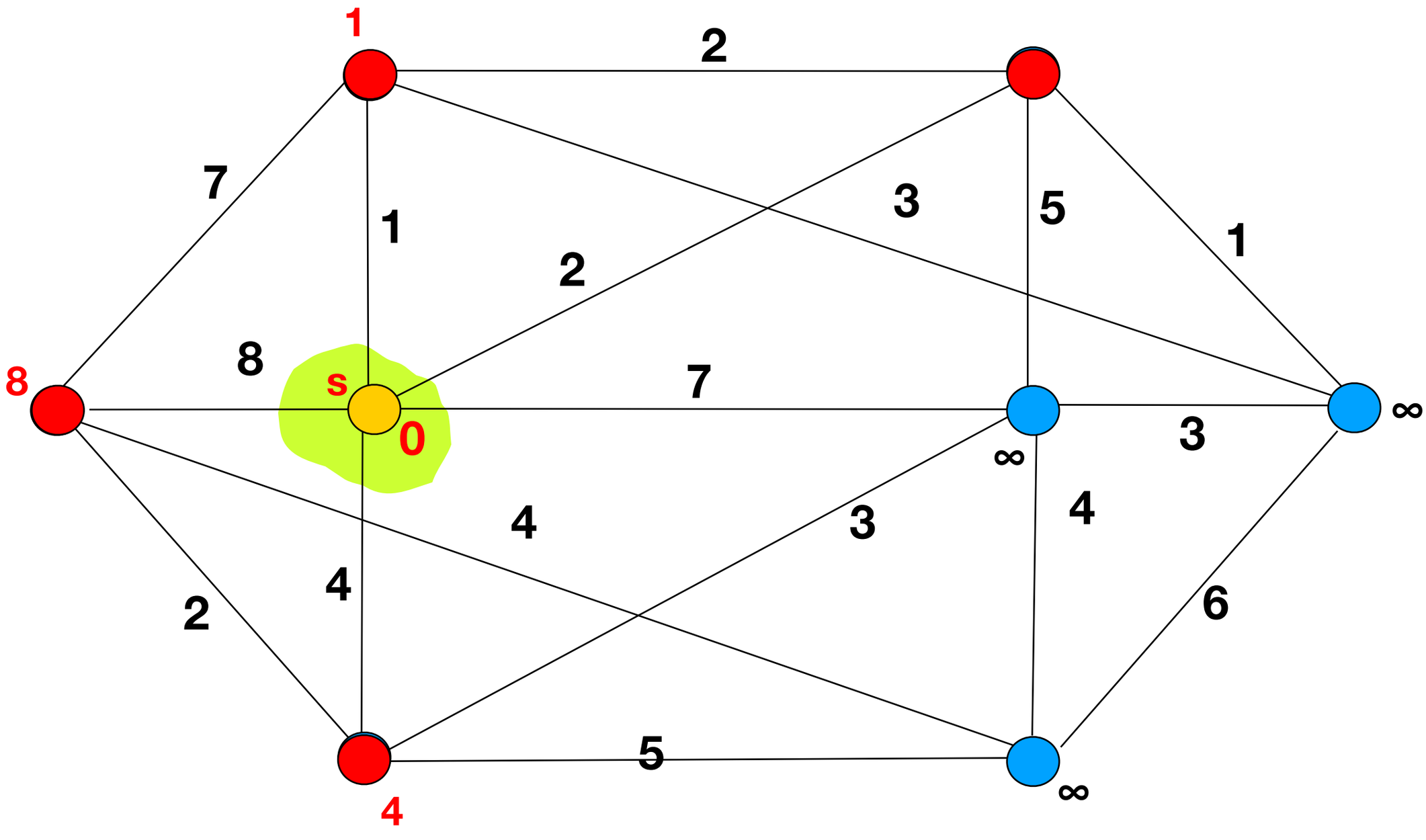
Dijkstra's Algorithm



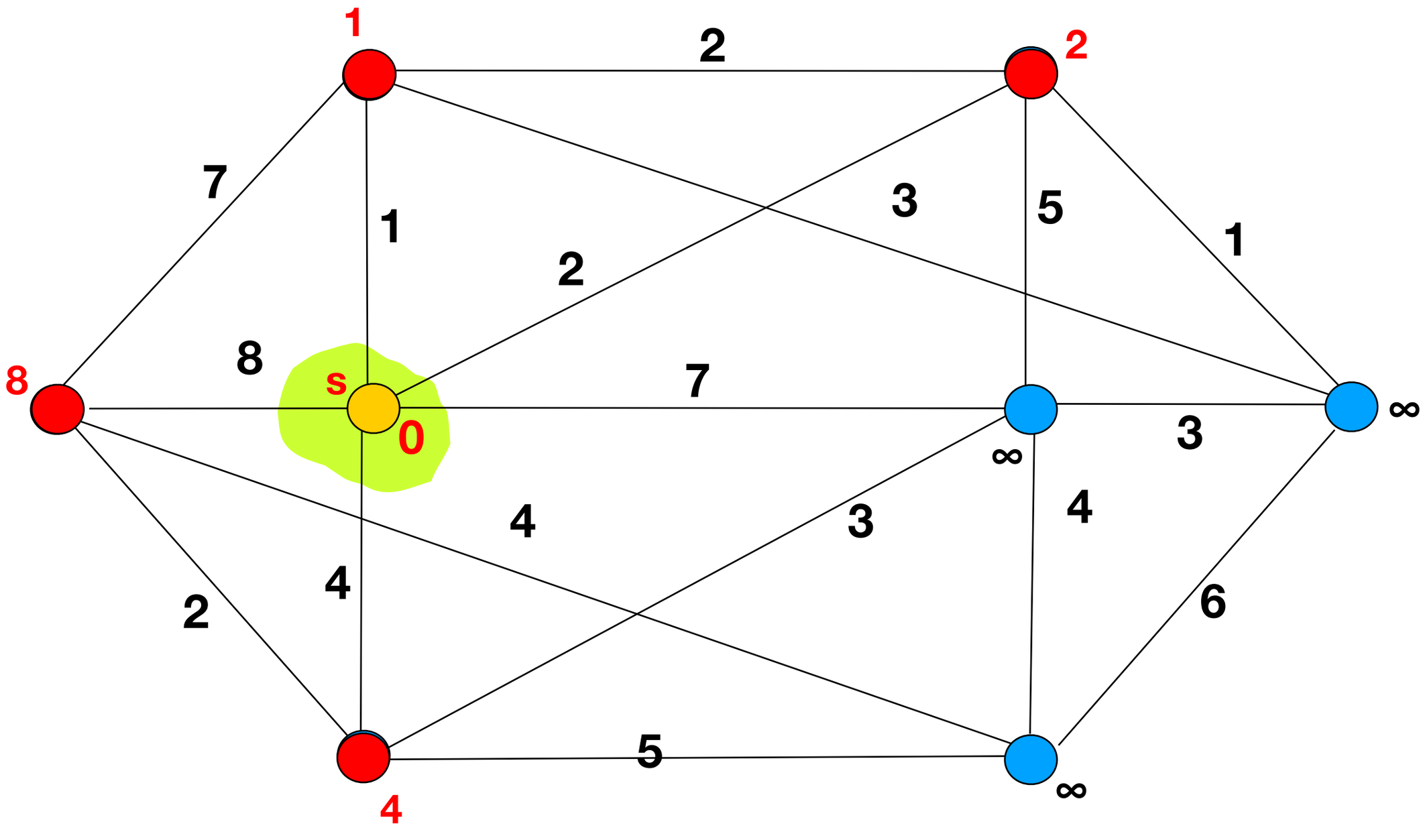
Dijkstra's Algorithm



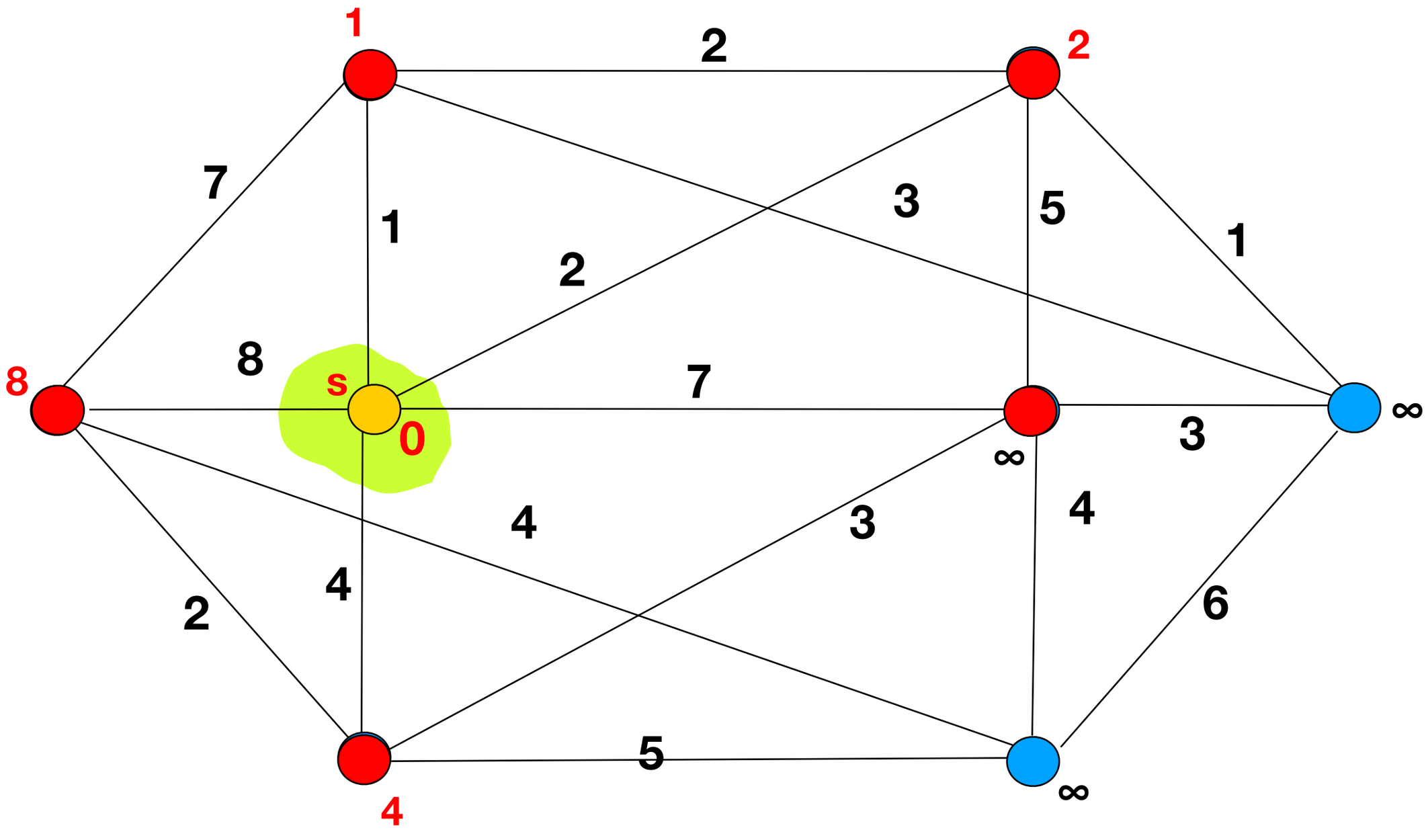
Dijkstra's Algorithm



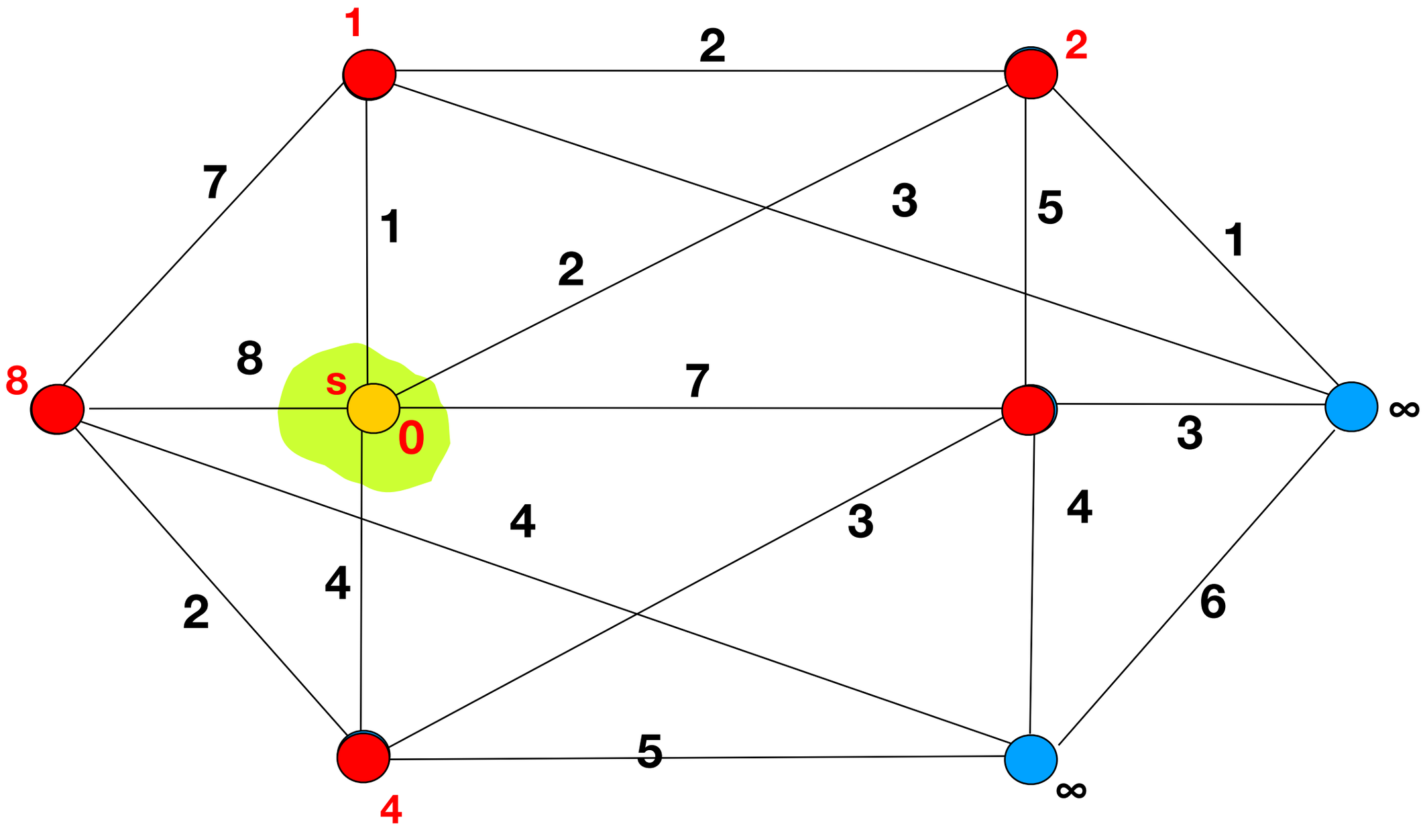
Dijkstra's Algorithm



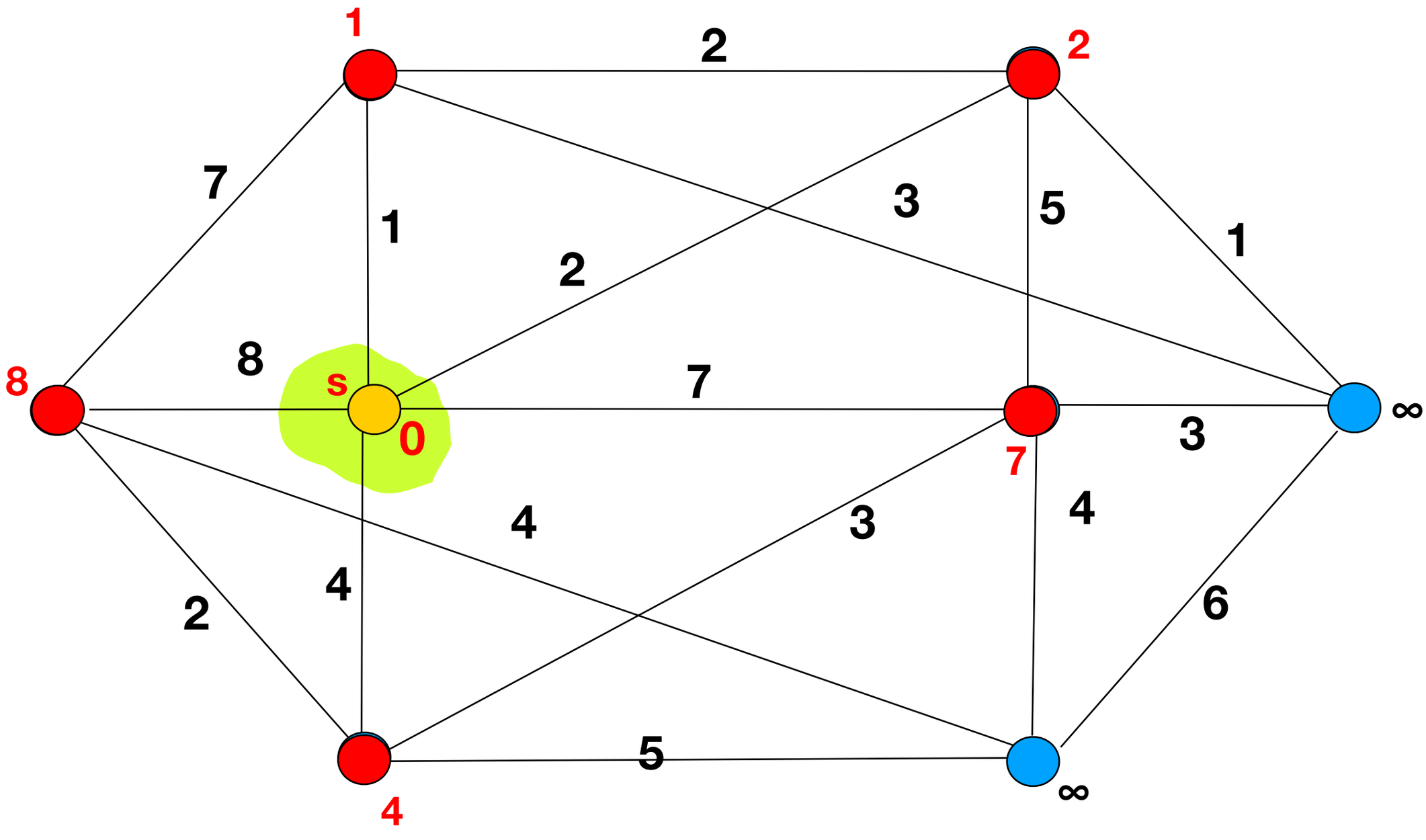
Dijkstra's Algorithm



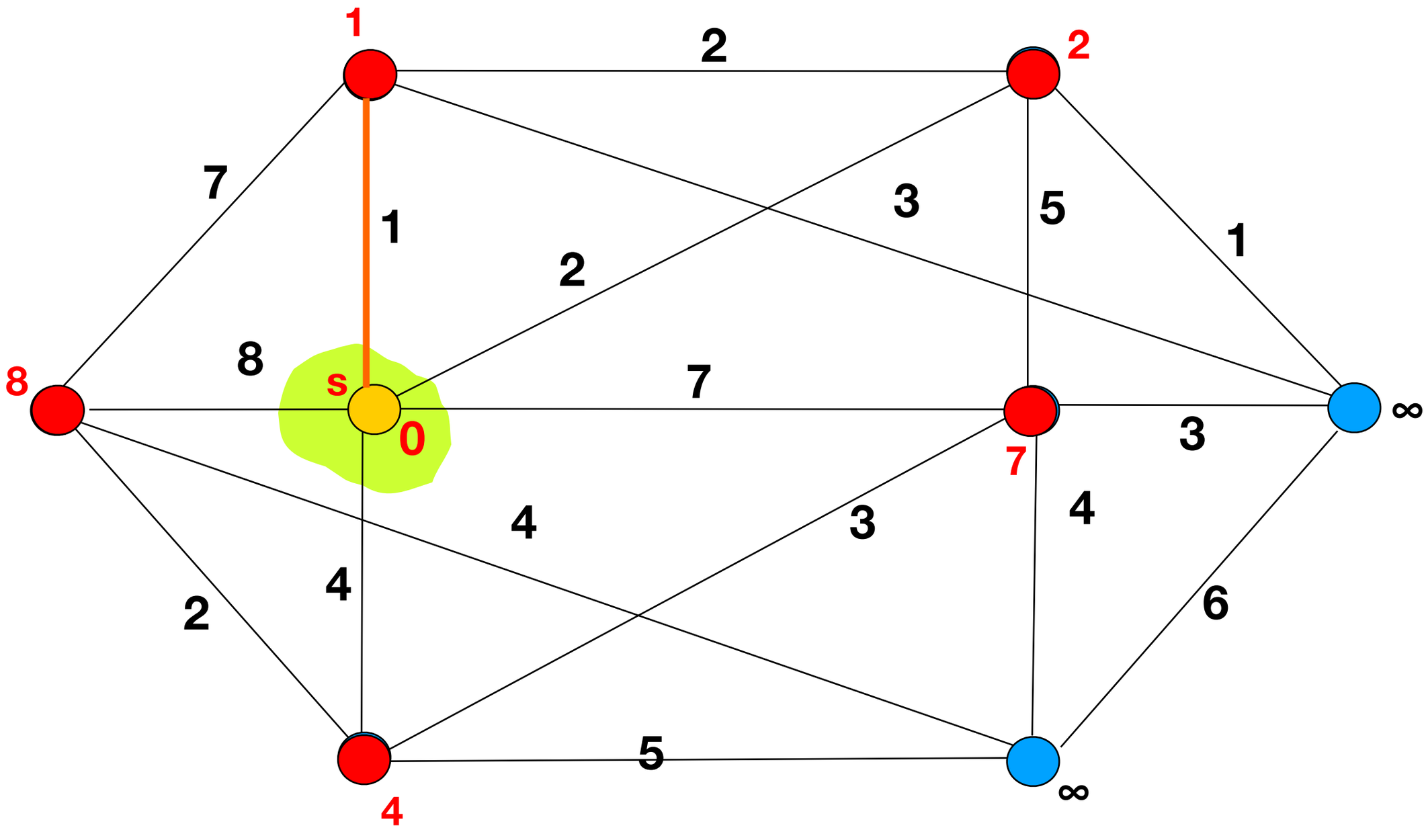
Dijkstra's Algorithm



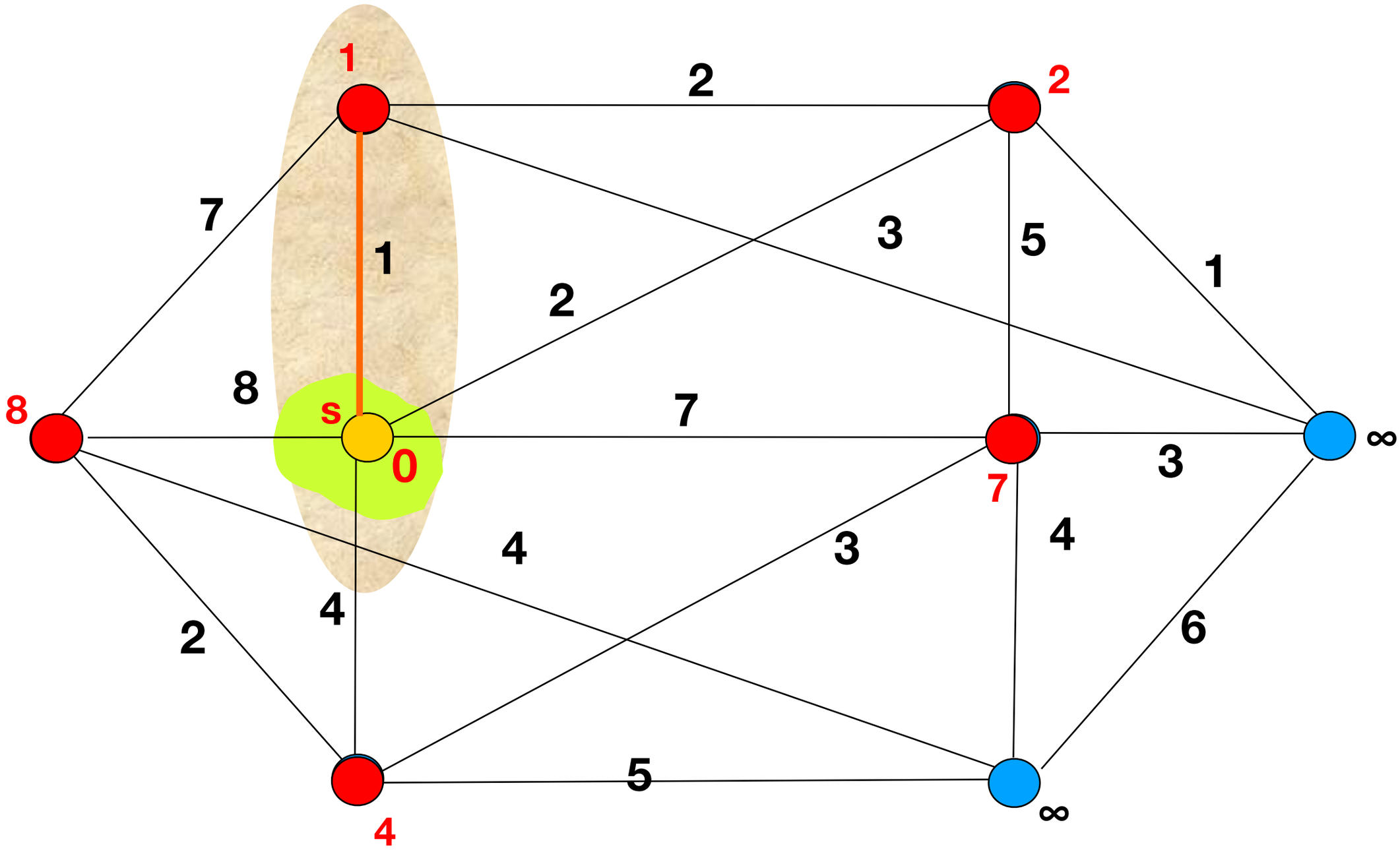
Dijkstra's Algorithm



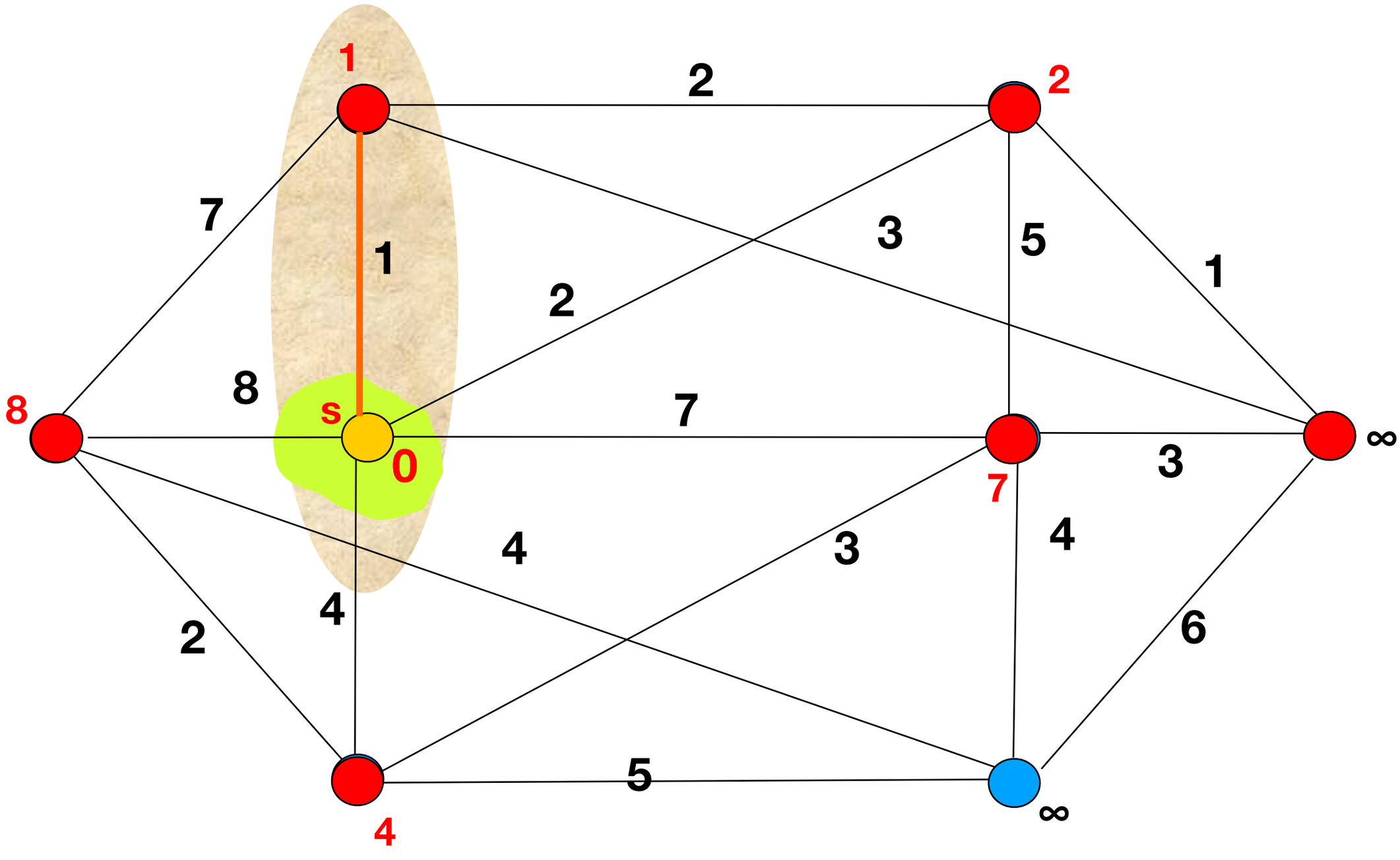
Dijkstra's Algorithm



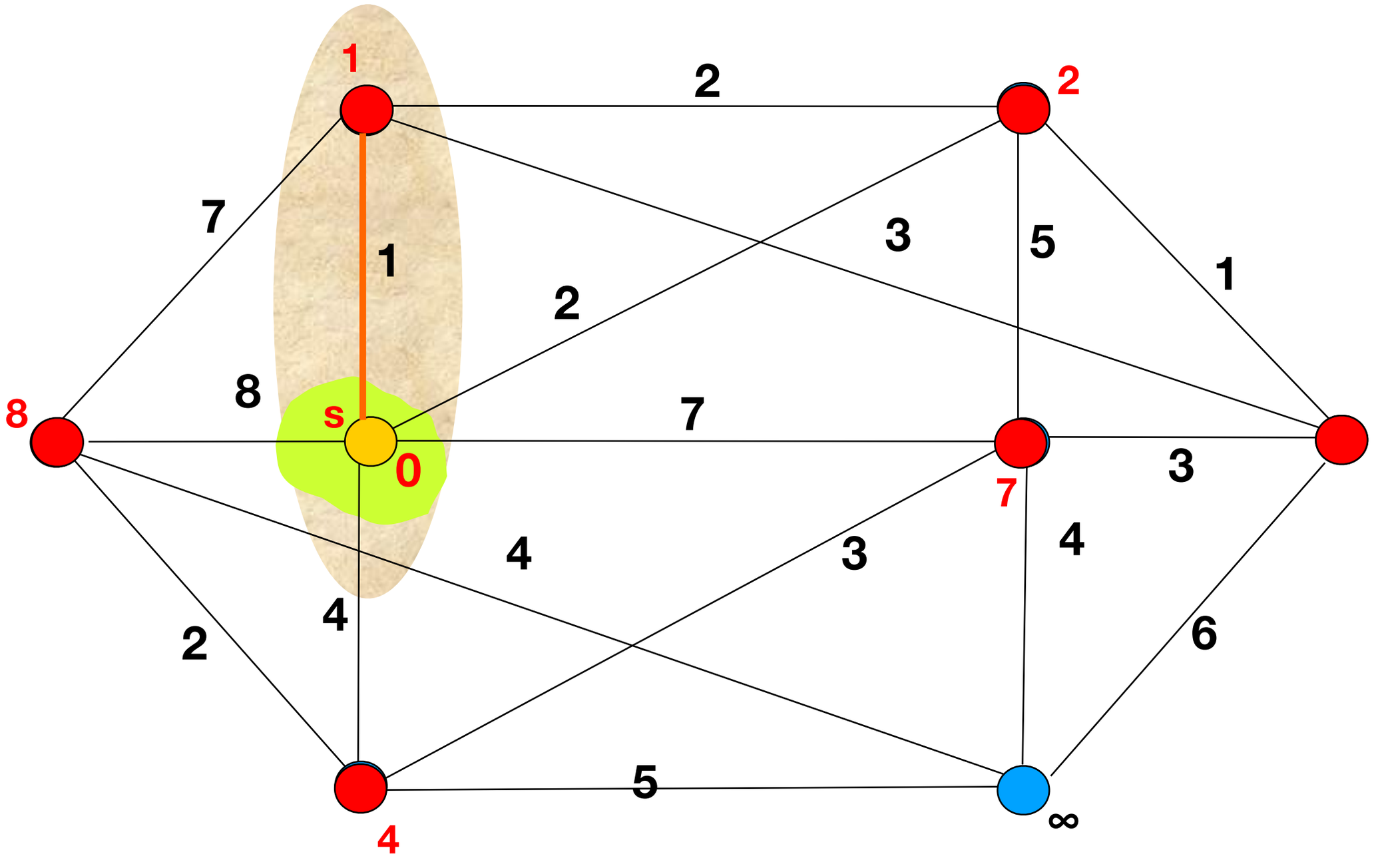
Dijkstra's Algorithm



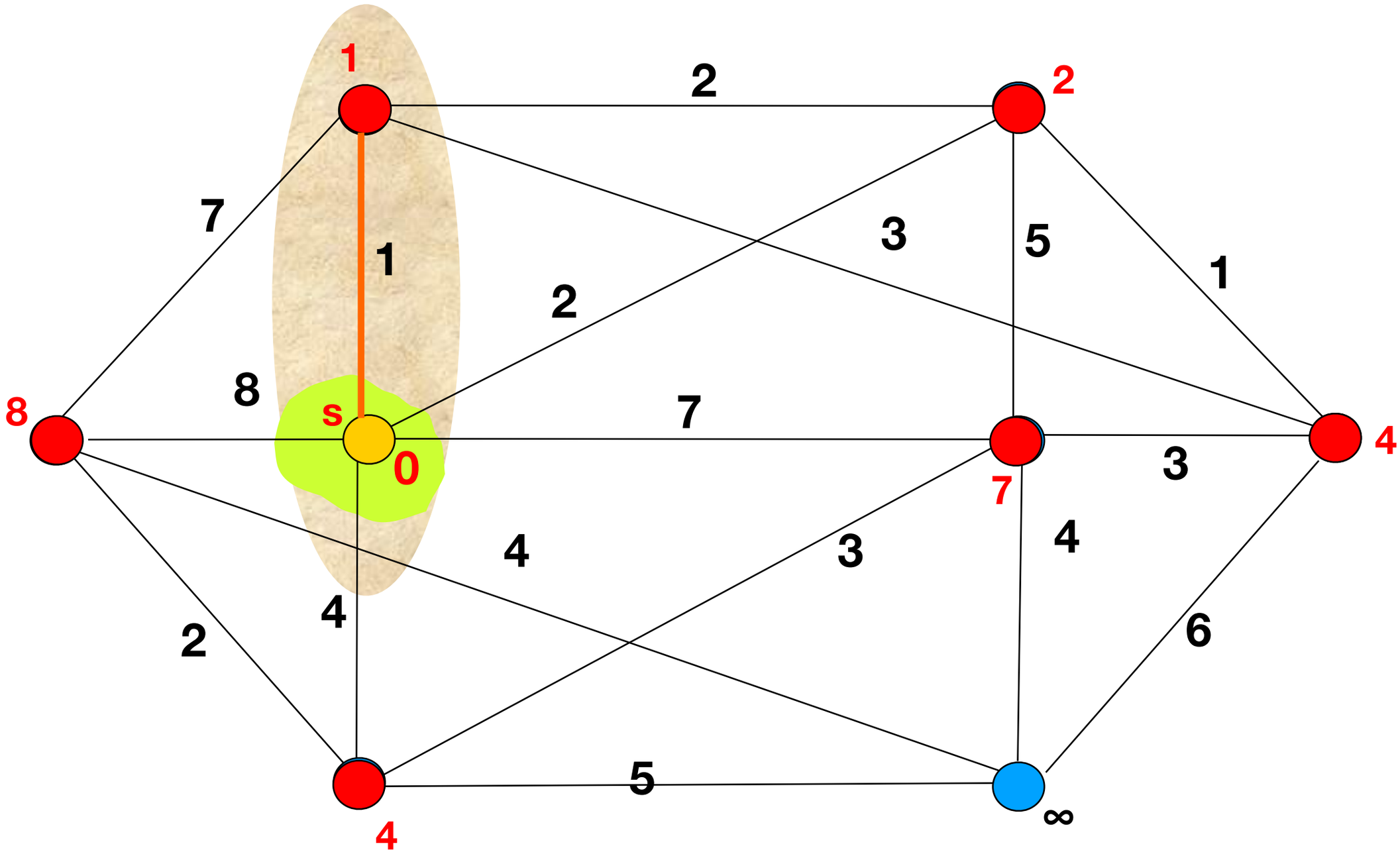
Dijkstra's Algorithm



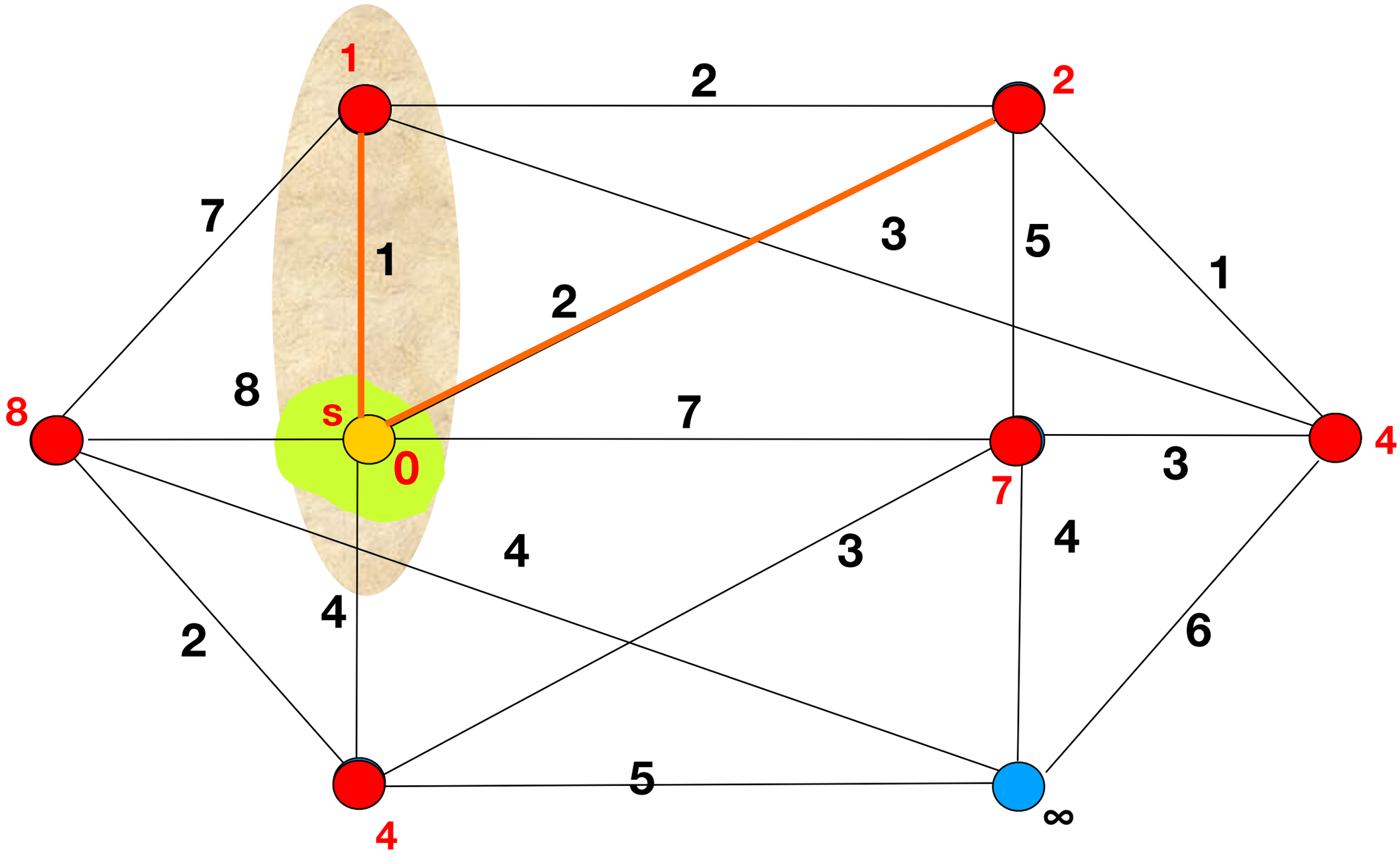
Dijkstra's Algorithm



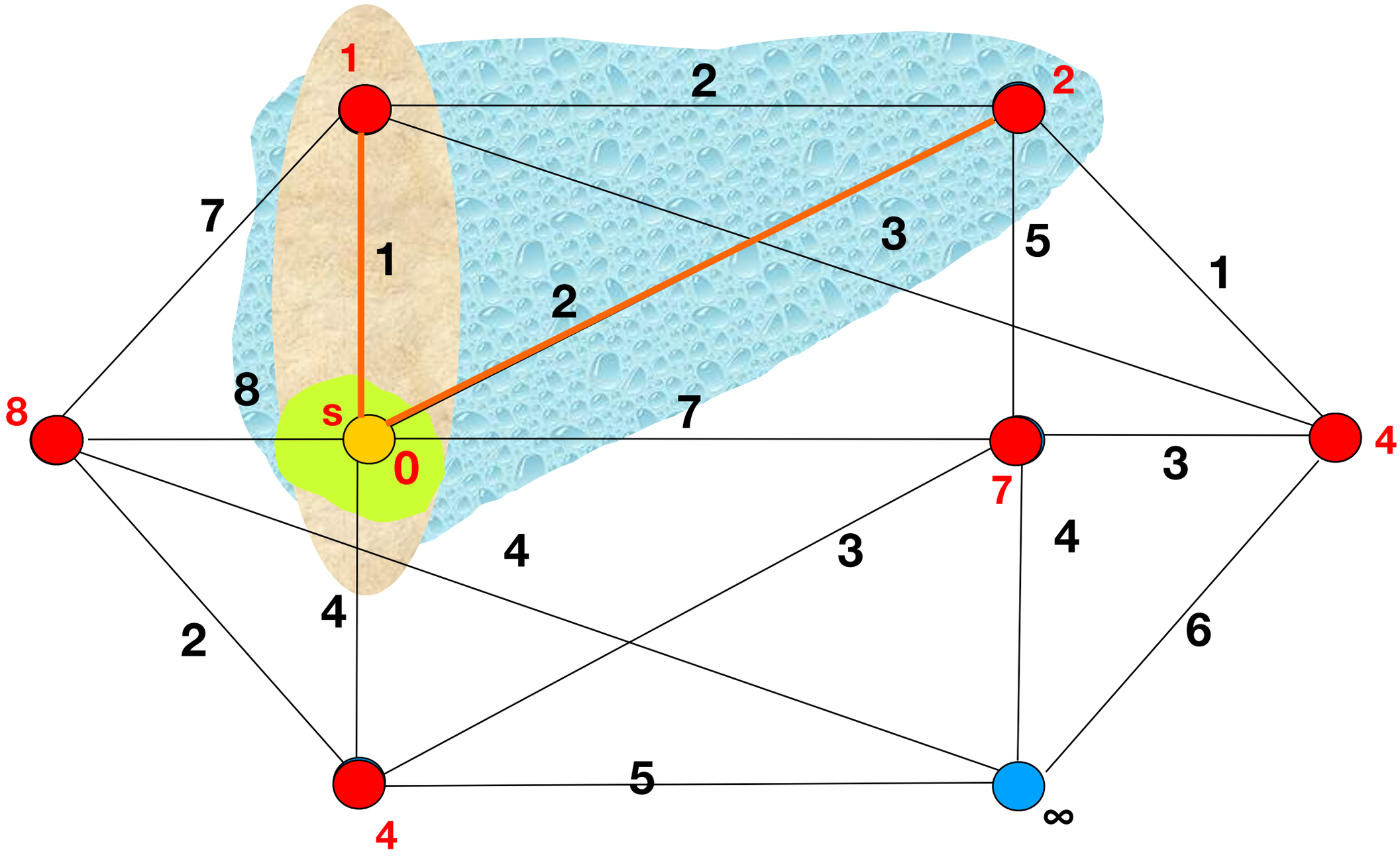
Dijkstra's Algorithm



