Searching on Graphs

Ganesh Ramakrishnan

October 6, 2008

Contents

Chapter 1

Searching on Graphs

Consider the following problem:

Cab Driver Problem A cab driver needs to find his way in a city from one point (A) to another (B). Some routes are blocked in gray (probably because they are under construction). The task is to find the path(s) from (A) to (B). Figure 1.1 shows an instance of the problem..

Figure 1.2 shows the map of the united states. Suppose you are set to the following task

Map coloring problem Color the map such that states that share a boundary are colored differently..

A simple minded approach to solve this problem is to keep trying color combinations, facing dead ends, back tracking, etc. The program could run for a very very long time (estimated to be a lower bound of 10^{10} years) before it finds a suitable coloring. On the other hand, we could try another approach which will perform the task much faster and which we will discuss subsequently.

The second problem is actually isomorphic to the first problem and to problems of resource allocation in general. So if you want to allocate aeroplanes to routes, people to assignments, scarce resouces to lots of tasks, the approaches we will show in this chapter will find use. We will first address search, then constraints and finally will bring them together.

1.1 Search

Consider the map of routes as in Figure 1.3. The numbers on the edges are distances. Consider the problem of organizing search for finding paths from S (start) to G (goal).

We humans can see the map geometrically and perhaps guess the answer. But the program sees the map only as a bunch of points and their distances. We will use the map as a template for the computer at every instance of time.

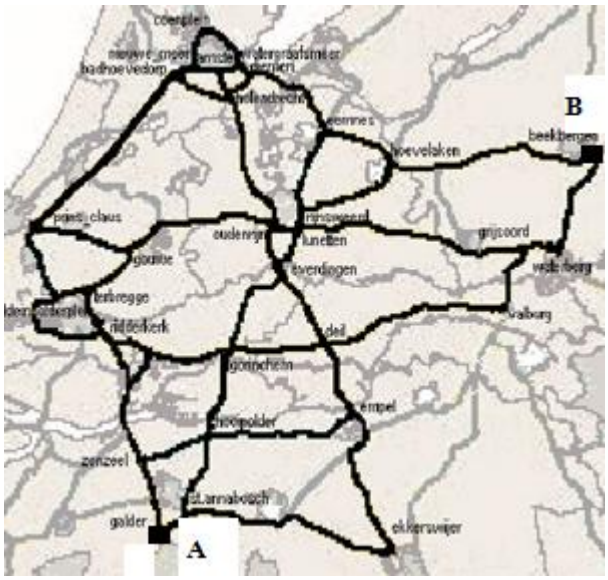

Figure 1.1: Routes in a city. The problem is to find the shortest path from point (A) to point (B).

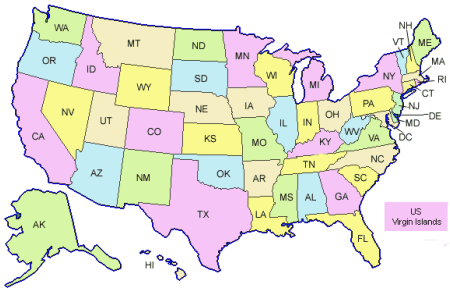

Figure 1.2: The map of the united states of America. The task is to color the map such that adjacent states have different colors.

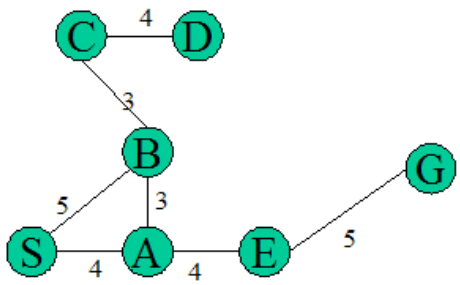

Figure 1.3: Map of routes. The numbers on the edges are distances.

At the beginning, the computer knows only about S. The search algorithm explores the possibilities for the next step originating from s. The possible next steps are A and B (thus excluding G). Further, from A, we could go to B or E. From B, we could go to A or C. From A, we could also go back to S. But we will never bother biting our own tail in this fasion; that is, we will never make a loop. Going ahead, from B, we could only go to C. From C, we can progress only to D and thereafter we are stuck on that path. From A, we could make a move to E and them from E to G. Figure 1.4 shows the exhaustive tree of possible paths (that do not bite their own tail) through this map. The process of finding all the paths is called the *British Museum Algorithm*[1].

