Chapter 1

Representación de problemas y búsqueda de soluciones

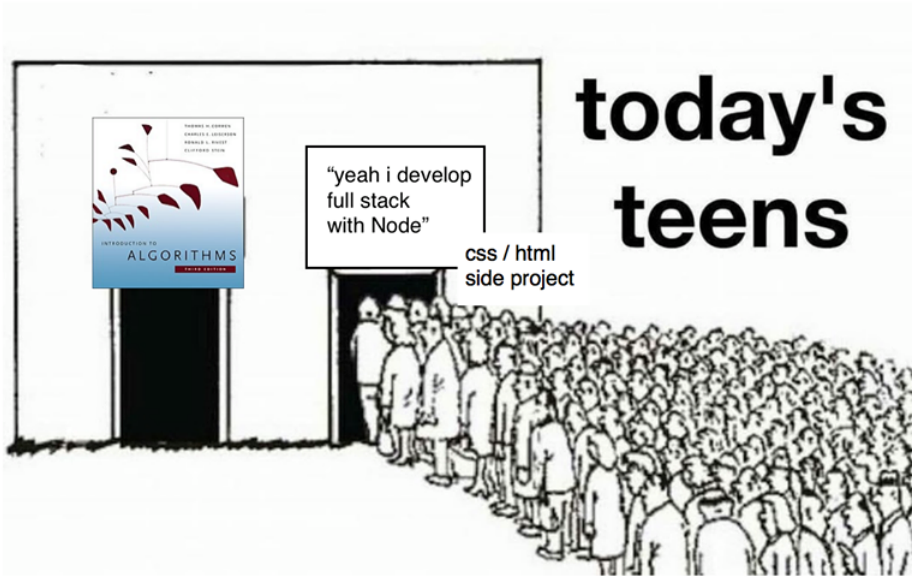

Figure 1.1: *full stack* (http://bit.ly/2kBKPAP)

Objetivo

El alumno explicará cómo actúan los agentes mediante la definición de metas y cómo consideran secuencias de acciones para alcanzarlas

En temas anteriores se presentaron algunas formas de representación, por ejemplo el cálculo proposicional. Expresiones bien formadas proveen un medio para describir objetos y relaciones en el dominio del problema; las reglas de inferencia permiten inferir conocimiento a partir de esas descripciones. Estas inferencias definen un espacio en el que se realiza la búsqueda de la solución. Este tema presenta la teoría de *búsqueda en espacio de estados* (http://bit.ly/14rKxMU).

1.1 Representación en espacio de estados

Para diseñar e implementar algoritmos de búsqueda, un programador debe ser capaz de analizar y predecir su comportamiento. Algunas preguntas que deben responderse son:

- ♢ ¿Está garantizado que el *resolvedor* de problemas encuentre la solución?
- ♢ ¿El programa siempre termina o puede caer en un ciclo infinito?
- ♢ Cuándo se encuentra una solución, ¿Se garantiza que es óptima?
- ♢ ¿Cuál es la complejidad del proceso de búsqueda en términos de tiempo y espacio?
- ♢ ¿Cómo se puede reducir la complejidad de la búsqueda?
- ♢ ¿Se puede modificar el diseño del intérprete para optimizar el uso del lenguaje de representación?

La teoría de búsqueda en espacio de estados es una herramienta poderosa para responder estas y otras preguntas. Al representar un problema como una gráfica de espacio de estados, se puede utilizar teoría de gráficas para analizar la estructura y complejidad tanto del problema como de los procedimientos que se emplean para resolverlos.

Una gráfica consiste de un conjunto de nodos y un conjuntos de arcos (aristas) que conectan a los nodos. En el modelo solución por espacio de estados, los nodos de una gráfica representan estados discretos en el proceso de solución del problema. Por ejemplo, las diferencias configuraciones del *puzzle* o los resultados de inferencias lógicas. Las aristas de las gráficas representan transiciones entre estados; estas transiciones corresponden con inferencias lógicas o movimientos válidos en el juego.

Por ejemplo, en sistemas expertos los estados describen en conocimiento de la instancia de un problema en cierta etapa del proceso de razonamiento. Las reglas del tipo $if \ldots then$ del sistema permiten generar nueva información: el acto de aplicar una regla se representa como un arco en la gráfica.

La teoría de gráficas es una excelente herramienta para razonar acerca de la estructura de objetos y relaciones; de hecho, la creación de esta teoría se debe a la necesidad de razonar. El matemático Leonhard Euler (http://bit.ly/he8age) inventó la teoría de gráficas para resolver el problema de "los puentes de Königsberg". La ciudad de Königsberg ocupaba ambos márgenes y dos islas de un río. Las islas y los márgenes del estaban conectadas por siete puentes, como se ve en la figura 1.2.

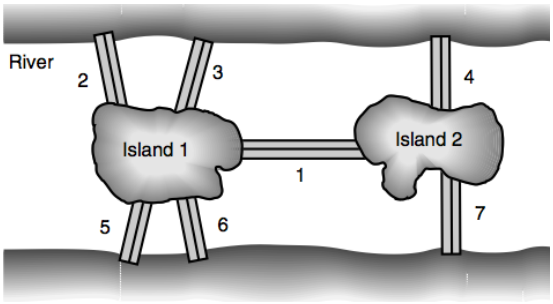

Figure 1.2: Ciudad de Königsberg

El problema de los puentes de Königsberg es determinar si existe una forma de visitar toda la ciudad cruzando cada puente exactamente una vez. Aún cuando los residentes habían fallado al buscar tal forma de visitar y dudaban que existiera, nadie había podido probar que es imposible. Euler creó una forma alternativa para representar el mapa, figura 1.3, concibiendo así a la teoría de gráficas. Los márgenes de río ($rb1$ y $rb2$) y las islas ($i1$ e $i2$) se describen como nodos en la gráfica; los puentes están representados como las aristas entre esos nodos ($b1, b2, \ldots, b7$). La representación como gráfica preserva la estructura esencial del sistema de puentes, mientras que ignora características extras tales como tamaño de los puentes, distancia y el orden de los puentes durante el recorrido.

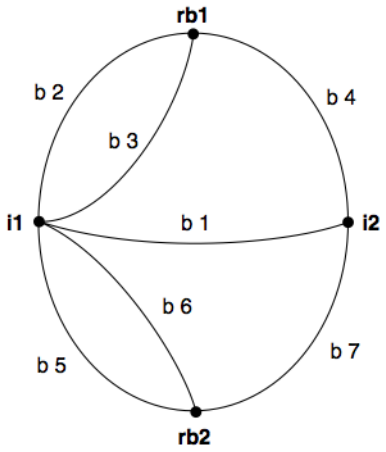

Figure 1.3: Gráfica del sistema de Königsberg

Para probar que un camino como tal es imposible de encontrar, Euler se enfocó en el *grado* de los nodos de la gráfica, observando que cada nodo tiene grado *par* o *impar*. Un nodo con grado par tiene un número par de arcos que lo conectan con un nodo vecino y un nodo impar tiene un número impar de aristas. Con excepción de los nodos inicial y final, el camino buscado debería dejar cada nodo en cuanto llega. Nodos de grado impar sólo pueden ser usados como inicio o fin de este tipo de camino, de otro modo el viajero no podría dejar ese nodo sin usar una arista utilizada previamente.

Euler notó que a menos que la gráfica tenga exactamente cero o dos nodos de grado impar, un camino como este es imposible de encontrar. Si hay dos nodos de grado impar, el camino puede empezar en uno y terminar en el otro; si no hay nodos de grado impar, el camino puede iniciar y terminar en el mismo nodo. El camino es imposible de obtener en gráficas con cualquier otro número de nodos con gado impar, como en el caso de la ciudad de Königsberg. Este problemas actualmente se enuncia como encontrar un *camino euleriano* en una gráfica.