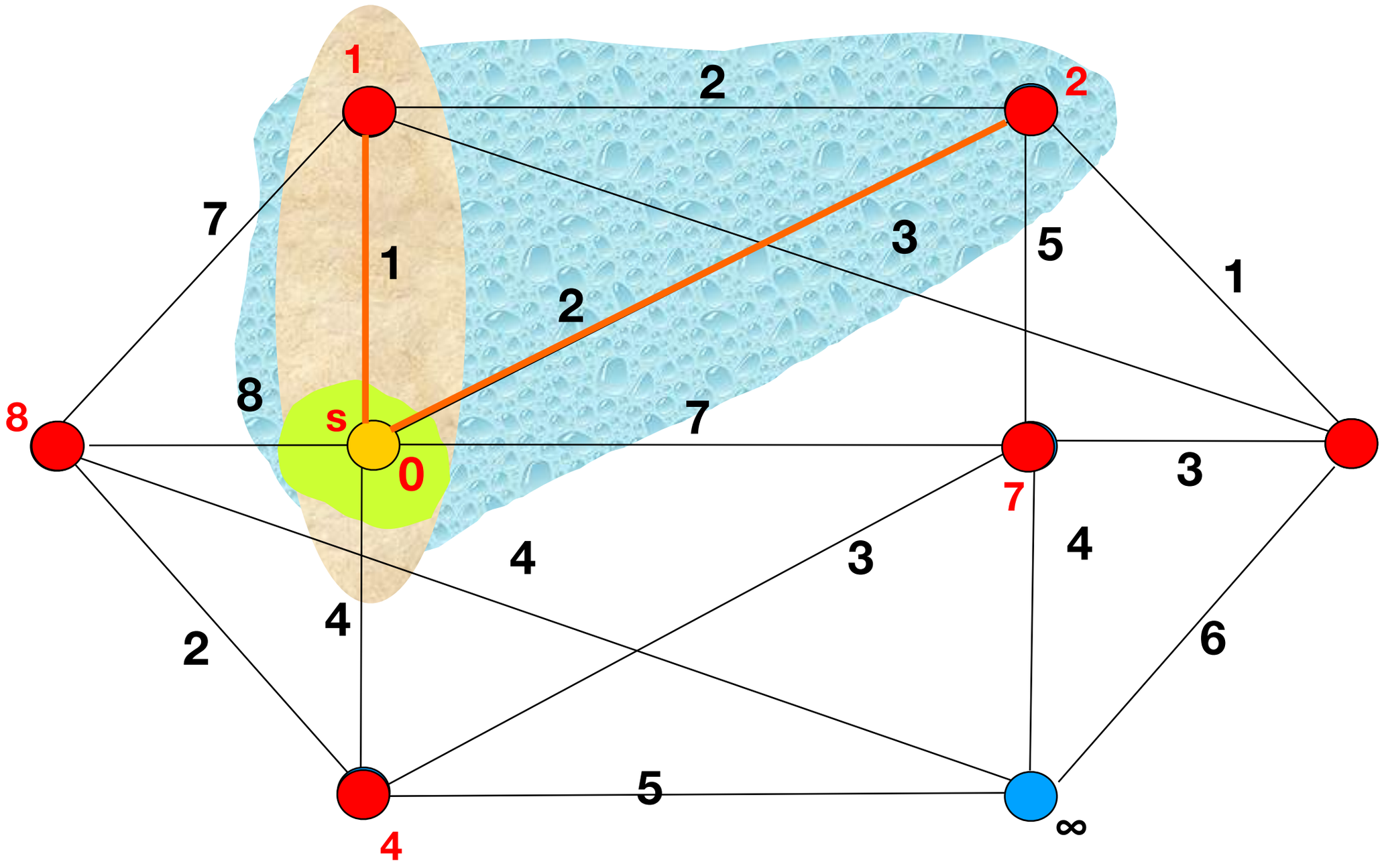
Dijkstra's Algorithm



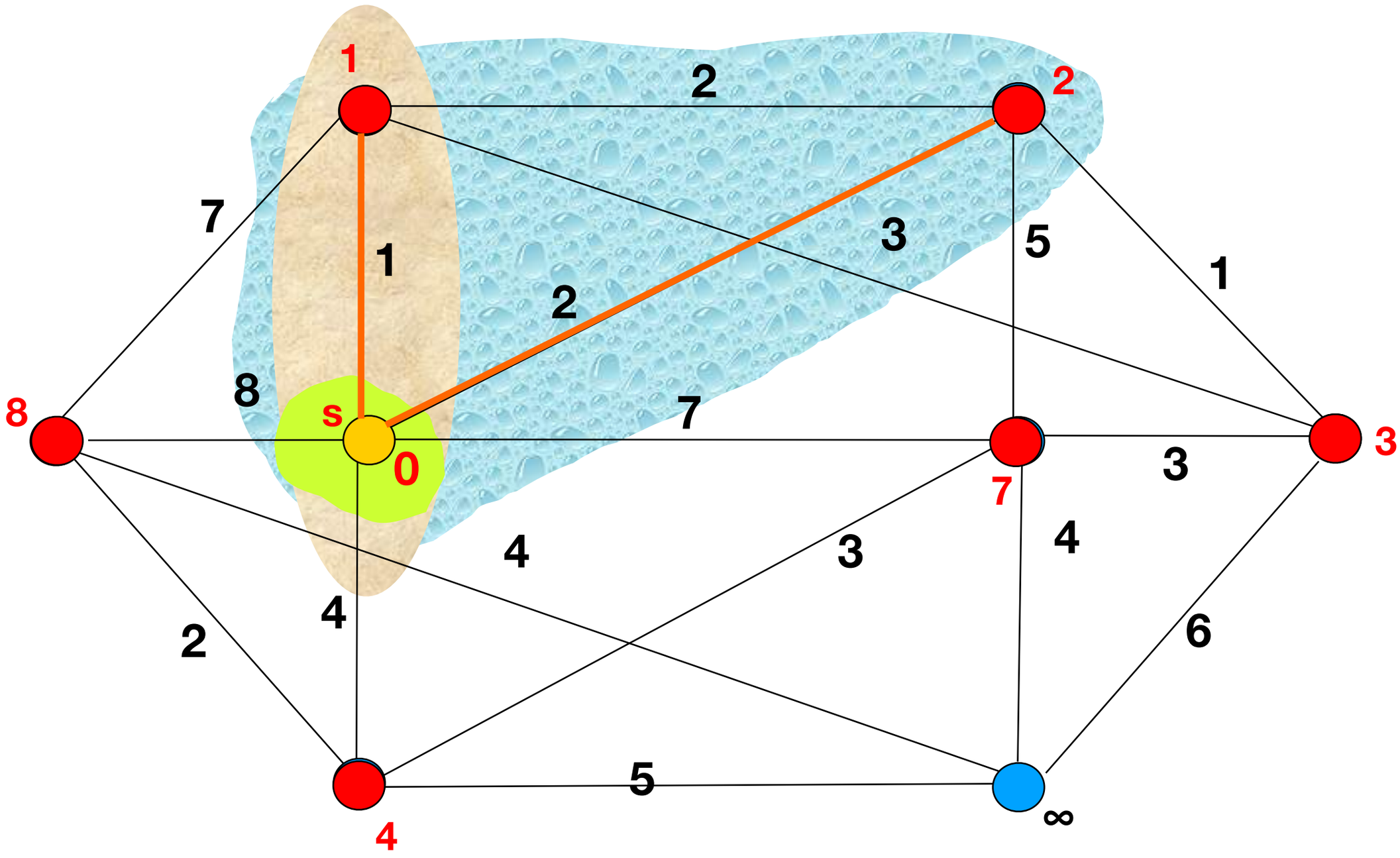
Dijkstra's Algorithm



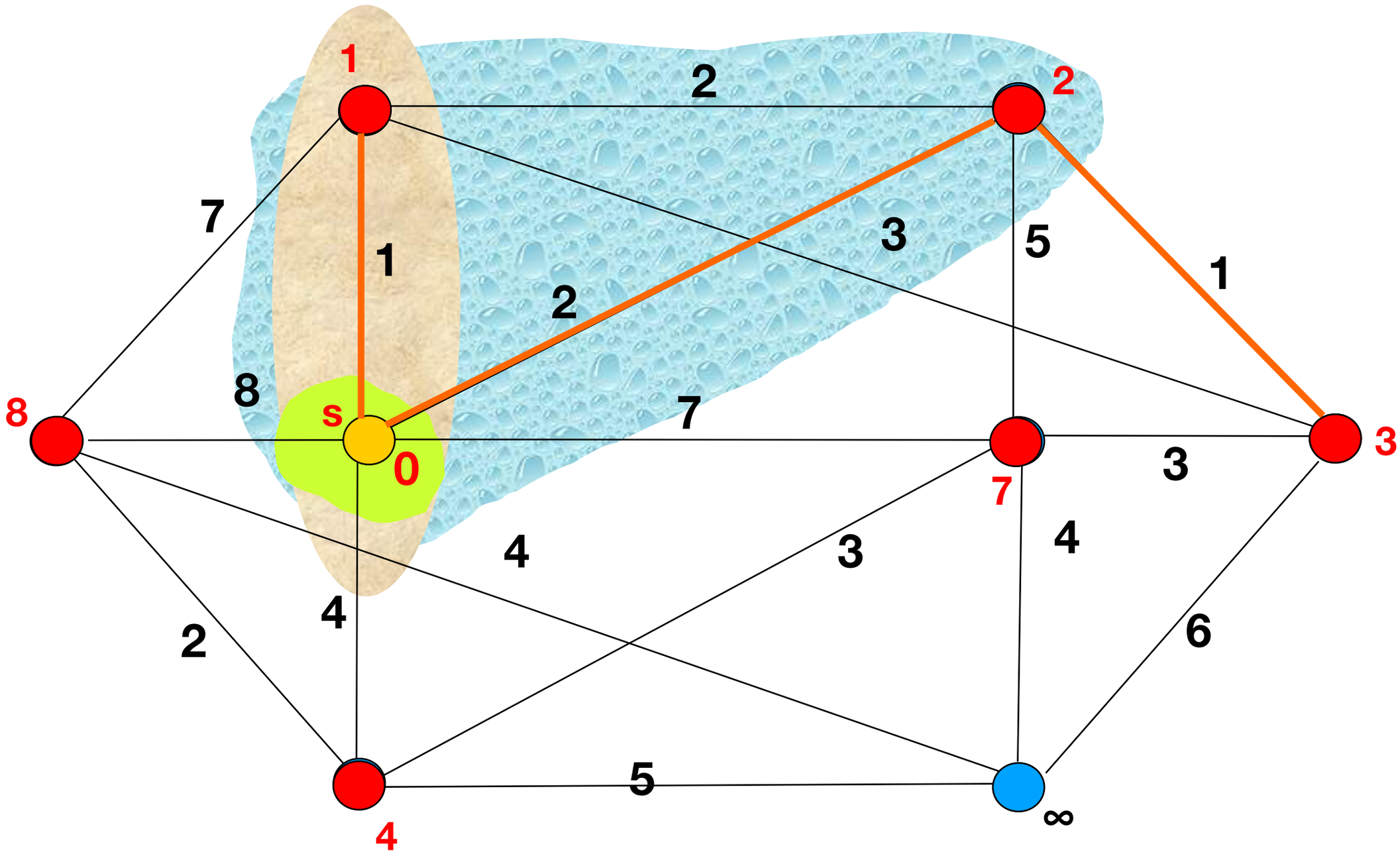
Dijkstra's Algorithm



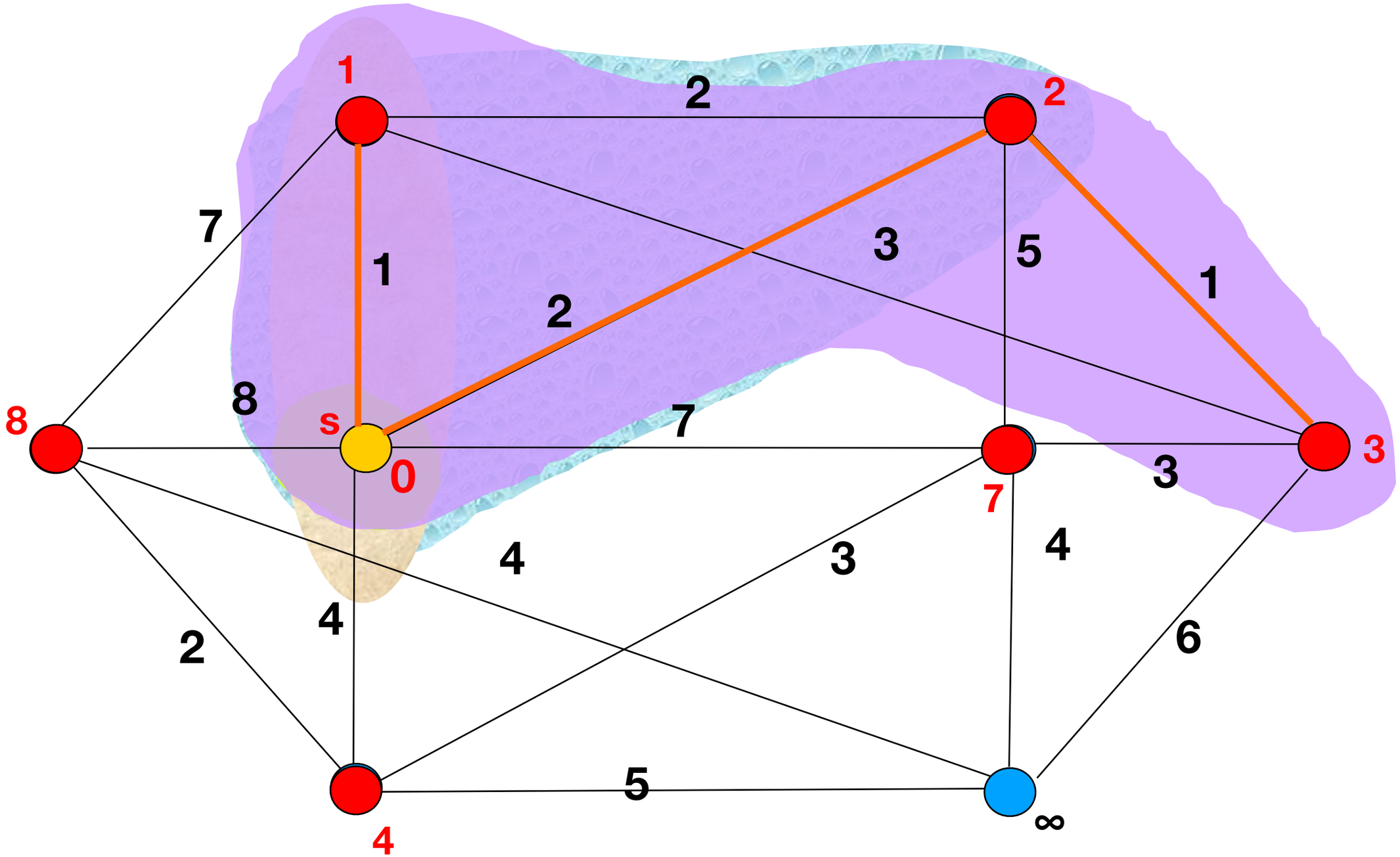
Dijkstra's Algorithm



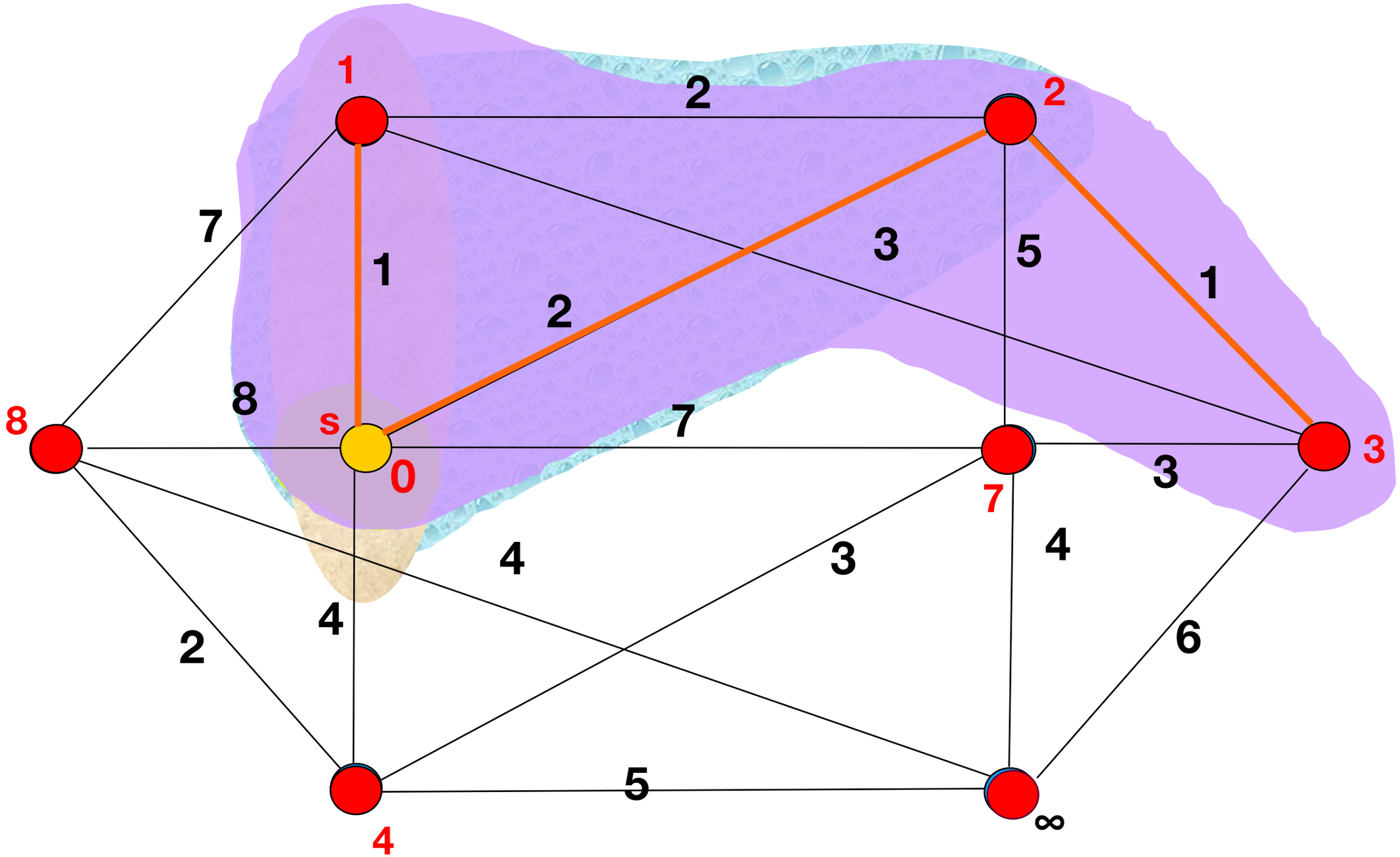
Dijkstra's Algorithm



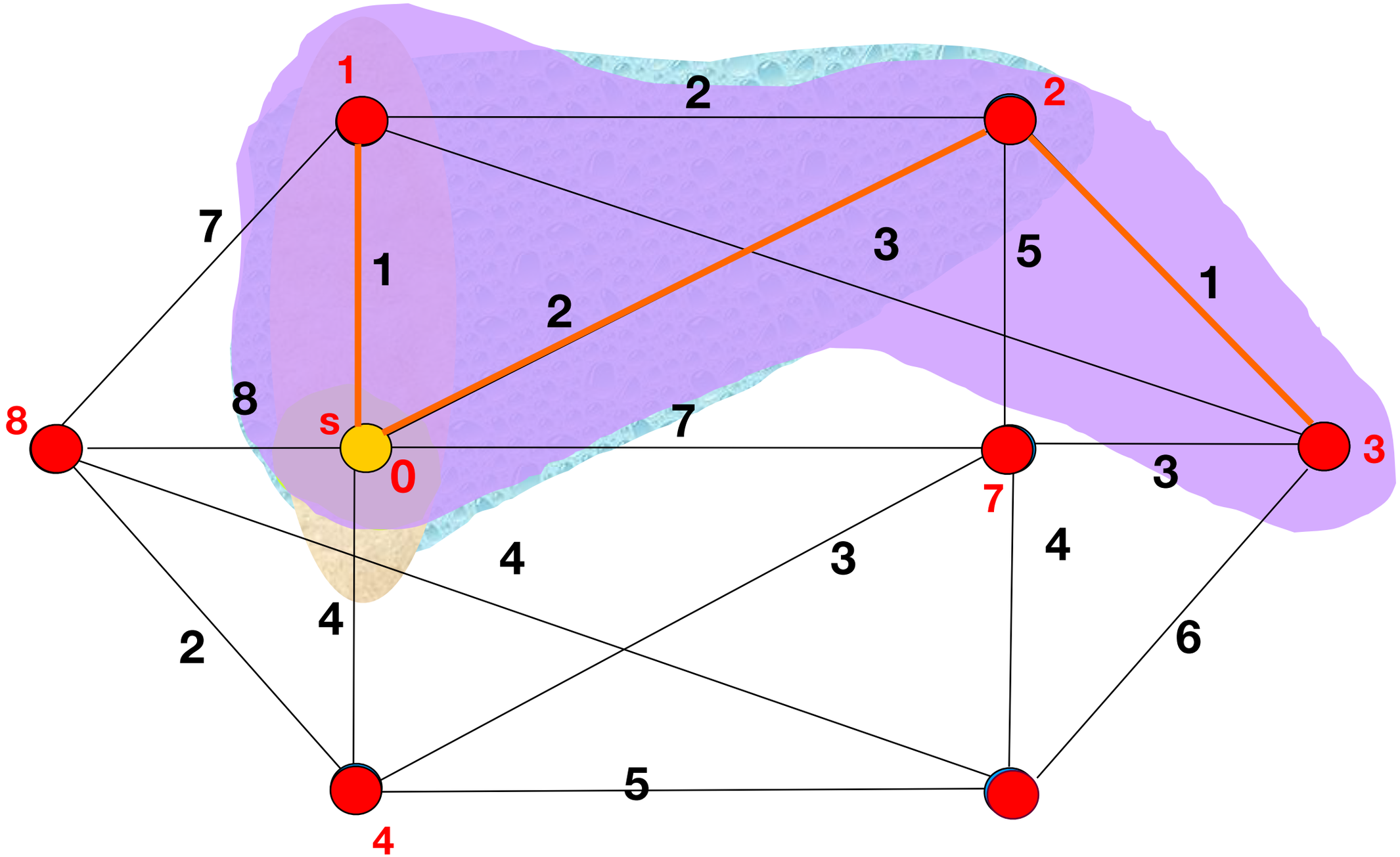
Dijkstra's Algorithm



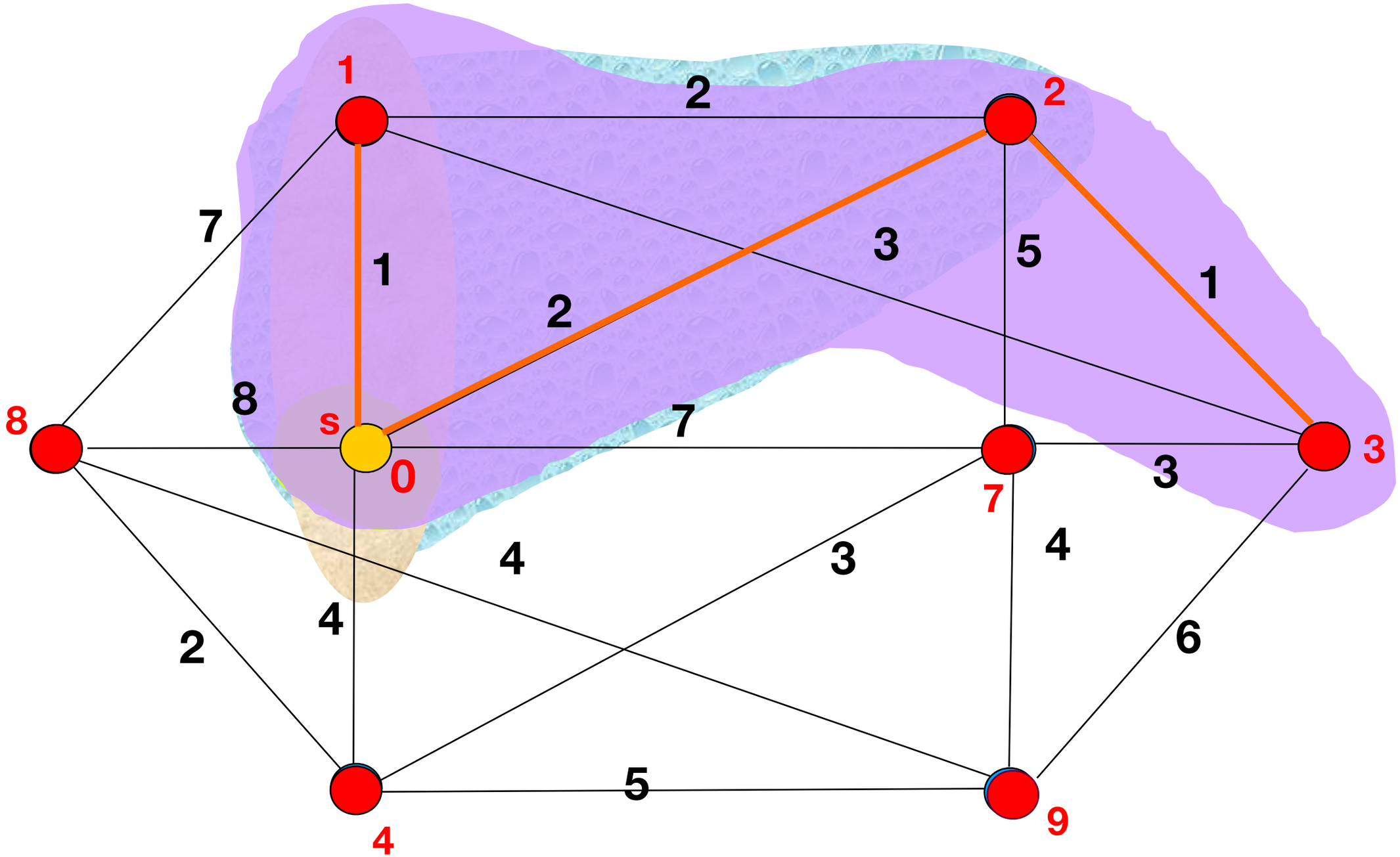
Dijkstra's Algorithm



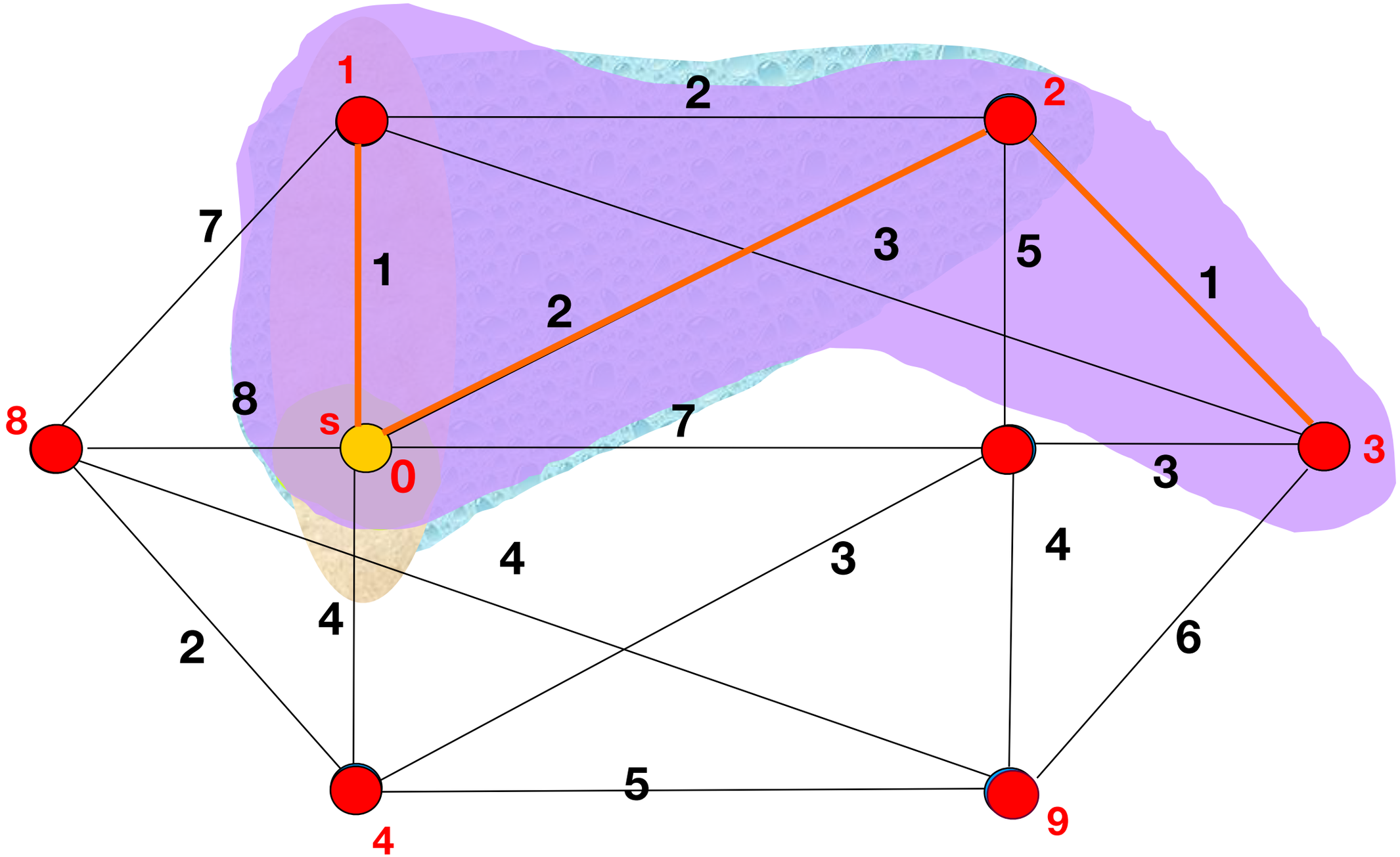
Dijkstra's Algorithm



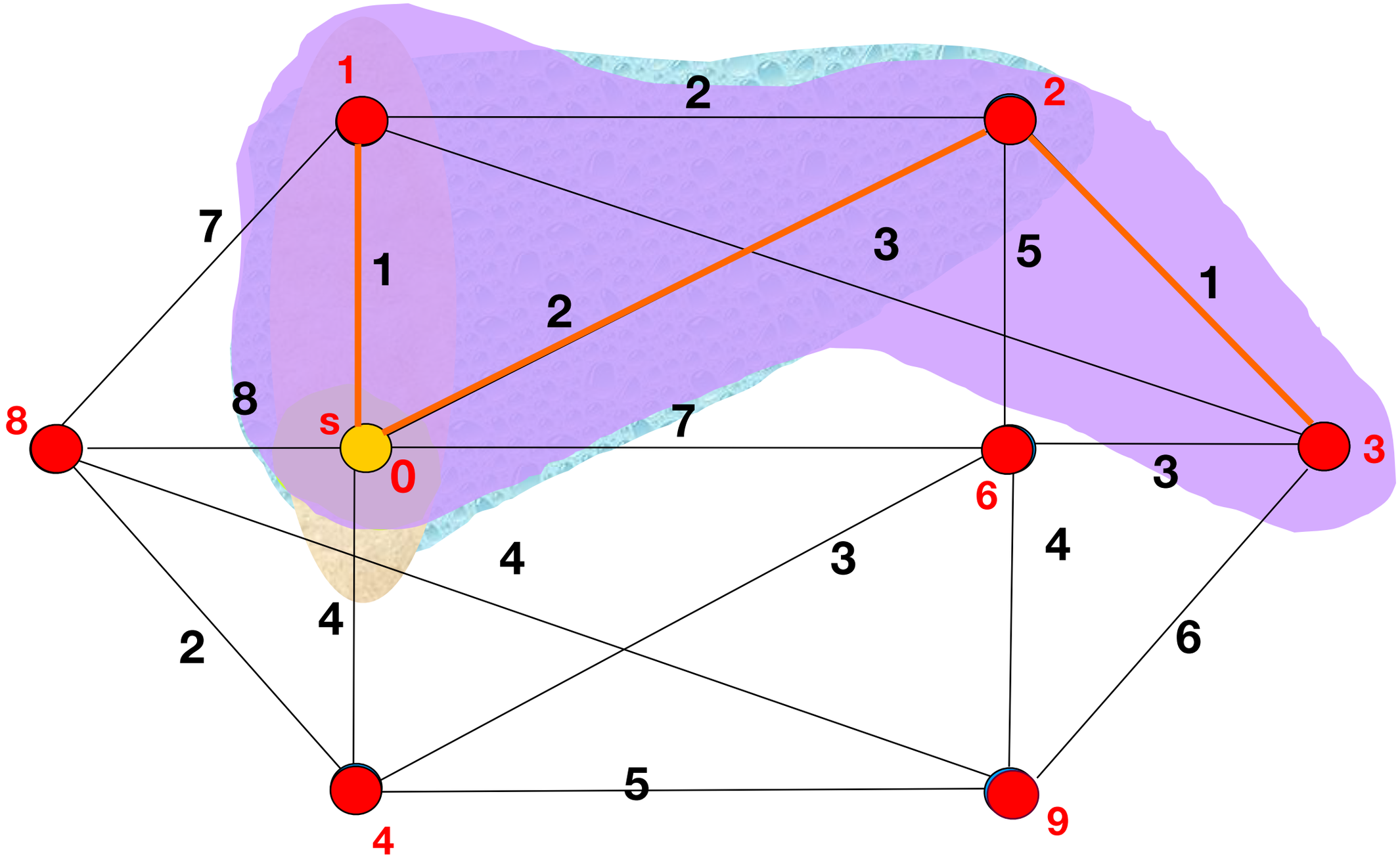
Dijkstra's Algorithm



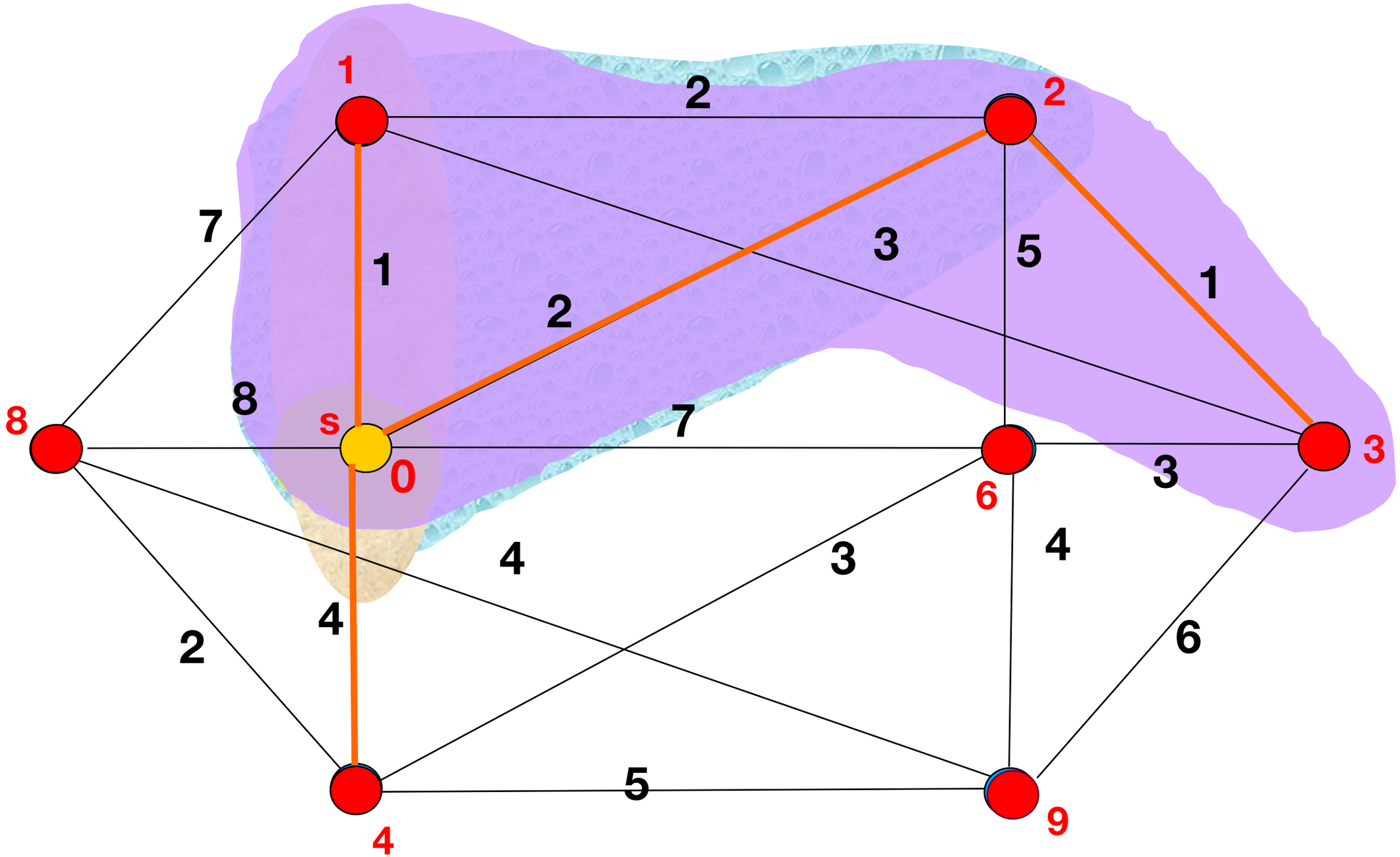
Dijkstra's Algorithm



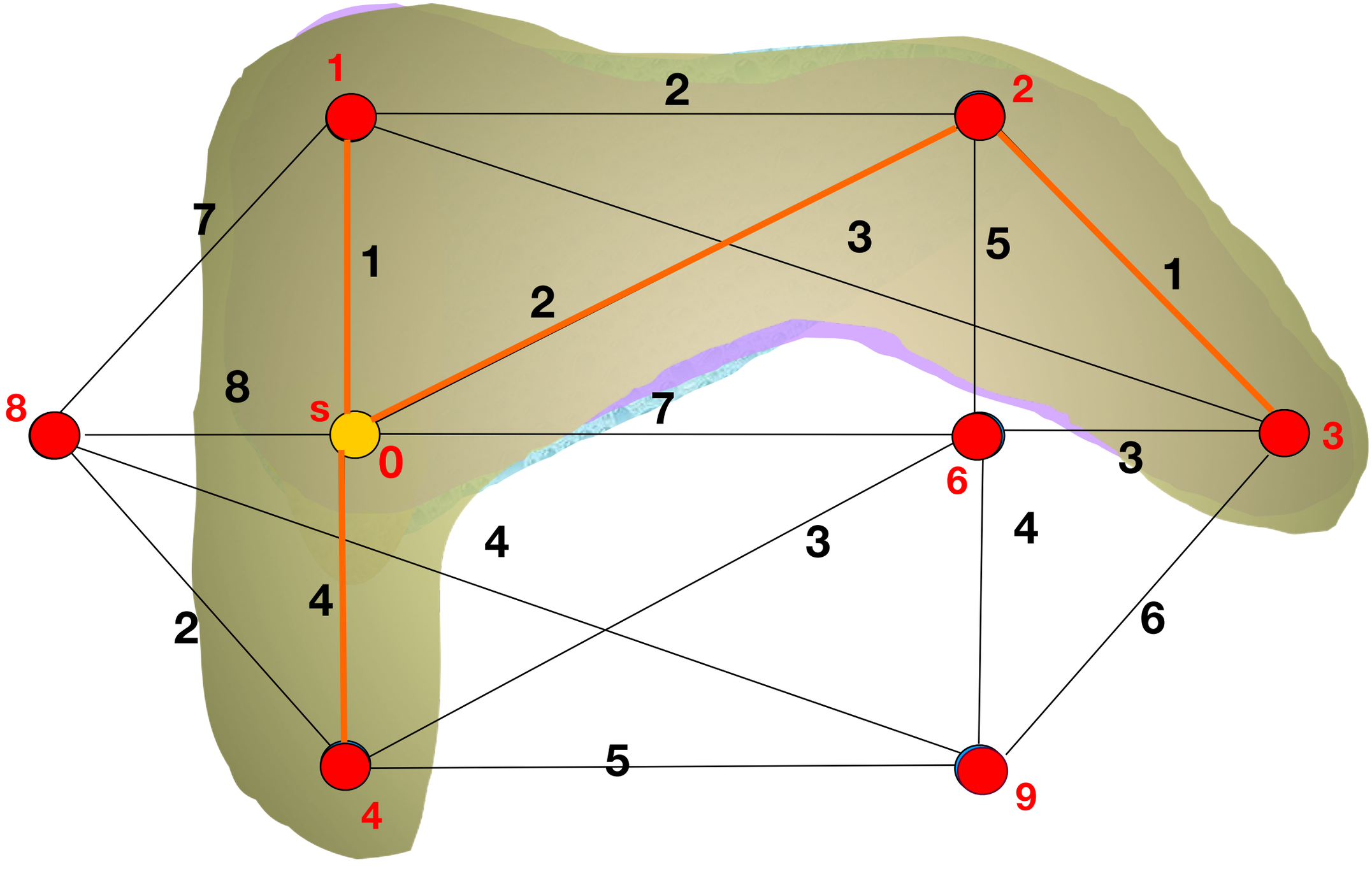
Dijkstra's Algorithm



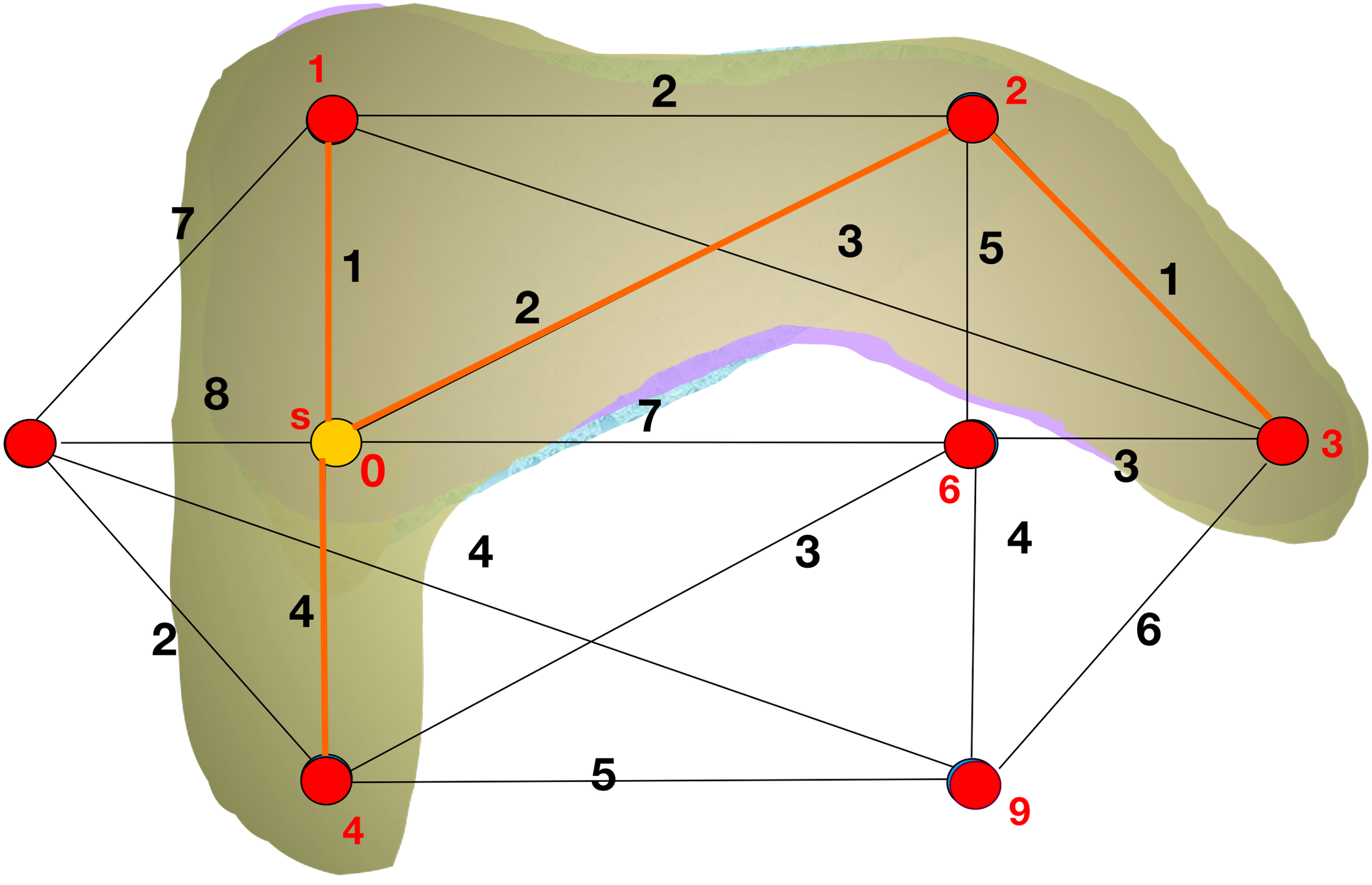
Dijkstra's Algorithm



