COMP2007 - Assignment 3

Matthew Watson
SID: 440267858

1. **Design an algorithm that chooses the subset of houses with maximum combined value, given the constraint that no two chosen houses can be adjacent. That is, you can't choose both a house and its neighbour.**

   (a) **Design a dynamic programming algorithm to determine the value of the optimal choice of houses. First derive and justify a recurrence relation and your base cases, then turn it into a bottom-up solution. A more efficient algorithm gives more points.**
   To determine the optimal choice of houses, we must first derive a recurrence relation. We can define a function $OPT(i)$ as the maximum house value from the first house in our sequence up to and including that of the i^{th} house, and the value of the i^{th} house being v_i. When we reach the i^{th} house, there are two options; to add this house to our subset of houses with maximum combined value, or not. This is best explained by the following equation:

   $$OPT(i) = max(v_i + OPT(i-2) \text{ when the } i^{th} \text{house is added}, OPT(i-1) \text{ otherwise})$$

   From this we can form a bottom-up algorithm that computes the subset of houses with the maximum combined value. The order of our dynamic program will run as follows:

   i. Fill an array from the provided house values called *house_values* and the number of houses represented by *num_houses*.
   ii. Check if the number of houses is zero, in which case the maximum value will be zero.
   iii. Check if the number of houses is one, in which case the maximum value is the value of that one house.
   iv. Check if the number of houses is two, in which case we can only pick one of these houses as the other is a neighbour. So we find the max of the two values and return that as our maximum value.

v. Otherwise, we have a sequence of 3 or more houses, and can prepare our algorithm by computing the maximum value of the first two homes and setting that as our variable to keep track of our current maximum value, called *curr_max_value.* Then,

vi. We run a for loop from index 2 (3rd house) until index *num_houses* (the previously calculated number of houses in our sequence).

vii. At each iteration, we simply call on our function to determine whether we pick the house at our current i^{th} iteration. That is, we determine the greater value of the following two cases:

   - We chose this house at i to be a part of our subset and so the maximum value of such is the value of this house at i plus the maximum value up until the last non-adjacent house to i, i.e. up until *i-2.* Or,
   - We don't chose this house at i and so the maximum value will be that up until the last adjacent house, i.e. up until *i-1*

   The greater value is then added to our *curr_max_value* variable.

viii. With the maximum obtained at house i, we iterate through our for loop repeating our function until we have reached the end of our input sequence of houses and then return the last and final value of *curr_max_value* to give the combined value of our subset of non-adjacent houses with maximum combined value.

(b) **Argue the correctness of your algorithm.**

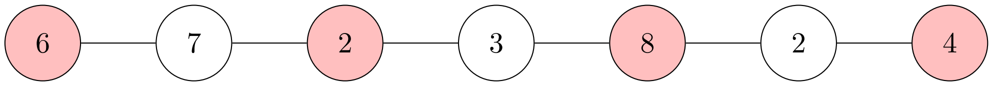
i. Case where $n = 0$: If there are no houses to visit, then there is 0 value and hence the final value is 0 as there can be no other houses to add value.

ii. Case where $n = 1$: If there is only one house, then the optimum value will be the value of this house, v_i as there would be no other houses to add value.

iii. Case where $n = 2$: If there are only two houses, then finding the optimum value will simply be the maximum value between these two houses as selecting none would give no value and both can't be selected as they cannot be adjacent. Hence the max value of the two houses where $n = 2$ is the optimum value.

Proof for n

iv. Case where $n = 7$:



The algorithm will start at the beginning of the list and compare the optimum values of $v_i + v_{i-2}$ (seen below on the left) and v_{i-1} (seen below on the right) until the list has been iterated through. See the following for an example iteration on the above houses list.

- $h1 = 6$
- $h2 = 6 \mid \mathbf{7}$
- $h3 = \mathbf{2{+}6} \mid\mid 7$
- $h4 = \mathbf{3{+}7} \mid\mid 8$
- $h5 = \mathbf{8{+}2{+}6} \mid\mid 3 + 7$
- $h6 = 2 + 3 + 7 \mid\mid \mathbf{8{+}2{+}6}$
- $h7 = \mathbf{4{+}8{+}2{+}6} \mid\mid 2 + 3 + 7$

Thus the value of our subset of houses of maximum combined value is $4 + 8 + 2 + 6 = 20$ which is the correct value of the 7 houses. As this continues for all sizes n, we conclude this algorithm is correct.

