Documentación - Connect Four

Santiago Irazoqui - 234730

Link al proyecto de wandb
Link al repo

Los colores de las runs intentan simbolizar un poco qué ocurrió: verde es que salió todo bien, se cumplió el propósito de la run, o logre algo. Rojo es que cometí un error y la run ya no sirve tanto.

¿Espectimax o Minimax?

La empresa deja a decisión del experto si utilizar la técnica espectimax o minimax. Cada una de estas técnicas hace suposiciones diferentes sobre el oponente SpaceGPTAgent, que no conocemos. Espectimax asume que su oponente utiliza una distribución de probabilidad desconocida para elegir sus acciones. Minimax por su parte supone que el oponente es inteligente, por lo que también va a intentar de maximizar su propia utilidad, o en otras palabras minimizar la nuestra (ya que es un juego de suma cero)

La realidad es que sin saber no es posible simplemente decir “cual es mejor”. Minimax es una técnica conservadora. Asume que el oponente siempre va a tomar la mejor decisión para él. Por eso, frente a un jugador inteligente, minimax es la mejor estrategia. Es una de las propiedades de minimax: la garantía de que es la mejor estrategia frente a un π_{min}.

Y como se supone que oponente está jugando de la mejor manera posible, minimax también nos da la garantía de que es una cota inferior ante cualquier $\pi_{oponente}$. No podemos “estar peor” que lo que nos dice minimax, independientemente de la estrategia que utilice el oponente. Podríamos estar mejor con otra estrategia, pero Minimax es conservadora.

Y ahí entra el dilema. Al ser conservadora, si bien nos da una cota inferior, es verdad que sí oponente no utiliza π_{min}, hay veces que podríamos estar mejor. Espectimax, por ejemplo, puede terminar mejor que minimax en estos casos.

Entonces no hay respuesta definitiva. Por esa razón voy a crear los dos agentes, y ver cual es su rendimiento.

Nota:

Para los tests: chequeo siempre empezando el jugador, y la misma cantidad de veces empezando el oponente. De esa forma descarto la ventaja de empezar.

Minimax

Para comenzar, voy a implementar la técnica de minimax.

Sin funciones de evaluación

En un principio, solo quiero ver como se comporta sin ningún tipo de función de evaluación (retorna valor 0 cuando se llega a la depth).

https://wandb.ai/santiago-irazoqui/ConnectFour/runs/srqm0igl/overview?workspace=

Key	Value
Average - Minimax first	1
Average - opponent first	1
Runs per test	100
Runs won Minimax first	100
Runs won by expectimax	200
Runs won opponent first	100
Total test runs	200

No se porque, pero esta ganando siempre minimax.

Funciones de evaluación

No voy a pegar el código de todas porque se hace muy pesado de leer.

1: P = “Cantidad de tiras de fichas propias”

Para esta función de evaluación, busqué dar un puntaje a las tiras de fichas que tenga el jugador, ya sean horizontales, verticales o diagonales. Cuantas más fichas, los pesos son más grandes (una fila de 3 es mejor que una fila de 3).

En el código esta heurística se llama:

```
def heuristic_score_lines(self, board: Board, current_player)
```

Evaluacion

1. Eval debe ordenar los estados terminales de la misma manera que la función de utilidad verdadera

Eval(win) > Eval(draw) > Eval(loss)

En un principio se puede ver que esta función no ordena. Es posible ganar y tener más líneas de 3 que el oponente, por lo que tendríamos más valor según esta heurística. Lo mismo ocurriría en el caso de loss.

Sin embargo, esto se soluciona fácilmente porque el puntaje de una tira de 4 fichas, la cual me hace ganar, tiene un valor asociado muy grande. Entonces, si gané, es porque tengo una tira de 4 fichas, por lo que P(win) > 0 independientemente de qué tenga el oponente. De la misma manera, una tira de 4 fichas del oponente, tiene mucho peso negativo, por lo que en P(loss), va a ser menor que 0.

