Using dafny to prove the upper-bound time complexity of solving Travelling Salesman Problem by A^* Search

Aditya[20925808] and Aditya Sharma[20905833]

University of Waterloo, Waterloo ON N2L 3G1, Canada

Abstract. Dafny is a language that is commonly used for static program verification, and ensuring the functional correctness of software programs. Dafny converts the pre-conditions and post-conditions provided into Hoare triples, and then uses the Z3 SMT Solver to construct required proofs. The evolution of formal verification through dafny in the last decade has aided in carrying out proofs of functional correctness of programs computationally, instead of written proofs using pen and paper [1]. This paper tries to specify, and prove the upper bound of the time complexity of the A* search implementation of the Travelling Salesman Problem(TSP) through the use of dafny.

Keywords: Dafny · Formal Verification · Travelling Salesman Problem.

1 Introduction

The Travelling Salesman problem or TSP involves finding the shortest path that visits n specified locations, starting and ending at the same place and visiting the other n-1 destinations exactly once. It belongs to the NP Set of problems and is known to be NP-hard [4]. It is one of the most popular as well as one of the most researched problems in the NP Set of problems, and is often used as an introduction to the world of computational complexity. NP is defined as the set of all decision problems that can be verified in polynomial time by a deterministic Turing machine. Polynomial time is often used as the definition of a quick or fast algorithm. Regardless of it's popularity, TSP still remains hard to solve with most deterministic solutions to the problem having an exponential time complexity [10]. Due to the significance of TSP in real-world applications, researchers have developed a multitude of heuristic algorithms that can provide non-optimal solutions that still work for specific requirements.

Heuristic Search Algorithms are often used for large problems that contain a huge number of states, or nodes in a graph, and have problem specific information that can be modelled into a mathematical function that assigns estimate values to all the edges between the nodes, or states, of the graph. This heuristic function can be further used to prune the non-optimal paths, based on the calculated heuristic values of the multiple different paths that exist from the initial state to the goal state.

A* Search is a complete heuristic search algorithm that always yields an optimal, or least cost, solution to the problem. It optimizes the Depth First Search(DFS) algorithm by pruning the longer paths and only searching for the shorter paths. It does so by adding a heuristic function that acts as a guess to what the best path would be. Based on the problem, A* search can use the Manhattan distance as it's Heuristic function or even use the Euclidean distance between nodes and the goal node as a measure. Since A* Search is a complete algorithm, it finds and stores all possible solutions from the initial state to the goal state [8]. Storing all paths and solutions makes A* Search a slow algorithm with a high upper-bound for the space complexity.

Modelling TSP. As a mathematical model, the TSP can be viewed as a Directed Graph G = (V,A), where V are the vertices or nodes of the graph that specify different locations and

$$c_{ij}, (i,j) \in A \tag{1}$$

is the cost of an edge between two vertices. The cost signifies the distance between two cities, and if also often referred to as the literal dollar cost of the edge [8]. The goal of the TSP can be written in mathematical terms as follows:

$$Minimize \sum_{i,j \in A} c_{ij} x_{i,j} \tag{2}$$

In order to solve TSP, four main constraints need to be satisfied. These constraints are listed below:

$$\sum_{i \in V \setminus \{j\}} x_{i,j} = 1, j \in V \tag{3}$$

$$\sum_{j \in V \setminus \{i\}} x_{i,j} = 1, i \in V \tag{4}$$

$$\sum_{i \in S} \sum_{j \notin S} x_{i,j} \geq 1, S \subseteq V, \mid S \mid > 2 \tag{5}$$

$$x_{i,j} = \{0,1\}, (i,j) \in A \tag{6}$$

The first two constraints specify that every single vertex 'i,j' should be connected to exactly one other vertex 'j,i'. This is necessary because optimal solutions should not have cycles in the path, considering negative weights are illegal. However, these two constraints are not enough to ensure that the solution does not have a cycle. The second last constraint ensures that the solution is not a collection of smaller routes, which could be thought of as sub-tours of the graph. In other words, it makes sure that the model always outputs a singular path connecting all the vertices. Finally, the last constraint ensures that the values of the variable x is set correctly and is one when vertices i and j are connected and 0 otherwise, for x_{ij} [8].

