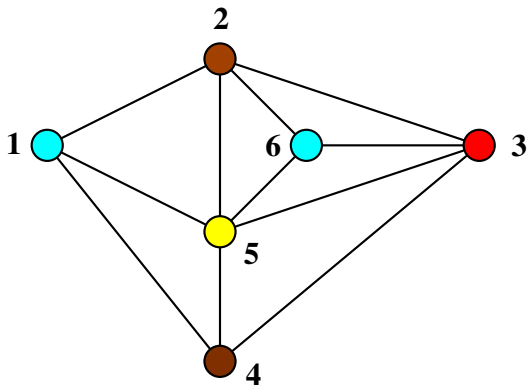
Brief notes on Backtracking and Branch-and-Bound

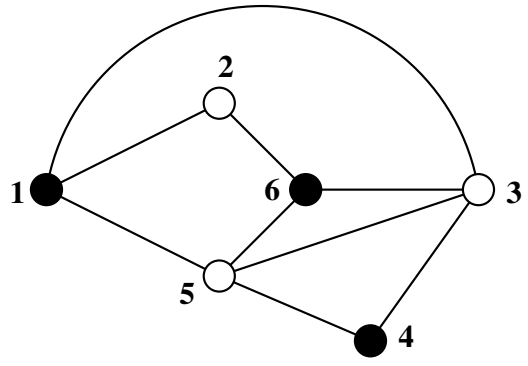
Brendan McKay
Research School of Computer Science
Australian National University

July 17, 2014

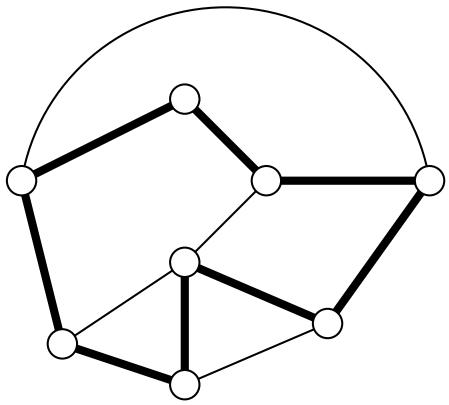
Problem 1. We have a graph G and wish to colour the vertices with a given set of colours so that adjacent vertices have different colours.



Problem 2. We have a graph G and wish to list all the independent sets of G. An *independent set* is a set of vertices which have no edges between them, such as the black vertices in the following.



Problem 3. We have a graph G and wish to find a Hamiltonian cycle in G. A *Hamiltonian cycle* is a cycle that visits every vertex, such as the bold cycle in the following.



In general we have a description of what a *solution* is, and want to find to all the solutions. Or, we might want to find just one solution. In the **backtracking** paradigm we deal with sets of solutions defined by properties, i.e.

"all solutions that have ⟨some property⟩".

We find the set of *all* solutions by successively partitioning the property "true" (which all solutions have) into stricter properties.

We need:

(a) A **test** $\boldsymbol{T}$ that sometimes reveals the complete list of solutions having a given property.

(b) A **refinement** $\boldsymbol{F}$ that partitions a property P into a finite set $P_1, \ldots, P_k$ of stricter properties. (By "partitions" we mean that any solution with property P must have exactly one of the properties $P_1, \ldots, P_k$.)

Then the backtracking procedure is:

```
procedure search(P : property)
    if test T applies to P then
        output all the solutions having property P
    else
        Say refinement F partitions P into P_1, ..., P_k.
        for i from 1 to k do search(P_i) endfor
    endif
end
```

We initially call the procedure with the property "true", which defines the set of all solutions. It then writes all the solutions exactly once each, provided it does not recurse to infinite depth.

Consider Problem 1. Let $(c_1, c_2, \ldots, c_k)$ denote the property that vertex 1 has colour c_1, vertex 2 has colour c_2, and so on. We will allow some entries to be written as $*$, meaning "any colour", and any vertices not listed at the end will also be allowed any colour. For example, ( ) defines all colourings, and $(a, *, c)$ defines all colourings with vertex 1 having colour a and vertex 3 having colour c. (Vertices 2, 4, 5, ..., can have any colour.)