En el caso de draw, ninguno tiene una tira de 4 fichas, por lo que el resultado puede variar, pero seguro va a estar dentro de win y loss.

Por lo tanto nos queda:

P(win) > P(draw) > P(loss)

2. El cálculo de Eval no debe tardar demasiado, dado que el objetivo es buscar más rápido.
Esta función es lineal, y luego local. Tengo que recorrer el tablero, y para cada ficha recorrer las posibles tiras que cumpla. Como no llega a ser exponencial, es aceptable.

3. Para todo estado no terminal s, Eval(s) debe estar fuertemente correlacionada con las posibilidades reales de ganar a partir de s.
Está fuertemente correlacionada ya que tener tiras de fichas más largas me acerca a la condición de ganar que es tener una tira de 4 fichas. Tener 3 fichas separadas es menos

deseable que tener una tira de 3 juntas. Una tira de 2 es menos deseable que una de 3. Conforme aumenta el valor de la heurística, también aumenta la probabilidad de que gane.

La heurística nos queda entonces:
$P(s) = \alpha(P_A(s) - P_O(s)), \alpha > 0$

[Ver run en wandb - Minimax vs SpaceAgent usando solo P]

2: F = “fichas que me pueden comer”

Me interesa esta heurística por varias razones. Primero que nada, porque es defensiva. Tener solo heurísticas agresivas puede terminar haciendo que el jugador pierda porque ignora buenas posiciones intentando atacar. Por ejemplo, puede ser más interesante proteger una ficha que tener una tira más larga.

Heurística en el código:

```
def heuristic_fichas_comibles(self, board: Board, current_player)
```

Evaluacion

1. Eval debe ordenar los estados terminales de la misma manera que la función de utilidad verdadera

Eval(win) > Eval(draw) > Eval(loss)

Esta heurística no ordena igual que la utilidad. En el caso de F(draw), es igual a 0, ya que draw solo ocurre si no hay más movimientos y por lo tanto no se pueden comer más fichas. Sin embargo, en los estados de win y loss es arbitrario cuanto son los valores. Puedo ganar y tener varias fichas que me pueden comer. O también puedo ganar y tener todas las fichas protegidas.

En teoría tiene sentido, pero la realidad es que no encuentro forma de hacer que ordene.

2. El cálculo de Eval no debe tardar demasiado, dado que el objetivo es buscar más rápido.
Tiene el mismo orden que P, ya que recorro el tablero y para cada ficha que encuentre recorro su entorno local

3. Para todo estado no terminal s, Eval(s) debe estar fuertemente correlacionada con las posibilidades reales de ganar a partir de s.

En general, que me coman fichas no es deseable. Cada vez que me comen una ficha es automáticamente una menos para mi, y una más para el oponente

Ver run en wandb - Minimax vs SpaceAgent usando solo F

3: C = “fichas en el medio”

El razonamiento de esta heurística es simple. En el tablero, tener fichas en el medio es una gran ventaja. Si miramos el tablero:

```
  0   1   2   3   4   5   6
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
-------------------------------
```

Si ponemos una ficha en la columna 3, estamos bloqueando a que nuestro oponente haga 4 en toda esa fila, ya que solo hay 3 para cada lado.También se bloquean las diagonales por la misma razón. Si tuviéramos toda la columna del medio ocupada con nuestras fichas, entonces el oponente solo podría hacer 4 de forma vertical. En esta versión del juego se pueden comer fichas, por lo que no son tan fuertes como antes, pero de todos modos son muy fuertes.

Nombre de la heurística en código:

```
def heuristic_middle_pieces(self, board: Board):
```

Evaluacion

1. Eval debe ordenar los estados terminales de la misma manera que la función de utilidad verdadera

Eval(win) > Eval(draw) > Eval(loss)

El problema con esta heurística es que tampoco ordena los valores igual que la utilidad real. Ocurre lo mismo: en los casos terminales no hay nada que defina quien pueda tener más fichas en el medio. Se puede ganar con muchas, pocas o igual cantidad de fichas en el medio para ambos. Se puede perder de la misma manera, y también se puede ganar. No ordena nada.