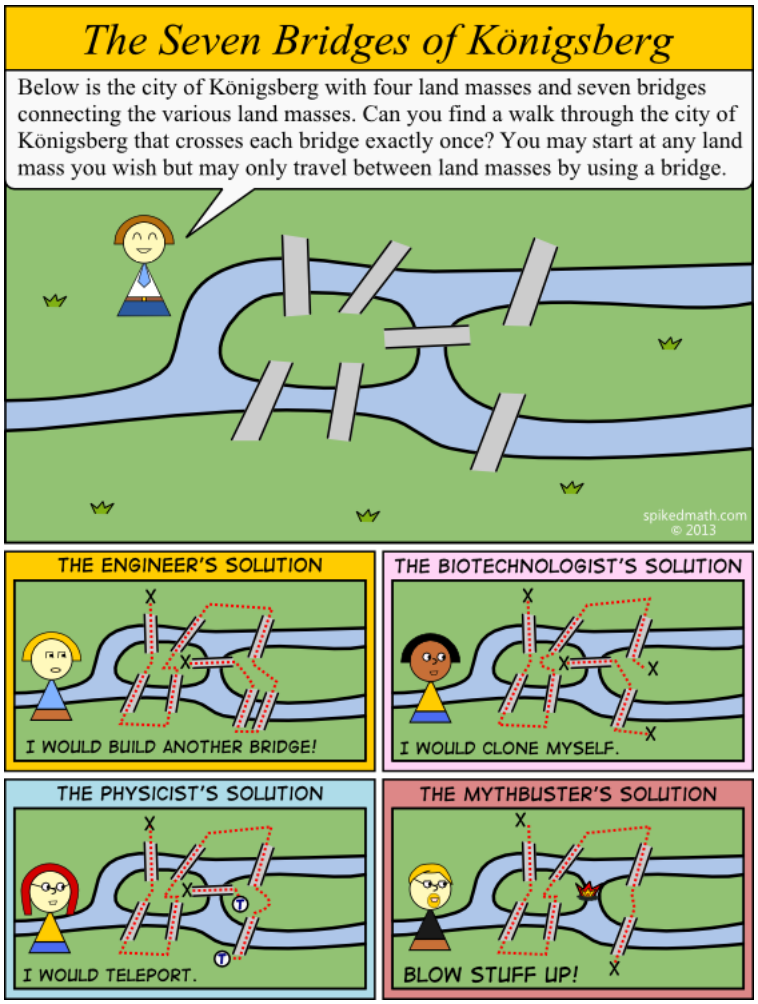

Figure 1.4: *Soluciones* alternativas para el sistema de Königsberg

1.2 Búsqueda de soluciones en espacio de estados

Al representar problemas como espacio de estados, los nodos de la gráfica corresponden con una solución parcial y los arcos corresponden a pasos en el proceso de solución. Uno o más estados iniciales corresponden a información conocida de una instancia del problema. La gráfica define también uno o más objetivos descritos (*Goal Description*, *GD*), que son también soluciones de la instancia del problema. La búsqueda en espacio de estados caracteriza el proceso de solución desde un estado inicial hasta el objetivo.

El camino de solución se busca a partir del estado inicial y recorriendo la gráfica, hasta que el objetivo descrito se satisface o esa ruta es abandonada. La generación de estados nuevos en el camino se realiza aplicando operaciones, tales como movimientos legales en un juego o reglas de inferencia lógica en un sistema experto. La tarea de un algoritmo de búsqueda es encontrar un *camino de solución* en el espacio de estados. Formalmente, la representación del espacio de

estados de un problema se define como:

Un *espacio de estados* (http://bit.ly/16l1jM3) se representa como una 4-tupla (N, A, S, GD), donde:

- ♢ N es el conjunto de estados de la gráfica. Corresponden a los *estados* en el proceso de solución.
- ♢ A es el conjunto de aristas de la gráfica. Corresponden a los *pasos* en el proceso de solución.
- ♢ S, un subconjunto no vacío de N, que contiene a los estados iniciales del problema.
- ♢ GD, un subconjunto no vacío de N, que contiene a los estados objetivo del problema. Los estados GD se describen como:

    1. Una propiedad medible en los *estados encontrados* en la búsqueda.
    2. Una propiedad medible en la *ruta desarrollada* en la búsqueda, por ejemplo, la suma o el promedio de los costos de las aristas en el camino.

Un *camino de solución* es un camino en la gráfica que va desde un nodo en S hasta un nodo en GD.

Una característica general de una gráfica, y también un problema que surge al diseñar el algoritmo de búsqueda en la gráfica, es que los estados algunas veces pueden alcanzarse por diferentes caminos. Esto hace importante saber elegir el mejor camino de acuerdo a las necesidades del problema particular. Además, múltiples caminos hacia un mismo nodo pueden llevar a ciclos en un camino, que impedirían alcanzar el objetivo.

Si el espacio de búsqueda es un *árbol*, el problema de los ciclos desaparece. Por tanto, es importante distinguir entre problemas en los que el espacio de búsqueda es un árbol y aquellos que pueden contener ciclos. Algoritmos de búsqueda en gráficas genéricas deben detectar y eliminar ciclos en caminos de solución potenciales; búsquedas en árboles pueden ganar eficiencia al eliminar este tipo de pruebas.

1.2.1 Ejemplos

El *gato* y el *puzzle* ejemplifican el espacio de estados para juegos simples; ambos muestran condiciones de término del tipo 1 en la definición de espacio de estados. El problema del agente viajero tiene una descripción del objetivo de tipo 2, el costo total del camino.

1.2.1.1 Juego de gato

La figura 1.5 muestra una parte de la representación del espacio de estados del juego gato (http://bit.ly/aM7iUm). El inicio es el tablero vacío y el objetivo o término es que el tablero tenga tres Xs en una fila, columna o diagonal (suponiendo que el objetivo es que gane X). El camino desde el inicio hasta el objetivo muestra la serie de movimientos en un juego ganador.

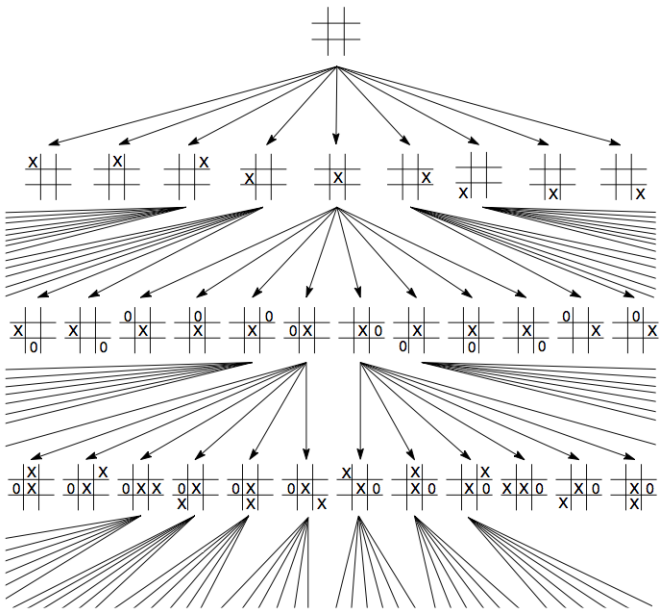

Figure 1.5: Gato

Los estados son todas las diferentes configuraciones de Xs y Os que pueden presentarse en el juego. Aunque existen 3^9 formas de acomodar $\{vacío, X, O\}$ en nueve espacios, muchos de ellos nunca ocurren en un juego real. Los arcos se generan por movimientos válidos del juego, alternando entre acomodar una X o una O en una posición libre. El espacio de estados es una gráfica (no un árbol) dado que algunos estados pueden alcanzarse en niveles avanzados del juego. Sin embargo, no existen ciclos en el espacio de estados: como las aristas de la gráfica son dirigidas no se permite deshacer movimientos. Es imposible *regresar* en la estructura una vez que se ha alcanzado un estado y no es necesario verificar ciclos. Una gráfica que tiene esta propiedad se llama gráfica acíclica dirigida (directed acyclic graph, DAG) y es muy común en búsqueda en espacio de estados y en modelos gráficos.

