Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

IHEAP: GENERALIZED HEAP MODULE WITH CASE STUDIES IN MARKETING AND EMERGENCY ROOM SERVICES

Aniruddha Mukherjee[1], and Vernon J. Rego[1]

[1]Department of Computer Science, Purdue University, West Lafayette, IN, USA

ABSTRACT

We introduce a novel `iheap` module, a flexible Python library for heap operations. The `iheap` module introduces a generalized comparator function, making it more flexible and general compared to the `heapq` that is commonly used. We demonstrate that the `iheap` module achieves parity in terms of time complexity and memory usage against established standard heap modules in Python. Furthermore, the `iheap` module provides advanced methods and customization options unavailable in its counterparts, which enable the user to implement the heap operations with greater flexibility and control. We demonstrate the `iheap` module through two case studies. The first case study focuses on the efficient allocation of funds across marketing campaigns with uncertain returns. The second case study focuses on patient triaging and scheduling in the emergency rooms of hospitals. The `iheap` module provides a powerful and easy-to-use tool for heap operations commonly used in simulation studies.

1 INTRODUCTION

The heap data structure is fundamental to many computational applications, including discrete event simulation (Schriber et al. 2017), network-based simulation (Kanezashi and Suzumura 2015), reinforcement learning (Ma et al. 2024), multi-arm bandit problems (Rinciog and Meyer 2021), and routing algorithms (Gutenschwager et al. 2012). The heap is an efficient data structure for creating and performing operations like push, pop, remove, replace, and merge on past, current, and future event lists for a discrete-event simulation (Liu et al. 2017). The heap data structure allows for $O(1)$ access to the top-priority event and $O(log\, n)$ insertion and deletion of events (Cormen et al. 2022). Therefore, maintaining and updating a scheduler for simulation is fundamentally a heap operation. The most common heap data structure that is used currently is the min-heap from the `heapq` library in Python programming language (Hannon et al. 2018). The current solution of building schedulers for simulation has several shortcomings. First, the current packages only support min-heap, and max-heap is achieved by negating the objects in a heap. This solution introduces additional steps in coding, makes code error-prone, and works only for alphanumeric values. Often in simulations, non-numeric values of objects are encountered such as triaging of patients in healthcare where negative values are meaningless. Second, the `heapq` does not provide an easy way to update priorities of objects in a heap. Third, and most importantly, the current solutions do not provide custom comparators for the simulation of objects with complex attributes or values, such as patient conditions and triages, and aircraft control with multi-dimensional priority rankings. For simulating objects with multiple weighted ranking fields, such as investment decisions with uncertain returns and risks and patients with severity, urgency, and deterioration possibilities, a general comparator that can work with flexible data-types and functions is required.

Therefore, we propose the package called `iheap` that addresses the shortcomings of existing packages by supporting both min- and max-heaps, user-defined comparator functions, several native heap property implementations, and flexible merge, replace, and remove functions. We have demonstrated the `iheap` package using two cases on funding marketing projects with uncertain rewards and risks, and dynamic scheduling of patients in an emergency room with patient triage and waiting. Both examples require comparisons of vector-valued functions and dynamic updates of the queues. The proposed `iheap` provides advantages over other heap implementations by supporting these functions at a native level. We provide a time and space complexity comparison of the proposed heap data structure with respect to the existing solutions. Researchers can use the package to conduct simulation experiments based on objects with composite attributes and general-purpose comparators. In Section 2, we describe the methods in detail. In Section 3, we analyze the two case studies. Finally, in section 4, we provide concluding remarks.

2 METHODS

The `iheap` module implements a robust and flexible binary heap data structure that supports both *min-heap* and *max-heap* heap properties. Emphasis is placed on a comprehensive set of configurable parameters, allowing for the user to have granular specification over heap properties, or the user can opt to rely on the default settings. Furthermore, the `iheap` module also allows for on-the-fly customization of the arrangement of the elements in the heap through the use of custom comparator functions. The `iheap` module provides the below set of operations available to the user: (i) `heapify`: Constructing a heap from any list, (ii) `push`: Inserting a new element to the heap while maintaining the heap property, (iii) `pop`: Removing the root element of the heap while maintaining the heap property, (iv) `merge`: Combining iterables which individually do not have to be arranged into a single heap, (v) `replace`: Updating any number of arbitrary elements while maintaining the heap property, and (vi) `remove`: Removing any number of arbitrary elements while maintaining the heap property.

2.1 Implementation of the `iheap` Module

The `iheap` module leverages the *Sift Down*, *Sift Up*, and *Find Index* routines as described in the Algorithms 1, 2, and 3. The primary purpose of the *Sift Down* routine is to restore the heap property after an element has been removed or altered near the top of the heap. Starting from a designated node, this procedure examines the node's children to ensure that the parent-child ordering follows the rules of the heap. For a max-heap, the parent must be larger than its children; for a min-heap, the parent must be smaller. If a user has supplied a custom comparator, the routine uses the comparator to decide the relative ordering. At each step, the procedure first checks whether the current element has the left child. If a right child is also available, the routine selects the child that should take precedence according to the heap's ordering. If swapping is necessary, determined by comparing the current element with the selected child, the elements are interchanged, and the routine continues from the child's position until the heap property is fully restored.