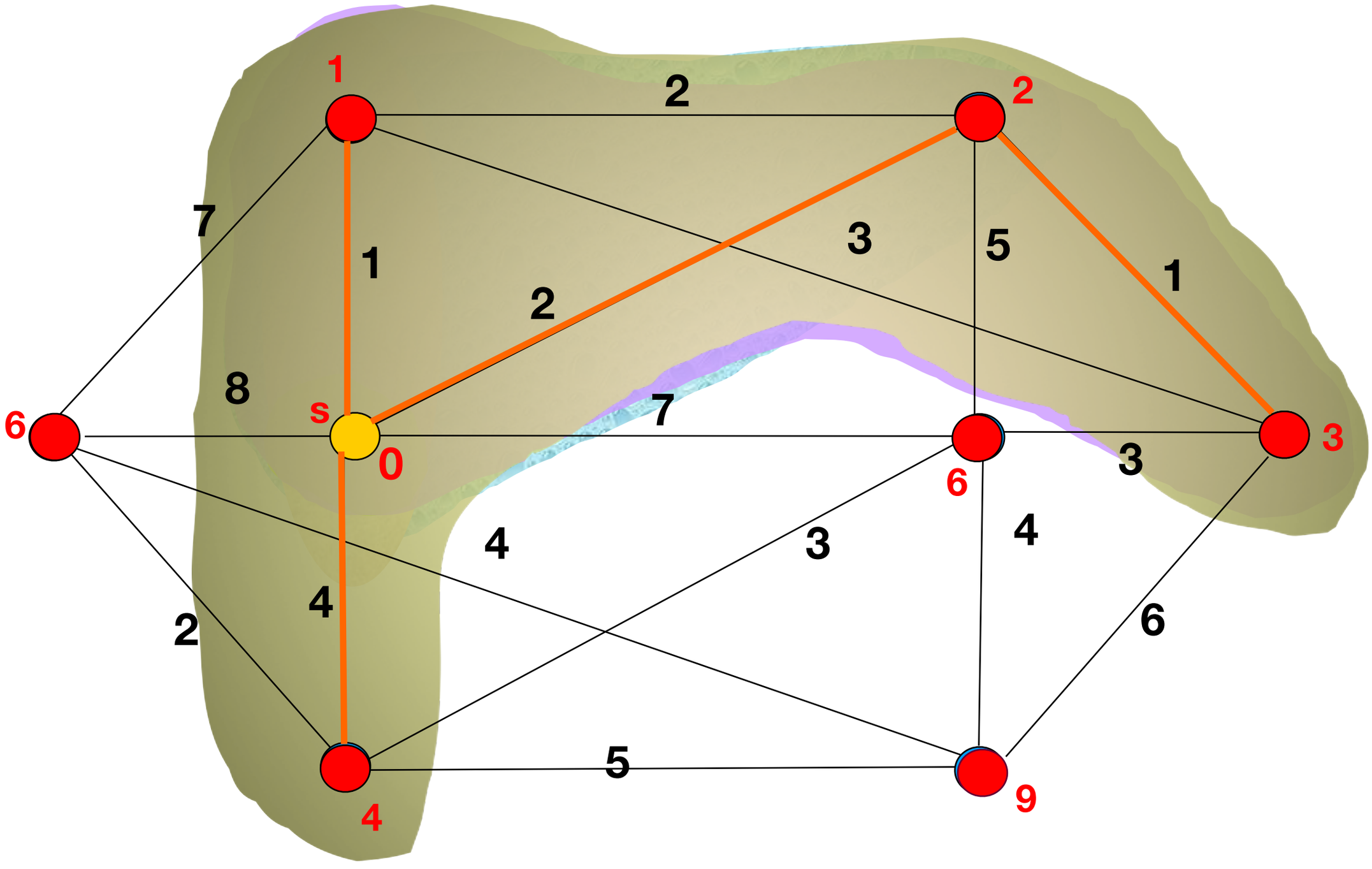
Dijkstra's Algorithm



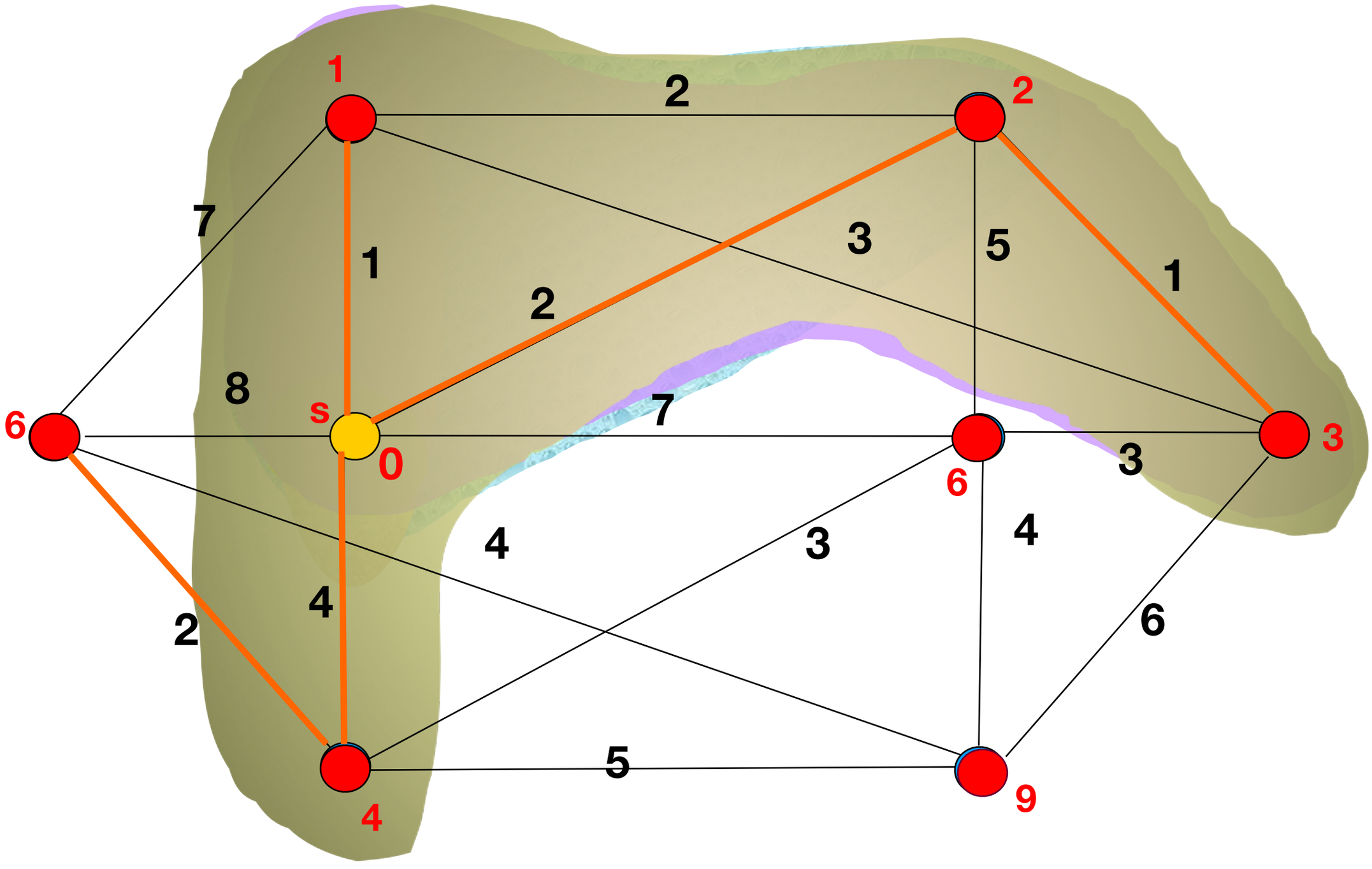
Dijkstra's Algorithm



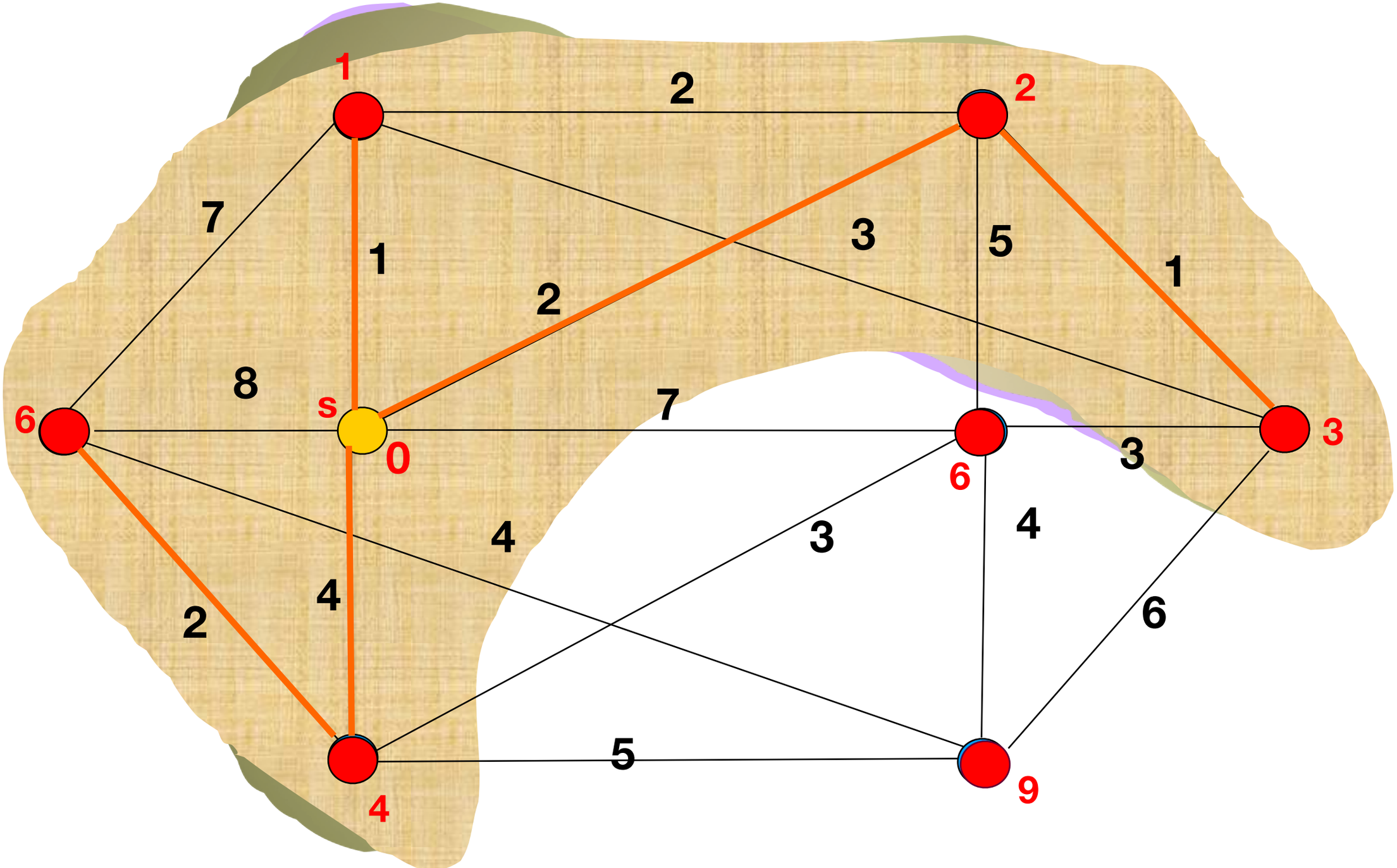
Dijkstra's Algorithm



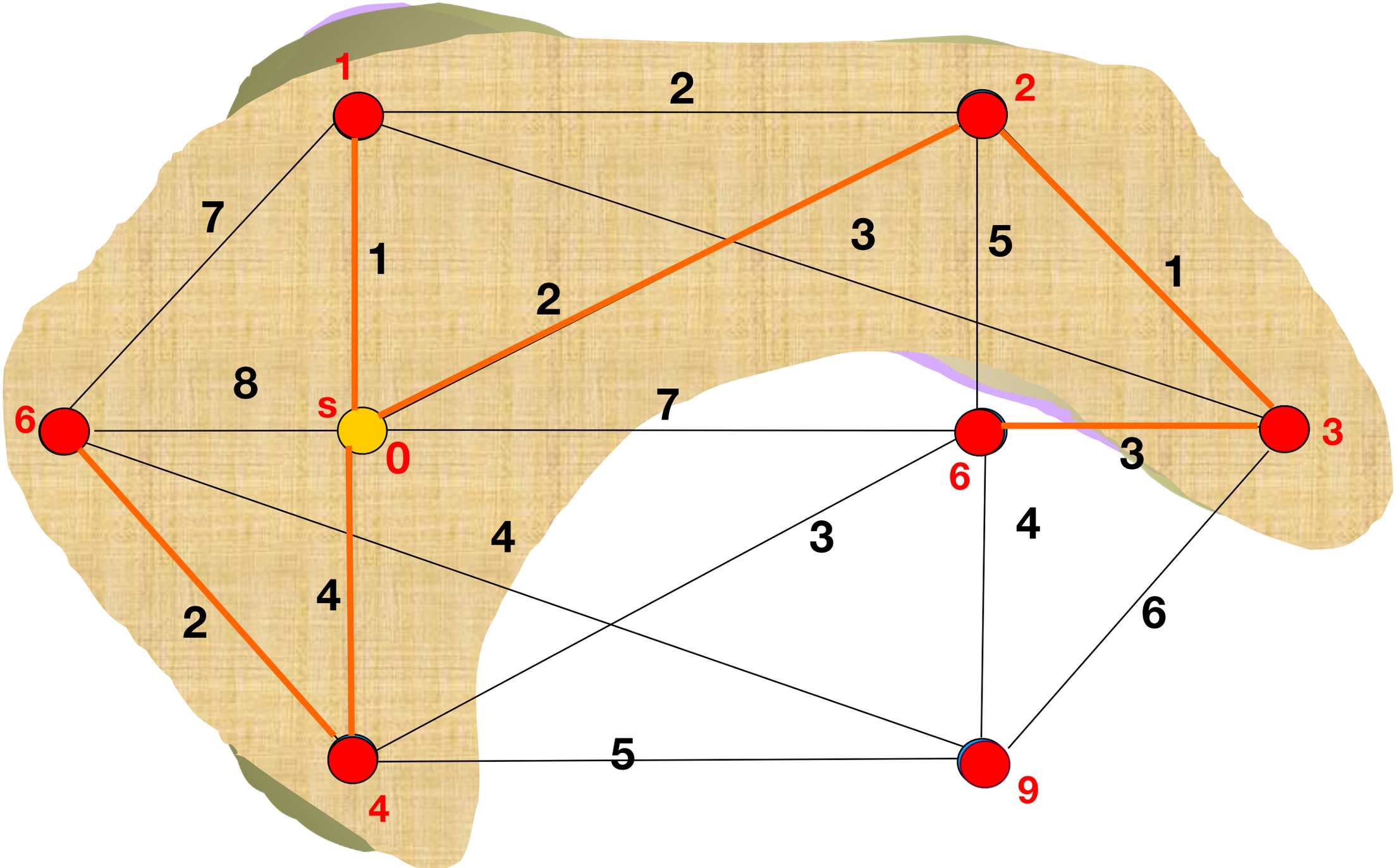
Dijkstra's Algorithm



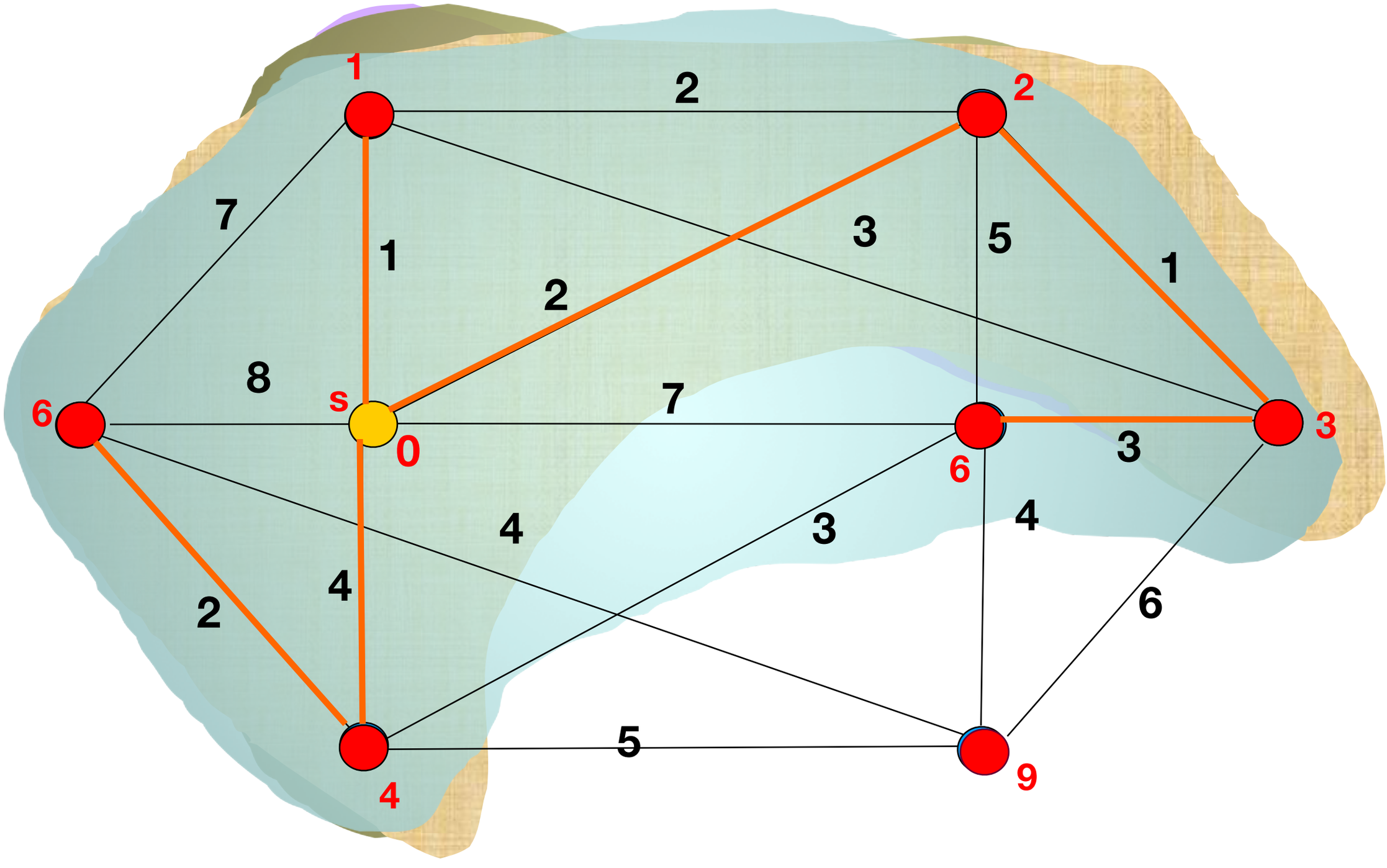
Dijkstra's Algorithm



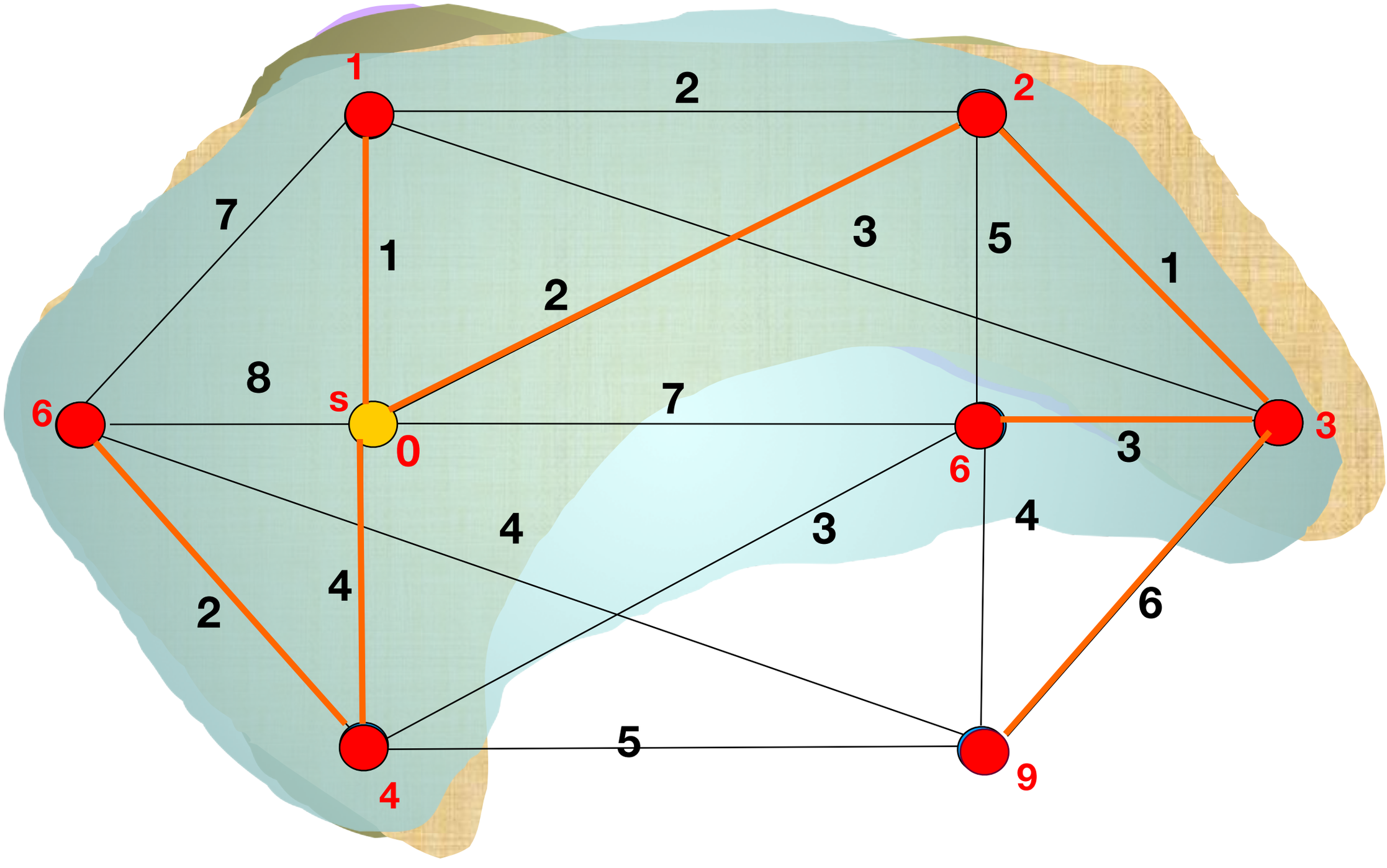
Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm



Priority Queue-based Implementation

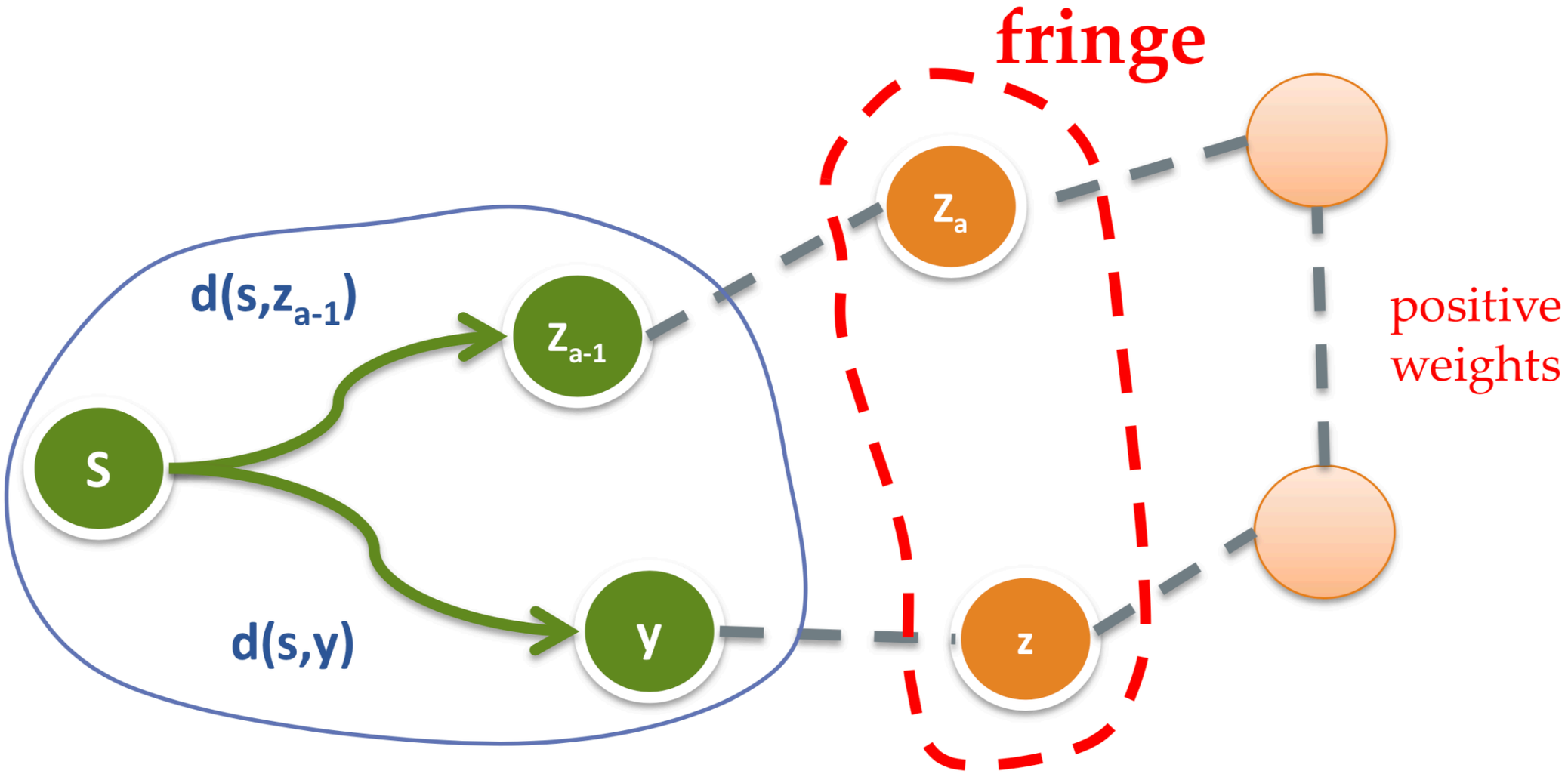
Shortest Paths

```
Void shortestPaths(EdgeList[] adjInfo, int n, int s,
int[] parent, float[]fringeWgt)

    int[] status = new int[n+1];
    MinPQ pq = create(n, status, parent, fringeWgt);

insert(pq, s, -1, 0);
while(isEmpty(pq)==false)
    int v = getMin(pq);
    deleteMin(pq);
