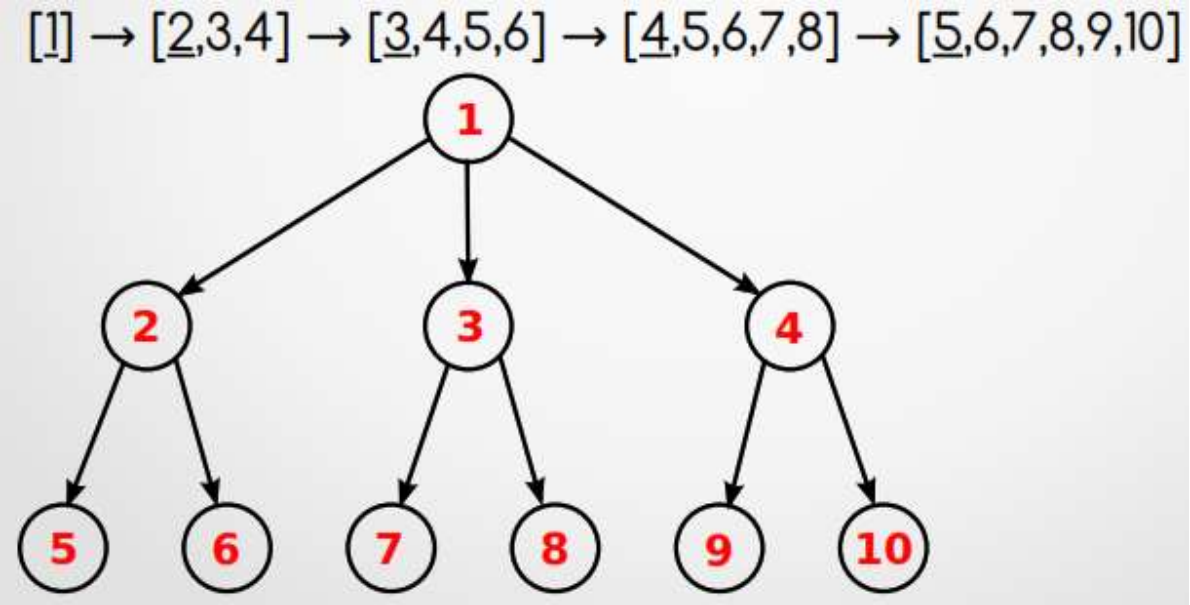
1.1 BÚSQUEDA NO INFORMADA

La búsqueda no informada es toda aquella que no se tiene mucha información o ningún tipo de heurística que guie, solo se aporta como información la definición del problema y lo único que queda es expandir nodos y comprobar si alguno de ellos es la solución al problema. Este tipo de algoritmos se diferencian en cuanto al siguiente nodo a expandir.

1.1.1 BÚSQUEDA PRIMERO EN ANCHURA (BREAD-FIRST SEARCH)

Se expande el nodo raíz, luego todos sus nodos hijos, luego los hijos de los hijos, hasta encontrar la solución. Lo más importante de este algoritmo es que se expande cada nivel antes de expandir los del siguiente nivel. Su implementación hace uso de una lista FIFO. Su funcionamiento se puede explicar a través del siguiente gráfico.



Donde se muestra como ingresan y salen los nodos de la lista frontera FIFO, además del criterio de expansión de nodos hijos.

MEDIDAS DE RENDIMIENTO:

Este algoritmo:

- Es completo, es decir si existe solución, la encuentra, por que termina visitando todos los nodos.
- Es optimo, por que la solución encontrada es la mas superficial (condición: coste de acciones iguales y no negativas)
- Tiene la siguiente complejidad:
    - Tiempo, exponencial $O(b^d) = b^d + \dots + b^2 + b + 1$.
    - Espacio, exponencial $O(b^d)$ en frontera y $O(b^{d-1})$ en explorada

El principal problema de este algoritmo cuando se aplica a problemas de la vida real es la cantidad de espacio de memoria y tiempo que requerirían para aplicarlos, por lo que no son aplicables a este ámbito de problemas. Los siguientes algoritmos buscarán mejorar los criterios de complejidad, a costos de ser completos u óptimos.