But the British Museum algorithm can be very expensive. In fact, some of these searches can be exponential. If you had to look through a tree of chess moves for example, in the beginning it is essentially exponential, which is a bad news since it will imply 10^{10} years or so. We need a more organized approach for searching through these graphs. There exist many such organized methods. Some are better than others, depending on the graph. For example, a depth first search may be good for one problem but horrible for another problem. Words like *depth first search*, *breadth first search*, *beam search*, A^* *search*, *etc.*, form the vocabulary of the search problem in Artificial Intelligence.

The representation we use will define the constraints (for example, the representation of the routes in Figure 1.3 defines the notion of proximity between nodes and also defines constraints on what sequences of vertices correspond to valid paths.

1.1.1 Depth First Search

This algorithm[2] boils down to the following method: Starting at the source, every time you get a choice for the next step, choose a next step and go ahead. We will have the convention that when we forge ahead, we will take the first

[1]The British Museums are considered some of the largest in the world.

[2]Fortunately, the names given to these algorithms are representative of the algorithms actually do.

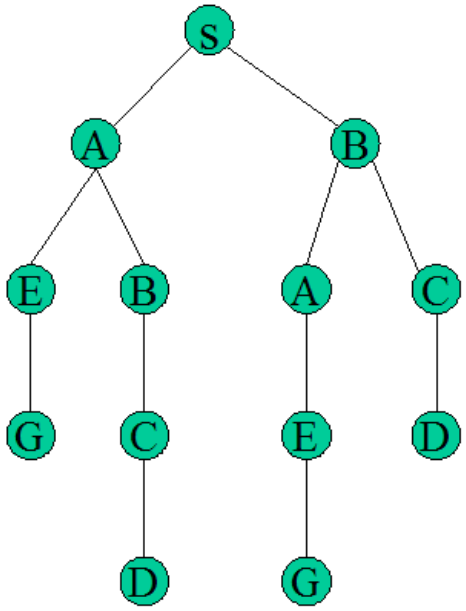

Figure 1.4: The tree of all possible paths from S to G through the map in Figure 1.3.

choice. Thus, a depth first search on the map in Figure 1.3 will tread along the path in Figure 1.5. In practice, what is used is a combination of depth first search and backup. What this means is that when a dead end is hit, the method goes back to the last state. In practice, this method could yield very complicated solutions.

1.1.2 Breadth First Search (BFS)

The way BFS organizes its examination of the tree is layer by layer; the algorithm first explores one layer of solutions (A or B), then forges ahead to another layer (B or E or A or C) and so on. Figure 1.6 illustrates the BFS traversal for the map in Figure 1.3. In practice, this search could expend a lot of time in useless parts of the graph. But it yields the shortest path in terms of number of number of streets covered.

1.1.3 Hill Climbing

Both BFS and DFS are completely uninformed about geography; neither information about distance nor direction is exploited. But often it helps if you have outside information about how good a place is to be in. For instance, in the map in Figure 1.3, between E and D, it is much better to be in E because that gets you closer to the goal. It makes less sense in general to head off to the right

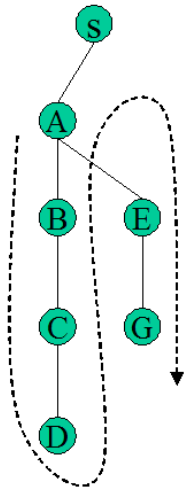

Figure 1.5: The DFS tree from S to G through the map in Figure 1.3.

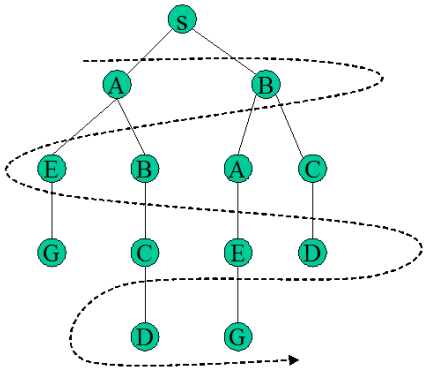

Figure 1.6: The BFS tree from S to G through the map in Figure 1.3.