(c) **Prove an upper bound on the time complexity of your algorithm**
We first read in a sequence of house values that are appended to a list. Appending to a list takes $O(1)$ time and is repeated n times. This means $O(N)$ overall time for the data input. We then begin our algorithm. If the number of houses is 0, we return 0 in $O(1)$ time. If the number of houses is 1, we return the value at index 0 from *house_values* in $O(1)$ time. If the number of houses is 2, we simply return the max of the two houses. Retrieval of two values from a list is $O(1)$ time done twice to be $O(1)$ overall. The max operator on two numbers takes $O(1)$ time and so combined with the list retrieval, this step takes $O(1)$ time. Now that our edge cases are covered, we can initialise our *curr_max_value* variable and append the first house as the max value at index 0 and then the max of the first and second houses as the max value at index 1. Appending to a list is $O(1)$ time as well as our max operation making this $O(1)$ time overall. With this done, we enter a for loop from $i = 2$ to n. On each iteration we get the current house value and the $i-1$ and $i-2$ value from *curr_max_value* each in $O(1)$ time. We call max on the formula of these as outlined above and so is $O(1)$ overall. We can conclude our for loop hence takes $O(N)$ time. Finally we can calculate the overall time complexity:

$$O(N) + O(1) + O(1) + O(1) + O(1) + O(1) + O(N)$$

$$= O(N)$$

2. **Find the subset $S \subset V$ that maximises the sum of node values, given that no two nodes in S can be connected by an edge**

   (a) **Design a dynamic programming algorithm to determine the value of the optimal choice of houses. First derive and justify a recurrence relation and your base cases, then turn it into a bottom-up solution. A more efficient algorithm gives more points.**

   To determine the optimal choice of houses, we must first derive a recurrence relation. We can define a function $OPT(i)$ as the value of the optimal choice of houses in our sequence up to and including that of the i^{th} house, where the value of the i^{th} house being v_i. When we reach the i^{th} house, there are two options, to add this house to our subset of houses with maximum combined value, or not. This is best explained by the following equation:

   $$OPT(i) = max((v_i + OPT[\text{left}] + OPT[\text{right}]), (OPT[\text{left's children}] + OPT[\text{right's children}]))$$

   This is because no two nodes in S can be connected by an edge; such that we can either pick the optimum of the children, or the optimum of the grand children plus current house i.

   From this we can now form a bottom-up algorithm that computes the subset of houses with the maximum combined value. The order of our dynamic program will run as follows:

   i. Fill a list of nodes from the input to represent our full binary tree of houses
   ii. Initialise two arrays to hold our optimal include and exclude values. Include will keep track of the optimum value of the current node i and its grand children. Exclude will keep track of the optimum value of the children.
   iii. Iterate through the list of nodes starting at the end going backwards toward the start. In relation to the tree structure, this means starting at the bottom right-most node and ending at the top root node.
   iv. If current node i has not got children or grand children (i.e. if the child node index formula runs out of bounds), then only fill the include max value with the current value v_i (as there are no grand children) and leave the exclude value as 0 as there are no children.
   v. Once the iteration reaches nodes with children and/or grand children, calculate the maximum exclude value by adding the maximum of the left and right children as well as the maximum include value by adding the value of the current node i plus the maximum of the left and right grand children.
   vi. Once the iteration has completed to the top of the tree, take the max value from the include and exclude array at index 0 and return that as the result. This will be the value of the optimal choice of houses.

(b) **Argue the correctness of your algorithm.**

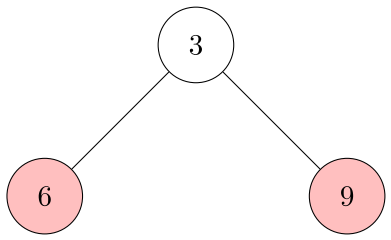
i. Case where $n = 0$: If there are no houses to visit, then there is 0 value and hence the final value is 0 as there can be no other houses to add value.

ii. Case where $n = 1$: With only one node available, the root is the start node, end node and the only element. Becuase of this, the weight of n + grand children is just n as there are no grand children. Our exclude list will be 0 as there are no children to be added to it. Returning the highest sum of this subset will be the highest value.

iii. Case where $n = 2$: If there are only two houses, then finding the optimum value will simply be the maximum value between these two houses as selecting none would give no value and both can't be selected as they cannot be adjacent. Hence the max value of the two houses where $n = 2$ is the optimum value.

