גרפים 1.5

Ido Talbi ; *University of Haifa*

מה נלמד היום

- נראה הרחבות לגרף הסטנדרטי שאנו מכירים
- נתעסק ב:
  - גרפים מכוונים
  - הוספת פונקציית משקל $w : E \to ??$

חזרה על השיעור הקודם

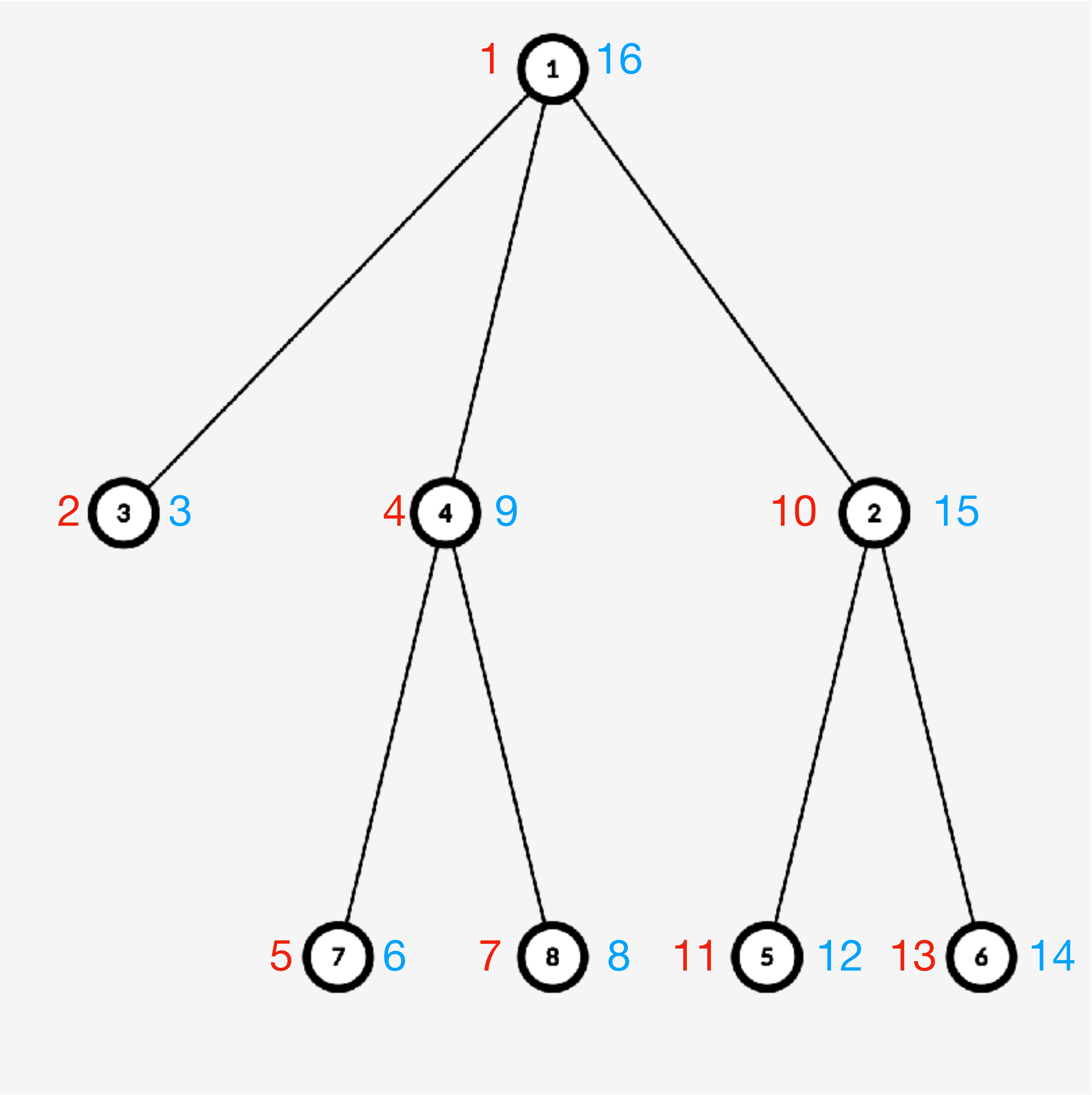
גרפים, ייצוג בקוד, וכו'

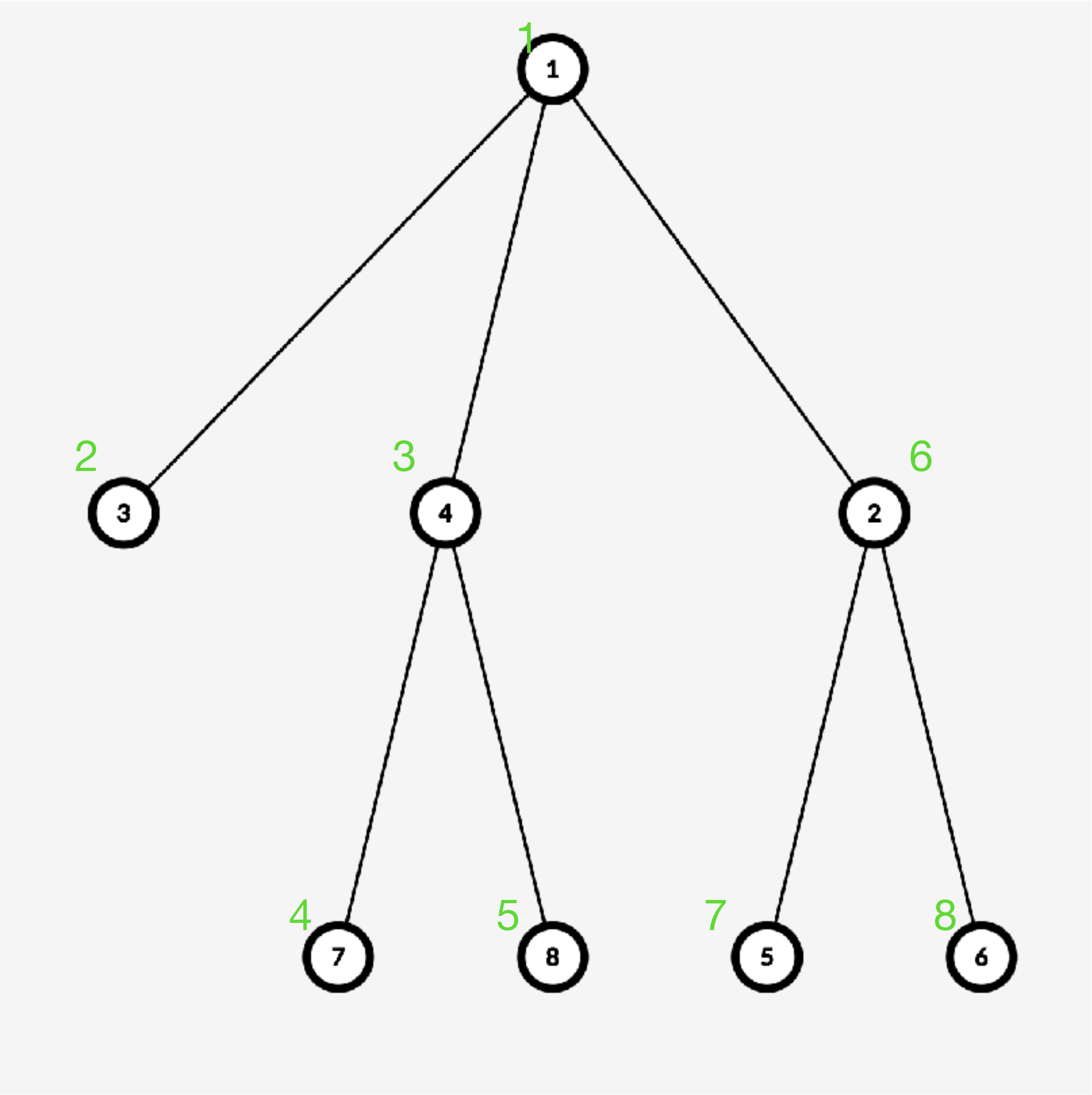
- גרף הוא זוג של קבוצות: $G = (V, E)$ כאשר V קבוצת הקודקודים (לרוב אצלנו $V = [1,n]$), ו-E קבוצת הקשתות (וקשת מכוונת הינה הזוג (u, v)).
- ישנן 2 שיטות (פופולאריות) לייצוג גרפים בקוד: **רשימת שכנויות** ו-**מטריצת שכנויות**. לרוב נשתמש ברשימת שכנויות מכיוון שלא צריך הרבה זכרון כדי להחזיק גרף.
- ברשימת שכנויות נשמור מערך בגודל n, וכל כניסה בו תהיה רשימה (או מערך דינאמי). כקוד, נשמור `vector<vector<int>>` (או לפעמים `vector<int>[maxn]`).
- לדוגמה, הכניסה $\mathrm{g}[v]$ תהיה הרשימה של כל הקודקודים השכנים לv (כלומר, קיימת קשת (v, x) כאשר $x \in \mathrm{g}[v]$).

