PRIMER TAREA PROGRAMADA, INTELIGENCIA ARTIFICIAL

JOSÉ CASTRO

Contents

1. Búsqueda

Este conjunto de problemas conciernen sobre la implementación de varias técnicas de búsqueda. Para cada tipo de búsqueda que se le pida programar, usted tenrá un grafo (lista de nodos, una lista de aristas, y una heurística), un nodo de inicio, y un nodo objetivo.

Un grafo es un objeto de tipo Graph, definido en el archivo `search.ph`, que contine listas `.nodes`, aristas `.edges` y una heuristica `.heuristic` (no va a necesitar tener acceso al diccionario directamente. Todos los grafos usados por el tester se definen en `graphs.py`

Un nodo solo es una hilera representando el nombre del nodo.

Una arista es un objeto con atributos `.name` (nombre, una hilera), `node1` (una hilera identificando el nodo en un extremo de la arista), `.node2` (una hilera identificando el nodo al otro extremo de la arista), y `.length` (un número).

Va a necesitar los siguientes métodos:

- `graph.get_connected_nodes(node)`: dado el nombre de un nodo, retorna una lista de todos los nombres de nodos que estan connectados por una arista al nodo indicado.
- `graph.get_edge(node1,node2)`: Dados los nombres de dos nodos, retorna la arista que los conecta, o `None` en caso que no exista.
- `graph.get_edge(start,goal)`: Dado el nombre de un nodo de inicio en un grafo, y el nombre del nodo objetivo, retorna el valor de la heurística desde el inicio hasta el objetivo. Si esa heurística no fue suministrada cuando se creo el grafo entonces retorna el valor de 0.

También puede que quiera utilizar los siguientes métodos:

- `graph.are_connected(node1,node2)`: Retorna `True` si y solo si hay una arista que conecta directamente a los nodos `node1` y `node2`, `Falso` en caso contrario.
- `graph.is_valid_path(path)`: Dado una lista ordenada de nombres de nodos, retorna `True` si y solo si hay una arista entre cada par de nodos que se encuentran adyecentes en la lista, `Falso` en caso contrario.

Puede utilizar listas, stacks o colas, como esta documentado en http://docs.python.org/tut/noc
sin embargo NO debe importar otros módulos (tal como `deque`, porque puede confundir al tester.

Puede que necesite ordenar listas en Python, Python tiene funcionalidad de sort ya incluida, documentada en

http://wiki.python.org/moin/HowTo/Sorting

2. El Programa

2.1. **La Agenda.** Distintas formas de búsqueda exploran los nodos de un grafo en distinto orden, y nosotros mantendremos la lista de nodos que quedan por explorar en una variable que llamaremos `agenda`. Algunas técnicas agregaran caminos al inicio de la agenda, tratandola como una Pila, mientras que otras los agregaran al final, tratandola como una cola. Otras agendas se organizan por su valor de heurística, otras por la distancia del camino, y otras por la profundidad del árbol de búsqueda. Su trabajo será demostrar su conocimiento de técnicas de búsqueda implementando distintos tipos de busquedas haciendo pequeñas modificaciones a como es que la agenda es accedida y actualizada.

2.2. **Extendiendo el camino en la agenda.** En esta tarea un camino consiste de una lista de nombres de nodos. Cuando se trata de extender un camino, un camino es seleccionado de la agenda. El último nodo en el camino es el identificado como el nodo a ser expandido. Los nodos que conectan al nodo expandido, sus nodos adyecentes, son las extensiones posibles al camino. De las posibles extensiones, los siguientes nodos NO son agregados.

- nodos que ya aparecen en el camino
- nodos que ya han sido extendidos (si es que esta utilizando un conjunto de nodos extendidos. esto depende del algoritmo)

Como ejemplo, si el nodo A conecta con los nodos S, B y C, entonces el camino ['S','A'] es extendido a los nuevos caminos ['S','A','B'], y ['S','A','C'], pero no en ['S','A','S'].

