PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IIC2613 - Inteligencia Artificial
2do semestre del 2021

Tarea 1

Instrucciones

Entrega

La tarea tiene plazo hasta el Miércoles 29 de Septiembre a las 23:59. Se recolectará el commit en el *default branch* a la hora de entrega.

Repositorio

Ésta tarea usa `git`. Usted trabajará individualmente en un *fork* **privado** del repositorio de la tarea al que se le dará acceso mediante github classrooms.

Su fork debe modificar el código y agregar un documento con las respuestas. Para las preguntas de implementación basta referenciar a su código.

Las preguntas públicas respecto a la tarea deben ser hechas a través de *issues* en github.

Integridad

Su repositorio debe permanecer privado y se espera que sigan el Código de Honor de la escuela.

Toda respuesta debe ser escrita por ti, individualmente. Esto significa que al momento de editar la respuesta, no debes copiar el material hecho por otra persona. Queremos tu comprensión y entendimiento de las cosas que estudies y aprendas. Por lo tanto, las definiciones y análisis que presentes deben venir de tu propio razonamiento.

También es importante dejar claro que sabemos que estás utilizando material hecho por otras personas, y debes reconocerlo. Es por eso que te pediremos que cites tus fuentes en esta oportunidad. Tienes dos formas de citar. La primera es bastante indirecta, y solamente exige que menciones a tu fuente en tu bibliografa. Harás esto cuando hayas aprendido conceptos, los hayas interiorizado, y estés dando tu interpretación. La segunda es más directa, utilizarás las comillas para parafrasear algo que se le ocurrió a otra persona, seguido de un paréntesis con la referencia de dónde se encuentra dicho texto. Solamente debes parafrasear cuando quieres darle fuerza a un argumento, jamás para contestar algo que se te pregunte a ti, pues tal respuesta, si bien te evita el plagio, tampoco te da puntaje. Las citas de publicaciones cientficas las puedes poner en cualquier formato de amplia aceptación como APA, ICONTEC e IEEE, por nombrar algunos. Las citas a sitios web las debes hacer referenciando el link del sitio, y la fecha de la que has tomado la referencia. Por favor, no cites a tus compañeros, aprende de ellos, sé crtico con sus opiniones, y entrega tu propio trabajo original. Esta misma regla también se cumple respecto del código que entregues.

Código

Hay implementaciones parciales para ordenar su código detro del proyecto y también tests que se refieren a estas funciones parcialmente implementadas. Como parte de la implementación se espera que pasen los tests deshabilitados.

```python
@pytest.mark.skip(reason="TODO: Implement as part of the assignment.")
def test_numbers():
    assert 0 <= 1, "This holds, right?"
```

Listing 1: Un test deshabilitado. Borrar la línea del *decorator* de pytest lo habilita.

Tests

Puede ejecutar los tests con `pytest`. Note que hay test ignorados marcados con una `s` (*skipped*) que eventualmente deberían habilitarse y pasar.

Formato

El proyecto usa `black` para evitar tener que tomar decisiones de formato. Se espera que cada commit pase los checks de `black .`. De lo contrario se descontarán 0.5 puntos.

`git` hooks

Se recomienda habilitar `pre-commit` hooks con tal de evitar romper tests o el formato accidentalmente.

Búsqueda

Problemas

15-puzzle

El Puzzle de 15 fue mencionado en varias ocaciones ya que es bastante sencillo obtener buenas heurísticas.

- [0.2] Implemente la heurística mencionada en clases de la suma de las distancias de Manhattan entre las losetas (*tiles*) y su posición final.
- [0.1] ¿Es esta heurística admisible? Justifique

  R: Si, porque es consistente. También se puede justificar con que estamos contando una cota inferior para el número de acciones que necesitamos ejecutar.
- [0.3] ¿Es esta heurística consistente? Justifique

  R: Esta heurística considera una relajación del problema y lo resuelve de forma óptima. La restricción de que sólo se puede mover la loseta hacia la casilla vacía es la que se quita.

Sokoban

El Sokoban es un puzzle en que un agente debe ordenar una bodega y ubicar cajas en algunas posiciones específicas.

En clases y ayudantías mencionamos que una buena heurística es la de sumar las distancias de Manhattan entre cada caja y su objetivo más cercano.

- [0.2] Implemente esta heurística.
- [0.3] Notando que el agente tiene que acercarse a las cajas es fácil notar que falta contar acciones. Implemente esta versión mejorada de la heurística.
- [0.7] Implemente una heurística estrictamente mejor que las anteriores.
  - ¿Por qué su heurística es mejor?
  - Agregue un test mostrando un estado en que:
    - Su heurística sea mejor.
    - Su heurística sea admisible. Use BFS para verificar el costo óptimo.

