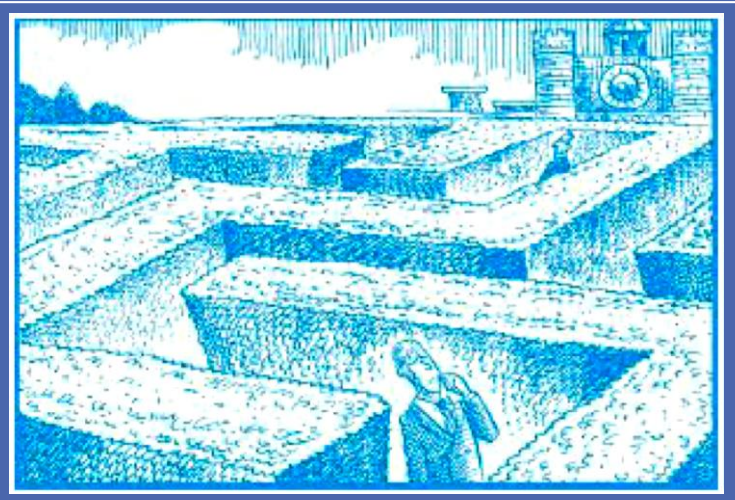
به نام خدا

فصل پنجم

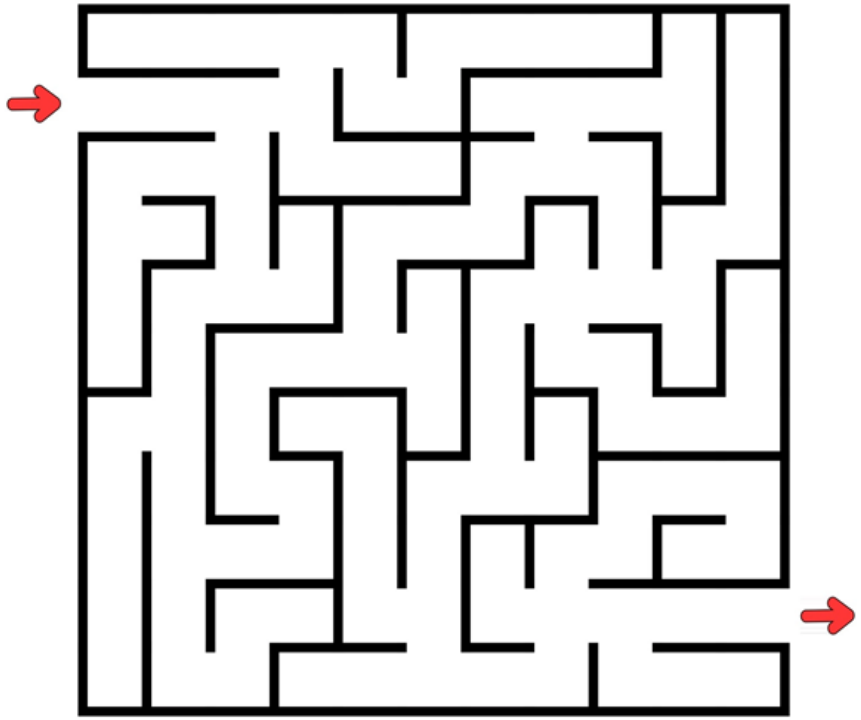
روش عقبگرد در طراحی الگوریتم

The Backtracking Technique



دانشکده مهندسی کامپیوتر – دانشگاه اصفهان
دکتر مرجان کائدی

روش عقبگرد



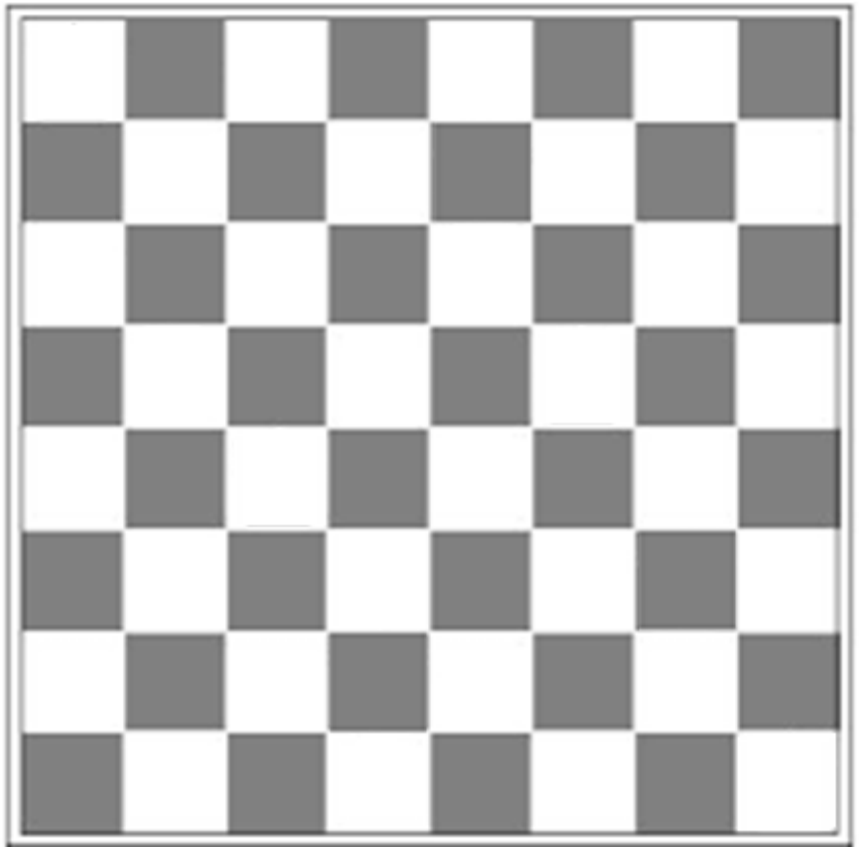
- حرکت در یک مسیر پر پیچ و خم (maze) را تصور کنید.
- فرض کنید تابلوهایی وجود دارند که مسیرهایی که به بن بست منتهی می شوند را نشان دهند. نتیجه: صرفه جویی در زمان برای رسیدن به هدف
- در الگوریتم های عقبگرد، در حین جستجو برای یافتن جواب مساله، این تابلوها در فضای جستجو وجود دارند.
- الگوریتم های عقبگرد برای اکثر مسائل دارای پیچیدگی زمانی نمایی (یا حتی بدتر از نمایی) هستند. ولی این الگوریتم ها از این رو مفید هستند که برای بسیاری از نمونه مساله ها با اندازه بزرگ موثر هستند (نه همه مساله‌ها)

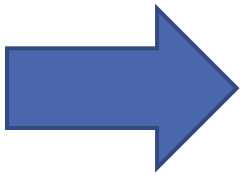
روش عقبگرد

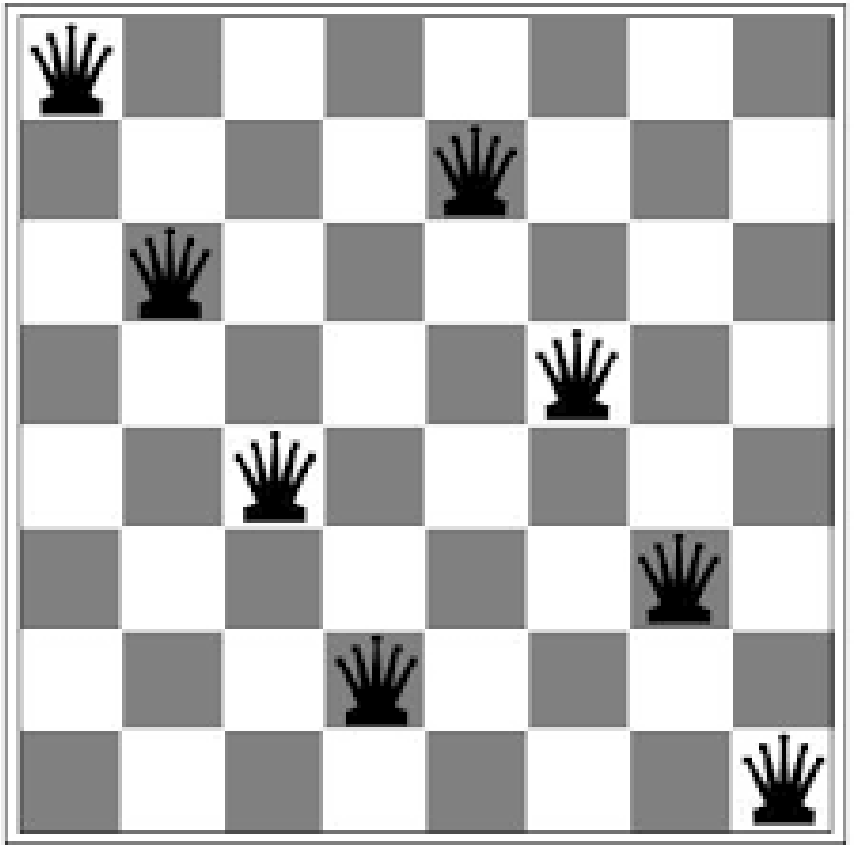
- از روش عقبگرد برای مسائلی استفاده می شود که در آنها دنباله ای از اشیاء از یک مجموعه مشخص انتخاب می شود به طوری که این دنباله انتخاب شده معیاری را برآورده کند.

- یک مثال: مساله چیدن n مهره وزیر در صفحه شطرنج n*n به نحوی که هیچ دو وزیری همدیگر را گارد ندهند.

در یک نمونه از این مساله، صفحه شطرنج 8*8 است (یعنی یک صفحه استاندارد) و ۸ وزیر هم داریم.