אלגוריתמי חיפוש: DFS ו-BFS

- חיפוש לעומק ולרוחב בגרף. זמן הפעולה של אלגוריתמים אלו הינו ליניארי ולכן לרוב ניתן להריץ אותם על גרפים מספר קבוע של פעמים.
- אלגוריתם ה-BFS מאפשר לנו לדעת מרחקים של כל הצמתים מצומת קבועה, בעוד ש-DFS מאפשר לנו לדעת דברים על מבנה הגרף, לדוגמה: רכיבי קשירות, איבר בתת עץ, דו-צדדיות ועוד.
- נזכר בריצה ובמימוש של האלגוריתמים:

DFS

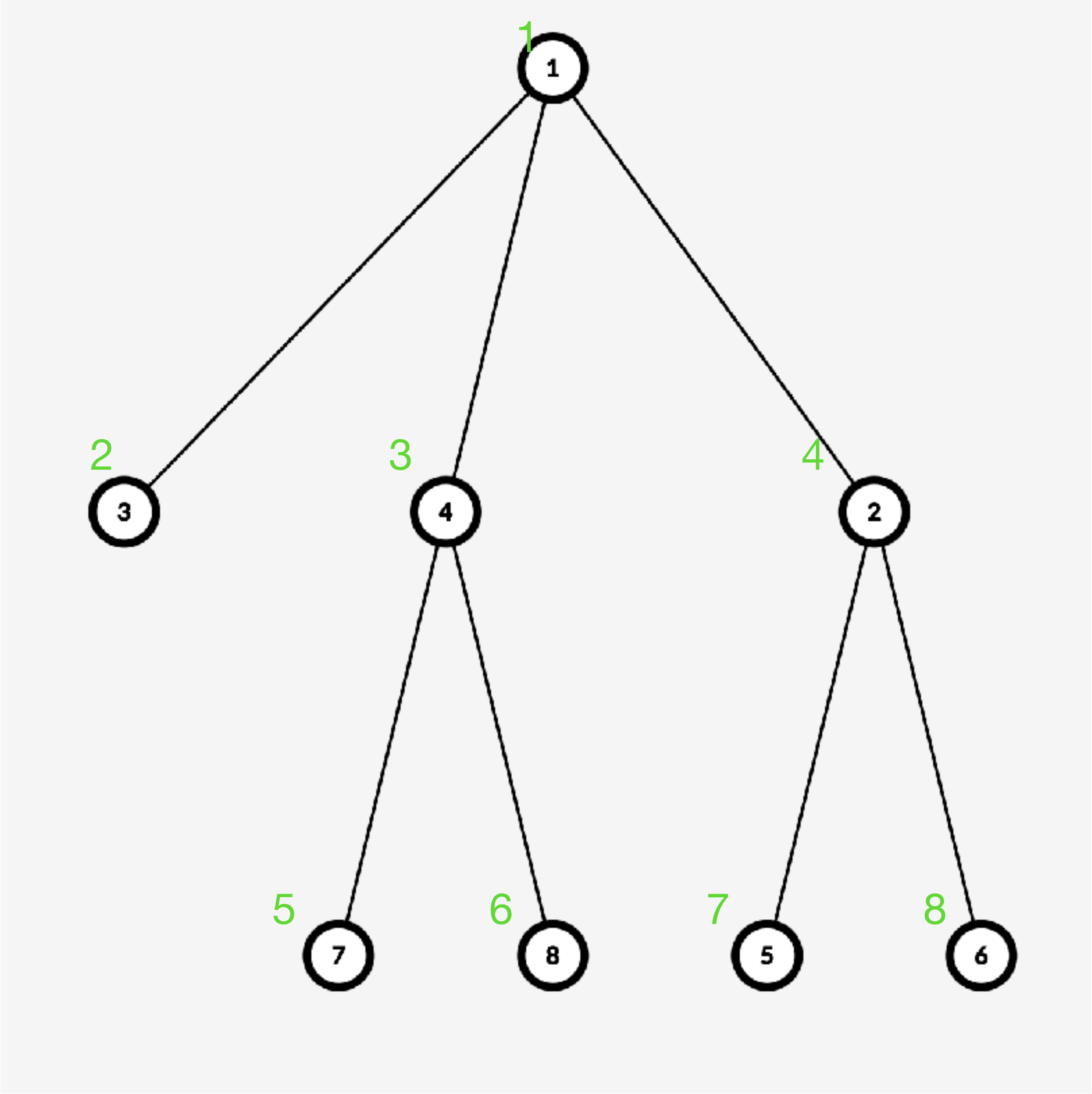




DFS

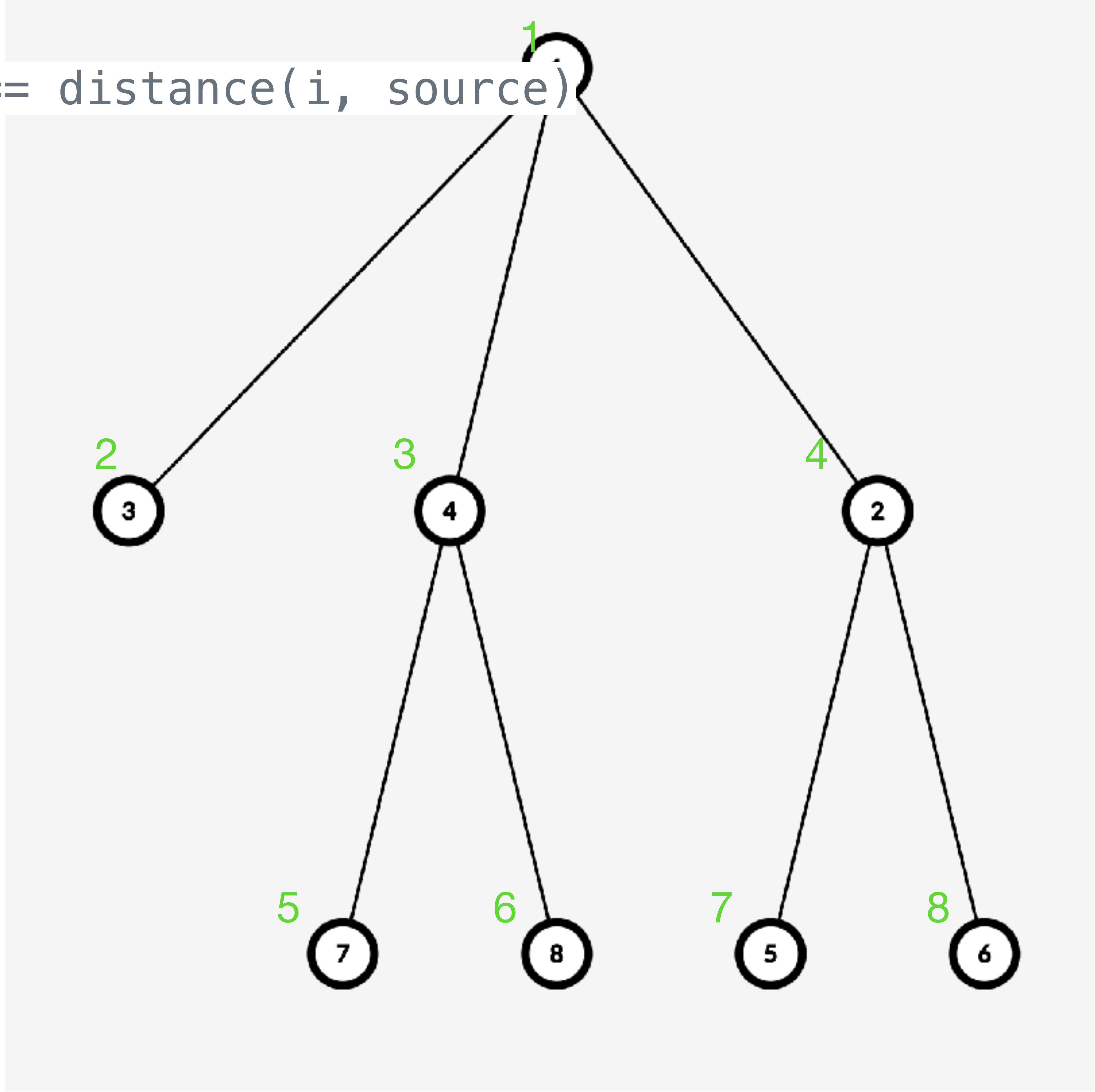
```
vector<vector<int>> graph;
int time = 1;
int in_time[maxn] = {0};
int out_time[maxn] = {0};

void DFS(int u) {
    if(in_time[u]) return; // already visited u

    in_time[u] = time++;
    for(auto &v : graph[u]) DFS(v);
    out_time[u] = time++;
}
```

BFS



BFS

```
vector<int> bfs(int source) {
    queue<int> q;
    vector<int> distances(n, 0); // distances[i] == distance(i, source)
    vector<bool> visited(n, false);
    q.push(source);
    visited[source] = true;

    while(!q.empty()) {
        int u = q.front();
        q.pop();
        for(auto v : graph[u]) {
            if(!visited[v]) {
                visited[v] = true;
                distances[v] = distances[u] + 1;
                q.push(v);
            }
        }
    }

    return distances;
}
```



גרף מכוון

גרף מכוון

- נזכר, גרף מוגדר כזוג $G = (V, E)$. במקרה והגרף מכוון אז הקבוצה E תראה כך:

$$E = \{(u, v) \mid \text{edge from } u \text{ to } v\}$$

- כאשר הדגש הוא על כך שקשת היא זוג סדור ולא קבוצה בגודל 2.
- בגלל שעכשיו היחס "exists path between u, v" $\equiv$ $\hookrightarrow$ אינו סימטרי אזי יש משמעות נוספת ל"קיים מסלול בין u לv"

גרף מכוון: רכיבי קשירות

- נגדיר כעת יחס חדש:

$$\mapsto\; = \{(u,v) \mid u \hookrightarrow v \wedge v \hookrightarrow u\} \subseteq E^2$$

- היחס $\mapsto$ הוא יחס שקילות ולכן נתעניין במחלקות שקילות שלו. כפי שאפשר לנחש, לכל מחלקה כזו קוראים **רכיב קשירות**, ולפי ההגדרה של מחלקת שקילות לכל שני צמתים u, v אשר נמצאים באותה מחלקה קיים מסלול בינהם.

- שימו לב שאם קיים מסלול בין u ל-v אז לא בהכרח הצמתים באותו רכיב/מחלקה.

