Universidad Nacional Autónoma de México

Facultad de Ciencias

Proyecto 2: Juego de Othello

Galeana Araujo Emiliano 314032324
Fernández Del Valle Diego Marcial 314198039
Reyes Granados Naomi Itzel 314141341
Vázquez Rodríguez Jésica 720034837

Inteligencia Artificial

Gustavo De la Cruz Martínez
Estefanía Prieto Larios
Osvaldo Baruch Acevedo Badillo
Rodrigo Eduardo Colín Rivera

October 2, 2019

Contents

1 Introducción

1.1 Árbol de busqueda

Opciones, siempre que queremos tomar una decisión, ya sea en un juego o en la vida, pensamos en las diferentes posibilidades y las repercusiones que cada una de estas tiene, con el objetivo de encontrar la mejor opción. En inteligencia artificial se ocupan los árboles de búsqueda para modelar esto, el cual representa todas las posibles transformaciones del sistema aplicando todas las acciones posibles recursivamente. Entonces un árbol de búsqueda consta de:

1. Nodo raíz : estado inicial.
2. Hijos de un nodo: Posibles sucesores (nodos correspondientes a estados resultantes de la aplicación de una acción al nodo padre).

Bueno, ya tenemos el árbol con todas las posibilidades,ahora, ¿cómo tomamos una decisión?

1.2 Recorridos

Lo más natural para encontrar la decisión correcta es recorrer el árbol de decisiones buscando el camino para el estado óptimo. Para recorrer un árbol conocemos diferentes formas:

1. BFS: Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

```
1  BFS grafoG, nodo_fuentes
       inputs : Árbol de busqueda
       output: Nodo objetivo
2      foreach uϵV[G] do
3          estado[u] = NO_VISITADO;
4          distancia[u] = INFINITO;
5          padre[u] = NULL;
6      estado[s] = VISITADO;
7      distancia[s] = 0;
8      padre[s] = NULL;
9      CrearCola(Q);
10     Encolar(Q, s);
11     while !vacia(Q) do
12         u = extraer(Q);
13         foreach vϵadyacencia[u] do
14             if estado[v] == NO_VISITADO then
15                 estado[v] = VISITADO;
16                 distancia[v] = distancia[u] + 1;
17                 padre[v] = u;
18                 Encolar(Q, v);
```

La complejidad en tiempo es $O(|V| + |E|)$
La complejidad en espacio es $O(|V|)$

2. DFS: Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtraking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

```
1  DFS grafo G
       inputs : Árbol de búsqueda
       output: Nodo objetivo
2      foreach u∈V[G] do
3          estado[u] = NO_VISITADO;
4          padre[u] = NULO;
5      tiempo = 0;
6      foreach u∈V[G] do
7          if estado[u] == NO_VISITADO then
8              Visitar(u, tiempo);
```

```
1  Visitar grafo G
2      estado[u] = VISITADO;
3      tiempo = tiempo + 1;
4      d[u] = tiempo;
5      foreach v∈Vecinos[u] do
6          if estado[v] = NO_VISITADO then
7              padre[v] = u;
8              Visitar(v, tiempo);
9          estado[u] = NO_VISITADO;
10         padre[u] = NULO;
11     tiempo = 0;
12     foreach u∈V[G] do
13         if estado[u] == NO_VISITADO then
14             Visitar(u, tiempo);
15     estado[u] = TERMINADO;
16     tiempo = tiempo + 1;
17     f[u] = tiempo;
```

La complejidad en tiempo: $O(b^m)$ con b el factor de ramificación y m la profundidad del árbol.
La complejidad en espacio: $O(b^n)$ con b el factor de ramificacion y n la profundidad del nodo ojetivo.