Algorithm 1 Sift down procedure for heap reordering.

```
1:  procedure SIFTDOWN(heap, start, end, max_heap, cmp)
2:      $i \leftarrow$ start
3:      while $(2i+1) <$ end do
4:          $child \leftarrow 2i+1$
5:          if $child+1 <$ end then
6:              if (cmp($heap[child]$) < cmp($heap[child+1]$)) equals max_heap then
7:                  $child \leftarrow child+1$
8:          if (cmp($heap[i]$) < cmp($heap[child]$)) equals max_heap then
9:              Swap(heap[i], heap[child])
10:             $i \leftarrow child$
11:         else
12:             break
```

The *Sift Down* routine performs one comparison and a possible swap per level of the heap, resulting in a time complexity of $O(\log n)$ for a heap with n elements. This procedure operates in-place and requires $O(1)$

additional memory. The *Sift Up* procedure manages upward reordering when an element's key is modified or a new element is appended at the bottom of the heap. At each iteration, a comparison is performed between the current node and its parent. As described previously, if a comparator function is supplied by the user, then the comparator function is used to compare the current node and the parent node. When the ordering between the node and its parent is inconsistent with the heap property, the node positions are exchanged and the process continues from the parent node's position. The operation terminates when the element reaches a specified boundary or its relative ordering is compliant.

Algorithm 2 Sift up procedure for heap reordering.

```
1: procedure SIFTUP(heap, index, limit, max_heap, cmp)
2:     while $index >$ limit do
3:         $parent \leftarrow \lfloor (index - 1)/2 \rfloor$
4:         if (cmp($heap[parent]$) < cmp($heap[index]$)) equals max_heap then
5:             Swap(heap[parent], heap[index])
6:             $index \leftarrow parent$
7:         else
8:             break
```

The *Sift Up* routine performs at most one comparison and corresponding swap per level, resulting in a time complexity of $O(\log n)$. This procedure operates in-place and requires $O(1)$ additional memory. The *Find Index* routine in Algorithm 3 implements a recursive search to identify the index of a target element within a heap. The routine first checks if the current index is within the bounds of the heap. If an index falls outside the heap, the search on that branch terminates. When a user-supplied comparator is provided, the routine applies it to both the current element and the target element to determine their relative ordering. The routine stops searching along a subtree if the current node's ordering indicates that the target cannot be present in that subtree. Once an equality condition is detected between the current node and the target, the routine returns the target node's index. If no match is found at the current node, the routine recursively searches the left child followed by the right child node.

Algorithm 3 Find index procedure in a heap data structure.

```
1: procedure FINDINDEX(heap, target, max_heap, cmp, i)
2:     if $i \geq$ length($heap$) then
3:         return None
4:     $current\_cmp \leftarrow heap[i]$
5:     $target\_cmp \leftarrow target$
6:     if $cmp \neq$ None then
7:         $current\_cmp \leftarrow cmp(current\_cmp)$
8:         $target\_cmp \leftarrow cmp(target\_cmp)$
9:     if ((not $max\_heap$) and ($current\_cmp > target\_cmp$)) or (($max\_heap$) and ($current\_cmp <$
   $target\_cmp$)) then
10:        return None
11:    if $current\_cmp = target\_cmp$ then
12:        return $i$
13:    $res \leftarrow$ FINDINDEX(heap, target, max_heap, cmp, $2 \cdot i + 1$)
14:    if $res \neq$ None then
15:        return $res$
16:    return FINDINDEX(heap, target, max_heap, cmp, $2 \cdot i + 2$)
```

In the worst case, *Find Index* traverses every node, resulting in $O(n)$ time complexity. Since the heap is a complete binary tree, the recursion stack is limited to $O(logn)$ space. The *Heapify* procedure in the Algorithm 4 reorders an input array to satisfy the heap property. It processes elements either from the last non-leaf node to the root (using *Sift Down*) when in bottom-up mode or from the beginning upward (using *Sift Up*) when in top-down mode. In bottom-up mode, the *Heapify* procedure runs in $O(n)$ time. in top-down mode, it runs in $O(n\log n)$ time. The *Heapify* procedure operates in-place and requires $O(1)$ additional memory. The *Pop* procedure removes the root element of the heap by replacing it with the last element. It then restores the heap property by applying *Sift Down* or *Sift Up* as required.

Algorithm 4 Heap construction procedure (heapify).

```
1: procedure HEAPIFY(heap, max_heap, cmp, bottom_up)
2:     $n \leftarrow \text{length}(heap)$
3:     if bottom_up is true then
4:         for $i \leftarrow \lfloor n/2 \rfloor - 1$ down to 0 do
5:             SIFTDOWN(heap, i, n, max_heap, cmp)
6:     else
7:         for $i \leftarrow 1$ to $n-1$ do
8:             SIFTUP(heap, i, 0, max_heap, cmp)
```

Algorithm 5 Removal procedure of first element in the heap.

```
1: procedure POP(heap, max_heap, cmp, bottom_up)
2:     if heap is empty then
3:         return None
4:     else if heap contains only one element then
5:         return Pop the single element from heap
6:     else
7:         $first \leftarrow heap[0]$
8:         Replace $heap[0]$ with the last element and remove the last element
9:         if bottom_up is true then
10:            SIFTDOWN(heap, 0, length(heap), max_heap, cmp)
11:            SIFTUP(heap, new_index, 0, max_heap, cmp)
12:        else
13:            SIFTDOWN(heap, 0, length(heap), max_heap, cmp)
14:        return $first$
```

The extraction run in $O(\log n)$ time with $O(1)$ additional space as in Algorithm 5. The *Push* procedure in the Algorithm 6 appends a new element to the heap and restores the heap ordering. In bottom-up mode, it performs a *Sift Up* from the new element's position. In top-down mode, it triggers a complete heap reordering by employing the *Heapify* function shown previously. When using *Sift Up*, the insertion takes $O(\log n)$ time. When using *Heapify*, the operation may take $O(n)$ time. Both approaches require $O(1)$ extra space. The *Remove* in the Algorithm 7 procedure locates occurrences of a specified target within the heap, replaces each found element with the last element, and ensures the heap ordering is maintained after the removal of the occurrences of the specified target through *Sift Down* and *Sift Up*.

