Variación de heurísticas en PDDL

Álvaro Cortés Casado
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
alvcorcas@alum.us.es
alvarocortescasado@gmail.com

Sergio Rojas Jiménez
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
serrojjim@alum.us.es
sergiorojasjimenez8@gmail.com

***Resumen*—El objetivo principal del trabajo es diseñar e implementar una nueva función heurística aplicando el protocolo "prego" para resolver problemas de planificación, y comprobar su eficacia realizando diversos experimentos tanto con la nueva heurística como con heurísticas vistas en teoría como $\Delta 0$ y analizando los resultados obtenidos.**

Además de la implementación de una nueva heurística, se han implementado dos técnicas de búsqueda, hacia delante y hacia atrás. Todo esto se ha juntado en una herramienta en la cual introduciendo un estado inicial,un estado final y diversas acciones además de elegir entre usar una u otra heurística y lo mismo con la técnica de búsqueda, puedes obtener una solución al problema de planificación.

***Palabras clave*—Inteligencia Artificial, PDDL, Backtracking, hacia delante, heurística, prego,$\Delta 0$, IA, Planificación, Técnica, Búsqueda**

I. Introducción

Este trabajo se enmarca en el contexto de los problemas de planificación en Inteligencia Artificial, que consisten en la búsqueda de una secuencia de acciones que nos conduzcan desde una situación inicial a un objetivo.

A continuación se definen una serie de conceptos que se verán a lo largo de este documento:

- Estado: Descripción de una posible situación en el problema mediante literales o predicados. Hay dos estados principales, estado inicial (situación inicial del problema) y estado objetivo (situación final del problema).

- Predicado: Propiedades del problema y de los objetos que lo componen.

- Acción: Operadores básicos que nos permiten transitar entre estados. Se caracterizan por tener precondiciones (condiciones necesarias para que se pueda aplicar dicha acción) y efectos (cambios que se producen en el estado actual y que desembocan en un nuevo estado)

- Heurística: Método de indagación que estima la distancia (entendida como un número) desde un estado hasta el estado objetivo. Su función es encontrar rutas óptimas entre el estado actual y el estado destino.

Para familiarizarnos con los problemas de PDDL, lo primero que hicimos fue realizar varios problemas de casuísticas sencillas del boletín de la asignatura.

Una vez ambos miembros del equipo estábamos familiarizados, comenzamos a plantear la implementación de clases que definen estados y acciones, la heurística a la que hemos denominado prego utilizando el protocolo homónimo, seguido de la heurística $\Delta 0$ y terminando por técnicas de búsqueda de soluciones.

De esta manera, el documento quedará estructurado por secciones en donde se detallarán los preliminares para la resolución, la metodología para la resolución del trabajo, los resultados obtenidos realizando diversas pruebas y las conclusiones sobre estos resultados.

II. Preliminares

En esta sección se hará una breve introducción de las técnicas empleadas para resolver el problema y el diseño de este.

A. Métodos empleados

Las técnicas empleadas para la resolución del problema han sido búsqueda hacia delante y búsqueda hacia atrás. .

Se han creado las siguientes clases y los siguientes ficheros:

- State, para definir el estado inicial, estado final u objetivo y las situaciones intermedias que se crean al aplicar acciones.
- Action, para definir una acción con sus precondiciones y efectos.
- Heuristics.py, como un fichero con dos funciones para el cálculo de las heurísticas prego y $\Delta 0$.
- Search.py, como un fichero donde se encuentran las funciones de búsqueda de soluciones.
- Interface.py, como un fichero donde se encuentra la implementación de la interfaz que utilizará el usuario para introducir los datos del problema. Para la interfaz se ha utilizado las librerias de tkinker [1]

Los puntos anteriormente descritos de manera general y explicados más adelante están inspirados en las transparencias de la asignatura [2].

B. Trabajo Relacionado

Ambos alumnos tenemos experiencia previa en la implementación de problemas de programación dinámica y backtracking. Algunos de estos problemas son:

- El problema de la mochila [3]
- El problema de las n-Reinas [4]

III. METODOLOGÍA

En esta sección se explican de manera detallada las clases y ficheros indicadas en la sección anterior.

A. Clase *State*

La clase **State** esta formada por un único atributo llamado **literals** que consistes en una lista de String. Cada String corresponde con un literal.

