طراحی و تحلیل الگوریتم ها

دکتر امیر لکی زاده

استادیار گروه مهندسی کامپیوتر دانشگاه قم

چهارمین روش طراحی الگوریتم: BackTracking: $O(2^n)$

BackTracking در حل مسائلی به کار می‌رود که در آنها یک دنباله (sequence) از اشیا از یک مجموعه مشخص (set) انتخاب شود و دنباله باید در بعضی شرایط صدق کند (criterion) صدق کند.

مثال: مسئله n وزیر (n-queen Problem): قرار دادن n وزیر در صفحه شطرنج nxn بطوریکه هیچ کدام یکدیگر را تهدید نکنند.

دنباله: n مکان مناسب برای وزیرها

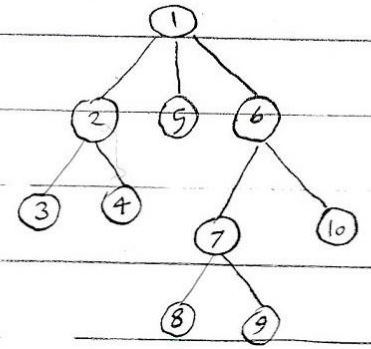
مجموعه: تمامی n^2 مکان ممکن در صفحه شطرنج

شرط صدق کننده: هیچ دو وزیری همدیگر را تهدید نکنند.

مثال: مسئله Maze می‌باشد.

BackTracking یک جستجوی اول عمق (DFS) در درخت فضای حالت (State Space tree) می‌باشد

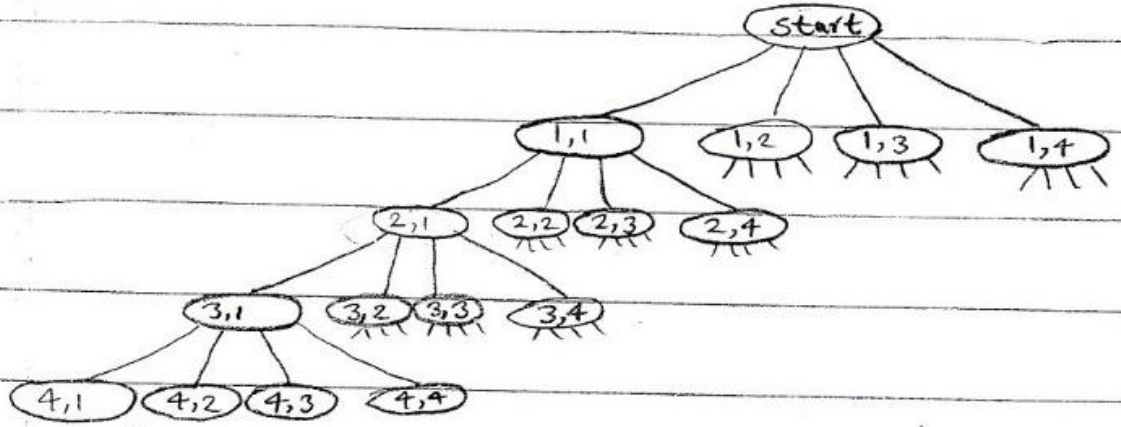
هر مسیر از ریشه به یک برگ در درخت فضای حالت یک جواب کاندید (candidate solution) می‌باشد.



درخت فضای حالت

در مسئله n-queen اگر $n=4$ باشد تعداد جواب های کاندیدا برابر است با: $4 \times 4 \times 4 \times 4 \ (n^n)$

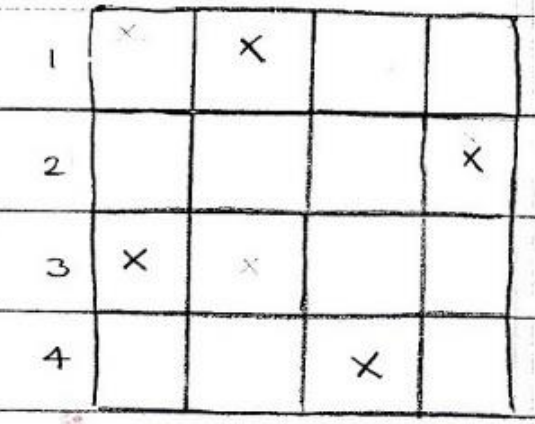
درخت فضای حالت در حالت کلی برای $n=4$



به عبارت دیگر BackTracking یک جستجوی اول عمق در درخت فضای حالت می باشد در این جستجو الگوریتم هر گره را بررسی (checking) می کند (هر گره نزدیک راه حل ساخته نشده تا کنون) اگر این گره راه حل امیدوار کننده (promising) بود گره را گسترش (Expand) می دهد وگر گره امیدوار کننده نبود (non promising) به گره والد برگشت به عقب می کند. این عملیات هرس زدن (Pruning) درخت فضای حالت نامیده می شود، زیر درخت شامل گره های ملاقات نشده را pruned state space tree می گویند

