به نام خدا

ساختمان داده ها

جلسه بیست ویکم

دانشگاه بوعلی سینا

گروه مهندسی کامپیوتر

نیم سال دوم 98-1397

گرافها

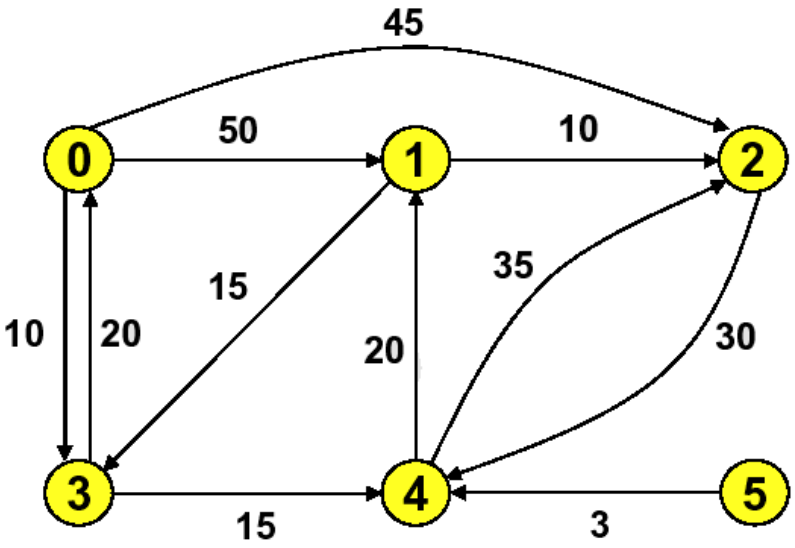
Graphs

مطالب این فصل

- مقدمه و تعاریف
- نمایش گراف
  - ماتریس مجاورتی
  - لیست مجاورتی
  - لیست مجاورتی چندگانه
  - یالهای وزن دار
- اعمال روی گرافها
  - جستجوی عمقی
  - جستجوی ردیفی
  - مولفه های همبند
  - درختهای پوشا
  - مولفه های دو اتصالی
- درخت پوشای کمترین هزینه
  - الگوریم راشال
  - الگوریتم پریم
  - الگوریتم سولین
- کوتاهترین مسیر
  - یک مبدا چند مقصد
  - بین دو زوج راس
  - بستار متعدی
- شبکه های فعالیت

یک مبدا و چند مقصد

در این مساله گراف جهت دار $G = (V, E)$ را در نظر مي گيريم. تابع وزني $w(e) > 0$ را براي لبه هاي G و راس مبدا v_0 فرض مي كنيم. مساله ، تعيين كوتاهترين مسير از v_0 به بقيه رئوس در G است. فرض مي كنيم تمام وزن ها مثبت باشند.



Starting Vertex: 0

Path	Length
1) 0, 3	10
2) 0, 3, 4	25
3) 0, 3, 4, 1	45
4) 0, 2	45

Increasing

الگوریتم پیدا کردن کوتاه ترین مسیر (دایکسترا)

- فرض کنید S مجموعه رئوسی باشد که دارای کوتاهترین مسیر است که تاکنون بدست آمده است.