2 Related Work

Much work has already been conducted in the area of computationally constructing mathematical proofs for verifying the upper bounds of time complexities. Morshtein et al. state and construct a proof to verify the upper-bound time complexity of the Binary Search algorithm, with the use of dafny [1]. The proof first builds a function in dafny to specify the logarithmic complexity class. It then specifies $O(log_2 n)$ in the form of predicates and a lemma. Once this has been done, the proof goes on to build a fully verified implementation of the binary search algorithm. Once implementation is complete, the proof creates a ghost variable in order to store the time complexity value for the algorithm.The ghost variable "counts the number of dominant operations performed by the algorithm, and it is added to the algorithm's post-condition" [1]. The ghost entries in dafny are a tool used for verification and are not executable parts of the code. Once all this is done, the proof goes on to find a suitable logarithmic function f(n) and show that $t \leq f(n)$.

Further work has also been done on encoding induction proofs in dafny by first decomposing sophisticated proof obligations into proofs of induction, and then encoding the gathered proof dependencies in dafny, by Jiang et al. [3]. Some work has also been done in verification for the correctness of more complex algorithms such as The Davis-Putnam-Logemann-Loveland(DPLL) Algorithm [7] and the Merkle Tree Algorithm [5]. Although algorithm verification differs from verifying the upper-bound of time complexities of algorithms, some concepts from proof of correctness on dafny can also be used to prove time complexities.

3 Defining the Heuristic Function for TSP

For any A* Search implementation to a problem, there are multiple admissible as well as monotone heuristic functions that can be picked and used to optimize the Depth-First Search(DFS) algorithm. For general grid-world problems, Manhattan or Euclidean distance is often used as a heuristic function. However, to solve the TSP problem with P time complexity, a more sophisticated heuristic function is required. As TSP is NP-hard, the implementation needs to reduce the non-polynomial time complexity into a polynomial one.

Initially, a heuristic function based on MST was tried, where the function was the sum of: the cost of the lowest cost edge between the current city and some city in the list of unvisited cities, the sum of all edges on any MST for the univsited cities, and the cost of the lowest cost edge between some unvisted city and the starting city. This heuristic function, can be found in "*heuristic.dfy*" on the github for this project. However, this function turned out to be hard for carrying out the time-complexity proof. Since the complexity involved in the heuristic stopped the progress of completing the implementation, another admissible heuristic was researched and constructed. The new heursitic function was be based on computing the Minimum Spanning Trees(MST) at each node of the graph. An algorithm for this heuristic to compute MST would start by first finding the shortest arc in the network, while breaking ties arbitrarily and highlighting the arc as well as the nodes connected to it. After this, it will pick the next shortest arc that does not form a cycle with the already highlighted arcs, and highlight the picked arc as well as the connected nodes like last time. Until all arcs are highlighted, the last step would be repeated, and the algorithm will eventually terminate once that is done. There are multiple different algorithms that can be used to compute MSTs but this implementation to compute MSTs is called the Prim's Algorithm for solving MST.

Once the MST is constructed, the MST based heuristic would work by picking the initial state or the root to be an arbitrary vertex V in the graph G. Then, all the vertices would be traversed starting from the root node. Fig. 1 depicts how the process of computing the heuristic function would take place. The time