یک الگوریتم کلی برای BackTracking:

```
void checknode(v){
node u;
if(promising(v))
    if(there is a solution at v)
        write the solution;
    else
        for(each child u of v) // left to right
            checknode(u);
}
```



به جهت پیاده‌سازی بهتر می‌توان به جای تابع checknode از تابع زیر استفاده کرد (در پیاده سازی BackTracking از حافظه Stack استفاده می‌شود نیز Stack می‌تواند صریح یا ضمنی باشد)

```
void expand (node v){
 node u;
 for (each child u of v)
  if((promising(u))
   if(there is a solution at u)
    write the solution
   else
    expand(u);
}
```

cal[i]: ستون متناظر با وزیر i ام (وزیر سطر i ام)

$cal[i] \neq cal[x]$ برای رفع تهدید ی ستونی دو وزیر i ام و x ام

برای رفع تهدید ی قطری:

$|cal[x] - cal[i]| \neq |x - i|$

اختلاف سطر ی دو وزیر باید مخالف اختلاف ستون ی دو وزیر باشد.

```
void queens (index i){
    index j ;
    if (promising(i))
        if (i==n)
            cout << col[1] through col[n] ;
        else
            for (j=1 ; j<=n ; ++j){
                col[i+1] = j ;
                queens(i+1);
            }
}
```

می خواهد مکان (i+1) امین وزیر را بدست آورد.

فراخوانی queens(0)

تمرین: غیر بازگشتی

اول وزیر قرار داده می‌شود

بعد Promising فراخوانی می‌شود

فراخوانی اولیه: queens(0)

```
bool promising (index i){
    index k;
    k = 1;
    bool switch = true;
    while (k<i && switch){
        if ((col[i] == col[k]) || (|col[i] - col[k]| == |i-k|))
            switch = false;
        k++;
    } //end while
    return (switch);
}
```

اگر تعداد کل گره ی درخت فضای حالت به عنوان یک حد بالا برای تعداد گره ی درخت فضای حالت حرص تدریجی باشد

$$1 + n + n^2 + \dots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

(ریشه)

$$n = 8 \quad \frac{8^{8+1} - 1}{8 - 1} = 19,173,961$$

حد بالا برای گره‌های درخت فضای حالت هرس شده (حد بالا برای گره‌های امیدوار کننده)

فقط برای تعریف‌های مترنی لحاظ شود. $i + n + n(n-1) + n(n-1)(n-2) + \dots + n!$

$n = 8 \quad : \quad i + 8 + 8(7) + \dots + 8! = 109,601$

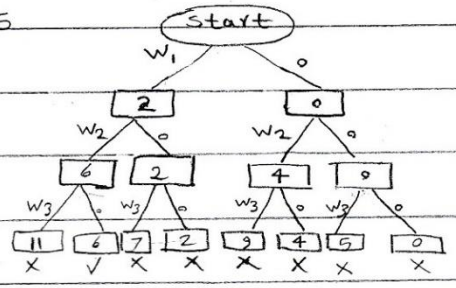
Sum_of_subsets problem

n شی هرکدام با وزن w_i $(w_i > 0)$

هدف: پیدا کردن همه زیر مجموعه‌ای از $S = \{w_1, w_2, \dots, w_n\}$ بطوریکه مجموع عناصر هر زیرمجموعه انتخابی برابر w شود.

$w_1 = 2 \quad w_2 = 4 \quad w_3 = 5$

$w = 6$



درخت فضای حالت

$\{w_1, w_2\}$

weight: در هر گام مجموع عناصر انتخابی تا آن گام را نشان می‌دهد.

