Case Study:
Lab Scheduling Algorithm

Ahmed Ashraf, Ahmed Ghuneim, Mostafa Elsharkawy

Attached Files on Google Drive

- **Header Files:**
  - *Lab.h*
    - Contains **struct** model of Lab
  - *Bruteforce.h*
    - Contains **brute force** struct with constructor and destructor to solve instances
  - *Approximation.h*
    - Contains **approximation** struct with all policies implemented in these slides.
- **CPP Files**
  - *main.cpp*
    - Contains **main** function, with functions to generate problem instances and call on both **brute force** and **approximation** algorithms.

The Problem

We are given L labs, each lab contains n_l students who need to access the lab. In each lab, the order of the students is fixed and cannot be modified.

Each student j gets access to the lab for a duration of p_{lj} hours (which includes any breaks that the student takes).

Whenever a student uses a lab, the lab must be disinfected before another student uses it. Disinfection time is negligible.

We have C disinfections, and any disinfection is applied to all labs. If a lab is still being used, disinfecting bears no effect on the lab.

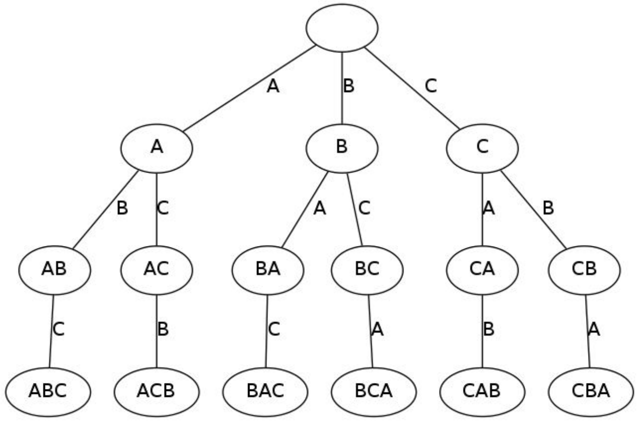
We need to schedule the disinfections for a duration of D days, such that we **maximize the total occupied time of all labs**.

Brute Force Solution

BF: Design Approach

We knew that the problem would be an NP-complete problem, so we started thinking about how we could easily generate all the possible combinations of visit times, and just return the set that produces the maximum amount of lab usage.

This would result in an exponential algorithm that would produce the optimal solution in any case.



** example of recursion tree*

BF: General Pruning (1)

Given a specific problem instance, the pure brute force solution would be to recur on all hours, trying every combination of scheduling visits in each of these hours.

This would result in a recursion tree that would try all possible combinations.

Complexity: $^{24D-1}C_C \rightarrow$ **Exponential**

BF: General Pruning (2)

However, due to the nature of the problem, we can prune our inputs, as well as the conditions we recur on, to bring the complexity down from a tight bound to just an upper bound, since not all paths will necessarily be taken.

Examples of the pruning techniques we implemented are in the following slides.

Pruning Techniques on Instances

As mentioned in the case study document provided, we started by working on the instances themselves.

We ensured that the slots schedule in each lab could not exceed **C+1** students, and that all the sum of **p**’s for any given lab could not exceed **24*D**, since it would be impossible to find a solution to fit all these regardless.

The brute force implementation already implicitly does this, by not trying combinations that exceed these limits.

Pruning Techniques on Recursive Calls

Instead of trying to schedule a **visit** at each hour, we have implemented a method to only try to schedule **visits** during **critical hours**.

By definition, a **critical hour** is an hour in which there is at least one lab slot that has just finished, or just started.

So, technically, the complexity would realistically drop the **24*D-1** to $\Sigma n \rightarrow L*(C+1)$, which is the maximum number of students in all labs, since n is bounded by **C+1**, and there is one **critical hour** per student (ignoring the beginning and end hours for simplicity)

Pruning Techniques on Recursive Calls

The concept of **critical hours** is further exploited when pruning the search space.

If for any given **critical hour** at time **t_1**, we find out that there are no other critical upcoming (i.e no critical hours in **(t_1, 24D]**), we then do not explore the branch resulting from a decision of not scheduling a disinfection at t_1, as it would have no room for progress/other viable decisions to make.

Pruning Techniques on Recursive Calls

Additionally, we introduced a Promising function, that determines whether or not it makes sense to even consider any sub-branch of the recursion tree.

Essentially, if at any critical hour, if the amount of time remaining + the amount of occupied time so far is less than the maximum achieved total amount of occupied time by any other branch, then **this branch can never produce a better solution**. Therefore, we do not proceed with recursion on this branch.

Finding Optimal Solutions

L=1 Solution (Greedy)

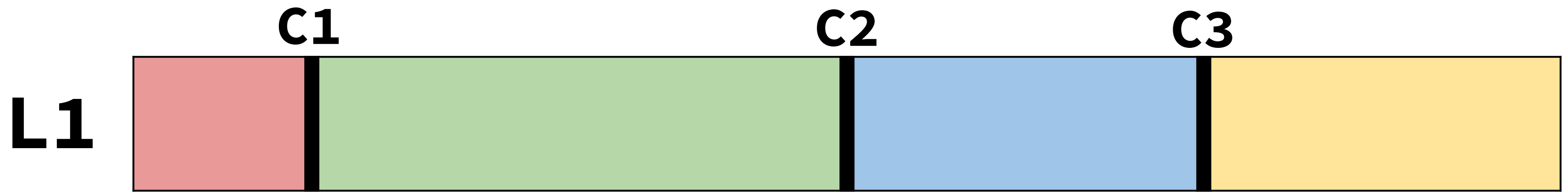
This was a fairly straight forward instance, in which a greedy approach can be used. All the algorithm does is **schedule a visit** at each deadline in the queue, until there are no more visits allowed.

There is no other way to optimize this solution, because delaying a visit only results in time wasted in between lab hours.

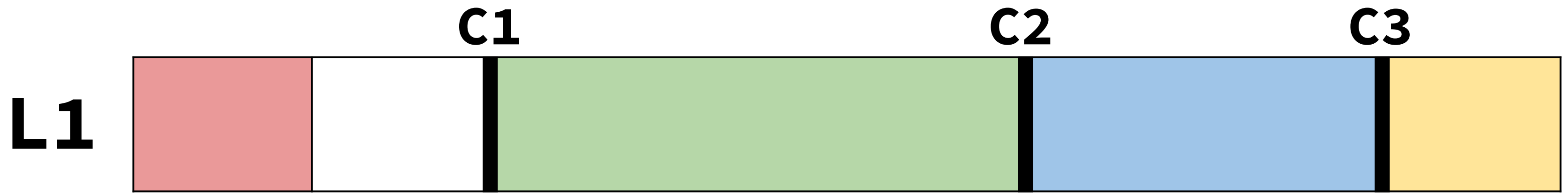
Complexity: Linear time → O(n)

More accurately **O(C)**, as we iterate over the critical hours, which is explained later in depth.

L=1 Visualization (C = 3)



* A visit is made after every slot, in order to maximize the lab time used in the given duration



* By delaying a visit time, no benefit is gained, as the number of hours used is less

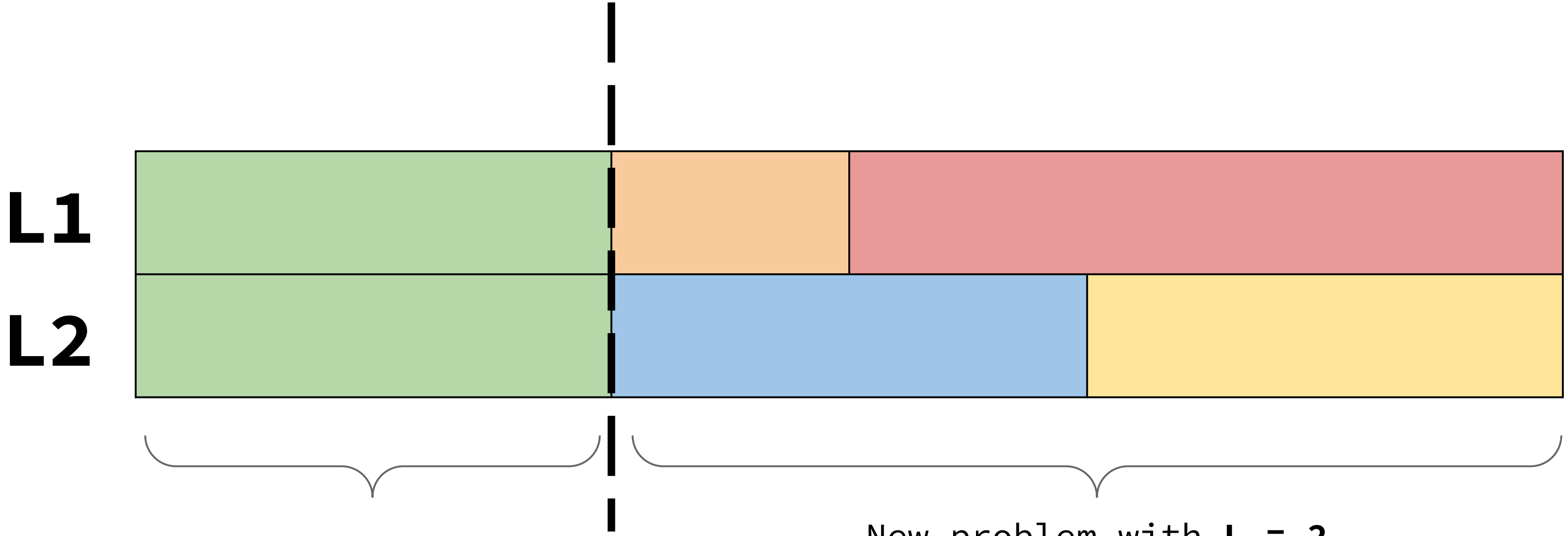
L=2 Solution (Dynamic Programming)

Essentially, we are looking for an optimal substructure, which occurs when a **visit** is scheduled after two labs have have been emptied.