Figure 1.7: The choices based on hill climbing for the route from S to G through the map in Figure 1.3.

if you know that the goal is on the left. It is heuristically good to be closer to the goal, though sometimes it may turn out not to be a good idea. So we could make use of heuristic information of this kind. And this forms the idea behind hill climbing. It is an idea drafted on top of depth first search. When initially at S, you could move to A or B. When you look at A and B, you that one of them is closer to the goal G and you choose that for the next move. And B happens to be closer to G, which you pick for the next move; but this turns out to be a bad idea as we will see. Nevertheless, it looks good from the point of view of the short-sighted hill climbing algorithm. From B, the possible choices are A and C. And A is a natural choice for hill-climbing, owing to its proximity to G. From A onwards, the choices are straightforward - you move to E and then to G. Figure 1.7 shows the route from S to G as chalked out by hill climbing.

Hill climbing always is greedy because it plans only for one step at a time. The method requires a metric such as distance from the goal.

1.1.4 Beam Search

We drafted hill climbing on top of depth first search. Is there a similar thing we could do with breadth first search? And the answer is 'yes'. And this approach

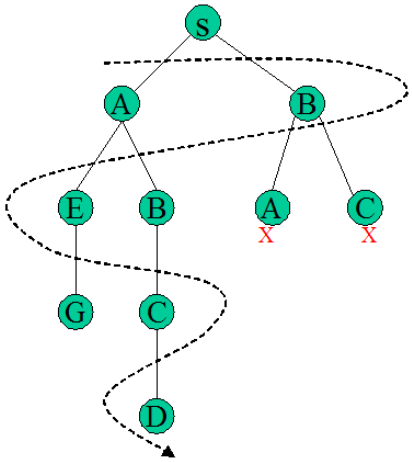

Figure 1.8: The choices based on beam search for the route from S to G through the map in Figure 1.3.

is called *beam search* The problem with breadth first search is that it tends to be exponential. But what we could do to work around is to throw away at every level of BFS, all but the 'most promising' of the paths so far. The most promising step is defined as that step which will get us 'closest' to the goal. The only difference from the hill climbing approach is that beam search does not pick up just one path, it picks up some fixed number of paths to carry down. Let us try beam search on the map in Figure 1.3. For this simple example, let us keep track of two paths at every level. We will refer to the BFS plan in Figure 1.6. From S, we could move to A or B and we keep track of both possibilities. Further, from A, there are two possibilities, *viz.*, B and E, while from B there are two possibilities in the form of A and C. Of the four paths, which two should we retain? The two best (in terms of the heuristic measure of how far we are from the goal) are B and E. Carrying forward from these points, we arrive at G in a straight-forward manner as shown in Figure 1.8.

While the vanila breadth first search is exponential in the number of levels, beam search has a fixed width and is therefore a constant in terms of number of levels. Beam search is however not guranteed to find a solution (though the original BFS is guaranteed to find one). This is a price we pay for saving on time. However, the idea of backing up can be employed here; we could back up to the last unexplored path and try from there.

1.2 Optimal Search

For optimal search, we will start with the brute force method which is exponential and then slap heuristics on top to reduce the amount of work, while still assuring success.

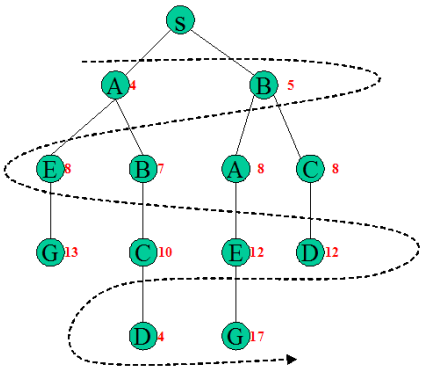

Figure 1.9: The tree of all possible paths from S to G through the map in Figure 1.3, with cumulative path lengths marked at each node.

1.2.1 Branch and Bound

Suppose the problem is to find the best possible path (in terms of distances) between S and G in the map as in Figure 1.3. An oracle suggests that the path $SAEG$ is the shortest one (its length is 13). Can we determine if $SAEG$ is indeed the shortest path? One method to answer this question is to verify if every other path is at least that long.

From S, we could go to A or B. The cumulative path length to B is 5. From B, you could either go to A or C. The cumulative path length upto A or C is 8. At this point, from A, you could go to E while from C you could move to D, with cumulative path lengths of 13 each and so on. Figure 1.9 shows the tree of possible paths from S to G, with cumulative path length (from S) marked at each node. It is evident from the figure that all paths to G have length greater than or equal to 13. We can therefore conclude that the shortest path to G from S is $SAEG$ and has length 13.