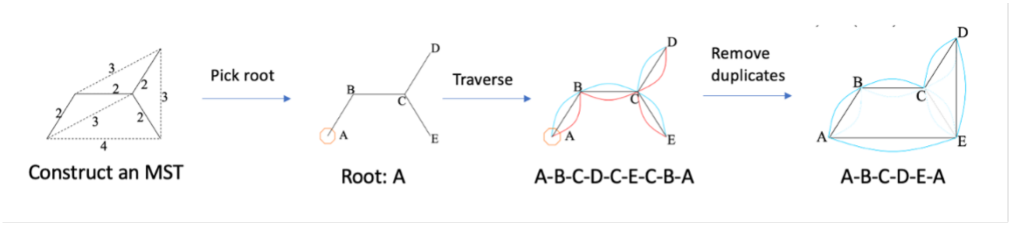


Fig. 1. Representation of the MST based heuristic being used

complexity of the above heuristic function, would define the overall time complexity for the TSP, since the computation of MST's will have a nested for loop spanning through the size of vertices V, leading to a quadratic function for the upper bound. Let L^{MST} denote the length of the tour that will be generated by the heuristic function that is implemented. Then,

$$L^{MST} \leq 2W^{*} \leq 2L^{*} \tag{7}$$

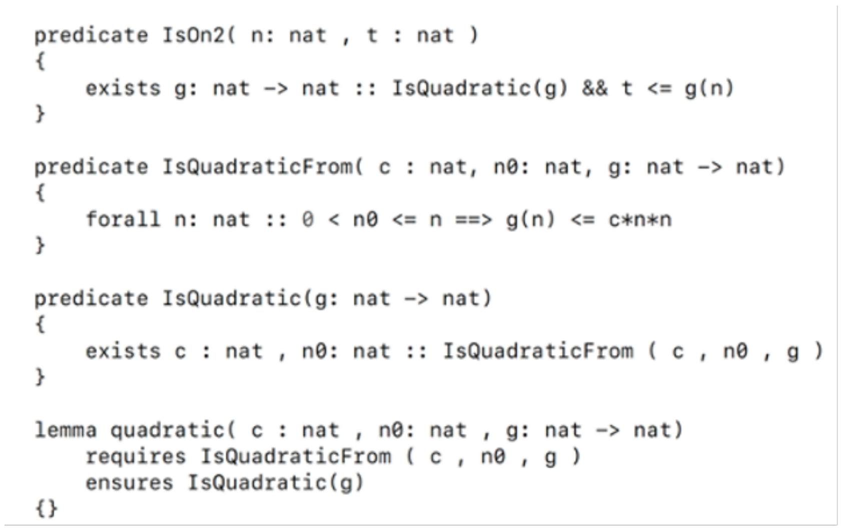
Where W^{*} denotes the length of the MST [11]. Since the MST can be solved in $O(n^2)$, it forms the lower bound for the TSP.

4 Quadratic Complexity Specification in Dafny

In order to complete the proof in dafny, the Big-O notation needs to be defined. The Big-O of an algorithm in computational theory, is defined as the function that upper-bounds the run-time of an algorithm. For the MST heuristic that is being implemented for the TSP problem in question, the function f belongs to $O(n^2)$ or the quadratic function. Thus, to implement the proof through dafny, the definition of the quadratic function will first need to be defined mathematically in the form of predicates. In dafny, a predicate is defined as function that returns a value of type Boolean. In the dafny syntax, it can be defined by replacing the function keyword with the predicate keyword [9]. In First Order Logic(FOL), the quadratic function is defined as [2]:

$$\exists i > 0 \ \&\& \ \exists x > 0 \ s.t. \ \forall n. \ n \geq x \ \Rightarrow \ f(n) \leq i * n^2 \tag{8}$$

Using the equation above, $O(n^2)$ can be defined in dafny [2]. Fig. 2 shows the required predicates and the lemma that would be needed to define the quadratic big-O complexity.