3. Búsqueda de costo uniforme: La búsqueda comienza por el nodo raíz y continúa visitando el siguiente nodo que tiene menor costo total desde la raíz. Así el algoritmo implica la

expansión de nodos mediante la adición, a una cola con prioridad, de todos los nodos vecinos no expandidos que están conectados al último nodo analizado. En la cola, cada nodo se asocia con su costo total desde la raíz, donde se les da mayor prioridad a los caminos de costo mínimo. El nodo en la cabeza de la cola es expandido, adicionando sus nodos vecinos con el costo total desde la raíz hasta el nodo respectivo.

```
1  UniformCostSearch Graph, raiz, meta
2      nodo := raiz, costo = 0;
3      fronteras := coladeprioridadesquecontienesolonodos;
4      explorados := conjuntovacio;
5      if fronteras es vacia then
6          returnfailure;
7      nodo := fronteras.pop();
8      if nodo es la meta then
9          returnsolucion;
10     explorados.add(nodo);
11     foreach n en los vecinos de nodo do
12         if n no ha sido visitado then
13             if n no esta en fronteras then
14                 fronteras.add(n);
15             else if n esta en fronteras con costo alto then
16                 remplazanodeconn;
```

La complejidad en espacio y tiempo:$O(b^{1+C*/e})$ donde $C*$ es el costo de la solución óptima y b es el factor de ramificación.

4. Búsqueda a profundidad iterativa: es un algoritmo utilizado para una estrategia de búsqueda en el espacio de estados en la que se realizan sucesivas búsquedas en profundidad limitada incrementando el límite de profundidad en cada iteración hasta alcanzar , la profundidad del estado objetivo de menor profundidad.

```
1  BPI raíz, objetivo
2      profundidad = 0;
3      resultado = BPL(raíz, objetivo, profundidad);
4      if resultado es una solución then
5          devolverresultado;
6      profundidad = profundidad + 1;
```

```
1 BPL nodo, objetivo, profundidad
2     if profundidad == 0ynodo == objetivo then
3         devolvernodo;
4     else if profundidad > 0 then
5         hijo en expandir(nodo) resultado = BPL(hijo, objetivo, profundidad − 1);
6         if resultado distinto de null then
7             devolverresultado;
8     else
9         devolvernull;
```

Complejidad en Espacio: $O(bd)$, donde b es el factor de ramificación y d es la profundidad de la solución más superficial.
Complejidad en Tiempo: $O(b^d)$

1.3 Agentes de Búsqueda

Como ya habíamos visto en el proyecto pasado un agente es cualquier cosa que sea capaz de percibir su ambiente con ayuda de sensores y actuar en dicho ambiente utilizando actuadores. Ahora si pensamos en nuestro árbol de búsqueda podemos diseñarlo de tal forma que los estados representen el entorno percibido por el agente y las derivaciones serán producidas por las acciones del agente. Una pregunta interesante que se puede hacer es ¿Por qué necesitamos un árbol de búsqueda?¿De qué nos sirve generar todos los posibles estados?
Pues bien, pensemos en un cuadro mágico, tenemos un estado inicial bien representado por como están las piezas en el primer momento que te dan el cuadro; y un estado final que representa el cuadro mágico que se desea obtener. Como humanos tenemos diferentes estrategias para solucionar el problema, una de ellas bien pude ser estar moviendo los cuadros que pueden hacerlo sin ningún sentido buscando eventualmente llegar a dicho estado objetivo, aunque es una buena opción, no suena a que sea factible resolver el problema de esa manera. Es cierto que solo vamos a considerar mover piezas que puedan hacerlo, en general no movemos la primera que vemos, si no, que pensamos en siguientes pasos, y buscamos que eso que movimos sea justo lo que se tiene que hacer para llegar a la solución; la idea es que ese "razonamiento" lo obtenga nuestro agente y como ya habíamos mencionado antes los árboles de búsqueda modelan bastante bien eso.
Dado que nuestro agente busca un estado objetivo será un agente basado en objetivos. Así nuestro agente de búsqueda va a necesitar :

1. Nuestro árbol de busqueda: Que contiene nuestro estado inicial como raíz.
2. Estado objetivo: El estado al que queremos llegar.
3. Estrategia de búsqueda: cualquier recorrido ya antes mencionado.

Entonces la tarea de nuestro agente es recorrer el árbol con su estrategia de búsqueda buscando el estado objetivo, en el momento que lo encuentra nos devolverá la secuencia de acciones que se tiene que hacer para llegar a dicho estado y el costo de esta (si las acciones no tienen costo este dato sera irrelevante).

1.4 Búsqueda informada y Heuristicas

Podemos encontrar deficiencias en las búsquedas que hemos mencionado anteriormente, entre ellas:

1. Busca la primera solución sin importar que tan óptima sea, en general la solución depende de la estrategia de búsqueda.

