Bidirectional Search

Bidirectional search is a neat trick that can often take an exponential algorithm and make it run on problems that are twice the size of those it could previously solve. This may sound like a small improvement, but consider a $\Theta(2^n)$ algorithm that works for n up to 25. If bidirectional search can be applied, and n can be increased to 50, then we have just made an improvement in running time by a factor of several million!

Let's start with an example. You probably know the classic "15 puzzle". It is a 4x4 grid with 15 squares sliding up, down, left and right on it. Each square has the a number between 1 and 15 on it, and the goal is to arrange the squares in increasing order with the hole appearing in the lower-right corner. Here is a nice, short description and a few random facts about the 15 puzzle: http://mathworld.wolfram.com/15Puzzle.html. Suppose that we are given a scrambled puzzle and asked to find the shortest sequence of moves that solves it.

This is just a graph search problem. Each puzzle configuration is a vertex, and it has up to 4 edges coming out of it because there are at most 4 moves available. The problem is – there are 16! vertices, so a simple BFS will not work, but let's consider it anyway. We start at the given configuration and explore its neighbours in the graph, marking them as visited and adding them to the BFS queue, as usual. We should also store the move that we used to construct each new configuration so that we can recover the move sequence later.

The first question here is: "How do we represent a configuration?" We could use a 4x4 array or a vector of vectors of integers, or a string. A neat approach is to notice that we have 16 squares, each containing an integer in the range 0 though 15 (0 representing the hole). So each cell can be represented by a 4-bit number for a total of 4*16=64 bits. This fits nicely into a 64-bit integer (long long type in gcc/g++). Since we will be generating lots of these configurations, using this compact representation is a very good idea, especially because we are going to need a map or a hashtable of some sort to store the depth and predecessor of each node during BFS.

Now we have a BFS algorithm on a graph of size 16 factorial. We could terminate our BFS as soon as we reach the target configuration and avoid exploring the whole graph. This will work for quite a few problem instances. Unfortunately, the in the worst case, the number of moves required to solve the puzzle is 80, and the BFS solution will crawl to a halt after about 20 or so moves. Although solving the puzzle in the general case is very difficult, we can at least improve our solution to work on instances that require at most 40 moves.

This is where bidirectional search comes in. We know two things – the initial, scrambled configuration (call it S) and the final, solved one (let's call it T). Until now, we have been searching from S to T, but the problem is symmetric – we might as well have been searching from T to S. In fact, this in itself is a better idea because T is fixed, and S is different in every instance of the problem, so we could do just one complete BFS backwards from T and then immediately solve any problem instance by tracing the path from S to T. This would work if we had enough memory to store all 16! configurations and enough CPU time to compute them.

A better idea is to run two BFS processes - one from S forward and one from T backward. Both will grow BFS trees with configurations as the vertices. If we ever hit the same vertex from both sides, then we are done. This vertex (call it v) is an intersection point between the BFS tree of S and the BFS tree of T, so we can trace the path from S to v and append to it the path from v to T, and we get a shortest path from S to T – our answer. If we can manage to get to the depth of 19 in each tree, then the total path length will be 38 - almost twice as good as with a single BFS, but we need twice the amount of memory for it.
Bidirectional search works in many situations where we are searching for a path of some sort from X

to Y, and each move on the path is reversible to allow for a backwards search. To implement the two searches, we could either alternate between them, making one move at a time (pop from the BFS queue and explore neighbours), or we could run one search to a fixed depth, and then run the other search.

Iterative Deepening

We have seen backtracking with branch-and-bound that looked like a DFS. We have also seen bidirectional search with BFS. The latter required a lot of memory – it is proportional to the number of visited vertices. Fortunately, each vertex only has at most 4 neighbours, many of them repeated, so we could get to a depth of about 20 before running into memory problems. What if each vertex has 10 neighbours? Then each level in the BFS tree would be 10 times larger than the previous level, and we would start having memory problems on level 7 or 8. "Forgetting" DFS from last time, on the other hand, requires very little memory; its memory usage is proportional to the depth of the current path.

Iterative deepening allows us to simulate BFS with multiple runs of DFS at the cost of doubling the running time, but in most cases – the extra cost is insignificant. The main idea is to add a depth limit to the DFS and make the recursive function return immediately once the limit is reached. We first run it with depth limit 0. This will only visit the starting vertex, S. Then we run it again with depth limit 1. This time DFS will visit S and all of its immediate neighbours. We continue this way, each time increasing the depth limit by 1 until we find the destination.

Consider the following problem. An *addition chain* for a given number, K, is an increasing sequence of integers that starts with 1 and ends with K. Each member of the sequence, except for the first one is the sum of two of the previous, not necessarily distinct members. For example, (1, 2, 3, 4, 7) is an addition chain for 7 because 1+1=2, 1+2=3, 1+3=4 and 3+4=7. The problem is to find the shortest addition chain for a given K. (1, 2, 3, 4, 7) is a shortest addition chain for 7, but it is not unique. (1, 2, 4, 6, 7) works, too.

Addition chains can be used in public-key cryptography. Computing the shortest addition chain of K is a famous problem in number theory, and there are several heuristics that can be used. We will look at how iterative deepening can help here.

The simplest backtracking solution is to start with 1 and successively add one integer to the sequence, trying all possible values for the new integer. However, this approach tends to generate long chains of the form (1,2,3,4,...). What we would ideally like is to run BFS from the initial sequence (1) – that would give us the shortest path (shortest addition chain) to K. However, this would require keeping lots of partially built addition chains in a queue, eating up too much memory. There is a better way.