This results in a subproblem with the same number of labs (2), and a reduced number of **allowed visits, hours,** and **students**.

The decision made on whether or not a visit is scheduled will depend on whether this visit **maximizes** the total used hours or not.

Optimal Substructure of L=2



Optimal
Substructure

New problem with **L = 2**
C = C$_{prev}$−1
n = n$_{prev}$−1 for all labs

Approximation Algorithms

Definitions & Prerequisite Information

- **Critical Hour**: Hour where one or more labs become empty
- **Scheduling Policy:** A boolean function that is used to determine whether or not to schedule a disinfection at the current **critical hour**.
- All instances are pruned regardless of the approximation algorithm used
- Upper bound for optimal algorithm: 24DL (no time slots are wasted)

Static Policies

These are a set of policies that do not consider the:

- Statistical attributes of the input instance
- The amount of disinfection visits already scheduled at the given point
- The time remaining before the schedule is over

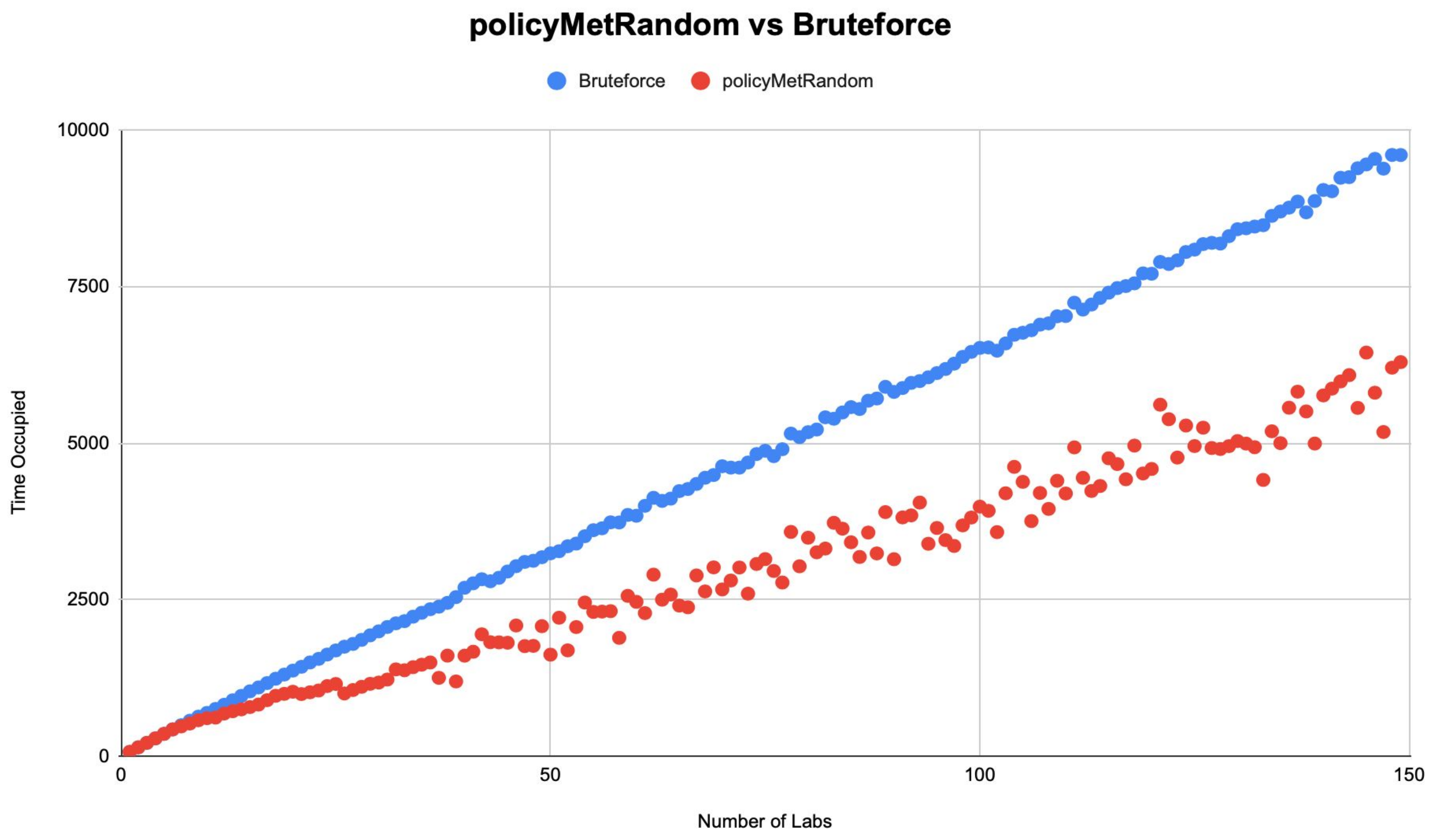
Basic Static Policies

Basic Static Policies

- At every critical hour, randomly choose whether or not to disinfect **(policyMetRandom)**
- Disinfect at every critical hour **(policyMetGreedy)**
- Disinfect when all labs are empty **(policyMetMax)**
- Disinfect at the first critical hour in an interval, where an interval runs for (24*D)/(C+1) clock cycles **(policyMetConstant)**

policyMetRandom

- “At every critical hour, randomly choose whether or not to disinfect”
- This is a control policy, whose results are compared against the results of other policies
- We know that for any policy, we expect that - on average - its total occupied time to be at least as good as that of the random policy
- The policy is not *purely* random, as we also force a disinfection to be taken if all labs are empty



Variables Tested: ∀L ∈ $\mathcal{Z}$ ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