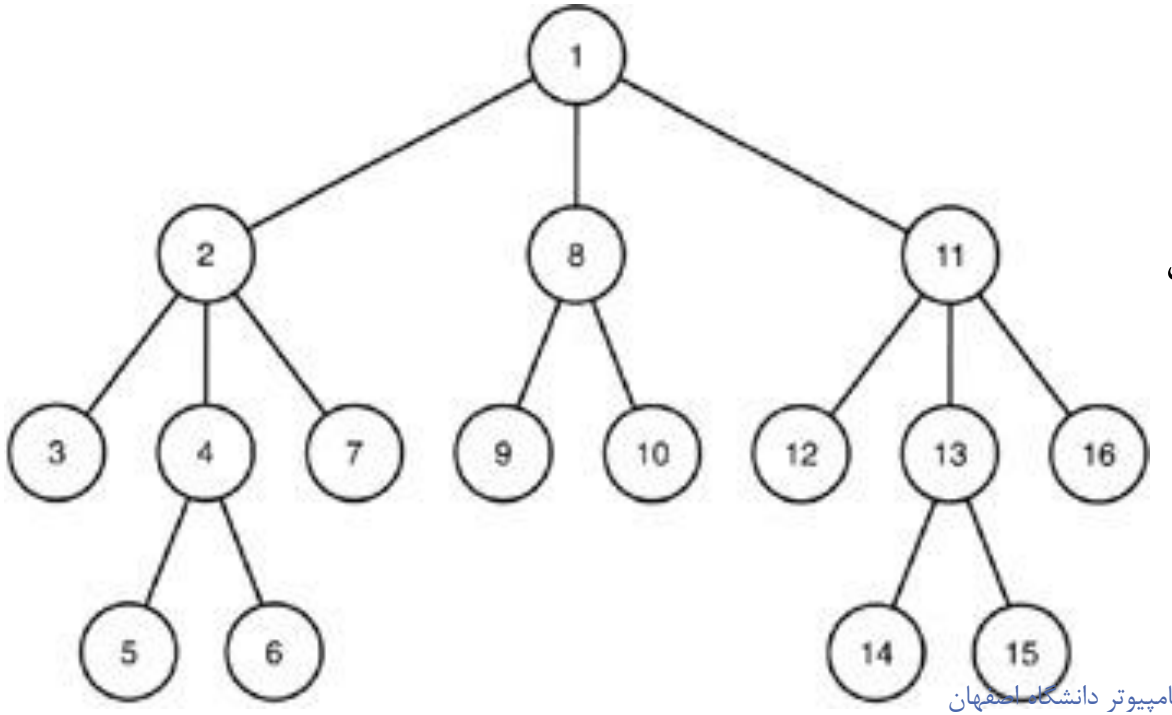


الگوریتم عقبگرد از جستجوی عمقی در یک درخت استفاده می کند.

به همین دلیل ابتدا جستجوی عمقی را مرور می کنیم.

جستجوی عمقی در یک درخت

- ابتدا ریشه ملاقات می شود.
- یک مسیر در درخت را با عمق هر چه بیشتر دنبال می کنیم تا به بن بست برسیم. در بن بست عقبگرد می کنیم تا به گره ای برسیم که فرزندی از آن ملاقات نشده باشد.
- سپس دوباره به عمق هرچه بیشتر پیشرفت می کنیم.
- قرارداد می کنیم که فرزندان یک گره را به ترتیب از چپ به راست ملاقات کنیم.



الگوریتم جستجوی عمقی در یک درخت

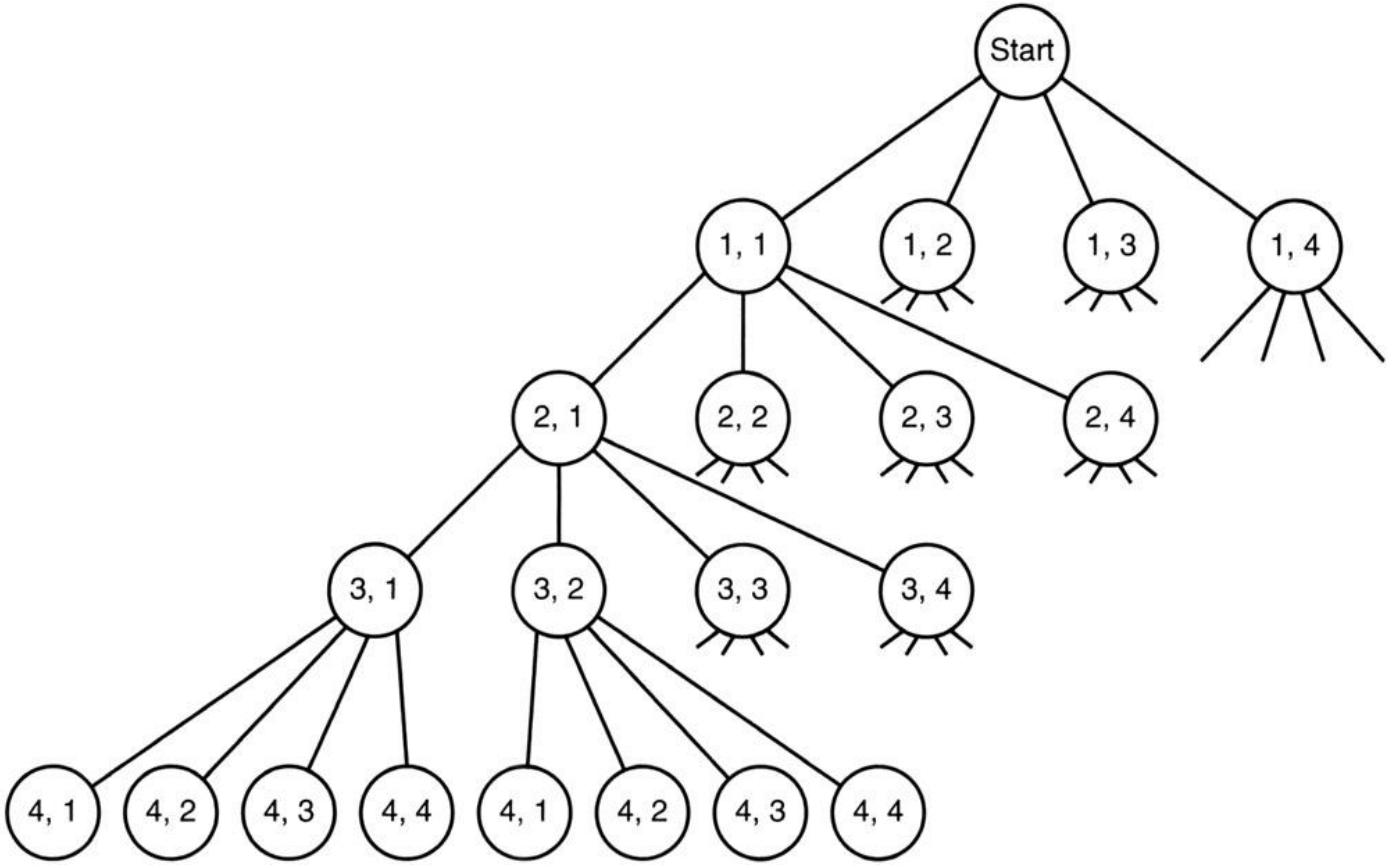
```
void depth_first_tree_search (node v)

   {
   node u;
   visit v;
   for (each child u of v)
     depth_first-tree_search (u);
}
```

مساله ۴ وزیر

- مساله ۴ وزیر و یک صفحه شطرنج ۴*۴ را در نظر می گیریم.
- مساله چیدن ۴ مهره وزیر در صفحه شطرنج ۴*۴ به نحوی که هیچ دو وزیری همدیگر را گارد ندهند.
- هر وزیر را در یک سطر قرار می دهیم.
- تعداد جواب های کاندید: ۴*۴*۴*۴

- درختی می سازیم که هر سطح آن ستون قرارگیری یکی از وزیرها را تعیین می کند.
- هر مسیر از ریشه به برگ، یک جواب کاندید برای مساله است که باید در آن کنترل شود که هیچ دو وزیری یکدیگر را گارد ندهند.
- به این درخت، فضای حالت (state space) می گویند.

بخشی از فضای حالت برای مساله ۴ وزیر



- هر زوج <i, j> :
- قرار گرفتن وزیر i ام در ستون j ام سطر i ام

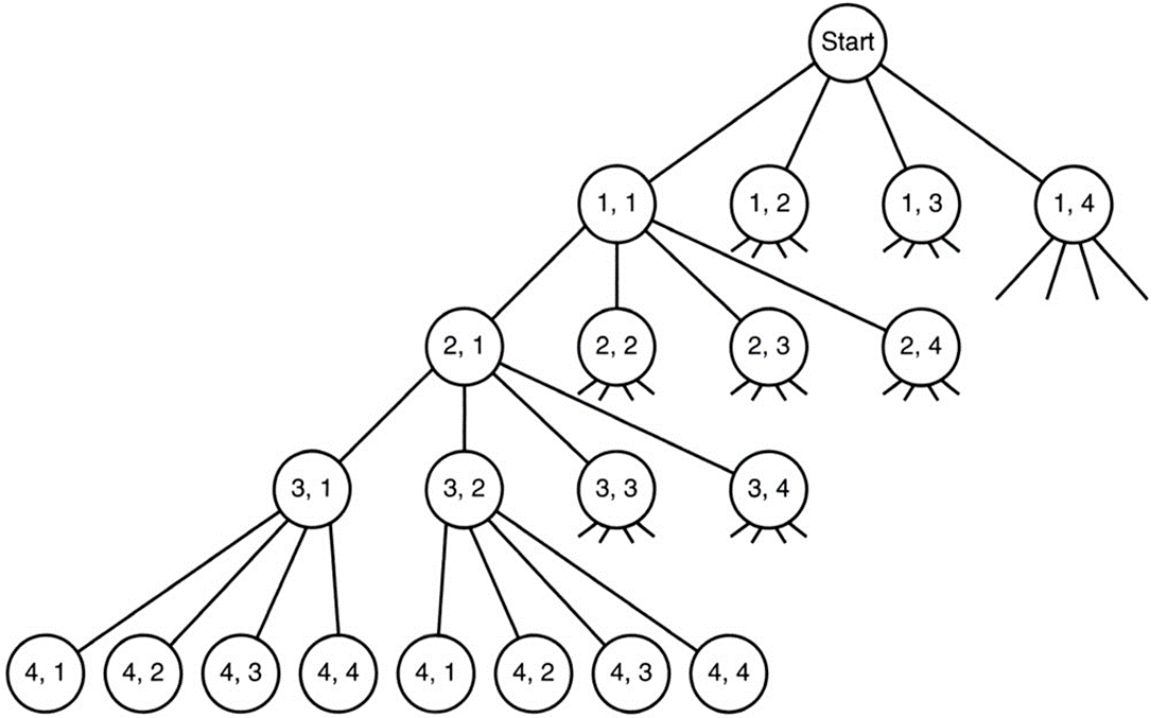
- چند مسیر در درخت فضای حالت:

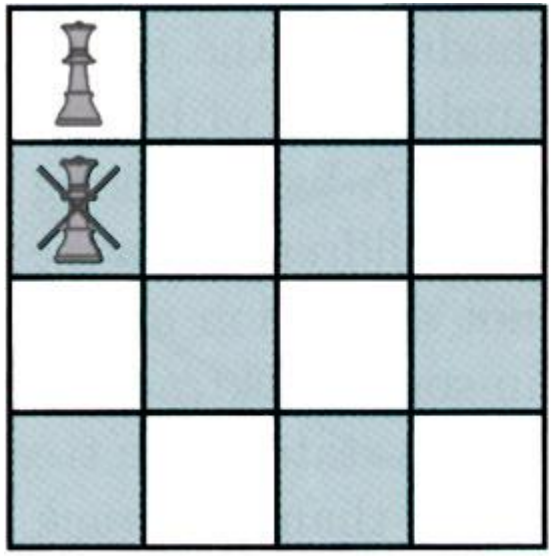
$$[<1,1>,<2,1>,<3,1>,<4,1>]$$
$$[<1,1>,<2,1>,<3,1>,<4,2>]$$
$$[<1,1>,<2,1>,<3,1>,<4,3>]$$
$$[<1,1>,<2,1>,<3,1>,<4,4>]$$
$$[<1,1>,<2,1>,<3,2>,<4,1>]$$

- اگر از جستجوی عمقی به سادگی در درخت فضای حالت استفاده کنیم، مانند حرکت در maze است که در آن از هیچ تابلو و علامتی برای مسیرهای بن بست استفاده نمی شود.