2. No detecta si se aleja o acerca a la solución en la rama que está buscando, esto provoca usar recursos en recorridos que no valen la pena.

3. No es capaz de encontrar una solución aceptable en caso de que no exista o sea demasiado costoso encontrar la solución.

4. Existe posibilidad de recorrer todo el árbol para encontrar la solución.

Así estaremos buscando una forma de hacer mejores búsquedas. ¿Qué pasa si metemos información dentro de la búsqueda, que nos ayude a saber si ese estado conviene o no?, ¿Si debería revisarlo o no vale la pena?, vamos a generar búsquedas informadas.
Primero tenemos que generar ese algo que nos diga justo que tanto vale la pena un estado, y para esto ocupamos Heurísticas; Las heurísticas son criterios, métodos o principios para decidir cual de entre varias acciones promete ser la mejor para alcanzar una determinada meta..
¿Y esos criterios como los sacamos? En general, de nuestro propio conocimiento, o bien las reglas del juego. En el caso de los cuadros mágicos podemos pensar en la Heurística como la cantidad de cuadros que están en su lugar (según nuestro estado objetivo), así cada acción que haga que le numero de cuadros en su lugar sea mayor será mejor valorado para nuestra heurística propuesta.
Conceptualmente creo que se entiende la idea de la heurística y cómo nos va a ayudar a hacer una búsqueda más acotada e informada. Como siempre la siguiente pregunta sería, ¿Cómo introducimos la heurística a la computadora? La respuesta es relativamente intuitiva, la heurística será una función que claramente nos devolverá un número. En el nuestro ejemplo sería justo el numero de casillas en su lugar.
Hay que tomar en cuenta que cuando un agente de búsqueda ocupa una heuristica para poder encontrar un estado "aceptable" se puede convertir a un agente basado en utilidades, ya que la misma heuristica hace el papel de utilidad.

1.5 Admisible y consistente

Como las Heuristicas muchas veces salen de intuición de humana hay que tener parámetros que nos ayuden a decir que una heurística es buena, y que efectivamente nos ayuda a mejorar nuestra búsqueda. Vamos a introducir dos conceptos para esto:

1. Admisible: Para cada paso el costo estimado real va a ser menor o igual que el real, o bien, no se sobre estima los pasos necesarios para llegar al objetivo.

2. Consistente: Sean A y B estados tales que se puede llegar de A a B con una acción, entonces el costo de llegar de nuestro estado inicial a B tiene que ser menor o igual a la de llegar del inicial a A mas el costo de ir de A a B.

2 Minimax

Minimax es una regla utilizada para juegos de suma cero de dos jugadores en la que los jugadores juegan alternadamente.

2.1 Algoritmo

Algorithm 1: Algoritmo Minimax

```
1  Minimax (nodo, altura, maximizar)
       inputs : Un nodo, la altura del árbol a explorar y un booleano maxizar que
                determina si hay que maximizar o minimizar
       output: El valor de realizar minimax
2      if altura = 0 o esTerminal(nodo) then
3          return heurística(nodo);
4      if maximizar then
5          valor = −∞;
6          foreach hijo en hijos(nodo) do
7              valor = max(valor, Minimax(hijo, altura - 1, falso));
8          return valor
9      else
10         valor = +∞;
11         foreach hijo en hijos(nodo) do
12             valor = min(valor, Minimax(hijo, altura - 1, verdadero));
13         return valor
```

2.2 Descripción

El algoritmo Minimax realiza una verificación para saber si llegó a la altura de exploración determinada o si está en una hoja. En esos casos, aplica la heurística definida y retorna el valor que devuelve la misma.
Los nodos internos (no terminales) realizarán llamados recursivos a sus hijos para obtener su valor. Si es un nivel en el que el jugador quiere maximizar el valor se inicializará el valor de retorno en infinito negativo (como se puede observar en la línea 6) y luego se calculará el máximo entre el llamado recursivo a Minimax de todos sus hijos. En el caso de que se esté en un nivel en el cual se quiere minimizar el valor (que corresponde al valor de la jugada que haría el oponente) se calculará el mínimo entre el valor inicial (que será infinito positivo como se puede observar en la línea 11) y los valores de retorno del llamado recursivo sobre sus hijos. [3]