מציאת רכיבי קשירות - האלגוריתם של Kosaraju

- האלגוריתם מאפשר לנו למצוא רכיבי קשירות בזמן ליניארי
- ע"י שני קריאות ל-DFS נוכל לגלות את רכיבי הקשירות של הגרף
- אומנם קיים אלגוריתם יעיל יותר (Tarjan), אבל נלמד את האלגוריתם הנ"ל מכיוון שיותר קל להבין את המימוש שלו.

מציאת רכיבי קשירות - האלגוריתם של Kosaraju

- <u>הכנה:</u> נצטרך לשמור את המשתנים הבאים:

```
vector<int> g[maxn], gt[maxn];
vector<vector<int>> scc{{}};
int sc = 0;
vector<bool> visited(maxn,false);
vector<int> order;
```

- כאשר gt מחזיק את הגרף ההופכי ($(v,u) \in E(G_T) \iff (u,v) \in E(G)$), scc יחזיק את רכיבי הקשירות (בכל כניסה נקבל לראות אילו צמתים ברכיב הזה)

- ו-order נגלה בהמשך :)

מציאת רכיבי קשירות - האלגוריתם של Kosaraju

- האלגוריתם סורק את הגרף ע"י שני הרצות של DFS. בהרצה הראשונה נשמור את הסדר בו ביקרנו בכל הצמתים, ולאחר מכן נעבור בצורה הפוכה על סדר הביקור ונסרוק את הגרף ההופכי.
- כך, לאחר ההרצה השנייה נקבל את רכיבי הקשירות.

מציאת רכיבי קשירות - האלגוריתם של Kosaraju

```cpp
void dfs1(int u) {
    if(visited[u]) return;

    visited[u] = true;
    for(auto &v : g[u]) dfs1(u);

    order.push_back(u);
}
```

מציאת רכיבי קשירות - האלגוריתם של Kosaraju

```
reverse(all(order));

for(auto &u : order)
    if(visited[u]) {
        dfs2(u);
        ++sc;
        scc.push_back({});
    }
```

```
void dfs2(int u) {
    if(!visited[u]) return;

    visited[u] = false;
    scc[sc].push_back(u);

    for(auto &v : gt[u]) dfs2(v);
}
```

מציאת רכיבי קשירות - האלגוריתם של Kosaraju

```cpp
reverse(all(order));

for(auto &u : order)
    if(visited[u]) {
        dfs2(u);
        ++sc;
        scc.push_back({});
    }
```

```cpp
void dfs2(int u) {
    if(!visited[u]) return;

    visited[u] = false;
    scc[sc].push_back(u);

    for(auto &v : gt[u]) dfs2(v);
}
```

```cpp
#define all(X) X.begin(), X.end()
```

דג (גמל)

- גרף מכוון $G=(V,E)$ ייקרא DAG (directed acyclic graph) אם אין בו מעגלים מכוונים.
- כלומר $\varnothing = \mapsto$.
- כלומר אין רכיבי קשירות.
- או: גרף יהיה DAG אם ניתן לעשות לו **מיון טופולוגי**.

מיון טופולוגי

- בהינתן DAG, נרצה לסדר את הצמתים לקבוצות כך שבתוך כל קבוצה אין קשתות, ובין קבוצות יש קשתות.
- בהנחה שהגרף שלנו הוא כבר DAG, אז מערך ה-order יהיה בדיוק המיון הטופולוגי שלנו.
- אחרת, נצטרך גם לבדוק שהגרף לא מכיל מעגלים (ניתן במקביל להרצת ה-DFS)

פונקציית משקל

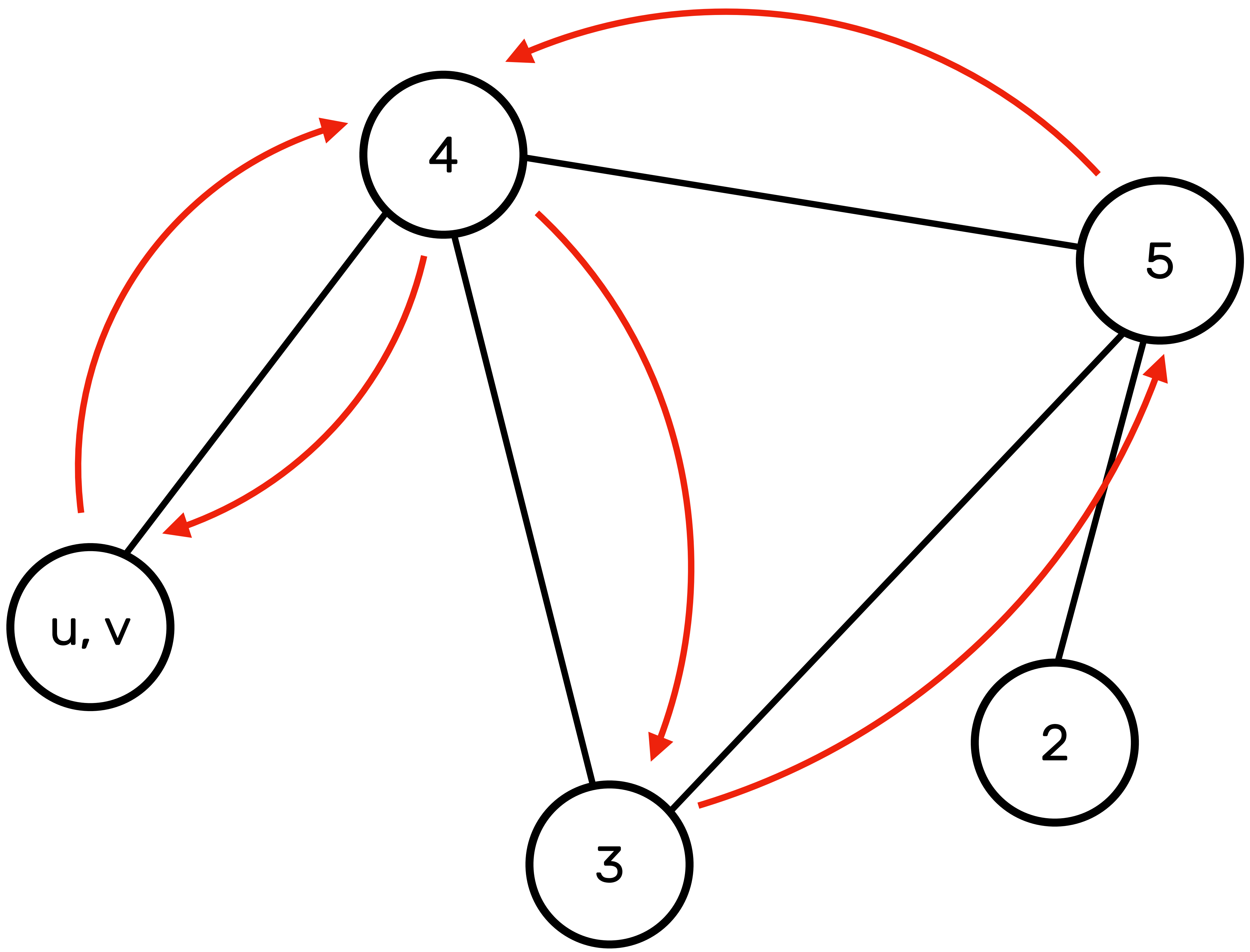
פונקציית משקל ◄ מסלולים

מסלולים

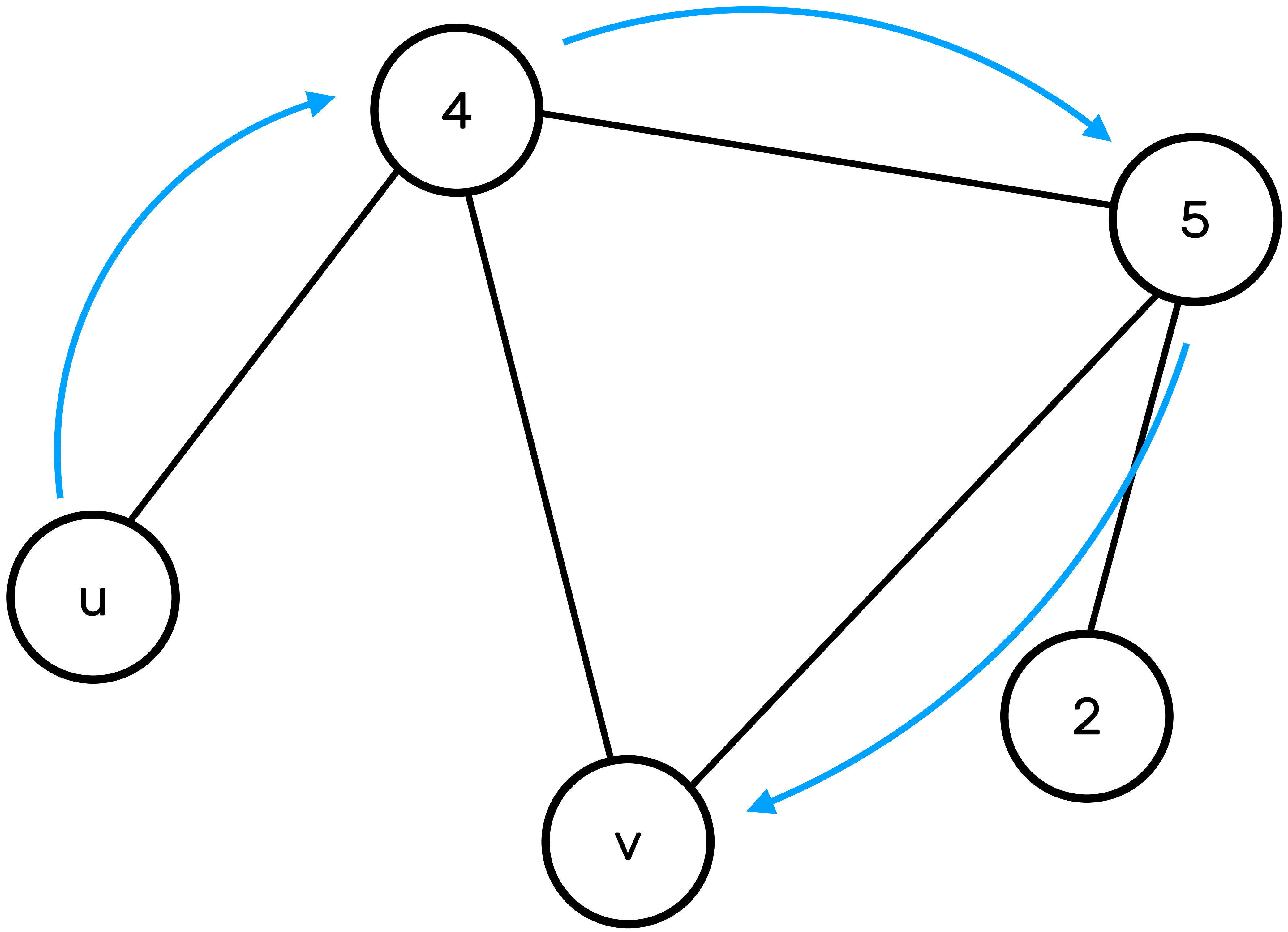
- יהי $G = (V, E)$ גרף, $u, v \in V$ צמתים ו-$(e_n)_n$ סדרה (סופית) של קשתות ($e_i \in E$) כך ש:
  - $e_1 = (u, \cdot\,)$ (או $\{u, \cdot\,\}$)
  - $e_n = (\,\cdot\,, v)$ (או $\{\,\cdot\,, v\}$)
  - אם $e_i = (x, y)$ אז $e_{i+1} = (y, \cdot\,)$ (וכן אם הגרף לא מכוון)