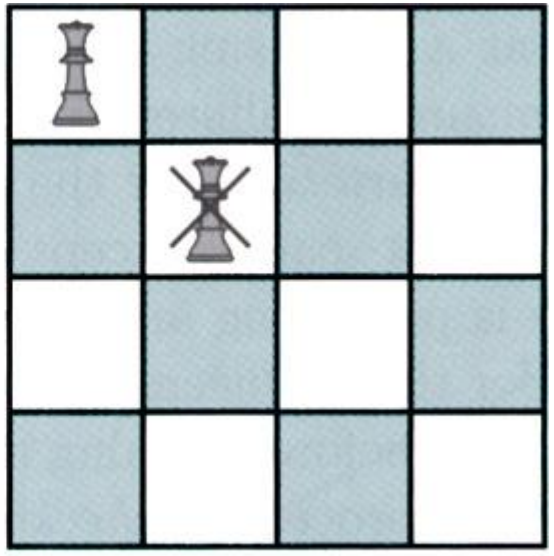
- درحالیکه می توان با تشخیص مسیرهای بن بست و علامت گذاری مسیرها، این جستجوی عمقی را کارآمدتر کرد.

- مثلا امیدی در ادامه دادن گره <2,1> و یا <2.2> نیست.





(a)



(b)

- در روش عقبگرد، وقتی تعیین می کنیم که ادامه دادن گره ای به بن بست منتهی می شود و امیدبخش نیست، به گره والد آن باز می گردیم و جستجو را از فرزند دیگر گره ادامه می دهیم.

- گره غیرامیدبخش (non-promising): گره ای که هنگام ملاقات آن مشخص شود که منجر به جواب نمی شود. در غیراینصورت، گره را امیدبخش (promising) می گوییم.

- روش عقبگرد یعنی: جستجوی عمقی در درخت فضای حالت، تعیین امیدبخش بودن یا نبودن گره و در صورت امیدبخش نبودن گره، بازگشت به گره والد.

- این کار، درخت فضای حالت را هرس می کند.

یک الگوریتم کلی برای روش عقبگرد

- عملکرد و پیاده سازی تابع promising در هر کاربردی متفاوت است.

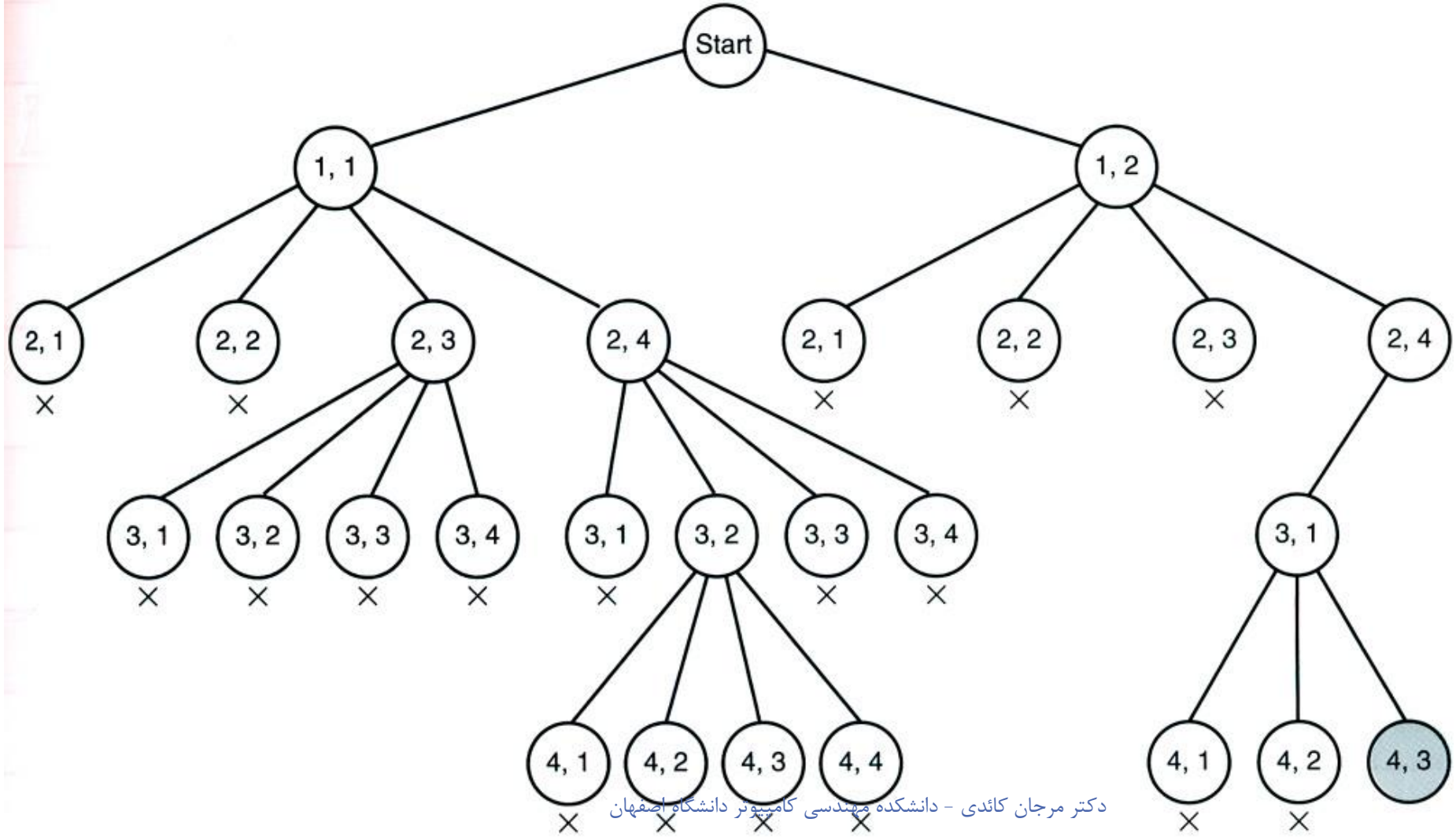
- الگوریتم عقبگرد مانند جستجوی عمقی است با این تفاوت که فرزندان یک گره درصورتی ملاقات می شوند که گره امیدبخش باشد و در آن گره، راه حلی وجود نداشته باشد.

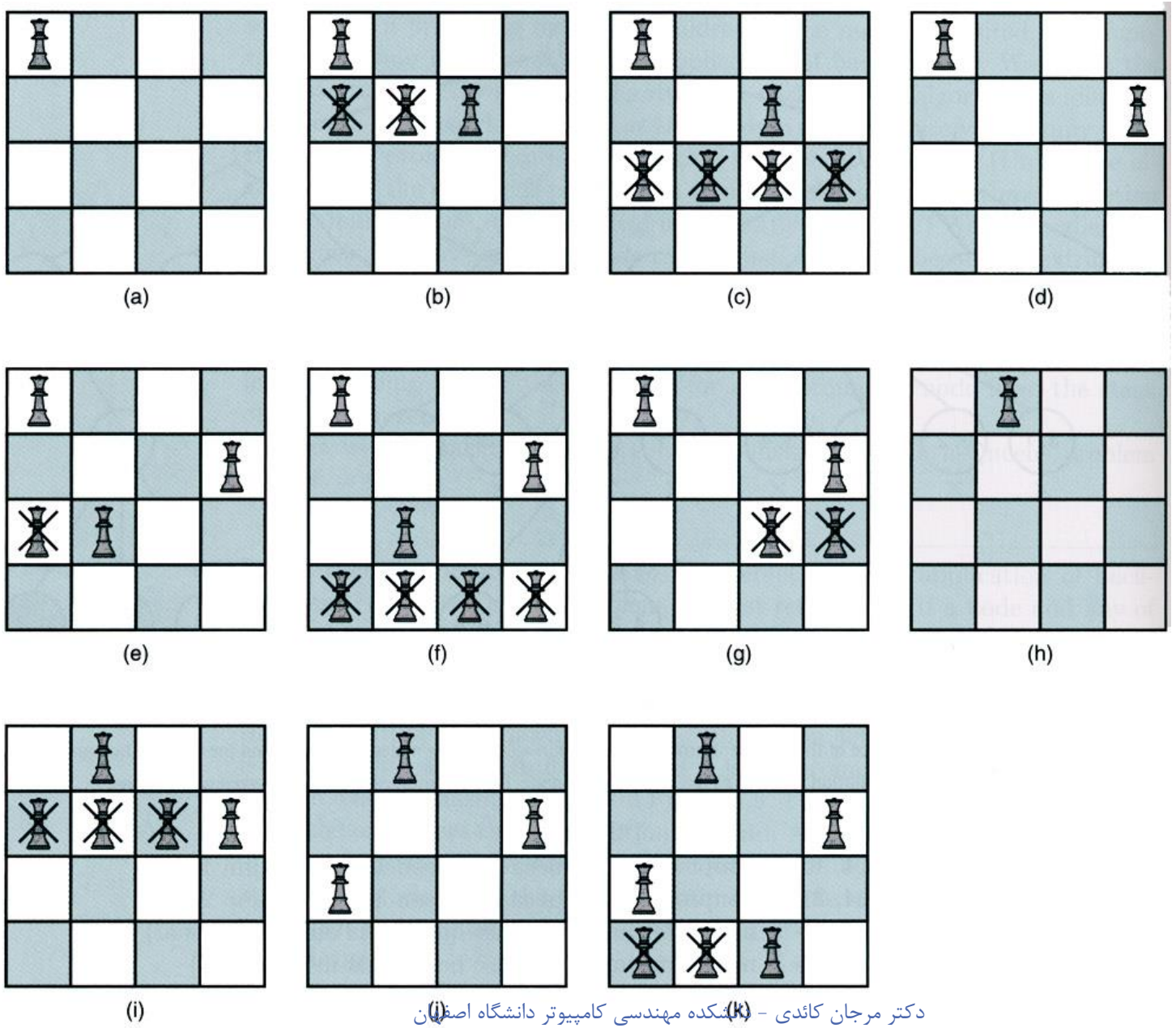
```
void checknode (node v)
  {
 node u;

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

جستجوی عمقی:

```
void depth_first_tree_search (node v)
  {
  node u;
  visit v;
  for (each child u of v)
    depth_first-tree_search (u);
}
```

درخت هرس شده برای مساله ۴ وزیر




(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)

بررسی امیدبخش بودن گره ها (پیمایش درخت به روش عقبگرد)

(a) <1, 1> is promising. {because queen 1 is the first queen positioned}

(b) <2, 1> is nonpromising. {because queen 1 is in column 1}
<2, 2> is nonpromising. {because queen 1 is on left diagonal}
<2, 3> is promising.

(c) <3, 1> is nonpromising. {because queen 1 is in column 1}
<3, 2> is nonpromising. {because queen 2 is on right diagonal}
<3, 3> is nonpromising. {because queen 2 is in column 3}
<3, 4> is nonpromising. {because queen 2 is on left diagonal}

(d) Backtrack to <1, 1>.
<2, 4> is promising.

(e) <3, 1> is nonpromising. {because queen 1 is in column 1}
<3, 2> is promising. {This is the second time we've tried <3, 2>.}

(f) <4, 1> is nonpromising. {because queen 1 is in column 1}
<4, 2> is nonpromising. {because queen 3 is in column 2}
<4, 3> is nonpromising. {because queen 3 is on left diagonal}
<4, 4> is nonpromising. {because queen 2 is in column 4}

(g) Backtrack to <2, 4>.
<3, 3> is nonpromising. {because queen 2 is on right diagonal}
<3, 4> is nonpromising. {because queen 2 is in column 4}

(h) Backtrack to root.
<1, 2> is promising.

(i) <2, 1> is nonpromising. {because queen 1 is on right diagonal}
<2, 2> is nonpromising. {because queen 1 is in column 2)
<2, 3> is nonpromising. {because queen 1 is on left diagonal}
<2, 4> is promising.

(j) <3, 1> is promising. {This is the third time we've tried <3, 1>.}

(k) <4, 1> is nonpromising. {because queen 3 is in column 1}
<4, 2> is nonpromising. {because queen 1 is in column 2}
<4, 3> is promising.

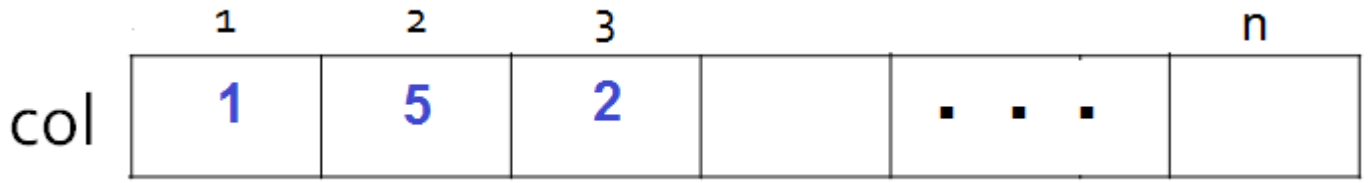
- در مساله هایی که در ادامه فصل به روش عقبگرد حل خواهند شد، تعداد گره ها در درخت فضای حالت نمایی است.

- از روش عقبگرد برای پرهیز از چک کردن بیهوده گره ها استفاده می شود.

مساله n وزیر

The n-Queens Problem

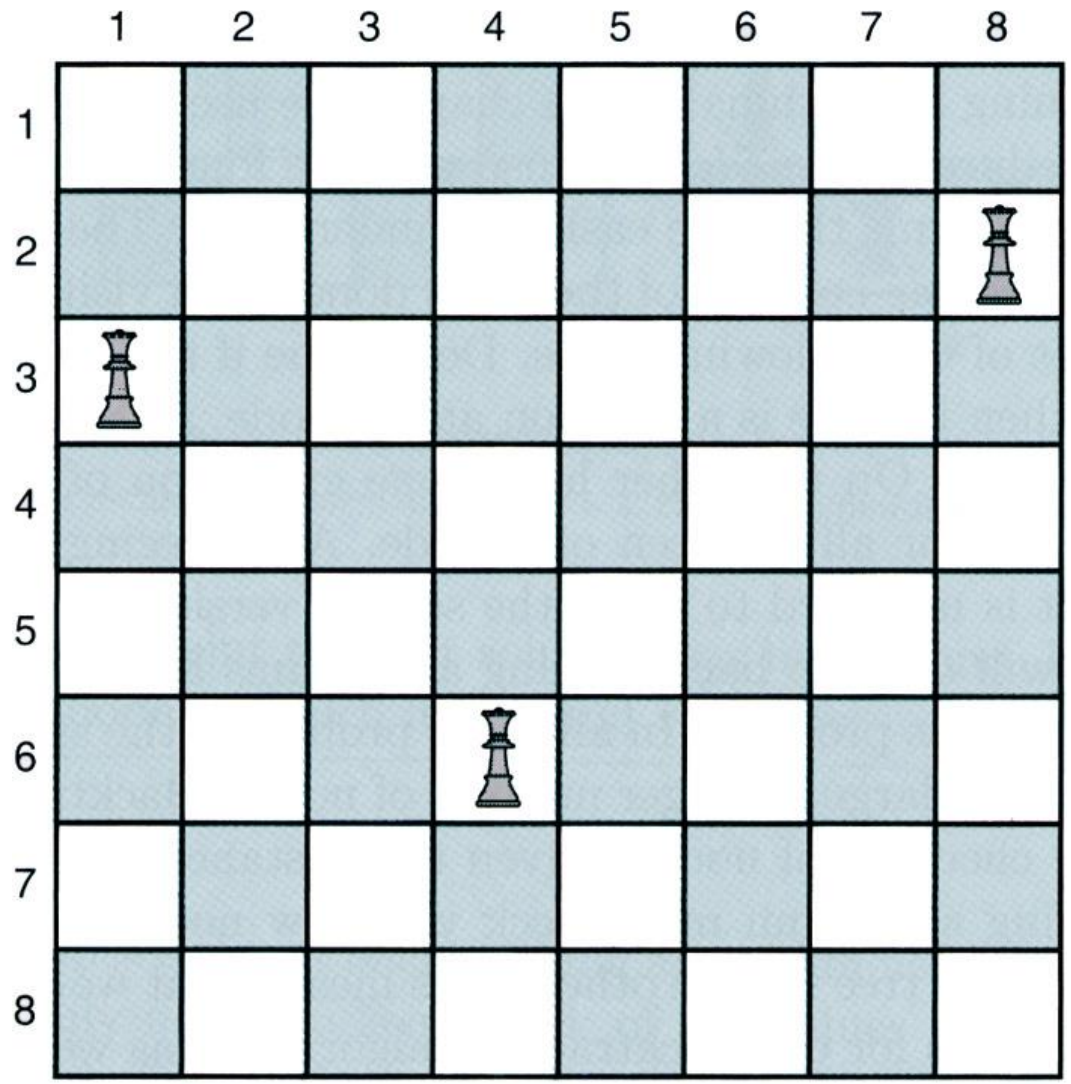
- مساله n وزیر قبلا توضیح داده شد.
- هر وزیر در یک سطر قرار داده می شود.
- استفاده از یک آرایه به نام col که در آن، ستونی که هر وزیر در آن قرار داده می شود نوشته می شود.



- تابع امیدبخش در این مساله باید بررسی کند که آیا دو وزیر در یک ستون یا یک قطر هستند یا خیر.

- بررسی اینکه دو وزیر در یک ستون هستند یا خیر:

$$col(i) = col(k).$$



- بررسی اینکه دو وزیر در یک قطر نباشند:

$$col\,(6) - col\,(3) = 4 - 1 = 3 = 6 - 3.$$

$$col\,(6) - col\,(2) = 4 - 8 = -4 = 2 - 6.$$

$$col\,(i) - col\,(k) = i - k \quad \text{or} \quad col\,(i) - col\,(k) = k - i.$$

الگوریتم عقبگرد برای مساله n وزیر

```
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++){// See if queen in (i + 1) st row can be positioned in each of the n columns
                col [i + 1] = j;
                queens (i + 1);
}
   }
```

جستجوی عمقی:

```
void depth_first_tree_search (node v)
   {
   node u;
   visit v;
   for (each child u of v)
     depth_first-tree_search (u);
}
```

تابع امیدبخش