2.3 Justificación de utilización en el juego Othello

El Othello es un juego de dos jugadores (jugador **A** y jugador **B**) en el cual luego de que **A** realice una jugada que lo acerque a ganar el juego, **B** intentará exactamente lo mismo. La utilización de un algoritmo *minimax* como estrategia para decidir qué movimiento realizar para el jugador **A** es adecuada debido a que retornará el movimiento de mayor utilidad (hasta el nivel que se especifique desarrollar el arbol) suponiendo que el oponente **B** realizará una jugada para minimizar la utilidad de la misma (por este motivo intercala *max* y *min*, *max* corresponde a la jugada que realizará **A** y *min* a la jugada que realizará **B** para minimizar la utilidad de la jugada de **A**). [8]

2.4 Modificaciones realizadas para el juego Othello

A nivel procedural no hubieron modificaciones significativas al algoritmo pero debido a nuestra implementación si modificamos algunos aspectos de implementación, primero segmentamos las llamadas del algoritmo ya que nosotros no solo necesitabamos el valor resultante de aplicar la heurística, sino que necesitabamos realizar la jugada que nos resultara la mas viable en el algoritmo, así la llamada para el primer tablero hacíamos la llamada a otra función auxiliar que sí trabajaba con el algoritmo Minimax regular, y al regresar a la llamada original buscabamos el valor que nos regresaba el algoritmo en los hijos del primer nodo y así, finalmente, regresamos con el resultante que necesitamos para continuar el juego

3 Heurística

Para definir nuestra herística, asignamos pesos a las distintas posiciones del tablero de la siguiente forma:

	A	B	C	D	E	F	G	H
1	2000	2	100	100	100	100	2	2000
2	2	0	1	1	1	1	0	2
3	100	1	10	4	4	10	1	100
4	100	1	4	3	3	4	1	100
5	100	1	4	3	3	4	1	100
6	100	1	10	4	4	10	1	100
7	2	0	1	1	1	1	0	2
8	2000	2	100	100	100	100	2	2000

3.1 Descripción y justificación

El procedimiento que utilizamos para definir la heurística fue analizar el valor de las casillas viendo que las esquinas del tablero son fichas que el adversario no puede tomar, dándoles una prioridad mayor a cualquier otra casilla, después las fichas de las orillas también son de mucho peso ya que en general es el conjunto de fichas que, después de las esquinas, tienen una mayor dificultad de ganar al rival en el punto en que el rival las obtiene, pero a las casillas de la orilla que colindan con las esquinas en algunos casos es posible que la esquina corra riesgo, así esas casillas en particular representan un riesgo importante y por eso tienen una prioridad considerablemente menor que el resto de las orillas.

Continuando con el análisis desde el exterior hacia el interior del tablero fijémomos en las casillas que colindan tanto con esquinas como con casillas en la orilla del tablero, veamos que al capturar estas casillas ponemos potencialmente en riesgo tanto la esquina con la que colinda como las casillas de la orilla del tablero, por lo tanto esta ficha es la de menor prioridad de todo el tablero, ahora las que no colindan con esquinas pero si con las orillas sufren del mismo problema que anteriormente mostramos pero dado que no ponen en riesgo las esquinas tienen una prioridad muy baja pero mayor que si son adyacentes a las esquinas.

La siguiente porción del tablero que analizaremos son las casillas que colindan con las 4 casillas centrales del tablero, veamos que estas fichas se encuentran a 2 casillas de distancia de la orilla del tablero y hay 4 que además están a esa misma distancia de las esquinas, ahora veamos que para competir esa posición en la orilla (esquina) tenemos la prioridad porque el acceso a dichas casillas para el rival no es directo, y su ruta más directa es por alguna de las fichas explicadas en el párrafo anterior, las cuales habíamos visto que en algunas ocasiones prácticamente aseguraban alguna orilla (esquina) para el rival, así el dominar estas posiciones resulta en jugadas muy fuertes posicionalmente, así las que tienen distancia de 2 casillas tanto a alguna orilla como a alguna esquina tienen una prioridad igual a las orillas, mientras que las que solo lo cumplen con casillas de la orilla del tablero tienen una prioridad menor a la de las piezas de la orilla o de la esquina pero de mayor prioridad que cualquier otra casilla.