La representación en espacio de estados permite además determinar la complejidad de un problema. En el gato, hay nueve posibles primeros movimientos, cada uno con ocho posibles respuestas, seguidos de siete, etc. Por tanto en total hay $9 \times 8 \times 7 \times \ldots$ o 9! caminos distintos pueden generarse. Aún cuando es posible que una computadora busque exhaustivamente en este número de caminos $(362,880)$, muchos problemas importantes presentan también complejidad exponencial o factorial. El ajedrez tiene 10^{120} posibles caminos, el juego de damas tiene 10^{40}, algunos de los cuales podrían no presentarse en juegos reales. Estos espacios son díficiles e incluso imposibles de recorrer de manera exhaustiva. Algunas estrategias para buscar en espacios tan grandes se basan en *heurísticas* que reducen la complejidad de la búsqueda.

1.2.1.2 *Puzzle*

El juego de *puzzle* consiste de varias piezas numeradas colocadas dentro de una pequeña cuadrícula, dejando sólo un espacio vacío para poder mover las piezas. En la figura 1.6 se muestra un 8-*puzzle*,

una versión de cuadrícula de 3×3. El objetivo es obtener una configuración determinada de los números en la cuadrícula.

Aún cuando los movimientos *reales* corresponden a mover las fichas ("mover 7 arriba" o "mover 3 abajo"), es mucho más sencillo razonar en términos de mover el *espacio vacío*. Esto simplifica la definición de las reglas válidas de movimiento: hay ocho piezas y solo un espacio vacío. Hay que asegurarse de no mover el espacio vacío *fuera* de la cuadrícula; por tanto, no siempre son posibles los cuatro movimientos; por ejemplo si se encuentra en una esquina solo hay dos movimientos posibles.

Los movimientos legales son:

♢ mover el espacio vacío arriba ↑

♢ mover el espacio vacío a la derecha →

♢ mover el espacio vacío a la izquierda ←

♢ mover el espacio vacío abajo ↓

Al especificar un estado inicial y un estado objetivo para el *puzzle*, es posible presentar el espacio de estados del proceso de solución (figura 1.6). Los estados pueden representarse usando un arreglo de 3×3 y cuatro procedimientos, uno por cada movimiento válido, definen las aristas de la gráfica.

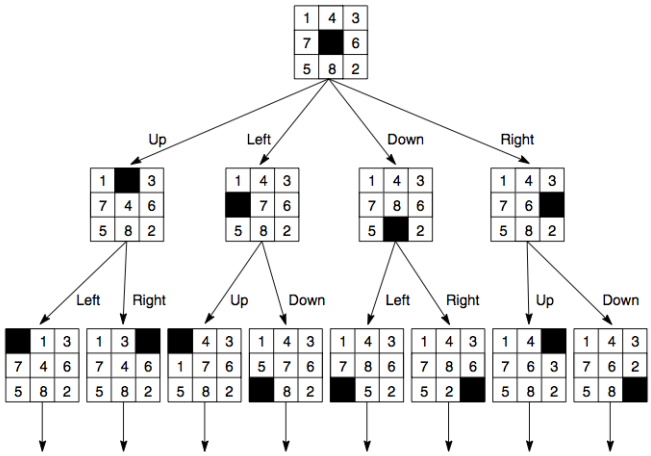

Figure 1.6: Puzzle

Similar al juego de gato, el espacio de estados del *puzzle* es una gráfica, pero a diferencia del gato, en el puzzle pueden presentarse ciclos. La descripción del objetivo del espacio de estados es una configuración particular de la cuadrícula. Cuando este estado se encuentra en la ruta, la búsqueda termina; el camino desde el inicio hasta el estado objetivo es la serie de movimientos deseados.

1.2.1.3 Problema del agente viajero (*travelling salesperson problem*)

Imaginemos una situación en la que un agente viajero (http://bit.ly/Jntmu) debe visitar varias ciudades en un mapa. El objetivo es encontrar la ruta más corta para que el agente visite cada una de las ciudades y regrese a la ciudad inicial. La figura 1.7 muestra un ejemplo sencillo de este problema. Los nodos representan las ciudades y cada arista tiene una etiqueta que indica el costo de utilizar esa arista. A modo de ejemplo, se puede suponer que el agente vive en la ciudad A y debe regresar a esa misma ciudad.

Nota: Este problema es muy importante e incluso tiene una película llamada *Travelling Salesman* (http://bit.ly/JvTCG4).

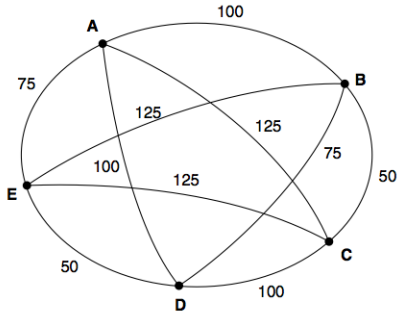

Figure 1.7: Problema del agente viajero

La ruta $[A, D, C, B, E, A]$, con costo asociado de 450 es un posible circuito para el agente. Sin embargo, la descripción del objetivo requiere que el circuito tenga costo mínimo. Esta es una descripción del objetivo de tipo 2 en la definición de búsqueda en el espacio de estados.

La figura 1.8 muestra una posible forma de generar y comparar posibles caminos solución. Comenzando en el nodo A, los posibles estados siguientes se agregan hasta que todas las ciudades son incluidas y se regresa al nodo A.

Como sugiere la figura 1.8, la complejidad de búsqueda exhaustiva en el problema del agente viajero es $(N-1)!$, donde N es el número de ciudades en la gráfica. Para 9 ciudades, es posible probar exhaustivamente todos los caminos, pero para instancias de tamaños mayores, por ejemplo 50, la simple búsqueda exhaustiva no es viable en tiempo *razonable*. El costo de complejidad computacional para $N!$ crece tan rápidamente que se considera *intratable*.

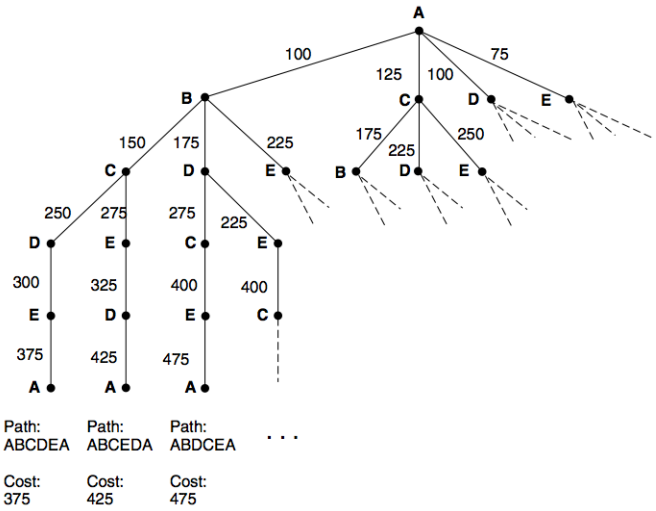

Figure 1.8: Búsqueda en el problema del agente viajero, cada arista se marca con el peso total actual del camino

Por tanto, se han desarrollado múltiples técnicas para reducir la complejidad de la búsqueda. Una de ellas se conoce como *ramificación y poda* (*branch and bound* B&B http://bit.ly/WZQFaZ). Este algoritmo genera un camino a la vez y almacena el mejor hasta el momento. Este valor se usa como un límite para futuros caminos candidatos. Mientras se construyen nuevas rutas, ciudad por ciudad, el algoritmo revisa cada camino parcial; si el algoritmo determina que la mejor extensión posible del camino actual tendrá mayor costo, lo elimina junto con todas sus posibles extensiones. Esto reduce considerablemente el número de caminos a revisar, pero aún es exponencial: 1.26^N.