Sudoku

El Sudoku es un juego bastante popular en que hay que llenar un tablero sin repetir dígitos en columnas, filas y bloques.

- Pese a que para este puzzle conocemos h^*, usar A^*no es ideal.

  NOTA: Realmente en este Puzzle no tenemos h^*, pero si sabemos exáctamente cuántas acciones faltan desde el estado inicial. Lo que no sabemos es si la heurística baja en 1 o se vuelve infinito al marcar un número adicional en la grilla.

  - [0.2] ¿Cómo se comportan BFS y DFS? ¿Cuál encuentra soluciones antes?
    R: BFS es muy lento al intentar todas las alternativas para cada jugada antes de intentar llenar una segunda casilla. Se necesita expandir la mitad del espacio de estados (entre los estados que son sucesores del estado inicial) antes de poder encontrar una solución.

    DFS en cambio va a intentar llenar las casillas con algún número, y va a revisar muchos tableros completos rápidamente. También va a depender muy fuertemente de las primeras decisiones, ya que al enfocarse en nodos a mayor profundidad se puede tardar bastante en considerar alternativas para estas decisiones más viejas. Si la implementación del Puzzle evita que se violen las restricciones al generar vecinos no debería ser un problema muy grande y se va a lograr un comportamiento mucho mejor que el de BFS.

- [0.5] ¿Por qué A^*no es ideal para resolver sudokus?

  R: Los algoritmos de búsqueda son ciegos a elegir jugadas forzadas como normalmente ocurren al resolver el Sudoku, y por podrían explorar sub-árboles que no tienen soluciones.

  Este problema es debido a que la interfaz (API?) entre el problema y el solver no lo permite al solver entender estos detalles del problema, y sólo pueden ver un grafo gigantesco que no evidencia las restricciones del Sudoku.

- [0.3] ¿Cómo se comporta A^*con h^* en este puzzle de costos uniformes?

  R: A^*al usar una heurística como la cantidad de casillas que falta por llenar no logra priorizar nodos de mejor manera, ya que todos los estados van a tener el mejor valor f. Si se usa la optimización de desempates por menor valor h (equivalente a mayor valor g) entonces se transforma simplemente en DFS.

  R': A^*con desempates al menor valor h cuando se usa realmente h^* va a hacer tantas expansiones como casillas falten por llenar (salvo expandir el estado final, ya que hacemos el *goal-check* justo antes). Lo anterior es muy bueno, pero el problema es que computar h^* no es viable. Calcular h^* parece ya ser tan o más difícil que resolver el Sudoku, incluso adivinando una solución del Sudoku, verificarla y usarla para determinar si un estado va bien encaminado o no puede necesitar más trabajo cuando hay soluciones múltiples. Es simple notar que agregar un par de *checks* para retornar infinito si se viola alguna de las restricciones del Sudoku no basta, de lo contrario no existirían las instancias en las que "hay que adivinarz hacer backtracking.

Algoritmos

ID-DFS

Los problemas más graves de DFS son que no es completo para grafos infinitos, y aún más lamentable, que no es óptimo ni si quiera en grafos finitos.

Iterative deepening DFS soluciona ambos problemas al imponer límites incrementales de profundidad.

- [0.3] Implemente ID-DFS y logre pasar los tests de optimalidad.

  R: La implementación de ID-DFS necesita usar DFS con límite de profundidad. Se puede extender la implementación de DFS para grafos, o implementar alguna versión de DFS con límite de profundidad directamente.

  Al agregarle límite de profundidad a DFS para grafos, hay que tener cuidado con el Closed set, ya que puede evitar que algunos caminos sean considerados.

  Implementar DFS para árboles (sin Closed) con límite de profundidad puede resultar más sencillo a pesar de ser más trabajo.

A^*

[0.8] Implemente A^* Consulte la implementación de Dijkstra y de Greedy search para familiarizarse con el *intrusive heap* de Open y el uso de heurísticas.

R: Implementar A^*desde la implementación de Dijkstra es bastante simple. Hay que cambiar la clase base a `HeuristicSearchAlgorithm` tal como lo hace Greedy, extender los Nodos para guardar el valor de la heurística, y definir el orden en torno a menor `self.g + self.h`.

Desempates Una implementación ingenua de A^*simplemente escoge nodos por mejor valor f, pero en muchos problemas el valor de f es bastante uniforme y no se logra rankear nodos efectivamente.

- [0.2] Implemente desempates optimistas usando mayor g o menor h.

  R: Esto es muy simple notando que se puede comparar pares (f, h) al implementar `__lt__`.

Búsqueda con Adversario

Algoritmos híbridos

Suponga que tiene implementaciones de,

- Poda alfa-beta con,
  - Ranking de acciones
  - Estimación de estados
  - Estimación de a qué profundidad cortar junto con un límite fijo.
- Monte-Carlo tree search (MCTS) con,
  - Estimación de a qué profundidad cortar junto con un límite fijo.

