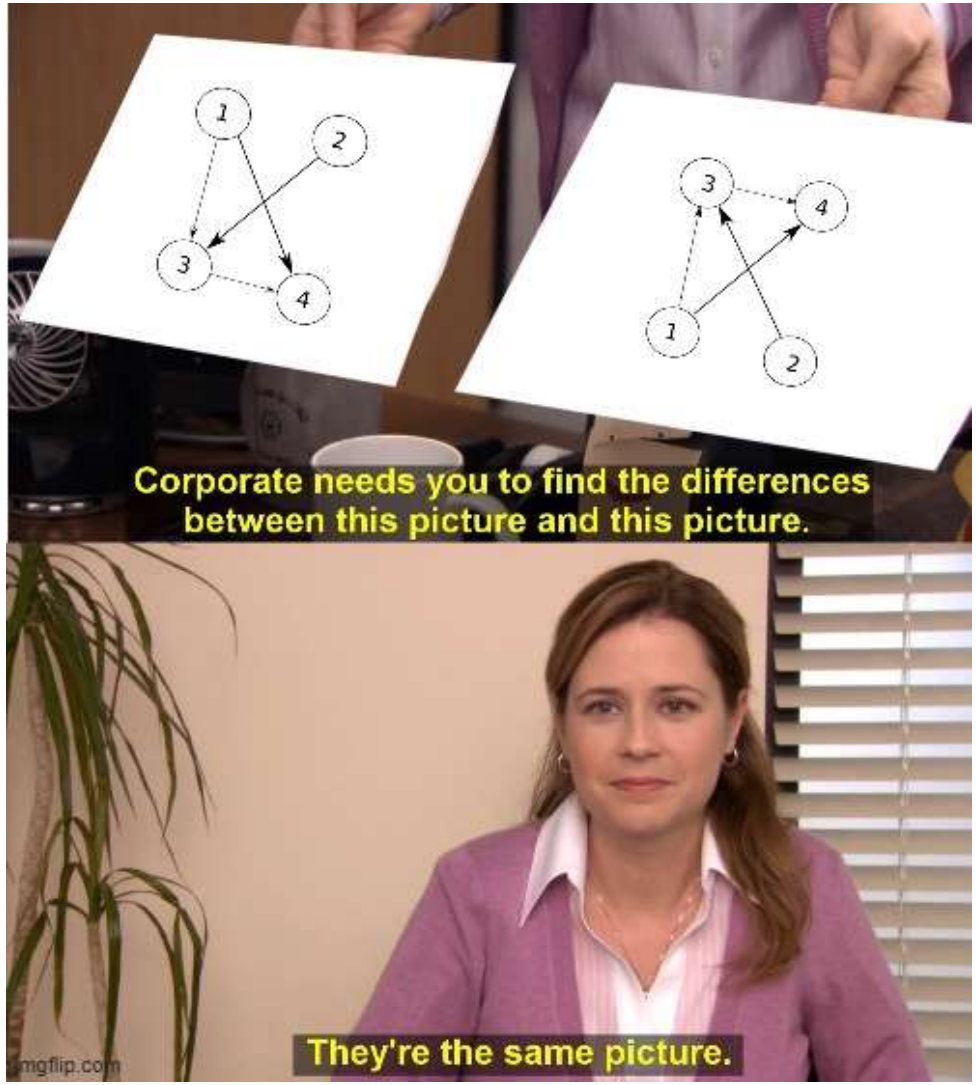


גרפים

Dynamic programming = DP

מה זה גרף?

אז לא מדובר בגרפים שאתם מכירים מהתיכון, מדובר במודל מתמטי לייצוג קשרים.