    updateFringe(pq, adjInfo[v], v);
```

```
void updateFringe(MinPQ pq, EdgeList
adjInfoOfV, int v)

    float myDist = pq.fringeWgt[v];
    EdgeList remAdj;
    remAdj = adjInfoOfV;
    while{remAdj != nil}
        EdgeInfo wInfo = first(remAdj);
        int w = wInfo.to;
        float newDist = myDist + wInfo.weight;
        if(pq.status[w]==unseen)
            insert(pq,w,v,newDist);
        else if(pq.status[w] = fringe)
            if(newDist < getPriority(pq,w))
                decreaseKey(pq,w,v,newDist);
        remAdj = rest(remAdj);
return;
```

Correctness of Dijkstra Algorithm

- $W(s->y->z) < W(s->z_{a-1}->z_a->z)$



The Dijkstra Skeleton

- **Single-source shortest path (SSSP)**
- **SSSP + node weight constraint**
  - E.g. in routing
    - Each router has its cost (node cost)
    - Each route has its cost (edge cost)
- **SSP + capacity constraint**
  - The “pipe problem”
    - Maximize the min edge weight
  - The “electric vehicle problem”
    - Minimize the max edge weight

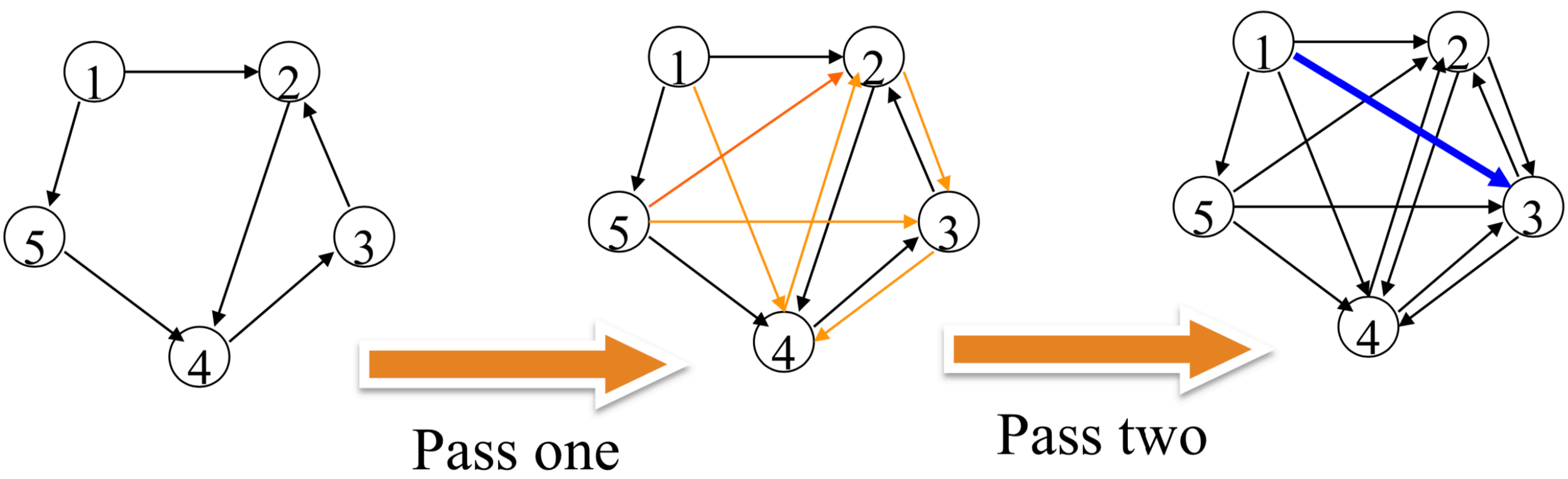
“Dijkstra Skeleton”

All-pairs Shortest Paths

- **For all pair of vertices in a graph, say, u, v:**
  - Is there a path from u to v?
  - What is the shortest path from u to v?
- **Reachability as a (reflexive) transitive closure of the adjacency relation**
  - Which can be represented as a bit matrix

Transitive Closure by Shortcuts

- The idea: if there are edges $s_i s_k$, $s_k s_j$, then an edge $s_i s_j$, the "shortcut" is inserted.



Shortcut Algorithm

- **Input: A, an n×n boolean matrix that represents a binary relation**