The *Replace* procedure in the Algorithm 8 scans the heap for instances of a given element, substitutes each instance with a new element, and maintains the heap property through the function calls to *Sift Down* and *Sift Up*. The *Merge-unsorted* procedure in the Algorithm 9 consolidates multiple unsorted iterables into a single list and then organizes that list into a heap using a bottom-up *Heapify* process. The merging

Algorithm 6 Insertion procedure of an element into the heap.

```
1: procedure PUSH(heap, item, max_heap, cmp, bottom_up)
2:     Append item to heap
3:     $n \leftarrow$ length($heap$)
4:     if bottom_up is true then
5:         SIFTUP(heap, n-1, 0, max_heap, cmp)
6:     else
7:         HEAPIFY(heap, max_heap, cmp, bottom_up = false)
```

Algorithm 7 Removal procedure of `n` elements from the heap.

```
1: procedure REMOVE(heap, target, max_heap, cmp, n)
2:     $count \leftarrow 0$
3:     while $count < n$ do
4:         $idx \leftarrow$ FINDINDEX(heap, target, max_heap, cmp, start=0)
5:         if idx is None then
6:             break
7:         if idx equals length(heap) - 1 then
8:             Remove the last element from heap
9:         else
10:            Replace $heap[idx]$ with the last element and remove the last element
11:            SIFTDOWN(heap, idx, length(heap), max_heap, cmp)
12:            SIFTUP(heap, idx, 0, max_heap, cmp)
13:        $count \leftarrow count + 1$
```

and heapification process executes in $O(n)$ time relative to the total element count and requires $O(n)$ space. The *Merge-sorted* in the Algorithm 10 consolidates multiple sorted iterables, all of which follow the same heap property. The sorted merge generates a sorted list in $O(n \log k)$ time for k sorted lists with n total elements.

Algorithm 8 Replacement procedure of `n` elements from the heap.

```
1: procedure REPLACE(heap, old, new, max_heap, cmp, n)
2:     $count \leftarrow 0$
3:     while $count < n$ do
4:         $idx \leftarrow$ FINDINDEX(heap, old, max_heap, cmp, start = 0)
5:         if idx is None then
6:             break
7:         Set $heap[idx] \leftarrow new$
8:         SIFTDOWN(heap, idx, length(heap), max_heap, cmp)
9:         SIFTUP(heap, idx, 0, max_heap, cmp)
10:        $count \leftarrow count + 1$
```

2.2 Comparative Analysis with Existing Modules

Overall, the *iheap* module strikes a careful balance between versatility, performance, and ease of use. Below, a comparison of functionalities of `iheap` with well-known Python module is provided: (i) **heapq.** The standard *heapq* module offers functions for *heapify*, *heappush*, and *heappop*. However, it is inherently restricted to min-heap structures and does not allow custom comparator functions. In contrast, *iheap* enables

Algorithm 9 Merge unsorted lists into a sorted min-heap or max-heap.

```
1: procedure MERGE(iterables, max_heap, cmp)
2:     Initialize empty list $merged\_heap$
3:     for each $iterable$ in iterables do
4:         for each $item$ in iterable do
5:             Append $item$ to $merged\_heap$
6:     HEAPIFY($merged\_heap$, max_heap, cmp, bottom_up = true)
7:     return $merged\_heap$
```

Algorithm 10 Merge sorted min-heaps or max-heaps into a single heap.

```
1: procedure MERGE SORTED(iterables, max_heap, cmp)
2:     Initialize empty list $merged\_heap$
3:     Initialize empty list $entries$
4:     for each $iterable$ in $iterables$ do
5:         let $it \leftarrow \text{ITER}(iterable)$
6:
7:         if there is a next element $item$ in $it$ then
8:             let $key \leftarrow (cmp(item)$ if $cmp$ else $item)$
9:             if $max\_heap$ then
10:                $key \leftarrow -key$
11:            Append tuple $(key, item, it)$ to $entries$
12:    if $entries \neq []$ then
13:        HEAPIFY($entries$, max_heap $= false$, cmp $= \lambda x.x[0]$, bottom_up $= true$)
14:    while $entries \neq []$ do
15:        let $(key, item, it) \leftarrow entries[0]$
16:        let $last \leftarrow$ remove last element of $entries$
17:        if $entries \neq []$ then
18:            $entries[0] \leftarrow last$
19:            SIFTDOWN($entries$, max_heap $= false$, cmp $= \lambda x.x[0]$, start $= 0$, end $= |entries|$)
20:        Append $item$ to $merged\_heap$
21:
22:        if there is a next element $nxt$ in $it$ then
23:            let $new\_key \leftarrow (cmp(nxt)$ if $cmp$ else $nxt)$
24:            if $max\_heap$ then
25:                $new\_key \leftarrow -new\_key$
26:            Append tuple $(new\_key, nxt, it)$ to $entries$
27:            SIFTUP($entries$, max_heap $= false$, cmp $= \lambda x.x[0]$, start $= |entries| - 1$, end $= 0$)
28:    return $merged\_heap$
```

