CAB320 Open Pit Mining Assignment

N10448888 Angus Macdonald
N10689753 Cassandra Nolan
N10112375 Riku Oya

Introduction

This report introduces two methods of finding the best dig plan possible for performing open-pit mining on a mine that is simplified to a 2D or 3D array. The 2D array representation of the mine has the x and z components, where the x components are parallel to the surface and the z components represent the direction of gravity. For the 3D representation, y components are added to the 2D representation, which also goes parallel to the surface.

The first algorithm that searches the mine is the Dynamic Programming method (DP). DP uses recursion to explore possible child states of the current state, and takes the action that finds the best payoff. Caching is also applied so that the recursive function does recomputed values, and can simply call repeated inputs.

The second algorithm was the Branch and Bound method (BB). BB is a tree search method, but prunes sub-trees that will not return higher payoff values. This is done by calculating an upper bound of a sub-tree while the problem constraints are relaxed. Utilising this upper bound, the algorithm can ignore branches that do not have the potential to find the best payoff. This results in a faster search as the program has been optimised.

Branch and Bound Pseudo Code

Input	A mine instance
Output	The best final payoff, best final state, and best action list from input mine instance

Algorithm

```
search_bb_dig_plan(mine)
    Best state is equal to the initial mine state
    Payoff initialisation as the payoff of the initial mine state
    Initialise empty frontier
    Add initial state to frontier
    Initialize set for explored variables
    While frontier is not empty:
        Remove and return last state added to frontier
        Add current state to explored states as hashed tuple
        For every possible action from current state:
            This_state equals the result child state from current state when possible action is
            performed
            Boolean_0 equals to boolean if This_state is not one of the explored states
            Boolean_1 equals to boolean if This_state is not in the frontier
            If Boolean_0 and Boolean_1:
                Add the child state to the frontier
                If the payoff of this child state is greater than the current best
                payoff:
                    Make best payoff and best state now equal to this child state and
                    this child payoff
    Return best state and best payoff
```

Dynamic Programming Pseudo Code

Input	A mine instance
Output	The best final payoff, best final state, and best action list from input mine instance

Algorithm

```
search_dp_dig_plan(mine)
    Initialise set for explored variables
    Let start state equal the initial state of the mine

    *dp state function(state)
        Convert state to tuple, hash the result and make it the current state
        Set payoff variable to equal to the payoff of the given state
        Best final state is equal to given state
        If state is not explored yet:
            For all possible actions from the given state:
                Child state equals the state when the possible action is done on the
                given state
                Child payoff , child final state equals output of *dp state function(child
                state)
                If child payoff is greater than payoff of given state:
                    Best payoff equals child payoff
                    Best final state equals child final state
            Add state to list of explored states
        Return best payoff, best final state

    Final equals output of *dp state function(start state)
    Final_payoff equals the first value in the returned *dp state function(start state)
    Final_state equals the second value in the returned *dp state function(start state)
    Seq equals list of actions that is required from start state to best final state using the
    find_action_sequence(Start_state, Final_State)

    Return Final_payoff, Seq, Final_state
```

Testing Methodology

Simple mine arrays, where the best final states and payoff values were clear, were used to test the functionality. Since 2D arrays were simpler than 3D arrays, functions were written for 2D arrays first, then were improved for 3D arrays. Functions were fixed by testing and comparing the output with the expected results.

A given sanity check file was also used for testing. The given file would run both DP and BB codes on 2D and 3D arrays. It would also print out the expected outputs with actual outputs of the code. It also returned run times of the functions to check code quality. Functionality was prioritised for this assignment, but code quality was improved whenever possible.

Please find the attached comparison table on page over.

