



YOLO

# Identificación de horarios óptimos para almorzar en el comedor de la UPCH mediante el conteo de sillas ocupadas y desocupadas con YOLO

**Mauricio Camino Quintanilla**

**Vanesa Morales Taipe**

**Luis Arenas Torres**

**Sophia Escalante Rodriguez**



# Justificación

El comedor de la UPCH es un espacio muy concurrido, en horas pico, el comedor está bastante lleno y es muy poco probable encontrar asientos libres, lo que genera frustración y pérdida de tiempo a los estudiantes.



Detectar la cantidad de sillas ocupadas y desocupadas a través del tiempo con YOLO.



Permitiría a los usuarios conocer el instante más óptimo y menos óptimo para ir a almorzar, lo que puede ayudarlo a ahorrar tiempo y mejorar su experiencia general en el comedor.

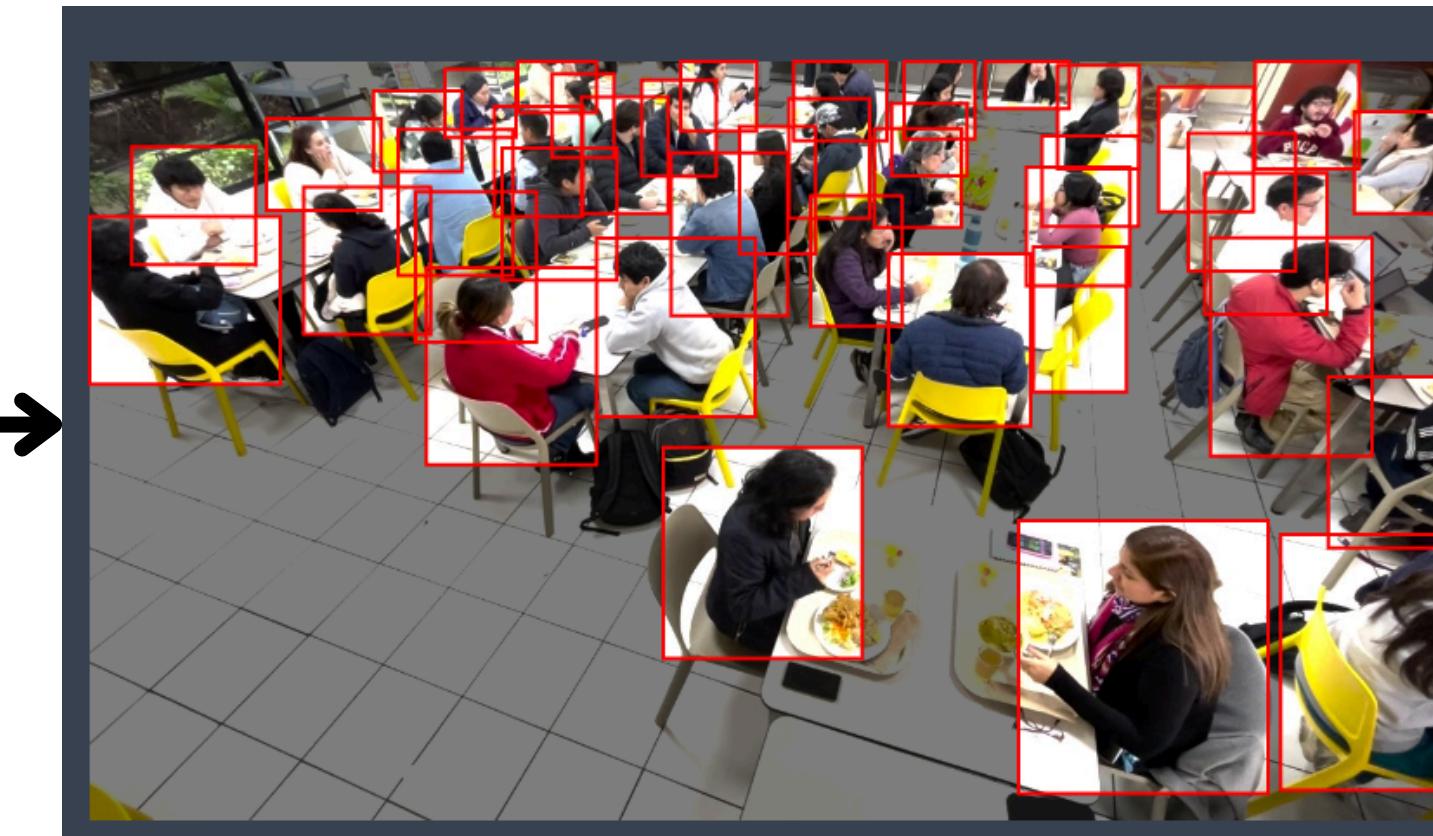
# Planteamiento

**Grabación del comedor de la  
UPCH en horas de almuerzo**



hora inicio : 11:34 am  
hora fin: 2:10 pm

**Dibujar bounding boxes en el espacio  
que ocupa una persona en una silla**



bounding boxes estáticos : representan  
la silla

**Obtenemos las  
coordenadas  
de los  
bounding  
boxes  
estáticos**

# Planteamiento

## 1. Inicialización

### a. Cargamos el modelo YOLO para la detección 'person'

```
# Cargar el modelo YOLOv8
model = YOLO('yolov9m.pt')
```

### a. Definir las coordenadas de los bounding boxes estáticos. Existen 41 sillas.

```
# Definir los bounding boxes estáticos (sillas)
static_boxes = [
    [918, 4, 77, 98.17],
    [888, 100, 98.26, 112.77],
    [684, 127, 86.54, 124.08],
    [758, 182, 134.5, 163.7],
    [744, 64, 82.37, 93.84],
    [850, 0, 79.59, 44.83],
    [895, 176, 88.38, 137.63],
    [903, 71, 91.68, 85.83],
    [761, 40, 71.52, 68.92],
    [772, 0, 66.97, 73.47],
    [1063, 167, 153.45, 207.41],
    [1058, 106, 115.74, 133.94],
    [1042, 68, 101.43, 130.69],
    [1014, 24, 90.38, 118.34],
```

# Planteamiento

## 2. Procesamiento de Cada Frame:

### a. CALCULAR SOLAPAMIENTOS Y DISTANCIAS MÍNIMAS

#### i. Para cada persona detectada en el frame:

1. Verificar si su bounding box se solapa con algún bounding box estático.

1. Si hay solapamiento, calcular la distancia entre el punto medio de la persona y el punto medio de la silla. Si solapa con otros bounding box, tambien hacer lo mismo

1. Almacenar estas distancias en una estructura de datos.

```
def calcular_overlap(box1, box2):
    x1, y1, w1, h1 = box1
    x2, y2, w2, h2 = box2

    overlap_x = max(0, min(x1 + w1, x2 + w2) - max(x1, x2))
    overlap_y = max(0, min(y1 + h1, y2 + h2) - max(y1, y2))

    return overlap_x * overlap_y > 0
```

```
def calcular_midpoint(box):
    x, y, w, h = box
    return (x + w/2, y + h/2)
```

```
def calculate_distance(point1, point2):
    return np.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)
