بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

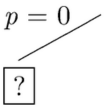
حسین فلسفین

unit propagation

The real value of simplification is that it enables a further optimization that can drastically decrease the search space.

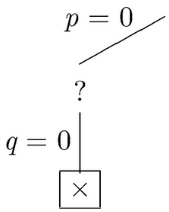
In the course of the backtracking search, if we see a sentence that consists of single atom, say p, we know that the only possible satisfying assignments further down the branch must set p to true. In this case, we can fix p to be true and ignore the subbranch that sets p to false. Similarly, when we encounter a sentence that consists of a single negated atom, say $\neg p$, we can fix p to be false and ignore the other subbranch. This optimization is called unit propagation because sentences of the form p or $\neg p$ are called units.

To start, let p be false:



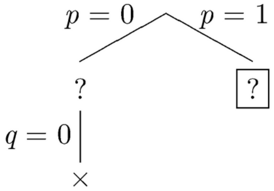
Original	Simplified
$p \vee q$	q
$p \vee \neg q$	$\neg q$
$\neg p \vee q$	—
$\neg p \vee \neg q \vee \neg r$	—
$\neg p \vee r$	—

In the simplified set of sentences, we have the unit $\neg q$, so we fix q to be false (unit propagation). (We also have the unit q, so we could have fixed q to true. The result is the same in either case.)



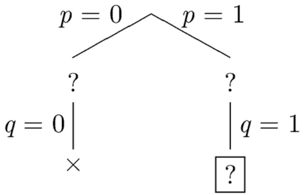
Original	Simplified
$p \vee q$	*false*
$p \vee \neg q$	–
$\neg p \vee q$	–
$\neg p \vee \neg q \vee \neg r$	–
$\neg p \vee r$	–

Δ is falsified, so we backtrack to the most recent decision point, all the way back at the root. Let p be true.



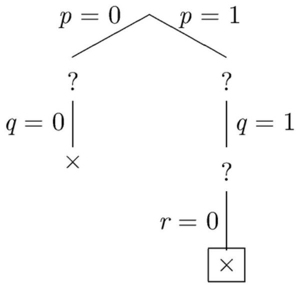
Original	**Simplified**
$p \vee q$	$-$
$p \vee \neg q$	$-$
$\neg p \vee q$	q
$\neg p \vee \neg q \vee \neg r$	$\neg q \vee \neg r$
$\neg p \vee r$	r

In the simplified set of sentences, we have the unit q so we do unit propagation, fixing q to be true. (We could also have performed unit propagation using the other unit r.)



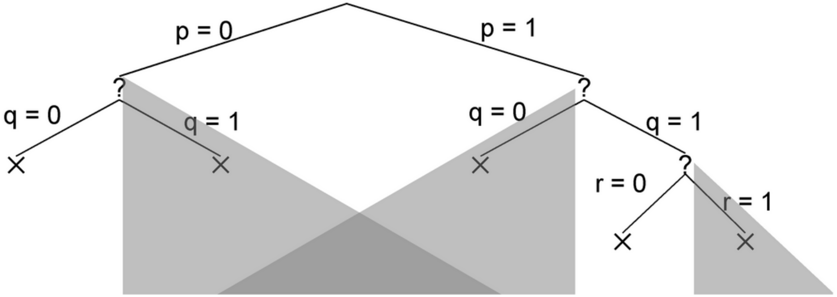
Original	**Simplified**
$p \vee q$	$-$
$p \vee \neg q$	$-$
$\neg p \vee q$	$-$
$\neg p \vee \neg q \vee \neg r$	$\neg r$
$\neg p \vee r$	r

In the simplified set of sentences, we have the unit $\neg r$ so we do unit propagation, fixing r to be false.



Original	**Simplified**
$p \vee q$	–
$p \vee \neg q$	–
$\neg p \vee q$	–
$\neg p \vee \neg q \vee \neg r$	–
$\neg p \vee r$	*false*

All branches are closed, so the method determines that $\triangle$ is unsatisfiable. Compared to the tree explored by the basic backtracking search, we see that the greyed out subtrees are pruned away from the search space.



Pure symbol heuristic

A pure symbol is a symbol that always appears with the same “sign” in all clauses. For example, in the three clauses

$$(A \vee \neg B) \wedge (\neg B \vee \neg C) \wedge (C \vee A)$$

the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure. It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals true, because doing so can never make a clause false. Note that, in determining the purity of a symbol, the algorithm can ignore clauses that are already known to be true in the model constructed so far. For example, if the model contains $B = 0$, then the clause $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.