El pseudocódigo de este algoritmo es el siguiente:

```
función BÚSQUEDA-PRIMERO-ANCHURA(problema)
devuelve solución o fallo
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  si problema.ES-OBJETIVO(nodo-raíz.ESTADO) entonces
    devolver nodo-raíz
  frontera ← CREAR-FIFO()
  frontera.AGREGAR(nodo-raíz)
  explorada ← CREAR-CONJUNTO()
  repetir
     si frontera.ESTÁ-VACÍA() entonces devolver fallo
     nodo ← frontera.POP()
     explorada.AGREGAR(nodo)
     por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
       hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
       si hijo.ESTADO no está en explorada y
          hijo.ESTADO no está en frontera.ESTADOS() entonces
            si problema.ES-OBJETIVO(hijo.ESTADO) entonces
              devolver hijo
            frontera.AGREGAR(hijo)
```

VENTAJAS:

- Si hay solución, la encuentra
- Encuentra la solución óptima

DESVENTAJAS:

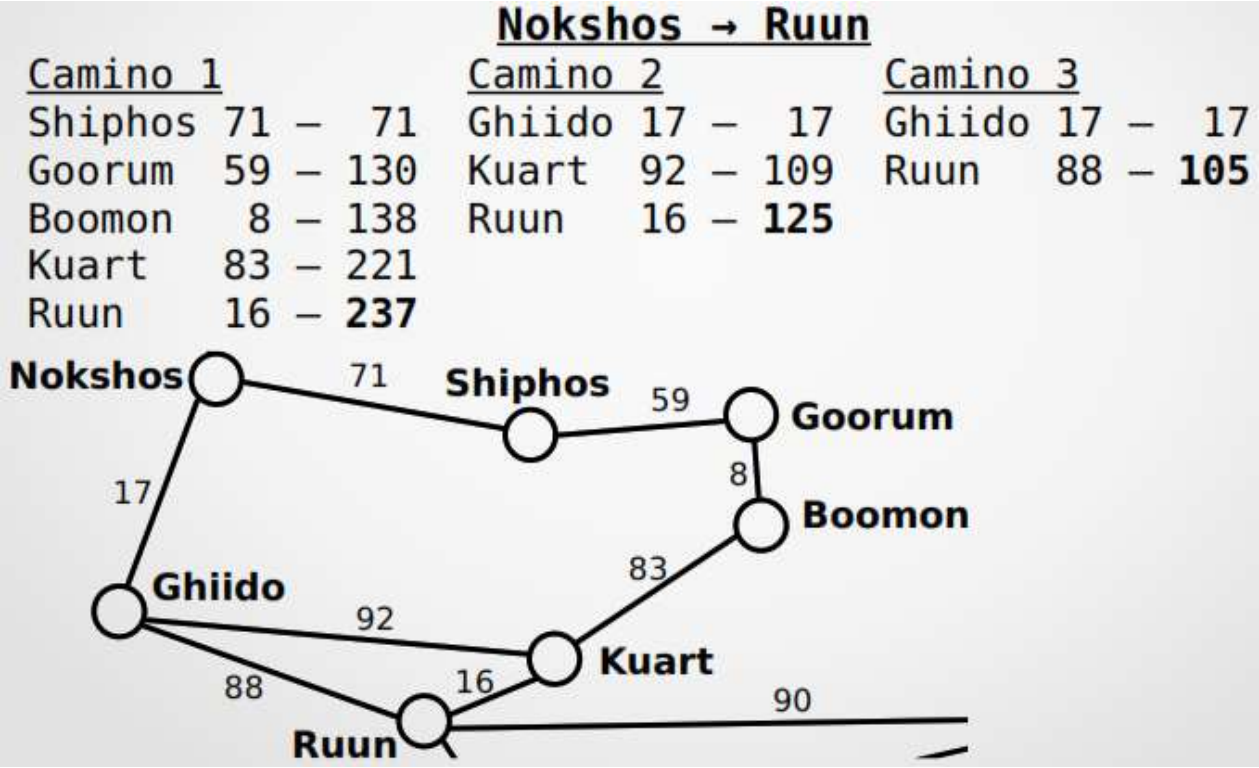
- Expande muchos nodos inútiles.
- Orden exponencial en espacio.
- Costo constante y no negativo
- Solo para problemas muy simples