Los caminos que usted crea deben ser objetos nuevos. Si usted trata de extender un camino modificando (o mutando) el camino existente, por ejemplo utilizando la funcion de listas `.append()`, va a tener problemas en sus algoritmos.

2.3. **Conjunto de Nodos Extendidos.** Un conjunto extendido, a veces llamado "lista extendida", "conjunto visitado" o "lista cerrada", consiste de nodos que ya han sido extendidos, y le permite a su algoritmo evitar expandir el mismo nodo multiples veces, a veces logrando una aceleración significativa de la búsqueda. Usted estara implementado tipos de búsqueda que utilizan conjuntos extendidos. Note que un conjunto de nodos extendidos en un conjunto, asi que si esta utilizando una lista para representarlo, debe tener cuidado de no repetir el nodo en la lista. Python le ofrece una forma de representar conjuntos que le permite evitar este

problema. El punto principal es revisar que los nodos no estan en el conjunto antes de extenderlos, y de poner nodos en el conjunto extendido cuando decida extenderlos. .

2.4. **Devolviendo el resultado de la búsqueda.** El resultado de la búsqueda es un camino que consiste de una lista de nombres de nodos, ordenada desde el nodo inicial, siguiendo aristas existentes, hasta el nodo objetivo. Si no hoy camino encontrado, la búsqueda debe retornar la lista vacia `[]`.

2.5. **Terminando la búsqueda.** Búsquedas subóptimas como profundidad primero, ancho primero, hill-climbing y Beam pueden terminar cuando:

- Cuando encuentran un camino al objetivo en la agenda
- La primera vez que camino al objetivo es removido de la agenda.

(Esto es porque la agenda esta ordenada por largo de camino (o heurística de largo de camino), así que un camino al objetivo no es necesariamente el mejor cuando es agregado a la agenda, pero cuando es eliminado de ella sí hay garantía de que es el camino mas corto.

Por asuntos de consistencia, en esta tarea usted debe implementar sus búsquedas para que terminen siempre:

- **La primera vez que un camino al objetivo es removido de la agenda**

Búsquedas óptimas tales como costo-primero (branch and bound) y A^* **siempre** deben terminar cuando:

- La primera vez que un camino al objetivo es removido de la agenda

3. Las Preguntas

3.1. **Selección múltiple.** Esta sección contiene las primeras preguntas de la tarea, se encuentran en `lab2.py` y le permiten revisar su conocimiento de distintos tipos de búsqueda. Usted debería, por supuesto, tratar de contestarlas antes de revisar las respuestas en el tester.

3.2. **Calentamiento.** Dos problemas son de calentamiento. Debe hacerlos para entender la estructura de los programas y las preguntas, Estos problemas son la implementación de profundidad primero y ancho primero. Sus estratégias de b'usqueda deben permitir **backtracking** en la búsqueda.

Usted debe implementar las siguientes funciones:

```
def bfs(graph, start, goal):

def dfs(graph, start, goal):
```

Las entradas a las funciones son:

- `graph`: El grafo
- `start`: El nombre del nodo desde donde usted quiere empezar a recorrer

- `goal`: El nombre del nodo al cual se quiere llegar

Cuando un camino al objetivo es encontrado, retorne el resultado tal como se explica anteriormente.

3.3. **Búsqueda informada.**

3.3.1. *Hill Climbing.* Hill climbing en este contexto es muy similar a profundidad primero. Solo hay una pequeño modificacion al acomodo de los caminos cuando son agregados a la agenda. Para esta parte, debe implementar la siguiente función:

```
def hill_climbing(graph, start, goal):
```

El procedimiento de hill climbing que usted define *debe* utilizar backtracking, para que sea consistente con los otros métodos, aún cuando hill-climbing típicamente no se implementa con backtracking. Hill-climbing no utiliza un conjunto extendido.