Most often we do not have any oracle suggesting the best path. So we have to think how to work without any oracle telling us what the best path is. One way is to find a path to the goal by some search technique (DFS or beam search) and use that as a reference (bound) to check if every other path is longer than that. Of course, our first search may not yield the best path, and hence we might have to change our mind about what the best path (bound) is as we keep trying to verify that the best one we got so far is in fact the best one by extending every other path to be longer than that. The intuition we work with is to always push paths that do not reach the goal until their length is greater than a path that does reach the goal. We might as well work only with shortest paths so far. Eventually, one of those will lead to the goal, with which we will almost be done, because all the other paths will be about that length too, following which we just have to keep pushing all other paths beyond the goal. This method is called branch and bound.

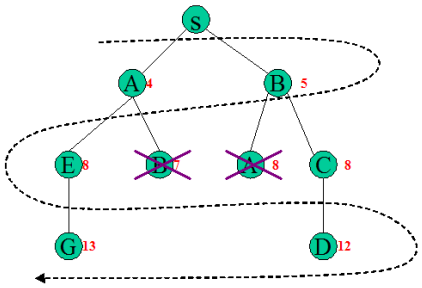

Figure 1.10: The pruned search tree for branch and bound search.

In the example in Figure 1.9, the shortest path upto B from S was of length 5 initially, which used as a bound, could eliminate the path SAB and therefore save the computation of the path $SABCD$. Similarly, the initial path to A, SA of length 4, sets a bound of 4 for the shortest path to A and can thus eliminate the consideration of the path SBA beyond A. Figure 1.10 illustrates the application of branch and bound to prune the BFS search tree of Figure 1.9. Crossed out nodes indicate search paths that are excluded because they yield paths that are longer than bounds (for the corresponding nodes). This crossing out using bounds is a variation on the theme of the dynamic programming principle.

1.2.2 A^* Search

The branch and bound algorithm can also be slapped on top of the hill climbing heuristic or the beam search heuristic. In each of these cases, the bound can be computed as the sum of the accumulated distance and the eucledian distance.

When the branch and bound technique is clubbed with shortest distance heuristic and dynamic programming principle, you get what is traditionally known as A^* search. Understanding A^* search is considered the culmination of optimal search. It is guaranteed to find the best possible path and is generally very fast.

1.3 Constraint Satisfaction

What we have discussed so far is mechanisms for finding the path to the goal (which may not be optimal). Rest of the discussion in this chapter will focus on resource allocation. Research allocation problems involve search. Too often people associate search only with maps. But maps are merely a convenient way[3] to introduce the concept of search and the search problem is not restricted to maps. Very often, search does not involve maps at all. Let us again consider the

[3]Maps involve making a sequence of choices and therefore could involve search amongst the choices.

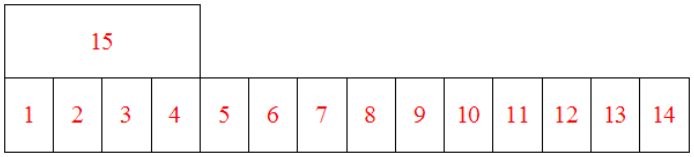

Figure 1.11: The sketch of a simple region for which we try to solve the map coloring problem.

"*Map coloring problem*" (refer to Figure 1.2). It involves searching for sequence of choices. Instead of the large city of USA, let us consider the simpler country of Simplea as in Figure 1.11. Let us say we need to pick a color from the set $\{R, G, B, Y\}$.

Let us say we number the regions arbitrarily. One approach is to try coloring them in the order of increasing numbers, which is a horrible way! This is particularly because the order chosen could be horrible. To color the region, let us start by considering the possible colors that could be assigned to each region. State number 1 could be assigned any of R, B, G or Y. Suppose we take a depth first search approach for this coloring job. Now we need to color the region number 2. Since we adopt depth first search, let us say we pick the color of R for region 1. And as we go from region to region, we keep rotating the set of colors.

The prescription for DFS is that you keep plunging head until you hit a dead end where you cannot do anything. But the problem is that we cannot infer we have hit a dead end till we assign colors to all 15 regions, to realise that the following constraint has been violated: *no two adjacent regions should have the same color*. And even if we backup a step or two, we will have to explore all 15 regions to infer if we have reached a dead end. At this rate, the projected time for successful coloring of the states in a US map is 10^{10} years[4]! That is no way to perform search. The culprit is the particular order in which we chose to color the regions. Figure 1.12 shows an example instance of random assignment of colors that leads to a dead-end (with no suitable color left for region number 15). We introduce the idea of *constraint checking* precisely to address this problem. It is essentially a (DFS) tree trimming method.