- **Output: R, the boolean matrix for the transitive closure of A**
- **Procedure**
  - void simpleTransitiveClosure(**boolean**[][] A, **int** n, **boolean**[][] R)
  - **int** i,j,k;
  - Copy A to R;
  - Set all main diagonal entries, r_{ii}, to true;
  - **while**(any entry of R changed during one complete pass)
  - **for**(i=1; i≤n; i++)
  - **for**(j=1; j≤n; j++)
  - **for**(k=1; k≤n; k++)
  - $r_{ij}=r_{ij}\vee(r_{ik}\wedge r_{kj})$

$O(n^4)$

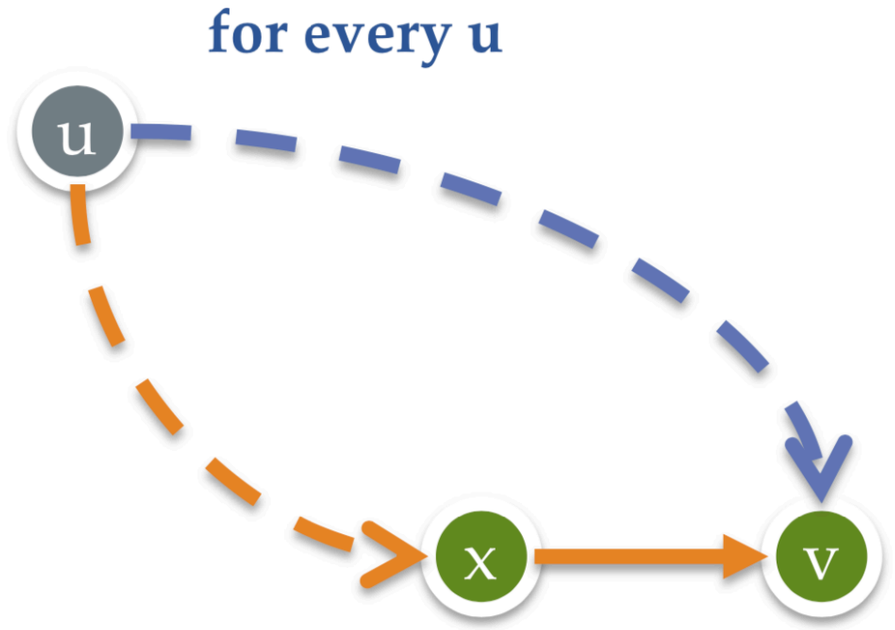
The order of (i,j,k) matters

Another Way to Add Shortcuts

- **Enumerate all edges (x,v)**
  - v as the destination
  - Enumerate all possible sources u

while any entry of R changed
  for all vertices u
    for every edge (x,v)
      $r_{uv}=r_{uv}\vee(r_{ux}\wedge r_{xv})$