Al crear un algoritmo híbrido tenemos la opción de correr primero poda alfa-beta y luego cambiar a MCTS, o viceversa.

- [0.2] ¿Qué orden de algoritmos usaría? Justifique.

  R: Usar Poda alfa-beta para cada estado que simula MCTS significaría que vamos a correr muchas veces la poda alfa-beta, que es algo que explora muchos nodos.

  Usar Poda alfa-beta una vez, y después usar MCTS para mejorar la estimación de estados al momento de cortar tiene bastante más sentido y se asegura que no vamos a repetir trabajo.

Búsqueda con Adversario

Algoritmos

Suponga que hay un árbol de juego con b ramas por nivel y una profundidad d.

- [0.1] ¿Cuántos nodos exploraría minimax?

  R: Todos, es decir $\sum_{i=0}^{d} b^i = \frac{b^{d+1}-1}{b-1}$ (WA). Decir que son $\mathcal{O}(b^d)$ también se lleva el puntaje completo.

- [0.1] ¿Cuántos nodos exploraría poda alfa-beta en el peor caso?

  R: Todos, ya que no puede podar en ningún caso. Ya los contamos.

- [0.5] ¿Cuántos nodos exploraría poda alfa-beta en el mejor caso?

  R: Hay que comenzar por describir el mejor caso de la poda. Para esto notamos que `max_value` y `min_value` tienen que encontrar las acciones en orden tal que pueden usar su *early-return* lo antes posible.

```python
def max_value(state: State, alpha: Score, beta: Score) -> Score:
    """The best outcome we can get."""
    if problem.is_terminal(state):
        return problem.linearized_utility(state)

    best_value = float("-inf")
    for (a, s) in state.neighbors():
        best_value = max(best_value, min_value(s, alpha_beta))
        if best_value >= beta:
            return best_value   # <-- Poda de Max
        alpha = max(alpha, best_value)
    return best_value

def min_value(state: State, alpha: Score, beta: Score) -> Score:
    """The worst outcome we may face."""
    if problem.is_terminal(state):
        return problem.linearized_utility(state)

    worst_value = float("inf")
    for (a, s) in state.neighbors():
        worst_value = min(worst_value, max_value(s, alpha, beta))
        if worst_value <= alpha:
            return worst_value   # <-- Poda de Min
        beta = min(beta, worst_value)
    return worst_value
```

Vamos a pensar en el caso en que el jugador Max gana, y recordar que siempre los valores de los nodos están con respecto al valor obtenido por este jugador.

Para entender lo que pasa, basta pensar en 2 niveles partiendo con un nodo Max en que las b jugadas están ordenadas de mejor a peor. `max_value` se va a conformar con la primera jugada al notar que nos da un valor $+\infty$ (que es mucho mejor que el perder por default). Con esto, de los b nodos Min hijos que "hay", vamos a evaluar sólo 1, por lo tanto ya hay $b-1$ nodos del árbol de juego que no visitamos, cada uno con b nodos que no se evaluaron.

Ahora nos falta evaluar ese nodo Min, el que como ya sabemos hace perder al oponente (en Max nos quedamos con este sin pensarlo), debe evaluar sus b hijos intentando no perder, y cada una de esas opciones no le va a ser favorable al jugador Min.

Llegar hasta esta descripción basta para otorgar 0.3/0.5 puntos.

Con esto hemos visto 2 niveles por completo, pero la poda nos permitió sólo tener que evaluar 1 nodo Min en `max_value`, el primer nivel, y b nodos Max hijos del nodo Min que evaluamos en `min_value`.

Entonces en vez de tener que evaluar $b + b^2$ nodos en estos 2 niveles, bastaron $1 + b$.

Si obviamos ese nodo de `max_value`, tenemos que cada 2 niveles evaluamos b nodos en `min_value`, es decir, podemos pensar en que es un árbol de profundidad $\frac{d}{2}$ y branching factor b. También podemos pensar que seguimos en un árbol de altura d, pero con un *branching factor* aparente de *sqrtb*, ya que luego de 2 niveles recién encontramos b nodos.

Con esto se llega a $\mathcal{O}(b^{\frac{d}{2}})$.

Multijugador

- Suponga que hay un juego determinista y conocimiento perfecto en que se compite por obtener puntaje de un pool finito, pero que se juegue de a 3 o más jugadores.
  - [0.2] ¿Qué problemas hay al tratar de usar minimax y poda alfa-beta?
  - [0.6] ¿Cómo podemos adaptar poda alfa-beta para resolver el problema con N jugadores?
  - [0.2] ¿Cómo podemos adaptar Monte-Carlo Tree Search?