Bitmask Dynamic Programming

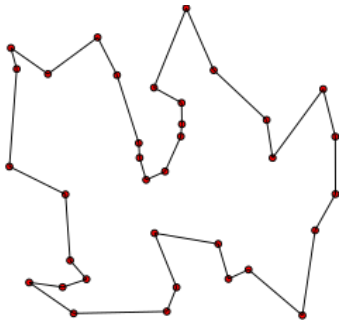
What is Bitmask DP?

Bitmask DP is a type of dynamic programming that uses *bitmasks*, in order to keep track of our current state in a problem.

A bitmask is nothing more than a number that defines which bits are *on* and *off*, or a binary string representation of the number.

Traveling Salesman Problem

The traveling salesman problem is a classic algorithmic problem defined as follows: given a list of *N* cities, as well as the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns back to the city that you started from? (*Note*: we can start from any city as long as we return to it)



Take I - Permutations

We can create a permutation of the *N* cities to tell us what order we should visit the cities.

For example, if we have *N* = 4, we could have the permutation {3, 1, 4, 2}, which tells us to visit the cities in the following order: 3 --> 1 --> 4 --> 2 --> 3

For a given *N*, there are *N!* possible permutations (*N* choices for the first city, *N-1* choices for the second city, *N-2* choices for the third city, etc.)

However, if *N* > 12, this algorithm will be too slow, so we need to shoot for something better

Take II - Bitmask DP

Let's use bitmasks to help us with this problem: we'll construct a bitmask of length *N*.

What does the bit at index *k* represent in the scope of the problem? What does it mean for the bit to be zero? What about one?

- if the bit at index *k* is zero, then we *have not yet* visited city *k*
- if the bit at index *k* is one, then we *have* visited city *k*

For a given bitmask, we can see which cities we have left to travel to by checking which bits have a value of zero.

We can construct a recursive function to find the answer to this problem:

```
// The following function will calculate the shortest route that visits all
// unvisited cities in the bitmask and goes back to the starting city
public double solve( int bitmask )
{
    if ( dp[ bitmask ] != -1 )
        return dp[ bitmask ];
    // Handle the usual case
}
```

What are some base cases for this problem? How do we know when we are done?

What does the bitmask look like when we have visited every city?

- 111...111

In this case, we need to return to our starting city, so we'll just return the distance from our current location to the start city, but what is our current location?

Imagine we currently have the bitmask 11111.

This bitmask tells us that we have visited all $N = 5$ cities and can return home.

However, this bitmasks *does not* tell us which city we are currently at, so we don't know which distance to return.

This means that in addition to the bitmask, we will need to keep track of the *current location* (anytime we travel to a city, that city becomes the current location).

```
// The following function will calculate the shortest route that visits all unvisited
// cities in the bitmask starting from pos, and then goes back to the starting city
public double solve( int bitmask, int pos )
{
    if ( dp[ bitmask ][ pos ] != -1 )
        return dp[ bitmask ][ pos ];
    // Handle the usual case
}
```

If we know that `solve( bitmask, pos )` will give us thge answer to the traveling salesman problem, and we say that we start at city 0, what should our initial function call be?

What is the value of `bitmask` ?

- 1 (we visited city 0, so our starting bitmask is 000...0001)

What is the value of `pos` ?

- 0

So how do we handle the usual case?

We have a bitmask that tells us which cities we have left to visit, and we know our current position, so which city do we go to next?

- whichever one will minimize the remaining route

To see which city will minimize the value we want, we need to try *all* possible remaining cities and see how it affects the future routes.

What does trying to visit a city look like?

Let's say that we are visiting some city at index *k*:

```
int newBitmask = bitmask | ( 1 << k );
double newDist = dist[ pos ][ k ] + solve( newBitmask, k );
```

We need to update our bitmask, marking city *k* as visited, and then we add the distance from our current loication to city *k*, and then solve the subproblem where we find the shortest route starting at city *k* and solving for the rest of the cities.

If we do this for each city that hasn't been visited yet, we want to take the minimum `newDist` that we find.

```
// The following function will calculate the shortest route that visits all unvisited
// cities in the bitmask starting from pos, and then goes back to the starting city
public double solve( int bitmask, int pos )
{
    // If we have solved this subproblem previously, return the result that was recorded
    if ( dp[ bitmask ][ pos ] != -1 )
        return dp[ bitmask ][ pos ];

    // If the bitmask is all 1s, we need to return home
    if ( bitmask == ( 1 << N ) - 1 )
        return dist[ pos ][ 0 ];

    // Keep track of the minimum distance we have seen when visiting other cities
    double minDistance = 2000000000.0;

    // For each city we haven't visited, we are going to try the subproblem that arises from visiting it
    for ( int k = 0; k < N; k++ )
    {
        int res = bitmask & ( 1 << k );

        // If we haven't visited the city before, try and visit it
        if ( res == 0 )
        {
            int newBitmask = bitmask | ( 1 << k );

            // Get the distance from solving the subproblems
            double newDist = dist[ pos ][ k ] + solve( newBitmask, k );

            // If newDist is smaller than the current minimum distance, we will override it here
            minDistance = Math.min( minDistance, newDist );
        }
    }

    // Set the optimal value of the current subproblem
    return dp[ bitmask ][ pos ] = minDistance;
}
```

Problems

- Collecting Beepers
- Forming Quiz Teams
- Square
- Pebble Solitaire
- Happy Valentine Day
- Cleaning Robot
- You Win!