IMPLEMENTACIÓN EN PYTHON

Ver el archivo bpa_01.py

1.1.2 BÚSQUEDA DE COSTE UNIFORME (UNIFORM-COST SEARCH)

El algoritmo de búsqueda en anchura tiene como mayor dificultad el coste computacional en relación a espacio (memoria), además otro elemento que no consideraba eran los costes, es decir para trabajar con este algoritmo todos los costes tenían que ser constantes. El algoritmo de búsqueda de costo uniforme, está basado en el algoritmo de búsqueda primero en anchura para trabajar con acciones donde cada una tiene un costo variable, no negativo. Su estrategia es expandir el nodo con menor coste asociado (no el mas superficial), para lo cual se reemplaza la lista LIFO por una lista de prioridad, en la cual se ordenan los nodos según coste, es decir el nodo con menor coste es el que se va a extraer. Dentro de los problemas que intenta solucionar el algoritmo de coste uniforme esta el de las ciudades y los caminos que las une, donde se busca el camino con menor coste. Problema que se representa con el siguiente gráfico:



Donde se muestra como la expansión de nodos empleada en el algoritmo primero en anchura, no sirve para el caso de que cada acción de un nodo tiene asociado un coste, y que la solución ya no será la que se encuentre más superficial en el árbol, siendo este el tipo de problemas que se pueden abordar con este tipo de algoritmo.

MEDIDAS DE RENDIMIENTO:

Este algoritmo:

- Es completo, es decir si existe solución, la encuentra, comprobando al expandir y no al generar.
- Es optimo, por que obtiene el camino con el menor coste.
- Tiene la siguiente complejidad:

- Tiempo, exponencial $O(b^{1 + C^* / e})$, C*coste camino óptimo, e coste minmo de acción. Puede ser superior a $O(b^d)$, coste constante -> C* / e = d -> $O(b^{d - 1})$
- Espacio, exponencial $O(b^{1+ [C^*/ e]})$

El pseudocódigo de este algoritmo es el siguiente:

```
función BÚSQUEDA-COSTE-UNIFORME(problema) devuelve solución o fallo
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  frontera ← CREAR-PRIORIDAD()
  frontera.AGREGAR(nodo-raíz)
  explorada ← CREAR-CONJUNTO()
  repetir
     si frontera.ESTÁ-VACÍA() entonces devolver fallo
     nodo ← frontera.POP()
     si problema.ES-OBJETIVO(nodo.ESTADO) entonces devolver nodo
     explorada.AGREGAR(nodo)
     por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
       hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
       si hijo.ESTADO no está en explorada y
          hijo.ESTADO no está en frontera.ESTADOS() entonces
            frontera.AGREGAR(hijo)
       sino
          nodo-frontera ← frontera.BUSCAR(hijo.ESTADO)
          si hijo.COSTE < nodo-frontera.COSTE entonces
            nodo-frontera ← hijo
```

Su lógica es similar al algoritmo primero en anchura, con la diferencia de que los elementos de frontera se los extrae en función a una prioridad, es decir el que tiene menor coste asociado. Realizar la comprobación de que si el nodo actual es el nodo objetivo fuera después de extraer el nodo de la frontera y no dentro del bucle y finalmente incorporar una función para buscar nodo en la lista frontera un nodo similar pero con mayor coste al nodo actual y a partir de ello reemplazar el antiguo nodo con el nuevo nodo que tiene menor costo.

