אלגוריתמים – סמסטר א' תשס"ט

מרצה: פרופ' מיכה שריר
מתרגל: דן פלדמן

תוכן הקורס

- **מבוא**
- **גרפי אוילר**
- **סריקת גרפים**
- **עץ פורש מינימלי**
- **מסלולים קצרים ביותר בגרפים עם משקלים**
- **זרימה ברשתות**
- **תכנות דינאמי**
- **התאמת מחרוזות**
- **נושאים נוספים ככל שיתיר הזמן**

אלגוריתמים – סמסטר א' תשס"ט

מרצה: פרופ' מיכה שריר
מתרגל: דן פלדמן

תוכן הקורס

- **מבוא:**
  גרפים: הגדרות וייצוג גרפים במחשב. שימושים לגרפים.
  עצים, יערות ותכונותיהם הבסיסיות – חזרה.

- **גרפי אוילר:**
  מעגל אוילר ומסלול אוילר.
  (מעגל המילטון ומסלול המילטון.)

- **סריקת גרפים:**
  חיפוש לרוחב (BFS) ושימושיו: מציאת מסלולים קצרים ביותר בקשתות.
  חיפוש לעומק (DFS), עץ DFS ותכונותיו, שימושים ל- DFS:
  בגרף לא מכוון: מציאת רכיבי דו-קשירות ומציאת גשרים.
  בגרף מכוון: מציאת רכיבי קשירות חזקה ומיון טופולוגי של גרף אציקלי.

- **עץ פורש מינימלי:**
  מציאת עץ פורש מינימלי: האלגוריתמים של Kruskal ו- Prim.

- **מסלולים קצרים ביותר בגרפים עם משקלים:**
  מציאת מסלולים קצרים ביותר מקודקוד נתון:
  האלגוריתמים של Dijkstra ו- Bellman-Ford.
  מציאת מסלולים קצרים ביותר מכל קודקוד לכל קודקוד:
  כפל מטריצות, Floyd-Warshall, Johnson.

- **זרימה ברשתות:**
  רשתות זרימה, האלגוריתם של Ford-Fulkerson, משפט Max Flow – Min Cut,
  האלגוריתם של Edmonds-Karp והאלגוריתם של Dinic.
  רשתות 0-1, מספר מקסימלי של מסלולים זרים בקשתות ובקודקודים, קשירות קשתות
  וקודקודים, זיווג מקסימלי בגרף דו-צדדי ומשפט Hall.

- **תכנות דינאמי**

- **התאמת מחרוזות:**
  בעיית התאמת מחרוזות, האלגוריתם הנאיבי ו- KMP.

- **נושאים נוספים ככל שיתיר הזמן**

ספרות

- Introduction to Algorithms – Cormen, Leiserson and Rivest
- האוניברסיטה הפתוחה (תרגום של הספר לעברית) - מבוא לאלגוריתמים
- Graph Algorithms – Shimon Even
- Data Structures and Algorithms – Aho, Hopcroft and Ullman
- Algorithm Design – Kleinberg and Tardos

חובות התלמידים וחישוב הציון

תרגילי בית יינתנו אחת לשבועיים. חובה להגיש את כל התרגילים פרט לאחד.
הגשת התרגילים אישית. במקרה של תרגיל מועתק בעליל, ייחשב התרגיל כלא מוגש לכל המעורבים בדבר (לא ייעשה ניסיון לקבוע מי המקור ומי המעתיק).
תרגיל שיוגש באיחור ייבדק אך לא יחשב כמוגש, אלא אם כן יצורף אליו אישור המצדיק את האיחור (כגון מילואים, מחלה וכו').
התרגילים יהוו כ-10% מהציון הסופי.

אתר הקורס:

http://www.cs.tau.ac.il/~dannyf/alg09/Algorithms09.htm

גרפים - הגדרות בסיסיות

בגרף $G=(V,E)$, V קבוצת ***הקדקודים*** (vertices) או ***צמתים*** (nodes), E קבוצת ***הקשתות*** (edges).
בגרף ***מכוון*** הקשתות מכוונות ובגרף ***לא מכוון*** הקשתות לא מכוונות.
גרף התשתית (הלא מכוון) של גרף מכוון הוא הגרף המתקבל מהגרף המכוון, כאשר מתעלמים מכווני הקשתות.