both min-heap and max-heap configurations and incorporates user-defined comparators. Moreover, *iheap* extends functionality to include merge, replace, and remove operations, which are not directly supported by *heapq*. (ii) **queue.PriorityQueue.** While *queue.PriorityQueue* builds upon *heapq* to provide thread-safety, this comes with locking overhead that can be detrimental in high-performance simulation contexts. *iheap* is optimized for single-threaded scenarios common in simulation studies, where minimizing latency and maximizing throughput are essential. (iii) **heapdict.** The *heapdict* module offers a heap-based dictionary that efficiently supports key-based priority updates. Its design is tailored to scenarios requiring rapid reassignment of priorities based on dynamic keys. Although efficient in its domain, its specialized structure limits general applicability. In contrast, *iheap* is a general-purpose implementation, allowing arbitrary element comparison through customizable criteria, making it more adaptable to a variety of simulation and optimization problems. (iv) **fibonacci-heap.** Fibonacci heap implementations, such as those provided by the *fibonacci-heap* module, provide excellent amortized performance on operations like *decrease-key*. However, the complexity and higher constant factors associated with Fibonacci heaps often render them impractical for discrete-event simulations, where binary heaps are sufficient.

2.3 Time and Memory Complexity Comparison

Table 1 provides a comparison of the time and memory complexities of `iheap` with other modules described above. In Table 2 we provide a list of method-features available in `iheap` that are not available in existing modules. In Figure 1, we provide a simulation of the time complexities of the different functions in `iheap` using hardware with the follwoing specifications: Intel(R) Core(TM) i5-7267U CPU @ 3.10GHz, number of processors: 1, total Number of Cores: 2, L2 Cache (per Core): 256 KB, L3 Cache: 4 MB. The graphs in Figure 1 demonstrate the near-linear complexity of each of the operations in the `iheap` package.

Table 1: Time and space complexity of selected heap operations (n = number of elements; k = number of sorted inputs; m = number of arbitrary elements).

Operation variant	`iheap`		`heapq`		`PriorityQueue`		`heapdict`		`Fibonacci-heap`	
	Time	Space	Time	Space	Time	Space	Time	Space	Time	Space
Heapify, min-heap	$O(n)$	$O(1)$	$O(n)$	$O(1)$	N/A	N/A	N/A	N/A	N/A	N/A
Heapify, max-heap	$O(n)$	$O(1)$	$O(n)$[1]	$O(1)$	N/A	N/A	N/A	N/A	N/A	N/A
Push, min-heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$[2]	$O(1)$
Push, max-heap	$O(\log n)$	$O(1)$	$O(\log n)$[1]	$O(1)$	$O(\log n)$[1]	$O(1)$	$O(\log n)$[1]	$O(1)$	$O(1)$[1,2]	$O(1)$
Pop, min-heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$[2]	$O(1)$
Pop, max-heap	$O(\log n)$	$O(1)$	$O(\log n)$[1]	$O(1)$	$O(\log n)$[1]	$O(1)$	$O(\log n)$[1]	$O(1)$	$O(\log n)$[1,2]	$O(1)$
Peek, min-heap	$O(1)$	$O(1)$	$O(1)$	$O(1)$	N/A	N/A	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Peek, max-heap	$O(1)$	$O(1)$	$O(1)$[1]	$O(1)$	N/A	N/A	$O(1)$	$O(1)$	$O(1)$[1]	$O(1)$
Merge sorted	$O(n\log k)$	$O(n)$	$O(n\log k)$	$O(k)$	N/A	N/A	N/A	N/A	$O(1)$	$O(1)$
Merge unsorted	$O(n)$	$O(n)$	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Replace root	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$[3]	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$[2]	$O(1)$
Replace arbitrary 1	$O(n)$	$O(1)$	N/A	N/A	N/A	N/A	$O(\log n)$	$O(1)$	$O(\log n)$[2]	$O(1)$
Replace arbitrary m	$O(mn)$	$O(1)$	N/A	N/A	N/A	N/A	$O(m\log n)$	$O(1)$	$O(m\log n)$[2]	$O(1)$
Remove arbitrary 1	$O(n)$	$O(1)$	N/A	N/A	N/A	N/A	$O(\log n)$	$O(1)$	$O(\log n)$[2]	$O(1)$
Remove arbitrary m	$O(mn)$	$O(1)$	N/A	N/A	N/A	N/A	$O(m\log n)$	$O(1)$	$O(m\log n)$[2]	$O(1)$

[*] Please note that the `Merge`, `Replace`, and `Remove` functionality have native support for both max-heap and min-heap. This distinction is not shown in the above table as it does not affect the time or space complexity.

[1] Emulation of max-heap by negating comparator keys.

[2] Amortized time complexity for Fibonacci-heap operations.

[3] Emulation of replacing the root element by employing the `pop` functionality followed by the `push` functionality.

Table 2: Advanced feature support across Python heap modules.