Otra estrategia para controlar la búsqueda es construir rutas de acuerdo a la regla "*ve a la ciudad no visitada con menor costo*". El camino usando vecino más cercano en la gráfica se muestra en la figura 1.9 es $[A, E, D, B, C, A]$ con un costo de 550.

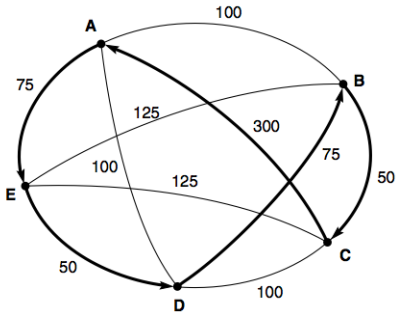

Figure 1.9: Ejemplo del problema del agente viajero con el camino generado por el *vecino más cercano* remarcado. Es importante notar que la ruta $[A, E, D, B, C, A]$ con costo de 550 no es el camino más corto. El alto costo de la arista (C, A) hace que la heurística falle

Este método es extremadamente eficiente, dado que solamente revisa un camino. La heurística del vecino más cercano, algunas veces llamada *ávida* (*greedy*), es falible: existen gráficas para las que no entrega el camino más corto global; pero es muy útil cuando la búsqueda exhaustiva resulta impráctica.

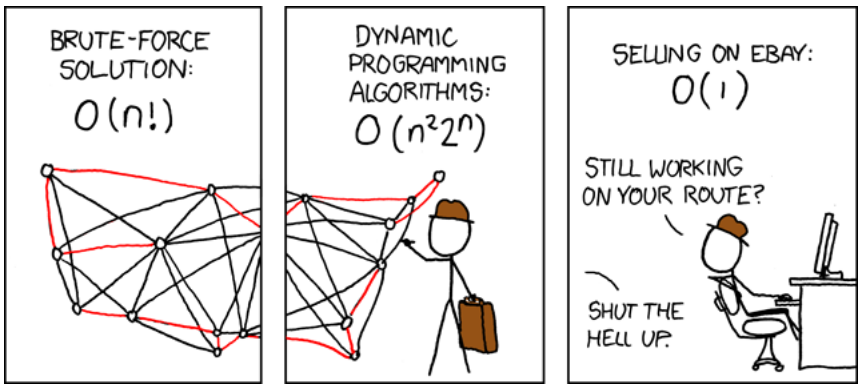

Figure 1.10: *Solución* al problema del agente viajero en *e-bay*

1.2.2 Métodos de búsqueda ciega

1.2.2.1 Búsqueda dirigida por datos y dirigida por objetivos

Un espacio de estados puede recorrerse en dos direcciones: desde los datos de una instancia del problema hasta el objetivo ó desde un objetivo hacia los datos.

En la *búsqueda dirigida por datos*, que corresponde con el razonamiento progresivo, el resolvedor inicia con datos conocidos del problema y un conjunto de movimientos legales o reglas para recorrer estados. La búsqueda se ejecuta aplicando reglas a los hechos para producir nuevos hechos, que se convierten en hechos para producir a su vez nuevos hechos. El proceso continúa hasta que se genere una ruta que satisfaga la condición objetivo.

Un enfoque alternativo: tomar un objetivo a resolver; revisar que reglas o movimientos legales pueden utilizarse para generar ese estado objetivo y determinar las condiciones que se deben cumplir para usar dichos movimientos. Estas condiciones se convierten en nuevos objetivos (o subobjetivos) para la búsqueda. La búsqueda hacia atrás continua hasta encontrar los hechos que responden al objetivo. Este enfoque se llama *búsqueda dirigida por objetivos* y corresponde con el razonamiento regresivo.

Además de especificar la dirección, un algoritmo de búsqueda debe determinar el orden en el que se examinarán los estados en la gráfica. Existen dos formas comúnes de hacer esto: búsqueda en profundidad (*depth-first*, dfs) y en amplitud (*breadth-first*, bfs).

1.2.2.2 Búsqueda en profundidad

En este tipo de búsqueda cuando un estado se ha examinado, se recorren todos sus hijos y los descendientes de estos antes de pasar a sus nodos hermanos. La búsqueda en profundida

va *descendiendo* en el espacio de estados siempre que sea posible; solo cuando no existen más descendientes del estado, se consideran los hermanos. Este método de búsqueda también suele llamarse *backtrack*.

La figura 1.11 presenta un ejemplo de búsqueda en profundidad para un 8-*puzzle* con profundidad acotada a 5.

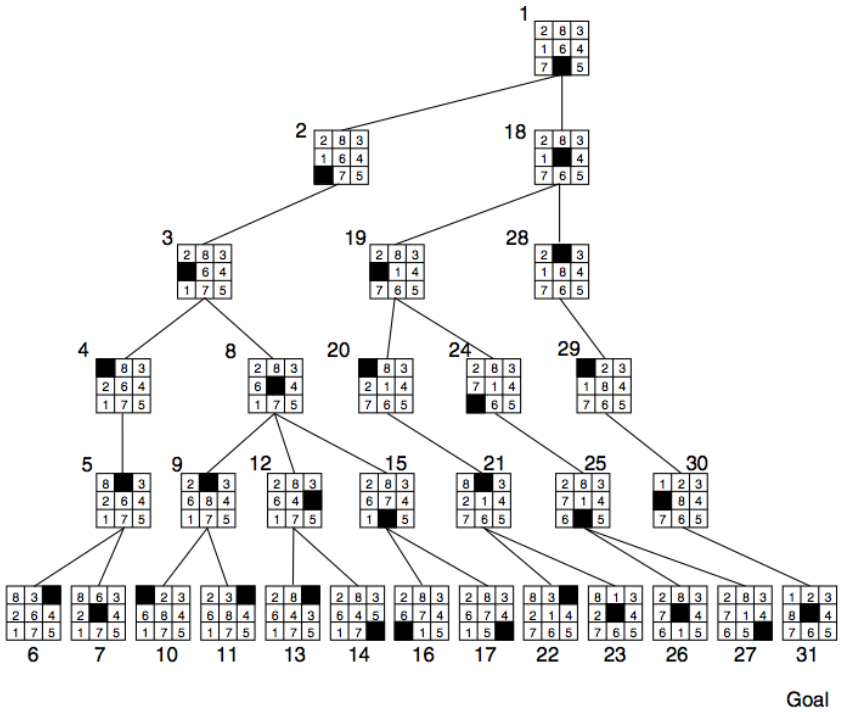

Figure 1.11: Búsqueda en profundidad para un 8-puzzle (profundidad acotada a 5)

1.2.2.3 Búsqueda en amplitud

Contrario a la búsqueda en profundidad, la búsqueda en amplitud explora el espacio de estados *nivel por nivel*. Solo cuando se hay visitado todos los nodos de un nivel dado, el algoritmo se mueve hacia un nivel más profundo.

La figura 1.12 presenta un ejemplo de búsqueda en amplitud para un 8-*puzzle*.