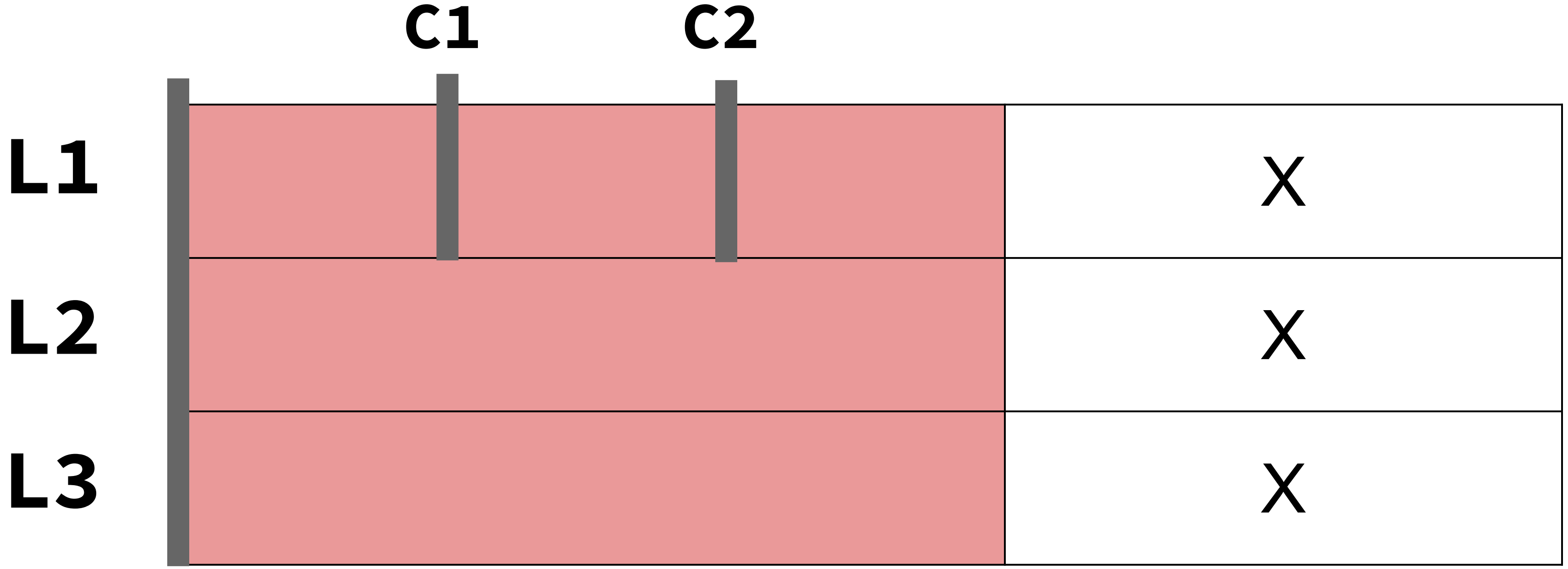
Minimum (policy/Bruteforce): 0.4703

Average (policy/Bruteforce): 0.6516

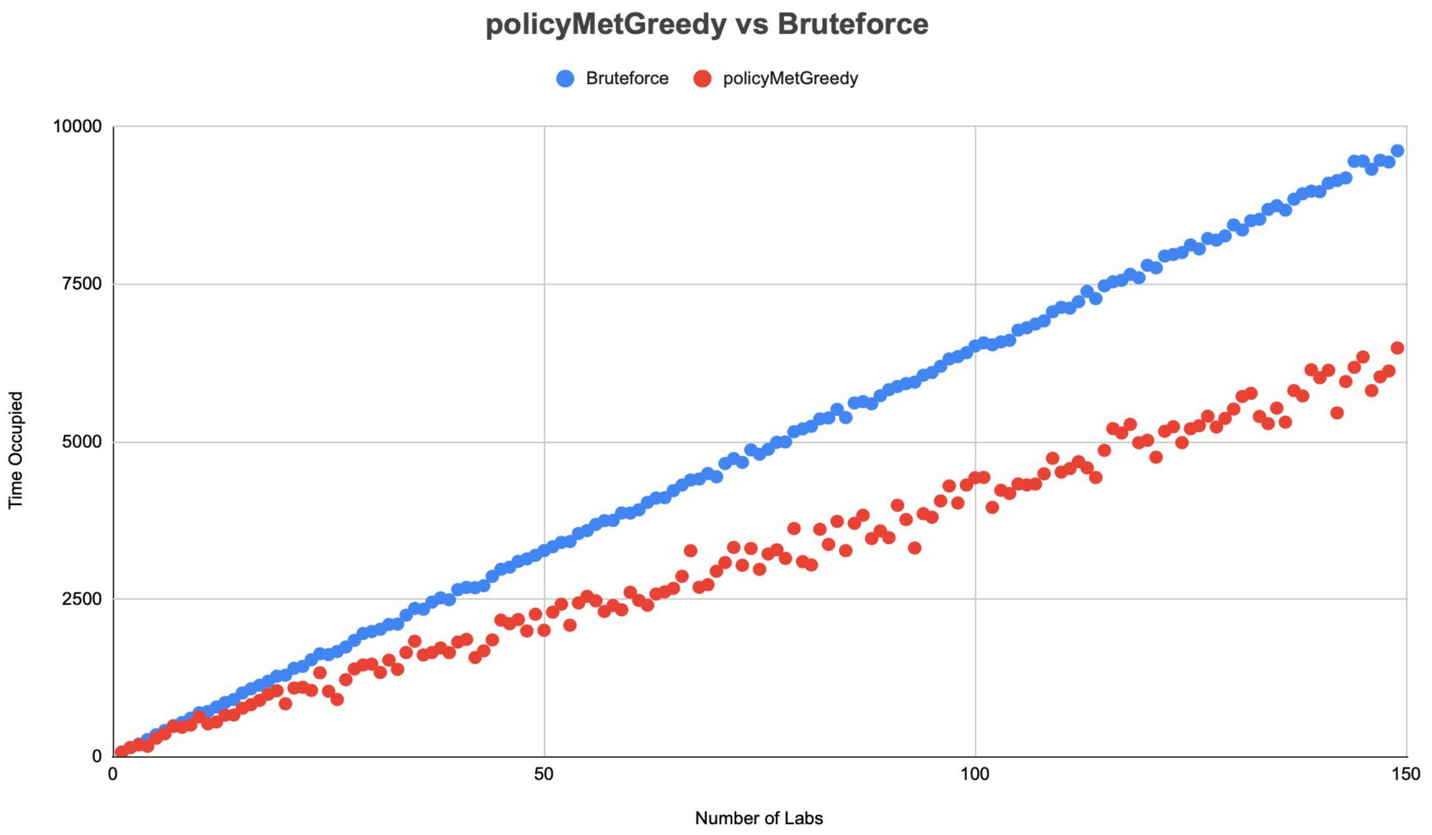
policyMetGreedy

- “Disinfect at every critical hour”
- **When does it fail:**
  - Whenever we have a lab whose students occupy the lab for short amounts of time, effectively “eating up” disinfections
- **Worst Case:**
  - We have C visits and a lab X whose students need the lab for only 1 hour each.The first student in all other labs leave their lab at least 1 hour after the last student has left lab X
  - Each lab is only occupied for C+1 time slots
- **Worst Case Total Occupied Time** = (C+1)L = CL+L
- **Ratio** = CL+L/24DL = C+1/24D
- So this is a **(24D/C+1)-approximation**
- **A visualization** of this example is on the following slide

policyMetGreedy Worst Case Visualization (C = 2)



*The red slots have been used by the students



Variables Tested: $\forall L \in \mathbb{Z} \cap [1, 150)$

Variables Controlled: **C** = 5
D = 3

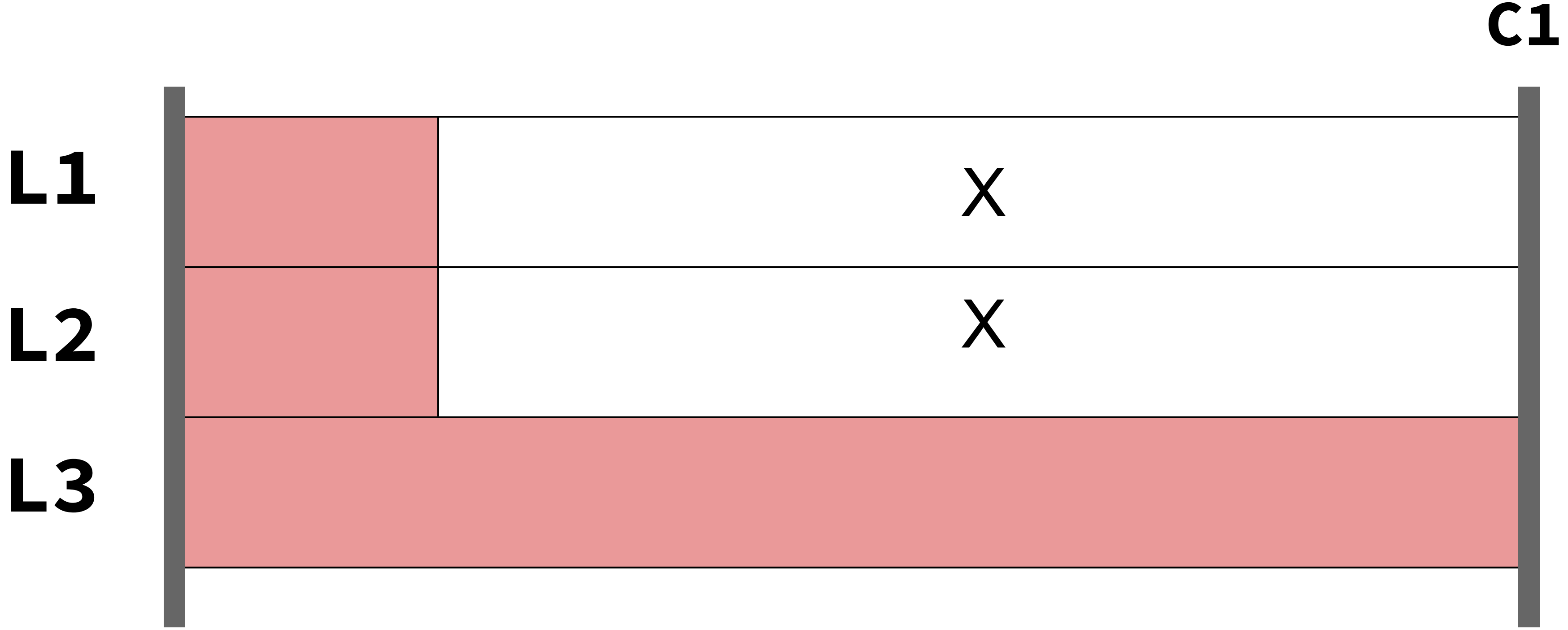
Minimum (policy/Bruteforce): 0.5437

Average (policy/Bruteforce): 0.6791

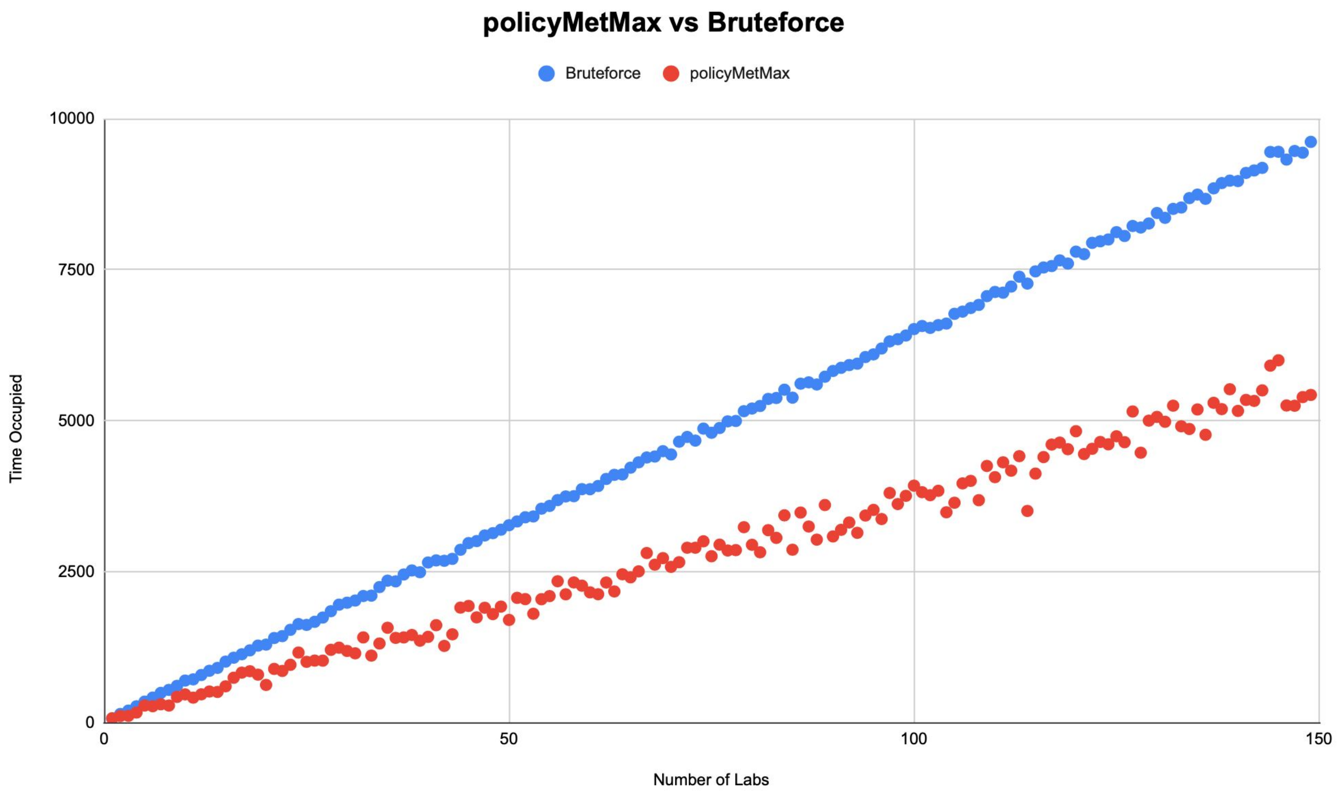
policyMetMax

- “Only Disinfect when all labs have been freed”
- **When does it fail?**
  - When an instance is given where one of the labs contains **one** student that uses the lab for the entire duration. This will result in no cuts being made until the end of the **24*D** hours
- **Worst Case:**
  - Out of a total of **24*D*L** hours, only **24D + (L-1)** hours are used, since 1 student uses a lab for the total hours, and the other labs are used for 1 hour each
- **Ratio: 1/C** → as provided in the document
- **A visualization** of this example is on the following slide