```

```
# Inicializar diccionario para almacenar distancias
distances = {i: {} for i in range(len(static_boxes))}
```

## Procesamiento de cada Frame

### Función completa

```
def proceso_frame(frame, static_boxes):
    # Detectar personas usando YOLO
    results = model(frame)

    # Extraer bounding boxes de personas detectadas
    person_boxes = []
    for r in results:
        boxes = r.bboxes
        for box in boxes:
            if box.cls == 0 : # Clase 0 es 'person' en COCO dataset
                x1, y1, x2, y2 = box.xyxy[0]
                person_boxes.append([int(x1), int(y1), int(x2-x1), int(y2-y1)])

    # Calcular puntos medios
    person_midpoints = [calcular_midpoint(box) for box in person_boxes]
    static_midpoints = [calcular_midpoint(box) for box in static_boxes]

    # Inicializar diccionario para almacenar distancias
    distances = {i: {} for i in range(len(static_boxes))}

    # Calcular distancias para pares que se solapan
    for i, person_box in enumerate(person_boxes):
        for j, static_box in enumerate(static_boxes):
            if calcular_overlap(person_box, static_box):
                distance = calculate_distance(person_midpoints[i], static_midpoints[j])
                distances[j][i] = distance

    # Asignar personas a sillas
    assignments = {}
    assigned_people = set()

    # Ordenar las sillas por la distancia mínima a cualquier persona
    sorted_chairs = sorted(distances.items(), key=lambda x: min(x[1].values()) if x[1] else float('inf'))

    for chair_id, chair_distances in sorted_chairs:
        if chair_distances:
            # Encontrar la persona más cercana que aún no ha sido asignada
            available_people = [p for p in chair_distances.keys() if p not in assigned_people]
            if available_people:
                person_id = min(available_people, key=chair_distances.get)
                assignments[chair_id] = person_id
                assigned_people.add(person_id)

    return person_boxes, assignments
```

```
def proceso_frame(frame, static_boxes):
    # Detectar personas usando YOLO
    results = model(frame)

    # Extraer bounding boxes de personas detectadas
    person_boxes = []
    for r in results:
        boxes = r.bboxes
        for box in boxes:
            if box.cls == 0 : # Clase 0 es 'person' en COCO dataset
                x1, y1, x2, y2 = box.xyxy[0]
                person_boxes.append([int(x1), int(y1), int(x2-x1), int(y2-y1)])

    # Calcular puntos medios
    person_midpoints = [calcular_midpoint(box) for box in person_boxes]
    static_midpoints = [calcular_midpoint(box) for box in static_boxes]

    # Inicializar diccionario para almacenar distancias
    distances = {i: {} for i in range(len(static_boxes))}
```