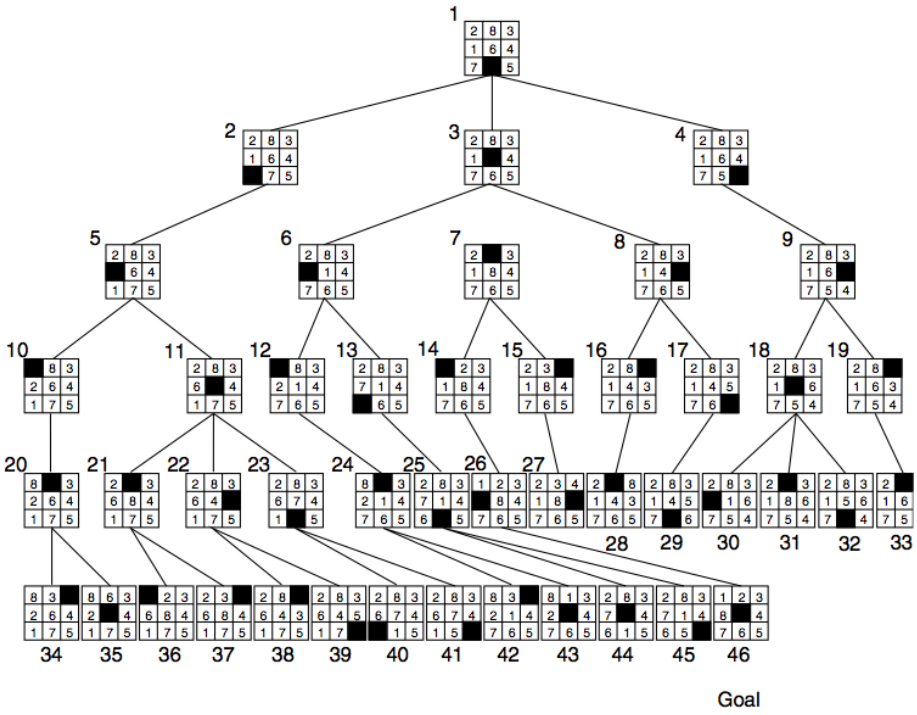

Figure 1.12: Búsqueda en amplitud para un 8 puzzle

Tarea: _____

1. Desarrolla un algoritmo para realizar: (1) búsqueda en profundidad y (2) búsqueda en amplitud. Debes indicar los ADT necesarios para recorrer la gráfica.

2. Muestra el resultado de recorrer la gráfica siguiente usando ambos enfoques:

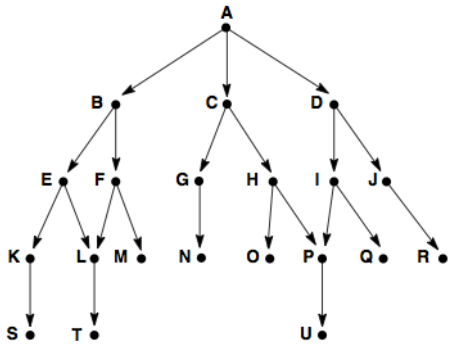

Figure 1.13: Gráfica para la tarea

_____ *

1.2.3 Métodos de búsqueda basados en *conocimiento*

George Pólya (http://bit.ly/XMDJm2) define heurística como "el estudio de métodos y reglas de descubrimiento e invención". Proviene de la raíz griega *eurisco* que significa "descubrir". En la búsqueda en espacio de estados, las heurísticas se formalizan como *guías* o reglas para elegir las ramas que más probablemente lleven a una solución aceptable del problema.

Los programas que resuelven problemas de de IA utilizan heurística principalmente por dos razones:

1. Un problema puede no tener solución exacta, debido a ambigüedades inherentes en el planteamiento del problema o en los datos disponibles. Por ejemplo, los médicos usan heurísticas para determinar el diagnóstico más plausible y así, formular un tratamiento adecuado; la misma idea se aplica a sistemas que realizan diagnósticos. Otro ejemplo de problemas inexactos es la visión; generalmente se presentan ambigüedades en las escenas (ilusiones ópticas, etc.) y los sistemas de visión utilizan heurísticas para elegir la mejor de todas las interpretaciones posibles.

2. Un problema puede tener solución exacta, pero el costo computacional de buscarlo lo hace prohibitivo. En muchos problemas (p.e. ajedrez) el espacio de estados es combinatoriamente explosivo, es decir, el número de estados posibles crece de forma exponencial o factorial. En ese caso tecnicas de búsqueda por *fuerza bruta* (exhaustiva) como búsqueda simple en profundidad o en amplitud pueden no entregar una solución en tiempo *razonablemente práctico*. Usando heurísticas, se puede guiar la búsqueda hacia las rutas más "prometedoras" en el espacio de estados. Eliminando estados "poco prometedores" y sus descendientes del espacio de búsqueda se puede encontrar una solución aceptable en tiempo aceptable.

Desafortunadamente, las heurísticas también son falibles: una heurística es solo una *suposición informada* (*informed guess*) de la próxima acción a tomar para resolver el problema; generalmente están basadas en experiencia o intuición. Como una heurística utiliza información limitada, como conocimiento de la situación presente o descripciones de estados disponibles actualmente, no hay garantía de predecir el comportamiento exacto del espacio de estados durante la búsqueda; por tanto, una heurística puede llevar a una solución subóptima o no encontrarla. Garey (http://bit.ly/ZJ8w5S) y Johnson (http://bit.ly/YIS4Dj) demostraron que esta limitación es inherente a las búsquedas heurísticas: no puede ser eliminada por una "mejor" heurística o algoritmos de búsqueda mejorados (http://bit.ly/YISjhu).

Las heurísticas y los algoritmos que las implementan han sido un tema muy importante de la IA desde hace mucho tiempo. La teoría de juegos y los demostradores de teoremas son dos de las aplicaciones más antiguas en IA: ambas requieren heurísticas para *podar* el espacio de posibles soluciones: en muchas situaciones el uso de búsquedas con heurísticas es la única respuesta práctica.

Los sistemas expertos han reafirmado la importancia del uso de heurísticas como un componente esencial en la solución de problemas. Por ejemplo, cuando un humano *experto* resuelve cierto problema examina la información disponible y toma una decisión; las "reglas de oro" que el humano experto usa para solucionar problemas son, casi siempre, de naturaleza heurística. Este tipo de heurísticas son extraídas y formalizadas por los diseñadores de sistemas expertos.

1.2.4 Métodos de búsqueda con adversarios

En situaciones en las que se tienen movimientos de dos o más agentes, son típicas de búsqueda con adversarios: cada agente busca obtener la mayor ganancia posible. Por esto es importante tomar en cuenta la forma en la que el adversario selecciona sus acciones.

Por ejemplo, en la figura 1.14 se observa una parte de la representación del espacio de estados del juego gato. Es importante notar que en ninguna situación el mismo jugador elige una acción en dos niveles consecutivos. El algoritmo más utilizado en este tipo de escenarios es el llamado *minimax* (http://bit.ly/md25h0).

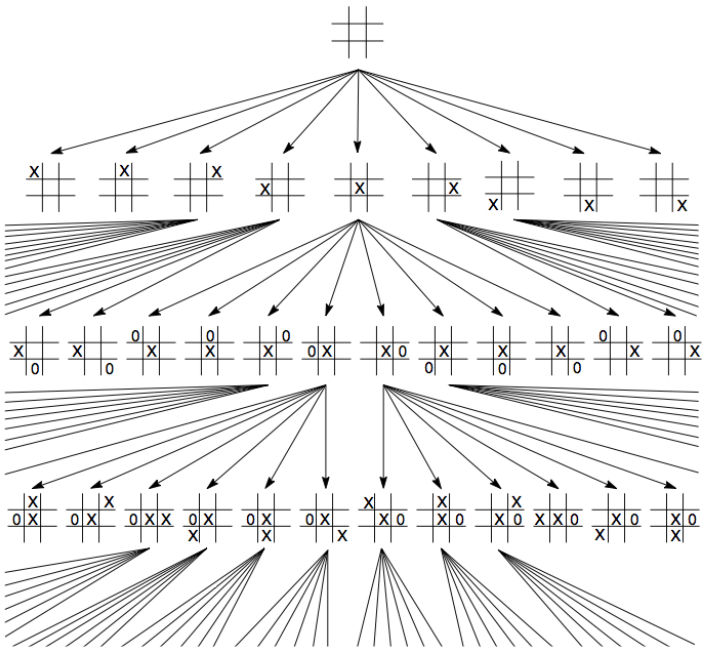

Figure 1.14: Gato

1.3 Representación reducida de problemas y búsqueda de soluciones

Recordando el juego de gato, la combinatoria para la búsqueda exhaustiva es muy alta, pero soluble: cada uno de los 9 movimientos iniciales tiene 8 posibles repuestas, etc. Un análisis sencillo muestra que para la búsqueda exhaustiva el número total de estados es 9!.

