UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Sistemas Inteligentes

2024/2025

Prof. Jaume Magí Jordán Prunera

Trabajo - Planificación Inteligente

Dominio emergencias

Contents

Chapter 1

Trabajo académico - Planificación Inteligente

Sé que un mi compañero prenguntó si podía escribir en inglés, y que está bien; Soy un estudiante italiano en Erasmus. Hablo español bastante bien, pero me resulta más natural escribir en inglés; sin embargo, decidí escribir en español para practicar, con la ayuda de algunos traductores cuando era necesario. Si hay algo mal escrito o poco claro, estoy a disposición para cualquier aclaración.

1.1 Introducción

He eligido el **dominio de emergencias** para mi trabajo. Resumiendo lo que está escrito en la tarea, en este dominio los puntos de interés son los siguientes:

- Localización de las víctimas, ambulancias y hospitales.
- Asignación de ambulancias a las víctimas, teniendo en cuenta la gravedad de la emergencia.
- Minimización del tiempo de respuesta, como objetivo.

La version más basica del escenario incluye una sola víctima, en vez la version ampliada mas victímas.

1.2 Implementación

1.2.1 Organización del código

El código se divide en tres ficheros:

- `emerg.py`: contiene la implementación de las funciones de los operadores y métodos.
- `scenario.py`: contiene la definición del estado inicial.
- `utils.py`: contiene funciones auxiliares, para trazar el grafico de la ciudad de Valencia y imprimir algunos mensajes.

`//TODO dockerize`

Para ejecutar el código se puede usar el comando `python3 emerg.py`, o si lo prefiere puede utilizar Docker, para evitar la instalación de las dependencias necesarias.

1.2.2 Representación del estado

El estado se representa con alguno diccionarios, uno para cada entidad:

```
state1.ambulances = [ {'label': 'A1', 'cap': 10, 'loc': 'El Carmen'},...
state1.victims = [ {'name': 'Carlos', 'sev': 5, 'loc': 'El Carmen'},...
state1.hospitals = [ {'name': 'H1', 'loc': 'Hospital La Fe'},...
```

En cada entrada del diccionario se encuentran propiedades de la entidad como la posición, el nombre y la gravedad.

Las localidades son barrios de la ciudad de Valencia, y cada barrio es caracterizado por coordenadas `(X,Y)`. Se asume que es posible viajar entre cualquier par de localizaciones y el tiempo de viaje es proporcional a la distancia euclídea entre ambas.

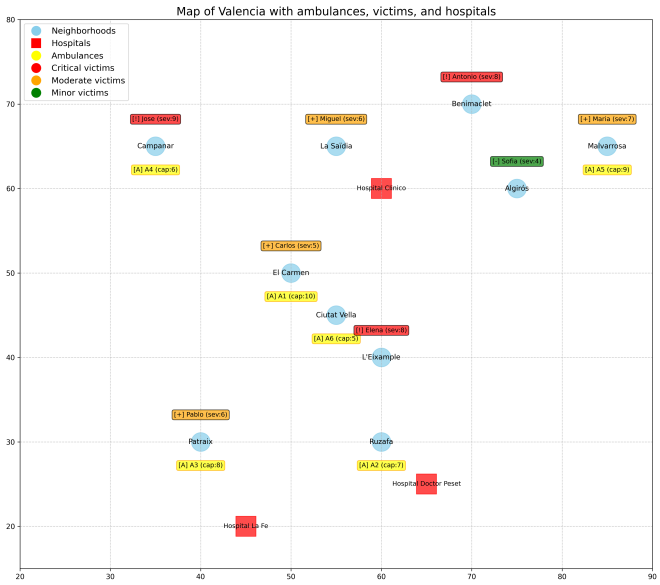

Figure 1.1: Barrios de Valencia

1.2.3 Operadores

Los operadores de pyhop representan las acciones que se pueden realizar en el dominio, y no so se pueden descomponerse. Se devuelve al nueve estado tras aplicarse el operador o `False` si el operador falla (i.e. no se aplicable). Entonces los operadores en nuestro dominio son lo siguientes, que son bastante self-explanatory:

- `op_drive(state, amb, dest)` - Mover la ambulancia a una localización. Puede parecer complicado pero es solo porque hay algunas instrucciones para trazar el camino en el mapa. No es necesario para la planificación, pero es útil para la visualización.
- `op_load(state, amb, victim)` - Cargar una víctima en una ambulancia. Aquí tenemos que verificar que la ambulancia tiene suficiente capacidad para la severidad de la víctima.
  Nel código esto control se hace también en el método `method_deliver_victim`.
- `op_unload(state, amb, victim, hospital)` - Descargar una víctima en un hospital.
- `op_treat(state, amb, victim)` - Tratar a una víctima en una ambulancia si su gravedad supera el umbral de la primera asistencia.
- `op_remove_victim(state, victim_name)` - Solo remueve la víctima de la lista de víctimas.

1.2.4 Métodos

Los métodos implementan la logica de la planificación, y determinan la secuencia de operadores que se deben aplicar para alcanzar un objetivo.

El método fundamental es `method_deliver_victim(state, victim_dict, hospital)`, que se utiliza por mover la ambulancia más cercana a la víctima, cargar la víctima, mover la ambulancia al hospital y descargar la víctima. Este método es lo más sencillo, y es invocado por los otros métodos para resolver el problema de las víctimas múltiples.

La manera más intuitiva y sencilla de implementar la gestión de multiple victímas es de llamar recursivamente el método de búsqueda para cada victima, escribiendo algo así:

```
def method_deliver_all(state):
if not state.victims:
    return []  # Base case: no more victims to deliver.

    # Take the first victim in the list.
    victim = state.victims[0]
    return [('deliver_victim', victim, 'H1'),
        ('op_remove_victim', victim['name']),
        ('deliver_all',)]
```

Listing 1.1: Esto no se encuentra en el código entregado

Sin embargo, el orden en que se entregan las víctimas afecta a la travel distance, porque el estado de las ambulancias cambia después de cada entrega, entonces, la solucción óptima no es la misma que la solucción de los subproblemas. Parece claro que el orden en que se entregan las víctimas afecta a la travel distance; las instrucciones del trabajo no piden específicamente tratar este tema, pero me intrigó y pensé que merecía la pena analizarlo.

He implementado cuatro métodos para entregar todas las víctimas, que vamos a describir más detalladamente abajo en la Sec. 1.3.1.

- `method_deliver_all_distance_priority(state)` - Entregar todas las víctimas en orden de distancia a la ambulancia más cercana.
- `method_deliver_all_severity_priority(state)` - Entregar todas las víctimas en orden de severidad.
- `method_deliver_in_order(state)` - Entregar las víctimas en el orden en que se encuentran en la lista.
- `try_all_permutations(state, max_victims=5)` - Probar todas las permutaciones de las víctimas y devolver la que minimiza la distancia total recorrida y la que minimiza el *makespan*.
  Parece complicado da leer, pero en realidad es bastante simple, solo tiene que manejar distintas variables de tracking para controlar las permutaciones y las distancias.
  Hay un umbral `max_victims` para limitar el número de permutaciones que se prueban, porque el número de permutaciones crece muy rápidamente con el número de víctimas. Sobre el umbral se generan solo `max_victims!` permutaciones, y se eligen casualmente.

1.3 Observaciones

1.3.1 Ordenar víctimas

Antes de decidir como ordinar las victímas es necesario definir un criterio de optimización. He considerado tres posibles funciones objetivo:

1. **Severidad** de emergencia - Prioritizar las víctimas más graves: esto es —más o menos— lo que se prefiere hacer en un escenario real, sin embargo, no conduce a minimizar la distancia de viaje[1]. Esta función objetivo es la más fácil de implementar, porque se puede ordenar la lista de víctimas por severidad y luego entregarlas en ese orden, no hay muchas decisiones que tomar, solo que la ambulancia más cercana a la víctima se mueve a su localización.

```
def method_deliver_all_severity_priority(state):
    if not state.victims:
        return []  # Base case: no victims left
    # Sort victims by severity (most severe first)
    victims_by_severity = sorted(state.victims, key=lambda v: v['sev'], reverse=
        True)
    victim = victims_by_severity[0] # The most severe victim
    best_hosp = # The closest hospital
    return [('deliver_victim', victim, best_hosp['name']),
            ('method_deliver_all_severity_priority',)]
```

2. **Travel distance** - Minimizar la distancia total recorrida por las ambulancias. Para hacer esto he implementado un algoritmo *greedy*, que elige primero la víctima que tiene la más cercana ambulancia.

```
def method_deliver_all_distance_priority(state):
    # ...
    for victim in state.victims:
        for amb in state.ambulances:
            if amb['cap'] >= victim['sev']:  # Ambulance can handle this victim
                dist = euclidean_distance(amb['loc']),victim['loc'] ))
                if dist < min_distance:
                    min_distance = dist
                    best_victim = victim
    # ...
```

   No estaba seguro de si esto garantizaba la solución óptima, así que intenté enumerar todas las permutaciones de víctimas y calcular la distancia total para cada permutación, y parece —salvo errores en la implementación —que la solución greedy ya proporciona la distancia óptima.
   Para hacer esto hay el método `try_all_permutations(state, max_victims=5)`, que prueba todas las permutaciones, y se puede observar como la mejor solucción que encuentra es la misma que la solucción greedy.
   No he probado todas las permutaciones de ambulancias para cada victima en cada permutación, he utilizado como criterio de elección de la ambulancia lo mismo que en los otros métodos, la ambulancia más cercana a la victima.
3. **Makespan** - Asumiendo como dice el texto que el tiempo de viaje es proporcional a la distancia euclídea entre dos localizaciones, se puede considerar que dos ambulancias distintas pueden viajar simultáneamente, entonces,

[1] Puede ocurrir, pero por casualidad.

el tiempo total de respuesta es el tiempo que tarda la última ambulancia en llegar al hospital.
Un buen algoritmo para minimizar el makespan puede ser de utilizar el mayor número posible de ambulancias, pero no he implementado este método. En vez, se observa que, como es lógico, en relación con el makespan, las permutaciones casuales pueden ofrecer a menudo mejores resultados que el algoritmo greedy utilizado para minimizar *travel distance*.

1.3.2 Estados iniciales

En `scenario.py` se definen cuatro estados iniciales

1. Una sola víctima
2. 5 víctimas, donde todas las permutaciones de las víctimas son evaluadas.
3. 10 víctimas, que no permite de evaluar todas la permutaciones, solo algunas eligidas casualmente, y en efecto observamos cómo distintas ejecuciones dan resultados diferentes.
4. 5 víctimas que tiene que fallar porque no hay ninguna ambulancia capaz de transportar a la víctima más grave.

1.4 Conclusiones

El problema de la planificación de emergencias es un problema interesante y complejo, y se puede complicar más si se van a consider multiplés funciones objetivo. En uno escenario real, la severidad de la emergencia es importante, pero no es el único factor a considerar, es necesario también minimizar el tiempo de respuesta para todas la víctimas.
En general, he leido online que el *Emergency Vehicle Dispatching Problem* es una variante del *Vehicle Routing Problem* (VRP), que es un problema NP-hard, que pero parece diferir dal nuestro problema en algunos aspectos, sobre todo que en el VRP hay un grafo con arcos que tienen un costo asociado y que determinen entre cuales nodos se puede viajar, mientras en nuestro problema se puede viajar entre cualquier par de nodos. Podemos imaginar que el nuestro es un caso particular en que el grafo es completo.

Y además, en la realidad, no siempre todos los hospitales están disponibles, el trafico afecta a la travel distance, y otros factores tecnicos (carburante, turnos de trabajo, mantenimiento de vehículos, ...) y éticos/humanos pueden influir en la solucción elegida.