# Planteamiento

## 2. Procesamiento de Cada Frame:

### a. ASIGNAR PERSONAS DETECTADAS A SILLAS

#### i. Para cada silla:

##### 1. Si tiene personas solapando:

###### a. Identificar la persona con la distancia mínima.

b. Asignar a esa persona a la silla. --> Cada persona se asigna solo a la silla más cercana con la que solapa y esa persona no será asignada a otra, al menos a que haya otra persona más cerca

## Función completa

```
def procesa_frame(frame, static_boxes):
    # Detectar personas usando YOLO
    results = model(frame)

    # Extraer bounding boxes de personas detectadas
    person_boxes = []
    for r in results:
        boxes = r.bboxes
        for box in boxes:
            if box.cls == 0 and box.conf > 0.35: # Clase 0 es 'person' en COCO dataset
                x1, y1, x2, y2 = box.xyxy[0]
                person_boxes.append([int(x1), int(y1), int(x2-x1), int(y2-y1)])

    # Calcular puntos medios
    person_midpoints = [calcular_midpoint(box) for box in person_boxes]
    static_midpoints = [calcular_midpoint(box) for box in static_boxes]

    # Inicializar diccionario para almacenar distancias
    distances = {i: {} for i in range(len(static_boxes))}

    # Calcular distancias para pares que se solapan
    for i, person_box in enumerate(person_boxes):
        for j, static_box in enumerate(static_boxes):
            if calcular_overlap(person_box, static_box):
                distance = calculate_distance(person_midpoints[i], static_midpoints[j])
                distances[j][i] = distance

    # Asignar personas a sillas
    assignments = {}
    assigned_people = set()

    # Ordenar las sillas por la distancia mínima a cualquier persona
    sorted_chairs = sorted(distances.items(), key=lambda x: min(x[1].values()) if x[1] else float('inf'))

    for chair_id, chair_distances in sorted_chairs:
        if chair_distances:
            # Encontrar la persona más cercana que aún no ha sido asignada
            available_people = [p for p in chair_distances.keys() if p not in assigned_people]
            if available_people:
                person_id = min(available_people, key=chair_distances.get)
                assignments[chair_id] = person_id
                assigned_people.add(person_id)

    return person_boxes, assignments
```

```
# Calcular distancias para pares que se solapan
for i, person_box in enumerate(person_boxes):
    for j, static_box in enumerate(static_boxes):
        if calcular_overlap(person_box, static_box):
            distance = calculate_distance(person_midpoints[i], static_midpoints[j])
            distances[j][i] = distance

# Asignar personas a sillas
assignments = {}
assigned_people = set()

# Ordenar las sillas por la distancia mínima a cualquier persona
sorted_chairs = sorted(distances.items(), key=lambda x: min(x[1].values()) if x[1] else float('inf'))

for chair_id, chair_distances in sorted_chairs:
    if chair_distances:
        # Encontrar la persona más cercana que aún no ha sido asignada
        available_people = [p for p in chair_distances.keys() if p not in assigned_people]
        if available_people:
            person_id = min(available_people, key=chair_distances.get)
            assignments[chair_id] = person_id
            assigned_people.add(person_id)

return person_boxes, assignments
```