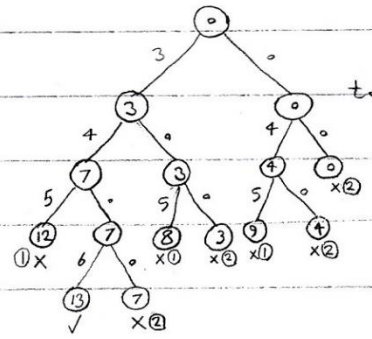
total: جمع وزن‌های اشیا باقی مانده.

nonpromising(i)

1) weight + w_{i+1} > w

2) weight + total < w

$w_1 = 3 \quad w_2 = 4 \quad w_3 = 5 \quad w_4 = 6 \quad w = 13$



// $w_1 \leq w_2 \leq \dots \leq w_n$ مراخرای sum_of_subsets(0, 0, total)

$total = \sum_{i=1}^{n} w_i$

```
void sum_of_subsets(index i, int weight, int total){
  if (Promising(i))
    if (weight==w) cout<<include[1] through include[i]
  else{ include[i+1]<-true
    sum_of_subsets(i+1, weight+w[i+1], total-w[i+1]);
    include[i+1]<-false;
    sum_of_subsets(i+1, weight, total-w[i+1]);
    }
}
bool promising(index i){
return((weight+total>=w) && ((weight==w)||(weight+w[i+1]<=w)))
}
```

تعداد کل گره‌ای درخت فضای حالت $= 2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$

بدترین حالت برای درخت فضای $\sum_{i=1}^{n-i} w_i < w \quad w_n = w$ ↑

حالت هرس شده

روش پنجم

Branch_and_Bound

۱- B&B مانند BT از طریق جستجو در درخت فضای حالت مسئله را حل می‌کند.

۲- تفاوت B&B و BT

الف: در B&B جستجو در درخت فضای حالت از نوع اول بهترین یا اول هزینه می‌باشد.

Breadth first search with Branch and Bound pruning

Best " " " " " " "

ب: کاربرد B&B در مسائل بهینه‌سازی

در الگوریتم‌های B&B یک Bound برای هر گره در درخت فضای حالت تعیین می‌شود که این Bound یک حد بالا برای بهترین جواب است که از طریق گسترش گره مورد نظر تا رسیدن به یک جواب نهایی می‌توانیم بدست آید.

اگر مقدار این Bound از بهترین جواب پیدا شده تا کنون، بهتر نباشد گره مربوط nonpromising می‌باشد