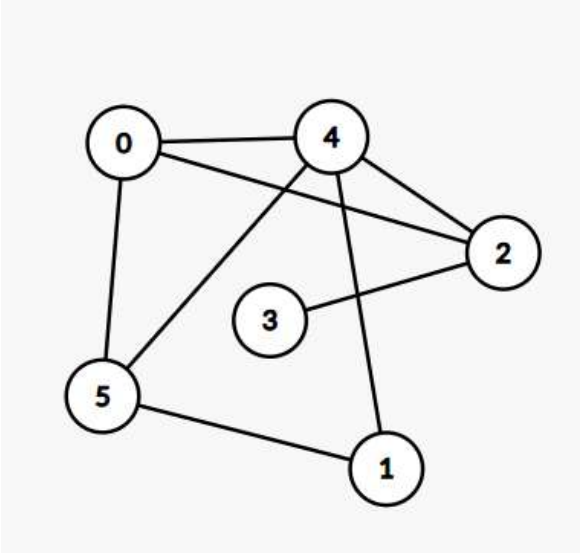
לעיגולים קוראים צמתים (nodes) ולקווים קוראים קשתות (edges).

הקשתות יכולות להיות עם כיוון או חסרות כיוון, ואם תחשבו על כל צומת כבן אדם

וכל קשת כיחס כלשהו (עוקב אחריו באינסטגרם), תוכלו לראות למה שמישהו

יתעניין בדבר שכזה.

סימון $G = (V,E)$ מייצג גרף בעל הצמתים V והקשתות E .



שנייה לפני תיאוריה - ייצוג

גרפים במחשב

על מנת לייצג גרפים במחשב יש שתי גישות מקובלות: $Adj\ List, Adj\ Mat$.

רשימת שכינויות: נוכל לשמור מערך של ווקטורים בגודל $|V|$ שבו כל תא מייצג צומת, ובכל ווקטור יש לנו את השכנים של הצומת שזה התא שלה במערך הגדול. במילים אחרות, עבור רשימה g מטיפוס vvi , נקבל $g[u]$ בתור קבוצת השכנים המכוונים של צומת u.

מטריצת שכינויות: מטריצה בינארית בגודל $|V|^2$ שבה התא ה- (i,j) דולק אם הקשת (v_i, v_j) נמצאת.

ברוב השאלות יותר שימושי להשתמש ברשימת שכינויות שכן המקום שהיא דורשת הוא $O(V+E)$ בניגוד למטריצה - למה?

מעברים על גרפים

בדומה למערך, הרבה פעמים נרצה לעבור על גרף בצורה חישובית (המקבילה ללולאת for במערך), אבל בגרפים זה לא כזה פשוט – בעיקר כי אין בהכרח סדר (בגרפים כלליים יכולים להיות מעגלים והם יכולים להיות מורכבים).

אנשים חכמים חשבו על שתי דרכים מרכזיות שהן מאוד אינטואיטיביות והן זכו לשמות DFS ו- BFS, על שם depth first search, ו- breadth first search בהתאמה.

DFS

DFS

- הרעיון הוא להתחיל מצומת כלשהו s על שם start, ולסמן את הצמתים שכבר ביקרנו בהם לאורך הדרך.
- כאשר הגענו ל צומת u נבקר רקורסיבית כל שכן שלו $v \in g[u]$ בתורו, ולאורך הדרך אפשר לעשות חישובים או דברים נוספים.

```
vector<vector<int>> g;
vector<int> vis;

void dfs(int u){
    vis[u] = 1;
    for(auto v : long long  : g[u]) if(!vis[v]){
        dfs( u: v);
    }
}
```

- יכולים לחשוב על קוד?

- הערה: קיימת גם גרסה שאינה רקורסיבית (סטאק),

אבל משמעותית יותר נוח לחשוב על הרקורסיבית.

BFS

- הרעיון במעבר זה הוא להתפשט החוצה מ s בסדר ממויין לפי מרחק (בשכבות).
- במילים אחרות: נתחזק תור שמתחיל עם s ונסמן שביקרנו בו. לאחר מכן כל עוד התור לא ריק, ניקח את u ראש התור, נוסיף את השכנים שלו לתור אם הם לא בוקרו,

  ונסמן שביקרנו בהם, ונדחוף אותם לתור.
- באופן זה האלגוריתם מתפשט דומה למגפה, כל פעם אנשים שנמצאים