Feature	`iheap`	`heapq`	`PriorityQueue`	`heapdict`	`Fibonacci-heap`
Custom key on all operations	✓	✗	✗	✗	✗
Sorted-return option on all operations	✓	✗	✗	✗	✗
Multiple algo. impl. on relevant operations	✓	✗	✗	✗	✗

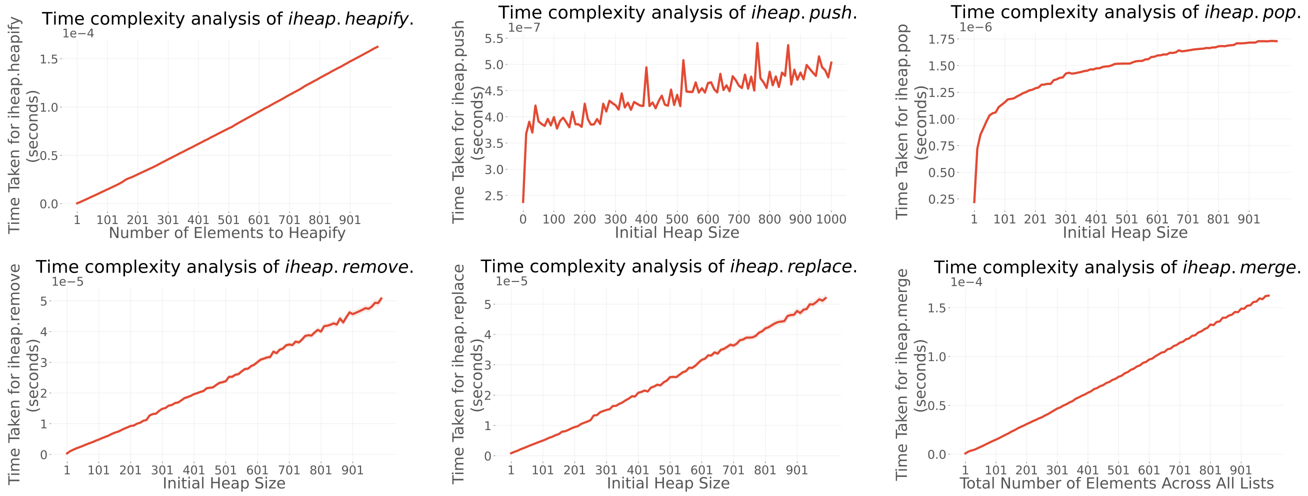


Figure 1: Run time graphs with increasing number of objects in the heap. The figures show that the practical time complexity is generally in-line with theoretical time complexities. The horizontal axis represents the number of objects. The vertical axis represents time in seconds. The perturbations in the graphs arise from hardware limitations.

3 CASE STUDIES

3.1 Marketing Campaigns

The first case study deals with the allocation of company funds from a limited budget to different marketing campaigns from a portfolio of possible campaigns. Each potential campaign in the portfolio is associated with an uncertain return. Accordingly, each campaign is characterized by two parameters, representing the scale and shape parameters of the Beta distribution. Additionally, the objective of the fund allocation process is to maximize a risk-weighted return from the investments, subject to the budget constraint. The mathematical problem formulation is as follows. Let r_i denote the expected return from campaign ID i, σ_i denote the standard deviation or risk of the campaign, B denote the total budget, λ denote the risk-aversion, n denote the total number of possible campaigns, and c_i denote the cost associated with each campaign. Let us define the decision variable as $x_i \in \{0,1\}$, which is 1 if the campaign i is selected, else 0. The problem is defined by a mean-variance objective function as in 1.

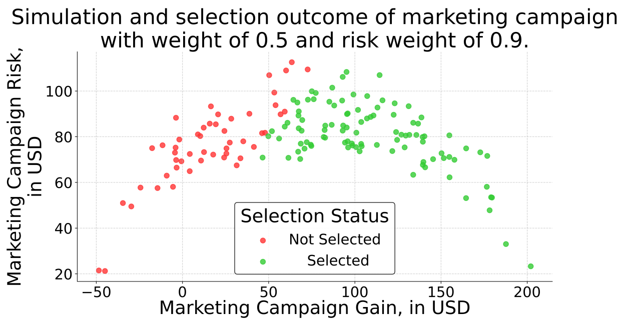
$$\max_x \left(\sum_{i=1}^{n} r_i x_i - \lambda \sum_{i=1}^{n} \sigma_i^2 x_i \right); \; s.t., \sum_{i=1}^{n} c_i x_i \leq B. \tag{1}$$

The above problem is a binary knapsack problem and is NP-hard. The problem can be solved using the Dynamic Program (DP) formulation as in 2.

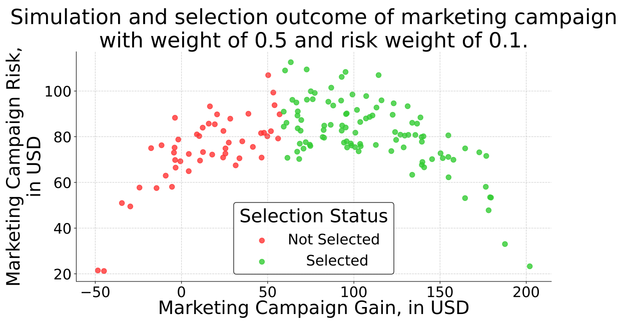
$$Sel[i][b] = \begin{cases} Sel[i-1][b] \text{ if } c_i > b, \\ \max\{Sel[i-1][b], Sel[i-1][b-c_i] + (r_i - \lambda \sigma_i^2)\} \text{ otherwise.} \end{cases} \quad (2)$$
$$Sel[0][b] = 0 \forall b \in [0, B]$$