Por último las 4 casillas centrales tienen una prioridad menor a la del párrafo anterior, dado que al contrario del caso anterior estas tienen una distancia impar con las casillas de la orilla (esquina) más cercana y por el análisis anterior vimos no es tan fuerte, pero posicionalmente no son malas ya que también tienen una distancia par para la siguiente más cercana (en general 3 para la impar y 4 a la distancia siguiente), además que en general éstas casillas tienden a dar mas opciones de tiro para nuestro turno.

3.2 Modificaciones realizadas para distinguir niveles de dificultad

Las modificaciones realizadas para la distinción de los niveles de dificultad están definidos, más que por la heurística, por el nivel de exploración al que llegamos en el árbol realizando el algoritmo Minimax.
Esto es, nuestra dificultad está dada por el nivel de profundidad que exploremos en el algoritmo. Recordemos que el algoritmo recibe como parámetro a la *altura*, entonces esta es la que nos dará el nivel de dificultad.

4 Agente

"Los agentes constituyen el próximo avance más significativo en el desarrollo de sistemas y pueden ser considerados como la nueva revolución en el software"[6].
Recordemos que podemos definir a un agente como una entidad que percibe y actúa sobre su entorno. En nuestro proyecto, se trata sobre un tablero y jugar en este con la meta de ganar o no perder.

4.1 Descripción

Proponemos que nuestro agente sea de tipo basado en utilidad pues tenemos (en la mayoría de jugadas) varias opciones para tirar una ficha dado un escenario de una jugada previa. El hecho de tener jugadas para elegir nos puede hacer pensar en varias metas, como por ejemplo cambiar más fichas en la jugada, evitar que el oponente no tire en algún lugar, o simplemente conseguir lugares estrátegicos del tablero.
El hecho se asignar "pesos" a nuestro tablero para saber en dónde es mejor tirar una ficha nos permite distinguir entre una jugada que es buena y una que pueda ser mejor. La función de utilidad, entonces nos permite juntar las metas anteriormente presentadas para ver dadas nuestras posibles tiradas, cuál abarcaría más y darnos la mejor jugada.
Recordemos que la representación teórica de nuestro agente nos pregunta por el estado de nuestro entorno, las acciones que pueda realizar nuestro agente y así se evalúa qué tan buenas son los resultados y poder evaluar de entre todas, la que mejor convenga.

4.2 REAS

Rendimiento	Entorno	Actuadores	Sensores
Mayor cantidad de fichas	Tablero, fichas de dos colores	Poner ficha	Visor de fichas en el tablero

4.3 Propiedades del entorno

Descripción del entorno en el que estará el agente[7] :

- **Totalmente observable**: se conocen las posiciones de las fichas tanto del agente como del oponente en todo momento.
- **Agente simple**: dado que se maneja un solo jugador (que no coopera ni trabaja con otros agentes).
- **Determinista**: El siguiente estado se determina teniendo en cuenta la posición en la que el agente colocará la ficha.
- **Estático**: El entorno no cambia mientras el agente decide en que posición del tablero jugar la ficha así como no cambio el puntaje del agente por tiempo invertido en la toma de decisión.
- **Discreto**: La cantidad de movimientos posibles en cada turno son finitas.
- **Episódico**: El agente percibe el entorno y en base a lo que observa toma una decisión (en qué posición jugar su ficha). Luego se avanza al siguiente episodio que sería el tablero actualizado con la ficha que acaba de jugar.

5 Complejidad

5.1 Complejidad teórica

La complejidad de nuestro agnete es la suma de la complejidad de MinMax mas la complejidad de nuestra Heurística.

1. Complejidad en Tiempo: La complejidad en tiempo de MinMax es $O(b^m)$ con b el factor de ramificación y m la profundidad del árbol, ya que es un recorrido DFS con algunas comparaciones y asignaciones intermedias.
   La complejidad de nuestro algoritmo es constante(O(1)) ya que a lo mas recorrerá 64 casillas para obtener su valor heuristico. Así nuestra complejidad sera de $O(b^m) + O(1)$ o bien $O(b^m)$.

