CS311: Homework #7

Due on December 6, 2013 at 4:30pm

Professor Lathrop Section 3

Josh Davis

Problem 1

Use simulated annealing to find approximate solutions to the optimization variant of **No-Three-In-Line**.

Part A

Specify in pseudocode a fitness function that evaluates potential solutions to **No-Three-In-Line**.

Solution

```
function NoThreeInLineFitness($n, board$)
    $cost \leftarrow 0$

    for all $i \in 1 \ldots n$ do
        for all $j \in 1 \ldots n$ do
            if more than three horizontally at $i, j$ then
                $cost \leftarrow cost + 1$
            end if
            if more than three vertically at $i, j$ then
                $cost \leftarrow cost + 1$
            end if
            if more than three diagonally at $i, j$ then
                $cost \leftarrow cost + 1$
            end if
        end for
    end for

    return $cost$
end function
```

Algorithm 1: Fitness algorithm

Part B

Specify an appropriate set of moves for the **No-Three-In-Line** problem.

Solution

Say the possible solution that the algorithm were checking had all of the points clustered in the top left of the board. The value returned by **NoThreeInLineFitness** would be quite high.

The problem is that it might continue searching for values with lots of values clustered. Then given the chance of randomly selecting a worse or similar valued solution, it might pick a distribution where all lines are in a straight line. This would still have a high cost but be closer to the actual solution.

Part C

Discuss strengths and weaknesses of your fitness function and set of moves. Are potential solutions that the set of moves label as 'near' actually similar in a useful sense? Why or why not? How does this affect the effectiveness of the simulated annealing algorithm?

Solution

The strength is that the cost of when three are in a line will add up real quickly to the cost, preventing it from being a good solution.

Problem 1 continued on next page. . .

The ones that are near aren't really that useful because they probably just represent mild improvements over the current positioning.

Just in my little experience, I feel that it would make the simulated annealing algorithm more effective because it keeps bad solutions near each other which makes it easier to not get trapped.

Problem 2

Backtracking.

Part A
Specify a set of partial candidates and a method for generating candidate extension steps.

Solution

We can start with an empty board and try going by one row at a time. If we do this, we just need to try to place a dot if it isn't collinear with two other dots. If we find a place that doesn't clash, we add it to our solution and continue.

Part B
Specify in pseudocode a method that efficiently determines whether a partial candidate should be pruned.

Solution

Pruning algorithm that takes in a candidate represented as the board and the next position in the board to place a point in, i, j:

```
function ShouldPrune($board, i, j$)
    if points in this row, $i$ then
        return $false$
    end if
    if points in diagonal from $i, j$ then
        return $false$
    end if

    return $true$
end function
```

Algorithm 2: Backtracking pruning function

Part C
Given answers to (a) and (b), would a backtracking algorithm correctly solve the decision variant of **NoThree-InLine** for every n?

Solution

It should for as long as $n \geq 2$ and $n \geq 3$. Other than that, the backtracking algorithm will go back to the last place that wrorked if it can't find one that does and go with it.

Reference

1. Best explanation I could find of the problem:

   http://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/

Problem 3

Problem 4

Problem 5

CountBooleanParenthesizations(e)

Input: A sequence called e of $T, F, \vee, \wedge, \oplus$ symbols.
Output: Number of ways to parenthsize e to be *true*.

Solution

First let n be the number of values of T and F that appear in e. This means that there are $n-1$ operators in e. Let these values be v_i and the operators be o_i for $i = 1 \ldots n$.

Now we can create two recurrence functions to calculate our dynamic programming solution.

Let **True**(i, j) be the number of ways to parenthsize e such that it evaluates to true and let **False**(i, j) be the number of ways to parenthsize such that it evaluates to false.

We can then define **True**(i, j) and **False**(i, j) where i is the beginning of the parenthsization and j is the end as such:

$$T(i,j) = \sum_{k=i}^{j} \begin{cases} (T(i,k) + F(i,k)) + (T(k+1,j) + F(k+1,j)) - F(i,k) \cdot F(k+1,j) & o_k = \vee \\ T(i,k) \cdot T(k+1,j) & o_k = \wedge \\ T(i,k) \cdot F(k+1,j) + F(i,k) \cdot T(k+1,j) & o_k = \oplus \end{cases}$$

The reasoning is below:

1. $\vee$: We find the times each operand can be true by totaling up the parenthesizations of each part of the expression and just subtracting when $\vee$ is false, which is only when both are false.
2. $\wedge$: The number of true parenthesizations can only be the product of the subproblems.
3. $\oplus$: Since xor is only true when neither are true or neither are false, we take the sum of each product of the true and false for each subexpression.

Similarly as above, we can now determine our **False**(i, j) by just negating **True**(i, j):

$$F(i,j) = \sum_{k=i}^{j} \begin{cases} F(i,k) \cdot F(k+1,j) & o_k = \vee \\ (T(i,k) + F(i,k)) + (T(k+1,j) + F(k+1,j)) - T(i,k) \cdot T(k+1,j) & o_k = \wedge \\ T(i,k) \cdot T(k+1,j) + F(i,k) \cdot F(k+1,j) & o_k = \oplus \end{cases}$$

Reference

1. Dynamic programming solution was greatly inspired by problem 6: `http://courses.csail.mit.edu/6.006/fall10/handouts/dpproblems-sol.pdf`
2. I used it to learn more about dynamic programming. It certainly took awhile to understand but it certainly helped a lot.

Problem 6

EggDroppings(n, h)

Input: n the number of durable eggs and h the height of an apartment building
Output: The minimum number of drops necessary to *guarantee* that the value of c will be discovered before you run out of eggs.

Part A
Let $n = 1$. What dropping strategy for minimum value of c? What is the largest number of drops your strategy could require, and what value(s) of c cause this situation to arise?

Solution

Strategy: When $n = 1$, the strategy of starting at the lowest floor and dropping it until it breaks. The value of c will be equal to one less than the floor that it broke on.

Largest Number: The largest number of drops would of course be if we had to drop it until we reach the top of the apartment or when $c = h$.

Part B
Let $n = 2$. What dropping strategy for minimum value of c? What is the largest number of drops your strategy could require, and what value(s) of c cause this situation to arise?

Solution

Strategy: When $n = 2$, the strategy of splitting the number of floors in half until an egg cracks, then reverting back to the base case of testing from the lowest known floor until we find when it cracks.

It is very similar to binary search, we have a low and $high$ values that are initially $low = 1$ and $high = h$, then take the mid point. If the egg doesn't crack, $low = mid$ and then we repeat. When the egg cracks, we set $high = mid$ and start at low and go until $high$.

Largest Number: The largest number of drops would be when $c = \lceil h/2 \rceil$ because we would drop it once at $\lceil h/2 \rceil$ and it would break and then require $(h/2) - 1$ additional drops resulting in $h/2$ total drops.

Part C
Let $n \in \mathbb{Z}^+$. Write a dynamic programming algorithm that efficiently solves **EggDroppings**.

Solution
The dynamic programming algorithm is as follows:

$$E(n,h) = \begin{cases} 0 & \text{if } n = 0, k = 1 \\ \min\limits_{i=1\ldots n} \{1 + \max(E(n, h-i), E(n-1, i-1))\} & \end{cases}$$

The reasoning is because it is the smallest over possible max values where we check the value of when the egg doesn't break $E(n, h - i)$, and when the egg does break, $E(n - 1, i - 1)$.