מסלולים

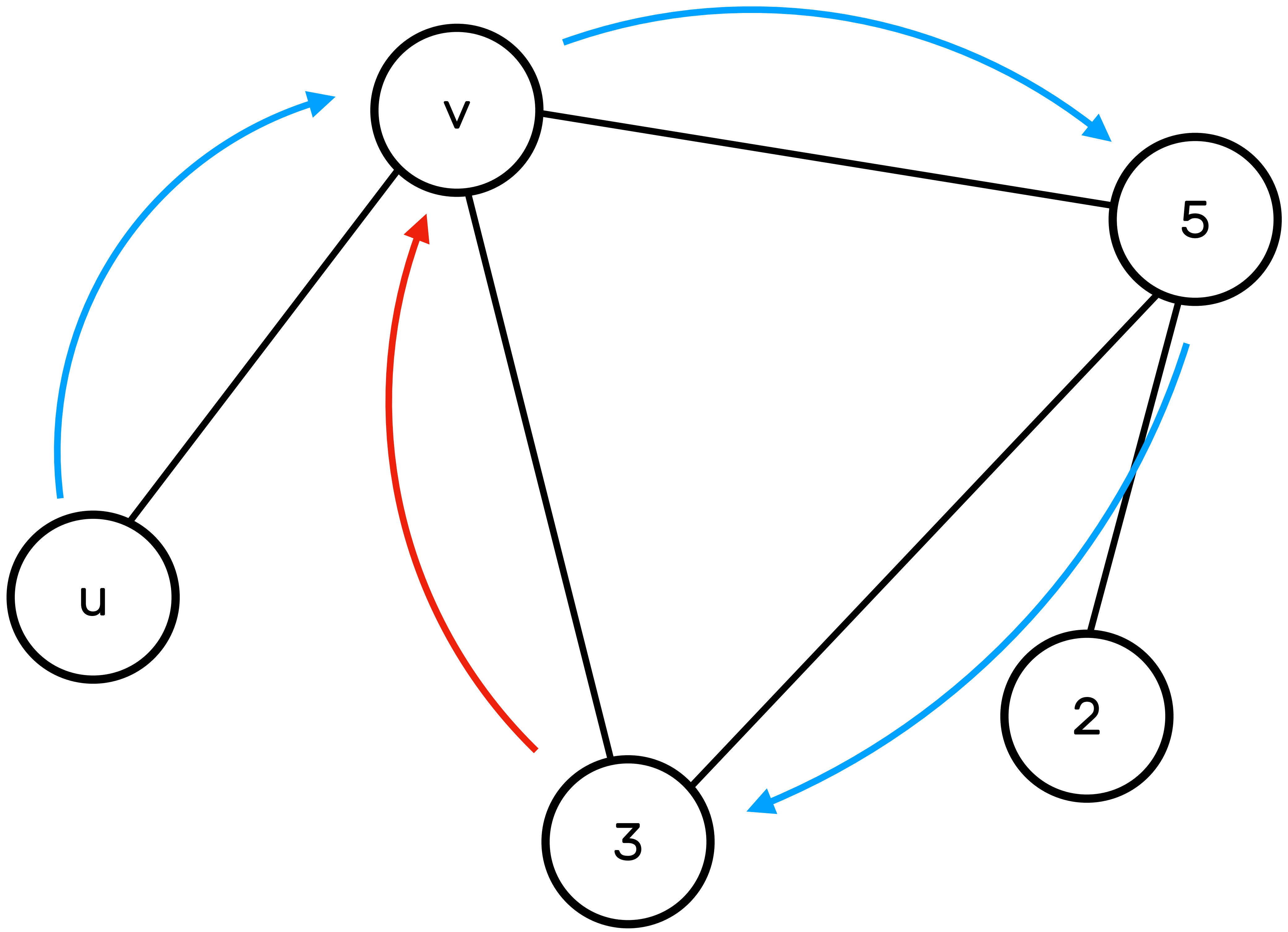
- יהי $G = (V, E)$ גרף, $u, v \in V$ צמתים ו-$(e_n)_n$ סדרה (סופית) של קשתות $(e_i \in E)$ כך ש:
[snip]
- אז:
    - הסדרה (e_n) נקראת **הליכה** בגרף
    - אם כל צומת מופיע לכל היותר פעם אחת ב-(e_n) אז הסדרה נקראת **מסלול פשוט**
    - אם $u = v$ אז הסדרה נקראת **מעגל**
    - אם $\bigcup \{e_i\} = E$ אז הסדרה נקראת **מעגל אוילרי**