Para esta clase se han definido las siguientes funciones:

1) **satisfy_preconditions(action)**: devuelve true si el estado self satisface todas las precondiciones de action. Para ello se comprueba si cada precondicion positiva está contenida en los literales del estado y que ninguna de las precondiciones negativas se encuentre entre los literales del estado.
2) **apply(action)**: devuelve el estado que corresponde al estado actual después de aplicarle una acción A. Los literales de este nuevo estado seran los literales del estado anterior quitando los efectos negativos de la acción y añadiendo los efectos positivos
3) **is_relevant(action)**: devuelve un boolean que indica si la acción action es relevante para el estado actual.
   Decimos que una acción A es relevante para G si al menos, uno de los efectos (positivo o negativo) de la acción está en G con el mismo "signo" y ninguno de los efectos (positivo o negativo) de la acción aparece en G con distinto "signo"
4) **disapply(action)**: devuelve un estado que corresponde al estado actual después de des-aplicar una acción A. Para ello a la lista de literales del estado actual, se le elimina los literales correspondiente a los efectos de la acción A y se realiza la unión con las precondiciones de la acción A.
5) **not_contains_any_in(visiteds)(action)**: devuelve un boolean que indica si el predecesor del estado actual respecto de action no es un objetivo que contiene a alguno de visiteds.

B. Clase *Action*

La clase **Action** está formada por tres atributos:.

- **name**: string con el nombre de la acción
- **preconditions**: como un set de strings que representan literales
- **effects**: como un set de strings que representan literales

C. Heurística $\Delta 0$

La heurística $\Delta 0$ vista en clase, se ha implementado en la función **delta0** que recibe como parámetros de entrada un estado inicial, un estado final u objetivo y una lista de acciones. $\Delta 0$ cuenta el menor número de pasos necesarios para que se verifique p a partir de e, suponiendo que las acciones no tienen precondiciones negativas ni efectos negativos.[2]

delta0$(state, targets, actions)$
Entrada: un State $state$, una lista de String $targets$, una lista de Action $actions$
Salida: un entero $result$ con el mínimo número de acciones a aplicar para satisfacer todos los literales de targets partiendo de state

```
1  result ← 0
2  Por cada target en targets:
3      Si target no está entre los literales de state entonces
4          causes ← []
5          Por cada action en actions
6              Si target está entre los efectos de action entonces
7                  causes ← causes + [action]
8          Si causes no tiene elementos entonces
9              result ← ∞
10         Si no entonces
11             heuristic ← []
12             Por cada cause en causes
13                 temp ← 1 + delta0(state,
14                         cause.preconditions, actions)
15                 heuristic ← heuristic + [temp]
16             minimun ← menor número de heuristic
17             result ← result + minimun
18 retornar result
```

Fig. 1. Algoritmo de cálculo de la heurística $\Delta 0$

D. Heurística **Prego**

La heurística Prego se ha implementado usando el protocolo homónimo definido en el documento explicativo del trabajo. Este refleja la longitud del camino (conformado por acciones) más corto que satisface una lista con los literales objetivos partiendo de un estado cualquiera.

prego$(state, targets, actions)$
Entrada: un State $state$, una lista de String $target$, una lista de Action $actions$
Salida: un número $result$
1 **retornar longitud de** $prego\_aux(state, targets, actions)$

Fig. 2. Algoritmo de cálculo de la heurística `Prego`

prego_aux$(state, targets, actions)$
Entrada: un State $state$, una lista de String $targets$, una lista de Action $actions$
Salida: una lista $result$ con las mínimas acciones necesarias para satisfacer $targets$ partiendo de state
1 $result \leftarrow []$
2 **Por cada** $target$ **en** $targets$**:**
3 **Si** $target$ no está entre los literales de $state$ **entonces**
4 $causes \leftarrow []$
5 **Por cada** $action$ **en** $actions$
6 **Si** $target$ está entre los efectos de $action$
7 **entonces**
8 $causes \leftarrow causes + [action]$
9 **Si** $causes$ no tiene elementos **entonces**
10 $result \leftarrow result + actions$
11 **Si no entonces**
12 $heuristic \leftarrow []$
13 **Por cada** $cause$ **en** $causes$
14 $temp \leftarrow [cause] + prego\_aux(state,$
15 $cause.preconditions, actions)$
16 $heuristic \leftarrow heuristic + [temp]$
17 $minimun \leftarrow$ lista de $heuristic$
18 con menor longitud
19 $result \leftarrow result + minimun$
20 **retornar** $result$