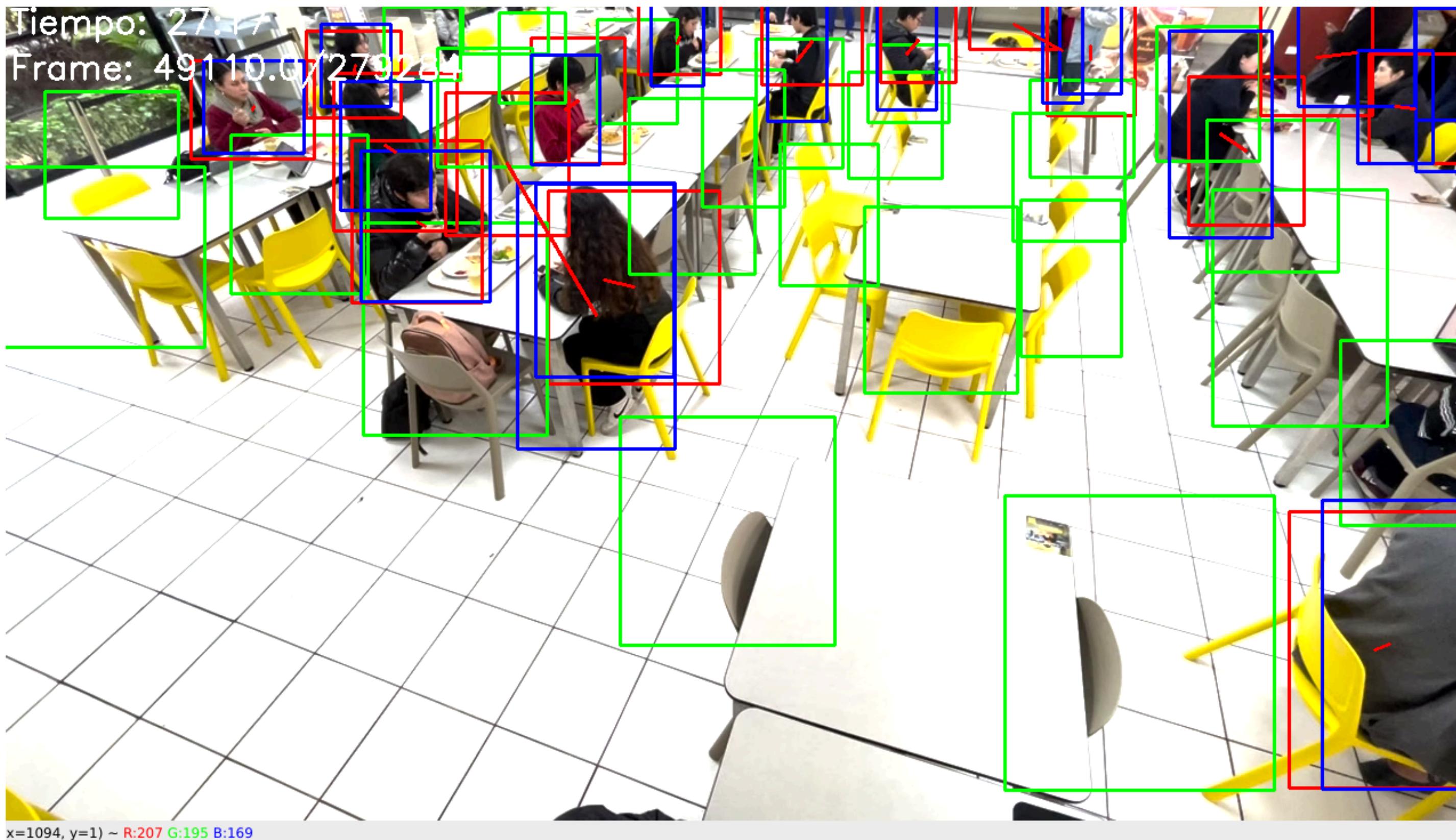
# Planteamiento

## 3. Visualización del video:

- Dibujar los bounding boxes de las sillas:
  - Verde si no está asignada.
  - Rojo si está asignada a una persona.
- Dibujar los bounding boxes de las personas detectadas en azul.
- Dibujar una línea roja entre cada silla asignada y la persona correspondiente.

```
def process_video(video_path):  
    cap = cv2.VideoCapture(video_path)  
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))  
    fps = cap.get(cv2.CAP_PROP_FPS) # Frames per second  
    current_frame = 0  
  
    while True:  
        cap.set(cv2.CAP_PROP_POS_FRAMES, current_frame)  
        ret, frame = cap.read()  
        person_boxes, assignments = proceso_frame(frame, static_boxes)  
  
        # Dibujar bounding boxes y asignaciones  
        for i, box in enumerate(static_boxes):  
            x, y, w, h = box  
            if i in assignments:  
                color = (0, 0, 255) # Rojo en BGR  
            else:  
                color = (0, 255, 0) # Verde en BGR  
            cv2.rectangle(frame, (int(x), int(y)), (int(x+w), int(y+h)), color, 2)  
  
        for i, box in enumerate(person_boxes):  
            x, y, w, h = box  
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)  
  
        for chair_id, person_id in assignments.items():  
            chair_mid = calcular_midpoint(static_boxes[chair_id])  
            person_mid = calcular_midpoint(person_boxes[person_id])  
            cv2.line(frame, (int(chair_mid[0]), int(chair_mid[1])),  
                    (int(person_mid[0]), int(person_mid[1])), (0, 0, 255), 2)  
  
        current_second = current_frame / fps  
        minutes = int(current_second // 60)  
        seconds = int(current_second % 60)  
        cv2.putText(frame, f"Tiempo: {minutes:02d}:{seconds:02d}", (10, 30),  
                   cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2)  
        cv2.putText(frame, f"Frame: {current_frame}/{total_frames}", (10, 70),  
                   cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2)  
  
        cv2.imshow("Video", frame)  
        key = cv2.waitKey(0) & 0xFF  
        if key == ord('q'):  
            break  
        elif key == ord('n'):  
            current_frame = min(current_frame + fps, total_frames - 1)  
        elif key == ord('p'):  
            current_frame = max(0, current_frame - fps)  
  
    cap.release()  
    cv2.destroyAllWindows()
```