```
bool promising (index i)
 {
index k;
bool switch;
k = 1;
switch = true;
while (k < i && switch){  // Check if any queen threatens queen in the ith row.
   if (col [i] == col [k] || abs (col [i] - col [k] )== i-k)
       switch = false;
    k++;
  }
  return switch;}
```

- الگوریتم همه جواب ها برای مساله n وزیر را پیدا می کند.
- مگر اینکه در هنگام یافتن اولین جواب دستور خروج از برنامه را بدهیم.

- فراخوانی اولیه: queens (0);

تحلیل پیچیدگی زمانی الگوریتم

- حد بالایی برای تعداد گره ها در درخت هرس شده به دست می آوریم.

$$1 + n + n^2 + n^3 + \cdots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

$$\frac{8^{8+1} - 1}{8 - 1} = 19,173,961 \text{ nodes}$$

راه دیگر برای تحلیل الگوریتم

- در نظر می گیریم که هیچ دو وزیر را نمی توان در یک ستون قرار داد.
- برای مساله ۸ وزیر داریم:

$$1+8+8\times7+8\times7\times6+8\times7\times6\times5+\cdots+8!$$
$$=109,601 \text{ promising nodes.}$$

- در حالت کلی برای n وزیر داریم:

$$1+n+n(n-1)+n(n-1)(n-2)+\cdots+n! \text{ promising nodes.}$$

- در هر صورت این روش های تحلیل، برخی از گره هایی که هرس خواهند شد را به شمارش می آورند و بنابراین تحلیل دقیقی نخواهند بود.

- یک راه دیگر برای تعیین کارایی الگوریتم، اجرای آن بر روی کامپیوتر و شمارش تعداد گره هایی است که تولید می شوند.

مساله جمع زیرمجموعه ها

The Sum-of-Subsets Problem

مساله جمع زیرمجموعه ها

- مشابه مساله کوله پشتی
- در مساله کوله پشتی، مجموعه ای از قطعات که هر کدام وزن و سودی داشت و می خواستیم زیرمجموعه ای از آنها را انتخاب کنیم و در کوله پشتی قرار دهیم به نحوی که سود حداکثر شود و جمع وزن ها از ظرفیت کوله پشتی (W) فراتر نرود.

- مساله ای که در اینجا مطرح می شود هم به مساله کوله پشتی شبیه است، با این تفاوت که در اینجا سود واحد وزن همه اشیاء با هم برابر است.

- بنابراین برای حداکثر شدن سود، باید جمع وزن اشیا حداکثر شود و از W نیز فراتر نرود.

- به عبارت ساده تر: n عدد صحیح مثبت w_i داریم و یک عدد صحیح مثبت W .
- هدف یافتن همه زیرمجموعه هایی از اعداد است که جمع آنها برابر با W است.

مثال:

W=21

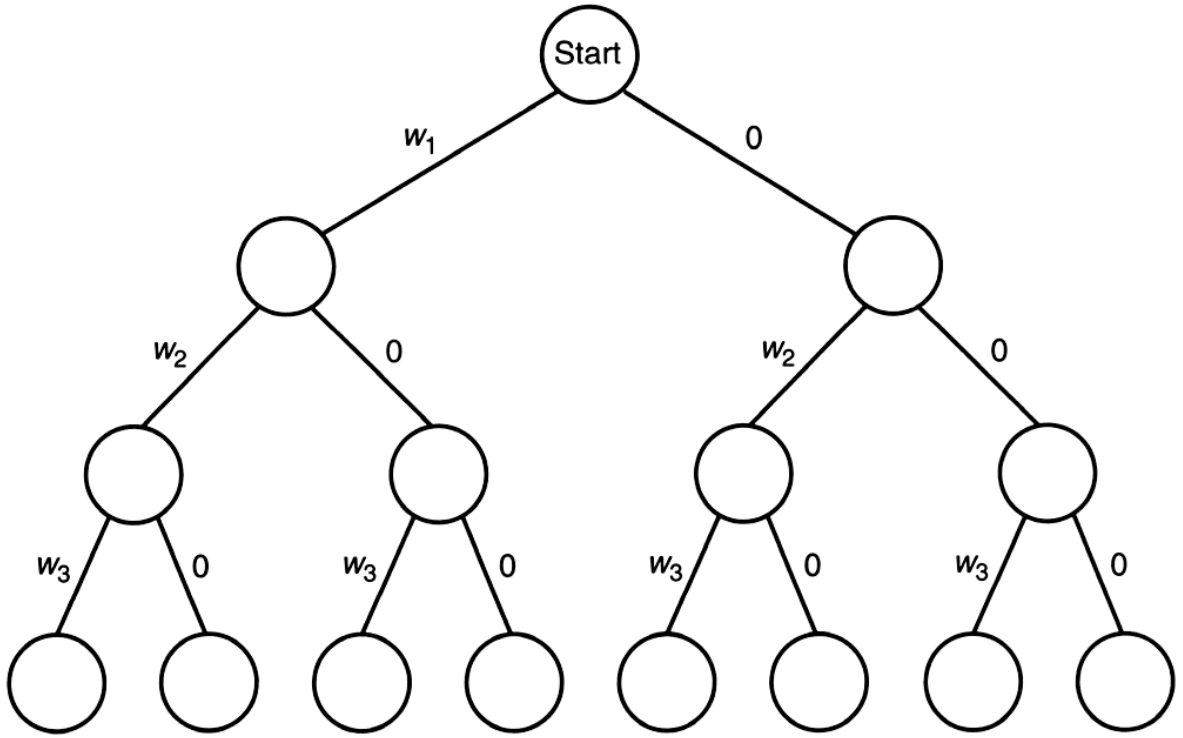
$$w_1 = 5 \qquad w_2 = 6 \qquad w_3 = 10 \qquad w_4 = 11 \qquad w_5 = 16.$$

$$w_1 + w_2 + w_3 = 5 + 6 + 10 = 21,$$
$$w_1 + w_5 = 5 + 16 = 21, \text{ and}$$
$$w_3 + w_4 = 10 + 11 = 21,$$

the solutions are $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$, and $\{w_3, w_4\}$.

درخت فضای حالت برای مساله جمع زیرمجموعه ها

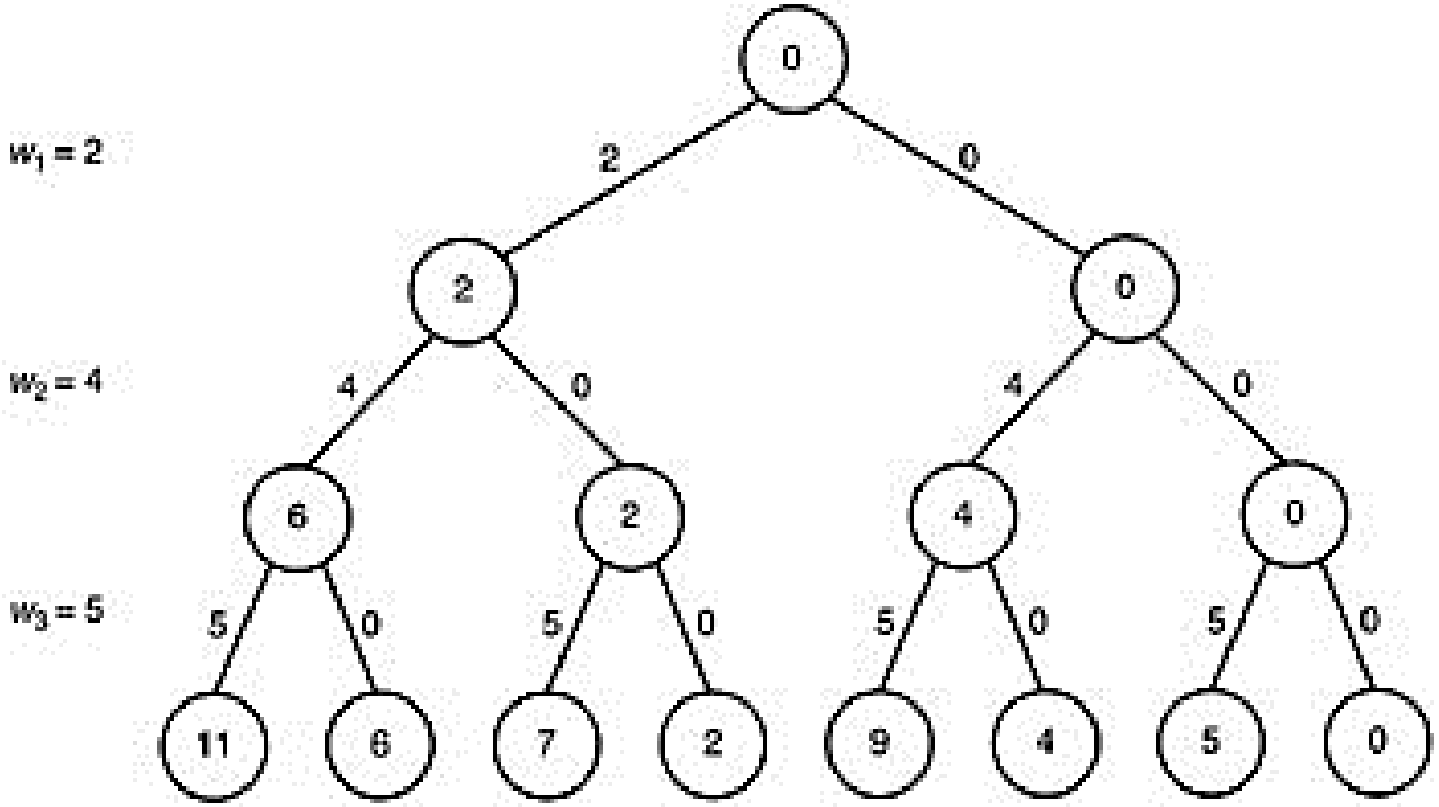
- در هر سطح درخت در مورد انتخاب شدن یا نشدن هر یک از اشیاء تصمیم می گیریم.



مثالی از درخت فضای حالت برای مساله جمع زیرمجموعه ها

$w_1 = 2 \qquad w_2 = 4 \qquad w_3 = 5.$

- W=6 و n=3



تعیین گره های غیرامیدبخش

- ابتدا اشیا را به ترتیب غیرنزولی مرتب می کنیم.
- هنگامی که در سطح i ام هستیم، w_{i+1} کمترین وزن باقیمانده است.
- اگر weight ، حاصل جمع وزن هایی باشد که تا آن لحظه انتخاب شده اند، و total جمع وزن اشیای باقیمانده باشد، آنگاه گره ای در سطح i ام در صورتی غیرامیدبخش است که:
  - **مساله هنوز حل نشده باشد و انتخاب یک شی دیگر، باعث شود که ظرفیت کوله پشتی سرریز کند (شی w_{i+1} کوچکترین شی باقیمانده است):**

$$weight + w_{i+1} > W.$$

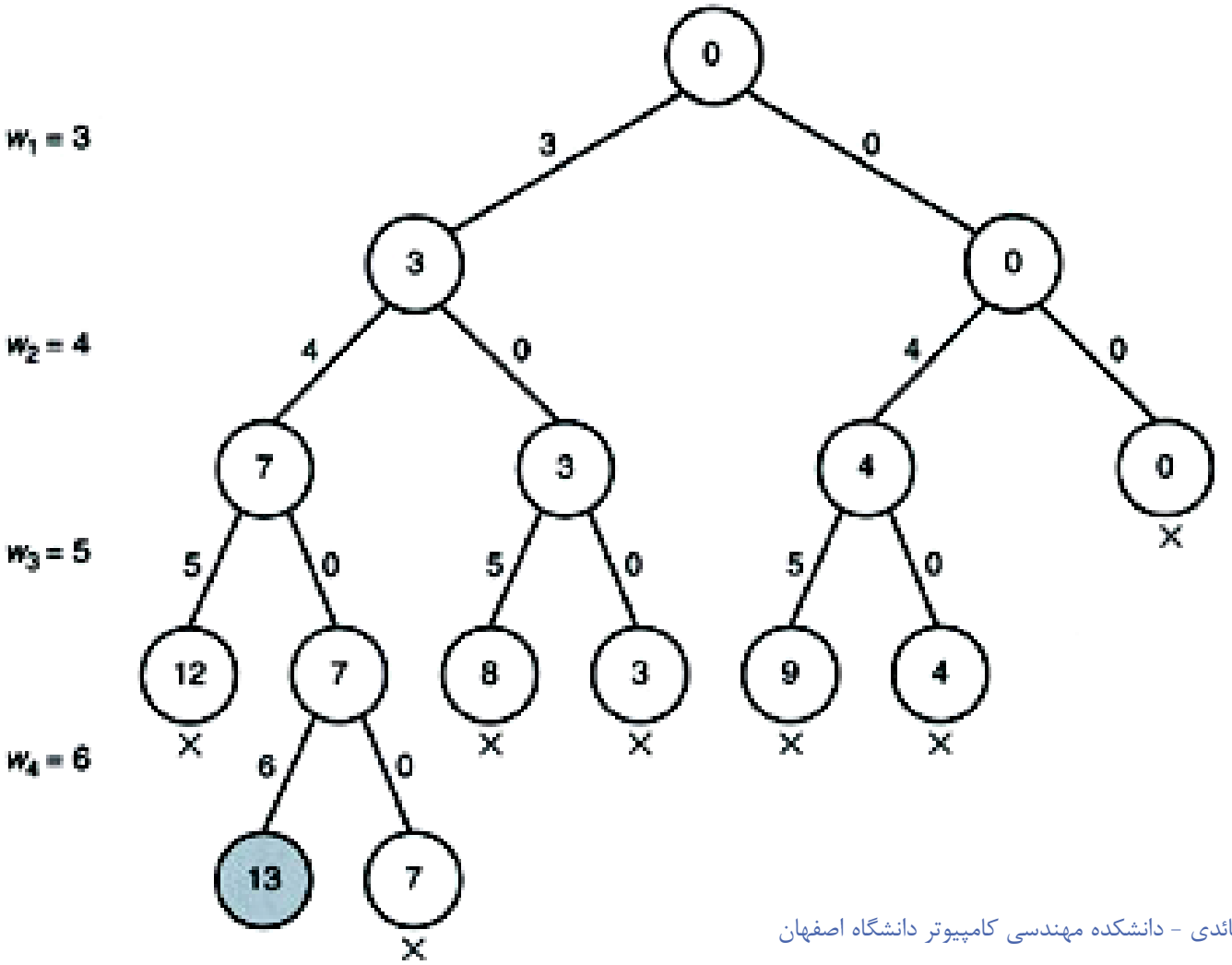
  - یا
  - **حتی استفاده از همه اشیاء باقیمانده، جمع وزن ها را به W نرساند:**

$$weight + total < W.$$

$w_1 = 3 \qquad w_2 = 4 \qquad w_3 = 5 \qquad w_4 = 6.$

مثال

- W=13 و n=4



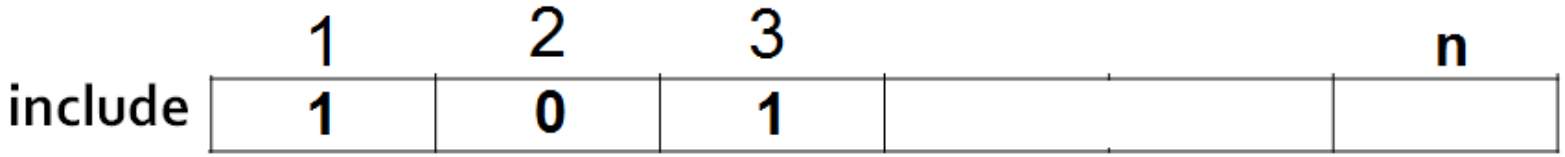
- اگر جمع وزن اشیاء انتخاب شده تا یک گره، برابر با ظرفیت کوله شود، یعنی

$$W = weight,$$

- آنگاه یک جواب به دست آمده است.

- این مساله از دسته مسائلی است که ممکن است در آن از قبل از رسیدن به آخرین انتخاب، یک جواب به دست آید.

الگوریتم عقبگرد برای مساله حاصل جمع زیرمجموعه ها