policyMetMax Worst Case Visualization



*The red slots have been used by the students



Variables Tested: ∀L ∈ Z ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

Minimum (policy/Bruteforce): 0.4741

Average (policy/Bruteforce): 0.5947

policyMetConstant

- Divide the timeline into equal intervals
- Define each interval to be of duration (24D)/(C+1) hours
- “Disinfect at the first critical hour in an interval”
- **When does it fail?**
  - The policy is identical to the greedy policy with the only difference being that the disinfection is done per interval. Therefore, for small enough intervals, the policy fails whenever the greedy policy fails.
- **Worst Case Total Occupied Time** = (C+1)L = CL+L
- **Ratio** = CL+L/24DL = C+1/24D
- So this is a **(24D/C+1)-approximation**

Advanced Static Policies

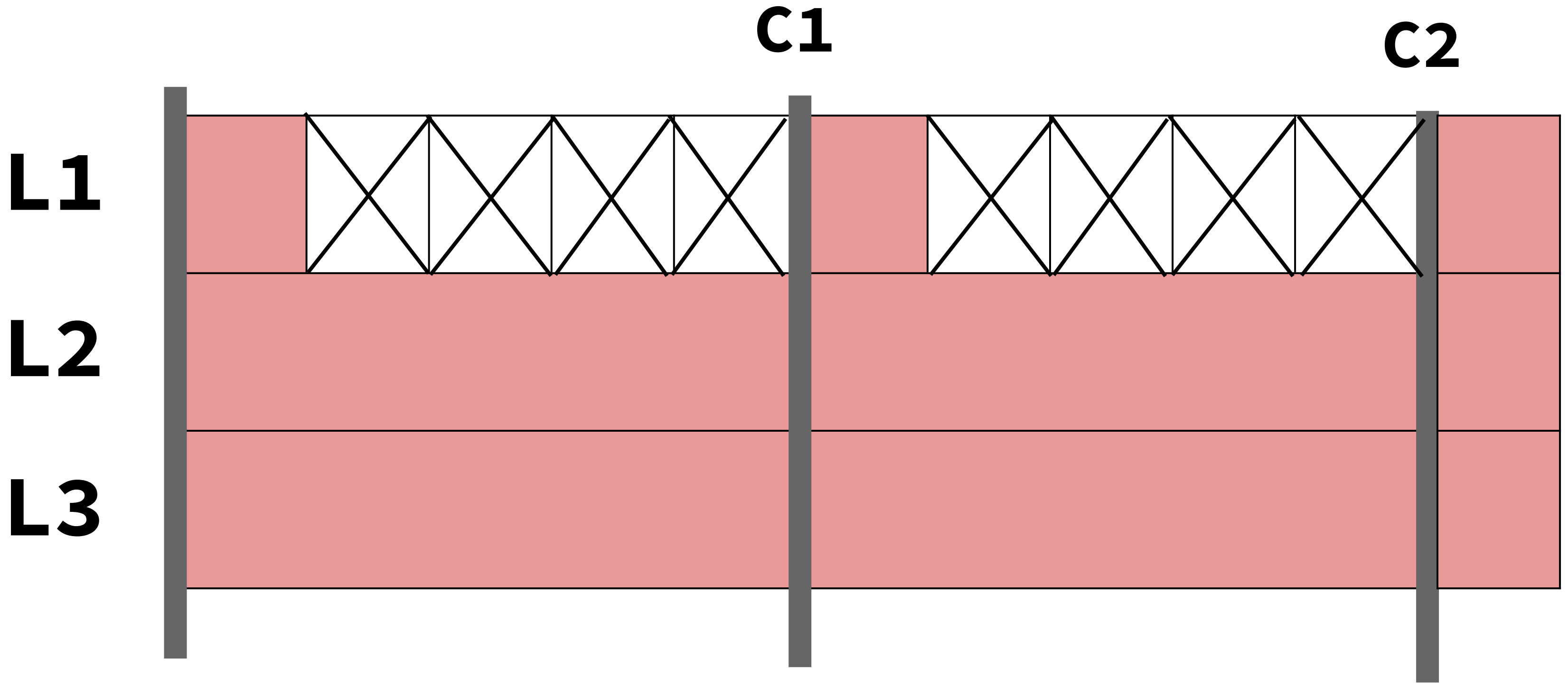
Advanced Static Policies

- Disinfect at every critical hour for which half or more of the labs are empty **(policyMetHalf)**
- At each critical hour, check if the time gained across all labs by disinfecting here, rather than at the next hour, is greater than the next items in the Queue of the labs that are still busy **(policyMetPL)**
- At each critical hour, check if aggregate of the next items in the Queue of the labs that are free is greater than the aggregate of the next items in the Queue of the labs that are still busy **(policyMetQueue)**

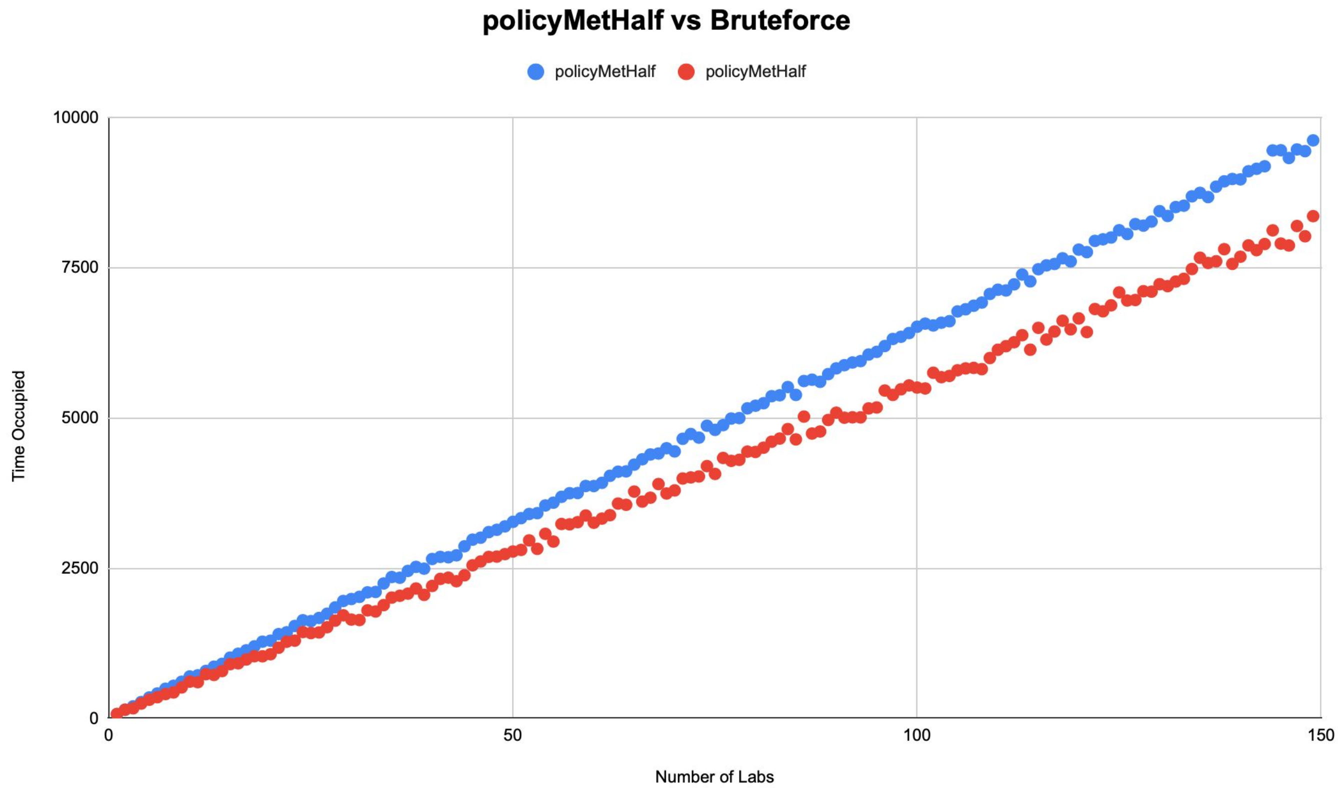
policyMetHalf

- “Only Disinfect when at least half the labs have been freed.”
- **When does it fail?**
  - When an instance has more than half of the labs being extremely positively deviant (larger than the mean) and the rest being extremely negatively deviant (less than the mean yet >= 1 hour)
- **Worst Case:**
  - Out of a total of **24 DL** hours, only 12**DL + ½ CL** hours are used, since half the labs are filled for the full time while the other half of the labs
- **Ratio: ½ (***assuming ½ CL is negl. W.r.t 12DL***) or more**
- **A visualization** of this example is on the following slide.