La *reducción por simetría* reduce el espacio de búsqueda: muchas de las configuraciones son equivalentes para las operaciones sobre el tablero de juego. Por tanto, no son 9 movimientos iniciales, hay solo 3: en una esquina, en el centro de un lado y en el centro de la cuadrícula. Usando simetría en el segundo nivel reduce aún más el número de rutas posibles del espacio. Las simetrías en un espacio de estados se describen matemáticamente como invariantes, cuando existen, generalmente reducen considerablemente el espacio de búsqueda.

La figura 1.15 muestra el espacio de búsqueda reducido por simetría para el juego de gato.

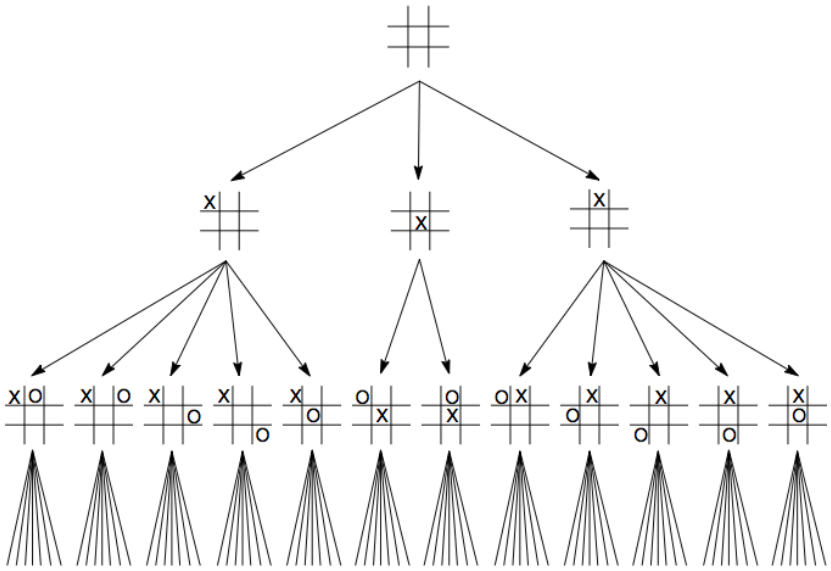

Figure 1.15: Tres niveles del espacio de estados del juego gato reducido por simetría

Además, aplicando una heurística es posible *casi* eliminar la búsqueda completa: elegir un estado en el que X tiene la mayor oportunidad de ganar. La figura 1.16 muestra la aplicación de esta heurística, las *medidas* de los primeros se muestran encerrados en un círculo. En el caso de tener varios estados con la misma medida de triunfo potencial, se puede elegir de forma arbitraria (p.e., el primero o al azar). El algoritmo entonces selecciona el estado marcado con el mayor número y se mueve a esa posición; en este ejemplo X en el centro de la cuadrícula. Es importante notar que no se eliminan solamente las otras dos alternativas: también sus descendientes; con esto se podan $^2/_3$ del espacio de estados con este movimiento.

Nota: Como el agente que juega X no sabe la forma en la que razona el agente O debe tomar en cuenta ambas opciones, aún cuando el estado "O en la esquina" tendría evaluación mayor.

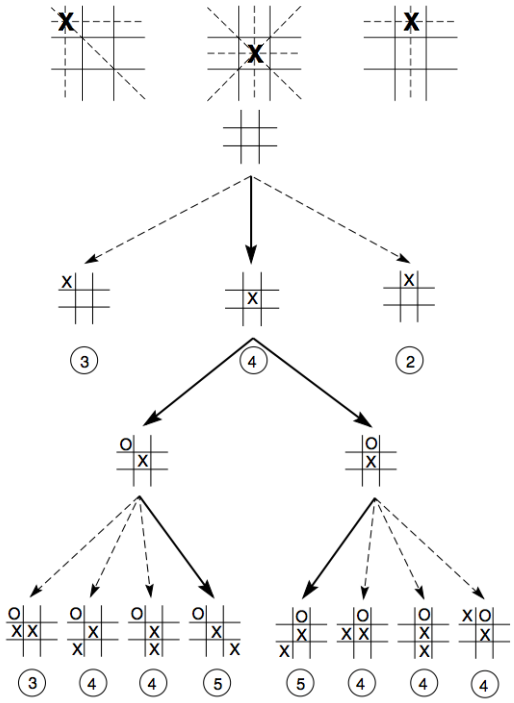

Figure 1.16: Espacio de estados del juego gato reducido heurísticamente

Después de la primera acción, el oponente elige alguna de las dos alternativas; cuando es nuevamente su turno, el algoritmo aplica la misma heurística al estado obtenido para seleccionar el mejor de los movimientos posibles (figura 1.16). Conforme avanza la búsqueda, en cada movimiento se evalúan solamente los hijos de un solo nodo: no se necesita la búsqueda exhaustiva. Aún cuando resulta difícil obtener un cálculo exacto del tamaño del espacio de búsqueda, es evidente que se obtiene una mejora considerable (de varios órdenes de magnitud) respecto al espacio inicial de 9!.

Tarea: _____

♢ ¿Es posible reducir por simetría el espacio de estados para el *puzzle*? ¿De alguna otra forma?

- Argumentar la respuesta.

_____ *

1.4 Solución de problemas mediante satisfacción de restricciones

Una de las formas más sencillas de implementar heurísticas, es un procedimiento llamado *hill-climbing* (http://bit.ly/oK56Rm). Las estrategias *hill-climbing* expanden el estado actual y evalúan a sus hijos: el "mejor" hijo es elegido para una nueva expansión; ni sus hermanos ni su padre son conservados. Como no se almacena un historial, el algoritmo no puede recuperarse

de fallos. La heurística "elegir un estado con la mayor oportunidad de ganar" mostrada en el juego de gato es un ejemplo de una estrategia *hill-climbing*.

El problema principal de las estrategias *hill-climbing* es su tendencia a *quedarse atascadas* en máximos locales: si llega a un estado que tiene mejor evaluación que sus hijos, el algoritmo falla. Se puede mejorar el rendimiento, pero dependiendo de la forma del espacio de búsqueda completo, puede no llegar al mejor global. Un ejemplo de máximo local ocurre en el *puzzle*: es común que para mover una pieza particular a su destino, otras piezas que ya estén en su posición final deban moverse. Esto es necesario para resolver el *puzzle* completo, pero temporalmente empeora el estado de la cuadrícula. Como "mejor" no implica necesariamente "el mejor" en sentido absoluto, métodos de búsqueda sin forma de regresar (*backtracking*) o algún otro mecanismo de recuperación no podrán distiguir entre máximos locales y globales.

1.4.1 Programación dinámica

La programación dinámica (*dynamic programming* DP http://bit.ly/eksCOe), también llamada *progresiva-regresiva* o, cuando se incluyen probabilidades, algoritmo de Viterbi (http://bit.ly/91RSRq).