Fig. 3. Algoritmo auxiliar para el cálculo de la heurística `Prego`

E. Técnica de búsqueda **hacia delante** *(figuras 3 y 4)*

El algoritmo de búsqueda hacia adelante con heurística consiste en, partiendo desde un estado inicial, la evaluación del camino más corto cada vez que se transita desde un estado actual hacia otro estado distinto al aplicar la mejor acción marcada por la heurística. Este se detiene cuando se satisfacen los literales objetivos o cuando tras evaluar todas las posibles soluciones no se ha encontrado ninguna ruta.

forward_search$(initial\_state, target, actions)$
Entrada: un State $initial\_state$,un State $target$, una lista de Action $actions$
Salida: una lista ordenada de acciones $result$ para satisfacer target partiendo de initial_state si se ha encontrado. En otro caso se devuelve un string $result$ que indica que no se ha encontrado camino
1 **retornar** $forward\_search\_aux([], [], initial\_state,$
2 $target, actions)$

Fig. 4. Algoritmo de búsqueda hacia delante `Forward search`

forward_search_aux$(path, visited, current, target, actions)$
Entrada: un State $initial\_state$,una State $target$, una lista de Action $actions$
Salida: una lista ordenada de acciones $result$ para satisfacer $target$ partiendo de $current$ si se ha encontrado. En otro caso se devuelve un string $result$ que indica que no se ha encontrado camino
1 **Si** $current$ satisface los literales de $target$ **entonces**
2 **retornar** $path$
3 $applicable \leftarrow []$
4 **Por cada** $action$ **en** $actions$
5 **Si** $current$ satisface las precondiciones de $action$
6 **y** el estado resultante de aplicar $action$ a $current$
7 no está en $visited$ **entonces**
8 $applicable \leftarrow applicable + [action]$
9 $sorted\_applicable \leftarrow$ acciones en $applicable$ ordenadas
10 por heurísticas
11 **Por cada** $action$ **en** $sorted\_applicable$
12 $e \leftarrow$ estado resultante de aplicar $action$ a $current$
13 $result \leftarrow forward\_search\_aux(path + [action],$
14 $visited + [e], e, target, actions)$
15 **Si** existe un $result$ **entonces**
16 **retornar** $result$
17 **retornar** There is no path from initial state to target

Fig. 5. Algoritmo auxiliar de búsqueda hacia delante `Forward search auxiliar`

F. Técnica búsqueda **hacia atrás** (figuras 5 y 6)

El algoritmo de búsqueda hacia atrás con heurística consiste en, partiendo desde un estado objetivo, la evaluación del camino hacia atrás más corto cada vez que se transita desde un estado actual hacia otro estado distinto al desaplicar la mejor acción marcada por la heurística. Este se detiene cuando se satisfacen única y exclusivamente los literales del estado inicial o cuando tras evaluar todas las posibles soluciones no se ha encontrado ninguna ruta.

backward_search$(initial\_state, target, actions)$
Entrada: un State $initial\_state$,un State $target$ $target$, una lista de Action $actions$
Salida: una lista ordenada de acciones $result$ para satisfacer target partiendo de initial_state si se ha encontrado. En otro caso se devuelve un string $result$ que indica que no se ha encontrado camino

```
1 retornar backward_search_aux([], [], initial_state,
2           target, actions)
```

Fig. 6. Algoritmo de búsqueda hacia atrás `Backward search`

backward_search_aux$(path, visited, current, target, actions)$
Entrada: un State $initial\_state$,una lista de String $target$, una lista de Action $actions$
Salida: una lista ordenada de acciones $result$ para satisfacer $target$ partiendo de $current$ si se ha encontrado. En otro caso se devuelve un string $result$ que indica que no se ha encontrado camino

```
 1 Si current initial_state entonces
 2     retornar path
 3 relevants ← []
 4 Por cada action en actions
 5     Si current satisface alguno de los efectos de action
 6     y el estado resultante de des-aplicar action a current
 7     no contiene a algún estado en visited entonces
 8         relevants ← relevants + [action]
 9 sorted_relevants ← acciones en relevants ordenadas
10                   por heurísticas
11 Por cada action en sorted_relevants
12     e ← estado resultante de des-aplicar action a current
13     result ← backward_search_aux(path + [action],
14               visited + [e], initial_state, e, actions)
15     Si existe un result entonces
16         retornar result
17 retornar There is no path from initial state to target
```

Fig. 7. Algoritmo auxiliar de búsqueda hacia atrás `Backward search auxiliar`

G. Interfaz de usuario

La interfaz de usuario **(figura 7)** consiste en un menú interactivo para el usuario final, en el cual con unos sencillos pasos se puede resolver un problema de PDDL utilizando una de las técnicas anteriormente descritas y una de las dos heurísticas anteriormente descritas.