# Resultados



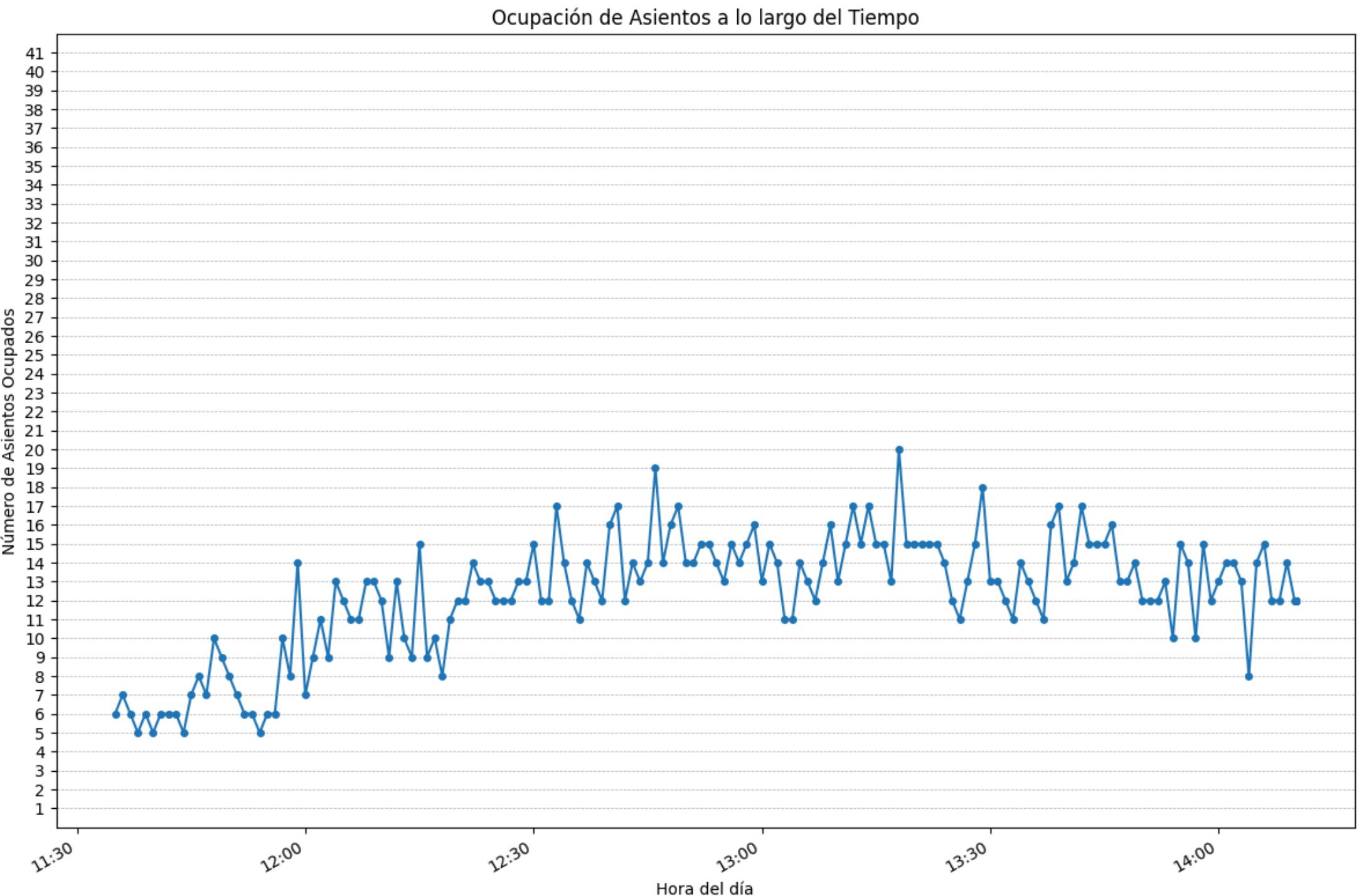
# Resultados



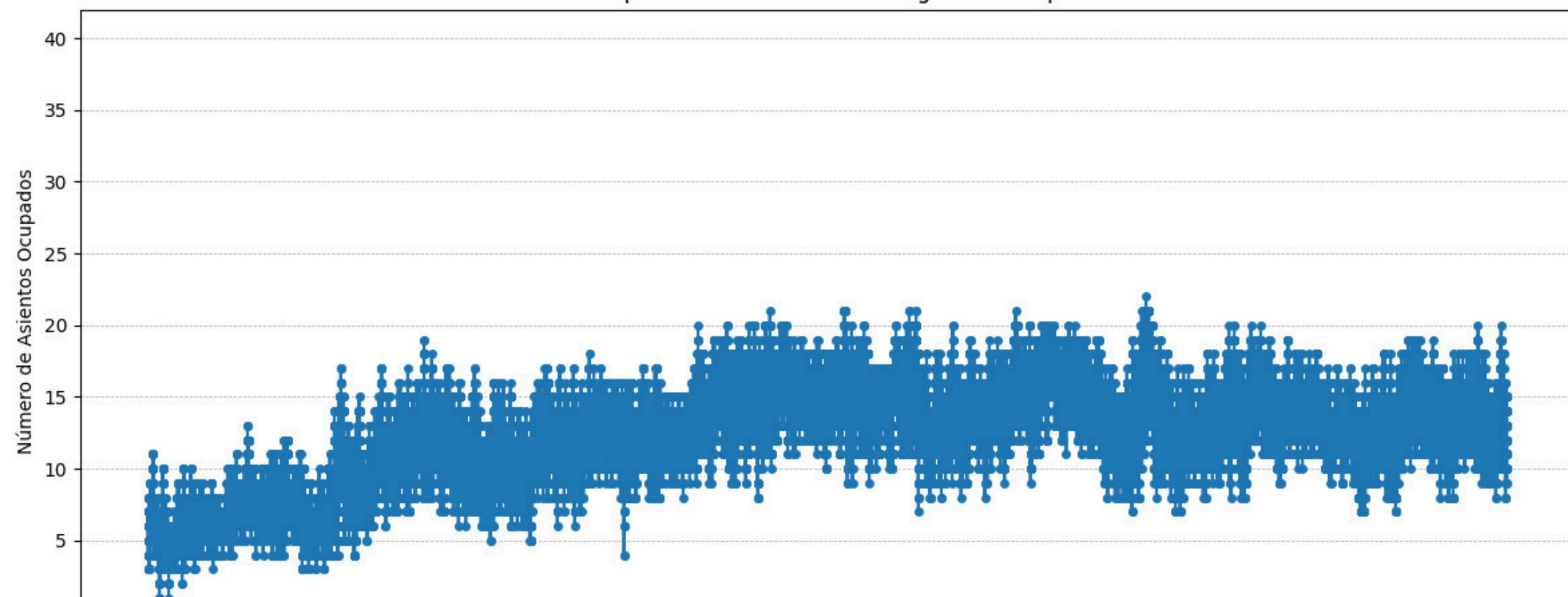
# Cálculo de la hora óptima

En la gráfica se puede observar el números de asientos ocupados en el tiempo.