مثال ۱: یک الگوریتم BT (جستجو نصورت BFS) برای مسئله Knapsack 0/1

weight: مجموع وزن اشیا انتخاب شده

profit: مجموع ارزش اشیا انتخاب شده

maxprofit: بهترین جواب پیدا شده تا کنون

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

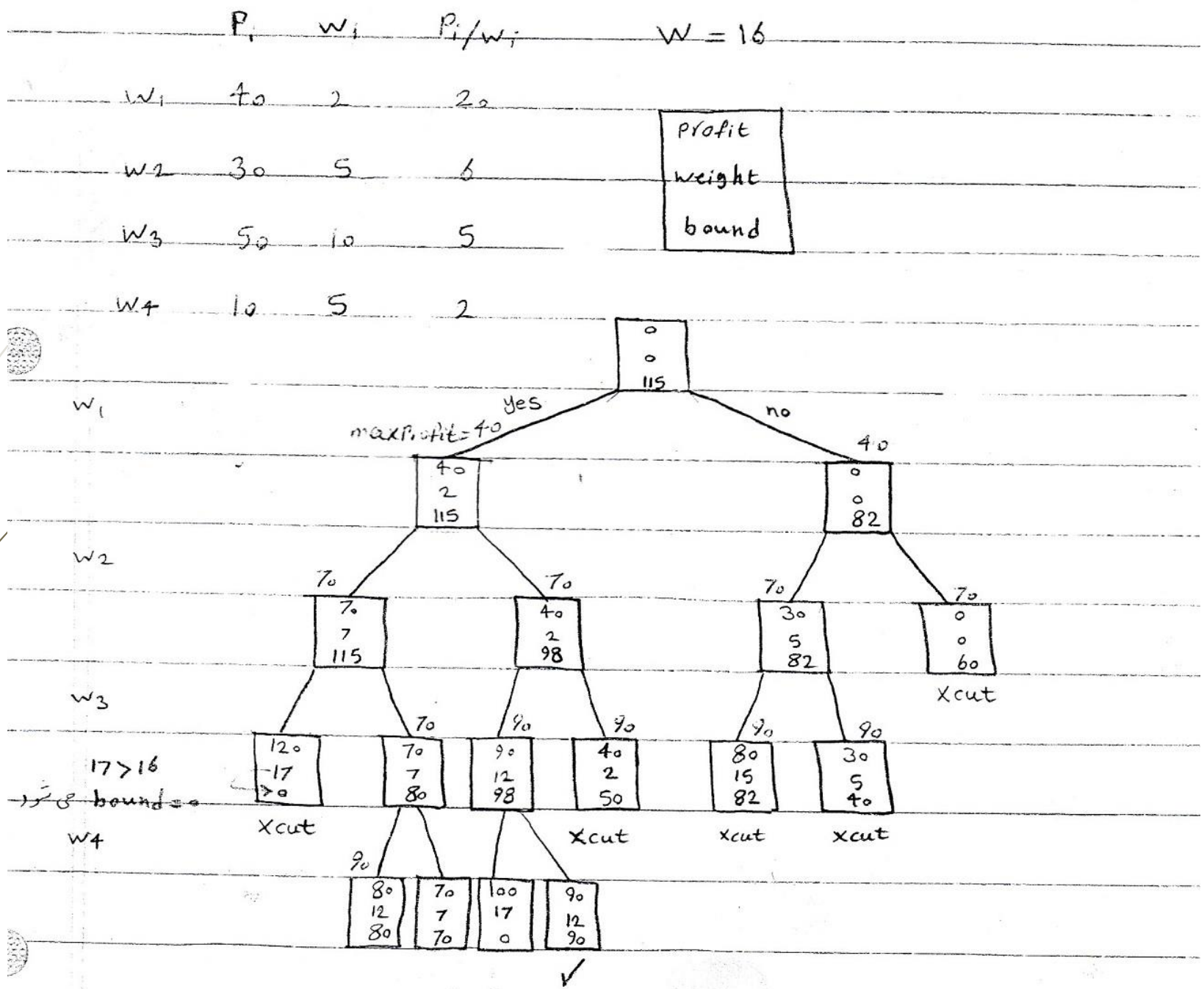
سطح k اولین جایی است که $totweight + w_k > w$

$$bound = (profit + \sum_{j=i+1}^{k-1} p_j) + (w - totweight)\frac{p_k}{w_k}$$

یک حد بالا برای Profit جواب بدست آمده از طریق گسترش این گره

$$nonpromising(i): \begin{cases} weight \geq w \\ bound \leq maxprofit \end{cases}$$

توجه: در الگوریتم وقتی $weight \geq w$ (یعنی گره امیدوار کننده نباشد) bound=0 می‌شود.


P_i w_i P_i/w_i W = 16
w1 40 2 20
w2 30 5 6
w3 50 10 5
w4 10 5 2
profit
weight
bound
w1
w2
w3
w4
yes
no
maxProfit=40
17>16
bound
Xcut

```
void knapSack2(int n, const int P[], const int w[], int W, int& maxprofit)
{ queue of node Q; node u, v;
  initialize(Q); // Q ← ∅
  v.level = 0; v.profit = 0; v.weight = 0;
  maxprofit = 0;
  enqueue(Q, v);
  while(!EMPTY(Q))
  { v = dequeue(Q);
    u.level = v.level + 1;
    u.weight = v.weight + w[u.level];
    u.profit = v.profit + P[u.level];
    if(u.weight <= W && u.profit > maxprofit)
        maxprofit ← u.profit;
    if(bound(u) > maxprofit)
        enqueue(Q, u);
    u.weight = v.weight;
    u.profit = v.profit;
    if (bound(u) > maxprofit)
        enqueue(Q, u);
  } //end while
} //end function
```

P = ⟨P1, P2, ..., Pn⟩

W = ⟨w1, w2, ..., wn⟩

```
struct node{
int level;
int profit;
int weight;
};
```

```
float bound(node u)
{ index j,k;
  int totweight;
  float result;
  if(u.weight>=w)  return 0;
  else{
       result = u.profit;
       j = u.level + 1;
       totweight = u.weight;
       while(j<=n && totweight+w[j]<w)
       { totweight = totweight+w[j];
         result = result + P[j];
         j++;
       }
       if(j<=n)
         result += (w - totweight) * P[j]/w[j];
         return result;
     }
}
```