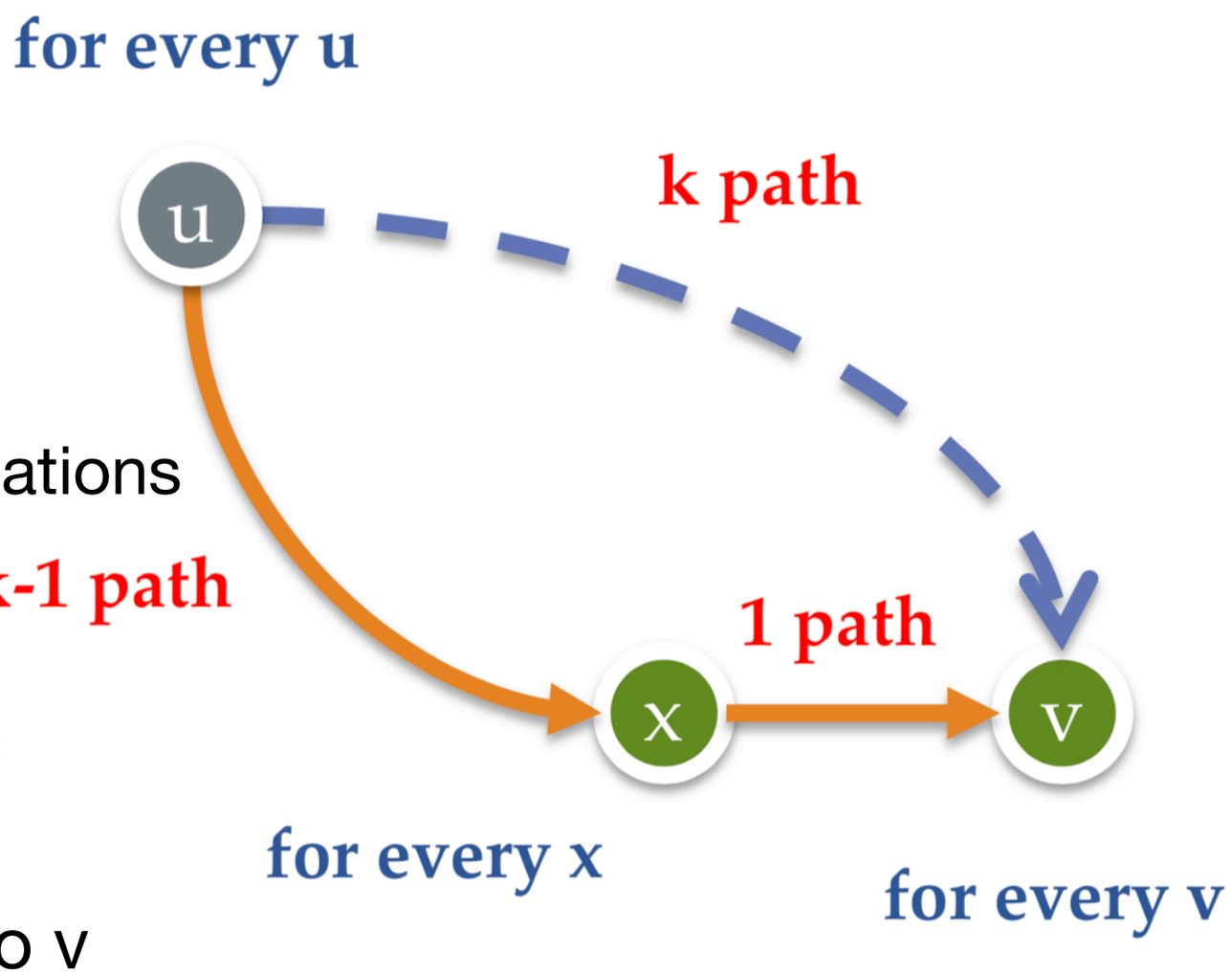
$O(n^2m)$



n-1 round iteration

Length of the Path

- **Recursion**
  - Reachable via at most k edges
- **Enumeration**
  - Enumerate all path length
  - Enumerate all sources and destinations



for k=1 to n-1
  for all vertices u
    for all vertices v
      for all vertices x pointing to v
        $r^k_{uv}=r^{k-1}_{uv} \vee (r^{k-1}_{ux} \wedge r_{xv})$

$r_{uv}=1$

$O(n^4)$

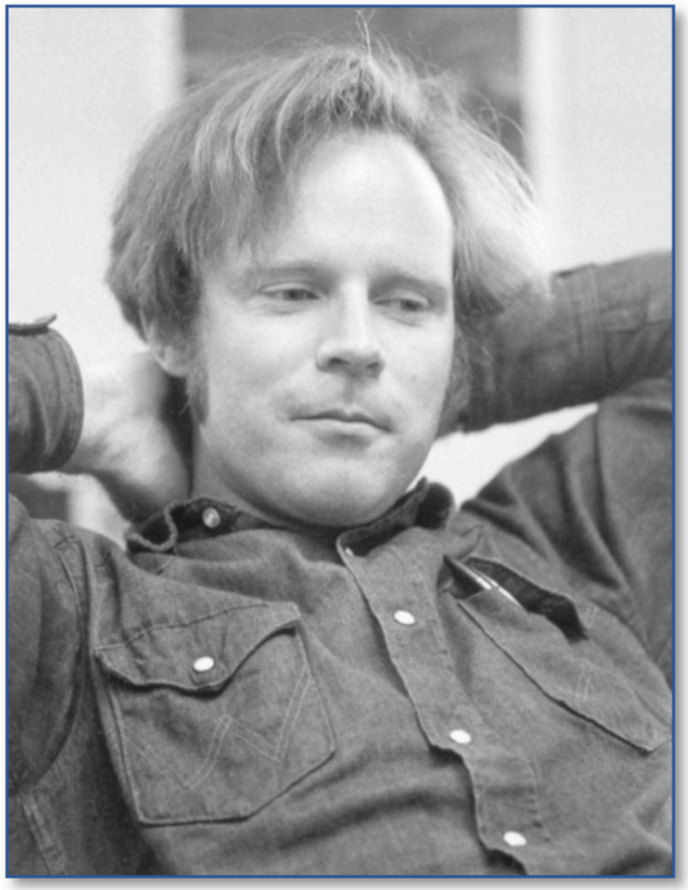
Floyd's Lemma

组合问题的优良算法具有巨大回报，这个事实激励了技术水平的突飞猛进。…… 大约从1970年起，计算机科学家们经历了所谓的‘**Floyd引理**’现象：看似需用n^3次运算的问题实际上可能用$O(n^2)$次运算就能求解，看似需用n^2次运算的问题实际上可能用O(nlogn)次运算就能处理，而且nlogn通常还可以减少到O(n)。一些更难的问题的运行时间也从$O(2^n)$减少到$O(1.5^n)$，再减少到$O(1.3^n)$，等等。

- Knuth, Volumn4A, TAOCP

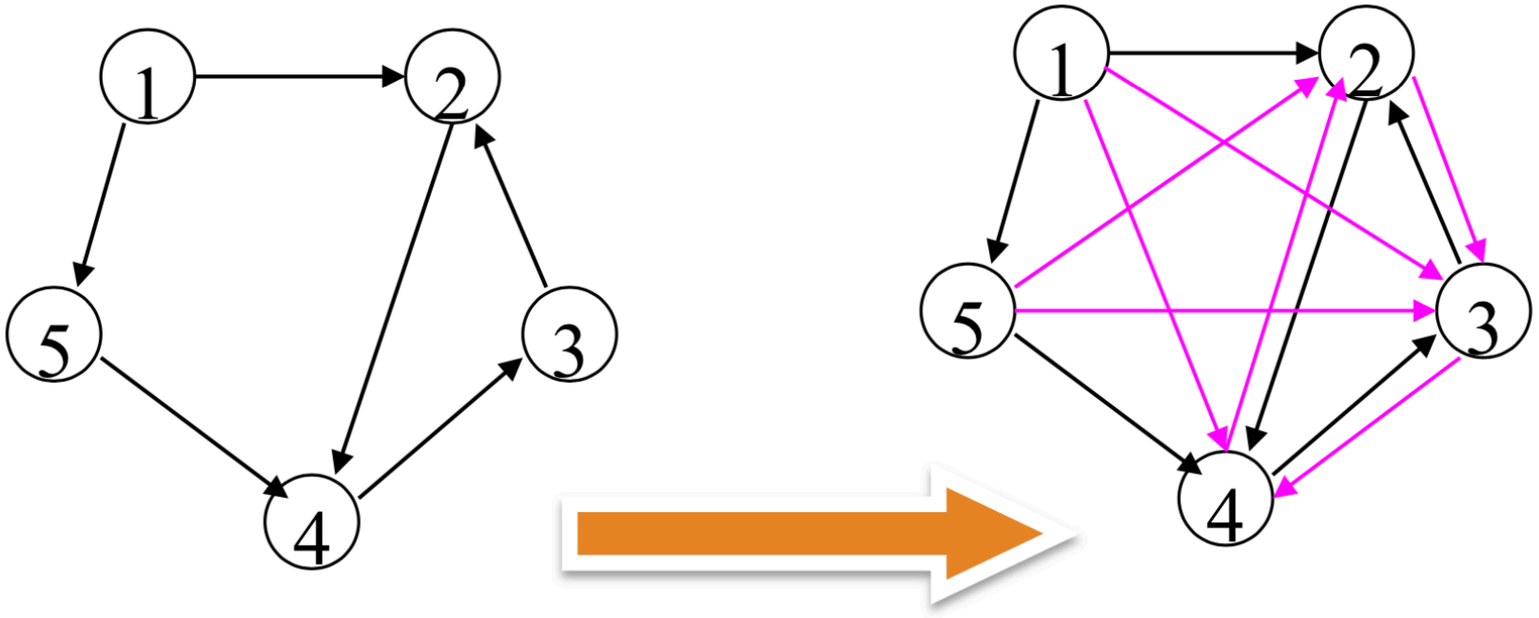
Robert W Floyd, In Memoriam

by Donald E. Knuth, Stanford University



Shortcuts in Different Order

- Duplicated checking may be deleted by changing the order of the vertices.



Pass one

Check the vertices in decreasing order.

No edge is added in Pass two. End.

Change the Order: The Warshall Algorithm

- void simpleTransitiveClosure(**boolean**[][] A, **int** n, **boolean**[][] R)
- **int** i,j,k;
- Copy A to R;
- Set all main diagonal entries, r_{ii}, to true;
- ~~**while**(any entry of R changed during one complete pass)~~
- **for**(k=1; k≤n; i++)
- **for**(i=1; i≤n; j++)
- **for**(j=1; j≤n; j++)
- $r_{ij}=r_{ij}\vee(r_{ik}\wedge r_{kj})$

k Varys in the outmost loop

Note: “false to true” can not be reversed

Why the Floyd-Warshall Algorithm Works

- **<k,i,j> or <i,j,k>**
  - The order matters
  - That’s why Dijkstra fails

Correctness of the Warshall Algorithm

- **Notation:**
  - The value of rij changes during the execution of the body of the “for k…” loop
    - After initializations: $r_{ij}^{(0)}$
    - After the k^{th} time of execution: $r_{ij}^{(k)}$

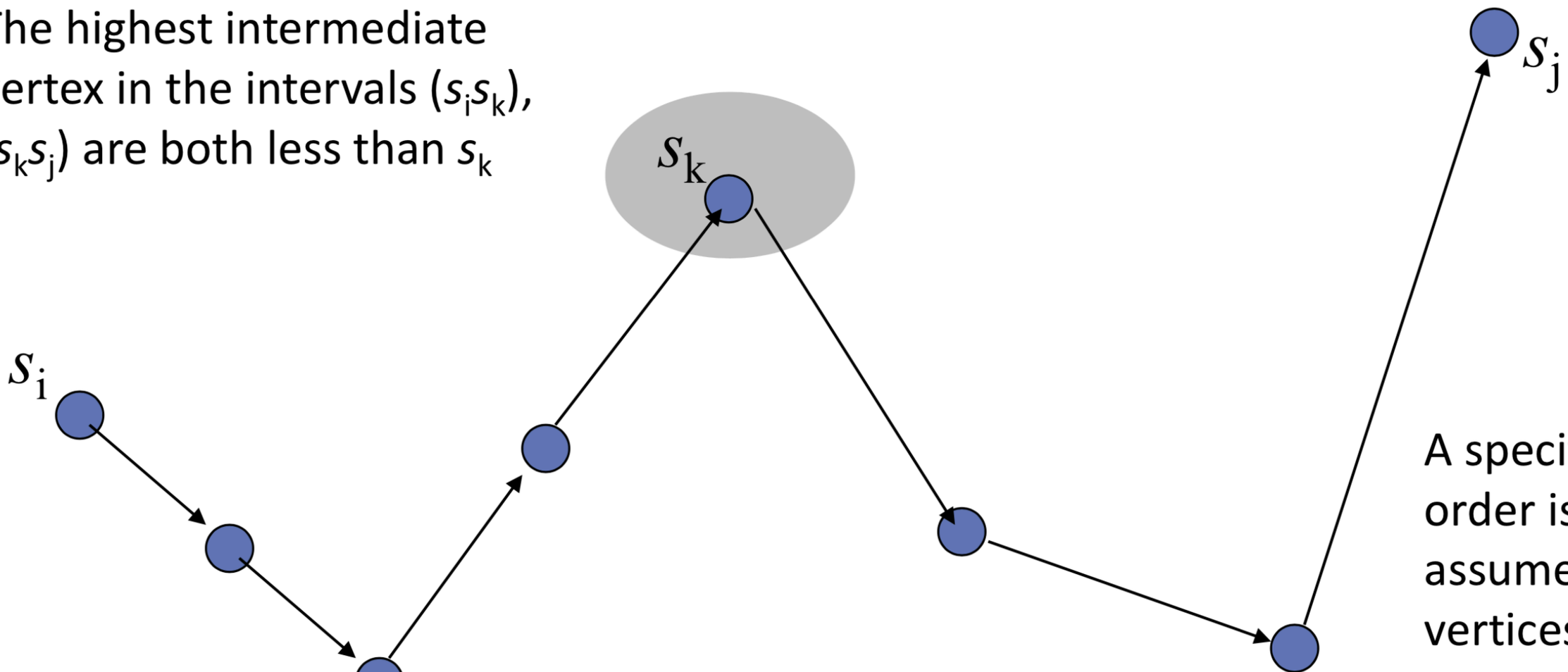
Correctness of the Warshall Algorithm

- If there is a simple path from s_i to s_j(i!=j) for which the highest-numbered intermediate vertex is s_k, then $r_{ij}^{(k)}$=true.

- Proof by induction:
  - Base case: $r_{ij}^{(0)}$=true if and only if $s_i s_j \in E$
  - Hypothesis: the conclusion holds for h<k(h≥0)
  - Induction: the simple $s_i s_j$-path can be looked as $s_i s_k$-path+$s_k s_j$-path, with the indices h_1, h_2 of the highest-numbered intermediate vertices of both segment **strictly**(simple path) less than k. So, $r_{ik}^{(h1)}$=true, $r_{kj}^{(h2)}$=true, then $r_{ik}^{(k-1)}$=true, $r_{kj}^{(k-1)}$=true(Remember, false to true can not be reversed). So, $r_{ij}^{(k)}$=true

Highest-numbered Intermediate Vertex

The highest intermediate vertex in the intervals ($s_i s_k$), ($s_k s_j$) are both less than s_k



A specific order is assumed for all vertices

Vertical position of vertices reflect their vertex numbers

$$r_{ij}=r_{ij}\vee(r_{ik}\wedge r_{kj})$$

Correctness of the Warshall Algorithm

- If $r_{ij}^{(k)}$=true, then there is a $(s_i, s_j)^{(k)}$ path
- Proof
  - If $r_{ij}^{(0)}$=true, then there is $(s_i,s_j)^{(0)}$ path
  - If r_{ij} first become true in round k, then
    - $r_{ik}^{(k-1)}$ = true, $r_{kj}^{(k-1)}$ = true
  - We have a "s_i->s_k->s_j" path
    - Intermediate nodes in {1, 2, ..., k-1} ∪ {k}

All-pairs Shortest Paths

- **Shortest path property**
  - If a shortest path from x to z consisting of path P from x to y followed by path Q from y to z. Then P is a shortest xy- path, and Q, a shortest yz-path.
  - The regular matrix representing a graph can easily be transformed into a (minimum) distance matrix D

(just replacing 1 by edge weight, 0 by infinity, and setting main diagonal elements as 0)

Computing the Distance Matrix

- **Basic formula:**
  - $D^{(0)}[i][j]=w_{ij}$
  - $D^{(k)}[i][j]=\min(D^{(k-1)}[i][j], D^{(k-1)}[i][k]+ D^{(k-1)}[k][j])$
- **Basic property:**
  - $D^{(k)}[i][j] = d_{ij}^{(k)}$
    - where $d_{ij}^{(k)}$ is the weight of a shortest path from v_i to v_j with highest numbered intermediate vertex v_k.

All-pairs Shortest Paths

- **Floyd algorithm**
  - Only slight changes on Washall's algorithm.

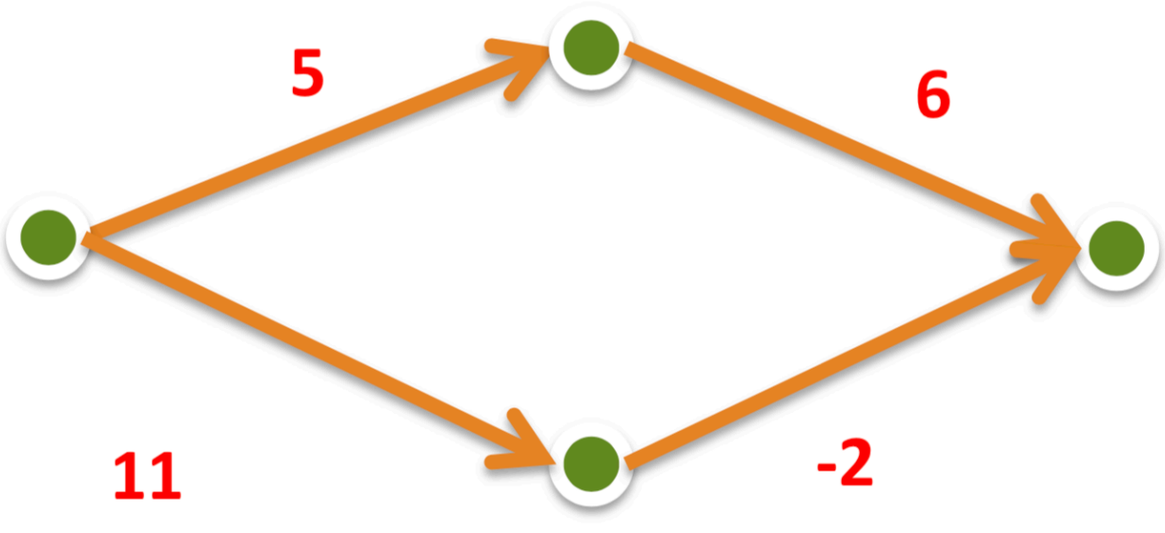
```
Void allPairsShortestPaths(float [][] W, int n, float [][] D)
    int i, j, k;
    Copy W into D;
    for (k=1; k≤n; k++)
        for (i=1; i≤n; i++)
            for (j=1; j≤n; j++)
                D[i][j] = min (D[i][j], D[i][k]+D[k][j]);
```

All-pairs Shortest Paths

- **Construction of the routing table**
  - Forward, backward
- **APSP + capacity constraints**
  - The pipeline problem
  - The electric vehicle problem

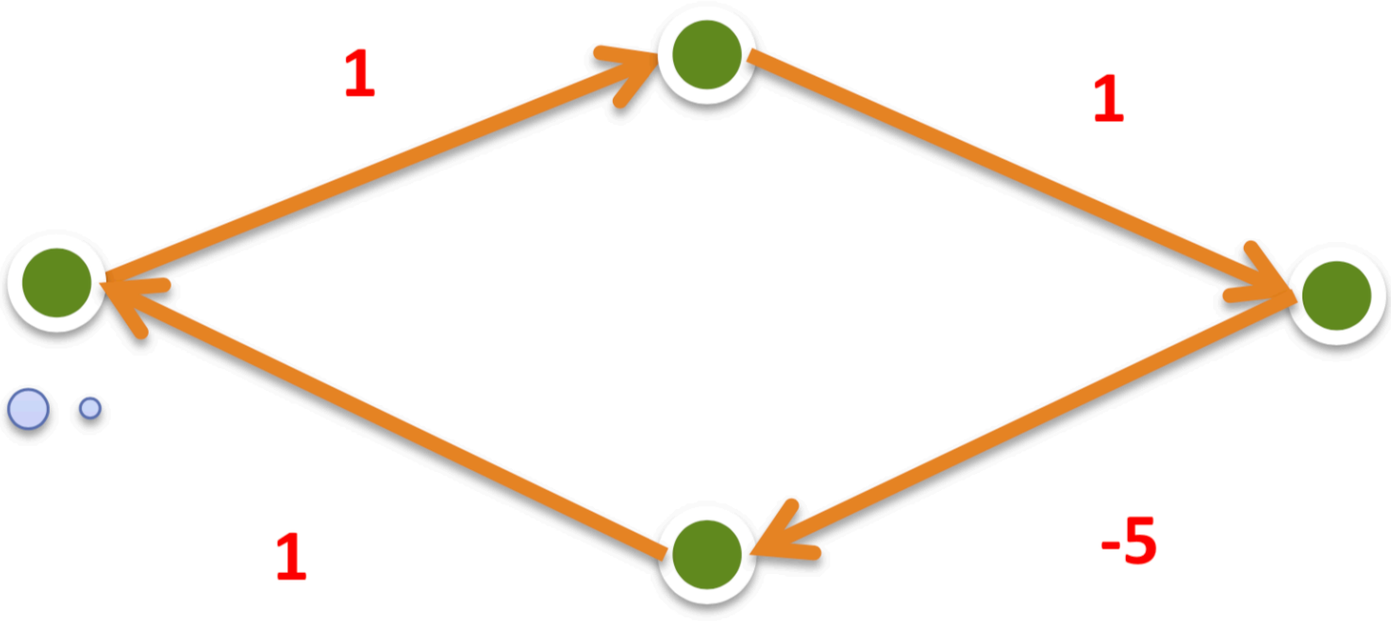
Floyd algorithm => Floyd skeleton

Negative Weight



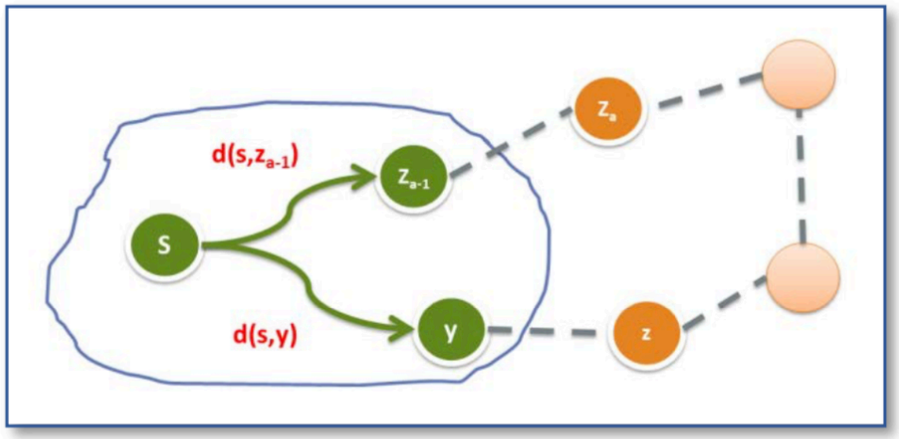
"shortest path" is not well defined

Negative weight cycle

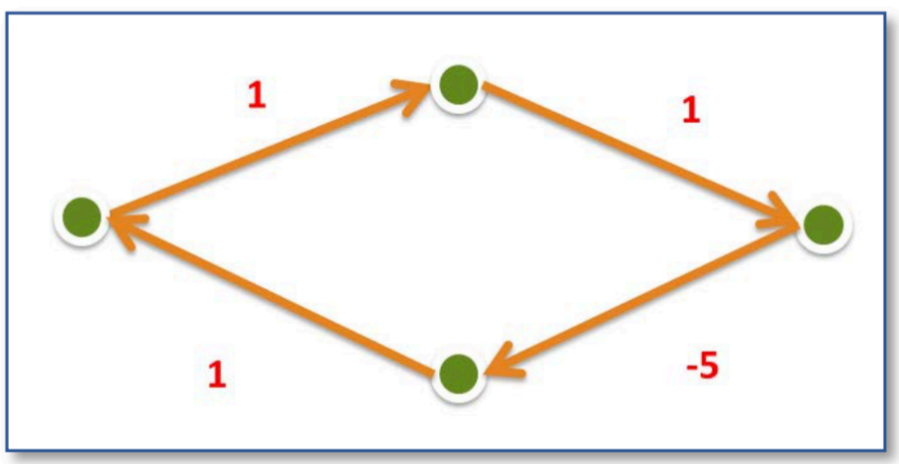


Negative Weight

- **Can the shortest path algorithm work correctly?**
  - Dijkstra's algorithm
    - No negative weight edge



  - Floyd's algorithm
    - No negative weight cycle



Matrix Representation

- **Define family of matrix $A^{(p)}$:**
  - $a_{ij}^{(p)}$=*true* if and only if there is a path of length p from s_i to s_j.
- $A^{(0)}$ is specified as identity matrix. $A^{(1)}$ is exactly the adjacency matrix.
- Note that $a_{ij}^{(2)}$=*true* if and only if exists some s_k, such that both $a_{ik}^{(1)}$ and $a_{kj}^{(1)}$ are *true*. So, $a_{ij}^{(2)}=\vee_{k=1,2,...,n}(a_{ik}^{(1)}\wedge a_{kj}^{(1)})$, which is an entry in the *Boolean matrix product*.