2. El cálculo de Eval no debe tardar demasiado, dado que el objetivo es buscar más rápido.

Es la más fácil. Solo recorro una columna y cuento fichas por jugador.

3. Para todo estado no terminal s, Eval(s) debe estar fuertemente correlacionada con las posibilidades reales de ganar a partir de s
No se que tan “fuertemente” relacionado este. Claramente aumenta las probabilidades por lo que dije anteriormente, pero en este juego se pueden comer fichas, por lo que ahora también es necesario protegerlas, sacando un poco de atractivo.

Comentarios

Al final decidí no utilizar las heurísticas F y C por las razones que especifique. Tengo que admitir que jugando contra los algoritmos, sentí que sin estas dos heurísticas su desempeño fue peor, especialmente el de minimax.

Ver run en wandb - Minimax vs SpaceAgent usando solo C

Minimax vs SpaceAgent

Key	Value
Average - Opponent first	1
Average - Player first	1
Runs per test	100
Runs won total	200

Ganó todas contra el oponente SpaceAgent, tanto arrancando minimax como oponente.

Ver run en wandb - Minimax vs SpaceAgent usando todas las heurísticas

Alpha beta prunning

Se aplicó la técnica de alpha beta prunning para ver sus efectos en el tiempo de ejecución. Esta técnica sirve para reducir el número de nodos del árbol de juego que minimax tiene que explorar para tomar una decisión.

El gran problema de minimax es que el árbol de exploración se vuelve muy grande muy rápidamente: su crecimiento es exponencial. Por lo tanto llega una profundidad a la que simplemente toma demasiado tiempo, y por eso es que esta se limita y utilizan las funciones de evaluación (por ejemplo). Si logramos podar algunas de estas partes del árbol (las que de

todos modos no influyen si se recorrieran, por eso “poda”), entonces el algoritmo sería más eficiente. Incluso podríamos quizá aumentar la profundidad.

Alfa es la mejor opción hasta el momento en el camino de max. Por eso, cuando estamos en un nodo max (jugamos nosotros), hacemos:

```
alpha = max(value, alpha)
```

Alfa es entonces la mejor opción que tenemos nosotros en general. Por lo tanto, si alfa, nuestro mejor movimiento actual, es mejor (mayor) que beta (la mejor jugada del oponente), entonces el oponente nunca va a tomar esa acción y podemos podar esa rama

```
alpha = max(value, alpha)
if alpha >= beta:
      break
```

Beta, por otro lado, es la mejor opción en el camino de min. Por lo tanto, cuando minimizamos (juega oponente):

```
beta = min(value, beta)
if alpha >= beta:
      break
```

Esto es análogo al caso de max.

Una observación importante es que alpha-beta prunning no modifica el árbol generado en si. Las acciones del árbol se siguen teniendo que generar (siempre llegamos a las hojas o max_depth, al final de la recursion). La diferencia es que estas se cortan **antes** de que se evaluen, y ahi la reducción de tiempo.

Por lo que entiendo (no he podido encontrar exactamente), alpha-beta prunning aproximadamente hace que en el mismo tiempo que demoraría Minimax, se pueda explorar el doble. Por lo tanto, esperaría ver una reducción del tiempo a la mitad.

Voy a adjuntar fotos de los tiempos que llevo correr los tests sin aplicar alpha/beta prunning, y luego aplicándola.

