**A logo with a lion and building

Description automatically generated with medium confidence**

POLITECNICO DI TORINO

DEPARTMENT OF CONTROL & COMPUTER ENGINEERING

*COMPUTATIONAL INTELLIGENCE*

**LAB I**

**SET COVERING USING A\* ALGORITHM**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Matricula** | **GitHub** | **Submission**  **Date** |
| Hossein Khodadadi | 313884 | [Link\_1](https://github.com/HOSSENkhodadadi/Computational_Intelligence/tree/main/Lab1) | 24/10/2023 |
| Abolfazl Javidian | 123456 | Link\_2 | 24/10/2023 |

Fall 2023/2024

1. **Report: Analysis of A\* Algorithm for Set-Covering Problem**

Build a new H, consider the old H (distance), what about the special sets what about the order of sets?

Don’t forget to mention the code

??

**1.1 Introduction:**

The set-covering problem is a classic optimization problem in which the goal is to identify the smallest possible collection of sets that covers all elements in a universe. It has various real-world applications, including scheduling, resource allocation, and logistics. In this report, we analyze an implementation of the A\* algorithm for solving the set-covering problem.

* 1. **Function Analysis:**

**goal\_check(state):** This function checks whether the current state satisfies the goal condition, ensuring that the union of sets selected covers all elements in the problem space.

**distance(state**): The distance function calculates the heuristic distance by determining the number of elements not covered by the selected sets. It serves as the heuristic for the A\* algorithm.

**g\_function(state):** The g\_function calculates the cost from the initial state to the current state by computing the number of sets selected.

**h\_function(state):** The h\_function estimates the cost from the current state to the goal state, utilizing the distance function as a heuristic to approximate the remaining number of uncovered elements.

**costFunction(state):** The costFunction function combines the g\_function and the h\_function to calculate the total cost, which is used to determine the priority in the priority queue of the A\* algorithm.

* 1. **Exploration of Complexity:**

The A\* algorithm provides an effective approach to solving the set-covering problem. By using heuristics and the priority queue, the algorithm efficiently explores the search space, prioritizing states with lower expected costs. However, the complexity of the A\* algorithm heavily depends on the characteristics of the problem instance, particularly the structure of the sets and the size of the problem space. In the case of a large number of sets and elements, the algorithm's performance might degrade due to the exponential growth of the search space. Additionally, the choice of heuristic plays a crucial role in balancing the trade-off between solution quality and computational efficiency.

* 1. **Conclusion:**

The A\* algorithm, combined with appropriate heuristics, presents a viable approach for solving the set-covering problem. It offers a balance between optimality and efficiency, making it suitable for a wide range of practical applications. However, the algorithm's performance can vary significantly based on the complexity and characteristics of the problem instance. Further exploration into the development of advanced heuristics and optimization techniques could enhance the algorithm's scalability and robustness for addressing larger and more complex set-covering instances.

1. **Report: Set Covering Depth First Recursive Algorithm with Reduction**

**2.1) Introduction**

The provided code is a Python implementation of an algorithm to solve the Set Cover problem. Set Cover is a well-known NP-hard optimization problem where you are given a universal set and a collection of subsets, and the goal is to find the minimum number of subsets required to cover the universal set.

**2.2) Code Flow**

Global Variable: node\_counter is a global variable used to count the number of nodes explored during the execution of the algorithm. It provides insight into the efficiency of the algorithm.

**2.3) find\_min\_set\_cover Function:**

This is the main function responsible for solving the set cover problem.

The function takes four parameters:

* universe: The set to be covered.
* subsets: A list of subsets.
* state: The current solution state (subset indices).
* subset\_indices: A list of indices for the subsets.

It uses a recursive approach to explore all possible combinations of subsets to cover the universe. The algorithm works by considering two possibilities: Including the first subset (chosen\_subset) in the solution and recursively solving the problem with the reduced universe and subsets.Not including the first subset and solving the problem with the original universe and the remaining subsets.The function returns the minimum cover found among these two possibilities.

**2.4) Constants and Data Generation:**

SETS is generated as a tuple of NUM\_SETS lists, where each list contains binary values (True/False) indicating whether an element is in the respective subset.subset\_indices is a list of indices corresponding to the subsets.

**2.5) Execution of the Algorithm:**

The universal set and subsets are generated from the constants and data.The find\_min\_set\_cover function is called with the universal set, subsets, an empty state, and subset indices to find the minimum set cover.

**2.6) Output and Reporting:**

If a valid set cover is found (result is not None), it prints the selected subsets and the number of subsets used for the minimum set cover.If no valid set cover is found, it prints "No valid set cover found." The code also prints the number of nodes explored (node\_counter), which represents the number of recursive calls made during the execution of the algorithm.

**2.7) Time Complexity Analysis**

The time complexity of this code is exponential, and it depends on the number of subsets and the size of the universal set. In the worst case, the algorithm explores all possible combinations of subsets, resulting in a time complexity of O(2^k), where 'k' is the number of subsets. The code is not optimized for large problem instances and can become very slow for a significant number of subsets.

Additionally, the global variable node\_counter is used to count the number of nodes explored. This variable provides insight into the algorithm's efficiency and can be used to analyze the actual number of recursive calls made during execution.