VENTAJAS:

- Si hay solución, la encuentra
- Encuentra la solución optima
- Admite costes variables

DESVENTAJAS:

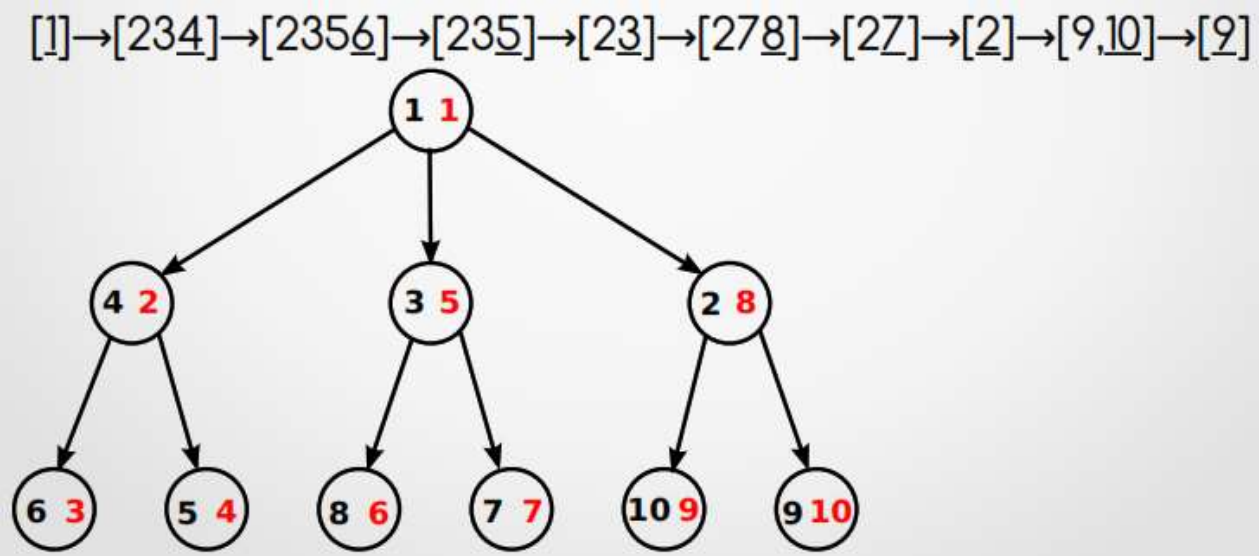
- Orden exponencial en espacio.
- Costes no negativos
- Solo para problemas muy simples

IMPLEMENTACION EN PYTHON

Ver el archivo bcu_01.py

1.1.3 BÚSQUEDA PRIMERO EN PROFUNDIDAD (DEPTH-FIRST SEARCH)

Se expande el nodo raíz, luego uno de sus nodos hijos, luego uno de los hijos del hijo, cuando se llega a una hoja, sino es solución, se retrocede y se prueba con el siguiente hijo hasta encontrar la solución. Se expande el nodo más profundo de la frontera. Su funcionamiento se puede explicar a través del siguiente gráfico.



Donde se muestra como ingresan y salen los nodos de la lista frontera LIFO o a través de un método recursivo, además del criterio de expansión de nodos hijos. Es importante prestar atención al algoritmo que considera recursividad, pues los siguientes algoritmos en su mayoría implementan esta técnica. El numero en negro de cada nodo indica el orden en el que se generan y el numero en rojo el orden en que se visitan.