Proof for n

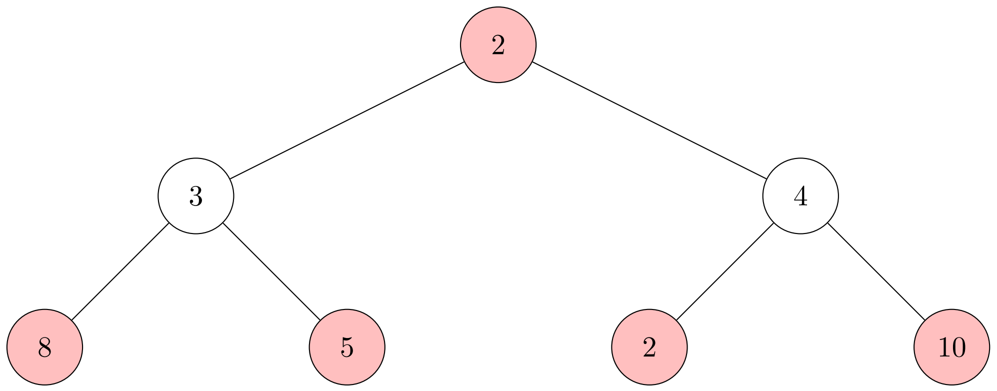
iv. Case where $n = 3$:



- $node2 = \mathbf{9} \,||\, 0$
- $node1 = \mathbf{6} \,|\, 0$
- $node0 = 3 \,||\, \mathbf{6+9}$

Node 0 is the root and node 1 and 2 are node 0's children. Nodes 1 and 2 will have an include summation of 6 and 9 respectively, with an exclude summation of 0 as neither of them have children. Node 0 will have an include summation of itself and the grand children (of which there are none) hence totalling 3. Node 0 will have an exclude of the optimum of the children which is 15. Hence, the optimum sum of the nodes in this tree will be 15 which is from our final subset of maximum value of houses such that no two nodes can be connected by an edge.

v. Case where $n = 7$:



- $node6$ = **10** || 0
- $node5$ = **2** || 0
- $node4$ = **5** || 0
- $node3$ = **8** || 0
- $node2$ = 4 || **2+10**
- $node1$ = 3 || **8+5**
- $node0$ = **2+(10+2+5+8)** || $3 + 4$

Here, we see the last 4 nodes *node6, node5, node4, node3* with values 10, 2, 5 and 8 respectively in the list are leaf nodes (no children). The 2 prior to them, *node2* and *node1* with values 4 and 3 respectively, are the parents of *(node6 & node5)* and *(node4 & node3)* respectively. Finally *node0* with value 2 is the root node with children *node1 & node2*. For nodes 6,5,4 and 3, they will all pick each other as the optimum as they have no children nor grand children to compare to. Stepping into *node2* with value 4, it will compare its include of 4+0 (as it has no grandchildren) and its exclude of *(node6 & node5)* with values 10 and 2 respectively. As $10 + 2 > 4$, the exclude will be chosen as optimal. Same with *node1* where its exclude of values 5 and 8 will sum to be greater than its include value of itself and so the exclude will be chosen. Finally at *node0*, it will compare its include (the value of itself, 2, and the optimum of its grand children, $8+5+2+10 = 25$) and its exclude, the optimum of its children, $2+4 = 6$. As $25 > 5$, the include will be chosen. As this is the root node, the algorithm exits returning what it has proven to be the value of the subset of the optimum choices of houses with maximum value.

(c) **Prove an upper bound on the time complexity of your algorithm.**
We first read in a sequence of house values that are appended to a list. Appending to a list takes $O(1)$ time and is repeated n times. This means $O(N)$ overall time for the data input. We then begin our algorithm. The algorithm is contained in a for loop that is iterates N times. Contained in this for loop are several instances of accessing values from a list in $O(1)$ time calling the max function on these also in $O(1)$ time. Thus, the for loop takes $O(N)$ time overall. Finally, to calculate the solution value, we retrieve two values from a list in $O(1)$ time and then call max on these values, also in $O(1)$ time. Finally we can calculate the overall time complexity:

$$O(N) + O(N) + O(1)$$

$$= O(N)$$