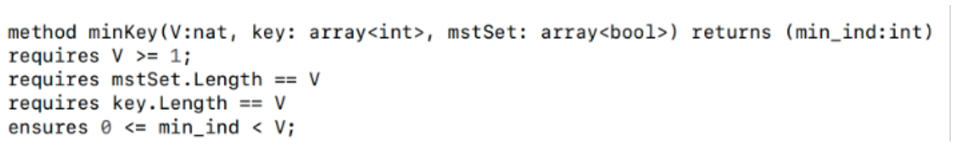
```
predicate IsOn2( n: nat , t : nat )
{
    exists g: nat -> nat :: IsQuadratic(g) && t <= g(n)
}

predicate IsQuadraticFrom( c : nat, n0: nat, g: nat -> nat)
{
    forall n: nat :: 0 < n0 <= n ==> g(n) <= c*n*n
}

predicate IsQuadratic(g: nat -> nat)
{
    exists c : nat , n0: nat :: IsQuadraticFrom ( c , n0 , g )
}

lemma quadratic( c : nat , n0: nat , g: nat -> nat)
    requires IsQuadraticFrom ( c , n0 , g )
    ensures IsQuadratic(g)
{}
```

Fig. 2. The predicates and lemma that define $O(n^2)$ in dafny

The three predicates that are needed for the proof are built in a modular fashion, and can be compared to proving something through induction while carrying out a proof mathematically. The first predicate in Fig. 2, is called *IsOn2*, and is required to confirm that the complexity function satisfies the quadratic function defined in terms of the FOL expression above in (8).The predicate could be thought of as the specification of the post-condition for any algorithm whose runtime complexity would be $O(n^2)$ [2]. As can be observed, the predicate *IsOn2* calls another predicate that is specified in Fig. 2, called *IsQuadratic*. This predicate is used to verify and specify whether i and x from expression (8) exist that can satisfy the predicate *IsQuadraticFrom*. The *IsQuadraticFrom* then determines whether the function under consideration is tightly upper bounded by the expression $i * n^2$ from (8), for every n that is larger than x. Once all these predicates are specified to satisfy the entire expression from (8), the lemma *quadratic* in Fig. 2 can be defined in order to combine predicates *IsQuadraticFrom* and *IsQuadratic* as the pre and post conditions respectively.

5 Implementing A^* Search Algorithm for TSP in Dafny

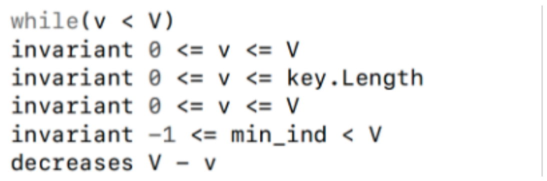
The following section shows a verified implementation of travelling salesman problem using Prim's minimum spanning tree algorithm as a heuristic using Dafny. This section also discusses the predicates used to prove the complexity of the problem is $O(n^2)$. Fig. 3 below depicts the pre-conditions and the post-condition for the *minKey* method in dafny. The function in Fig. 3 is a utility



```
method minKey(V:nat, key: array<int>, mstSet: array<bool>) returns (min_ind:int)
requires V >= 1;
requires mstSet.Length == V
requires key.Length == V
ensures 0 <= min_ind < V;
```

Fig. 3. *minKey* method signature

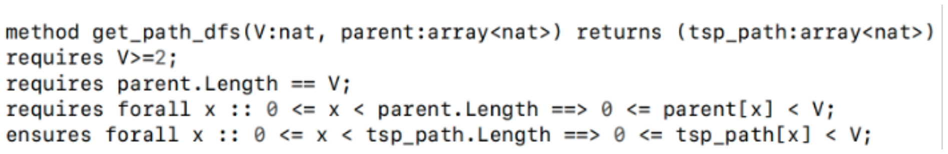
method used by Prim's algorithm that returns *min_ind* the minimum key from the "key" array which is a list of integers. This method also accepts a parameter *mstSet* which is a Boolean list with visited indices as true and rest as false. This method checks the returned value is not less than 0, and not greater or equal to the maximum number of vertices. The method iterates over the key list and maintains all the invariants. Fig. 4 below lists all the invariants for the loop. Another utility method used as part of the heuristic is the *get_path_dfs* method



```
while(v < V)
invariant 0 <= v <= V
invariant 0 <= v <= key.Length
invariant 0 <= v <= V
invariant -1 <= min_ind < V
decreases V - v
```