Fue creada por Richard Bellman (http://bit.ly/14GKTzM) en 1956 para resolver el problema de memoria restringida en problemas compuestos de mútiples subproblemas interrelacionados. DP mantiene un registro de subproblemas ya recorridos y resueltos y reutiliza este registro para resolver el problema principal. Un ejemplo sencillo es la reutilización de subseries para obtener una determinada serie de Fibonacci.

Esta técnica de *cacheo* de subproblemas para reutilizarlos algunas veces se conoce como memorizado de soluciones de subobjetivos. El resultado es un algoritmo muy importante utilizado en muchas situaciones: correspondencia de cadenas, revisión de sintaxis y muchas otras aplicaciones de procesamiento del lenguaje. Ejemplos de algoritmos que utilizan DP: http://bit.ly/14GMd5v.

Ejemplo: Fibonacci con memoria.

```
1  mem_dict = {0:1, 1:1}
2  def mem_fib(n):
3    if n not in mem_dict:
4      mem_dict[n] = mem_fib(n-1) + mem_fib(n-2)
5    return mem_dict[n]
```

Figure 1.17: Fibonacci con memoria

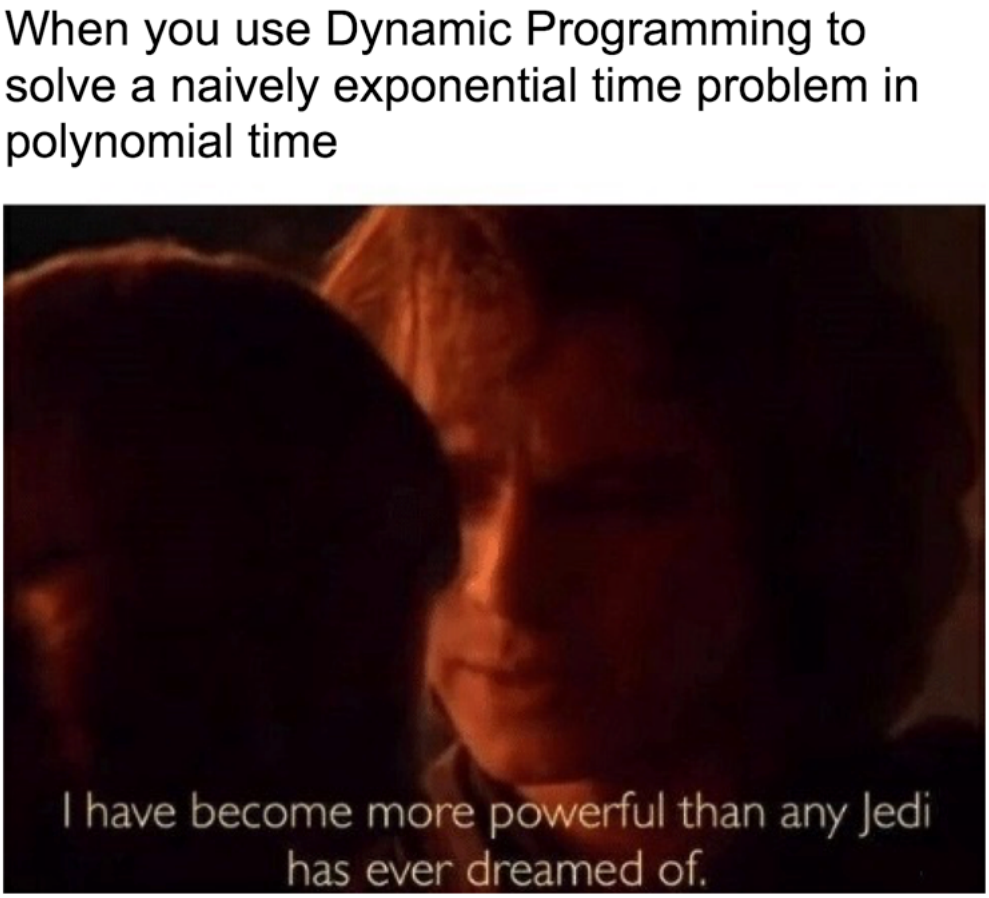

Figure 1.18: *Empoderamiento* (http://bit.ly/2kBKPAP)

1.4.1.1 Cálculo de la ruta más corta en una gráfica con pesos

El algoritmo de Dijkstra Creado por el científico holandés Edsger Dijkstra (http://bit.ly/QwxLSC) en 1956 es un algoritmo de búsqueda en gráficas que resuelve el problema de la ruta más corta para una gráfica con pesos no negativos en sus aristas y produce un árbol como solución. Este algoritmo se utliza comúnmente como subrutina en algoritmos de enrutamiento o en GPS.

La complejidad de este algoritmo es $O\left(|E| + |V|^2\right)$; si se utiliza una *cola de prioridad* para obtener al siguiente nodo a revisar, la complejidad baja a $O\left(|E| + |V|\, lg\, |V|\right)$. Se puede ver en ejecución en https://bit.ly/1NcsLoq.

Algoritmo 1.1 Algoritmo de Dijkstra para encontrar la ruta más corta

Llamamos **nodo inicial** a aquel en el que se inicia la búsqueda, **nodo destino** a aquel al que se desea llegar y **distancia del nodo** a la distancia desde el nodo inicial hasta el nodo actual.

1. Asignar a cada nodo una distancia *tentativa*: 0 para el nodo inicial, ∞ para todos los nodos restantes
2. Establecer al nodo inicial como nodo actual y crear un conjunto de nodos no visitados, llamado *conjunto no visitado* que contiene a todos los nodos excepto al actual
3. Para el nodo actual, considerar a todos sus vecinos no visitados y recalcular sus distancias tentativas:
   - ♢ Si la distancia del nodo actual sumada a la distancia desde el nodo actual hasta algún vecino es menor que la distancia tentativa actual de ese vecino, se debe sobreescribir la distancia con la suma obtenida. El nodo vecino aún no debe marcarse como *visitado* y se conserva dentro del conjunto no visitado.
4. Cuando se termina de revisar a todos los vecinos del nodo actual, se marca como *visitado* y se elimina del conjunto no visitado: un nodo visitado no se revisará nuevamente.
5. Si el nodo destino se ha marcado como visitado, el algoritmo ha terminado. Para obtener la ruta, se realiza la etapa regresiva: a partir del nodo destino, recorrer la gráfica tomando el estado de peso mínimo. Si se desea obtener la ruta más corta desde el nodo inicial a todos los nodos de de la gráfica, se continúa la ejecución hasta vaciar al conjunto no visitado.
6. Seleccionar el nodo no visitado con menor distancia tentativa y marcarlo como el nuevo nodo actual, regresar al paso 3.

Ejemplo: Obtener la ruta más corta desde el nodo 1 hacia los demás nodos en la gráfica de la figura 1.19.

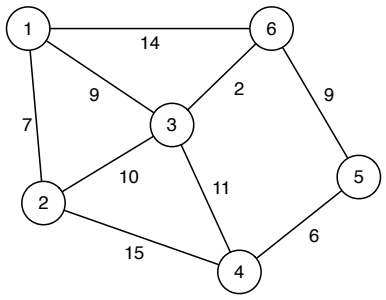

Figure 1.19: Gráfica del ejemplo

El algoritmo de Bellman-Ford-Moore Es un algoritmo que determina la ruta más corta desde un nodo hacia todos los demás en una gráfica dirigida con pesos asociados a las aristas. Fue desarrollado por Richard Bellman (http://bit.ly/1MHCrSe) y Lester Ford, Jr. (http://bit.ly/1CocDtd) en 1958 y 1956 respectivamente; Edward F. Moore (http://bit.ly/1BLAu3W) obtuvo el mismo algoritmo en 1957 y en ocasiones se conoce también como el *algoritmo Bellman–Ford*. Es más lento que el algoritmo de Dijkstra pero es más versátil, incluso puede trabajar con gráficas que incluyen pesos negativos en algunas aristas.