  במרחק i מכניסים את השכנים שלהם שבמרחק $i+1$ מ s.

```
vector<vector<int>> g;
vector<int> vis;

void bfs(int s){
    queue<int> q;
    vis[s] = 1;
    q.push( x: s);
    while(q.size()){
        auto u :long long  = q.front();
        for(auto v :long long  : g[u])if(!vis[v]){
            vis[v] = 1;
            q.push( x: v);
        }
    }
}
```

BFS vs DFS

- יש הרבה יתרונות לשני האלגוריתמים – בכל שקשור למרחק בגרף חסר משקלים (כמו שהיה עד עכשיו) – BFS לוקח, וכנראה הוא האופציה הנכונה, עם זאת DFS הוא אלגוריתם מאוד פשוט למימוש והרבה פעמים אבחנות על סדר המעבר ב DFS יוצרות אלגוריתמים מאוד מורכבים שנעשים אלגנטיים לכתיבה (ראו toposort בהמשך המצגת ☺).

- הסיבוכיות של שני האלגוריתמים זהה - $O(V+E)$.
- מבחינת מקום הם דורשים גרף ומערך vis ולכן זו גם סיבוכיות המקום.
- וכמובן שאפשר להוסיף עוד פעולות בין הפעולות של האלגוריתמים כדי לחשב דברים כמו מרחק, עומק, כמות ילדים וכו' בהתאם לתנאי השאלה.

בעיית המסלול הקצר ביותר

יהיה גרף $G = (V,E)$, נרצה בסיבוכיות $O(V + E)$ לחשב את המסלול הכי קצר מצומת u לצומת v נתונים.

```
vector<vector<int>> g;
vector<int> vis, d( n: 0);

void bfs(int s){
    queue<int> q;
    vis[s] = 1;
    d.assign( n: g.size(), val: 0);
    d[s] = 0;
    q.push( x: s);
    while(q.size()){
        auto u : long long  = q.front();
        for(auto v : long long  : g[u])if(!vis[v]){
            vis[v] = 1;
            q.push( x: v);
            d[v] = d[u] +1;
        }
    }
```

ברור ש BFS הוא הרלוונטי כאן, מדובר על מרחק!

(צריך להיות q.pop() אחרי ה(q.front().

בעיית המסלול הקצר ביותר - הוכחה

נוכיח באינדוקציה על המרחק:

עבור צומת במרחק $i = 0$ (זה רק הצומת s) , ברור שאי אפשר לקבל מסלול שיותר טוב מאורך 0.

נניח את נכונות האלגוריתם עבור מרחק i ונוכיח את נכונותו עבור $i + 1$.

יהי u במרחק i מ s , אזי $d[u] = i$ הוא האורך האופטימלי. נשים שכל שכן שטרם ביקרנו בו של u ע"פ הנחת האינדוקציה במרחק שגדול מi, (שכן אחרת היה מתווסף לתור בשלב i, ולכן $d[v] \geq i + 1$.

בנוסף, המסלול $s \to u \to v$ הוא תקין ובאורך $i + 1$ ולכן זהו המסלול המינימלי ל v.

בעיית המסלול הקצר ביותר 2

יהיה גרף $G = (V,E)$, נרצה בסיבוכיות טובה לחשב את המסלול הכי קצר מצומת u לצומת v נתונים.

מה ההבדל? הפעם לכל קשת יש משקל! כלומר כל קשת נחשבת המספר שכתוב עליה, והמשקלים האלה בהכרח אי שליליים (הגדרת השאלה).

בדומה לגרף רגיל, גרף זה מיוצג ע"י רשימת שכינויות של זוגות $\{v,w\}$ בתוך $g[u]$

אם קיימת קשת (u,v) במשקל w.

זו בעיה מוכרת ביותר, והחלק המעניין הוא שהיא נפתרת גרידית.

אלגוריתם דייקסטרה dijkstra

- הרעיון הוא להבין שאם נסתכל על כל השכנים של צומת ההתחלה s, זה בעל המשקל המינימלי הוא בהכרח הדרך הכי יעילה להגיע מ s אליו.
- פורמלית: נשים לב כי כל מסלול אופטימלי יהיה פשוט בה"כ, שכן נוכל להוריד כל מעגל בתוכו ורק להקטין את המשקל.
- יהי $u \in g[s]$ שהקשת אליו היא בעלת משקל w, ולכל $v \neq u$ מתקיים $w(s,v) \geq w(s,u)$ אז מתקיים $d(u)$ $= w(s,u)$, כאשר $d(u)$ הוא סכום המשקלים המינימלי במסלול מs לu שכן אחרת קשת זו לא הייתה קטנה ביותר. (ציירו דוגמה!)