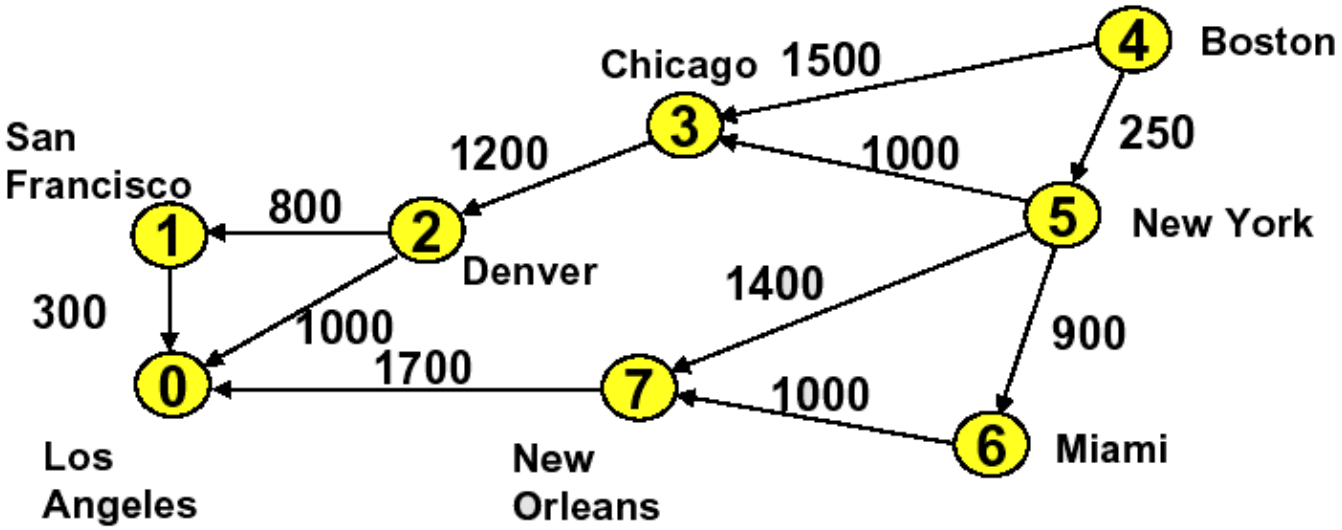
1. اگر کوتاهترین مسیر بعدی به راس u باشد آن گاه مسیری که از V آغاز شده و به u ختم می شود فقط از راسهایی می گذرد که در S هستند.
2. بین همه راسهایی که در S نیستند مقصد مسیر بعدی باید راس u باشد که کمترین فاصله را دارد.
3. با انتخاب راس u و تولید کوتاهترین مسیر از v به u ، u در S ذخیره می شود.

```
// nmax: maximum number of vertices
class Graph {
  int length[nmax][nmax];  // length-adjacency matrix
  int dist[nmax];  // min-distance
  bool s[nmax];
  int choose(int); // used in ShortestPath
public:
  void ShortestPath(int, int);
};

// length[i][j] is set to a large number
// if <i, j> is not an edge
```

```
void Graph::ShortestPath(int n, int v) {
// n vertices, source vertex = v
  for(int i = 0; i < n; ++i) { // initialize
    s[i] = false;  dist[i] = length[v][i];
  }
  s[v] = true; dist[v] = 0;
  for(int j = 0; j < n-2; ++j) { // determine n-1 paths
    int u = choose(n);
    // choose u s.t. dist[u] is minimum where s[u] = false
    s[u] = true;
    for(int w = 0; w < n; ++w) {
    // update dist[w] where s[w] is false
    if( (! s[w]) && (dist[u] + length[u][w] < dist[w]) )
      dist[w] = dist[u] + length[u][w]; // shorter path found
  }
```

مثال



	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4			0	1500	0	250		
5			0	1000		0	900	1400
6			0				0	1000
7	1700		0					0

Length-adjacency matrix

iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	--	---	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
3	{4,5,6}	3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
	{4,5,6,3,7,2,1}									

کوتاهترین مسیر بین هر جفت از رئوس

$A^k[i][j]$ را طول کوتاهترین مسیر از **i** به **j** تعریف مي کنيم که اين مسير هيچ راس مياني که انديس بزرگتر از **k (index > k)** داشته باشيم ، عبور نمي کند.

$A^{n-1}[i][j]$ هزينه کوتاهترين مسير از **i** به **j** در **G** است.

نکته اصلي در همه الگوريتم هايي که روي زوج ها عمل مي کنند ، اين مي باشد که همه آنها با A^0 شروع کرده و ماتريس هاي متوالي $A^0, A^1, A^2, ..., A^{n-1}$ را ايجاد مي کند.