לולאה היא קשת מקדקוד אל עצמו. 
לקשתות מקבילות אותן קצוות, ובגרפים מכוונים, שתי קשתות מקבילות שכוונן הפוך נקראות קשתות ***אנטי מקבילות***.
מולטיגרף יכול להכיל קשתות מקבילות. 
גרף פשוט לא מכיל לולאות או קשתות מקבילות.

לכל קשת שני ***קדקודי קצה***, בין שני ***קדקודים שכנים*** עוברת קשת, ולשתי ***קשתות סמוכות*** יש קדקוד קצה משותף. קשת ***סמוכה*** לכל קדקוד קצה שלה. ***דרגת קדקוד*** היא מספר הקשתות שהוא מהוה קצה שלהן, ובגרף מכוון ***דרגת הכניסה*** שווה למספר הקשתות המכוונות אל הקדקוד, ***ודרגת היציאה*** שווה למספר הקשתות המכוונות מהקדקוד. קדקוד מדרגה 0 נקרא ***מבודד***.

הגרף $G'=(V',E')$ הוא ***תת-גרף*** של הגרף $G=(V,E)$ אם $V'\subseteq V$ ו- $E'\subseteq E$.
תת-הגרף $G'=(V',E')$ הוא ***תת-גרף מושרה*** של $G=(V,E)$ אם לכל $u,v\in V'$, $(u,v)\in E' \Leftrightarrow (u,v)\in E$.

שני גרפים $G=(V,E)$, $G'=(V',E')$ הם ***איזומורפיים*** אם קיימת פונקציה הפיכה חח"ע ועל $f:V\rightarrow V'$ כך ש- $(f(u),f(v))\in E' \Leftrightarrow (u,v)\in E$.

מסלול או ***מסילה*** הוא תת-גרף מהצורה $u_1—u_2—\ldots—u_n$ בגרף לא-מכוון ו- $u_1\rightarrow u_2\rightarrow\ldots\rightarrow u_n$ בגרף מכוון.
מעגל הוא תת-גרף מהצורה $u_1—u_2—\ldots—u_n$ בגרף לא-מכוון ו- $u_1\rightarrow u_2\rightarrow\ldots\rightarrow u_n$ בגרף מכוון.

במסלול/מעגל פשוט אין חזרה על קדקודים.

בגרף לא מכוון:
קליק הוא תת-גרף שבו כל שני קדקודים מחוברים בקשת.
בקבוצה בלתי תלויה (ב"ת) של קדקודים אין אף זוג קדקודים המחובר בקשת.

גרף דו-צדדי הוא גרף לא מכוון שניתן לחלק את קדקודיו לשתי קבוצות זרות, כך שלכל קשת בגרף יש קצה אחד בכל אחת מהקבוצות.

P_n הוא גרף לא מכוון שצורתו מסלול פשוט על n קדקודים.
C_n הוא גרף לא מכוון שצורתו מעגל פשוט על n קדקודים.
K_n הוא ***גרף שלם*** על n קדקודים, כלומר גרף לא מכוון שבו קיימת קשת בין כל שני קדקודים.
$K_{m,n}$ הוא גרף דו-צדדי שלם, כלומר יש לו m קדקודים בצד אחד, n קדקודים בצידו האחר וקשת בין כל שני קדקודים שאינם באותו צד.

u ***נגיש*** מ- v אם קיים מסלול (מכוון אם הגרף מכוון) מ- v אל u.
בגרף לא מכוון ***קשיר*** יש מסלול מכל קדקוד אל כל קדקוד אחר.
רכיב קשירות בגרף לא מכוון הוא תת-גרף קשיר מקסימלי, כלומר אין תת-גרף קשיר אחר שמכיל אותו.
גרף קשיר בחזקה הוא גרף מכוון שבו יש מסלול מכוון מכל קדקוד אל כל קדקוד אחר.
רכיב קשירות חזקה (רק"ח) הוא תת-גרף קשיר בחזקה מקסימלי, כלומר אין תת-גרף קשיר בחזקה אחר שמכיל אותו.