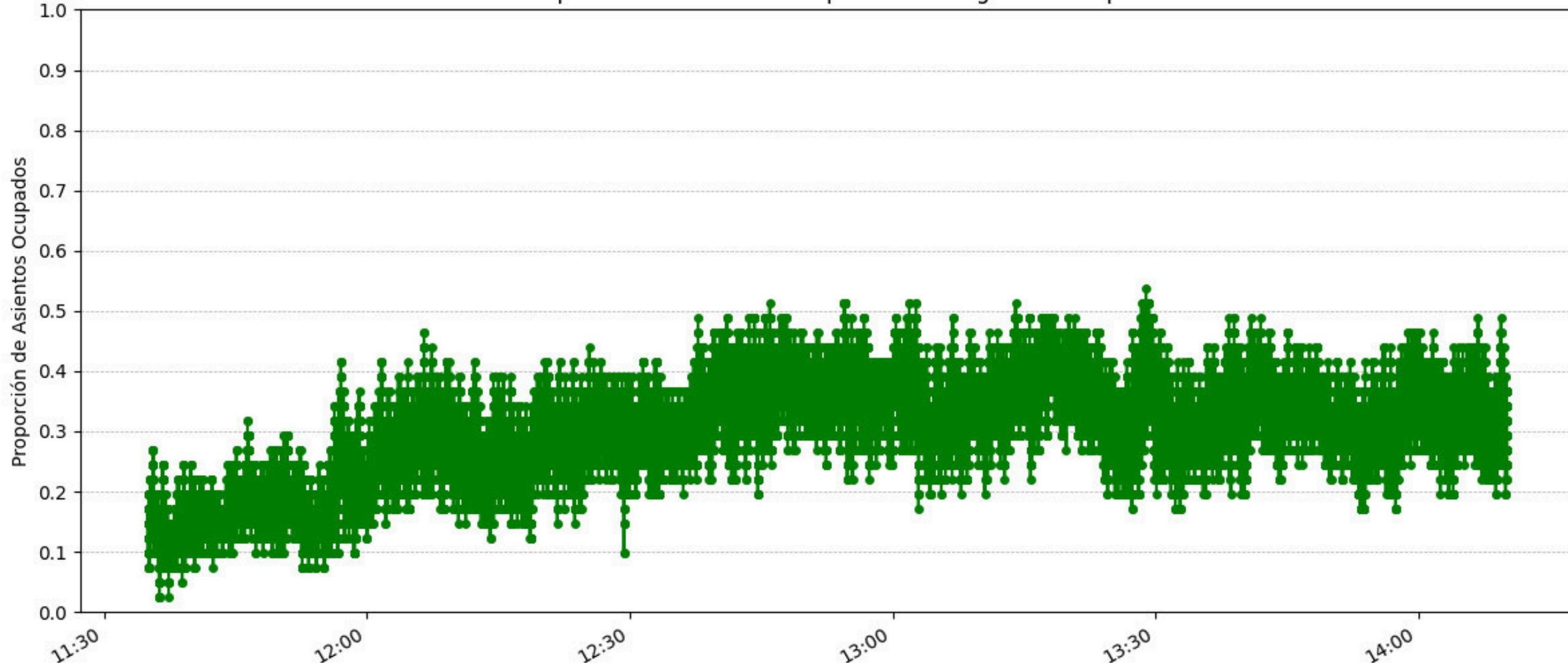
Esto ayuda a los usuarios a decidir a qué hora pueden ir a almorzar generando ahorro tiempo y mejora la experiencia de los usuarios en el comedor.