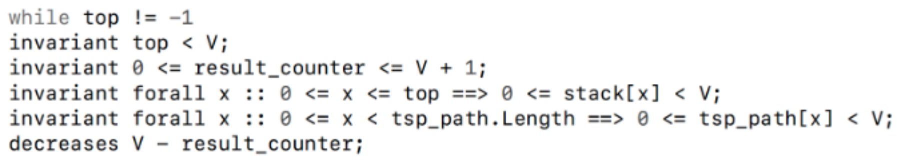
Fig. 4. *minKey* method invariants for the while loop

that, given a list of vertices denoting parents of nodes, returns a path by using depth first search. Fig. 5 shows the pre-conditions and the post-conditions for the *get_path_dfs* method. This method carries out the implementation of steps 2 to 4 that are described in Fig. 1. There are two inputs to the function, first is the total number of vertices and second is the list of nodes denoting parents such that $parent[i]$ is the parent node of the vertex i in the minimum spanning tree. The return value from this method is the actual path calculated using depth first search on a graph created using the parent list. This path is returned as the solution to the travelling salesman problem. One of the input to the above method is the parent list which is the main component of the minimum spanning tree.



```
method get_path_dfs(V:nat, parent:array<nat>) returns (tsp_path:array<nat>)
requires V>=2;
requires parent.Length == V;
requires forall x :: 0 <= x < parent.Length ==> 0 <= parent[x] < V;
ensures forall x :: 0 <= x < tsp_path.Length ==> 0 <= tsp_path[x] < V;
```

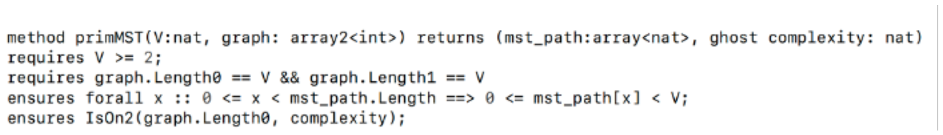
Fig. 5. *get_path_dfs* method signature



```
while top != -1
invariant top < V;
invariant 0 <= result_counter <= V + 1;
invariant forall x :: 0 <= x <= top ==> 0 <= stack[x] < V;
invariant forall x :: 0 <= x < tsp_path.Length ==> 0 <= tsp_path[x] < V;
decreases V - result_counter;
```

Fig. 6. get_path_dfs method invariants

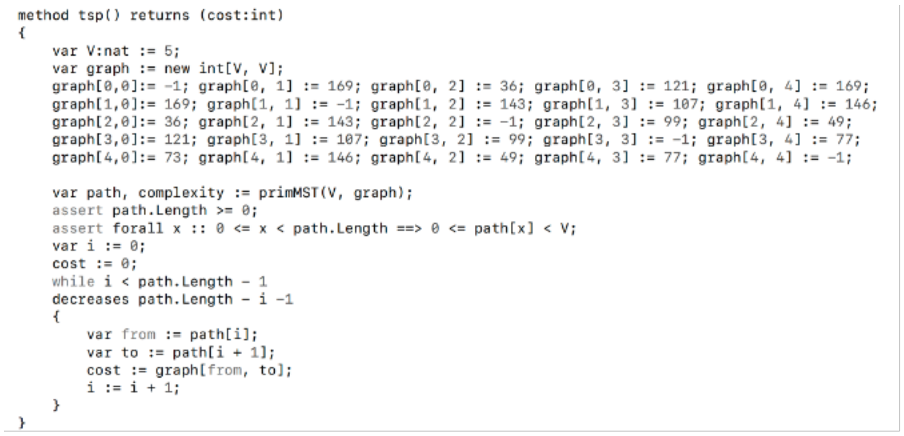
Using the *minKey* utility function, below is the implementation of Prim's minimum spanning tree method that finds a minimum spanning tree in the input graph matrix and a natural number denoting the number of vertices. It generates the parent list which is passed as an input to the get_path_dfs method.



```
method primMST(V:nat, graph: array2<int>) returns (mst_path:array<nat>, ghost complexity: nat)
requires V >= 2;
requires graph.Length0 == V && graph.Length1 == V
ensures forall x :: 0 <= x < mst_path.Length ==> 0 <= mst_path[x] < V;
ensures IsOn2(graph.Length0, complexity);
```