2. Complejidad en Espacio: Justificando de la misma manera la complejidad en espacio sera la generada por minmax, es decir la de una búsqueda por DFS, esto es, $O(b^n)$ con b el factor de ramificación y n la profundidad del nodo objetivo, dado que tenemos que buscar en todo el árbol $n = m$ (con m la profundidad del árbol).
   Nuestra heuristica ocupa un entero, por lo que su complejidad en espacio es O(1).
   Entonces nuestra complejidad en espacio sera de $O(b^m) + O(1)$ o bien $O(b^n)$.

5.2 Complejidad práctica

1. Complejidad en Tiempo: Recordando que la forma en que simulamos la dificultad en nuestro agente es por medio de la profundidad a la que exploramos con el algoritmo Minimax, así que para la dificultad 0 exploramos 2 niveles de profundidad y analizando el peor caso dentro de nuestra experimentación fue de 10 segundos para tomar una decisión en el tablero. Para la dificultad 1 que explora 4 nivéles del árbol nuestro peor caso experimental tomó 30 segundos. Por último en nuestra dificultad 2 que explora 5 nivéles del árbol encontramos que en nuestro peor caso experimental el agente tomó 55 segundos para decidir en donde tirar.

2. Complejidad en espacio: Como para cada nodo del árbol se generan nuevos tableros, y la cantidad promedio de nuevas jugadas por tablero es 8 (Estimación hecha por ensayos) nuestro árbol minmax va a generar 8^n tableros con n nuestra profundidad.

6 Conclusiones

El agente puede jugar al juego del Otello constituyéndose como un oponente competitivo. Es un juego relativamente pequeño, a diferencia de por ejemplo Go, lo cual facilitó mucho su implementación, y gracias a nuestra heurística y los pesos asisgnados al tablero es más fácil programarlo, pues no tuvimos que implementar estrategias con las que una persona podría jugar, como por ejemplo, guiar al oponente a que tire en casillas específicas, o lo que se conoce como jugadas predefinidas.
Fue un proyecto divertido y ninguno de nosotros había implementado antes un juego, entonces, puedes enseñárselo a algún amigo que no estudie computación y ya no son solo números. Y no es como el anterior, pues este no requiere de mucha capacidad en nuestra computadora. Sería un buen ejercicio poder implementar funciones que nos ayuden con este tipo de escenarios pues el agente a parte de usar la heurística, podría identificar jugadas que ya conoce que le convienen para ganar y en caso de no encontrarlas, seguir jugando con la heurística.

6.1 Ventajas

La ventaja más significativa es el hecho de que el agente no "piensa" tanto, pero tampoco juega por jugar, solo evalúa las posibles jugadas y cuál le conviene más, es algo que alguien en una partida de Otello podría hacer si pudiera pensar muy rápido y recordar las jugadas que ha analizado. Y programar eso es relativamente sencillo, pues hay muchos sitios que tienen el pseudocódigo, lo complicado es integrarlo al agente.

6.2 Desventajas

La mayor desventaja que encontramos fue la integración de nuestro tablero con el algoritmo de minimax, pues aunque en la teoría estaba bien, tuvimos problemas con python y tuvimos que ir viendo en dónde no habíamos integrado bien. Otro pequeño problema es el cómo pedimos la dificultad pues inicialmente queríamos una interfaz y con proccessing no pudimos, y quedó desde la línea de comandos.

References

[1] Bfs. https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura. Accessed: 2019-09-30.

[2] Dfs. https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad. Accessed: 2019-09-30.

[3] https://en.wikipedia.org/wiki/minimaxin_zerosum_games.

[4] Fernando Berzal. Búsqueda en inteligencia artificial. http://elvex.ugr.es/decsai/iaio/slides/A3 Accessed: 2019-09-30.

[5] Dr. Edgard Iván Benítez Guerrero. Resolución de problemas mediante búsquedas. https://www.uv.mx/personal/edbenitez/files/2010/09/CursoIA10-II-1.pdf. Accessed: 2019-09-30.

[6] Nicholas Jennings.

[7] P. Russell, S. Norman. Artificial intelligence a modern approach - section 2.4, 2009.

[8] P. Russell, S. Norman. Artificial intelligence a modern approach - section 5.2, 2009.