First of all, we will need some terminology.

1. *Variables (V):* V is a set of variables. Variables names will be referred to in upper case, whereas their specific instatiations will be referred to in lower case.

2. *Domain (D):* The domain D is the bag of values that the variables can take on. D_i is the domain of the $V_i \in V$. A good example of a domain

[4]The United states has 48 contiguous states. If the number of colors is 4, the branching factor of DFS will be 4 and the height will be 48. Thus, the number of computations will be 4^{48} or 2^{96} which is of the order of 10^{27} and since the length of a year is the order of 10^7, it will take in the order of 10^{11} years assuming that a computation is performed in a nano second.

15 ?													
1 R	2 G	3 B	4 Y	5 R	6 G	7 B	8 Y	9 R	10 G	11 B	12 Y	13 R	14 G

Figure 1.12: A sequence of bad choices for the map coloring problem.

would be $\{R, B, G, Y\}$. In the map coloring problem, all variables have the same domain.

3. *Constraints (C):* A constraint is a boolean function of two variables. C_{ij} represents the constraint between variables X_i and X_j. $C_{ij}(x_i, x_j) = true$ *iff*, the constraint is satisfied for the configuration $X_i = x_i, x_i \in D_i$ and $X_j = x_j, x_j \in D_j$. Else, $C_{ij}(x_i, x_j) = false$. The form of the constraint function could vary from problem to problem. An example of a constraint is: *no two adjacent regions should not have the same color.* Note that there need not be a C_{ij} for all i and j. For instance, in the map coloring problem, the only constraints are those involving adjacent regions, such as region 15 with the regions 1, 2, 3 and 4.

Suppose we are trying to explore if region number $i = 15$ is going to object to any coloring scheme. First we define the constraint $C_j, i = 15, j \in \{1, 2, 3, 4\}$ as follows:

$C_{ij}(x, y), i = 15, j \in \{1, 2, 3, 4\}$ *is true iff* $x \neq y$.

We can state the check as in Figure 1.13.

```
for x ∈ D_i (i = 15) do
  for All constraints C_ij do
    Set X_i = x iff ∃y ∈ D_j such that C_ij(x, y) = true
  end for
end for
```

Figure 1.13: The algorithm for constraint checking

1.3.1 Algorithms for map coloring

When assigning a color to each region, we could do different types of checks.

1. *Check all:* At one end of the spectrum, while assigning a color to a region, we check constraint satisfaction with respect to all regions (no matter how far they are from the region under consideration). This is an extreme case and potentially, we could consider all variants of what to check and what not to check. The check is only a speedup mechanism. It neither ensures nor prevents a solution. We could get a solution (if there is one) using depth first search and no search, thought it might take 10^{10} years. If we

want to have minimum number of checks such that they are really helpful, our method for deciding what to check will be checks on regions that are in the immediate vicinity of the last region that had a color assigned.

2. *Check Neighbors:* Check for constraint satisfaction, only all regions in the immediate neigborhood of the last region that had a color assigned. This is at another end of the spectrum.

3. *Check Neighbors of Neighbors:* Check for constraint satisfaction, only all regions in the immediate neigborhood of the last region that had a color assigned as well as well as regions in the immediate neighborhood of the neighbors.

4. *Check Neighbors of 'unique' neighbors:* Check for constraint satisfaction, only all regions in the immediate neigborhood of the last region that had a color assigned as well as well as regions in the immediate neighborhood of the 'unique' neighbors. A neighbor is 'unique', if the constraint set for the region has been reduced to a single element (which means that it has a very tight constraint).

We will evaluate the above approaches on two fronts, *viz.*,

1. *Number of assignments performed:* An *assignment* corresponds to putting a color on a region. If we are very lucky, we might be able to color the map of the United states with just 48 assignments (that is we never had to backup). Constraint checking will enable the algorithm realise that it will soon hit a dead end and will have to back up. In practical situations, we could expect a bit of backup and also expect the constraints to show you some dead ends. So it could happen that the constraint checking reduces the number of assignments to some number slightly above the ideal (say 52).

2. *Number of checks performed:* A *check* corresponds to determining which color could be assigned to a region based on the color assigned so far to another regions.

A simple experiment for coloring the Unites States reveals the statistics in Table 1.1.