DPLL

The Davis-Putnam-Logemann-Loveland method (DPLL) is a classic method for SAT solving. It is essentially backtracking search along with unit propagation and pure literal elimination. Most modern, complete SAT solvers are based on DPLL, with additional optimizations not discussed here. These SAT solvers are routinely used to solve SAT problems with large numbers of propositions.

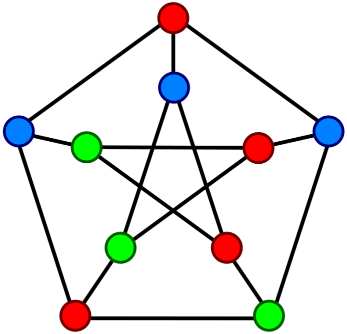
```
**proc** DPLL ( $F$ : clause set ) : **bool**
// outputs 1, if $F$ is satisfiable, 0 otherwise
**if** $\square \in F$ **then return** $0$
**if** $F = \emptyset$ **then return** $1$
**if** $F$ contains a unit clause $\{u\}$ **then return** DPLL($F\{u = 1\}$)
**if** $F$ contains a pure literal $u$ **then return** DPLL($F\{u = 1\}$)
choose with the adequate strategy a variable $x \in$ *Var*($F$)          $(*)$
**if** DPLL ($F\{x = 0\}$) **then return** $1$
**return** DPLL ($F\{x = 1\}$)
```

Graph Coloring

توجه کنید که این برای دومین بار است که مسئلهٔ رنگ آمیزی گراف مواجه می‌شویم. بار اول (در فصل راهبرد حریصانه) با صورت بهینه‌سازی آن مواجهه داشتیم: یعنی یافتن کمترین تعداد از رنگ‌ها که به‌وسیلهٔ آنها می‌توان رئوس یک گراف را رنگ کرد. اما این‌بار با صورت تصمیم‌گیری آن روبرو هستیم: آیا می‌توان رئوس یک گراف را تنها با استفاده از حداکثر m رنگ، رنگ‌آمیزی کرد یا خیر؟ m یک عدد صحیح داده‌شده و معین است. این صورت از مسئله را مسئلهٔ mــ رنگ آمیزی گراف می‌نامیم.

مسئلهٔ ۲ــ رنگ‌آمیزی گراف یک مسئلهٔ سخت نیست، و برای حل نمونه‌های این مسئله، الگوریتمی با مرتبهٔ چندجمله‌ای نسبت به تعداد رئوس گراف موجود است. اما اگر $3 \geq m$، آنگاه مسئلهٔ mــ رنگ‌آمیزی گراف در حالت عمومی ***NP-complete*** است.

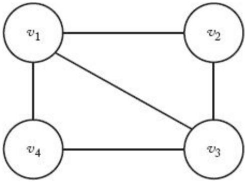
The m-Coloring problem concerns finding all ways to color an undirected graph using at most m different colors, so that no two adjacent vertices are the same color. We usually call the m-Coloring problem a unique problem for each value of m.

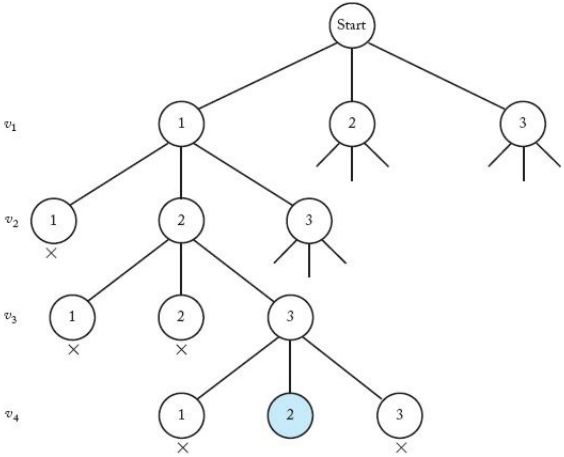


State Space Tree

A straightforward state space tree for the m-Coloring problem is one in which each possible color is tried for vertex v_1 at level 1, each possible color is tried for vertex v_2 at level 2, and so on until each possible color has been tried for vertex v_n at level n. Each path from the root to a leaf is a candidate solution. We check whether a candidate solution is a solution by determining whether any two adjacent vertices are the same color. To avoid confusion, remember in the following discussion that "node" refers to a node in the state space tree and "vertex" refers to a vertex in the graph being colored.