Suppose that the optimal addition chain for K has length R. Then if we build only those additions chains that are no longer than R, we will find the answer. Unfortunately, we do not know R in advance, but we can try all lengths incrementally. First, we generate all addition chains of length 1 (there's only one of those). Then we generate all chains of length 2 (only 1 again). Then find all of length 3, etc. This is done by adding a maximum recursion depth variable to the backtracking function. Once the depth limit is reached, we cut the recursion branch and move on to a different branch.

After trying one maximum depth, we throw away everything and start clean searching to a higher depth. This technique works quite well for this problem. Straightforward backtracking will not be able to handle K values higher than 30 or 40. Iterative deepening increases the limit to over 1000.
It may seem like a waste of computation time to throw away all previous computation and start anew

with the next depth limit, but we are not losing that much time. Suppose that we have an exponential backtracking algorithm that takes 2^n steps to explore every configuration up to depth n. Then running it to all depths up to n will require $2^0+2^1+...+2^n=2^{n+1}$ steps – only twice as many as for depth n alone. If the branching factor is larger than 2, then the difference in running time is even smaller. And, of course, the benefit of iterative deepening is that we save a lot of memory compare to plain BFS search.

Split and Merge

A famous problem solving strategy is "divide and conquer" – break down a problem into sub-problems, then combine their results into a solution of the bigger problem. When a problem can be solved this way, the algorithm is usually fast and efficient, because the "division" step generates a linear number of problem instances. While it's unlikely that we can use divide and conquer on an NP-complete problem, its idea is still quite useful. Split and Merge is a clever trick that uses this idea and makes an algorithm able to solve problems twice the size of those that it could solve previously.

Consider a famous NP-complete problem – the **Partition Problem**. The partition problem asks, if given a set of positive integers, can we divide the set into two disjoint subsets so that the sum of one subset equals the sum of the other. For example, given the set S={1, 2, 4, 7, 9, 13}, we can break it into S_1={1, 4, 13} and S_2={2, 7, 9}, and we get 1+4+13=18=2+7+9. There are other solutions, too, for this set, but given an arbitrary set, can we determine whether there is such a division? (We ignore the trivial case when the total sum of the given set is odd.)

Let's look at the straightforward backtracking solution to this problem. Although we can still model this problem as a graph search problem, the graph becomes very messy and inconvenient to use. We can use the structure of the problem to our advantage. The problem statement asks that we divide the set into **two** disjoint subsets, so each number in the set can be either **in** the first subset or **not in** it. Hence, the algorithm is quite simple – we recurse on each number in some order, and for each number we first use it and recurse, and then backtrack to not use the number and recurse again. For a set of size n, this gives a $\Theta(2^n)$ algorithm. Implementing this algorithm is a fun and rewarding experience, as it illustrates some of the most fundamental concepts of recursive search.

This is, however, quite slow. When n gets to about 29, the running time of the algorithm hits 10 seconds (worst case), and a value of n = 60 would be unbearable. We can do better though, by using the Split and Merge trick mentioned above. This method first **splits** the problem into two parts, runs the exponential algorithm on each part individually, and tries to combine the results from both sides to solve the main problem. For the partition problem, we would first split the set into two parts of the same (or nearly the same) size arbitrarily. Note that this split has nothing to do with the two disjiont subsets; the split simply divides the size by 2 so that the exponential algorithm can handle each part.

Now, to combine the results from the two parts, we need to know **all** possible sums from the first and second part, so that we can check whether there is a sum from the first part and another from the second part that add up to half the total sum. What this means is that, when we recurse on the first part and get to the last integer, we store the accumulated sum in a STL set. Now, when we recurse on the second part and get to the last integer, we check in the STL set for a corresponding sum that gives us a solution. In effect, we are **merging** the sums from the two parts together to form one solution.

How does split and merge algorithm affect the run-time and memory of our algorithm? First, the memory usage is huge, as for a set of size n, we would have $\Theta(2^{(n/2)})$ number of sums for each part, so the STL set has exponential size. A practical limit for today's computers is around 256MB, which means n/2 can be as big as 24-26 (depends on implementation). The time complexity is quite interesting. For the first recursion, we have an algorithm that inserts $O(2^{(n/2)})$ sums into the set. Each insertion takes logarithmic time, so the toal time is

$$O(2^{(n/2)}\log(2^{(n/2)}))=O(\frac{n}{2}2^{(n/2)}) \quad .$$

For the second recursion we don't insert anything, but we search within the STL set the same number of times, and a search also takes logarithmic time, so the time for the second part is the same as the first part. This totals up to a $O(n2^{(n/2)})$ algorithm. Hence, at a large cost on memory, we manage to turn an algorithm into one that solves problem instances that is almost twice the size. With this algorithm, it is possible to solve the problem with n as big as 50 (bearing memory limit).

One may ask now, why couldn't we split the smaller parts even more, and solve their sub-problems, like in Divide and Conquer? The answer is that when we split, the question that we ask on the smaller parts is different from the question that we ask on the original set. For the bigger problem, we want to know whether there is some partition of the set; but for each smaller part, we want to know every possible sum that the part can make. We can merge the results only because the results we gather from the smaller parts are compatible in solving the larger problem. To split the smaller parts even more, we would have to find all sums of the part in time less than $\Theta(2^{(n/2)})$, but this is theoretically impossible (in general) as the output is of the same size. (Although later on in Dynamic Programming we'll see how some restrictions on the size of the integers in the input set can improve this part significantly).

The discerning reader will have probably noticed that Split and Merge is much like bidirectional search. Indeed, split and merge is only a special case of bidirectional search. We mention it particularly because some problems cannot be easily transformed into a graph search problem, and are more combinatorial in nature. In these problems (like the partition problem), we get to **choose** at each point some variables to determine the next point in the search. In general, these problems are more susceptible to attack by split and merge, because the trick itself is combinatorial in nature.