MEDIDAS DE RENDIMIENTO:

Este algoritmo:

- Es completo, si evita caminos redundantes y el espacio de estados es finito. Sin embargo, el comprobar estados repetidos no evita los caminos redundantes. En un espacio de estados infinito no es completo porque no podrá encontrar nunca la solución. Por las características de los problemas se puede mencionar que no es completo.
- No es óptimo, porque puede encontrar otras soluciones antes de la óptima.
- Tiene la siguiente complejidad:
    - Tiempo, exponencial $O(b^m)$, donde m puede ser mayor que d (incluso puede ser infinito).Como mínimo se tardara el tiempo que emplea la búsqueda primero en anchura, si se tiene suerte, caso contrario el tiempo será mayor.

- Espacio, lineal O(b*m), ya que solo almacena el camino y los nodos hijos de los nodos intermedios. Si se expande solo un nodo hijo: O(m), si no hace falta guardar el camino: O(1), en estos dos casos no hay frontera. Definitivamente con este algoritmo se logra superar la limitante de la complejidad espacial, llegando de una exponencial a una lineal.

No olvidemos que este algoritmo esta descrito en su funcionamiento básico, posteriormente se realizarán adecuaciones para superar sus limitaciones o dificultades.

El principal problema de este algoritmo cuando se aplica a problemas de la vida real es la cantidad de espacio de memoria y tiempo que requerirían para aplicarlos, por lo que no son aplicables a este ámbito de problemas. Los siguientes algoritmos buscaran mejorar los criterios de complejidad, a costos de ser completos u óptimos.

El pseudocódigo de este algoritmo utilizando una lista LIFO, es el siguiente:

```
función BÚSQUEDA-PRIMERO-PROFUNDIDAD(problema)
devuelve solución o fallo
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  si problema.ES-OBJETIVO(nodo-raíz.ESTADO) entonces
    devolver nodo-raíz
  frontera ← CREAR-LIFO()
  frontera.AGREGAR(nodo-raíz)
  explorada ← CREAR-CONJUNTO()
  repetir
     si frontera.ESTÁ-VACÍA() entonces devolver fallo
     nodo ← frontera.POP()
     explorada.AGREGAR(nodo)
     por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
       hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
       si hijo.ESTADO no está en explorada y
          hijo.ESTADO no está en frontera.ESTADOS() entonces
            si problema.ES-OBJETIVO(hijo.ESTADO) entonces
              devolver hijo
            frontera.AGREGAR(hijo)
```

Muy similar al algoritmo de búsqueda primero en anchura, solo cambia FIFO por LIFO.

El pseudocódigo de este algoritmo utilizando recursividad, es el siguiente:

```
función BÚSQUEDA-PRIMERO-PROFUNDIDAD(problema)
devuelve solución o fallo
  explorada ← CREAR-CONJUNTO()
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  devuelve BPP-RECURSIVA(nodo-raíz, problema, explorada)

función BPP-RECURSIVA(nodo, problema, explorada)
devuelve solución o fallo
  si problema.ES-OBJETIVO(nodo.ESTADO) entonces
    devolver nodo
  por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
     hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
     si hijo.ESTADO no está en explorada entonces
       resultado ← BPP-RECURSIVA(hijo, problema, explorada)
       si resultado ≠ fallo entonces devolver resultado
  devolver fallo
```

VENTAJAS:

- Ocupa muy poco espacio.

DESVENTAJAS:

- No es completo ni óptimo.
- Puede probar muchos caminos inútiles.
- Puede quedar atrapado en bucles infinitos.
- Coste constante y no negativo.
- Solo para problemas muy simples.

Estas búsquedas solo podrán eliminar las desventajas aplicando heurísticas.

IMPLEMENTACION EN PYTHON

Ver el archivo bpp_01.py para la implementación regular y bpp_02.py, para la implementación utilizando recursividad.