Fig. 7. *primMST* method signature

The input to this method are two parameters, one is the number of vertices in the graph and second is a two dimensional array that is a matrix representation of the original graph such that $graph[i][j] = x$ implies that node i is at x distance from node j. The method returns two values - the resultant path for the TSP and a ghost variable that is used to test the complexity of the variable. This method ensures that the complexity of this function is $O(n^2)$ by calling *IsOn2* function on complexity and ensuring that it is verified by dafny. The working of *IsOn2* function is described in section 4. The parent list is calculated using Prim's algorithm and is passed to get_path_dfs method. Rest is the initialization code is dafny to setup a graph, and calculate cost of the path returned from the primMST method. Fig. 8 below shows the dafny program *tsp.dfy* which is responsible for initializing the graph for TSP and calling the methods that carry out the cost and complexity calculations.



```
method tsp() returns (cost:int)
{
    var V:nat := 5;
    var graph := new int[V, V];
    graph[0,0]:= -1; graph[0, 1] := 169; graph[0, 2] := 36; graph[0, 3] := 121; graph[0, 4] := 169;
    graph[1,0]:= 169; graph[1, 1] := -1; graph[1, 2] := 143; graph[1, 3] := 107; graph[1, 4] := 146;
    graph[2,0]:= 36; graph[2, 1] := 143; graph[2, 2] := -1; graph[2, 3] := 99; graph[2, 4] := 49;
    graph[3,0]:= 121; graph[3, 1] := 107; graph[3, 2] := 99; graph[3, 3] := -1; graph[3, 4] := 77;
    graph[4,0]:= 73; graph[4, 1] := 146; graph[4, 2] := 49; graph[4, 3] := 77; graph[4, 4] := -1;

    var path, complexity := primMST(V, graph);
    assert path.Length >= 0;
    assert forall x :: 0 <= x < path.Length ==> 0 <= path[x] < V;
    var i := 0;
    cost := 0;
    while i < path.Length - 1
    decreases path.Length - i -1
    {
        var from := path[i];
        var to := path[i + 1];
        cost := graph[from, to];
        i := i + 1;
    }
}
```

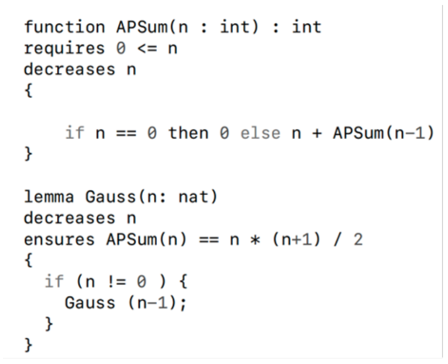
Fig. 8. Initialization of the graph and cost calculations

6 Derivation of the Tight Upper-Bound Function

In order to keep track of the time complexity in dafny, a ghost variable needs to be created and returned. In dafny, a ghost variable is a type of variable that stores values that do not pertain to the execution of the program, but rather store values that are required for functional specification. To declare a ghost variable in dafny, the keyword *ghost* is used before the declaration [9]. This ghost variable will store the the number of operations that occur during the creation of the MST, which leads to the time complexity of $O(n^2)$.

In section 4, it has already been shown that a function is quadratic if it satisfies the written predicates and lemmas. The previous section has also shown mathematically that the current implementation of the TSP will have a time complexity of $O(n^2)$. Now it's time to prove that the time complexity of the implementation is quadratic, formally using predicates and lemmas in dafny. The two important lemmas that achieve this are called *quadraticCalcLemma* and *On2Proof* respectively. These lemmas build on other functions and lemmas and guarantee that the ghost variable with the time-complexity function is a quadratic function.