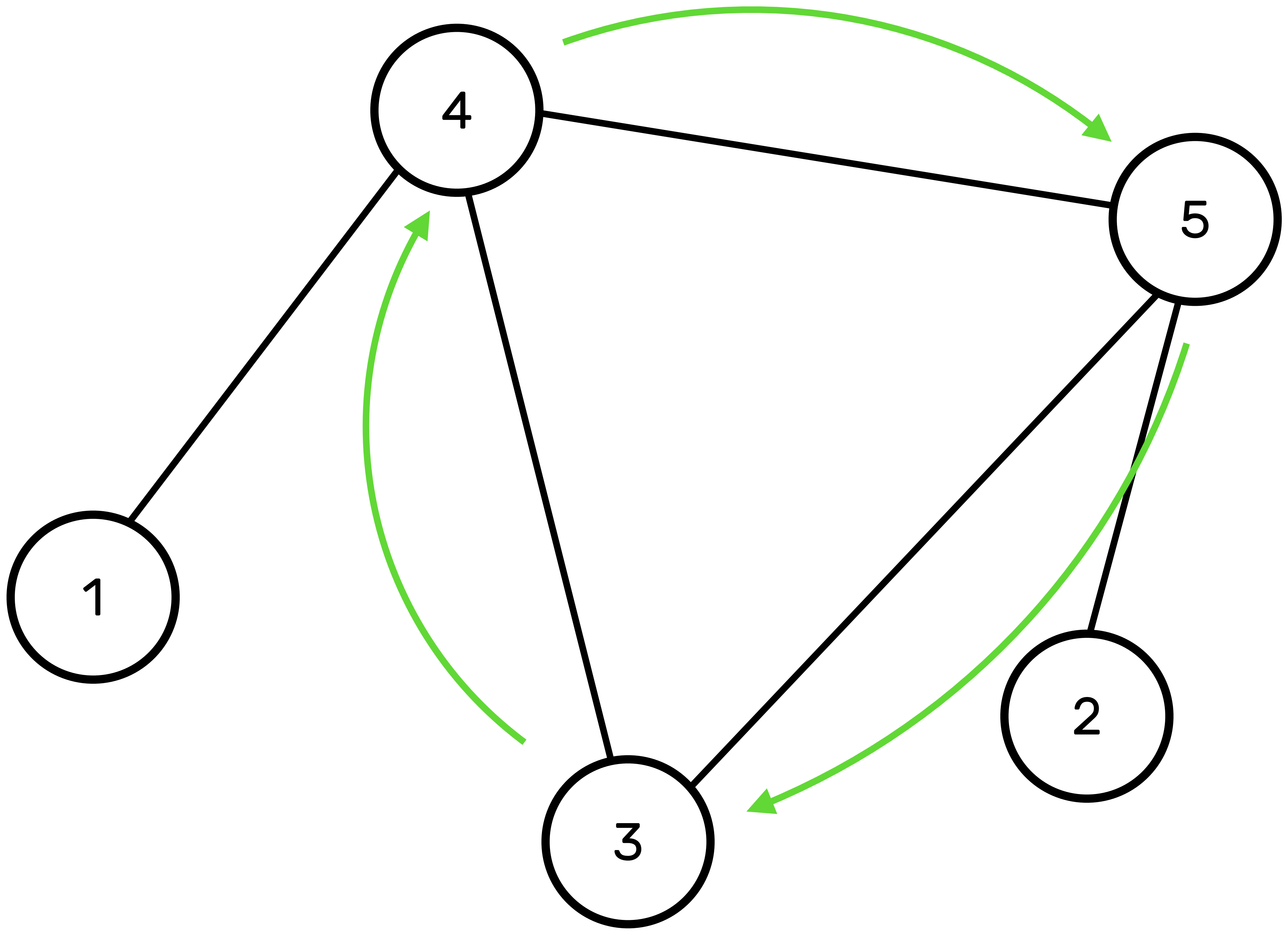
מעגל אוילרי מעגל מסלול פשוט **הליכה**



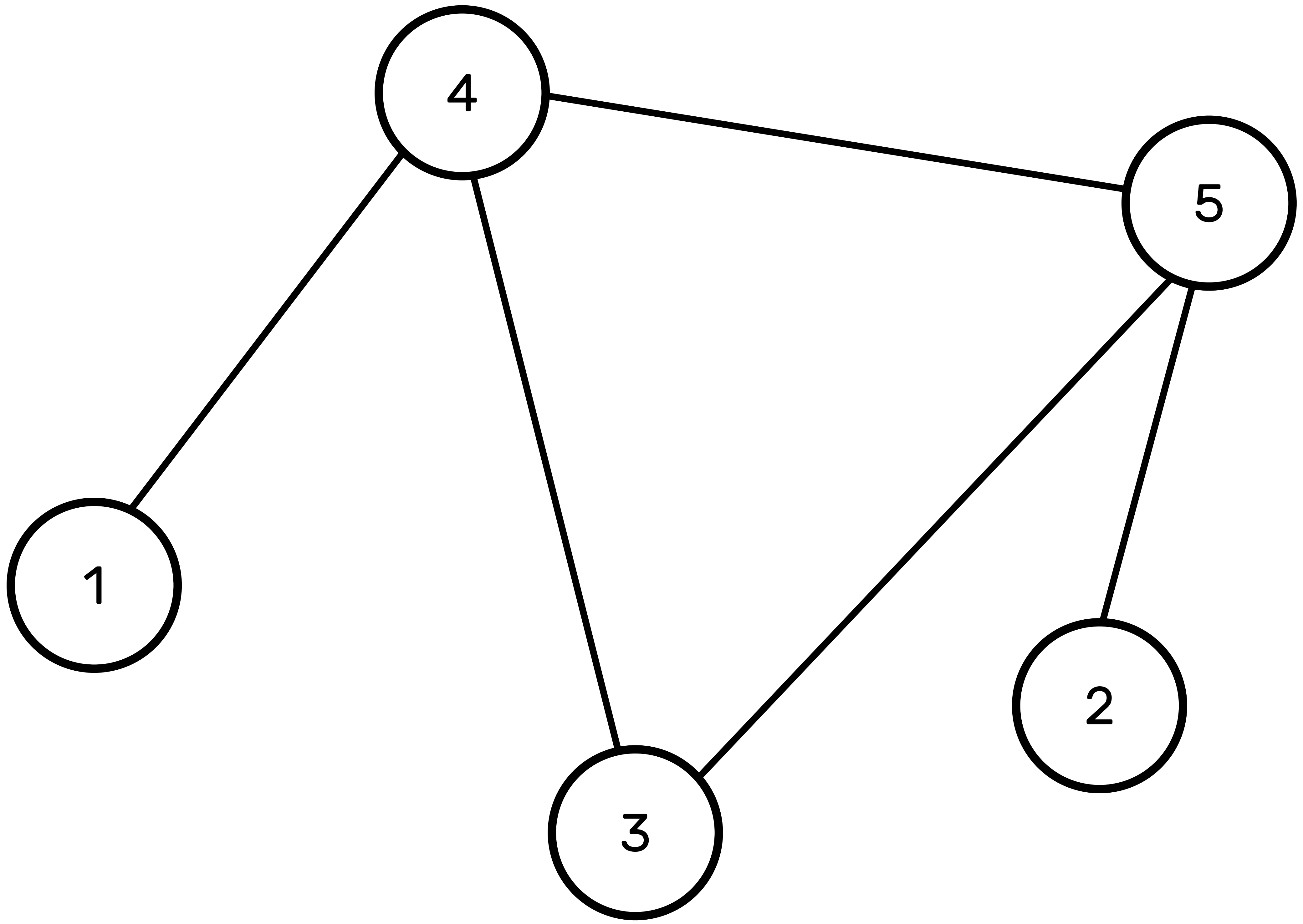
מעגל אוילרי   מעגל   **מסלול פשוט**   הליכה



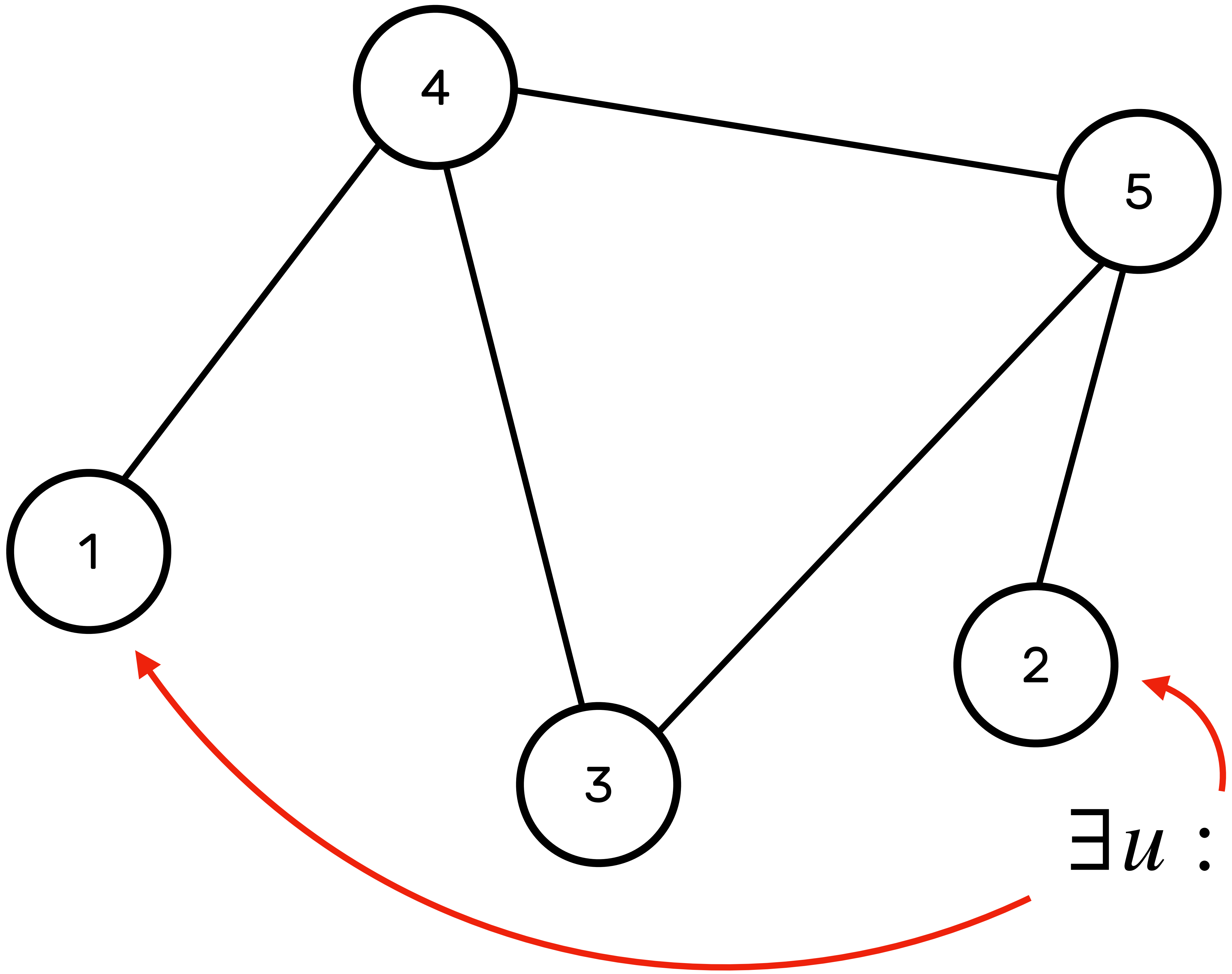
הליכה ~~**מסלול פשוט**~~ מעגל מעגל אוילרי