Test T: If every vertex has a colour specified, that is a solution if it is a valid colouring; otherwise there is no solution here.

Refinement F: Specify a colour for some vertex that was not specified before. To avoid equivalent colourings, we can restrict the colour of the new vertex to be either (1) a legal colour that has been used already, or (2) the first unused colour (if any).

To start the search, we call the procedure with property ( ).

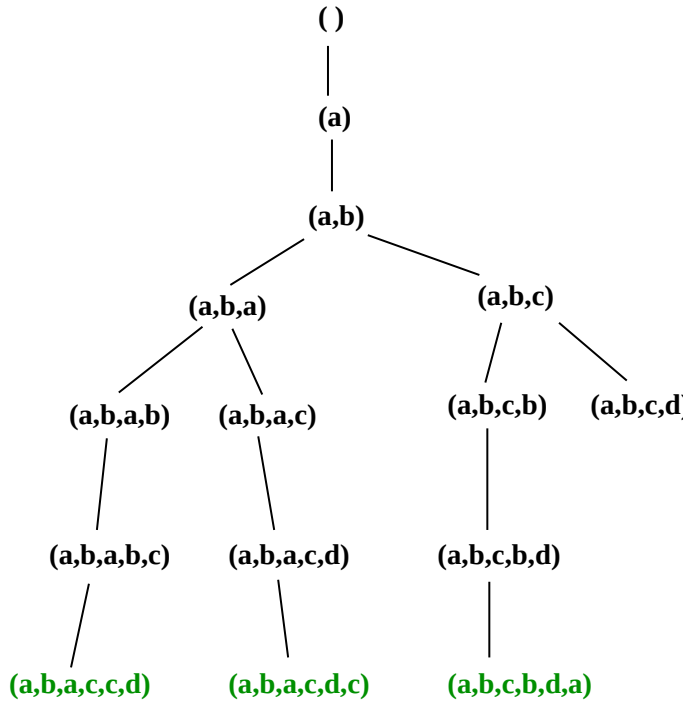
We always have a choice about which vertex to colour next when defining F. This can be very important for efficiency. Common choices are
(a) Choose the first uncoloured vertex.
(b) Choose the most restricted uncoloured vertex.
(c) Choose the first uncoloured vertex adjacent to the most coloured vertices.

More clever choices will cost more per node of the search, but might produce a much smaller search tree. The optimal choice needs to be determined by experiment.

Let's solve problem 1 for the graph above and available colours a, b, c, d. In defining F, we will use the first uncoloured vertex. For example

$$F : (a, b, a) \mapsto (a, b, a, b), (a, b, a, c).$$

We find 3 solutions:



Consider Problem 2. This is a little different, since the solutions are sets and we don't want to make any solution more than once. For example, we don't want to make the solution $\{2, 5\}$ and later make the solution $\{5, 2\}$, since these are the same.

The simplest way to achieve this is to arrange for the vertices of the independent set to be chosen in increasing order. One way is to write the solutions as boolean vectors, so that the solution $\{1, 4, 6\}$ shown in the figure is written as $(1, 0, 0, 1, 0, 1)$. The search can then fill in one extra 0 or 1 at each level.

Another way is to distinguish between sets that are *completed* and sets that are *partial*, which means possibly not completed. We will use "+" to indicate the second type. For example:

$\{1, 4\}$ means the set containing 1, 4, and nothing else
$\{2, +\}$ means all the independent sets containing 2 and possibly more vertices > 2.

Now we define the components of the solution:

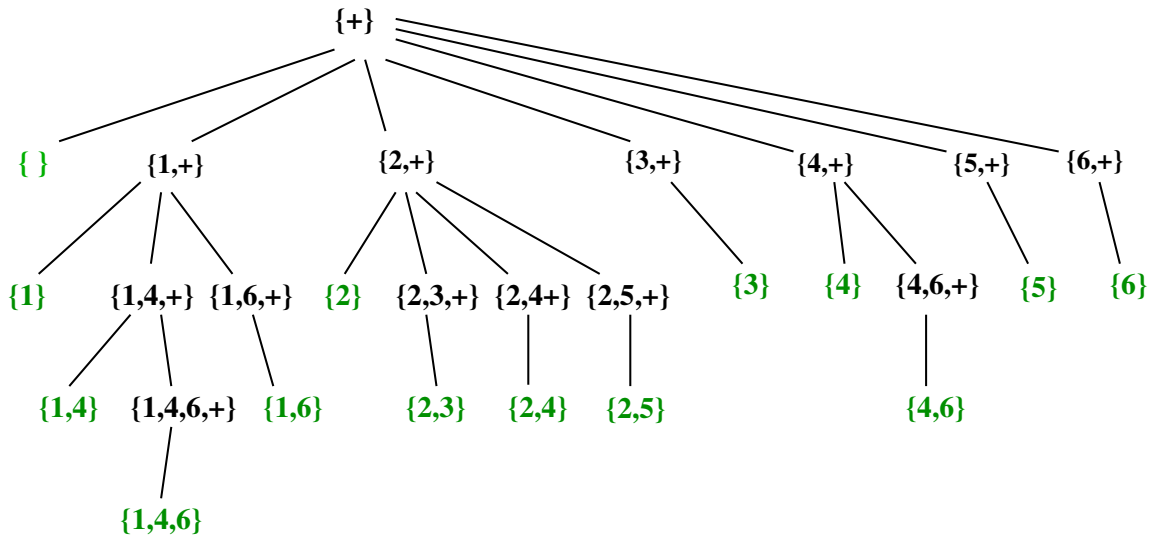
Test T: Every completed independent set is a solution by itself.

Refinement F: If $\{S, +\}$ is a partial solution, then F partitions it into (1) the completed solution $\{S\}$, (2) any partial solutions $\{S, v, +\}$ where v is a vertex greater than any vertex in S and not adjacent to any vertex in S.

For example,

$$F : \{1, +\} \mapsto \{1\}, \{1, 4, +\}, \{1, 6, +\}.$$

We find 14 solutions:



Consider Problem 3. The standard method for this problem is to start at some vertex and form the cycle one edge at a time. Thus, the refinement consists of adding one edge to a partial path.

A better way, at least if the number of edges in the graph is small, is to classify edges. Let's say that an edge e has type **yes** if it is assumed to be in the Hamiltonian cycle, type **no** if it is assumed to be not in the Hamiltonian cycle, and type **dunno** if we aren't assuming anything about it yet.

Once we have classified some edges, we can often classify more edges:

(a) If a vertex has less than 2 (**yes** or **dunno**) edges, or it has 3 or more **yes** edges, this classification cannot lead to a Hamiltonian cycle.

(b) If a vertex has exactly 2 (**yes** or **dunno**) edges, both can be classified **yes** and all its other edges can be classified **no**.

(c) Suppose a path P of **yes** edges has ends joined by a **dunno** edge e. If P includes all the vertices of the graph then e can be classified **yes**; otherwise it can be classified **no**.

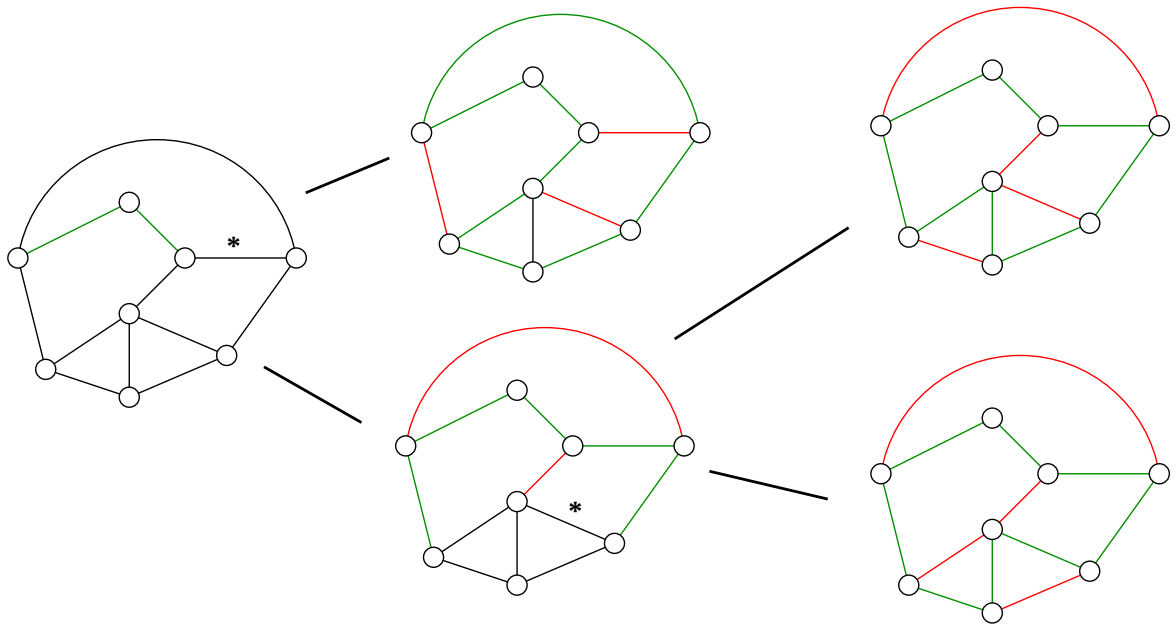
We can apply these classification rules at the beginning, and also after any edges are classified. To complete the definition of the algorithm, we need to define T and F.