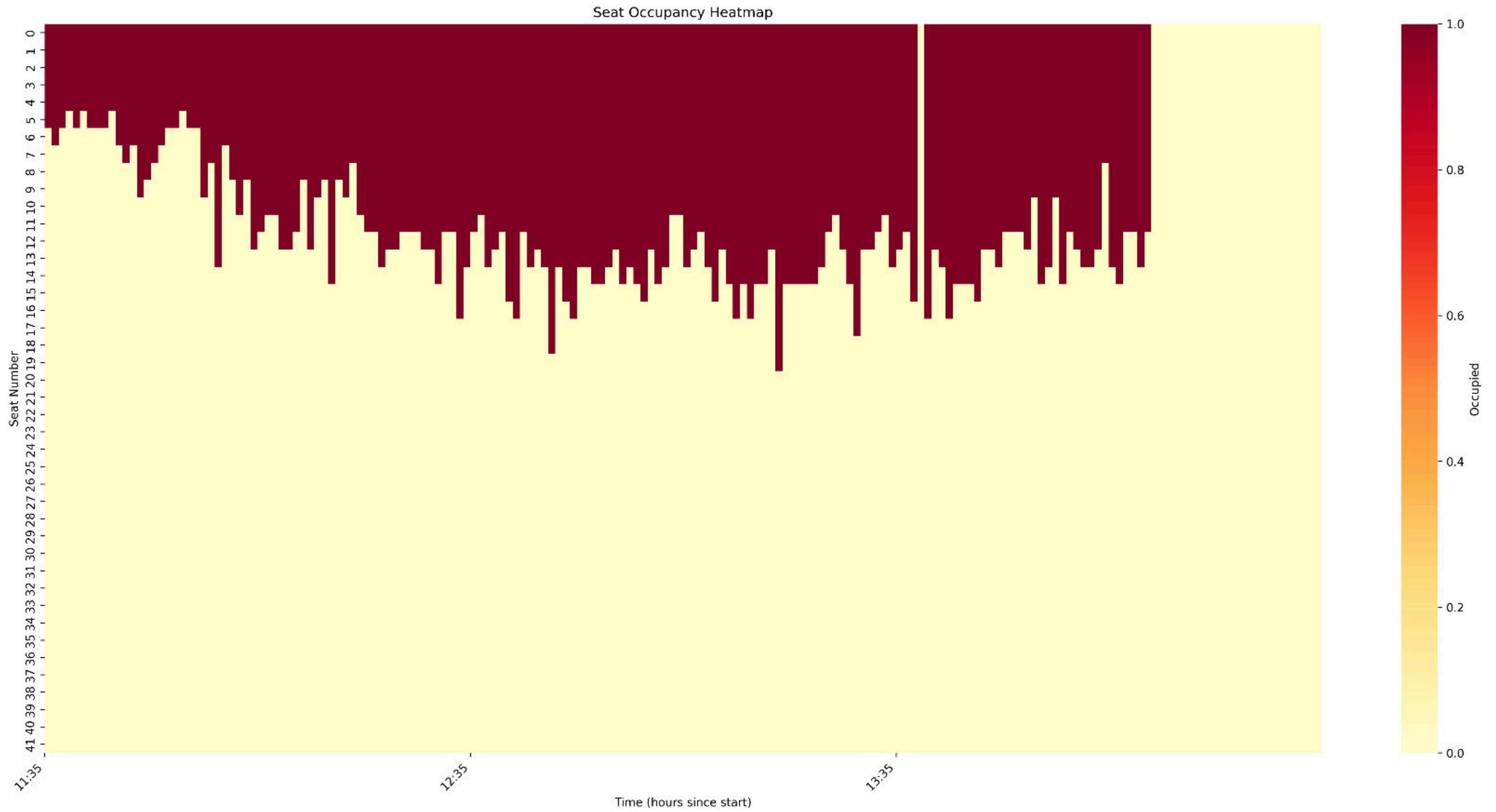


### Ocupación de Asientos a lo largo del Tiempo

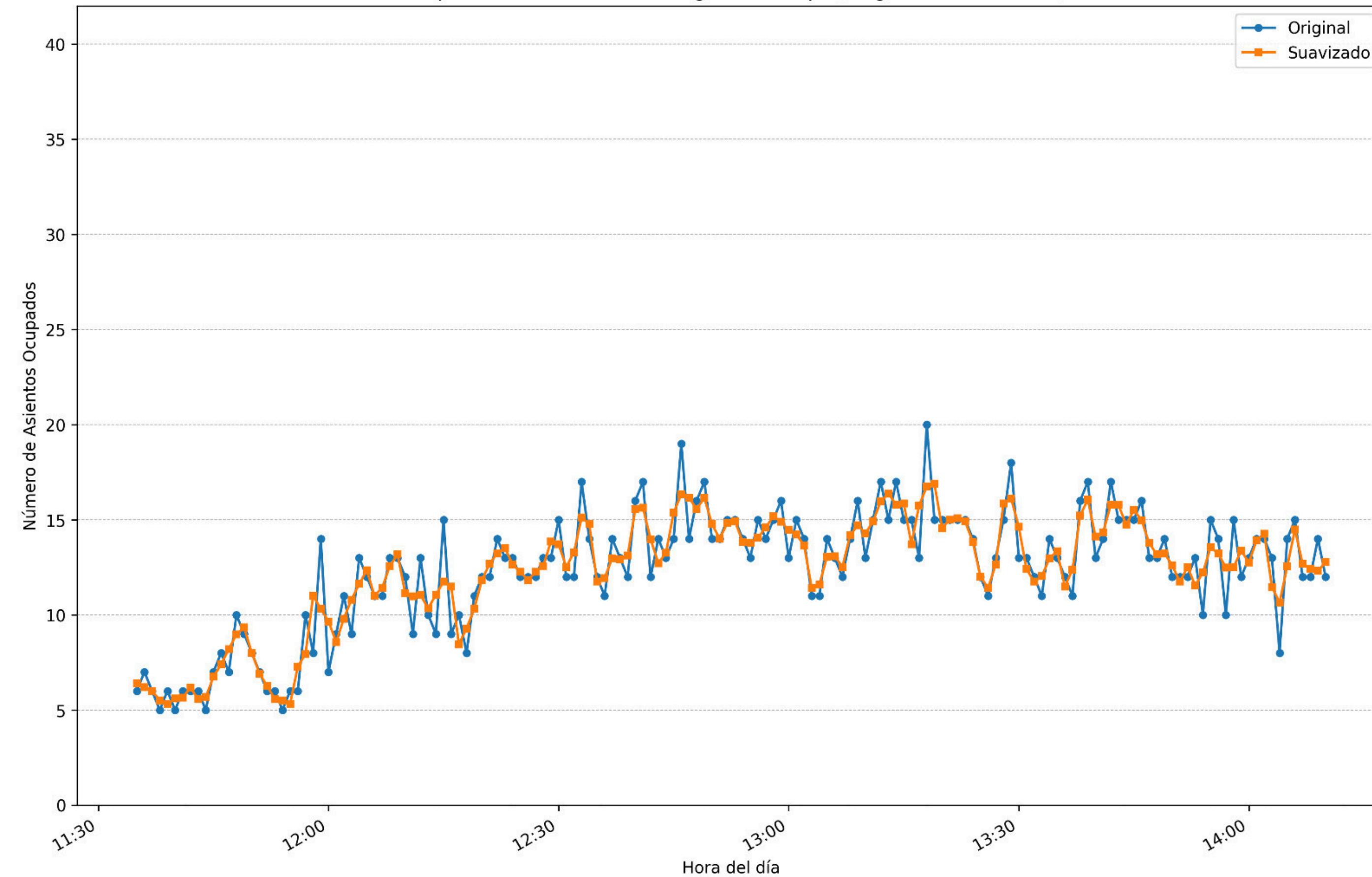


Proporción de Asientos Ocupados a lo largo del Tiempo





### Ocupación de Asientos a lo largo del Tiempo (Original vs Suavizado)



Gracias :3