Testing Method Comparison Table

	2D	**3D**
Input	*some_sanity_tests.py *some_2d_underground_1 numpy array*	*some_sanity_tests.py *some_3d_underground_1 numpy array*
Dynamic Programming Function		
Expected Output	**Best payoff :** 2.9570000000000003 **Best final state :** (3, 2, 3, 4, 3) **Best action list:** [(0,), (1,), (2,), (3,), (4,), (0,), (1,), (2,), (3,), (4,), (0,), (2,), (3,), (4,), (3,)]	**Best payoff :** 5.713 **Best final state :** ((2, 1, 1, 1), (1, 1, 0, 1), (0, 0, 0, 1)) **Best action list:** ((0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (0, 0), (1, 3), (2, 3))
Our Output	**Best payoff** : 2.9570000000000003 **Best final state** : (3.0, 2.0, 3.0, 4.0, 3.0) **Best action list** : (3, 0, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4)	**Best payoff :** 5.713 **Best final state :** ((2, 1, 1, 1), (1, 1, 0, 1), (0, 0, 0, 1)) **Best action list :** ((0, 0), (2, 3), (1, 3), (1, 1), (1, 0), (0, 3), (0, 2), (0, 1), (0, 0))
Expected Run Time	0.015289068222045898 seconds	6.34877872467041 seconds
Our Run Time	0.05596637725830078 seconds	39.13559365272522 seconds
Branch and Bound Function		
Expected Output	**Best payoff :** 2.9570000000000003 **Best final state :** (3, 2, 3, 4, 3) **Best action list:** [(0,), (1,), (2,), (3,), (4,), (0,), (1,), (2,), (3,), (4,), (0,), (2,), (3,), (4,), (3,)]	**Best payoff :** 5.713 **Best final state :** ((2, 1, 1, 1), (1, 1, 0, 1), (0, 0, 0, 1)) **Best action list** : [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 3), (2, 3), (0, 0)]
Our Output	**Best payoff :** 2.9570000000000003 **Best final state :** (3.0, 2.0, 3.0, 4.0, 3.0) **Best action list:** (3, 0, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4)	**Best payoff :** 5.713 **Best final state :** ((2, 1, 1, 1), (1, 1, 0, 1), (0, 0, 0, 1)) **Best action list:** ((0, 0), (2, 3), (1, 3), (1, 1), (1, 0), (0, 3), (0, 2), (0, 1), (0, 0))
Expected Run Time	0.027152538299560547 seconds	3.002690076828003 seconds
Our Run Time	0.04997062683105469 seconds	38.15501928329468 seconds

Performance and Limitations

Performance

As seen in the previous testing method comparison table, our Dynamic Program function is running up to approximately 6.5 times slower than the sanity_test.py. This is due to the use of loops within our function, most notably in the *actions* function, in which the function creates *n* copies of *n* length, and iterates over the copies *n* amount of times, where *n* is equal to the amount of elements within a state (surface cells). We have the opportunity to remove this second *n* iterable with better generation or list comprehension in the creation of *n* copies. This function also requires *n**2* blocks of memory for every state passed, where the scaling of our x, or x,y coordinates will increase the memory usage greatly.

Within the *is_dangerous* function for a 2-dimensional state passed, we have a double nested loop that iterates over each element with a 2-dimensional surface plane, and runs a comparison with every neighbour. Although the function does return a dangerous state if any exceeds the dig_tolerance, the resulting doubled nested loop ultimately creates longer performance times.

Limitations

The *find_action_sequence* also does not account for a cost of moving from C1 to C2 (if such cost were to occur) in its handling of dig_tolerance constraints. Although this project did not account for a *moving* cost before digging, such implementation would create large movement costs in our function.

Within the *is_dangerous* functionality, we implemented functionality to see if a state exceeds the depth of the mine. This functionality was not required, and would create issues with a mine of an unknown depth, or a mine with a growing depth.

The *Dynamic Programming* recursive function has limitations within the functionality of choices of its children. Due to the nature of list comprehension, it will follow the first child generated of possible actions, creating a similar functionality of a depth-first search. This functionality ultimately leads the program to a state where all columns have been mined, before returning back up and finding other pathways first. Without the implementation of an *explored* set, this function would not be optimal or complete, as it would find recurring states and explore the children of that state again.