```
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] = "yes";        // Include w[i + 1].
            sum_of_subsets (i + 1, weight + w[i + 1], total - w[i + 1]);
            include [i + 1] = "no";         // Do not include w[i + 1].
            sum_of_subsets (i + 1, weight, total - w [i + 1]);
         }
  }
```

تابع امیدبخش

```
bool promising (index i);
  {
   return (weight + total >=W) && (weight == W || weight + w[i + 1] <= W);
  }
```

- اولین فراخوانی:

$$sum\_of\_subsets(0,\ 0,\ total);$$

- که در آن در ابتدا داریم:

$$total = \sum_{j=1}^{n} w[j].$$

تحلیل پیچیدگی زمانی الگوریتم

- تعداد گره هایی که در فضای حالت بررسی می شوند:

$$1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

یک نمونه شرایط که در آن درخت تا عمق زیادی پیش می رود:

$$\sum_{i=1}^{n-1} w_i < W \qquad w_n = W,$$

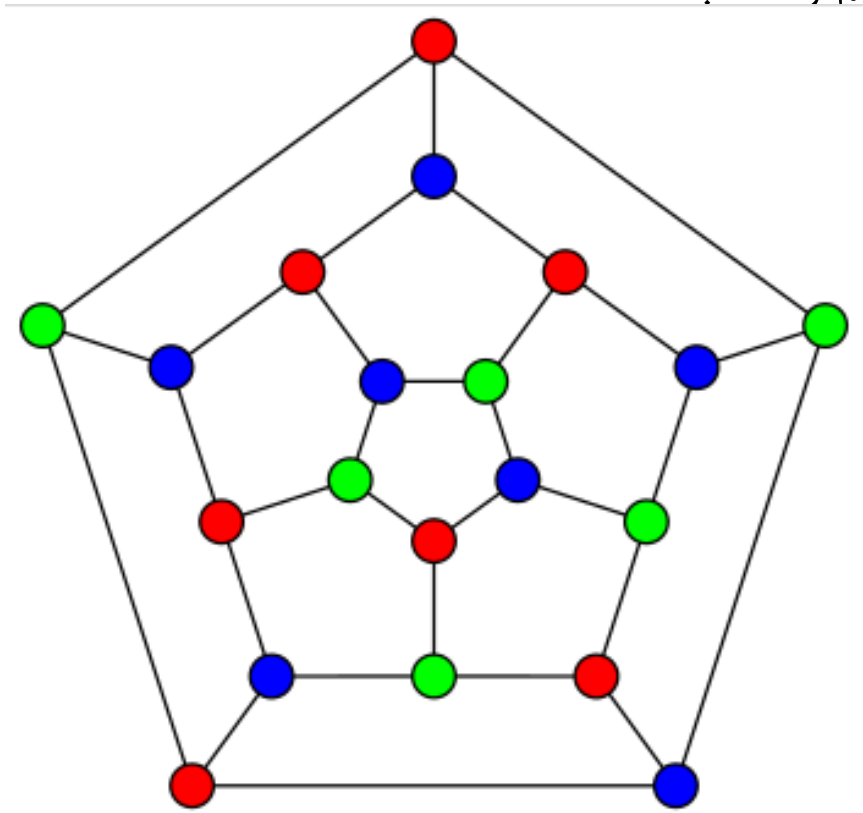
- هرچند که این بدترین حالت، از مرتبه نمایی است ولی این الگوریتم برای بسیاری از نمونه های بزرگ می تواند موثر واقع شود.

رنگ آمیزی گراف

Graph Coloring

مساله رنگ آمیزی گراف

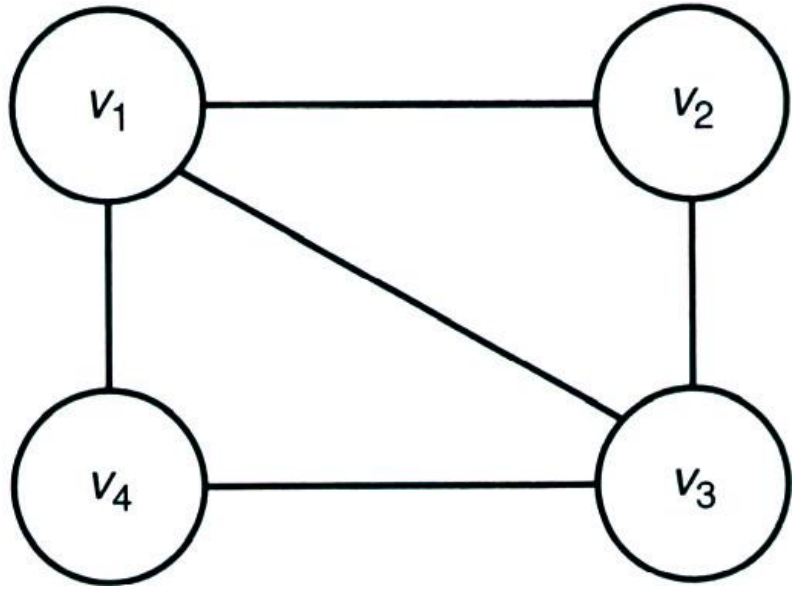
- یافتن همه راه های ممکن برای رنگ آمیزی یک گراف بدون جهت، با استفاده از حداکثر m رنگ متفاوت به نحوی که هیچ دو راس مجاوری هم رنگ نباشند.



مثال:

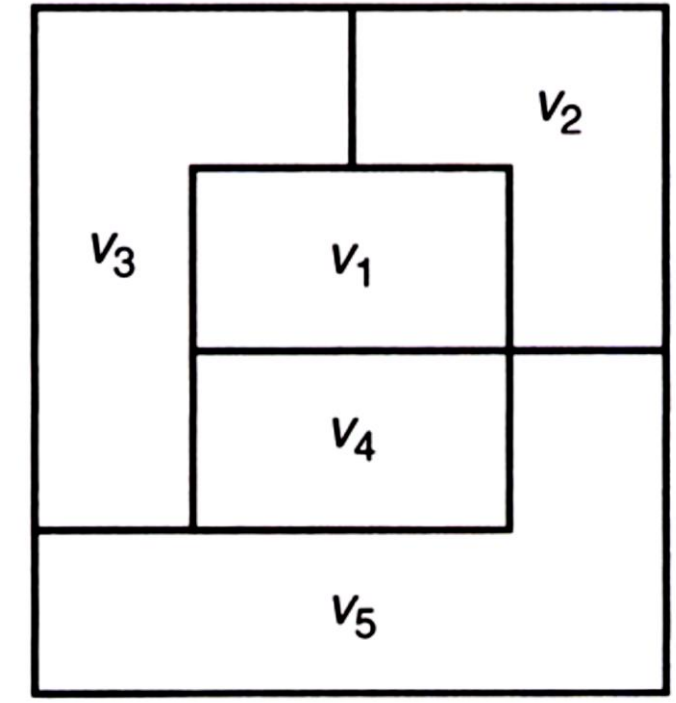
- برای m=2 رنگ آمیزی ممکنی برای این گراف وجود ندارد.
- برای m=3 داریم:

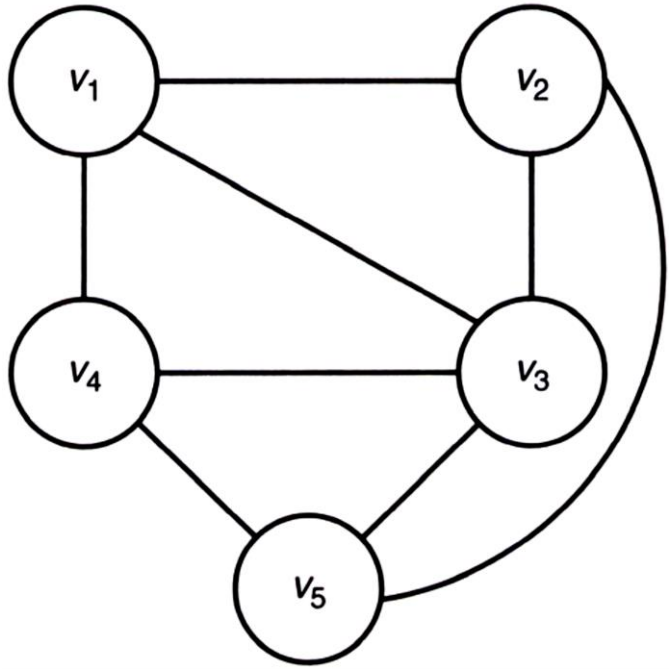
Vertex	Color
v_1	color 1
v_2	color 2
v_3	color 3
v_4	color 2

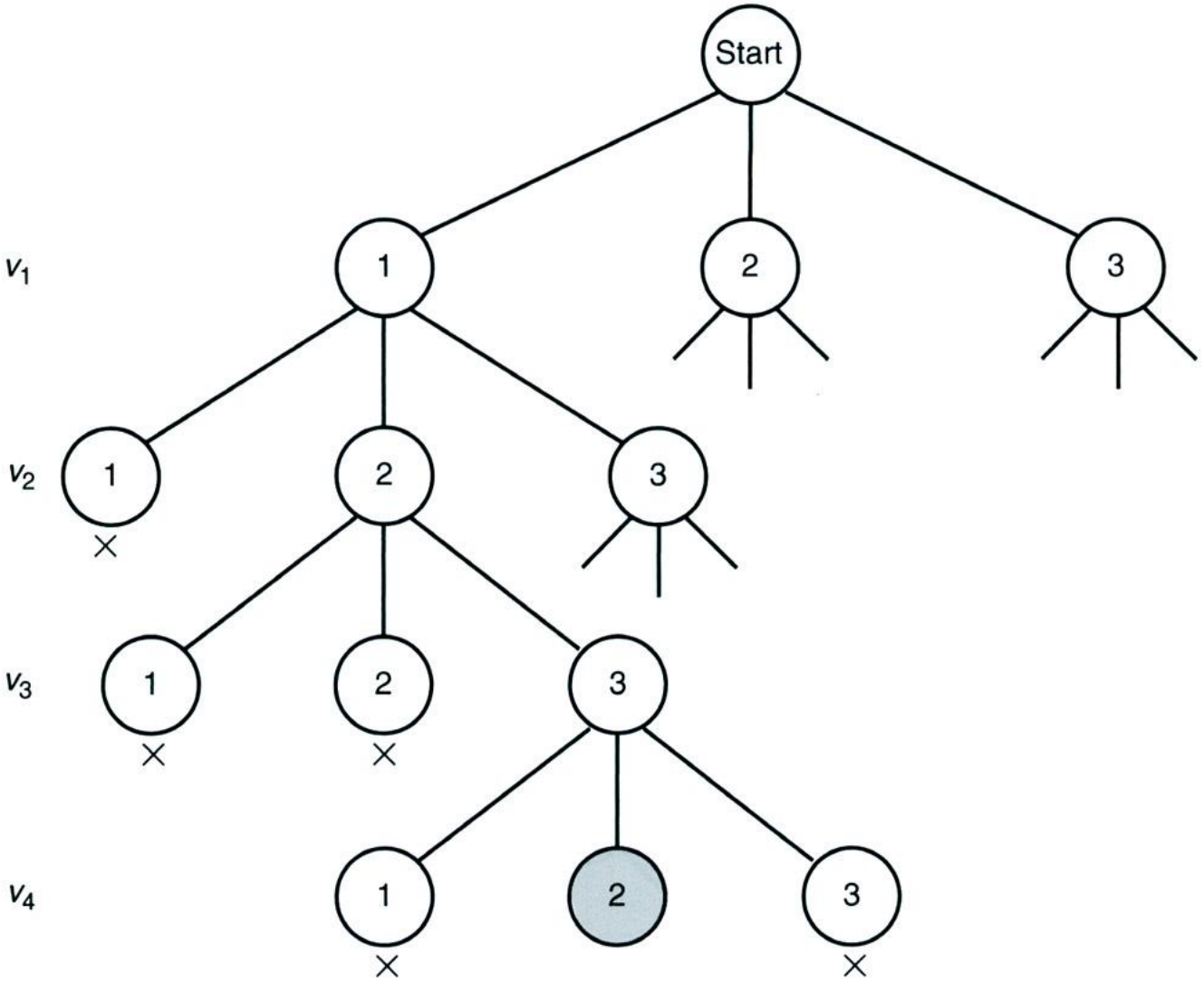


یک کاربرد: رنگ آمیزی نقشه ها

- برای هر نقشه یک گراف مسطح وجود دارد یعنی گرافی که بتوان آن را روی صفحه رسم کرد بدون اینکه یال ها یکدیگر را قطع کنند.







- در درخت فضای حالت، در هر سطح درخت،

رنگ های ممکن برای یک راس گراف امتحان می شوند.

- در سطح ۱، رنگ های ممکن برای راس ۱
- در سطح ۲، رنگ های ممکن برای راس ۲
- و .....

- یک مسیر از ریشه به برگ، یک راه حل کاندید حساب می شود.

الگوریتم عقب گرد برای رنگ آمیزی گراف