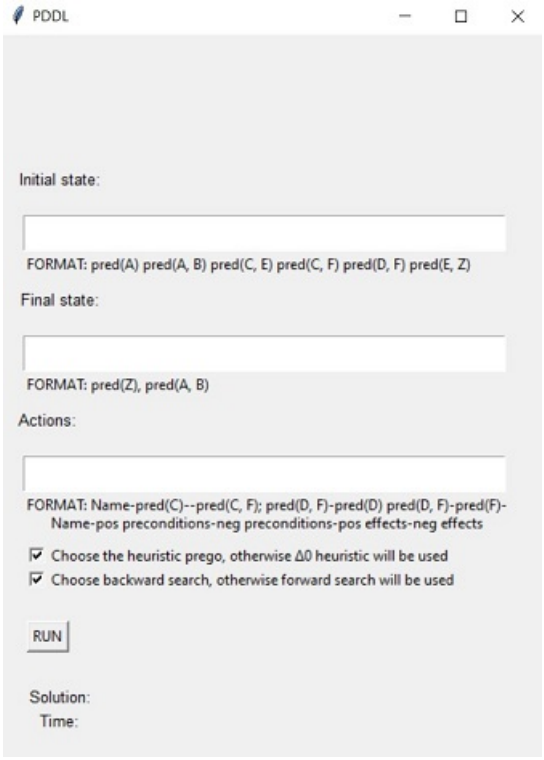

Fig. 8. Interfaz de usuario final

Para resolver un problema de Planificación utilizando la interfaz debemos completar el campo Initial State con el estado inicial del problema, un Final State con el estado final u objetivo del problema y por último el campo Actions donde se especifican las acciones que se pueden realizar con el formato que se especifica debajo de la entrada. A continuación se puede ver una captura de la resolución de un ejemplo **(figura 8)**.

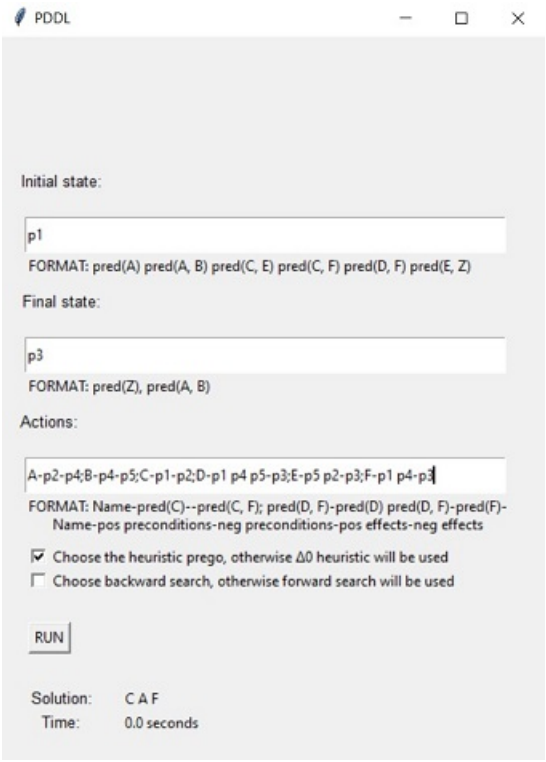

Fig. 9. Ejemplo de uso de la interfaz de usuario final

Se puede observar en la figura 9 la solución encontrada al problema planteado y el tiempo que se ha tardado en resolver(en segundos).
En este caso para los datos introducidos que han sido:

1) **Initial state**: p1
2) **final state**: p3
3) **Actions**: A-p2-p4;B-p4-p5;C-p1-p2;D-p1,p4,p5-p3;E-p5,p2-p3;F-p1,p4-p3
4) **Heurística utilizada**: Heurística ”prego”
5) **Técnica de búsqueda utilizada**: Búsqueda hacia delante

Se ha obtenido la solución **C A F** con un tiempo de ejecución de **0.0 segundos**.