Boolean Matrix Operations

- **Boolean matrix product *C=AB* as:**
  - $c_{ij}=\vee_{k=1,2,...,n}(a_{ik}\wedge b_{kj})$
- **Boolean matrix sum *D=A+B* as:**
  - $d_{ij}=a_{ij}\vee b_{ij}$
- ***R*, the transitive closure matrix of *A*, is the sum of all A^p, *p* is a non-negative integer.**
- **For a digraph with *n* vertices, the length of the longest simple path is no larger than *n*-1.**

Bit Matrix

- A bit string of length *n* is a sequence of *n* bits occupying contiguous storage(word boundary) (usually, *n* is larger than the word length of a computer)

- If *A* is a bit matrix of *n*×*n*, then *A*[*i*] denotes the *i*th row of *A* which is a bit string of length *n*. a_{ij} is the *j*th bit of *A*[*i*].

- The procedure bitwiseOR(a,b,n) compute a∨b bitwise for *n* bits, leaving the result in *a*.

Straightforward Multiplication of Bit Matrix

- **Computing *C=AB***
  - <Initialize *C* to the zero matrix>
  - **for** (i=1; i≤*n*, i++)
  - **for** (k=1; k≤*n*, k++)
  - **if** (a_{ik}==*true*) bitwiseOR(*C*[*i*], *B*[*k*], *n*)

Thought as a union of sets (row union), n^2 unions are done at most

In the case of a_{ik} is true, $c_{ij}=a_{ik}b_{kj}$ is true iff. b_{kj} is true. As a result: $C[i]=\cup_{k\in A[i]}B[k]$, $(A[i]=\{k|a_{ik}=true\})$

Union for B[k] is repeated each time when the kth bit is true in a different row of A is encountered.

Reducing the Duplicates by Grouping

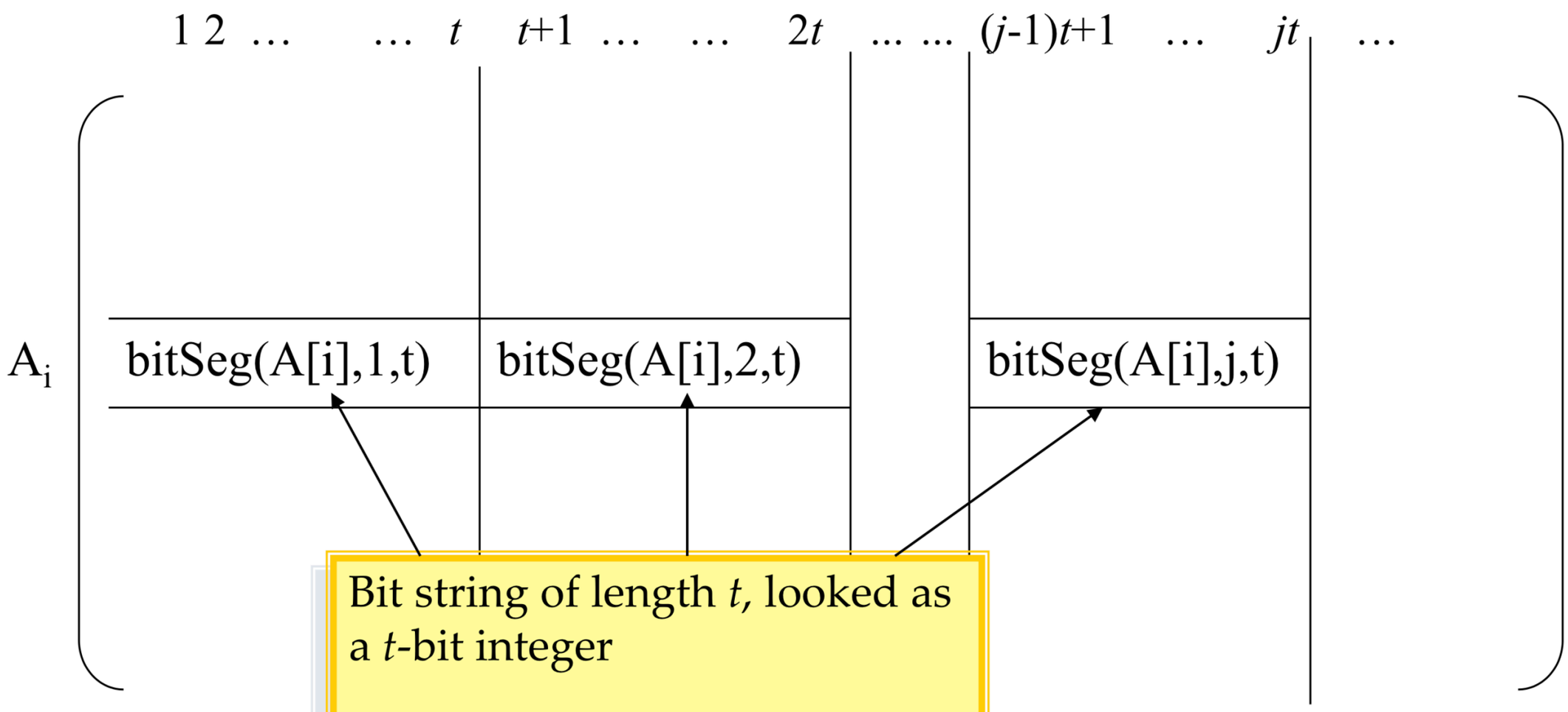
- **Multiplication of A, B, two 12 x 12 matrices**

$$
\begin{array}{c}
A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \\ A_8 \\ A_9 \\ A_{10} \\ A_{11} \\ A_{12}
\end{array}
\begin{pmatrix}
1\,0\,1\,1 & 0\,1\,0\,1\,0\,0\,0\,1 \\
 & \\
1\,0\,1\,1 & 1\,0\,0\,1\,1\,0\,1\,1 \\
 & \\
 & \\
 & \\
 & 0\,1\,0\,1 \\
1\,0\,1\,1 & 1\,0\,0\,1\,1\,1\,1\,0 \\
 & \\
 & \\
 & \\
 & \\
 &
\end{pmatrix}
$$

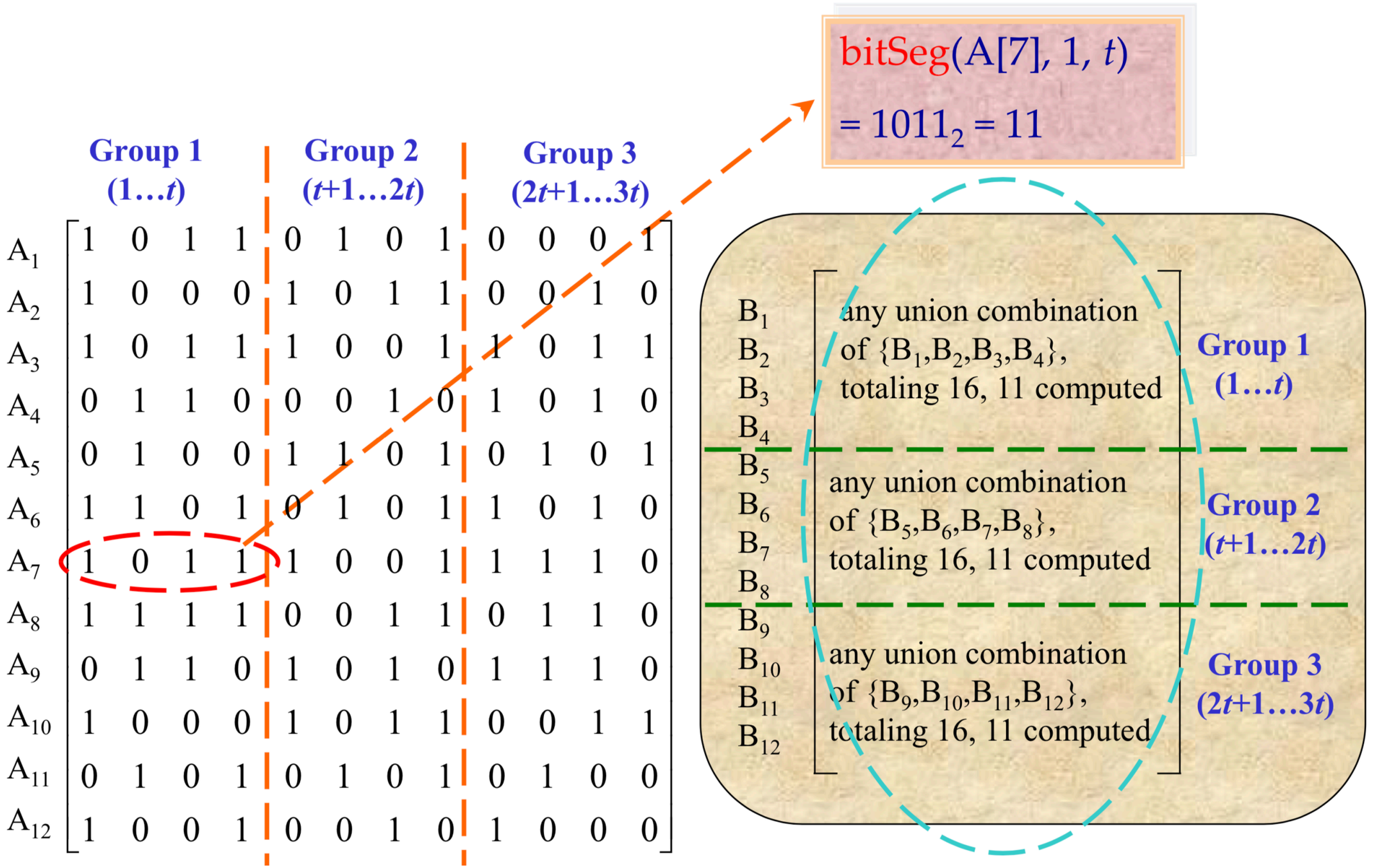
Segment Length t

- 12 rows of B are divided evenly into 3 groups, with rows 1-4 in group 1, etc.
- With each group, all possible unions of different rows are pre-computed. (This can be done with 11 unions if suitable order is assumed.)
- When the first row of AB is computed, $(B[1] \cup B[3] \cup B[4])$ is used in stead of 3 different unions, and this combination is used in computing the 3rd and 7th rows as well.

The Segmentation for Matrix A



An Example


bitSeg(A[7], 1, t)
= 1011₂ = 11
Group 1 (1…t)
Group 2 (t+1…2t)
Group 3 (2t+1…3t)
A1 1 0 1 1 0 1 0 1 0 0 0 1
A2 1 0 0 0 1 0 1 1 0 0 1 0
A3 1 0 1 1 1 0 0 1 1 0 1 1
A4 0 1 1 0 0 0 1 0 1 0 1 0
A5 0 1 0 0 1 1 0 1 0 1 0 1
A6 1 1 0 1 0 1 0 1 1 0 1 0
A7 1 0 1 1 1 0 0 1 1 1 1 0
A8 1 1 1 1 0 0 1 1 0 1 1 0
A9 0 1 1 0 1 0 1 0 1 1 1 0
A10 1 0 0 0 1 0 1 1 0 0 1 1
A11 0 1 0 1 0 1 0 1 0 1 0 0
A12 1 0 0 1 0 0 1 0 1 0 0 0
t=4
B1 B2 B3 B4
any union combination of {B1,B2,B3,B4}, totaling 16, 11 computed
Group 1 (1…t)
B5 B6 B7 B8
any union combination of {B5,B6,B7,B8}, totaling 16, 11 computed
Group 2 (t+1…2t)
B9 B10 B11 B12
any union combination of {B9,B10,B11,B12}, totaling 16, 11 computed
Group 3 (2t+1…3t)

Storage of the Row Combinations

- **Using one large 2-dimensional array**
- **Goals**
  - keep all unions generated
  - provide indexing for using
- **Coding with in a group**
  - One-to-one correspondence between a bit string of length *t* and one union for a subset of a set of *t* elements
- **Establishing indexing for union required**
  - When constructing a row of *AB*, a segment can be notated as a integer. Use it as index.

Storage the Unions

allUnion

one row for
one group

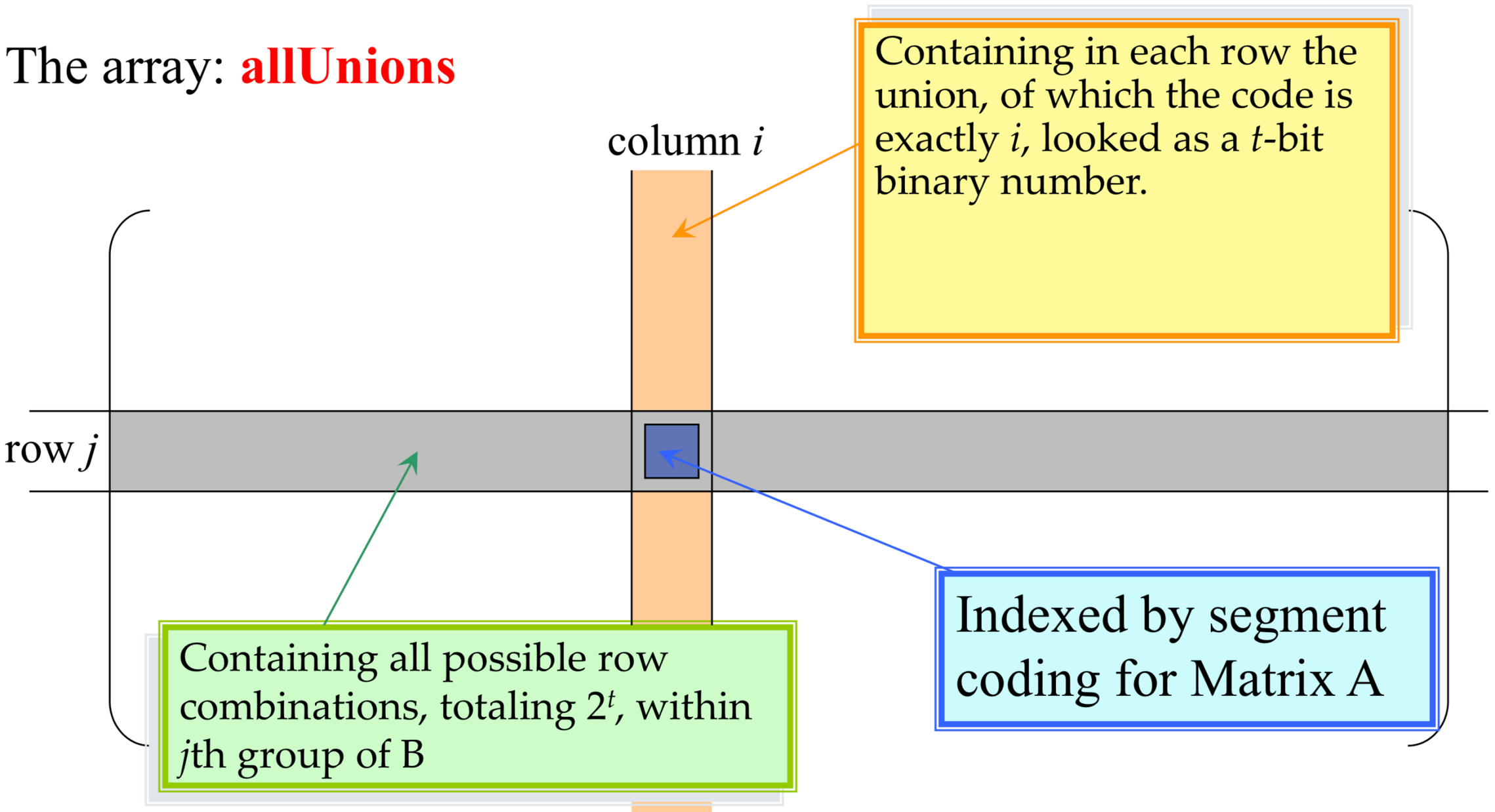
column indexed by bitSeg(A[i],j,t)

$$\begin{bmatrix} \phi & 4 & 3 & 3,4 & 2 & 2,4 & 2,3 & 2,3,4 & 1 & 1,4 & 1,3 & 1,3,4 & 1,2 & 1,2,4 & 1,2,3 & 1,2,3,4 \\ \phi & 8 & 7 & 7,8 & 6 & 6,8 & & & & & & & & & & \\ \phi & 12 & 11 & 11,12 & 10 & 10,12 & & & & & & & & & & \end{bmatrix}$$

i,j,k stands for $B_i \cup B_j \cup B_k$

Array for Row Combinations

The array: **allUnions**



Cost as Function of Group Size

- **Cost for the pre-computation**
  - There are 2^t different combination of rows in one group, including an empty and t singleton. Note, in a suitable order, each combination can be made using only one union. So, the total number of union is **$g[2^t-(t+1)]$**, where g=*n*/t is the number of group.
- **Cost for the generation of the product**
  - In computing one of *n* rows of *AB*, at most one combination from each group is used. So, the total number of union is ***n*(g-1)**

Selecting Best Group Size

- **The total number of union done is:**
  - $g[2^t-(t+1)]+n(g-1) \approx (n2^t)/t+n^2/t$ (Note: $g=n/t$)
- **Trying to minimize the number of union**
  - Assuming that the first term is of higher order:
    - Then $t \geq \lg n$, and the least value is reached when $t=\lg n$.
  - Assuming that the second term is of higher order:
    - Then $t \leq \lg n$, and the least value is reached when $t=\lg n$.
- **So, when $t \approx \lg n$, the number of union is roughly $2n^2/\lg n$, which is of lower order than n^2. We use $t=\lfloor \lg n \rfloor$**

Sketch for the Procedure

- t=⌊lg*n*⌋; g=⌈*n*/*t*⌉;
- <Compute and store in allUnions unions of all combinations of rows of *B*>
- for (i=1; i≤*n*; i++)
  - <Initialize C[i] to 0>
  - for (j=1; j≤*g*; j++)
    - C[i] = C[i] ∪ allUnions[j][bitSeg(A[i],j,t)]

Kronrod Algorithm

- Input: A,B and n, where A and B are n×n bit matrices.
- Output: C, the Boolean matrix product.
- Procedure
  - The processing order has been changed, from “row by row” to “group by group”, resulting the reduction of storage space for unions.

Complexity of the Kronrod Algorithm

- For computing all unions within a group, $2^t - 1$ union operations are done.

- One union is bitwiseOR'ed to n row of C

- So, altogether, $(n/t)(2^t-1+n)$ row unions are done.

- The cost of row union is $\lceil n/w \rceil$ bitwise or operations, where w is word size of bitwise or instruction dependent constant.

Thank you!

Q & A