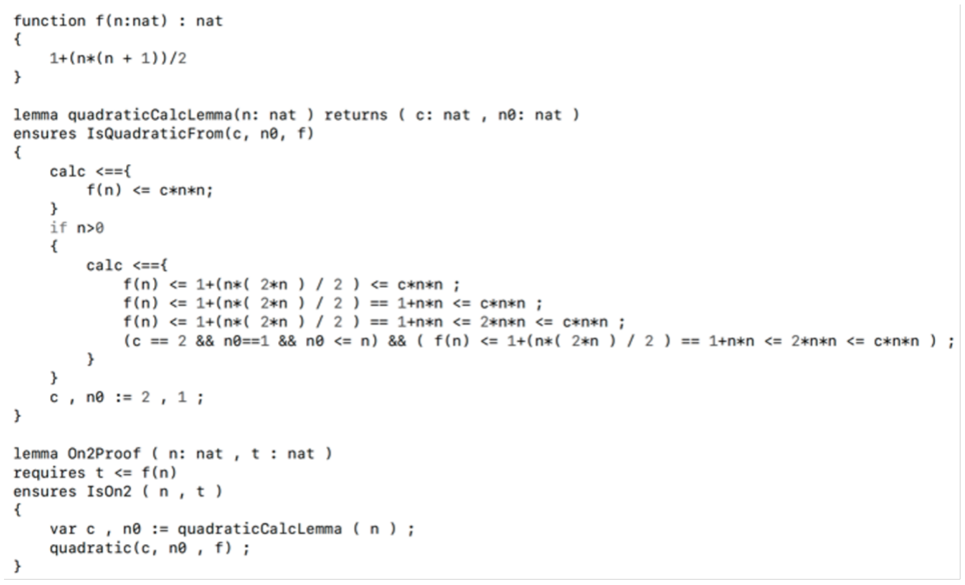
Fig. 9 and Fig. 10 show all the necessary functions and lemmas that will be required to carry out the proof to completion. The lemma *quadraticCalcLemma* returns two natural numbers c and n_0. These values are nothing but i and x in the expression (8). In other words, this lemma requires that the values of c and n_0 satisfy the expression (8). Dafny's calc construct [6] is used to implement *quadraticCalcLemma*. In other words, the lemma ensures that proper values of c, n_0, and function f exists. Function f is a quadratic function equal to $1 + (n * (n + 1))/2$. As can be observed, the term n^2 would make function f quadratic, and further ensure that the conditions of the other lemma *On2Proof* hold. Once this ensuring is done, *On2Proof* can be used to finally prove that the implementations run-time is quadratic. Another thing that is required for



```
function APSum(n : int) : int
requires 0 <= n
decreases n
{

    if n == 0 then 0 else n + APSum(n-1)
}

lemma Gauss(n: nat)
decreases n
ensures APSum(n) == n * (n+1) / 2
{
  if (n != 0 ) {
    Gauss (n-1);
  }
}
```

Fig. 9. the *Gauss* lemma and the *APSum* function

On2Proof to hold is that inputs are not empty as well as the step counter value does not exceed the value of *f*. However, the implementation of *primMST* has already been ensured to not contain empty inputs or a value of step counter higher than the function value. Finally, the post-condition of *IsOn2* from Fig. 2 will ensure that the sum of all required terms is a quadratic equation. Once this is achieved, the proof under consideration is considered complete.



```
function f(n:nat) : nat
{
    1+(n*(n + 1))/2
}

lemma quadraticCalcLemma(n: nat ) returns ( c: nat , n0: nat )
ensures IsQuadraticFrom(c, n0, f)
{
    calc <=={
        f(n) <= c*n*n;
    }
    if n>0
    {
        calc <=={
            f(n) <= 1+(n*( 2*n ) / 2 ) <= c*n*n ;
            f(n) <= 1+(n*( 2*n ) / 2 ) == 1+n*n <= c*n*n ;
            f(n) <= 1+(n*( 2*n ) / 2 ) == 1+n*n <= 2*n*n <= c*n*n ;
            (c == 2 && n0==1 && n0 <= n) && ( f(n) <= 1+(n*( 2*n ) / 2 ) == 1+n*n <= 2*n*n <= c*n*n ) ;
        }
    }
    c , n0 := 2 , 1 ;
}

lemma On2Proof ( n: nat , t : nat )
requires t <= f(n)
ensures IsOn2 ( n , t )
{
    var c , n0 := quadraticCalcLemma ( n ) ;
    quadratic(c, n0 , f) ;
}
```

