Tarea 1

Parte 1 : Crítica al artículo Alan Turing, Computing machinery and Intelligence, Mind, Octubre, 1950, 59:433-460

En el artículo Alan Turing plantea la pregunta ¿puede una máquina pensar? Esta pregunta en un principio puede parecer trivial, sobre todo ante nuestras concepciones y creencias. Sin embargo, como Turing en el artículo nos hace ver, esta pregunta tiene muchas implicaciones en diferentes ámbitos y no sólo le corresponde a uno solo resolverla.

Más allá de responder esta pregunta, en su artículo Turing ciento las bases para analizar esta cuestión. Más allá de reinterpretar la pregunta a través del juego de la imitación, el análisis se centra principalmente en la posibilidad. Aunque no lo dice explícitamente, la mayoría de las argumentaciones que responde en el artículo se enfocan en el contexto de la computación contemporánea al escrito. Admiro la valentía con que Turing se adelanta a su época y logra ver más allá de las limitaciones técnicas de su momento para poder analizar esta pregunta a futuro. Como él mismo indica que en unos años esta pregunta se podrá realizar sin recibir burlas.

En un principio me pareció poco seria la argumentación que plantea Turing. Pero al terminar de leer por completo el artículo me percaté que más que tratar de contestar las opiniones en contra, plantea un primer paso al debate en cada uno de los aspectos que le cuestionaban. Esto nos da un punto de partida con el cual el debate, cada día más vigente, se debe de seguir.

Dejando un lado los aspectos filosóficos de la pregunta, enfatizaría la afirmación de qué una computadora digital teórica es suficiente, universal. Esto nos lleva a que sea suficientemente interesante para ser objeto de estudio. Y ante esta aseveración, creo que Turing podría haber ido un paso más allá al ver concluido que la verdadera pregunta no es si las máquinas pueden pensar, si no, cuando podrán.

Parte 2 : Implementación de los algoritmos

Se implementó árbol de búsqueda en el paquete `search.algorithms.tree_search`, el cual puede recibir una estrategia de las siguientes:

1. Búsqueda en anchura
2. Búsqueda en profundidad
3. Búsqueda iterativa
4. Búsqueda de costo uniforme
5. Búsqueda voraz
6. Búsqueda A*

Estas estrategias se implementaron en el módulo `search.strategies`.

La gráfica vista en clase se codifico como lista de adyacencia en el archivo `Rumania.txt`

La ejecución de los algoritmos en el problema de las ciudades de Rumanía se localiza archivo `tree_search.py`, el cual se puede ejecutar con el siguiente comando:

```
python3 tree_search.py
```

El resultado se trascribe a continuación.

```
---dfs---
0 - Arad
-1 - Sibiu
-2 - Fagaras
-3 - Bucharest
[('Arad', 140), ('Sibiu', 99), ('Fagaras', 211), ('Bucharest', 0)]
Cost 450: Arad->Sibiu->Fagaras->Bucharest


---bfs---
0 - Arad
1 - Sibiu
1 - Timisoara
1 - Zerind
2 - Fagaras
2 - Oradea
2 - Rimnicu Vilcea
2 - Lugoj
3 - Bucharest
[('Arad', 140), ('Sibiu', 99), ('Fagaras', 211), ('Bucharest', 0)]
Cost 450:Arad->Sibiu->Fagaras->Bucharest


---iterative---
0 - Arad
0 - Arad
-1 - Sibiu
-1 - Timisoara
-1 - Zerind
0 - Arad
-1 - Sibiu
-2 - Fagaras
-2 - Oradea
-2 - Rimnicu Vilcea
-1 - Timisoara
-2 - Lugoj
-1 - Zerind
0 - Arad
-1 - Sibiu
-2 - Fagaras
-3 - Bucharest
[('Arad', 140), ('Sibiu', 99), ('Fagaras', 211), ('Bucharest', 0)]
Cost 450:Arad->Sibiu->Fagaras->Bucharest


---ucs---
0 - Arad
```

```
75 - Zerind
118 - Timisoara
140 - Sibiu
146 - Oradea
188 - Lugoj
220 - Rimnicu Vilcea
239 - Fagaras
258 - Mehadia
317 - Pitesti
333 - Dobreta
366 - Craiova
450 - Bucharest
[('Arad', 140), ('Sibiu', 99), ('Fagaras', 211), ('Bucharest', 0)]
Cost 450:Arad->Sibiu->Fagaras->Bucharest


---greedy---
0 - Arad
253 - Sibiu
178 - Fagaras
0 - Bucharest
[('Arad', 140), ('Sibiu', 99), ('Fagaras', 211), ('Bucharest', 0)]
Cost 450:Arad->Sibiu->Fagaras->Bucharest


---A*---
0 - Arad
393 - Sibiu
413 - Rimnicu Vilcea
415 - Pitesti
417 - Fagaras
418 - Bucharest
[('Arad', 140), ('Sibiu', 80), ('Rimnicu Vilcea', 97), ('Pitesti', 101),
('Bucharest', 0)]
Cost 418:Arad->Sibiu->Rimnicu Vilcea->Pitesti->Bucharest
```