1.1.4 BÚSQUEDA EN PROFUNDIDAD LIMITADA (DEPTH-LIMITED SEARCH)

Este algoritmo es un paso intermedio entre la búsqueda primero en profundidad y aquél algoritmo que sería completo y óptimo. Se basa en el algoritmo de búsqueda primero en profundidad, estableciendo una profundidad máxima (p) como diámetro del espacio de estados (camino máximo), cuando se llega a la profundidad máxima p, si no hay solución, se retrocede. Se expande el nodo más profundo de la frontera hasta el límite. Es importante mencionar que si bien el algoritmo primero en profundidad tiene un problema importante en relación a la cantidad de espacio de estados este puede ser infinitos o generar bucles o ramas demasiado extensas, sin encontrar ninguna solución. Siendo uno de los objetivos del presente algoritmo atacar los aspectos detallados anteriormente.

MEDIDAS DE RENDIMIENTO:

Este algoritmo:

- No es completo, las soluciones pueden estar más allá del límite máximo p.
- No es óptimo, puede encontrar otras soluciones antes que la óptima, que podría estar más por debajo del límite máximo p.
- Tiene la siguiente complejidad:
  - Tiempo, exponencial $O(b^p)$
  - Espacio, lineal O(b*p), O(p), O(1)

El pseudocódigo de este algoritmo es el siguiente:

```
función BÚSQUEDA-PROFUNDIDAD-LIMITADA(problema, límite)
devuelve solución o fallo o corte
  explorada ← CREAR-CONJUNTO()
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  devuelve BPL-RECURSIVA(nodo-raíz, problema, límite, explorada)

función BPL-RECURSIVA(nodo, problema, límite, explorada)
devuelve solución o fallo o corte
  si problema.ES-OBJETIVO(nodo.ESTADO) entonces devolver nodo
  si límite = 0 entonces devolver corte
  por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
     hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
     si hijo.ESTADO no está en explorada entonces
       resultado ← BPL-RECURSIVA(hijo, problema,
                                 límite – 1, explorada)
       si resultado ≠ fallo entonces devolver resultado
  devolver fallo
```

VENTAJAS:

- Ocupa muy poco espacio
- No cae en bucles infinitos

DESVENTAJAS:

- No es completo ni óptimo.
- Puede probar muchos caminos inútiles.
- Coste constante y no negativo
- Solo para problemas simples

IMPLEMENTACION EN PYTHON

Ver el archivo bpl_01.py

1.1.5 BÚSQUEDA EN PROFUNDIDAD ITERATIVA (ITERATIVE DEEPENING SEARH)

Este algoritmo es una ampliación del algoritmo de profundidad limitada, ampliando gradualmente el límite máximo p. Se va ampliando el limite hasta alcanzar una solución, que además será la óptima.

MEDIDAS DE RENDIMIENTO:

Este algoritmo:

- Es completo, es decir si existe solución, la encuentras.
- Es óptimo, porque cuando encuentra una solución es la más superficial.
- Tiene la siguiente complejidad:
  - Tiempo, exponencial Espacio, lineal $O(b^d)$, $(d) * b + (d - 1) * b^2 + \ldots + (2) * b^{d-1} + (1) * b^d$
  - Espacio, lineal O(b*d), O(p), O(1)

El pseudocódigo de este algoritmo es el siguiente:

```
función BÚSQUEDA-PROFUNDIDAD-ITERATIVA(problema)
devuelve solución o fallo
  para límite de 0 a máximo hacer
    resultado ← BÚSQUEDA-PROFUNDIDAD-LIMITADA(problema, límite)
    si resultado = solución entonces devolver solución
  devolver fallo
```

VENTAJAS:

- Ocupa muy poco espacio
- No cae en bucles infinitos
- Es completo y optimo

DESVENTAJAS:

- Puede probar muchos caminos inútiles
- Visita muchas veces los nodos superficiales
- Coste constante y no negativo.

El algoritmo de búsqueda en profundidad iterativa es el mejor algoritmo de búsqueda no informada, sobre todo si el espacio de búsqueda es muy grande y no se conoce a que profundidad esta la solución. Una mejora sugerida es utilizar primero en anchura hasta casi este agotada la memoria y luego utilizar búsqueda en profundidad iterativa desde cada uno de los nodos frontera, esto evita visitar los nodos superficiales de nuevo.