3.3.2. *Beam Search.* Beam search o búsqueda por haz (beam) es muy similar a ancho primero, pero hay una modificación a cuales caminos estan en la agenda. La agenda puede tener en todo momento no mas de **k** caminos de largo textbfn; **k** se conoce como el ancho del haz (beam width), y **n** corresponde al nivel del grafo de búsqueda. Tiene que ordenar los caminos por la heurística para asegurarse que los mejores **k** caminos a cada nivel estan en su agenda. Puede que le sirva utilizar un array o diccionario para darle seguimiento a caminos los distintos largos tiene. Beam search NO usa un conjunto extendido, y NO utiliza backtracking a los caminos que son eliminados en cada nivel.

Recuerde que **k** es el número de caminos que debe mantener en un *nivel completo*, no el número de caminos que debe mantener al extender cada nodo.

También recuerde que beam search ordena los caminos *solo* por el valor de la heurística, no el largo del camino (como lo hace branch and bound), o el largo del camino + la heurística (como lo hace A^*).

En esta parte debe implementar la función:

```
def beam_search(graph, start, goal, beam_size):
```

3.3.3. *Búsqueda Óptima.* Las técnicas de búsqueda que ha implementado hasta ahora no han tomado en cuenta la distancia de las aristas y corresponden a búsqueda no informada, esto es, búsqueda que sólo utiliza la estructura del grafo para encontrar los caminos y no aprovecha conocimiento relacionado con el problema. Los problemas que siguen se refieren a encontrar un camino que es la distancia más corta desde el nodo incial hasta el objetivo. Los tipos de búsqueda que garantizan esto son branch and bound (muy parecido a Dijkstra) y A^*.

Como este tipo de problema requiere conocimiento del largo del camino, debe implementar la siguiente función:

```
def path_length(graph, node_names):
```

Esta función toma un grafo y una lista de nombres de nodos que conforman un camino en el grafo, y calcula el largo de este camino utilizando los valores de "LENGTH" que se encuentran en las aristas. Puede asumir que el camino es valido (hay aristas en el grafo para cada par de nodos adyecentes en la lista `node_names`. Si sólo hay un nodo en el camino, su función debe retornar 0.

3.3.4. *Branch and Bound.* Ahora que tiene una manera de calcular la distancia de un camino, esta parte debería ser fácil de completar. Puede que encuentre útil el procedimiento de listas `remove`, y/o la palabra reservada de Python '`del`'. Para esta parte complete el siguiente procedimiento:

```
def branch_and_bound(graph, start, goal):
```

Branch and bound no utiliza un conjunto extendido.

3.3.5. A^*. Ha utilizado estimados de heurística para acelerar la búsqueda y largos de aristas para calcular caminos óptimos. Ahora combinemos las dos ideas para implementar el método de búsqueda A^*. En A^*, el camino con el valor mínimo de heurística + largo de camino es el que es tomado de la agenda para ser extendido. A^* siempre utiliza un conjunto extendido – asegurese de utilizar uno. (Nota: Si la heur'istica no es consistente, entonces el utilizar un conjunto extendido puede impedir que A^* encuentre una solución óptima).

```
def a_star(graph, start, goal):
```

3.4. **Heurísticas de Grafos.** Una heurística da un valor aproximado de un nodo al objetivo. Sabemos que para que una heurística sea admisible, el valor de la heurística de un nodo debe ser menor o igual que la distancia del camino mas corto desde ese nodo hasta el objetivo. Para que una heurística sea consistente, para cada arista en el grafo, la distancia (LENGTH) de la arista debe ser mas grande o igual que el valor absoluto de la diferencia de las dos heuristicas de sus nodos.

Para esta parte, complete las siguientes funciones, que deben retornar `True` si y solo si la heurísticas dadas para el objetivo son admisibes o consistentes respectivamente, y `False` en caso contrario.

```
def is_admissible(graph, goal):


def is_consistent(graph, goal):
```

Responda las preguntas

- Cuantas horas le tomo resolver estos probemas?
- Cuales partes de los problemas encontró interesantes?
- Cuales partes de los problemas encontró aburridas?

4. Entregables

Debe entregar su archivo `lab2.py` en el tecDigital