Por cada algoritmo se muestran los nodos visitados y la prioridad que tenían en la cola. Se imprime el resultado del la implementación del árbol de búsqueda con la rama que lleva de la raíz a la solución. Finalmente se muestra el costo total y de forma gráfica la solución encontrada.

Parte 3: Análisis del algoritmo de búsqueda voraz

¿Es completo?

Sí, ya que recorre todos los nodos.

¿Es óptimo?

No. Un contra ejemplo es el ejemplo de las ciudades de Rumanía, en la salida podemos ver que la solución que encuentra no es óptima. En el peor de los casos, resulta ser equivalente a búsqueda en profundidad pero mal guiado, y DFS no es óptimo.

Complejidad en tiempo

Sea `b` el factor de ramificación, y `m` la profundidad máxima.

`O(b^m)`, al igual que DFS en el peor de los casos recorre todo el árbol.

Complejidad en espacio

`O(b\*m)`, al igual que DFS solo se almacena en la frontera los hijos de una rama a la vez.

Parte 4: Problema de la mochila

Para la solución se implementó búsqueda voraz con la heurística peso/valor. La cola de prioridad implementada retorna el elemento con prioridad mínima. Así, a menor peso o mayor valor se dará preferencia.

El algoritmo está implementado en el archivo `knapsack_problem.py` y se puede ejecutar verificado que en la linea 43 se lea la ruta del archivo a analizar.

```
43   (size, data) = read_problem('./ks_19_0')
```

```
python3 knapsack_problem.py
```

Para el archivo `ks_19_0`, se trascribe la salida en consola.

```
Bag size: 31181

0 - 13:[3878, 9656]
2.600109110747409 - 17:[1833, 4766]
3.0494752623688157 - 16:[667, 2034]
2.5898123324396782 - 15:[1865, 4830]
2.5948446794448117 - 12:[1513, 3926]
3.015965166908563 - 11:[689, 2078]
2.869565217391304 - 10:[805, 2310]
3.5576323987538943 - 1:[321, 1142]
TotalWeight: 22228, TotalValue: 30742, Choses: 13->17->16->15->12->11->10-
>1
```

Parte 5: Bonus

El algoritmo está implementado en el archivo `knapsack_problem.py` y se puede ejecutar verificado que en la linea 43 se lea la ruta del archivo a analizar.

```
43   (size, data) = read_problem('./ks_10000_0')
```

```
python3 knapsack_problem.py
```

Para el archivo `ks_19_0`, se trascribe la salida en consola.

```
Bag size: 1000000

0 - 3003:(92264, 83877)
0.9490509163904092 - 9999:(152937, 145145)
1.077395277688032 - 9998:(151663, 163401)
0.9373150723764639 - 9997:(56441, 52903)
0.9914672861956139 - 9996:(164426, 163023)
0.9482118322427936 - 9995:(73511, 69704)
0.9790428398083212 - 9994:(194492, 190416)
1.1089879503354885 - 9993:(114609, 127100)
1.0622524052065647 - 9952:(1767, 1877)
0.9453376205787781 - 9770:(933, 882)
1.0806451612903225 - 9722:(124, 134)
1.0785498489425982 - 9710:(662, 714)
0.9544554455445544 - 9287:(505, 482)
0.9459459459459459 - 9117:(74, 70)
0.956140350877193 - 7662:(228, 218)
0.9545454545454546 - 7498:(22, 21)
1.125 - 6113:(8, 9)
TotalWeight: 916108, TotalValue: 999976, Choses: 3003->9999->9998->9997-
>9996->9995->9994->9993->9952->9770->9722->9710->9287->9117->7662->7498-
>6113
```

Repositorio

https://github.com/hyfi06/pcic-ai241/tree/main/Tarea%201