Sin alpha beta prunning

```
runs_won_player_first = 0
for i in range(total_test_runs):
    play_vs_loaded_agent_no_render(env, agent=minimax_agent)
    if (env._grid._winner == minimax_agent.player):
        runs_won_player_first = runs_won_player_first + 1

print("runs won: " + str(runs_won_player_first))
average_player_first = runs_won_player_first / total_test_runs
print("average: "  + str(average_player_first))
```

✓ 14m 20.0s

```
runs_won_opponent_first = 0
for i in range(total_test_runs):
    play_vs_other_agent_no_render(env, agent1=load_enemy_agent(), agent2=minimax_agent)
    if (env._grid._winner == minimax_agent.player):
        runs_won_opponent_first = runs_won_opponent_first + 1

print("runs won: " + str(runs_won_opponent_first))
average_opponent_first = runs_won_opponent_first / total_test_runs
print("average: "  + str(average_opponent_first))
```

✓ 6m 32.7s

Con alpha beta prunning

```
runs_won_player_first = 0
for i in range(total_test_runs):
    play_vs_loaded_agent_no_render(env, agent=minimax_agent)
    if (env._grid._winner == minimax_agent.player):
        runs_won_player_first = runs_won_player_first + 1

print("runs won: " + str(runs_won_player_first))
average_player_first = runs_won_player_first / total_test_runs
print("average: "  + str(average_player_first))
```

✓ 6m 36.8s

```python
runs_won_opponent_first = 0
for i in range(total_test_runs):
    play_vs_other_agent_no_render(env, agent1=load_enemy_agent(), agent2=minimax_agent)
    if (env._grid._winner == minimax_agent.player):
        runs_won_opponent_first = runs_won_opponent_first + 1

print("runs won: " + str(runs_won_opponent_first))
average_opponent_first = runs_won_opponent_first / total_test_runs
print("average: "  + str(average_opponent_first))
```

3m 58.6s

Como vemos, aproximadamente nos llevó la mitad de tiempo ejecutar todos los tests, una diferencia bastante grande. Esto concuerda con la reducción de tiempo de ejecución que supuse podría darse.

Ver la run en wandb

Espectimax

Key	Value
Average - espectimax first	1
Average - opponent first	1
Runs per test	100
Runs won by expectimax	200
Runs won espectimax first	100
Runs won opponent first	100
Total test runs	200

Espectimax siempre le pudo ganar a SpaceAgent.
Ver run en wandb

Minimax vs Espectimax

Key	Value
Runs per test	100
Runs won by espectimax when minimax first	0
Runs won by espectimax when_espectimax first	0
Runs won by minimax when espectimax first	100
Runs won by minimax when minimax first	100
Total test runs	200

Ver run en wandb

Minimax vs minimax

Supuestamente MiniMax contra MiniMax debería terminar siempre en empate.
No fue lo que ocurrió: ganó siempre el que fue primero.

Runs per test	500
Runs won by first minimax	500
Runs won by second minimax	0
Total test runs	500

No entiendo muy bien porqué ocurrió esto.

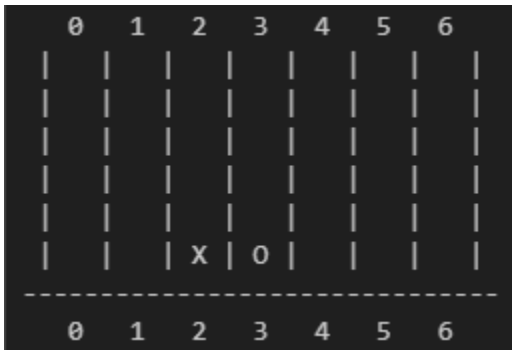
Ver run en wandb

Yo vs Algoritmo

Yo vs Minimax

Es mejor de lo que esperaba contra una persona. Tengo que admitir que si no estaba prestando atención, supe perder, aunque generalmente era por razones bobas como olvidarse que entel juego se puede comer.

Más allá de eso, el algoritmo tiene algunos errores que hace que caiga en un trampa por si solo. Por ejemplo:



Con una jugada asi ya puedo ganar. Simplemente tengo que poner mi ficha en la columna 1, comer la de la columna 2, y voy a tener una tira de 3 con dos extremos libres. Osea, gano.

Yo vs Expectimax

Como es de esperarse, a Espectimax le fue peor. Toma decisiones que no parecen tener mucho sentido para un humano, como estar construyendo por un lado del tablero, y poner una del otro lado que no afecta a nada que ni yo ni él estuviéramos intentando.

Contra espectimax fue más difícil perder