מקור בגרף מכוון הוא קדקוד שדרגת הכניסה שלו 0.
בור בגרף מכוון הוא קדקוד שדרגת היציאה שלו 0.

גרף מכוון אציקלי (DAG) הוא גרף מכוון חסר מעגלים מכוונים.
עץ הוא גרף לא מכוון **קשיר** וחסר מעגלים.
יער הוא גרף לא מכוון חסר מעגלים.
עלה הוא קדקוד מדרגה 1 בעץ או ביער.

טענות בסיסיות

טענה 1: לכל גרף לא מכוון $G = (V,E)$ מתקיים $\sum_{v\in V} d(v) = 2|E|$, כאשר $d(v)$ הדרגה של v.

הוכחה: מכיון שלכל קשת שני קצוות, אז כל קשת תורמת 2 לסכום דרגות הקדקודים. □

טענה 2: לכל גרף מכוון $G = (V,E)$ מתקיים $\sum_{v\in V} d_{in}(v) = \sum_{v\in V} d_{out}(v) = |E|$,

כאשר $d_{in}(v)$, $d_{out}(v)$ הן דרגות הכניסה והיציאה של v בהתאמה.

הוכחה: כל קשת מכוונת תורמת 1 לסכום דרגות היציאה של הקדקודים, ו- 1 לסכום דרגות הכניסה של הקדקודים. □

טענה 3: לכל גרף יש מספר זוגי של קדקודים מדרגה אי-זוגית.

הוכחה: נובע ישירות מטענה 1 (כי סכום הדרגות זוגי). □

טענה 4: מספר הקשתות בגרף שלם על n קודקודים הוא $n(n-1)/2$.

הוכחה: כל קדקוד סמוך ליתר $n-1$ הקדקודים, ואם נסכם את דרגות כל הקדקודים ונשתמש בטענה 1, נקבל את הנדרש. □

טענה 5: מספר הקשתות בעץ על n קודקודים הוא $n-1$.

הוכחה: באינדוקציה על מספר הקדקודים. □

טענה 6: בגרף קשיר ופשוט $G = (V,E)$ מתקיים $\Omega(|V|) = |E| = O(|V|^2)$.

הוכחה: גרף קשיר ופשוט מינימלי הוא עץ, וגרף קשיר ופשוט מקסימלי הוא גרף שלם, לכן $|V|-1 \le |E| \le |V|(|V|-1)/2$. □

Euler Circuit

EULER(graph G=(V,E))

// Find an Euler circuit.

```
L ← {v}                    // v∈V  any vertex ,{v} - an empty circuit

while there is a vertex on L with unused edges
      v ← first such vertex on L
      L' ← Find-Circuit(v)
      "paste" L'  into L instead of v

return L
```

FIND-CIRCUIT(v_0)

// Find a circuit starting at v_0. Return list of vertices.

```
L ← {v0}          // initialize list
v ← v0

repeat
      u ← a neighbour of v via an unused edge
      mark (v,u) used
      L ← Append(L,u)
      v ← u
until  v = v0

return L
```

total complexity O(E+V)

BFS

```
BFS(G,s)
for each vertex u ∈ V - {s}
   do color[u] ← WHITE
      d[u] ← ∞
      π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ Ø
   do u ← head[Q]
      for each v ∈ Adj[u]
         do if color[v] = WHITE
               then color[v] ← GRAY
                    d[v] ← d[u] + 1
                    π[v] ← u
                    ENQUEUE(Q,v)
      DEQUEUE(Q)
      color[u] ← BLACK
```

אינדקס
כל הקודקודים
פרט ל-s

אינדקס
s

עוברים על השכנים של u

אם לא רואים לדלג

DFS

```
DFS(G)
for each vertex u ∈ V
   do color[u] ← WHITE
      π[u] ← NIL
time ← 0
for each vertex u ∈ V
   do if color[u] = WHITE
         then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY
d[u] ← time ← time + 1
for each v ∈ Adj[u]
   do if color[v] = WHITE
         then π[v] ← u
              DFS-VISIT(v)
color[u] ← BLACK
f[u] ← time ← time + 1
```