where, $Sel[i][b]$ is the maximum risk-adjusted return using the first i investments and budget b. The DP can be time-consuming to solve for large problems. Rather, we use a branch-and-bound-based simulation algorithm that utilizes the `iheap` implementation, specifically a max-heap, which is only available in the current implementation. We construct a score that is the ratio of the risk-weighted returns and costs, $S_i = \frac{r_i - \lambda\sigma_i^2}{c_i}$, that prioritizes campaigns with high return-to-cost ratios. We score each campaign using the score function and store the campaigns in a max-heap. Then, we evaluate each item at a time starting from the root node and doing a breadth-first search. Each node indicates one campaign, and at each node, we either select the node or not. Our implementation approach begins by modeling each marketing campaign with the `MarketingCampaign` class. Each instance encapsulates: (i) **ID and Name:** Unique identifiers and labels for each campaign. (ii) **Score:** A two-element list representing parameters (sampled from a uniform distribution) that define the shape parameters for a Beta distribution. These scores model the inherent variability in campaign performance. (iii) **Cost and Revenue:** Financial measures where cost represents the marketing investment while revenue represents the expected return.

A secondary class, `MarketingCampaignSimulation`, simulates each campaign's performance. Key aspects include: (i) **Weight (`MCS_WEIGHT`):** A factor balancing emphasis between revenue and cost. A higher weight gives greater importance to revenue. (ii) **Risk Weight (`MCS_RISKW`):** A penalty factor applied to the standard deviation of the gain, reflecting the manager's risk aversion. (iii) **Number of Samples (`n`):** The total number of simulation runs to statistically assess performance. The key parameters used in the `heapify` function are: (i) **max_heap:** Set to `True`, ensuring that campaigns with the highest heuristic values appear first. **cmp:** The custom comparator extracts the key metric from each simulation object. (ii) **sort_:** When `True`, the module performs an in-place heap sort to output a fully sorted list. (iii) **bottom_up:** Enables the bottom-up construction of the heap for increased efficiency. Figure 2 shows the different campaign selections under different risk weights. We can observe that the risk tolerance affects the campaigns that get selected. This process can be used for any project selection and appraisal scenario.



(a) $MCS\_WEIGHT = 0.5$, $MCS\_RISKW = 0.9$.



(b) $MCS\_WEIGHT = 0.5$, $MCS\_RISKW = 0.1$.

Figure 2: Scatter plots of 150 marketing campaigns under different risk settings. The revenue-to-cost weight (MCS_WEIGHT) is fixed at 0.5 while the risk adjustment factor (MCS_RISKW) varies. Green markers indicate campaigns selected under a fixed budget constraint, whereas red markers denote non-selected campaigns.

3.2 Emergency Room

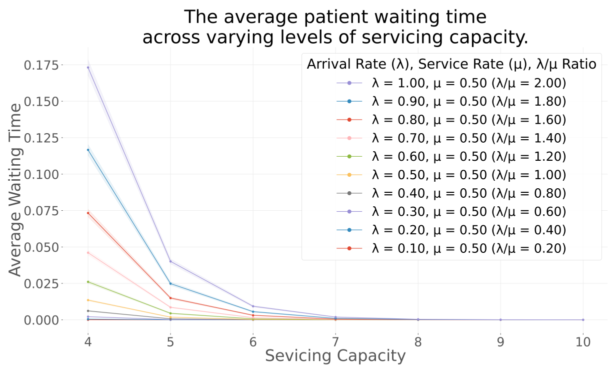
The emergency room (ER) simulation case study illustrates how the `iheap` module can be effectively employed in critical healthcare operations research. This simulation uses discrete event methods to model patient arrivals, dynamic severity updates, and service processes in an ER setting. The framework is designed to demonstrate that with minimal programming expertise, researchers and industry professionals can build high-fidelity simulation models. The `iheap` module simplifies the management of priority queues, which are central to scheduling patient treatment and discharge events under stochastic conditions.

In this study, an ER environment is modeled by representing patients as objects with attributes such as arrival time, initial severity, and signal-based severity updates. Industry professionals are provided with a modular framework: (i) **Patient Generation and Scheduling:** Patient entities are instantiated with randomly generated interarrival times (exponentially distributed) and a severity level that evolves over time. This reflects real-world variations in patient flows. (ii) **Priority Queue Management:** Two distinct event queues are maintained using the `iheap` module. One queue orders patients awaiting treatment (sorted by a custom severity-then-arrival criterion) while the other processes patients in service. The module's functions—`heapify`, `push`, and `pop`—ensure efficient event scheduling. (iii) **Dynamic Severity Updating:** The model incorporates a mechanism whereby each patient's severity is periodically updated. This dynamic aspect underscores changing clinical priorities that influence queue order.