IV. Pruebas realizadas y resultados obtenidos

En esta sección se detallarán tanto los experimentos realizados como los resultados conseguidos para comprobar la fiabilidad y rendimiento de los métodos de resolución implementados.
Todas las pruebas se han llevado a cabo con la finalidad de resolver un problema de planificación. Las primeras pruebas realizadas fueron utilizando únicamente las heurísticas implementadas y literales.
Se detallarán a continuación:

1) **Primera prueba**: Prueba sencilla de funcionamiento para heurística ”prego”
   - **Estado inicial**: p1
   - **Objetivo**: p3
   - **Acciones**: A-p2-p4;B-p4-p5;C-p1-p2;D-p1 p4 p5-p3;E-p5 p2-p3;F-p1 p4-p3
   - **Resultado obtenido**: 3
2) **Segunda prueba**: Prueba de cierta complejidad de funcionamiento para heurística ”prego”
   - **Estado inicial**: p1 p7 p14
   - **Objetivo**: p20 p13 p17
   - **Acciones**: A-p1-p4 p8;B-p4-p20 p14 p16;C-p18 p20-p5;D-p7 p3-p6;E-p1-p7 p2;F-p5-p10 p13;G-p9 p13 p12-p11 p19;H-p14 p8-p17;I-p14-p5;J-p15-p21
   - **Resultado obtenido**: 5
3) **Tercera prueba**: Prueba sencilla de funcionamiento para heurística ”$\Delta 0$”
   - **Estado inicial**: p1
   - **Objetivo**: p3
   - **Acciones**: A-p2-p4;B-p4-p5;C-p1-p2;D-p1 p4 p5-p3;E-p5 p2-p3;F-p1 p4-p3
   - **Resultado obtenido**: 3
4) **Cuarta prueba**: Prueba de cierta complejidad de funcionamiento para heurística ”$\Delta 0$”
   - **Estado inicial**: p1 p7 p14
   - **Objetivo**: p20 p13 p17
   - **Acciones**: A-p1-p4 p8;B-p4-p20 p14 p16;C-p18 p20-p5;D-p7 p3-p6;E-p1-p7 p2;F-p5-p10 p13;G-p9 p13 p12-p11 p19;H-p14 p8-p17;I-p14-p5;J-p15-p21
   - **Resultado obtenido**: 5
5) **Quinta prueba**: Prueba sencilla de funcionamiento para técnica de búsqueda hacia delante con heurística ”prego”
   - **Estado inicial**: p1
   - **Objetivo**: p3
   - **Acciones**: A-p2-p4;B-p4-p5;C-p1-p2;D-p1 p4 p5-p3;E-p5 p2-p3;F-p1 p4-p3
   - **Resultado obtenido**: C A F
   - **Tiempo de ejecución**: 0.0 segundos
6) **Sexta prueba**: Prueba sencilla de funcionamiento para técnica de búsqueda hacia delante con heurística ”$\Delta 0$”
   - **Estado inicial**: p1
   - **Objetivo**: p3
   - **Acciones**: A-p2-p4;B-p4-p5;C-p1-p2;D-p1 p4 p5-p3;E-p5 p2-p3;F-p1 p4-p3
   - **Resultado obtenido**: C A F
   - **Tiempo de ejecución**: 0.0 segundos
7) **Séptima prueba**: Prueba sencilla de funcionamiento para técnica de búsqueda hacia atrás con heurística ”prego”
   - **Estado inicial**: p1
   - **Objetivo**: p3
   - **Acciones**: A-p2-p4;B-p4-p5;C-p1-p2;D-p1 p4 p5-p3;E-p5 p2-p3;F-p1 p4-p3
   - **Resultado obtenido**: C A F
   - **Tiempo de ejecución**: 0.00098 segundos

8) **Octava prueba**: Prueba sencilla de funcionamiento para técnica de búsqueda hacia atrás con heurística "Δ0"

   - **Estado inicial**: p1
   - **Objetivo**: p3
   - **Acciones**: A-p2-p4;B-p4-p5;C-p1-p2;D-p1 p4 p5-p3;E-p5 p2-p3;F-p1 p4-p3
   - **Resultado obtenido**: C A F
   - **Tiempo de ejecución**: 0.0 segundos

9) **Novena prueba**: Prueba de cierta complejidad de funcionamiento para técnica de búsqueda hacia delante con heurística "prego"

   - **Estado inicial**: p1 p7 p14
   - **Objetivo**: p20 p13 p17
   - **Acciones**: A-p1-p4 p8;B-p4-p20 p14 p16;C-p18 p20-p5;D-p7 p3-p6;E-p1-p7 p2;F-p5-p10 p13;G-p9 p13 p12-p11 p19;H-p14 p8-p17;I-p14-p5;J-p15-p21
   - **Resultado obtenido**: A H B I F
   - **Tiempo de ejecución**: 0.0 segundos

10) **Décima prueba**: Prueba de cierta complejidad de funcionamiento para técnica de búsqueda hacia delante con heurística "Δ0"