policyMethHALF Worst Case Visualization



*The red boxes have been used by the students, while the cross represents unoccupied hours



Variables Tested: ∀L ∈ Z ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

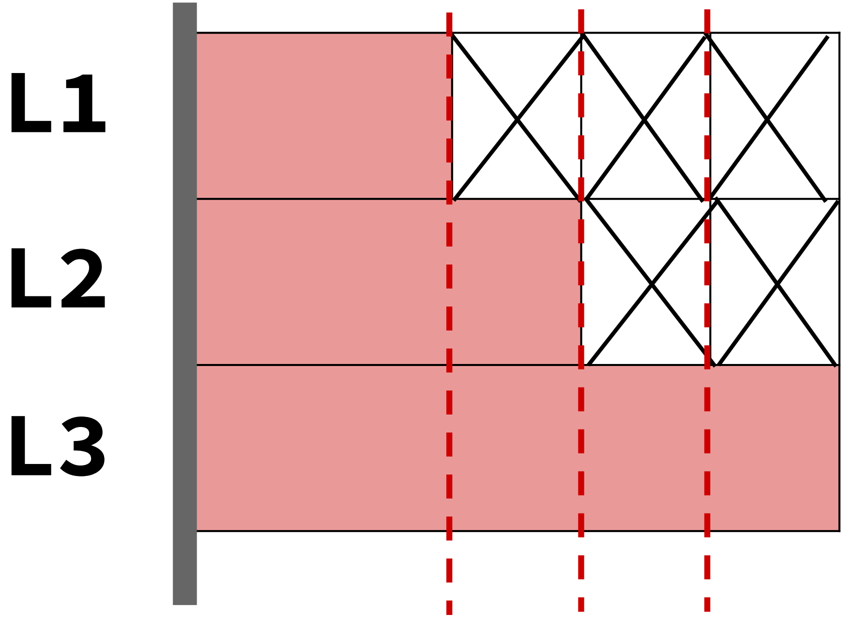
Minimum (policy/Bruteforce): 0.7956

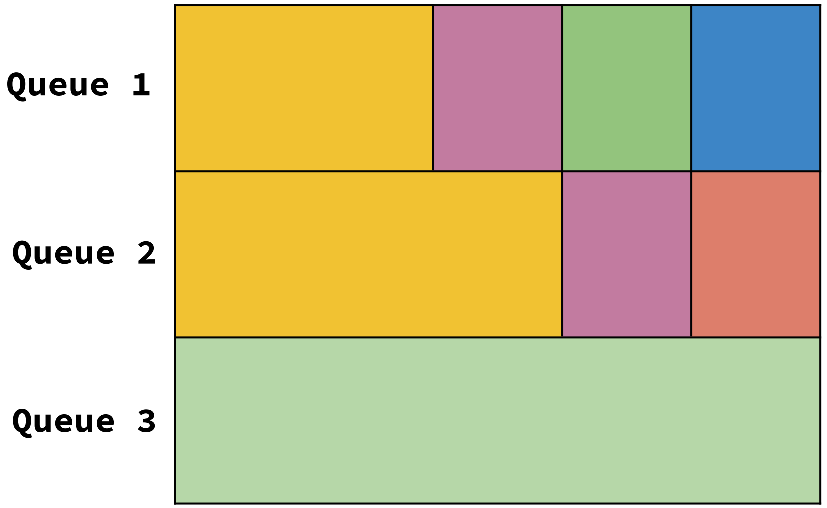
Average (policy/Bruteforce): 0.8586

policyMetPl

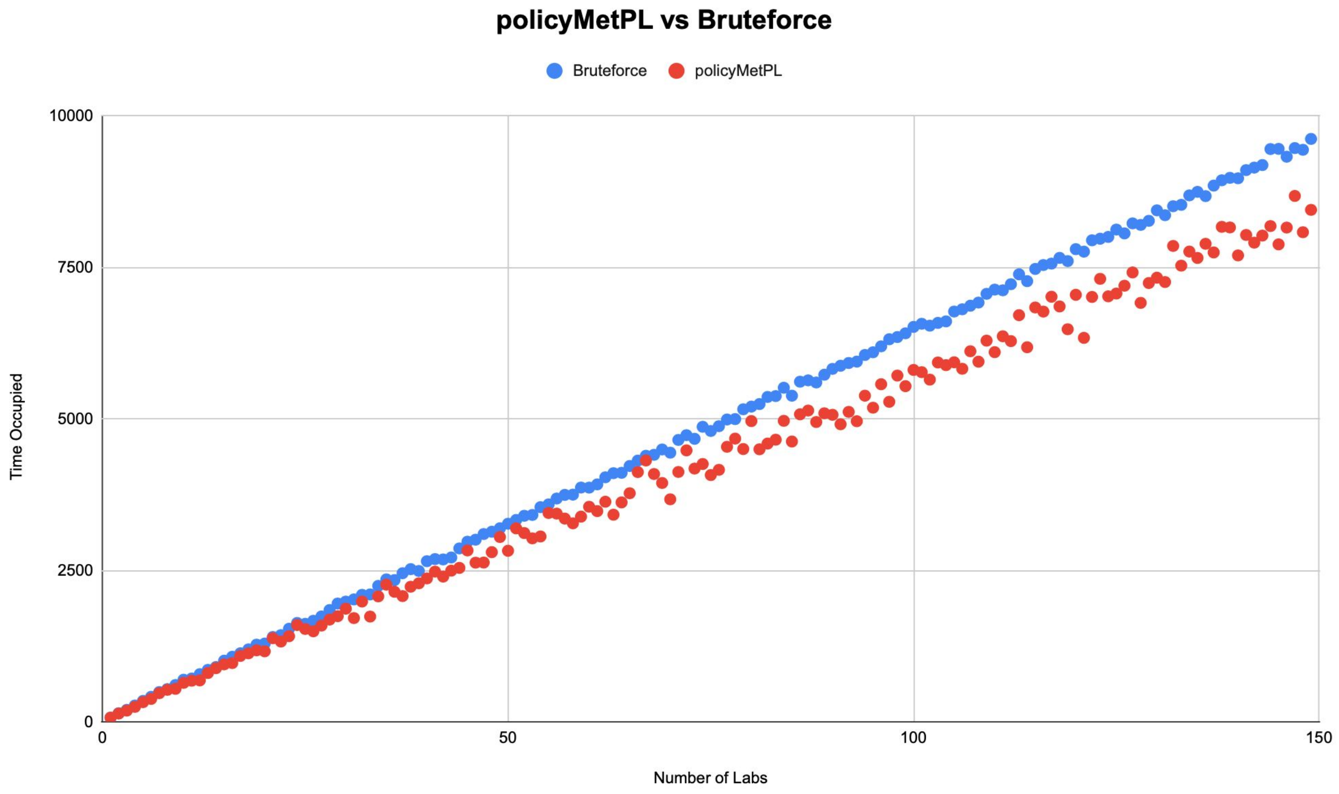
- “Assume this is your last disinfection, only Disinfect when you’d gain more hours disinfecting now (when compared to the next) than you’d lose (queue items in Labs that are still busy)
- **When does it fail?**
  - Assumption is inherently faulty; we might not be at last disinfection, so we are facing the problems inherent with greedy.
  - It would especially fail when we have items in the queue that are positively deviant (larger than the mean), thus higher than average “loss” and frequent critical hours, thus lower than average “gain”
- **A visualization** of this example is on the following slide.

policyMetPL Worst Case Visualization





*The red boxes have been used by the students
*The cross represents unoccupied hours
*The dotted lines represent the critical hours not disinfected



Variables Tested: ∀L ∈ $\mathbb{Z}$ ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