מעגל אוילרי **מעגל** מסלול פשוט הליכה

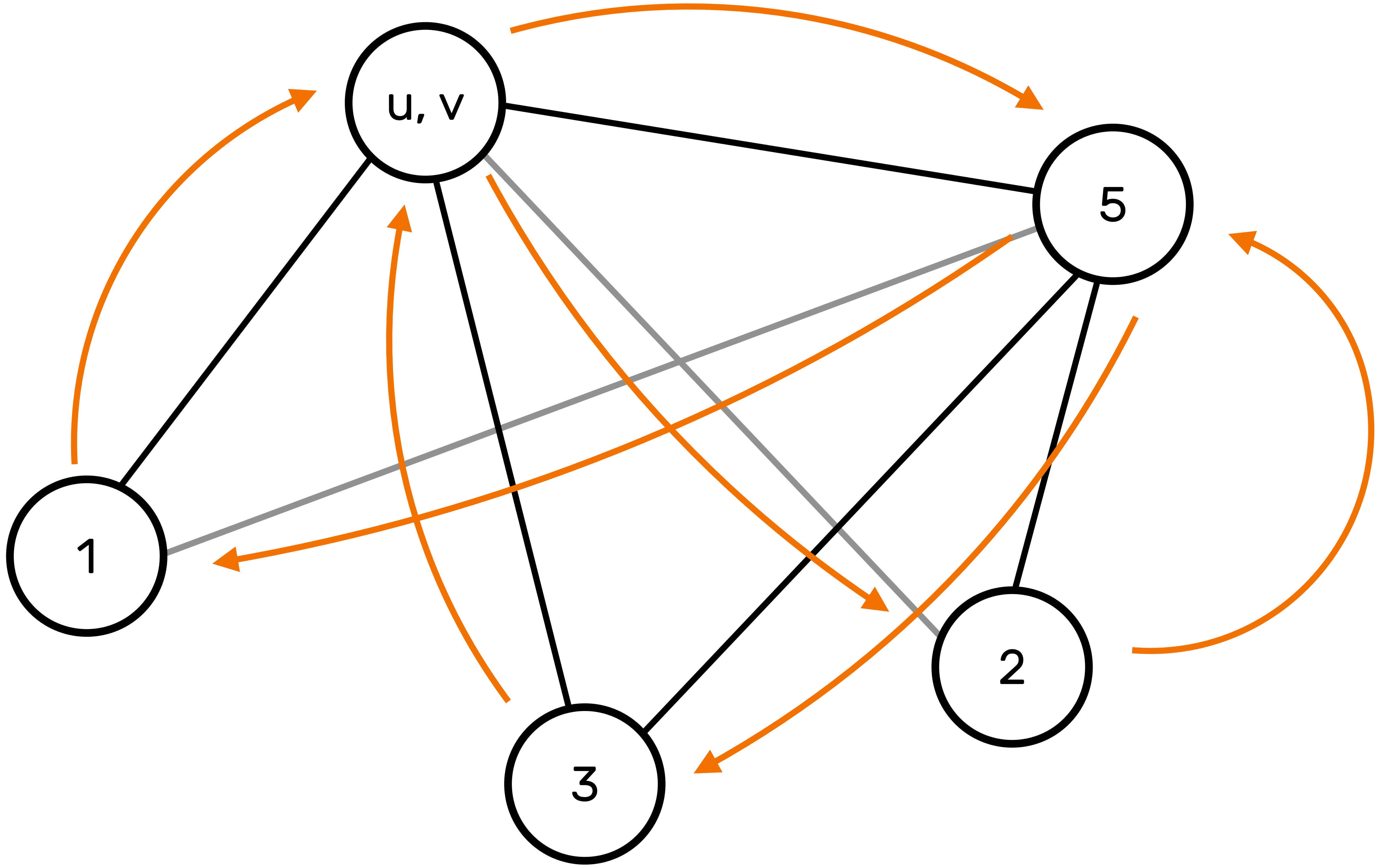


הליכה מסלול פשוט מעגל **מעגל אוילרי**



$\exists u : \deg(u)$ not even

מעגל אוילרי מעגל מסלול פשוט הליכה



הליכה מסלול פשוט מעגל **מעגל אוילרי**

פונקציית משקל

- נתעניין בפונקציות משקל $w : E \to A$. בפשטות, הפונקציה מעניקה מחיר לכל קשת.
- נגדיר מחיר/משקל מסלול כסכום הקשתות במסלול, כלומר:

$$w(e_n) = \sum_i w(e_i)$$

פונקציית משקל

- נתעניין בפונקציות משקל $w : E \to A$. בפשטות, הפונקציה מעניקה מחיר לכל קשת.
- עבור A-ים שונים נראה כי צריך אלגוריתמים אחרים.
- נסתכל על הA-ים הבאים:
  - $A = \{1\}$
  - $A = \mathbb{R}^+$
  - $A = \mathbb{R}$

$w : E \to \{1\}$

- כבר ראינו פונקציית משקל כזאת!
- נשים לב שכעת מחיר מסלול שווה למספר הקשתות במסלול.
- אלגוריתם: נריץ BFS. (ראינו כבר)

$$w : E \to \mathbb{R}^+$$

- משקלים ממשיים חיוביים
- אלגוריתם: Dijkstra. בעצם BFS אבל עם תכנות דינאמי

$w : E \to \mathbb{R}^+$

- משקלים ממשיים חיוביים
- אלגוריתם: Dijkstra. בעצם BFS אבל עם תכנות דינאמי
- מימוש:

1. ניקח את המימוש של BFS
2. נחליף את queue בset (פורמאלית, בpriority_queue)
3. כאשר נבקר צומת v נבדוק האם $d[v] < d[u] + w(u, v)$ - אם כן נעדכן את המרחק אל v

```cpp
vector<int> dijkstra(int source) {
    set<pair<int,int>> s; // set of {dist_to_source, vertex}
    vector<int> distances(n, inf); // equiv. use double
                                   // or any other big number
    distances[source] = 0;
    s.insert({0,source});
    while(!s.empty()) {
        auto [du,u] = *s.begin();
        s.erase(s.begin());

        if(du>distances[u]) continue;

        for(auto& [v, w] : w_graph[u]) {
            if(distances[v] > distances[u] + w) {
                distances[v] = distances[u] + w;
                s.insert({distances[v],v});
            }
        }
    }

    return distances;
}
```

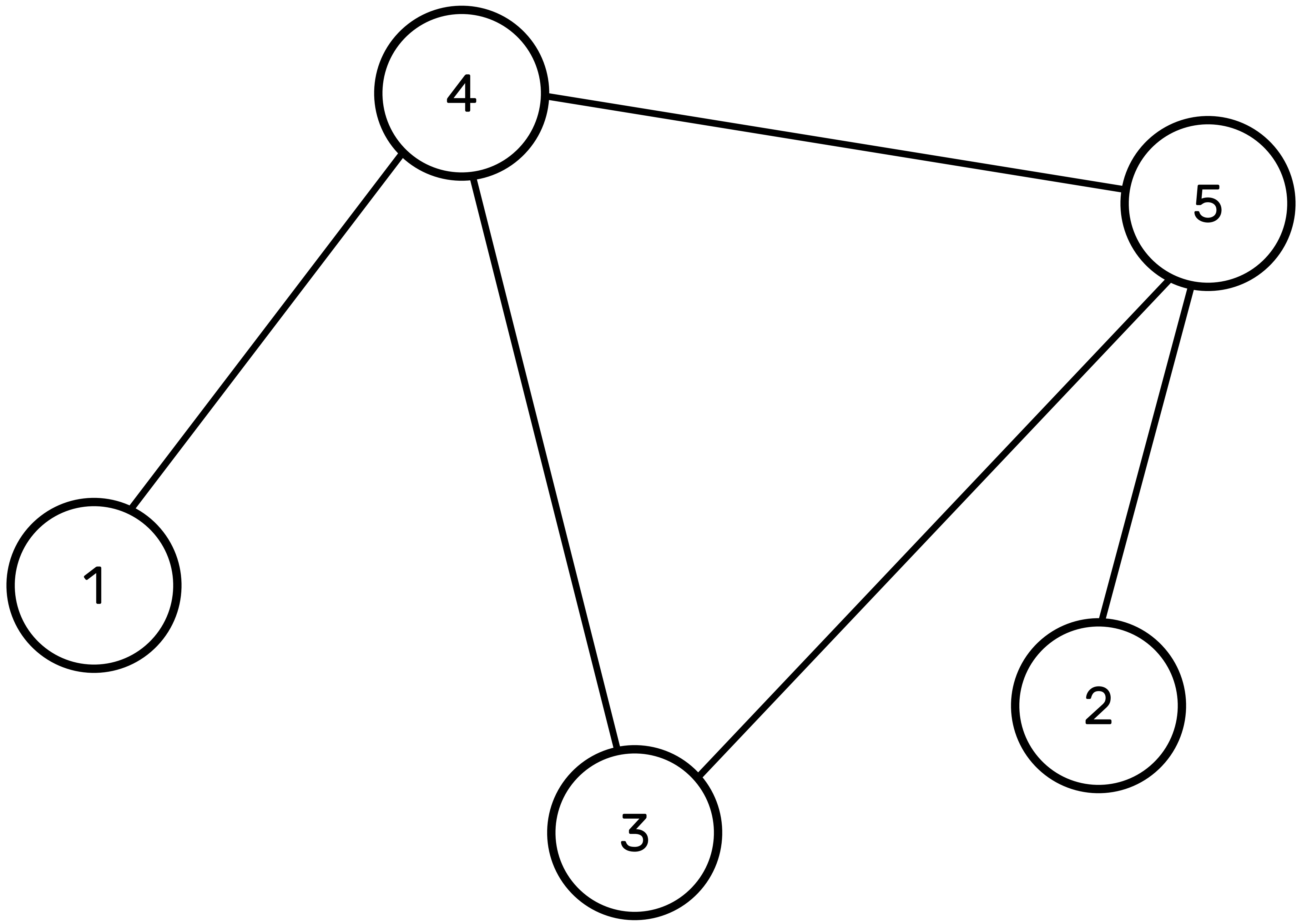
```cpp
vector<int> dijkstra(int source) {
    set<pair<int,int>> s; // set of {dist_to_source, vertex}
    vector<int> distances(n, inf); // equiv. use double
                                   // or any other big number
    distances[source] = 0;
    s.insert({0,source});
    while(!s.empty()) {
        auto [du,u] = *s.begin();
        s.erase(s.begin());

        if(du>distances[u]) continue;

        for(auto& [v, w] : w_graph[u]) {
            if(distances[v] > distances[u] + w) {
                distances[v] = distances[u] + w;
                s.insert({distances[v],v});
            }
        }
    }

    return distances;
}
```

העניין המרכזי:

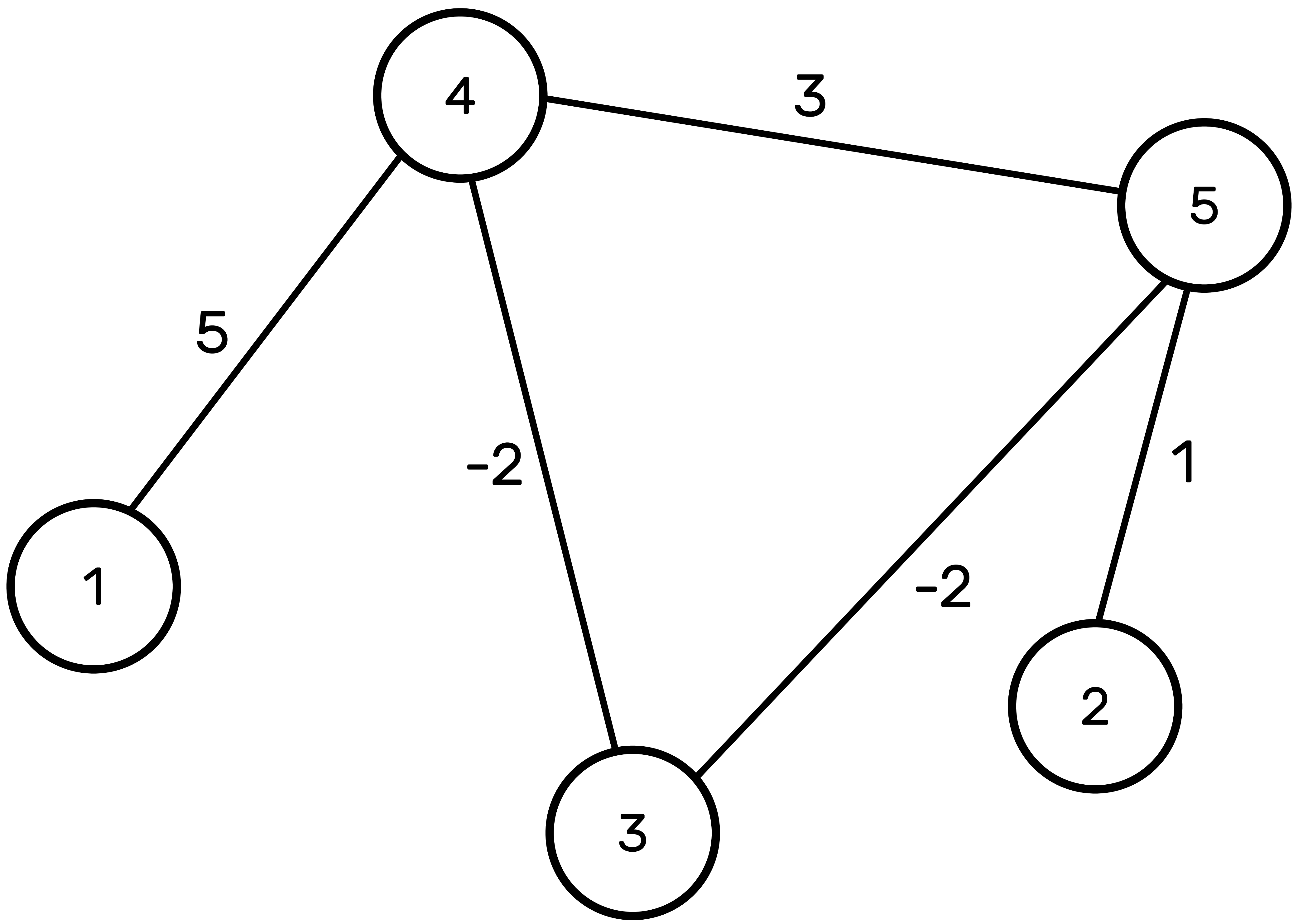
באופן איטרטיבי נמצא את
המרחק הקצר ביותר
מsource אל כל צומת v.

מכיוון שהמשקלים חיובים
אז בהכרח אפשר להבטיח
שקיים מספר כזה
(האם תמיד אפשר?)