IMPLEMENTACION EN PYTHON

1.1.6 BÚSQUEDA DE COSTE ITERATIVO

Este algoritmo es una ampliación del algoritmo de profundidad iterativa, que considera que los costes de las acciones son variables, pero no negativas.

El pseudocódigo de este algoritmo es el siguiente:

```
función BÚSQUEDA-COSTE-ITERATIVO(problema)
devuelve solución o fallo
  para límite de 0 a ∞ hacer # Límite en coste, no en profundidad
    resultado ← BÚSQUEDA-COSTE-LIMITADO(problema, límite)
    si resultado = corte entonces devolver fallo
    devolver resultado

función BÚSQUEDA-COSTE-LIMITADO(problema, límite)
devuelve solución o fallo o corte
  explorada ← CREAR-CONJUNTO()
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  devuelve BCL-RECURSIVO(nodo-raíz, problema, explorada, límite)
```

```
función BCL-RECURSIVO(nodo, problema, explorada, límite)
devuelve solución o fallo
  si problema.ES-OBJETIVO(nodo.ESTADO) entonces devolver nodo
  si límite ≤ 0 entonces devolver fallo
  por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
     hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
     si hijo.ESTADO no está en explorada entonces
       resultado ← BCL-RECURSIVA(hijo, problema, explorada,
                                 límite – hijo.COSTE-ACCIÓN)
       si resultado ≠ fallo entonces devolver resultado
  devolver fallo
```

Los algoritmos son muy similares a los anteriores con pequeñas modificaciones debidas al efecto que ejerce el costo. Sin embargo, la solución podría no ser la óptima, pero estaría muy cerca de ella porque se encontraría a la misma profundidad. Con alguna modificación se podría garantizar que encuentre la solución óptima, sin embargo, es mejor asumir algoritmos que empleen heurísticas.

VENTAJAS:

- Ocupa muy poco espacio.
- No cae en bucles infinitos.
- Es completo y óptimo (o muy cerca).

DESVENTAJAS:

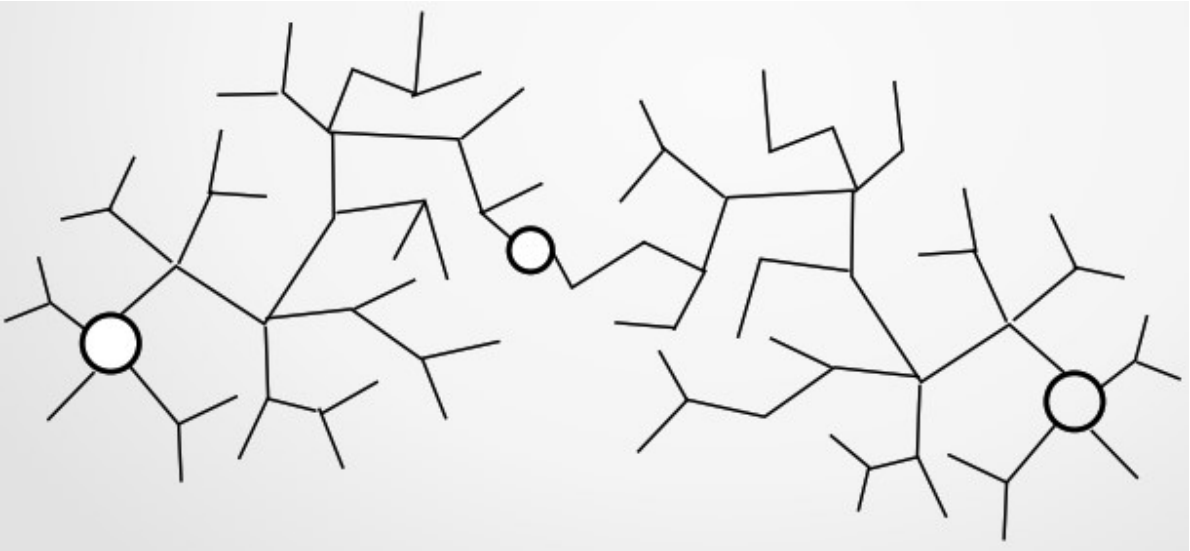
- Puede probar muchos caminos inútiles, por lo cual puede ser que no encuentre la solución óptima.
- Costes no negativos.