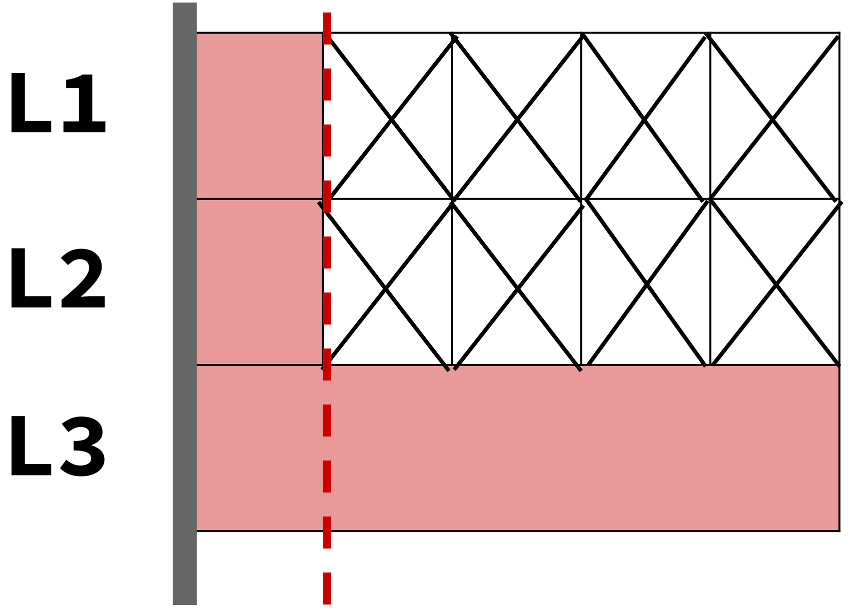
Minimum (policy/Bruteforce): 0.8167

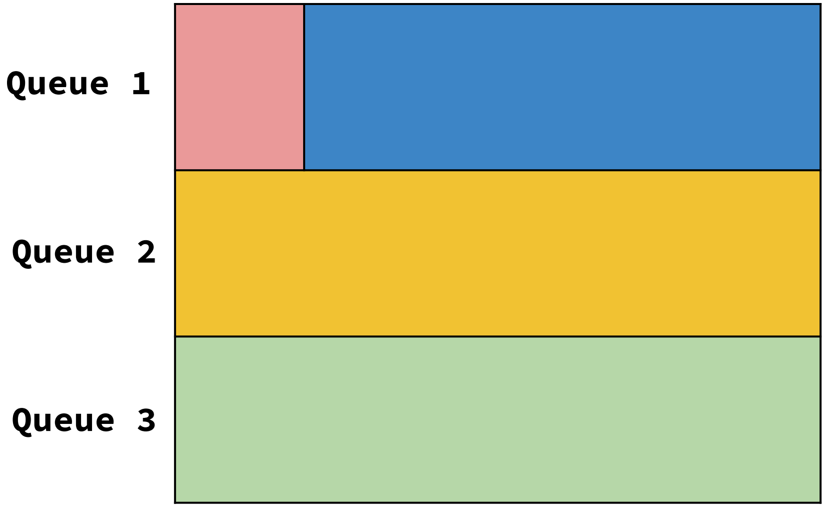
Average (policy/Bruteforce): 0.8975

policyMetQueue

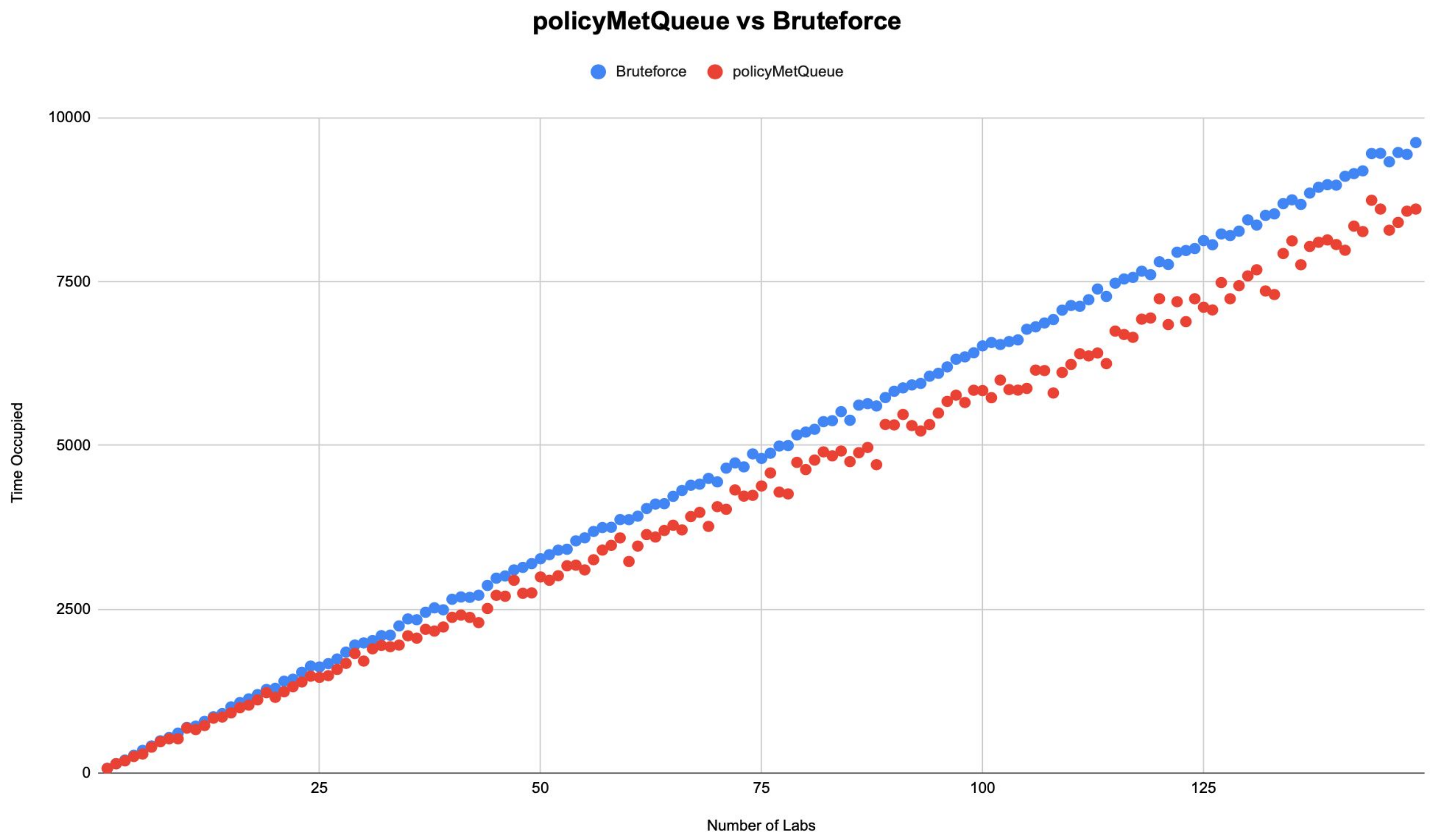
- “Assume this is your last disinfection, only disinfect when you’d gain more hours disinfecting now (Σp of items in front of the queue of labs free now) than you’d lose (Σp of items in front of the queue of labs busy now)”
- **When does it fail?**
  - Assumption is inherently faulty; we might not be at last disinfection, so we are facing the problems inherent with greedy.
  - It would especially fail when we have items in one set of queues is significantly positively deviant (larger than the mean), with the other half significantly negatively deviant (smaller than the mean), which is the case with the mean of the student periods in each lab has a large standard deviation
- **A visualization** of this example is on the following slide.

policyMetQueue Worst Case Visualization





*The red boxes have been used by the students
*The cross represents unoccupied hours
*The dotted lines represent the critical hours not disinfected



Variables Tested: ∀L ∈ $\mathbb{Z}$ ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

Minimum (policy/Bruteforce): 0.8350

Average (policy/Bruteforce): 0.8995

Dynamic Policies

Dynamic Policies

These are a set of policies that take into consideration :

- Statistical attributes of the input instance
- The amount of “slack” time provided by the queue of each lab
- The amount of disinfection visits already scheduled at the given point, and consequently the amount of disinfection left to be scheduled
- The aggregate of positions (Σp) left in the Queues of each lab

Statistical Attributes of the Input Instance

- All attributes are computed after pruning the instance
- For each lab:
    - Compute the mean time needed by students to access the lab
- For the above dataset:
    - Compute the mean
    - Compute the variance
    - Compute the standard deviation
- For each lab:
    - Compute the Z-Score transformation of the lab’s mean
    - Compute the Mean Deviance (Mean of means - Lab’s mean)

SLACK

- Slack is defined as the number of hours we can afford to waste per each lab
- Slack is initialized as $24D - \Sigma p$
- Slack is dynamically updated at each critical hour, by subtracting the number of unused hours from the current value of the slack

Weight & Priority - part I

- Our proposed dynamic policy is a modification to the policyMetHalf, but we will also need to prioritize “more important” labs
- This creates a need to rank the labs based on their importance, which must be based statistical attributes and the slack of the input instances
- This ranking is dynamic and is updated at every critical hour

Weight & Priority - part II

- Prioritize Labs with lower slack
    - Because we can afford to delay the start time of the latter with a lower risk of more unused time slots)
- Prioritize Labs with smaller absolute mean deviance (or smaller absolute Z-Score)
    - Because labs with greater absolute mean deviance (or greater absolute Z-Score) either contain much slower students or much quicker students (compared to other labs), who heavily skew the scheduling times

Weight & Priority - part III

- Define the weight of a lab to be the product of the reciprocal of the mean deviance and the reciprocal of the slack (in case of denominator==0, hardwire the reciprocal to be = 1)
- Define the priority of a lab to be its weight / total weight of labs

Dynamic Policies: Dynamic Weighted Half

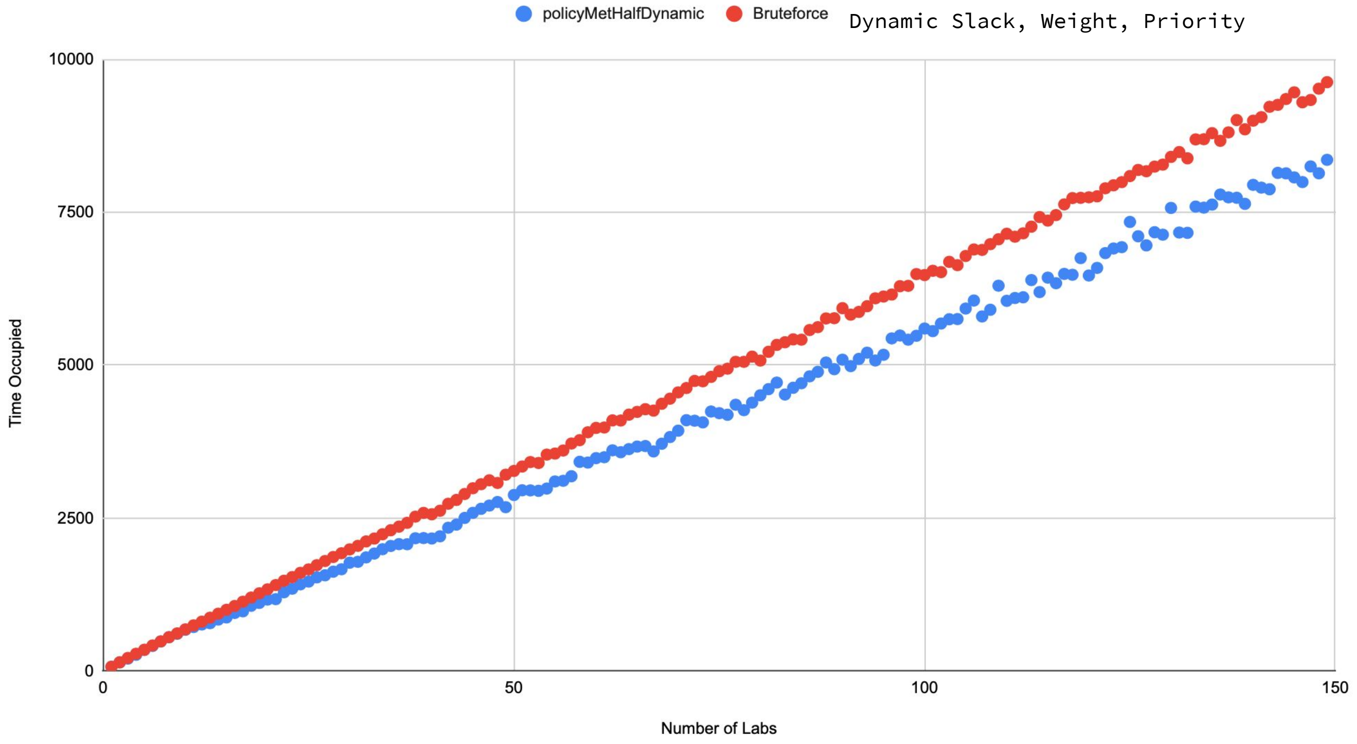
- This policy builds on policyMetHalf, but uses the statistical attributes mentioned previously, to aid in determining whether or not a **visit** should be scheduled.
- Accordingly, at each critical point, we update the **weights** and **priorities** of each lab, and decide to schedule a visit if the priorities of the unvisited labs exceed **50%** of the total priorities (or if half of the unvisited labs are free regardless of priority).
- The **weights** in this case would depend on the **mean deviance**, and the total **slack time** for the current instance, at any given critical hour.