המשך דייקסטרה

האבחנה הזו מאפשרת למצוא את המסלול האופטימלי מ-s לאחד השכנים שלו, והאבחנה הבאה היא שתמיד נוכל לחבר את s עם u מהאבחנה הקודמת לכדי צומת אחד גדול, ואז להפעיל את האבחנה מחדש.

במילים אחרות נרצה לתחזק קבוצה שמתחילה ב s , שניתן למצוא את המינימום מבין הקשתות שלה ולמחוק את הצומת אליו היא מובילה, ונוכל להוסיף אליה קשתות.

יכולים לחשוב על מבנה כזה?

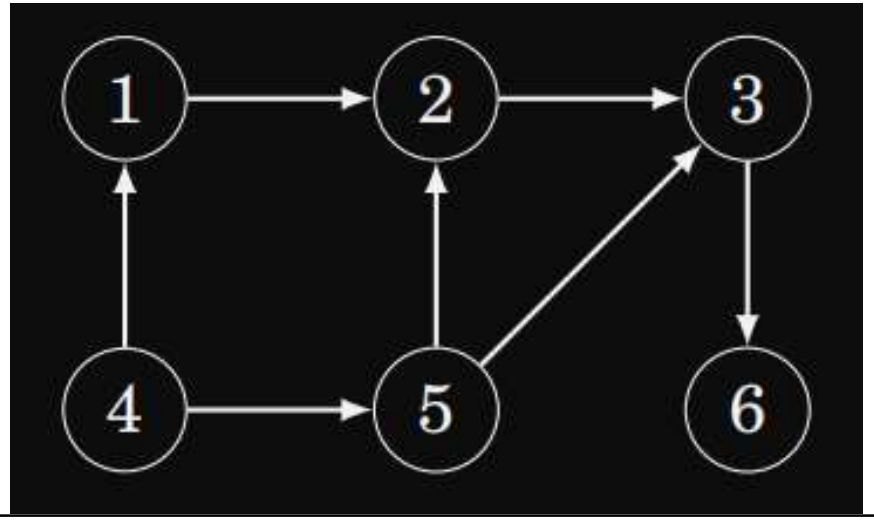
סוף דייקסטרה

- לתחזק ערימת מינימום – לכאורה הכי טוב פיבונאצ'י או relaxed אבל פשוט נשתמש ברגיל של STL.
- כך נראה הקוד:
- שימו לב שבנוסף למסלול $u \to v$, זוג האלגוריתמים מחשבים את $d(u)$ של כל הצמתים בגרף, אז ניתן להריץ רק פעם אחת אותם ולקבל את כל המרחקים (BFS, DIJKSTRA).

```
cin >> n >> m;
rep(i, 0, m){
  int a, b, c; cin >> a >>b >> c;
  g[a].push_back({b, c});
}
rep(i, 2, n+1) dis[i] = inf;
priority_queue<pii, vector<pii>, greater<pii>> q;
q.push({0, 1});
dis[1] = 0;
while(q.size()){
  int u= q.top().second;q.pop();
  if(vis[u]) continue;
  vis[u]=1;
  for(auto [v, df] : g[u])
    if(dis[u] + df < dis[v])
      dis[v] = dis[u] + df, q.push({dis[v], v});
}
```