IMPLEMENTACION EN PYTHON

Ver el archivo bci_01.py

1.1.7 BÚSQUEDA BIDIRECCIONAL (BIDIRECTIONAL SEARCH)

Se basa en la búsqueda primero en anchura, y realiza dos búsquedas simultaneas, una desde el estado inicial y otra desde el estado final, en busca de un estado intermedio común. Una de las búsquedas puede ser en profundidad iterativa. Cuando se conoce la solución óptima y las acciones son reversibles y se busca el camino más corto. Es un tipo de algoritmo divide y vencerás, y es aplicable siempre y cuando se pueda conocer el estado final. El siguiente grafo permite comprender con mayor precisión este algoritmo.



En este algoritmo el espacio de estados es más pequeño

MEDIDAS DE RENDIMIENTO:

Este algoritmo:

- Es completo, es decir si existe solución, la encuentra.
- No es óptimo, porque no se puede garantizar que la solución encontrada sea la mejor.
- Tiene la siguiente complejidad:
    - Tiempo, exponencial $O(b^{d/2})$.
    - Espacio, exponencial $O(b^{d/2})$.

Considerando:

2^{32} = 4.294.967.256

2^{16} = 65.536

2^8 = 256

Este algoritmo en cuestión de tiempo es el mejor.

El pseudocódigo de este algoritmo es el siguiente:

```
función BÚSQUEDA-PRIMERO-ANCHURA(problema)
devuelve solución o fallo
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  si problema.ES-OBJETIVO(nodo-raíz.ESTADO) entonces
    devolver nodo-raíz
  frontera ← CREAR-FIFO()
  frontera.AGREGAR(nodo-raíz)
  explorada ← CREAR-CONJUNTO()
  repetir
     si frontera.ESTÁ-VACÍA() entonces devolver fallo
     nodo ← frontera.POP()
     explorada.AGREGAR(nodo)
     por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
       hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
       si hijo.ESTADO no está en explorada y
          hijo.ESTADO no está en frontera.ESTADOS() entonces
            si problema.ES-OBJETIVO(hijo.ESTADO) entonces
              devolver hijo
            frontera.AGREGAR(hijo)
```

VENTAJAS:

- Mucha mejor complejidad en tiempo que el resto de los algoritmos

DESVENTAJAS:

- Complejidad espacial exponencial.
- No garantiza que la solución sea optima.
- Requiere que se conozca la solución
- Requiere de acciones reversibles

IMPLEMENTACION EN PYTHON

Ver el archivo bbd_01.py

1.2 RESUMEN BUSQUEDA NO INFORMADA

Se resumen todos los algoritmos revisados en la siguiente tabla:

Tipo de Búsqueda	Completo	Óptimo
Primero en Anchura	Sí	Sí
Coste Uniforme	Sí	Sí
Primero en Profundidad	No	No
Profundidad Limitada	No	No
Profundidad Iterativa	Sí	Sí
Bidireccional	Sí	No

Tipo de Búsqueda	Tiempo	Espacio
Primero en Anchura	$O(b^d)$	$O(b^d)$
Coste Uniforme	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$
Primero en Profundidad	$O(b^m)$	$O(b \cdot m)$
Profundidad Limitada	$O(b^p)$	$O(b \cdot p)$
Profundidad Iterativa	$O(b^d)$	$O(b \cdot d)$
Bidireccional	$O(b^{d/2})$	$O(b^{d/2})$

1.3