Dynamic Policies: Dynamic Weighted Half

- Our **two** dynamic policies differ from one another based on their level of dynamism. Under both policies, weights and priorities are updated every critical hour. The policies differ on the dynamism of the slack and the statistical attributes.
- Under the initially proposed policy, only the **slack** is updated at every critical hour, and the statistical attributes of the labs are not modified from when they were initially calculated.
- Under the second proposed policy, both the **slack** and the **statistical attributes** are updated at every critical hour.

policyMetHalfDynamic vs Bruteforce

Dynamic Slack, Weight, Priority



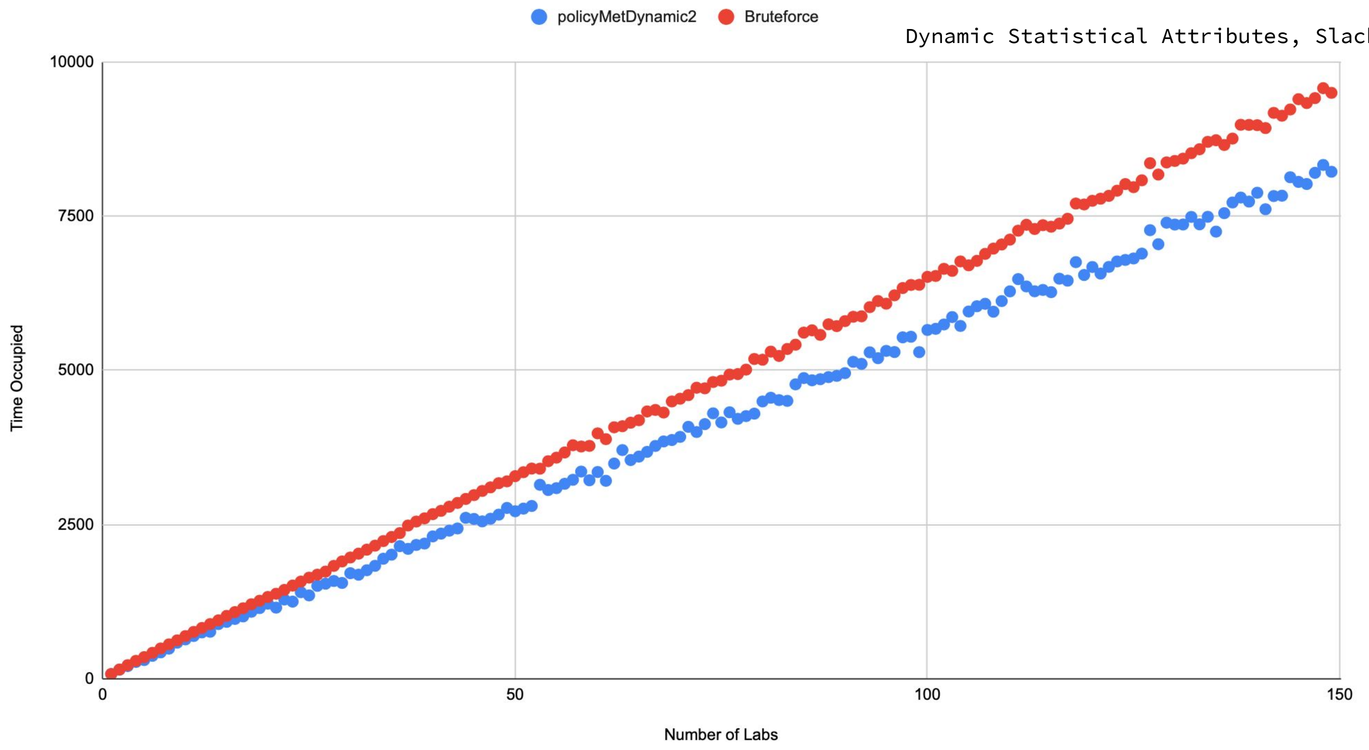
Variables Tested: ∀L ∈ Z ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

Minimum (policy/Bruteforce): 0.8333

Average (policy/Bruteforce): 0.8759

policyMetDynamic2 vs Bruteforce



Dynamic Statistical Attributes, Slack, Weight, Priority

Variables Tested: ∀L ∈ $\mathbb{Z}$ ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

Minimum (policy/Bruteforce): 0.8162

Average (policy/Bruteforce): 0.8692

Observations

Analysis of Empirical Data - policyMetHalfDynamic

- **Varying # of Labs (L) and P_{lj} of each lab:**
  - As the number of labs increases (while C and D remain constant), we can observe that the gap between the **optimal** and **approximation** solutions slightly increases at a steady rate.
  - The ratio between the approximate solution and the optimal solution decreases. This ratio is always bounded by the theoretical values that we have calculated for each policy.
  - We primarily focused on varying the number of labs and the student queues per lab, due to the fact that the **brute force** algorithm crashes for large instances of C and D.

Analysis of Empirical Data - policyMetHalfDynamic

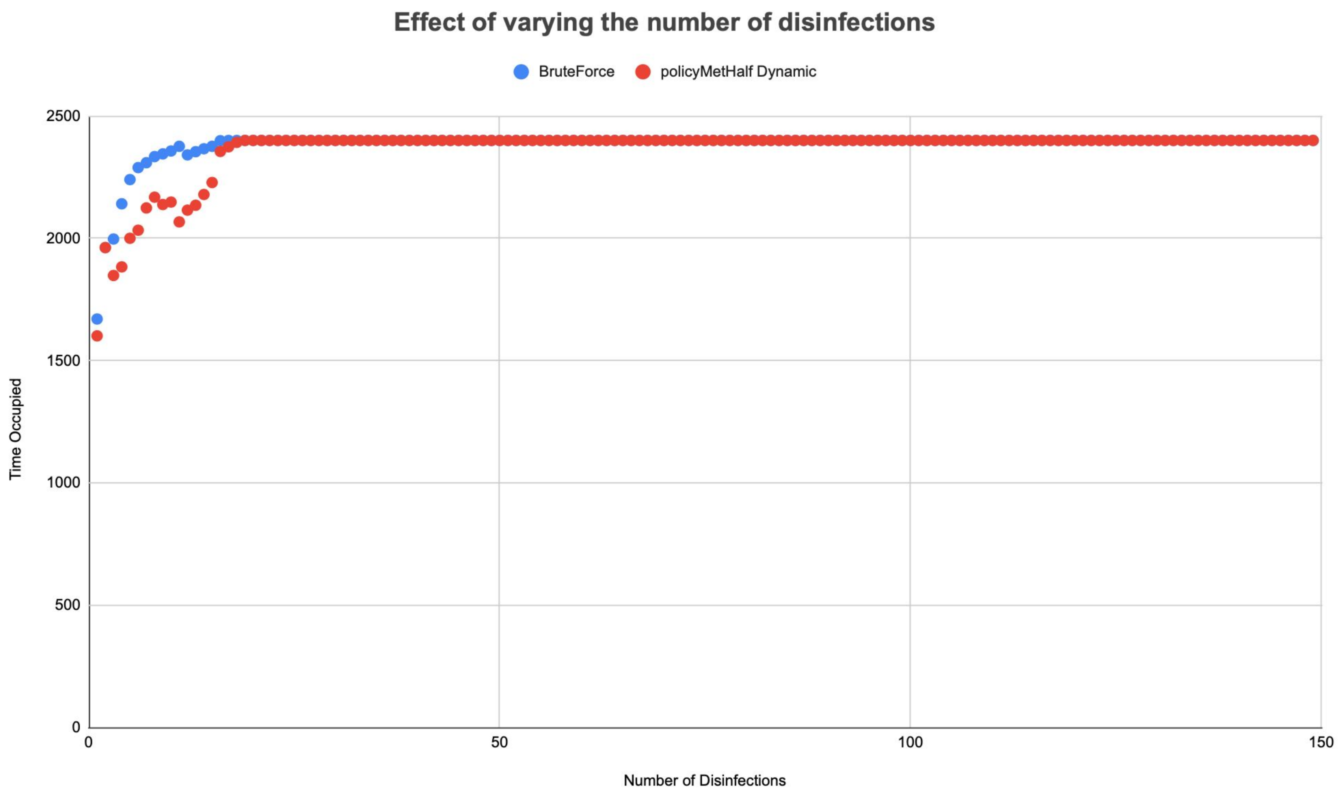
- **Varying # of Visits (C):**
  - As the number of allowed visits increases (everything else held constant), we notice that the graph generally becomes flat, and the approximation meets the optimal solution. This makes sense, because as the number of visits increases, so does the flexibility of assigning extra visits to accommodate more labs. Thus, varying C won’t help us as much in analysis.
- **Varying # of Days (D):**
  - As the number of days increases, we have noticed that the ratio steadily decreases until day 76, where it jumps then steadily decreases once more. Generally, the ratio is high, which also makes sense, since adding days adds flexibility (similar to increasing C).