$$w : E \to \mathbb{R}$$

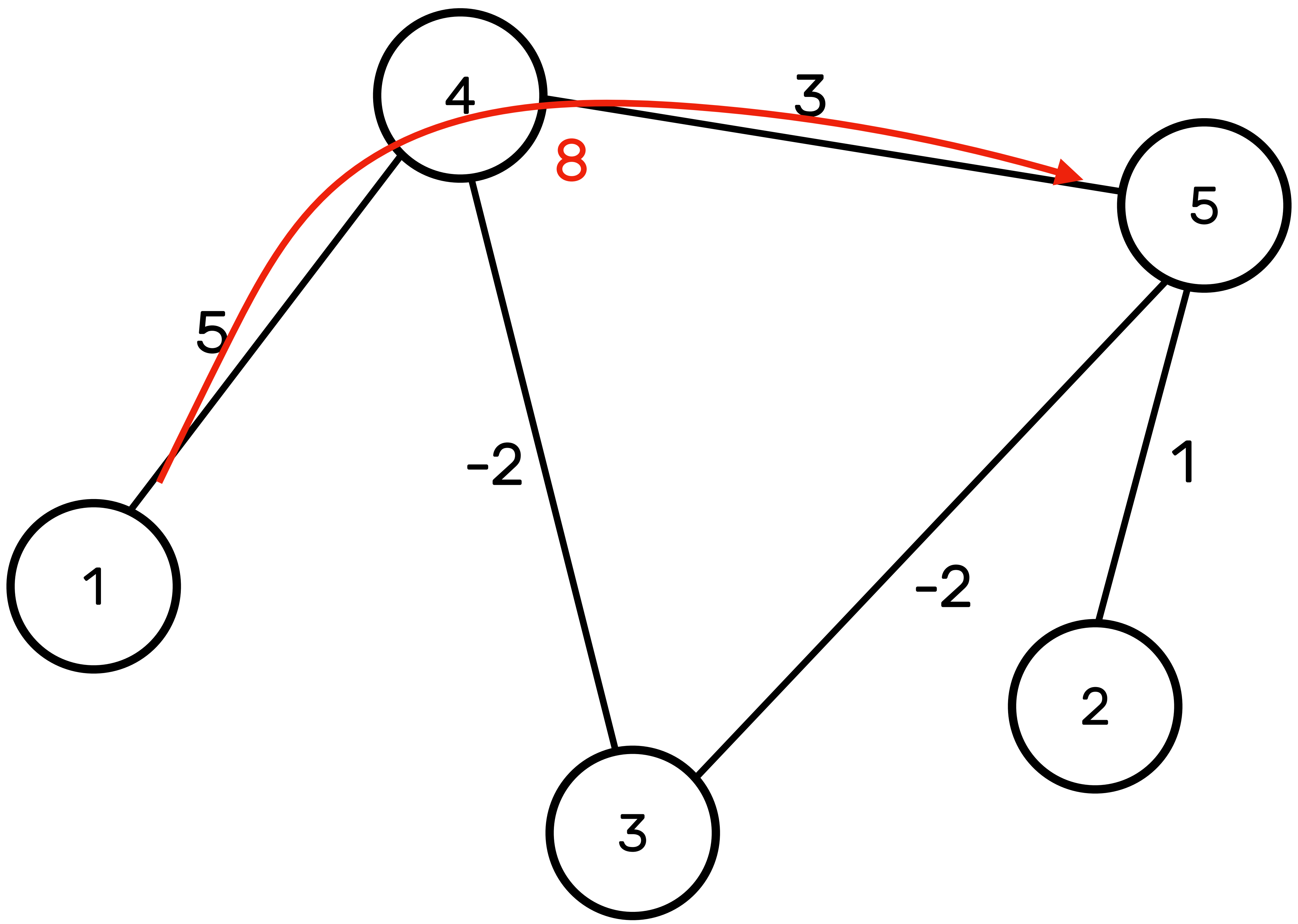


$$w : E \to \mathbb{R}$$



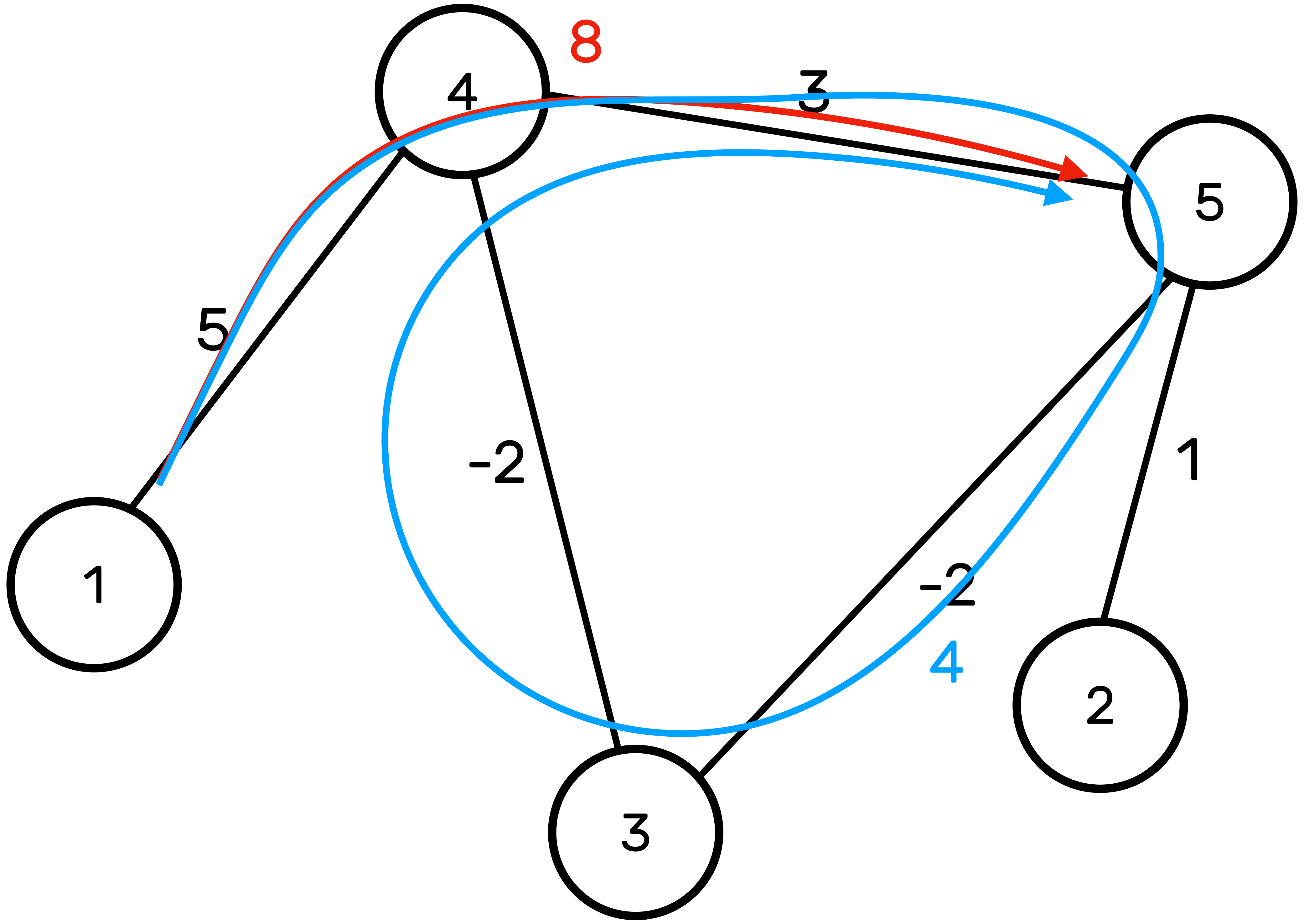
$$d[1,5] = ?$$

$w : E \to \mathbb{R}$



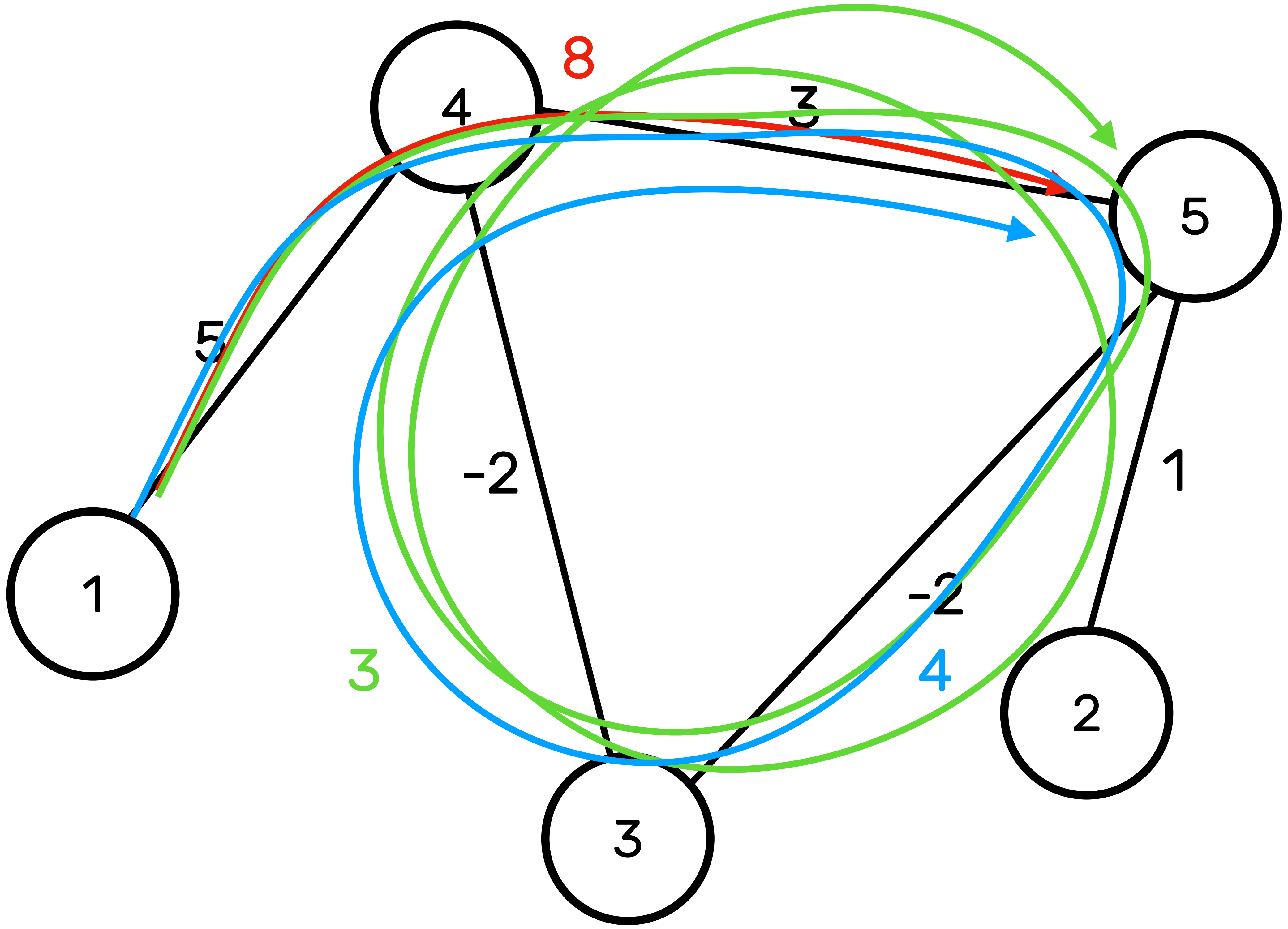
$d[1,5] = ?$

$$w : E \to \mathbb{R}$$



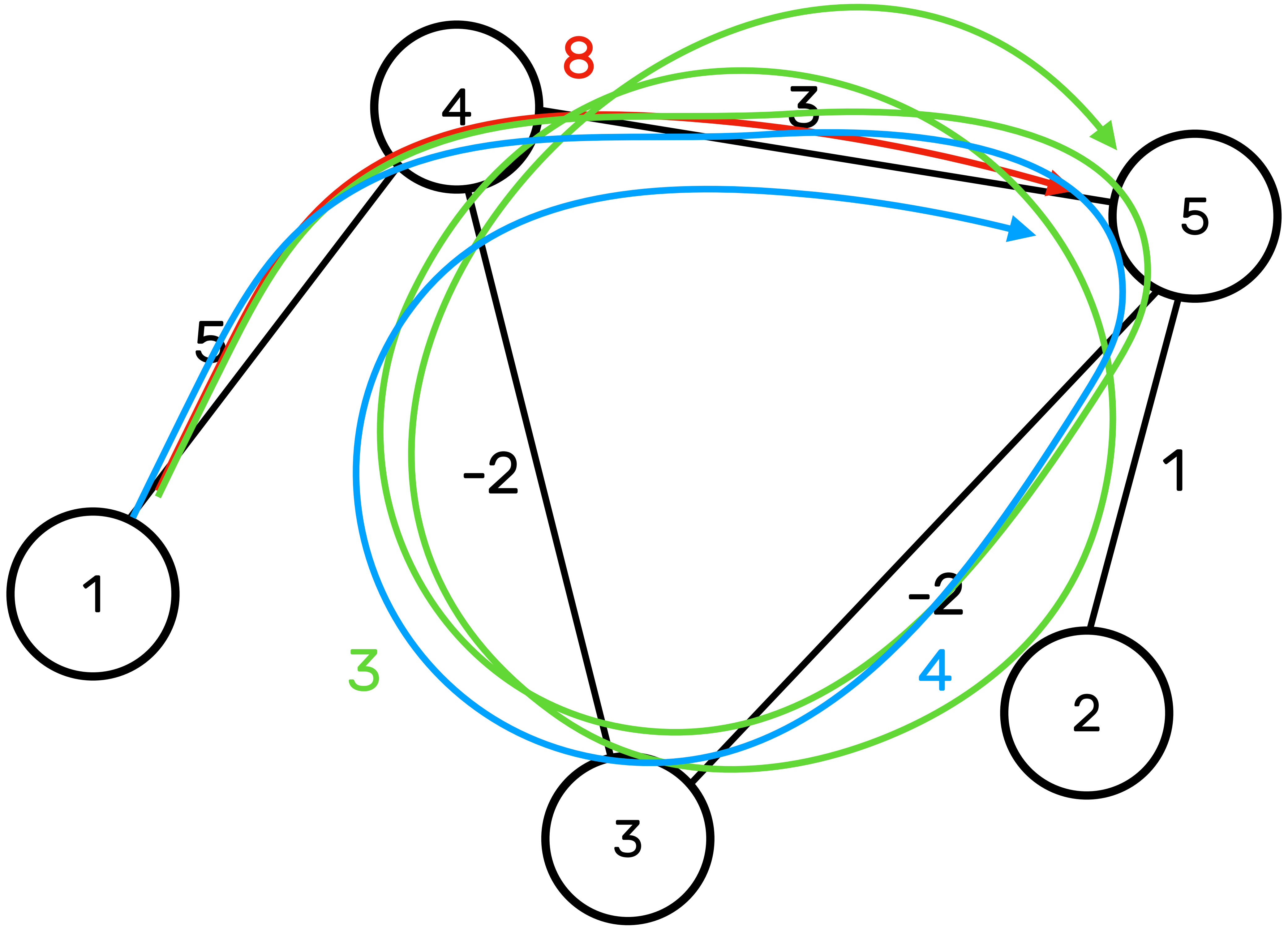
$$d[1,5] = ?$$

$w : E \to \mathbb{R}$



$d[1,5] = ?$

$$w : E \to \mathbb{R}$$



$$d[1,5] = -\infty$$

פתרון: Bellman-Ford

- בשיעור הבא :)