```
void m_coloring (index i)
 {
 int color;
 if (promising (i))
      if (i == n)
        cout << vcolor [1] through vcolor [n];
      else
        for (color = 1; color <= m; color++){     // Try every color for next vertex.
            vcolor [i + 1] = color;
            m_coloring (i + 1);
          }
 }
```

تابع امیدبخش

```
bool promising (index i)
 {
 index j;
 bool switch;

 switch = true;
 j = 1;
 while (j < i && switch){                    // Check if an adjacent vertex is already this color.
     if (W[i][j] && vcolor[i] == vcolor[j])
         switch = false;
     j++;
    }
return switch;
 }
```

- اولین فراخوانی

```
m_coloring(0).
```

تحلیل پیچیدگی زمانی الگوریتم

$$1 + m + m^2 + \cdots + m^n = \frac{m^{n+1} - 1}{m - 1}.$$

- مانند الگوریتم های عقبگرد دیگر، این الگوریتم می تواند برای یک نمونه بزرگ خاص کارایی داشته باشد.

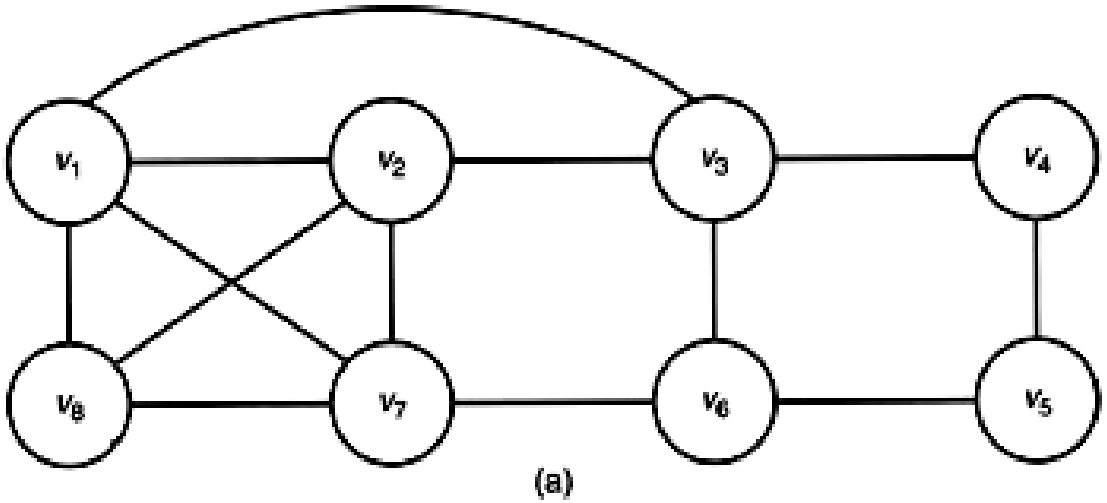
مساله مدارهای هامیلتونی

The Hamiltonian Circuits Problem

مساله یافتن دور هاملیتونی

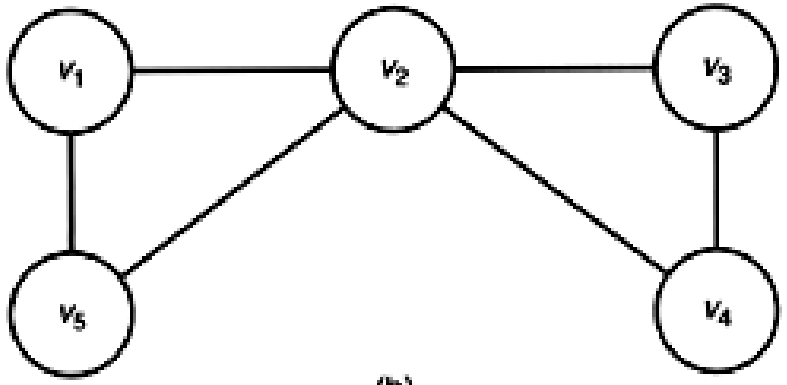
- مساله یافتن تورهای ممکن در یک گراف متصل بدون جهت

- در اینجا مساله یافتن دورهای هامیلتونی را در نظر می گیریم که در آن گراف بدون وزن در نظر گرفته می شود.



(a)

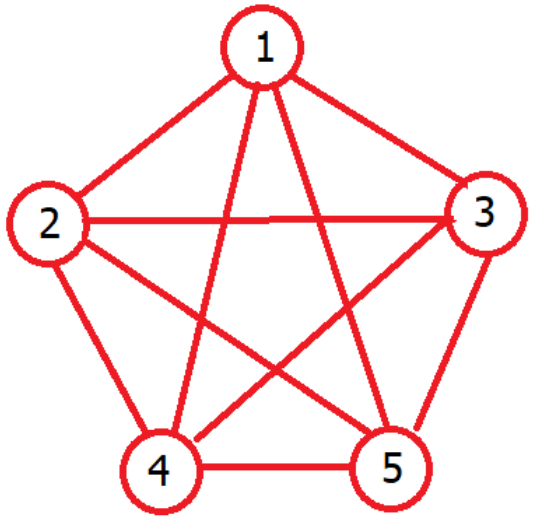
- یک دور هاملیتونی برای گراف a
- $[v_1, v_2, v_8, v_7, v_6, v_5, v_4, v_3, v_1]$

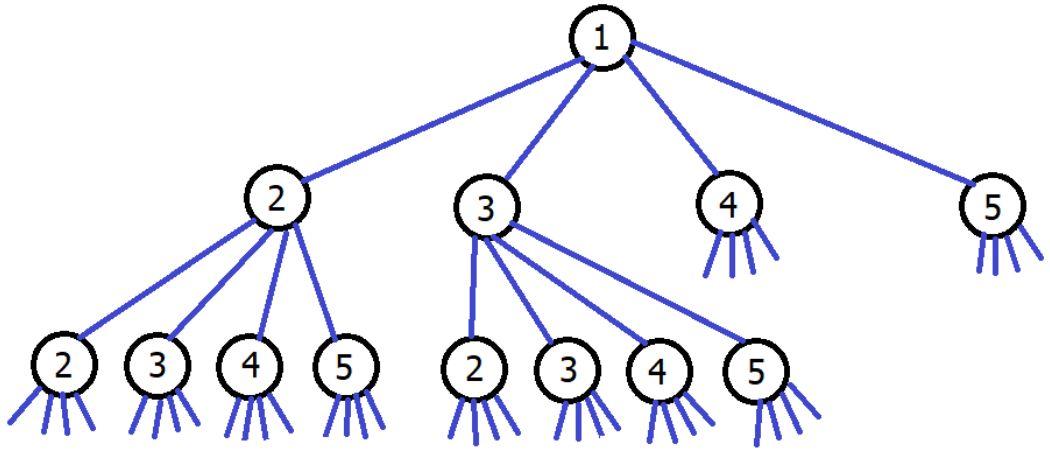


(b)

- گراف b دور هاملیتونی ندارد

درخت فضای حالت برای مساله دور هاملیتونی





- راس شروع را در سطح صفر درخت قرار می دهیم.
- در سطح ۱، همه راس ها به غیر از راس شروع را به عنوان نخستین راس در نظر می گیریم
- در سطح ۲، همه راس ها را به عنوان دومین راس موجود در دور در نظر می گیریم.
- و ....
- و در سطح n-1 هر یک از این راس ها را به عنوان n-1 امین راس موجود در دور در نظر می گیریم.

شرایط امیدبخش بودن یک گره درخت

- یک گره در صورتی امیدبخش است که:
  - راس i ام مسیر باید مجاور (همسایه) راس i-1 ام مسیر باشد.
  - راس n-1 ام باید مجاور راس صفرم مسیر (یعنی راس شروع) باشد.
  - راس i ام نمی تواند یکی از i-1 راس قبلی باشد (راس تکراری نباید در دور موجود باشد).

الگوریتم عقبگرد برای یافتن دور هامیلتونی