    - **Estado inicial**: p1 p7 p14
    - **Objetivo**: p20 p13 p17
    - **Acciones**: A-p1-p4 p8;B-p4-p20 p14 p16;C-p18 p20-p5;D-p7 p3-p6;E-p1-p7 p2;F-p5-p10 p13;G-p9 p13 p12-p11 p19;H-p14 p8-p17;I-p14-p5;J-p15-p21
    - **Resultado obtenido**: A I H F B
    - **Tiempo de ejecución**: 0.0 segundos

11) **Undécima prueba**: Prueba de cierta complejidad de funcionamiento para técnica de búsqueda hacia atrás con heurística "Prego"

    - **Estado inicial**: p1 p7 p14
    - **Objetivo**: p20 p13 p17
    - **Acciones**: A-p1-p4 p8;B-p4-p20 p14 p16;C-p18 p20-p5;D-p7 p3-p6;E-p1-p7 p2;F-p5-p10 p13;G-p9 p13 p12-p11 p19;H-p14 p8-p17;I-p14-p5;J-p15-p21
    - **Resultado obtenido**:
    - **Tiempo de ejecución**: 0.0 segundos

12) **Duodécima prueba**: Prueba de cierta complejidad de funcionamiento para técnica de búsqueda hacia atrás con heurística "Δ0"

    - **Estado inicial**: p1 p7 p14
    - **Objetivo**: p20 p13 p17
    - **Acciones**: A-p1-p4 p8;B-p4-p20 p14 p16;C-p18 p20-p5;D-p7 p3-p6;E-p1-p7 p2;F-p5-p10 p13;G-p9 p13 p12-p11 p19;H-p14 p8-p17;I-p14-p5;J-p15-p21
    - **Resultado obtenido**:
    - **Tiempo de ejecución**: 0.0 segundos

TABLA I
PRUEBAS CON PROBLEMAS SENCILLOS

	Tiempo de ejecución	Heurística utilizada	Técnica utilizada
Prueba 1	-	Prego	-
Prueba 2	-	Prego	-
Prueba 3	-	Δ0	-
Prueba 4	-	Δ0	-
Prueba 5	0.0 segundos	Prego	Hacia delante
Prueba 6	0.0 segundos	Δ0	Hacia delante
Prueba 7	0.00098 segundos	Prego	Hacia atrás
Prueba 8	0.0 segundos	Δ0	Hacia atrás
Prueba 9	0.0 segundos	Prego	Hacia delante
Prueba 10	0.00098 segundos	Δ0	Hacia delante
Prueba 11	0.0 segundos	Prego	Hacia atrás
Prueba 12	0.0 segundos	Δ0	Hacia atrás

Pruebas utilizando acciones complejas: El problema planteado para probar la eficacia de las técnicas de búsqueda implementadas ha sido obtener cual es el camino mas corto que lleva desde una habitación X hasta una habitación Y pasando por múltiples habitaciones cuyas entradas pueden estar bloqueadas. Para reducir la complejidad del problema y posibilitar la aplicación de heurísticas se ha impuesto como técnica de relajación el hecho de que no se pueda regresar a una habitación ya visitada para la ruta actual, además se ha tomado un sentido posible en los caminos para evitar que se produzcan ciclos.