In summary, this code provides a basic implementation of the Set Cover problem-solving algorithm with a focus on clarity and correctness, but it is not optimized for large problem sizes due to its exponential time complexity.

1. **Mathematical Reduction of Set Covering to Linear Programming**

We begin by formulating the set cover problem as an Integer Linear Programming problem. Given an input (U, S1, . . . , Sn) of the set cover problem, we introduce a variable xi for every set Si, with the intended meaning that xi = 1 when Si is selected, and xi = 0 otherwise. We can express the set cover problem as the following integer linear program:

minimize

subject to

i:v∈Si xi ≥ 1 ∀v ∈ U

xi ≤ 1 ∀i ∈ {1, . . . , n}

xi ∈ N ∀i ∈ {1, . . . , n}

(1)

From which we derive the linear programming relaxation

minimize

subject to

i:v∈Si xi ≥ 1 ∀v ∈ U

xi ≤ 1 ∀i ∈ {1, . . . , n}

xi ≥ 0 ∀i ∈ {1, . . . , n}

More generally, it is interesting to consider a *weighted* version of set cover, in which we are given the set *U* , the collection of sets *S*1*, . . . , Sn*, and also a *weight wi* for every set. We want to find a sub-collection of *minimal total weight* whose union is *U* , that

S Σ

is, we want to find *I* such that *i*∈*I Si* = *U* , and such that *i*∈*I wi* is minimized. The unweighted problem corresponds to the case in which all weights *wi* equal 1.

The ILP and LP formulation of the unweighted problem can easily be generalized to the weighted case: just change the objective function from *i xi* to *i wixi*.

Σ Σ

minimize Σ*n*

*i*=1

*wixi*

subject to

Σ

*i*:*v*∈*Si xi* ≥ 1 ∀*v* ∈ *U*

*xi* ≤ 1 ∀*i* ∈ {1*, . . . , n*}

*xi* ≥ 0 ∀*i* ∈ {1*, . . . , n*}

(3)

Suppose now that we solve the linear programming relaxation ([3](#_bookmark0)), and we find an optimal fractional solution **x**∗ to the relaxation, that is, we are given a number *xi*∗ in the range [0*,* 1] for every set *Si*. Unlike the case of vertex cover, we cannot round the *xi*∗ to the nearest integer, because if an element *u* belongs to 100 sets, it could be that *xi*∗ = 1*/*100 for each of those sets, and we would be rounding all those numbers to zero, leaving the element *u* not covered. If we knew that every element *u* belongs

to at most *k* sets, then we could round the numbers 1*/k* to 1, and the numbers

≥

*<* 1*/k* to zero. This would give a feasible cover, and we could prove that we achieve a *k*-approximation. Unfortunately, *k* could be very large, even *n/*2, while we are trying to prove an approximation guarantee that is never worse than *O*(log *n*).

Maybe the next most natural approach after rounding the *x*∗*i* to the nearest integer is to think of each *xi*∗ as a *probability*, and we can think of the solution **x**∗ as describing a probability distribution over ways of choosing some of the subsets *S*1*, . . . , Sn*, in which we choose *S*1 with probability *x*1∗, *S*2 with probability *x*∗2, and so on.

Algorithm *RandomPick*

* + Input: values (*x*1*, . . . , xn*) feasible for ([3](#_bookmark0))
  + *I* := ∅
  + for *i* = 1 to *n*

**–** with probability *xi*, assign *I* := *I* ∪{*i*}, otherwise do nothing

* + return *I*

Σ

Using this probabilistic process, the expected cost of the sets that we pick is *i wix*∗*i* , which is the same as the cost of **x**∗ in the linear programming problem, and is at most the optimum of the set cover problem. Unfortunately, the solution that we construct

could have a high probability of missing some of the elements.

Indeed, consider the probabilistic process “from the perspective” of an element *u*. The element *u* belongs to some of the subsets, let’s say for simplicity the first *k* sets *S*1*, . . . , Sk*. As long as we select at least one of *S*1*, . . . , Sk*, then we are going to cover

*u*. We select *Si* with probability *x*∗*i* and we know that *x*∗1 + *x*∗*k* 1; what is the probability that *u* is covered? It is definitely not 1, as we can see by thinking of the case that *u* belongs to only two sets, and that each set has an associated *x*∗*i* equal to 1*/*2; in such a case *u* is covered with probability 3*/*4. This is not too bad, but

· · · ≥

maybe there are *n/*10 elements like that, each having probability 1*/*4 of remaining uncovered, so that we would expect *n/*40 uncovered elements on average. In some examples, Ω(*n*) elements would remain uncovered with probability 1 2−Ω(*n*). How do we deal with the uncovered elements?

−

First of all, let us see that every element has a reasonably good probability of being covered.

1. **Comparison Table**

As an extra activity we evaluated the performance of other algorithms to perform a complete comparison.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Algorithm*** | ***# Iteration*** | ***Counter*** | ***Problem\_Size*** | ***# Subsets*** |
| Depth First |  |  |  |  |
| Breadth First |  |  |  |  |
| A\* |  |  |  |  |
| Recursive |  |  |  |  |