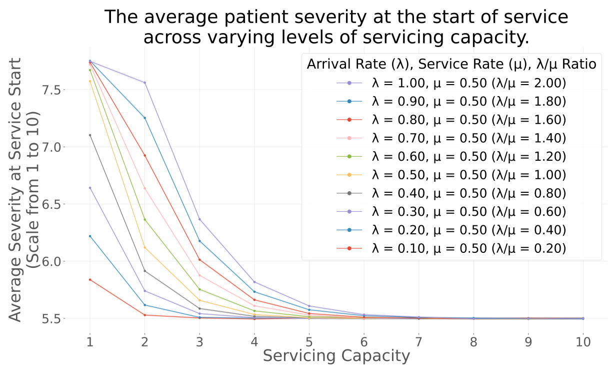
In scenarios with high arrival variability, the dynamic update of patient severity and the responsiveness of the heap-based event selection mechanism both contribute to a reduction in overall patient waiting times and improved utilization of available treatment capacity. To substantiate our findings, we provide a series of graphical analyses that delineate the impact of service capacity on emergency room performance. These figures, generated with the aid of the `iheap` module, reveal critical trends in patient waiting times, severity levels at service initiation, and overall queue dynamics under varying capacity constraints. As illustrated in Figure 3, increasing the service capacity leads to significant improvements in emergency room performance. Notably, higher capacity results in reduced waiting times and lower waiting queue lengths while simultaneously preserving patient condition through earlier and more timely service initiation and patients' initial severity at start of treatment (Refer to Figure 4). The exponential increase in waiting time for patients with lower severity under limited capacity scenarios underscores the need for dynamic resource allocation in emergency departments. Finally, Figure 5 illustrates a box plot analysis of patient severity at service start for varying capacities. The subplots clearly show that reduced capacities lead to significant patient severity deterioration at service initiation, while enhanced capacity minimizes variability and maintains baseline patient condition.

4 CONCLUSION

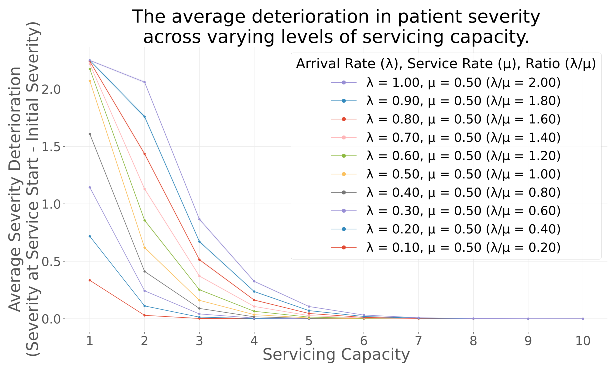
In conclusion, we present the `iheap` module developed for providing a flexible data-structure implementation for advanced simulation applications. The flexible and general heap implementation is aimed towards enabling simulation application developments that require a general comparator and scheduling interface. Accordingly, the module supports both min-heap and max-heap configurations with user-defined comparator functions. Also, the module provides a comprehensive set of utilities such as `push`, `pop`, `heapify`, `remove`, `replace`, and `merge`. The complete Python source code, documentation, and development history of the `iheap` module are hosted in a dedicated GitHub repository (iHeapPythonModule 2025)We have demonstrated the usage of these utilities in two case studies related to funding marketing campaigns and patient scheduling in emergency rooms. Our work contributes to simulation literature and applications by expanding the operational scope of heap data structures by providing a relatively more general and flexible implementation. We provide a robust and convenient tool for researchers and practitioners by streamlining the development of priority queue operations and handling arbitrary element updates efficiently. In the future, we will extend the current implementation by incorporating multi-threading support.



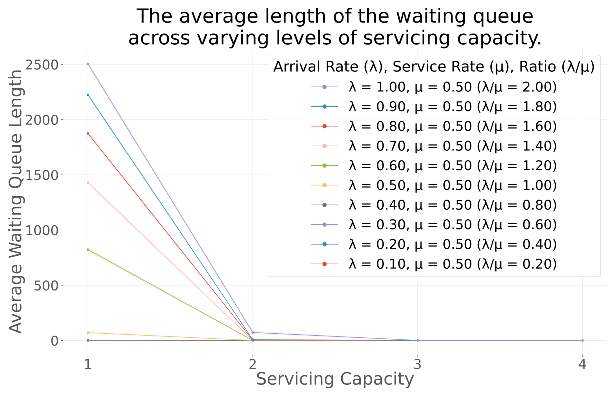
(a) Waiting time vs. service capacity.



(b) Severity at service start vs. service capacity.

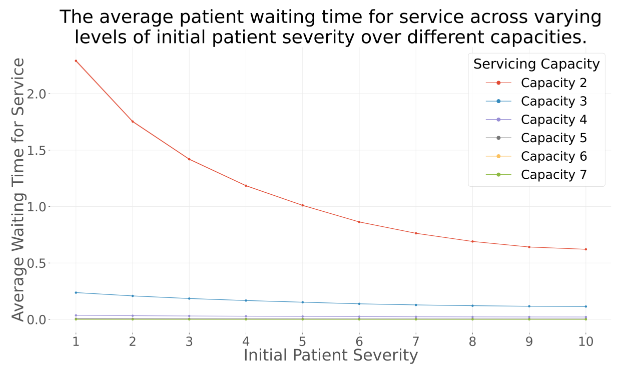


(c) Severity deterioration vs. service capacity.



(d) Waiting queue length vs. service capacity.

Figure 3: Impact of service capacity on key emergency room metrics.