کوتاهترین مسیر بین هر جفت از رئوس

اگر قبلا A^{k-1} را توليد كرده باشيم ، مي توانيم A^k را با عمل روي هر زوج از رئوس i و j با توجه به يكي از دو قاعده زير به دست آوريم :

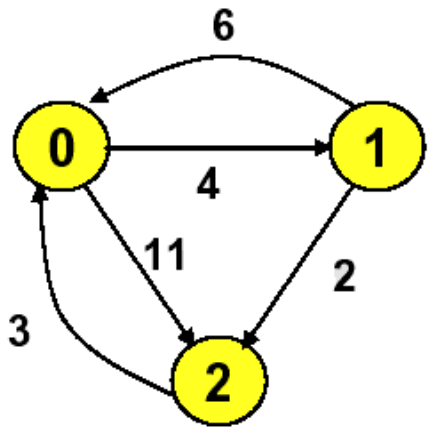
1) كوتاهترين مسير از i به j از هيچ راسي با انديس بزرگتر از k عبور نمي كند و حتي از راسي با انديس k نيز نمي گذرد و بنابراين هزينه آن $A^{k-1}[i][j]$ است.
2) چنين كوتاهترين مسيري اگر از راس k بگذرد شامل مسيري از i به k و به دنبال آن مسيري از k به ت مي باشد. هيچ كدام از اين مسيرها با انديس بزرگتر از k-1 عبور نخواهد كرد ، بنابراين هزينه آن ها $A^{k-1}[k][j]$ و $A^{k-1}[i][k]$ مي باشد.

کوتاهترین مسیر بین هر جفت از رئوس

فرمول 1 :
$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$$

فرمول 2 :
$$A^{-1}[i][j] = length[i][j]$$

گراف جهت دار G و ماتریس هزینه آن (مثال)



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^{-1}

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^{0}

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{1}

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

A^{2}

```
void Graph::AllLengths(int n) {
// length[n][n] is the length-adjacency matrix
// a[i][j] is the length of the shortest path between i and j

  for( int i = 0; i < n; ++i)
    for ( int j = 0; j < n; ++i)
      a[i][j] = length[i][j]; // construct A^-1

 for(int k = 0; k < n; ++k)
   for(int i = 0; i < n; ++i)
     for(int j = 0; j < n; ++j)
        if( a[i][j] > (a[i][k] + a[k][j]) )
          a[i][j] = a[i][k] + a[k][j];
}
```

مرتب سازی وجستجو

Searching & Sorting

مطالب این فصل

- جستجو خطی
- جستجوی دودویی
- مرتب سازی درجی
- مرتب سازی سریع
- مرتب سازی ادغام
- مرتب سازی هرمی

مرتب سازي

- [illegible]
- [illegible]
- [illegible]
- [illegible]

اصطلاحات

- زمانی که یک لیست از رکوردها را جستجو می کنیم ، هدف پیدا نمودن رکوردهایی است که دارای فیلدی با مشخصات مورد نیاز باشند. این فیلد را کلید می نامند.

- کارایی روش و خط مشی جستجو بستگی به آرایش و نحوه قرار گرفتن رکوردها در لیست دارد.

جستجوي ترتيبي يا خطی

فرض کنید که لیست و یک کلید جستجو به نام Searchnum داشته باشیم . هدف بازیابی رکوردی است که کلید آن منطبق بر searchnum باشد. اگر این لیست دارای n رکورد باشد ، با list[i].key به مقدار کلید رکورد i دسترسی پیدا می کنیم، لیست را با جستجوی مقادیر کلیدهای list[0].key. . .list[n-1].key مورد بررسی قرار می دهیم تا به رکورد مورد نظر برسیم یا تمام لیست را جستجو کنیم.

جستجوي ترتيبي يا خطی

```
class Element {
  int key;
  // other fields;
public:
  int getKey() const { return key; }
  void setKey(int k) { key = k; }
  // ...
}

int SeqSearch(Element *f, int n, int k) {
// f is [1:n] array, seach if f[i].key == k
  int i = n;
  f[0] = setKey(k); // f[0] is a sentinel
  while(f[i].getKey() != k) --i;
  return i; // return 0 → not found!
}
```