DFS(graph G=(V,E))

```
// Init
for each vertex u∈V
        Colour[u] ← WHITE
        Parent[u] ← NULL
time ← 0

// Main Loop
for each vertex u∈V
        CurrentRoot ← u
        if Colour[u] = WHITE
                Visit(u)
```

VISIT(vertex u)

```
colour[u] ← GRAY
Root[u] ← CurrentRoot
d[u] ← ( ++ time )

for each v ∈Adj[u]
        if Colour[v] = WHITE
                mark (u,v) "TREE-EDGE"
                Parent[v] ← u
                Visit(v)
        else if Colour[v] = GRAY
                mark (u,v) "BACK-EDGE"
        else // v is BLACK
                if d[v] > d[u]
                        mark (u,v) "FORWARD EDGE"
                else // d[v] < d[u]
                        mark (u,v) "CROSS EDGE"

Colour[u] ← BLACK
f[u] ← ( ++ time )
```

Topological Sort

TOPOLOGICAL-SORT (graph G=(V,E))

```
Set S=∅;          // Set of sources
List L=∅;         // List of the vertices in a topological sort

// Init       compute in-degrees of the vertices in Indegree[v]
for each v∈V
      Indegree[v]← 0

for each (u,v)∈E
      Indegree[v]← Indegree[v]+1

for each v∈V
      if Indegree[v]=0
            add v to S

// Main  Loop
while S≠∅
      v ← remove a vertex from S
      append  v  to L
      for each u∈Adj[v]
            Indegree[u]← Indegree[u]-1
            if Indegree[u]=0
                  add u to S

//Check if all vertices reached  L
for each v∈V
      if Indegree[v]≠0
            return "CYCLE!"   // no topological sort found
return L                      // topological sort of all vertices
```

total complexity O(E+V)

Single-Source Shortest Paths

```
INITIALIZE-SINGLE-SOURCE(G,s)
for each v ∈ V
   do d[v] ← ∞
      π[v] ← NIL
d[s] ← 0
```

```
RELAX(u,v,w)
if d[v] > d[u] + w(u,v)
   then d[v] ← d[u] + w(u,v)
        π[v] ← u
```

```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S ← Ø
Q ← V  // priority queue
while Q ≠ Ø
   do u ← EXTRACT-MIN(Q)
      S ← S ∪ {u}
      for each v ∈ Adj[u]
         do RELAX(u,v,w)
```

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V| - 1
    do for each edge (u, v) ∈ E
            do RELAX(u, v, w)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
            then return FALSE
return TRUE
```

algorithm for shortest paths in DAG

DAG-Shortest-Paths(graph G, vertex s, weight w)

```
//Pre-Algorithm
Topological-Sort(G)

Initialize(G, s)

// Main Loop
for each vertex u∈V          //taken in topologically sorted order
    for each v∈ Adj[u]
        Relax(u, v, w)
```

total complexity O(E+V)

All-Pairs Shortest Paths

```
EXTEND-SHORTEST-PATHS(D,W)
n ← rows[D]
let D′ = (d′ij) be an n × n matrix
for i ← 1 to n
   do for j ← 1 to n
         do d′ij ← ∞
            for k ← 1 to n
               do d′ij ← min(d′ij,dik + wkj)
return D′
```

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
n ← rows[W]
D(1) ← W
for m ← 2 to n - 1
   do D(m) ← EXTEND-SHORTEST-PATHS(D(m-1),W)
return D(n-1)
```

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
n ← rows[W]
D(1) ← W
m ← 1
while m < n- 1
   do D(2m) ← EXTEND-SHORTEST-PATHS(D(m),D(m))
m ← 2m
return D(m)
```

```
FLOYD-WARSHALL(W)
n ← rows[W]
D^(0) ← W
for k ← 1 to n
   do for i ← 1 to n
         do for j ← 1 to n
               do d^(k)_ij ← min(d^(k-1)_ij, d^(k-1)_ik + d^(k-1)_kj)
return D^(n)
```