La complejidad de este algoritmo es $O(|E| \cdot |V|)$.

Algoritmo 1.2 Algoritmo de Bellman-Ford-Moore para encontrar la ruta más corta

---

Llamamos **nodo inicial** a aquel en el que se inicia la búsqueda, **distancia del nodo** a la distancia desde el nodo inicial hasta el nodo actual y **predecesor** al nodo desde el cual se llega al nodo actual con el camino de menor peso.

1. Asignar a cada nodo una distancia un nodo predecesor *tentativo*s: (0 para el nodo inicial, ∞ para todos los nodos restantes); (predecesor *nulo* para todos los nodos)
2. Repetir $|V| - 1$ veces
   (a) Para cada arista (u, v) con peso w:
   ♢ Si la distancia del nodo actual u sumada al peso w de la arista que llega a v es menor que la distancia tentativa al nodo v, sobreescribir la distancia a v con la suma mencionada y guardar a u como predecesor de v.
3. Verificar que no existan ciclos de pesos negativos:
   (a) Para cada arista (u, v) con peso w:
   ♢ Si la distancia del nodo actual u sumada al peso w de la arista que llega a v es menor que la distancia tentativa al nodo v, devolver un mensaje de error indicando que existe un ciclo de peso negativo.

Ejercicio: Obtener la ruta más corta desde el nodo 1 hacia los demás nodos en la gráfica de la figura 1.19.

Tarea:

Obtener la ruta más corta desde el nodo a hacia los demás nodos utilizando ambos algoritmos para la gráfica de la figura 1.20.

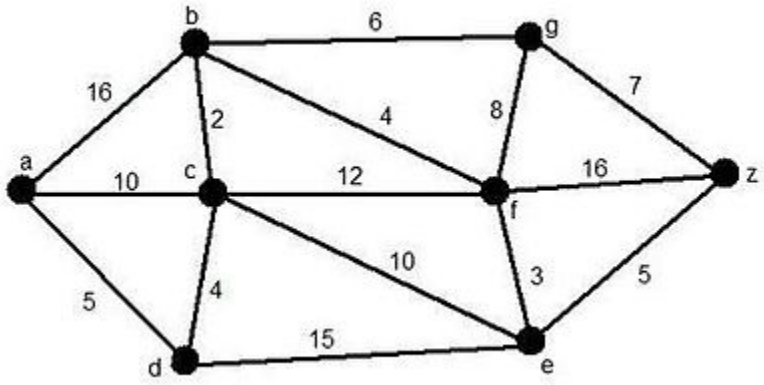

Figure 1.20: Tarea

---

*

El algoritmo A^* Es un algoritmo de búsqueda en gráficas dirigidas con pesos asociados a las aristas. Descrito originalmente en 1968 por Peter Hart, Nils Nilsson y Bertram Raphael del *Stanford Research Institute* (actualmente *SRI International*). Es otra exensión del algoritmo de Dijkstra. A^* tiene mejor rendimiento siempre que la heurística utilizada sea buena.

Este algoritmo utiliza una función para evaluar el costo en los nodos: $f(n) = g(n) + h(n)$; donde $g(n)$ es el costo hasta el nodo actual y $h(n)$ es el costo *estimado* (heurística) para llegar del nodo actual al nodo buscado. Se dice que A^* combina búsqueda en amplitud con $g(n)$ y en profundidad con $h(n)$.

La complejidad de este algoritmo depende de la heurística elegida: en el peor de los casos es exponencial, pero si la heurística es buena, se vuelve lineal con respecto al número de nodos. Al igual que el algoritmo de Dijkstra, si se utiliza una cola de prioridad para la lista de nodos no evaluados, el rendimiento mejora.

A^* garantiza que devolverá el camino óptimo entre el nodo inicial y el objetivo si existe dicho camino.

Para ver el algoritmo en ejecución: https://qiao.github.io/PathFinding.js/visual/

Algoritmo 1.3 Algoritmo de búsqueda A^*

Llamamos **inicio** a aquel nodo en el que se inicia la búsqueda, **objetivo** aquel al que se desea llegar y **predecesor** al nodo desde el cual se llega al nodo actual con el camino de menor peso.

1. Inicialización:

   ♢ Conjunto de nodos evaluados iniclamente vacío. Conjunto de nodos *descubiertos* pero que aún no han sido evaluados inicialmente sólo contiene a **inicio** (*conjunto_abierto*)

   ♢ Predecesor nulo para todos los nodos. Asignar a cada nodo una distancia *tentativa* ($g(n)$) para llegar a ese nodo (0 para el nodo inicial, ∞ para todos los nodos restantes)

   ♢ Asignar a cada nodo una distancia *tentativa* ($f(n)$) para llegar al nodo destino con ayuda de la heurística ($estimacion\_heuristica(inicio, objetivo)$ para el nodo inicial y ∞ para todos los nodos restantes)

2. Mientras *conjunto_abierto* tenga elementos

   ♢ *actual* ← nodo del conjunto abierto con la menor distancia tentativa heurística

   ♢ Si $actual = objetivo$
   Regresar $reconstruir\_camino(predecesor, actual)$

   ♢ Sacar *actual* del *conjunto_abierto*

   ♢ Agregar *actual* al *conjunto_cerrado*

   ♢ Para cada *vecino* de *actual*:

   - Si $vecino \in conjunto\_cerrado$
     ignorar y pasar al siguiente vecino
   - Calcular distancia desde el inicio hasta el vecino:
     $g\_tentativa \leftarrow g(actual) + peso\_arista(actual, vecino)$
   - Si $vecino \notin conjunto\_abierto \Rightarrow$ nuevo nodo *descubierto*, agregar *vecino* al *conjunto_abierto*
   - Otro, si $g\_tentativa \geq g(vecino) \Rightarrow$ no hay mejora en el camino, ignorar y pasar al vecino siguiente
   - Este camino es mejor, guardarlo:
     $predecesor(vecino) \leftarrow actual$
     $g(vecino) \leftarrow g\_tentativa$
     $f(vecino) \leftarrow g(vecino) + estimacion\_heuristica(vecino, objetivo)$

3. Regresar *error*

♢ función $reconstruir\_camino(predecesor, actual)$

- $camino \leftarrow [actual]$
- Mientras $actual \in claves(predecesor)$
  $actual \leftarrow predecesor(actual)$
  $camino.agregar(actual)$
- Regresar *camino*

Ejemplo-tarea: Obtener la ruta más corta desde el nodo 0 al nodo 6 en la gráfica de la figura 1.21; la heurística utilizada es la distancia en línea recta desde cada nodo hasta el objetivo.

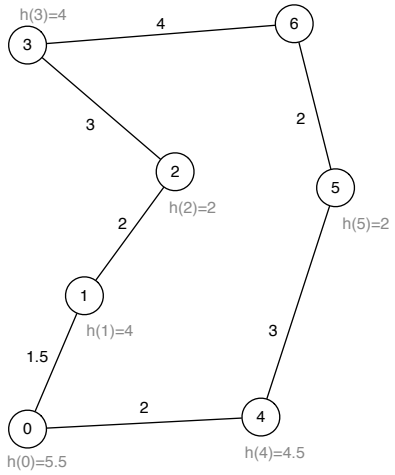

Figure 1.21: Gráfica del ejemplo

Programa:

♢ Implementar el algoritmo de Dijkstra para obtener la ruta más corta.

♢ Elegir entre el algoritmo de Bellman-Ford-Moore y $A*$ para implementarlo.

El lenguaje es de libre elección, pero se deben implementar los programas de forma que quede bien claro cada uno de los pasos del mismo; NO está permitido utilizar alguna biblioteca o programa desarrollado por terceros.