Time complexity: O(n) **Better search method?**

جستجوي دودويي

در جستجوي دودويي بايستي ليست بر اساس فيلد كليد مرتب شود ، يعني :

$$list[0].key \leq list[1].key \leq \ldots \leq list[n-1].key$$

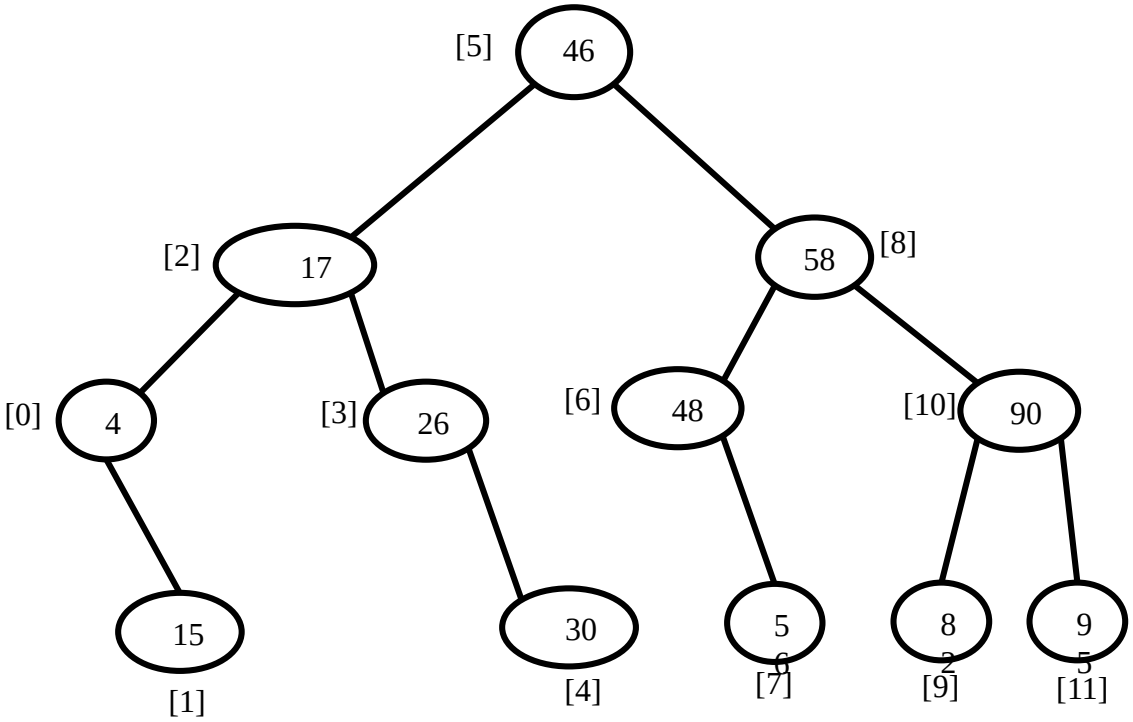
جستجوي دودويي

این جستجو با مقایسه searchnum و list[middle].key ، به ازای middle=(n-1)/2 شروع می شود.

هنگام مقایسه سه حالت ممکن است روی دهد :

1) **searchnum< list[middle].key : در این حالت رکوردهایی بین list[middle] و list[n-1] کنار گذاشته شده و جستجو با رکوردهای List[0] تا List[middle-1] دنبال می شود.**
2) **Searchnum= list[middle].key : در این حالت جستجو با موفقیت به پایان می رسد.**
3) **searchnum> list[middle].key : در این حالت ، رکوردهای بین List[0] و list[middle] کنار گذاشته شده و جستجو با رکوردهای list[middle+1] تا list[n-1] دنبال می شود.**

درخت تصميم گيري براي جستجوي دودويي



مرتب سازي

- مرتب سازی می تواند در موارد زیر مفید باشد:
  - در جستجوی لیستها
  - در بررسی ورودیهای لیستها

- مرتب سازی را می توان به طور کلی به دو دسته تقسیم کرد:
  - مرتب سازی داخلی
    - لیست به اندازه کافی کوچک است و در حافظه اصلی جا می گیرد
    - مرتب سازی درجی, ادغام ، سریع و هرمی
  - مرتب سازی خارجی
    - در این حالت لیست خیلی بزگ است و در حافظه اصلی جا نمی گیرد و در درون دیسک یا نوار عمل مرتب سازی انجام می گیرد.