We can backtrack in this problem because a node is non-promising if a vertex that is adjacent to the vertex being colored at the node has already been colored the color that is being used at the node. The number in a node is the number of the color used on the vertex being colored at the node. The first solution is found at the shaded node. Nonpromising nodes are labeled with crosses.


v_1
v_2
v_3
v_4


Start
v_1
v_2
v_3
v_4
1
2
3
×

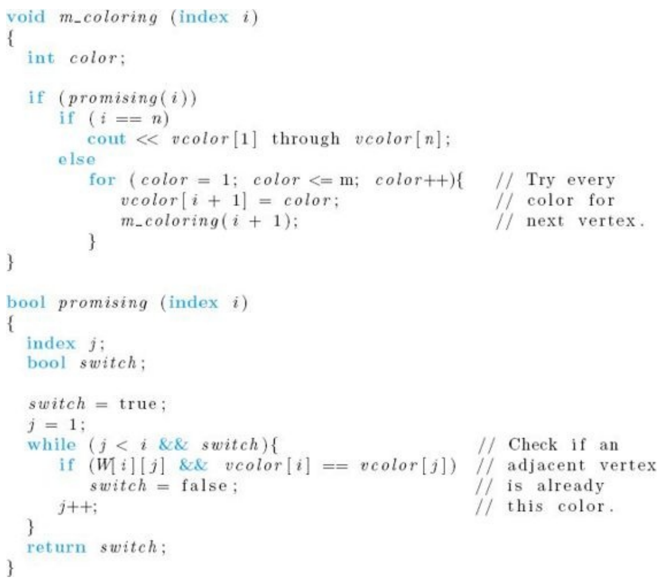
After v_1 is colored color 1, choosing color 1 for v_2 is nonpromising because v_1 is adjacent to v_2. Similarly, after v_1, v_2, and v_3 have been colored colors 1, 2, and 3, respectively, choosing color 1 for v_4 is nonpromising because v_1 is adjacent to v_4.

The Backtracking Algorithm for the m-Coloring Problem

☞ *Problem: Determine all ways in which the vertices in an undirected graph can be colored, using only m colors, so that adjacent vertices are not the same color.*

☞ *Inputs: positive integers n and m, and an undirected graph containing n vertices. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is true if there is an edge between ith vertex and the jth vertex and false otherwise.*