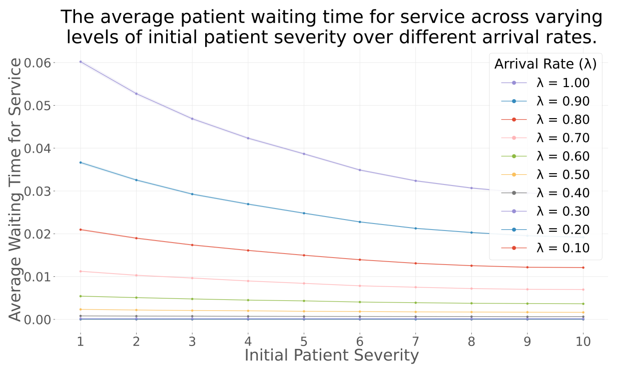


Figure 4: Impact of ED capacity (2 to 7) on average waiting time. The plot reveals that patients with lower initial severity experience markedly longer waiting times when capacity is constrained.

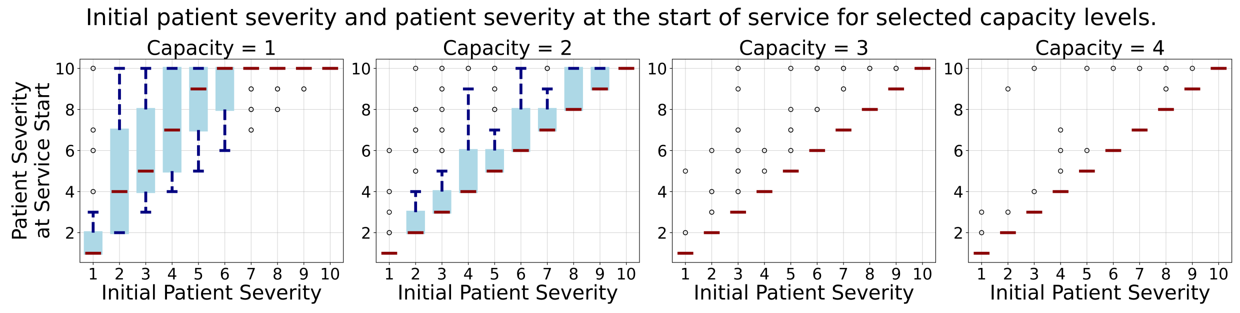


Figure 5: Box plot analysis of patient severity at service start across different service capacities. The results indicate that when capacity is lower (1 and 2), patients are more prone to an increase in severity.

REFERENCES

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. 2022. *Introduction to Algorithms*. 4th ed. Cambridge, MA, USA: MIT Press.

Gutenschwager, K., S. Völker, A. Radtke, and G. Zeller. 2012. "The Shortest Path: Comparison of Different Approaches and Implementations for the Automatic Routing of Vehicles". In *Proceedings of the 2012 Winter Simulation Conference (WSC)*, 1–12 https://doi.org/10.1109/WSC.2012.6465023.

Hannon, C., D. Jin, N. Santhi, S. Eidenbenz, and J. Liu. 2018. "Just-in-Time Parallel Simulation". In *2018 Winter Simulation Conference (WSC)*, 640–651 https://doi.org/10.1109/WSC.2018.8632357.

iHeapPythonModule 2025. "iheap". https://github.com/iHeapPythonModule/iheap. GitHub repository.

Kanezashi, H., and T. Suzumura. 2015. "Performance Optimization for Agent-Based Traffic Simulation by Dynamic Agent Assignment". In *2015 Winter Simulation Conference (WSC)*, 757–766 https://doi.org/10.1109/WSC.2015.7408213.

Liu, W., S. Gao, and L. H. Lee. 2017. "A Multi-Objective Perspective on Robust Ranking and Selection". In *2017 Winter Simulation Conference (WSC)*, 2116–2127 https://doi.org/10.1109/WSC.2017.8248226.

Ma, X., Z. Zhong, Y. Li, D. Li, and Y. Qiao. 2024. "A Novel Reinforcement Learning Based Heap-Based Optimizer". *Knowledge-Based Systems* 296:111907 https://doi.org/10.1109/ACCESS.2021.3087449.

Rinciog, A., and A. Meyer. 2021. "Fabricatio-rl: a Reinforcement Learning Simulation Framework for Production Scheduling". In *2021 Winter Simulation Conference (WSC)*, 1–12 https://doi.org/10.1109/WSC52266.2021.9715366.

Schriber, T. J., D. T. Brunner, and J. S. Smith. 2017. "Inside Discrete-Event Simulation Software: How it Works and Why it Matters". In *2017 Winter Simulation Conference (WSC)*, 735–749 https://doi.org/10.1109/WSC.2010.5679165.

AUTHOR BIOGRAPHIES

ANIRUDDHA MUKHERJEE earned an Undergraduate degree in Computer Science from UIUC, and a Graduate degree in Computer Science from Purdue University. His graduate thesis focuses on emotion diffusion in dynamic networks using transformer architectures. His work integrates computer vision, neural radiance fields (NeRF), and simulation methods. Email: mukher66@purdue.edu.

VERNON J. REGO is a Professor of Computer Sciences at Purdue University. He is an experimental scientist with interests in high-performance distributed computing (Gordon Bell Prize), parallel stochastic simulation, network protocols, security, and software systems. His work includes systems for parallel simulation (EcLiPSe/ParaSol), architectures for distributed computing (Aces/Clam), and thread systems supporting homogeneous and heterogeneous thread migration. He was an Editor for the IEEE, and was on the advisory board of the DoD Advanced Distributed Simulation Consortium. Email: rego@purdue.edu.