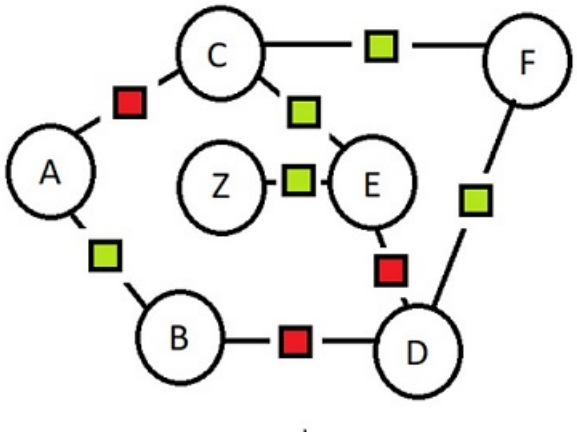

Fig. 10. Estado inicial problema del robot

1) **Primera prueba**: Prueba con acciones complejas para técnica de búsqueda hacia delante con heurística "Prego"
   - **Estado inicial**: en(A) abierta(A,B) abierta(C,E) abierta(C,F) abierta(D,F) abierta(E,Z)
   - **Objetivo**: en(Z)
   - **Acciones**: abrir_puerta(A,B)-en(A)-abierta(A,B)-abierta(A,B)-;abrir_puerta(B,D)-en(B)-abierta(B,D)-abierta(B,D)-;ir(D,E)-abierta(D,E) en(D)–en(E)-en(D);abrir_puerta(E,Z)-en(E)-abierta(E,Z)-abierta(E,Z)-;ir(B,D)-abierta(B,D) en(B)–en(D)-en(B);ir(A,C)-abierta(A,C) en(A)–en(C)-en(A);abrir_puerta(C,F)-en(C)-abierta(C,F)-abierta(C,F)-;ir(E,Z)-en(E) abierta(E,Z)–en(Z)-en(E);ir(C,F)-en(C) abierta(C,F)–en(F)-en(C);abrir_puerta(A,C)-en(A)-abierta(A,C)-abierta(A,C)-;abrir_puerta(D,F)-en(D)-abierta(D,F)-abierta(D,F)-;ir(A,B)-en(A) abierta(A,B)–en(B)-en(A);abrir_puerta(C,E)-en(C)-abierta(C,E)-abierta(C,E)-;ir(D,F)-abierta(D,F) en(D)–en(F)-en(D);ir(C,E)-en(C) abierta(C,E)–en(E)-en(C);abrir_puerta(D,E)-en(D)-abierta(D,E)-abierta(D,E)-;
   - **Resultado obtenido**: abrir_puerta(A, C) ir(A, C) ir(C, E) ir(E, Z)
   - **Tiempo de ejecución**: 0.001 segundos

2) **Segunda prueba**: Prueba con acciones complejas para técnica de búsqueda hacia delante con heurística "$\Delta 0$".
   - **Estado inicial**: **en(F)** abierta(A,B) abierta(C,E) abierta(C,F) abierta(D,F) abierta(E,Z)
   - **Objetivo**: **en(Z)**
   - **Acciones**: abrir_puerta(A,B)-en(A)-abierta(A,B)-abierta(A,B)-;abrir_puerta(B,D)-en(B)-abierta(B,D)-abierta(B,D)-;ir(D,E)-abierta(D,E) en(D)–en(E)-en(D);abrir_puerta(E,Z)-en(E)-abierta(E,Z)-abierta(E,Z)-;ir(B,D)-abierta(B,D) en(B)–en(D)-en(B);ir(A,C)-abierta(A,C) en(A)–en(C)-en(A);abrir_puerta(C,F)-en(C)-abierta(C,F)-abierta(C,F)-;ir(E,Z)-en(E) abierta(E,Z)–en(Z)-en(E);ir(C,F)-en(C) abierta(C,F)–en(F)-en(C);abrir_puerta(A,C)-en(A)-abierta(A,C)-abierta(A,C)-;abrir_puerta(D,F)-en(D)-abierta(D,F)-abierta(D,F)-;ir(A,B)-en(A) abierta(A,B)–en(B)-en(A);abrir_puerta(C,E)-en(C)-abierta(C,E)-abierta(C,E)-;ir(D,F)-abierta(D,F) en(D)–en(F)-en(D);ir(C,E)-en(C) abierta(C,E)–en(E)-en(C);abrir_puerta(D,E)-en(D)-abierta(D,E)-abierta(D,E)-;ir(F,C)-en(F) abierta(C,F)–en(C)-en(F);
   - **Resultado obtenido**:ir(F, C) ir(C, E) ir(E, Z)
   - **Tiempo de ejecución**: 0.001 segundos

3) **Tercera prueba**: Prueba con acciones complejas para técnica de búsqueda hacia atrás con heurística "prego".
   - **Estado inicial**: **en(A)** abierta(A,B) abierta(C,E) abierta(C,F) abierta(D,F) abierta(E,Z)
   - **Objetivo**: **en(D)**
   - **Acciones**: abrir_puerta(A,B)-en(A)-abierta(A,B)-abierta(A,B)-;abrir_puerta(B,D)-en(B)-abierta(B,D)-abierta(B,D)-;ir(D,E)-abierta(D,E) en(D)–en(E)-en(D);abrir_puerta(E,Z)-en(E)-abierta(E,Z)-abierta(E,Z)-;ir(B,D)-abierta(B,D) en(B)–en(D)-en(B);ir(A,C)-abierta(A,C) en(A)–en(C)-en(A);abrir_puerta(C,F)-en(C)-abierta(C,F)-abierta(C,F)-;ir(E,Z)-en(E) abierta(E,Z)–en(Z)-en(E);ir(C,F)-en(C) abierta(C,F)–en(F)-en(C);abrir_puerta(A,C)-en(A)-abierta(A,C)-abierta(A,C)-;abrir_puerta(D,F)-en(D)-abierta(D,F)-abierta(D,F)-;ir(A,B)-en(A) abierta(A,B)–en(B)-en(A);abrir_puerta(C,E)-en(C)-abierta(C,E)-abierta(C,E)-;ir(D,F)-abierta(D,F) en(D)–en(F)-en(D);ir(C,E)-en(C) abierta(C,E)–en(E)-en(C);abrir_puerta(D,E)-en(D)-abierta(D,E)-abierta(D,E)-;ir(F,C)-en(F) abierta(C,F)–en(C)-en(F);
   - **Resultado obtenido**:ir(A, B) abrir_puerta(B, D) ir(B, D)
   - **Tiempo de ejecución**: 0.00099 segundos