Differences Between Dynamic Policies

- **As shown in the graphs:**

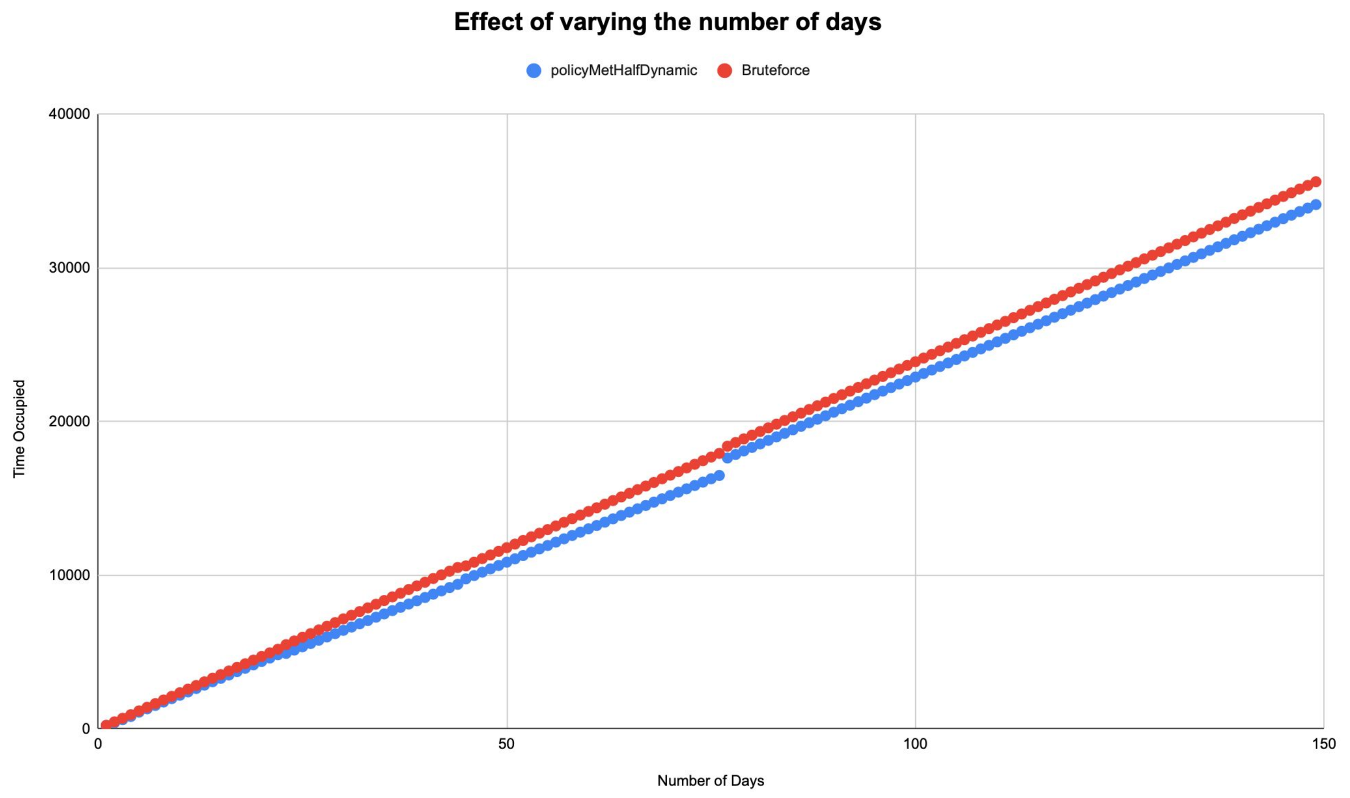
  policyMetHalfDynamic produces higher averages in the ration between **approximation** and **optimal** solutions, than *policyMetHalfDynamic2*. We can attribute this to the fact that the first function calculates the means and standard deviations once, without updating it each time a new visit is scheduled.

  This obviously reduces the optimality of the overall approximation, which is why we have settled on the first dynamic policy as our best policy.



Variables Tested: ∀C ∈ $\mathbb{Z} \cap [1, 150)$

Variables Controlled: **L** = 10
D = 10



Variables Tested: $\forall D \in \mathbb{Z} \cap [1, 150)$

Variables Controlled: **L** = 10
D = 10

Choosing Approximations

Best Approximation

- **Default Constructor:**

We have added a default constructor for our **Approximation** struct, which calls a wrapper function that runs all of the approximation policies we have implemented (each in tractable time).

The function then returns all results, as well as the **approximation policy** with the best ratio for that instance, and this helped us focus on the more important approximations throughout the case study.

Calling: Approximation *instanceName*(L, C, D, P);

Choosing Approximations

- **Parameterized Constructor:**

The parameterized constructor has an extra parameter which indicates the policy the user decides to use when running the algorithm. The indices can be found in the next slide.

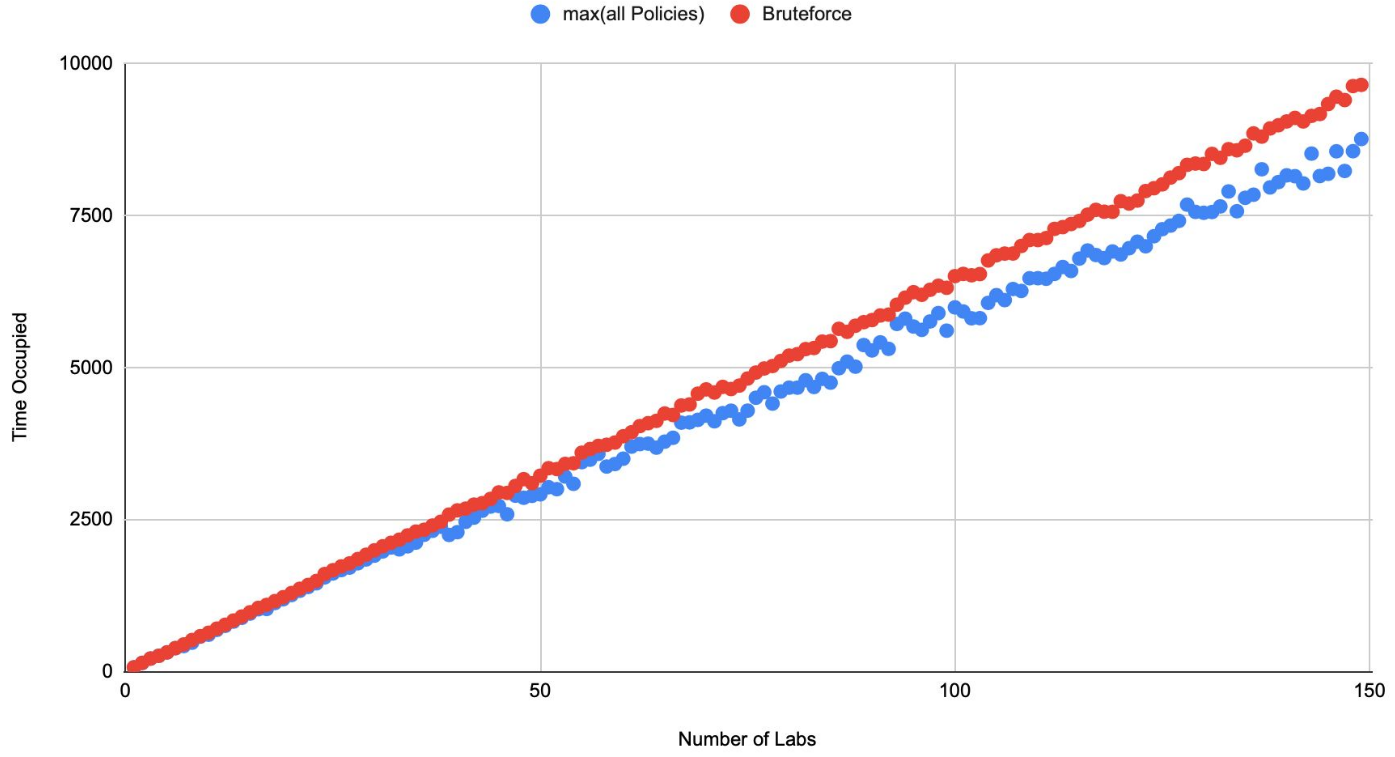
Calling: Approximation *instanceName*(L, C, D, P, i)

Where **i** is the policy chosen.

Approximation Indices

1. `policyMetHalf`
2. `policyMetMax`
3. `policyMetConstant`
4. `policyMetQueue`
5. `policyMetPL`
6. `policyMetRand`
7. `policyMetGreedy`
8. `policyMetHalfDynamic`
9. `policyMetHalfDynamic2`

max(All Policies) vs Bruteforce



Variables Tested: ∀L ∈ $\mathbb{Z}$ ∩ [1, 150)

Variables Controlled: **C** = 5
D = 3

Minimum (policy/Bruteforce): 0.8650

Average (policy/Bruteforce): 0.9227

That's Approximately it.
Thank you!