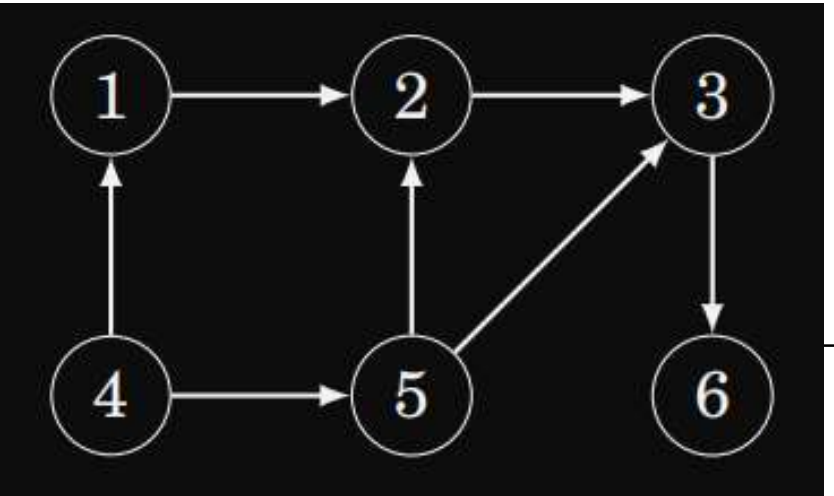
DAG - Directed Acyclic Graph

- דאג הוא גרף מכוון ללא מעגלים.
- נשים לב שניתן לתת לדג סדר מסוים. הסדר הזה נקרא מיון טופולוגי של הדג.



סדר מיון טופולוגי

- הסדר של המיון הטופולוגי יהיה מוגדר כך: כל צומת שיכול להגיע אל צומת v, יהיה לפני v במיון הטופולוגי.
לרוב יש כמה מיונים טופולוגיים אפשריים, כי יש צמתים שאי אפשר להגיע מאחד מהם לאחר, אז ניתן לבחור בעצמנו מי יהיה יותר מוקדם במיון הטופולוגי.
- בגלל הסדר הזה הרבה פעמים ניתן להשתמש ב dag בשביל פתרונות דומים לתכנות דינאמי.
- סדרים אפשריים לדג בדוגמה:
- [4,1,5,2,3,6]
- [4,5,1,2,3,6]



מימוש פשוט של מיון טופולוגי

```
void solve(){
    int n, m, a, b;
    cin >> n >> m;
    queue<int> q;
    vector<int> in(n, value: 0);
    vector<vector<int>> g(n);
    for(int i = 0; i < m; i++){
        cin >> a >> b; a--; b--;
        g[a].push_back(b);
        in[b]++;
    }
    for(int i = 0; i < n; i++) if(in[i] == 0){
        q.push( x: i);
    }
    while(!q.empty()){
        int v = q.front();
        q.pop();
        for(auto u : long long  : g[v]){
            //do something
            in[u]--;
            if(in[u] == 0) q.push( x: u);
        }
    }
}
```

- לכל אחד נזכור את דרגת הכניסה שלו.
- כשנפעל על מישהו נוריד את דרגת הכניסה של כל הבנים שלו ב1.
- נכניס מישהו לתור כשדרגת הכניסה שלו היא 0.

שאלה לדוגמה

- A game has n levels, connected by m teleporters, and your task is to get from level 1 to level n. The game has been designed so that there are no directed cycles in the underlying graph. In how many ways can you complete the game?

רעיון לפתרון

- כמות הדרכים להגיע לכל צומת זה סכום כמות הדרכים להגיע להורים שלו (אלה שיש קשת מהם אליו). זה נכון כי כדי להגיע אל מישהו חייבים להגיע אליו דרך אחד מההורים שלו.
- נאתחל את כמות הדרכים להגיע לצומת הראשונה ל1.
- קוד בשקופית הבאה.

```
54  const int md = 1e9+7;
55  void solve(){
56      int n, m, a, b;
57      cin >> n >> m;
58      queue<int> q;
59      vector<int> in(n, value: 0), ways(n, value: 0);
60      vector<vector<int>> g(n);
61      for(int i = 0; i < m; i++){...}//input
66      for(int i = 0; i < n; i++) if(in[i] == 0){
67          q.push( x: i);
68      }
69      ways[0] = 1;
70      while(!q.empty()){
71          int v = q.front();
72          q.pop();
73          for(auto u : long long  : g[v]){
74              ways[u] += ways[v];
75              ways[u] %= md;
76              in[u]--;
77              if(in[u] == 0) q.push( x: u);
78          }
79      }
80      cout << ways[n-1];
```