Fig. 10. The function *f*, *quadraticCalcLemma*, and On2Proof lemma

7 Conclusion and Future Work

This paper was written with the main aim of presenting the methods that can be used for the calculation, verification, and information retrieval of the non-functional characteristics of an algorithm which are an important part of the design specification. The most important non-functional requirement for any algorithm that is used in day-to-day life is its worst-case asymptotic time complexity or run-time. Proving the big-O time complexities has been a tedious task for generations. With the advancements in static-verification languages, it has now become possible to non-manually prove such complexities for a plethora of popular algorithms. The paper defined a function in dafny that can be used by any algorithm with a quadratic run-time complexity. Furthermore, it proved the functionality of how such a proof for the upper bound of time complexity can be carried out by implementing the A^* search algorithm for TSP, and showing that the tight upper bound would be quadratic.

Carrying out this proof for the time complexity, also aided in showing the potential of using dafny for carrying out such proofs for the space complexity as well, which hasn't been explored in this paper yet. This is because further research needs to be done in order to understand how the space complexity can be correctly stored for variables that change the space complexity more than once inside a loop. Normal loop-invariants would not work for such a scenario as they only check if a condition holds before and after each iteration of a certain loop [2]. Moreover, this paper also only considers the worst-case time complexity or the big-O notation. Considering the average-case time complexity of an algorithm isn't covered as it would require a deeper understanding on the computation of probabilistic run-time complexities and proficiency in probability theory.

References

1. Morshtein, S., Ettinger, R. & Tyszberowicz, S.: Verifying Time Complexity of Binary Search using Dafny.Electronic Proceedings in Theoretical Computer Science. 338, 68–81 (2021). https://doi.org/10.48550/arXiv.2108.02966.
2. Morshtein, S.: Methods of Verifying Complexity Bounds of Algorithms using Dafny. Master's thesis, School of Computer Science The Academic College Tel-Aviv ... (2020).
3. Jiang, H., Li, Y., Tan, S. & Zhao, Y. Encoding Induction Proof in Dafny. *2021 International Symposium On Theoretical Aspects Of Software Engineering (TASE)*. pp. 95-102 (2021)
4. Kellerer, H., Pferschy, U. & Pisinger, D. Introduction to NP-Completeness of knapsack problems. *Knapsack Problems*. pp. 483-493 (2004)
5. Cassez, F. Verification of the Incremental Merkle Tree Algorithm with Dafny. *International Symposium On Formal Methods*. pp. 445-462 (2021)
6. Lucio, P. A tutorial on using dafny to construct verified software. *ArXiv Preprint ArXiv:1701.04481*. (2017)
7. Andrici, C. & Ciobâcă, Ş. Verifying the DPLL algorithm in Dafny. *ArXiv Preprint ArXiv:1909.01743*. (2019)

8. Rego, C., Gamboa, D., Glover, F. & Osterman, C. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal Of Operational Research.* **211**, 427-441 (2011)
9. Leino, K. Developing verified programs with Dafny. *2013 35th International Conference On Software Engineering (ICSE).* pp. 1488-1490 (2013)
10. Goyal, S. A survey on travelling salesman problem. *Midwest Instruction And Computing Symposium.* pp. 1-9 (2010)
11. Shen, Z.: The Traveling Salesman Problem, https://aswani.ieor.berkeley.edu/teaching/FA13/151/lecture_notes/ieor151_lec17.pdf, last accessed 2022/08/12.