☞ *Outputs: all possible colorings of the graph, using at most m colors, so that no two adjacent vertices are the same color. The output for each coloring is an array $vcolor$ indexed from 1 to n, where $vcolor[i]$ is the color (an integer between 1 and m) assigned to the ith vertex.*



```
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
           vcolor[i + 1] = color;                 // color for
           m_coloring(i + 1);                     // next vertex.
        }
}

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

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

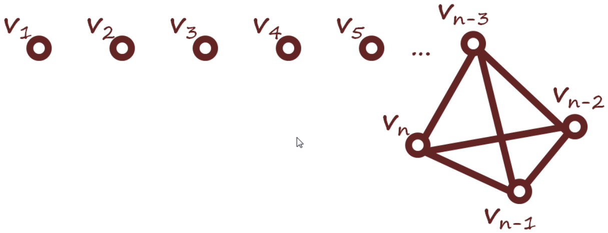
☞ *Following our usual convention, n, m, W, and $vcolor$ are not inputs to either routine. The top level call to $m\_coloring$ would be $m\_coloring(0)$.*
☞ *The number of nodes in the state space tree for this algorithm is equal to*

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

☞ *For a given m and n, it is possible to create an instance that checks at least an exponentially large number of nodes (in terms of n).*

مثال (عملکرد ناکارآمد)

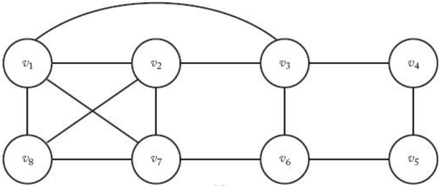
الگوریتم ما برای آنکه بفهمد گراف زیر ۳ــ رنگ پذیر نیست، درخت فضای حالتی می‌سازد که تعدادی نمایی نود دارد (نسبت به n). چرا؟



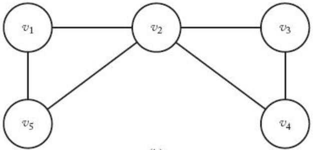
تمرین: یک گراف **همبند** با n رأس بسازید (برای هر n داده‌شدهٔ دلخواه) که الگوریتم ما برای پی‌بردن به این حقیقت که آن گراف ۳ــ رنگ پذیر نیست، مجبور به ملاقات تعدادی نمایی نود باشد.

The Hamiltonian Circuits Problem

Given a connected, undirected graph, a Hamiltonian Circuit (also called a tour) is a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex.



(a)



(b)

A state space tree for this problem is as follows. Put the starting vertex at level 0 in the tree; call it the zeroth vertex on the path. At level 1, consider each vertex other than the starting vertex as the first vertex after the starting one. At level 2, consider each of these same vertices as the second vertex, and so on. Finally, at level $n-1$, consider each of these same vertices as the $(n-1)$st vertex.

The following considerations enable us to backtrack in this state space tree:

1. *The ith vertex on the path must be adjacent to the $(i-1)$st vertex on the path.*
2. *The $(n-1)$st vertex must be adjacent to the 0th vertex (the starting one).*
3. *The ith vertex cannot be one of the first $i-1$ vertices.*

The Backtracking Algorithm for the Hamiltonian Circuits Problem

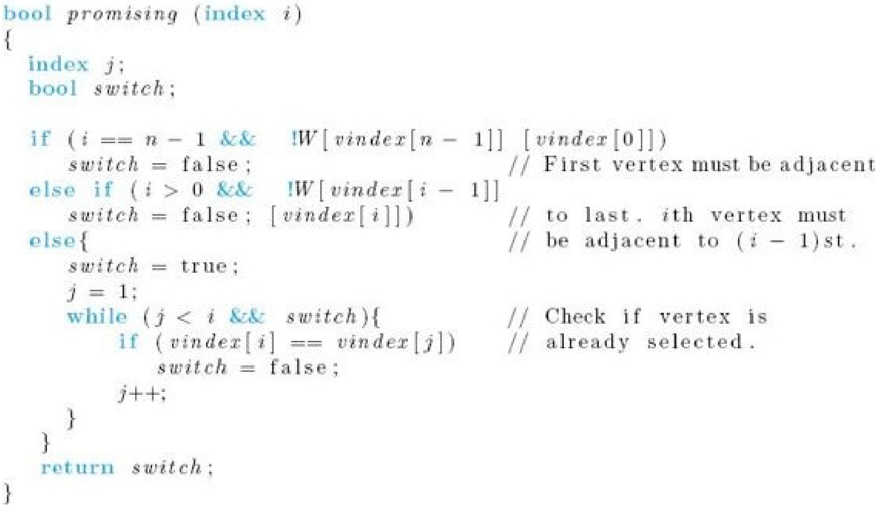
☞ *Problem:* *Determine all Hamiltonian Circuits in a connected, undirected graph.*

☞ *Inputs:* *positive integer n and an undirected graph containing n vertices. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is true if there is an edge between the ith vertex and the jth vertex and false otherwise.*

☞ *Outputs:* *For all paths that start at a given vertex, visit each vertex in the graph exactly once, and end up at the starting vertex. The output for each path is an array of indices vindex indexed from 0 to $n-1$, where $vindex[i]$ is the index of the ith vertex on the path. The index of the starting vertex is $vindex[0]$.*

```
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
           vindex[i + 1] = j;                 // next one.
           hamiltonian(i + 1);
        }
}
```



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

  if (i == n - 1 &&  !W[vindex[n - 1]] [vindex[0]])
     switch = false;                         // First vertex must be adjacent
  else if (i > 0 &&  !W[vindex[i - 1]]
     switch = false; [vindex[i]])            // to last. ith vertex must
  else{                                      // be adjacent to (i - 1)st.
     switch = true;
     j = 1;
     while (j < i && switch){                // Check if vertex is
          if (vindex[i] == vindex[j])        // already selected.
               switch = false;
          j++;
     }
   }
   return switch;
}
```

Following our convention, n, W, ***and*** $vindex$ ***are not inputs to either routine. If these variables were defined globally, the top-level called to hamiltonian would be as follows:***

$vindex[0] = 1;$ (یعنی رأس آغازین ما v_1 است.)
$hamiltonian(0);$

The number of nodes in the state space tree for this algorithm is

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

which is much worse than exponential.

مثال (عملکرد ناکارآمد)

Although the following instance does not check the entire state space tree, it does check a worse-than exponential number of nodes. Let the only edge to v_1 be one from v_2, and let all the vertices other than v_1 have edges to each other. There is no Hamiltonian Circuit for the graph, and the algorithm will check a worse-than-exponential number of nodes to learn this.