The message we can extract from Table 1.1 is that some of the constraint checking is essential, else it takes 10^{10} years. We can conclude that full propogation is probably not a worthy effort. Because full propogation, relative to propogation through unique values yielded the same number of assignments but relatively fewer checks. When we compare heuristic 2 against heuristic 4, we find that there is a tradeoff between number of assignments and number of checks. Checking all is simply a waste of time. The net result is that people tend to invoke either of heuristics 2 and 4. From the point of view of programming simplicity, one might just use heuristic number 2.

SrNo	Constraint Sat Heuristic	Assignments	Checks
1	Check all	54	7324
2	Check neighbors	78	417
3 of neighbors	Check neighbors	54	2806
4 of 'unique' neighbors	Check neighbors	54	894

Table 1.1: Number of assignments and checks for each of the constraint satisfaction heuristics for the problem of coloring the map of the USA using 4 colors.

The computational complexity of the discussed methods is still exponential[5]. But usually, the constraints try to suppress the exponential nature.

1.3.2 Resource Scheduling Problem

Let us forget about maps for a while and talk about airplanes. An upstart airline is trying to figure out how many airplanes they need. They have a schedule outlining when and where they need to fly planes. And the task is to figure out the minimum number of airplanes that need to be purchased. For every airplane saved, let us say the reward is saving on half the cost of the airplane.

$F_1, F_2, \ldots, F_n$ are the flights. Table 1.2 shows the schedule for the flights.

Flight No.	From	To	Dept. Time	Arr. Time
F_1	Boston	LGA	10:30	11:30
F_2	Boston	LGA	11:30	12:30
F_3	Boston	LGA	12:30	13:30
F_4	Boston	LGA	13:30	14:30
F_4	Boston	LAX	14:30	15:30
F_6	...	...	...	
...	...	...	...	
F_15	Boston	LAX	11:00	15:30

Table 1.2: Number of assignments and checks for each of the constraint satisfaction heuristics for the problem of coloring the map of the USA using 4 colors.

To fly this schedule, we have some number $m = 4$ of airplanes: P_1, P_2, P_3, P_4.

[5]Note that these are NP complete problems. Any polynomial time algorithm will fetch field medals.

Of course, we do not want to fly any plane empty (dead head).

The assignment $P_1 \rightarrow F_1$, $P_2 \rightarrow F_2$, $P_3 \rightarrow F_3$ and $P_4 \rightarrow F_4$ will leave no airplane for flight F_5. But if we could fly P_1 back from New York (LGA), the same plane could be used for flight F_4, thus sparing P_4 for F_5. We can draw the correspondence between this problem and the map coloring problem; the 4 airplanes correspond to 4 colors while the flights correspond to regions. The task is to assign planes (colors) to flights (regions), at all times honoring constraints between the flights. The constraints are slightly different from the ones we had in the US map. The constraint is that no airplane can be on two routes at the same time. This implies that there is a constraint between flights F_1 and F_2. Similarly, there is a constraint between the pairs $< F_2, F_3 >$, $< F_3, F_5 >$, $< F_4, F_5 >$, $< F_1, F_5 >$, $< F_2, F_5 >$ and $< F_3, F_5 >$. We assume that the turn around duration for F_1 (which is of a one hour duration) will end by 14:30 hours, which sounds reasonable. Thus, there is no constraint $< F_1, F_4 >$.

You schedule planes the same way you do map coloring. An assignment is tried and constraints for all unassigned flights are checked to ensure that there is at least one airplane that can fly each flight. There is one important difference between the flight scheduling problem and the map coloring problem: we know that we can color maps with 4 colors[6], but we do not know how many airplanes it is going to take to fly a schedule. Hence, we will need to try the flight scheduling problem with different number of airplanes.

Let us say we over-resource the map coloring problem with 7 colors instead of 4. A sample run yields 48 assignments (that is no backup was required) and 274 checks. With 6 colors, you get 48 assignments and 259 checks. If on the extreme end, you used only 3 colors, you could never color Texas.

Frequently the problem is over-constrained and there is no solution with available resources (like say coloring the United States with 2 or 3 colors). In those circumstances, constraints can be turned into preferences so that some regions will not be allowed to be adjacent to regions of the same color. Or we might have to allow some 'dead-hit' flights. And on top of preferences, we could layer beam-search or some other search that tries to minimize the penalty cumulated or maximize the number of constraints that are satisfied.

[6]The 4 color theorem.

References