4) **Cuarta prueba**: Prueba con acciones complejas para técnica de búsqueda hacia atrás con heurística "$\Delta 0$".
   - **Estado inicial**: **en(D)** abierta(A,B) abierta(C,E) abierta(C,F) abierta(D,F) abierta(E,Z)
   - **Objetivo**: **en(Z)**
   - **Acciones**: abrir_puerta(A,B)-en(A)-abierta(A,B)-abierta(A,B)-;abrir_puerta(B,D)-en(B)-abierta(B,D)-abierta(B,D)-;ir(D,E)-abierta(D,E) en(D)–en(E)-en(D);abrir_puerta(E,Z)-en(E)-abierta(E,Z)-abierta(E,Z)-;ir(B,D)-abierta(B,D) en(B)–en(D)-en(B);ir(A,C)-abierta(A,C) en(A)–en(C)-en(A);abrir_puerta(C,F)-en(C)-abierta(C,F)-abierta(C,F)-;ir(E,Z)-en(E) abierta(E,Z)–en(Z)-en(E);ir(C,F)-en(C) abierta(C,F)–en(F)-en(C);abrir_puerta(A,C)-en(A)-abierta(A,C)-abierta(A,C)-;abrir_puerta(D,F)-en(D)-abierta(D,F)-abierta(D,F)-;ir(A,B)-en(A) abierta(A,B)–en(B)-en(A);abrir_puerta(C,E)-en(C)-abierta(C,E)-abierta(C,E)-;ir(D,F)-abierta(D,F) en(D)–en(F)-en(D);ir(C,E)-en(C) abierta(C,E)–en(E)-en(C);abrir_puerta(D,E)-en(D)-abierta(D,E)-abierta(D,E)-;
   - **Resultado obtenido**:abrir_puerta(D, E) ir(D, E) ir(E, Z)
   - **Tiempo de ejecución**: 0.0 segundos

TABLA II
PRUEBAS CON EL PROBLEMA DEL ROBOT

	Tiempo de ejecución	Heurística utilizada	Técnica utilizada
Prueba 1	0.001 segundos	Prego	Hacia delante
Prueba 2	0.001 segundos	$\Delta 0$	Hacia delante
Prueba 3	0.00099 segundos	Prego	Hacia atrás
Prueba 4	0.0 segundos	$\Delta 0$	Hacia atrás

V. CONCLUSIONES

Las conclusiones obtenidas tras el análisis del problema, el diseño e implementación de la resolución del mismo y la experimentación con distintos conjuntos de datos es que la nueva heurística ofrecida es similar en cuanto a eficiencia a $\Delta 0$.

Desde un punto de vista teórico es fácilmente observable que el cálculo de caminos óptimos es idéntico para ambas heurísticas. La única diferencia entre ambas es el tipo de datos con el que se opera. En el caso de $prego$ se utilizan listas y en el caso de $\Delta 0$ se utilizan números enteros.

En Python, las operaciones de concatenación de listas y de obtención de longitud de la misma tienen una complejidad algorítmica constante O(1) según la wiki del lenguaje [5].

Lo mismo pasa para operaciones sobre enteros. Sumar 2 números es algorítmicamente constante en cuanto al tiempo de ejecución O(1).

El número de operaciones es el mismo excepto para el cálculo de prego, en el que se añade una instrucción final para retornar el tamaño de la lista obtenida en el método auxiliar.

Desde un punto de vista práctico se observa una sutil diferencia a favor de $\Delta 0$ (despreciable teniendo en cuenta que las condiciones de ejecución no son ideales y que no va a crecer según se aumente el tamaño del problema) .

REFERENCIAS

[1] Libreria Tkinker
[2] Definición de $\Delta 0$. Presentación del tema.
[3] Problema de la mochila
[4] Problema de las n-Reinas
[5] Complejidad temporal en operaciones sobre listas