مرتب سازي درجي Insertion Sort

در اين نوع مرتب سازي ، در هر لحظه فقط يک رکورد در داخل ليست قابل رويت است بنابراين رکور R_i را در بين رکوردهاي مرتد $R_0, R_1, ..., R_{(i-1)}$ طوري قرار مي دهيم که رشته حاصل با اندازه i نيز مر $K_0 \le K_1 \le ... \le K_{i-1}$.

```
void insert(const Element e, Element *list, int i) {
// insert an Element e into an ordered sequence list[1]~
// list[i] and list[0] is a sentinel with the smallest key
  while(e.getKey() < list[i].getKey()) {
    list[i+1] = list[i];
    --i;
  }
  list[i+1] = e;
}
```

Time Complexity: O(i)

```
void InsertSort(Element *list, int n) {
// sort the given list in non-decreasing order of key
  list[0] = MININT; // list[0] works as a sentinel
  for(int j = 2; j <= n; ++j)
    insert(list[j], list, j-1);
}
```

Record R_i is left out of order (LOO) iff $R_i < \max_{1 \le j < i} \{ R_j \}$

Time Complexity: $O(n^2)$

Insertion sort is stable

The simplicity makes it the fastest sorting method for about $n \le 20$

تحلیل تابع insertion sort

زمان محاسباتي جهت درج یک رکورد به داخل لیست مرتب شده ، $O(i)$ خواهد بود.

زمان کل در بدترین حالت برابر است با :

$$O(\sum_{i=0}^{n-1} i) = O(n^2)$$

مرتب سازي درجي(مثال)

Case 1:
worst-case of insertion sort

j	[1]	[2]	[3]	[4]	[5]
-	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

Case 2:
only R_5 is LOO, just need to move R_5

j	[1]	[2]	[3]	[4]	[5]	
-	**2**	3	4	5	1	
2	**2**	**3**	4	5	1	O(1)
3	**2**	**3**	**4**	5	1	O(1)
4	**2**	**3**	**4**	**5**	1	O(1)
5	**1**	**2**	**3**	**4**	**5**	O(n)

مرتب سازي سريع (Quick Sort)

این روش در بین همه مرتب سازیها دارای بهترین متوسط زمانی است.

درمرتب سازي سريع کليد محور يا مفصل (K_i) (pivot) که عمل درج را کنترل می کند با توجه به زیر لیست مرتب شده ($R_1,\ldots R_{i-1}$) در مکان صحیح قرار می گیرد. یعنی K_i نسبت به کل لیست در جای صحیح قرار می گیرد.

یعنی اگر $K_j \le K_{s(i)}$ در محل $s(i)$ قرار بگیرد ، پس به ازاي $j <= s(i)$ داريم $j > s_{(i)}$ $K_j \ge K_{s(i)}$

، داريم $R_0,\ldots, R_{s(i)-1}$ بوده و به ازاي $R_{s(i)+1},\ldots, R_{n-1}$. از اين رو بعد از جايگذاري ، ليست اصلي به دو زير ليست که شامل رکوردهاي و مي باشند ، تقسيم مي شوند.

مرتب سازي سريع

```
void QuickSort(Element *list, int left, int right) {
// sort list[left] ~ list[right]
// put list[left] to the correct position and
// partition the given list into 2 sublists
  if(left < right) {
    int i = left;
    int j = right + 1;
    int pivot = list[left].GetKey();
    do {
      do {++i;} while (i < right && list[i].GetKey() < pivot);
      do (--j;} while (list[j].GetKey() > pivot);
      if(i < j) interchange(i, j);
    } while (i < j);
    interchange(left, j);
    QuickSort(list, left, j-1);
    QuickSort(list, j+1, right);
  }
}
```

مرتب سازي سريع

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	Left	Right
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

مرتب سازي سريع

در بد ترین حالت زمان مرتب سازی سریع o(n^2) است.

- در این حالت لیست اولیه مرتب می باشد.

ميانگين زمان محاسبه براي مرتب سازي سريع $O(n \log_2 n)$ **مي باشد.**