```
JOHNSON(G)
compute G'=(V',E'), where V' = V ∪ {s},
 E' = E ∪ {(s,v) | v ∈ V }, and ∀ v ∈ V w(s,v) = 0
if BELLMAN-FORD(G',w,s) = FALSE
   then print "G contains a negative-weight cycle"
else for each vertex v ∈ V'
         do h(v) ← δ(s,v) computed by BELLMAN-FORD
      for each edge (u,v) ∈ E'
         do w'(u,v) ← w(u,v) + h(u) - h(v)
      for each vertex u ∈ V
         do run DIJKSTRA(G,w',u) to compute
             δ'(u,v) for all v ∈ V
            for each vertex v ∈ V
               do d_uv ← δ'(u,v) + h(v) - h(u)
return D
```

Minimum Spanning Trees

```
MST-KRUSKAL(G,w)
A ← Ø
for each vertex v ∈ V
   do MAKE-SET(v)
sort E by nondecreasing weight w
for each edge (u,v) ∈ E, by nondecreasing weight order
   do if FIND-SET(u) ≠ FIND-SET(v)
         then A ← A ∪ {(u,v)}
              UNION(u,v)
return A
```

```
MST-PRIM(G,w,r)
Q ← V
for each u ∈ Q
   do key[u] ← ∞
key[r] ← 0
π[r] ← NIL
while Q ≠ Ø
   do u ← EXTRACT-MIN(Q)
      for each v ∈ Adj[u]
         do if v ∈ Q and w(u,v) < key[v]
               then π[v] ← u
                    key[v] ← w(u,v)
```

UNION-FIND
data structure for disjoint sets

מבנה נתונים המייצג אוסף של קבוצות זרות (במקרה הפשוט, סדרה עולה של מספרים).

לכל קבוצה יש שם ייחודי (נציג הקבוצה, id) הנבחר על ידי מנגנון מבנה הנתונים

Supports next operations:

- MakeSet (v) - creates set v
- Find (v) - returns id of the set that v belongs to
- Union (s_1,s_2) - unites sets s_1 and s_2

Priority Queue – תור הקדימויות

	בניית תור עם V איברים	הוצאת המינימום	הקטנה של ערך המפתח
	Build	DeleteMin	DecreaseKey
עץ חיפוש מאוזן	O(Vlog V)	O(log V)	O(log V)
רשימה ממוינת	O(V log V)	O(1)	O(V)
מערך לא ממוין	O(V)	O(V)	O(1)
ערימה בינארית	O(V)	O(log V)	O(log V)
ערימת פיבונצ'י	O(1)	O(log V)	O(1)

V פעמים E פעמים

Ford-Fulkerson

```
FORD-FULKERSON-METHOD(G,s,t)
Initialize flow f to 0 everywhere
while there exists an augmenting path p
   do augment flow f along p
return f
```

```
FORD-FULKERSON(G,s,t)
   // Initialize
   for each edge (u,v) ∈ E(G)
      do f[u,v] ← 0
         f[v,u] ← 0
   // Main loop
   repeat until f does not change
      // Construct residual network
      E_f ← 0
      for each edge (u,v) ∈ E[G]
         do c_f[u,v] ← c[u,v]-f[u,v]
            if c_f[u,v] > 0
               then add (u,v) to E_f
      // Augment the flow
      if there exists a path p from s to t in
         the residual network G_f=(V,E_f)
         do c_f(p) ← min { c_f[u,v] | (u,v) ∈ p }
            for each edge (u,v) ∈ p
               do f[u,v] ← f[u,v]+c_f(p)
                  f[v,u] ← -f[u,v]
   return f
```

String Matching

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← q0
for i ← 1 to n
   do q ← δ(q, T[i])
      if q = qm
         then print "pattern occurs with shift" i - m
```

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for j ← 0 to m
   do for each character a ∈ Σ
         do k ← min(m, j + 1)
            while Pk ⊋ Pja
               do k ← k - 1
            δ(qj, a) ← qk
return δ
```

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
j ← 0
for i ← 1 to n
   do while j > 0 and P[j + 1] ≠ T[i]
         do j ← π[j]
      if P[j + 1] = T[i]
         then j ← j + 1
      if j = m
         then print "pattern occurs with shift" i - m
              j ← π[j]
```

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for j ← 2 to m
   do while k > 0 and P[k + 1] ≠ P[j]
         do k ← π[k]
      if P[k + 1] = P[j]
         then k ← k + 1
      π[j] ← k
return π
```