```
void  hamiltonian (index i)
 {
 index j;

 if (promising (i))
      if (i == n - 1)
          cout << vindex [0] through vindex [n - 1];
      else
         for (j = 2; j <=n; j++){          // Try all vertices as  next one.
             vindex [i + 1] = j;
             hamiltonian (i + 1);
         }
 }
```

تابع امیدبخش

```
bool promising (index i)
 {
 index j;
 bool switch;

if (i == n - 1 && ! W[vindex[n - 1]] [vindex [o]])
      switch = false;                          // First vertex must be adjacent to last
 else if (i > o && ! W[vindex[i - 1]] [vindex [i]])
     switch = false;                           // ith vertex must be adjacent to (i - 1) st.

 else{
  switch = true;
   j = 1;
   while (j < i && switch){                    // Check if vertex is
      if (vindex[i] == vindex [j])             // already selected.
           switch = false;
      j++;
     }
  }
 return switch;
 }
```

- اولین فراخوانی:

```
vindex [o] = 1;        //Make v1 the starting vertex.
 hamiltonian (o);
```

تحلیل پیچیدگی زمانی الگوریتم

- تعداد گره های درخت فضای حالت:

$$1 + (n-1) + (n-1)^2 + \cdots + (n-1)^{n-1} = \frac{(n-1)^n - 1}{n-2},$$

- این پیچیدگی بسیار بدتر از نمایی است.
- گرچه کل درخت حالت چک نمی شود، ولی تعداد گره هایی که چک می شود بدتر از نمایی است.
- این احتمال وجود دارد که الگوریتم عقبگرد (برای مساله دور هامیلتونی)، حتی زمان بیشتری نسبت به الگوریتم برنامه ریزی پویا (برای مساله فروشنده دوره گرد) برای گرافی با ۴۰ راس داشته باشد.

مساله کوله پشتی صفر و یک

تعریف مساله و فضای حالت آن

- درخت فضای حالت برای این مساله شبیه به مساله جمع زیرمجموعه ها است.
- در هر سطح درخت در مورد یک شی تصمیم گیری می شود و شاخه های چپ و راست، انتخاب شدن یا انتخاب نشدن هر شی را نشان می دهند.

- این مساله با مساله های دیگری که در این فصل با روش عقبگرد حل شدند، یک تفاوت عمده دارد.
- زیرا مساله کوله پشتی صفر و ۱ یک مساله بهینه سازی است.

- با بررسی یک گره به تنهایی نمی توان فهمید که آیا این گره امیدبخش است یا خیر (ادامه دادن این گره موجب رسیدن به بهترین جواب می شود یا خیر). زیرا این موضوع نسبی است و در مقایسه با بقیه گره ها مشخص می شود.

شکل کلی یک الگوریتم عقبگرد برای مسائل بهینه سازی

- best: بهترین جوابی که تا کنون یافته شده است.
- value(v) : ارزش گره v

```
void checknode (node v)
    {
    node u;

    if (value(v) is better than best)
        best = value(v);
    if (promising(v))
        for (each child u of v)
            checknode(u);
    }
```

شرط های امیدبخش نبودن در مساله کوله پشتی: **شرط اول**

- Weight: مجموع وزن اشیائی که تا کنون در کوله پشتی هستند.
- W : ظرفیت کوله پشتی

$$weight \geq W.$$

شرط های امیدبخش نبودن در مساله کوله پشتی: **شرط دوم**

- فرض می کنیم در مورد شی اول تا iام تصمیم گرفته ایم. اکنون ادامه مساله را به عنوان مساله کوله پشتی پیوسته به روش حریصانه حل می کنیم تا برای جوابی که در ادامه این شاخه به دست می آید یک تخمین دست بالا به دست آوریم.

- برای این کار اشیاء باقیمانده را به ترتیب نسبت ارزش به وزنشان مرتب می کنیم و به ترتیب در کوله پشتی قرار می دهیم.

شرط های امیدبخش نبودن در مساله کوله پشتی: شرط دوم (ادامه)

- فرض می کنیم تا شی k-1 ام در فضای باقی مانده کوله پشتی جا می شود و شی k ام به طور کامل در کوله پشتی جا نمی گیرد.

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

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for } k\text{th item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}.$$

شرط های امیدبخش نبودن در مساله کوله پشتی: **شرط دوم (ادامه)**

- اگر maxprofit بهترین جوابی باشد که تا کنون به دست آمده است (یعنی همان best)، آنگاه یک گره در صورتی غیرامیدبخش است که:

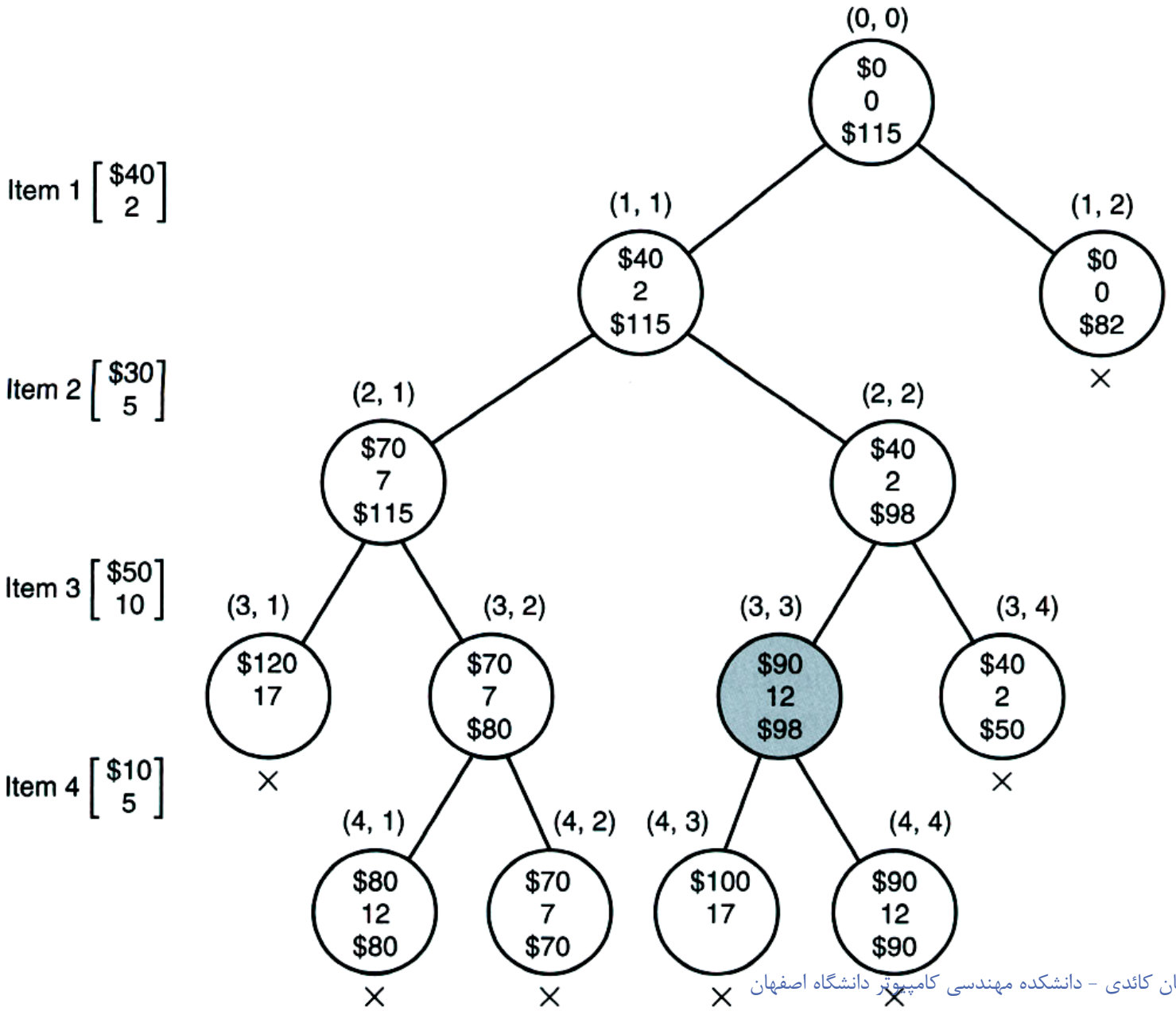
$$bound \leq maxprofit.$$

مثال:

- یک مساله کوله پشتی با ۴ شی:

n = 4, W = 16

i	p_i	w_i	$\frac{p_i}{w_i}$
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2



- در ابتدا: maxprofit = o
- عددها در هر گره از بالا به پایین:
- Profit
- totweight
- bound

الگوریتم عقبگرد برای کوله پشتی صفر و ۱

```
void knapsack (index i, int profit, int weight)
  {
 if (weight <= W && profit > maxprofit){     // This set is best so far.

      maxprofit = profit;
      numbest = i;                  // Set numbest to number of items considered
      bestset = include;            // Set bestset to this solution.
  }

if (promising(i)){
      include [i + 1] = "yes";                                  // Include w[i + 1].
      knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
      include [i + 1] = "no";                                   // Do not include w[i + 1].
      knapsack (i + 1, profit, weight);   }
  }
```

تابع امیدبخش

```
bool promising (index i)
 {index j, k;
 int totweight;
float bound;

if (weight >= W)
      return false;
else {
      j = i + 1;
      bound = profit;
      totweight = weight;
      while (j <= n && totweight + w[j] < = W){
            totweight = totweight + w[j];
            bound = bound + p[j];
            j++;
           }
     k = j;
     if (k <=n)
     bound = bound + (W - totweight) * p[k]/w[k];
     return bound > maxprofit;
    }
}
```

- وقتی i=n می شود، قطعا گره ها توسعه نمی یابند زیرا در این حالت bound<=maxprofit می شود و این گره ها غیرامیدبخش به نظر می رسند.

فراخوانی برنامه knapsack در یک تابع main

```
numbest = 0;
maxprofit = 0;
knapsack(0, 0, 0);
cout << maxprofit;                    // Write the maximum profit.
for (j = 1; j <= numbest; j++)            // Show an optimal set of items.
    cout << bestset[i];
```

پیچیدگی

$\Theta(2^n)$

پایان فصل پنجم