Test T: If the **yes** edges form a Hamiltonian cycle, that is a solution.

Refinement F: (Assuming classification rule (a) hasn't eliminated this situation entirely:) Choose an edge classified **dunno** and classify it as either **yes** or **no**. In practice it is good if F chooses an edge close to previously classified edges.

In the following picture, **yes** edges are green, **no** edges are red, and **dunno** edges are black. The edge to be classified by F is indicated with an asterisk.

We find that there are three Hamiltonian cycles using at most two applications of F.



Optimization problems

Sometimes we don't just want solutions, we want an **optimal** solution. To be definite, let $f(s)$ be a real-valued function of a solution s. Let's seek a solution for which $f(s)$ is as *small* as possible.

- In Problem 1, we might want to minimize the number of colours. This is the **chromatic number problem**.
- In Problem 2, we might want to maximize the number of vertices in an independent set; this is the same as minimizing the number of vertices not in an independent set.
- In Problem 3, suppose that each edge has some *length* and we want to find a Hamiltonian cycle whose total length is minimum. This is the **travelling salesperson problem**.

We could of course find all the solutions and evaluate f for each of them. The method of **branch-and-bound** tries to improve this by using a *bounding function*.

Recall that backtracking deals with properties P that represent all solutions with that property. A **lower bound function** is a function $\mathrm{LB}(P)$ which is a lower bound $f(s)$ for all solutions s with property P. That is

$$\text{For all solutions } s \text{ satisfying } P,\ f(s) \geq \mathrm{LB}(P).$$

- In Problem 1, if P specifies some colours, LB(P) could be the number of different colours used so far.
- In Problem 2, if P specifies some vertices in an independent set, LB(P) might be the number of vertices that cannot be added to the independent set due to being adjacent to a vertex already there.
- In Problem 3, if P has selected some edges from the Hamiltonian cycle, LB(P) might be their total length (assuming edges can't have negative length).

We need a global variable f^* initialized to ∞. It will be set to the minimum solution value. We could also have a global variable to remember an example of a solution with that minimum value.

```
procedure BandB(P : property)
    if test T applies to P then
        f_P := the minimum f(s) for s with property P
        if f_P < f* then f* := f_P endif
    else
        Say refinement F partitions P into P_1, ..., P_k.
        for i from 1 to k do
            if LB(P_i) < f* then BandB(P_i) endif
        endfor
    endif
end
```

Branch-and-bound would be the same as backtracking except that the test

if LB(P_i) $< f^*$ **then ...**

prunes some parts of the search tree that cannot contain a better solution than seen so far. The efficiency thus depends crucially on

1. how accurate the lower bound function LB(P) is, and
2. how soon a very good solution is found.

Design of a good LB(P) is a a trade-off between the time taken to compute it and the amount of pruning it will cause.

To assist in finding a good solution early, the sub-problems $P_1, \ldots, P_k$ should be ordered so that the best is likely to be first. For example, in Problem 1, trying an existing colour is probably better on average than trying a new colour.