Sistemas Inteligentes

Jaime Hernández Delgado
@nickhernd

Curso 2024-2025

Índice general

Capítulo 1

Estrategias de Búsqueda

1.1. Introducción a las Estrategias de Búsqueda en IA

La **búsqueda** es un componente fundamental en la Inteligencia Artificial (IA). Se utiliza para resolver una amplia gama de problemas, desde la planificación de rutas hasta la toma de decisiones complejas. Las estrategias de búsqueda nos permiten explorar eficientemente un espacio de estados para encontrar soluciones óptimas o satisfactorias a problemas dados.

Algunas aplicaciones comunes de las estrategias de búsqueda incluyen:

- Problemas de búsqueda en rutas (por ejemplo, navegación GPS)
- Planificación de líneas aéreas
- Optimización de rutas en redes de computadores
- Resolución de problemas turísticos (planificación de itinerarios)
- El problema del viajante de comercio
- Distribución VLSI (diseño de circuitos integrados)
- Navegación de robots
- Secuenciación para el ensamblaje automático
- Diseño de proteínas
- Búsqueda en Internet

1.2. Especificación de Problemas

Para resolver un problema mediante búsqueda, es necesario definir varios elementos clave:

1. **Espacio de estados**: Es el conjunto de todas las posibles configuraciones o situaciones del problema.
2. **Estado inicial**: Es el punto de partida de la búsqueda.
3. **Estado(s) meta**: Son las configuraciones que consideramos soluciones al problema.
4. **Reglas de transformación**: Son las acciones que nos permiten pasar de un estado a otro.

El proceso de búsqueda se puede visualizar como la exploración de un árbol (árbol de búsqueda) o, en general, un grafo. Es importante notar que la búsqueda completa del espacio de estados suele ser inviable para problemas reales debido a su tamaño exponencial. Por ejemplo, en un espacio de tamaño 10^{20}, la búsqueda exhaustiva sería impracticable.

1.2.1. Sistemas de Producción (SP)

Los **Sistemas de Producción** son una forma de formalizar los problemas de búsqueda de estados. Propuestos por Post en 1943, un Sistema de Producción se define como una terna (BH, RP, EC):

- **BH (Base de Hechos)**: Conjunto de representaciones de uno o más estados por los que atraviesa el problema. Constituye la estructura de datos global.
- **RP (Reglas de Producción)**: Conjunto de operadores para la transformación de los estados del problema. Cada regla tiene dos partes: precondiciones y postcondiciones.
- **EC (Estrategia de Control)**: Determina el conjunto de reglas aplicables mediante un proceso de pattern-matching y resuelve conflictos entre varias reglas a aplicar mediante el filtrado.

El algoritmo básico de un Sistema de Producción se puede expresar como:

```
Algoritmo SP(BH0, RP, EC)
BH = BH0
repetir
R = Aplicables(BH)
Ri = Seleccionar(R)
BH = Ri(BH)
hasta CondicionesTerminación(BH)
```

1.3. Estrategias de Búsqueda Básicas

Las estrategias de búsqueda básicas se pueden clasificar en dos categorías principales:

1.3.1. Estrategias Irrevocables

Las estrategias irrevocables, también conocidas como de **descenso por gradiente**, tienen las siguientes características:

- No permiten la vuelta atrás.
- Mantienen una frontera unitaria.
- Requieren suficiente conocimiento local.
- Asumen que las equivocaciones solo alargan la búsqueda.
- Buscan optimalidad global a partir de la local.

En estas estrategias, se utiliza una función de evaluación $f()$ que proporciona un mínimo (o máximo) en el estado final. La selección de la regla a aplicar se basa en la optimización local de esta función.

1.3.2. Estrategias Tentativas

Las estrategias tentativas pueden ser de dos tipos:

- **Multi-camino**: Mantienen múltiples estados de vuelta atrás.
- **Mono-camino**: Mantienen un único estado de vuelta atrás.

Estas estrategias se dividen a su vez en:

No Informadas

- **Búsqueda en profundidad**: Explora el nodo más profundo del árbol de búsqueda.
- **Búsqueda en anchura**: Explora todos los nodos de un nivel antes de pasar al siguiente.
- **Coste uniforme**: Selecciona el nodo con menor coste acumulado desde el nodo inicial.

Informadas

Las estrategias informadas utilizan una función heurística $h(n)$ que estima el coste desde un nodo hasta el objetivo. La función de evaluación general es:

$$f(n) = g(n) + h(n)$$

donde $g(n)$ es el coste real desde el nodo inicial hasta n, y $h(n)$ es la estimación heurística desde n hasta el objetivo.

1.4. Búsqueda Heurística

La búsqueda heurística es fundamental en la IA para abordar problemas complejos. Se basa en el uso de funciones heurísticas para guiar la búsqueda hacia soluciones prometedoras.

1.4.1. Conceptos Básicos

- **Completitud**: Un algoritmo es completo si encuentra una solución cuando esta existe.
- **Admisibilidad**: Un algoritmo es admisible si encuentra la solución óptima.
- **Dominación**: Un algoritmo A1 domina a A2 si todo nodo expandido por A1 también es expandido por A2.
- **Optimalidad**: Un algoritmo es óptimo si es dominante sobre todos los algoritmos del conjunto.

1.4.2. Algoritmo A*

El algoritmo A* es uno de los más populares en búsqueda heurística. Su función de evaluación es:

$$f^*(n) = g^*(n) + h^*(n)$$

donde:

- $g^*(n)$ es el coste real del camino más corto desde el nodo inicial s hasta n.
- $h^*(n)$ es el coste real del camino más corto desde n hasta cualquier estado solución.
- $f^*(n)$ es el coste del camino más corto desde el nodo inicial hasta un nodo solución pasando por n.

Una función heurística $h(n)$ se considera admisible si:

$$h(n) \leq h^*(n) \quad \forall n$$

Un algoritmo A que utiliza una función heurística admisible se denomina algoritmo A*.

1.5. Implementación en Python

A continuación, se presenta un ejemplo simple de implementación del algoritmo de búsqueda en anchura en Python:

```
from collections import deque

def bfs(graph, start, goal):
queue = deque([[start]])
visited = set([start])

while queue:
path = queue.popleft()
node = path[-1]

if node == goal:
return path

for neighbor in graph[node]:
```

```
if neighbor not in visited:
visited.add(neighbor)
new_path = list(path)
new_path.append(neighbor)
queue.append(new_path)

return None

# Ejemplo de uso
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

print(bfs(graph, 'A', 'F'))  # Output: ['A', 'C', 'F
    ']
```

1.6. Comparación de Estrategias

La siguiente tabla compara las principales características de las estrategias de búsqueda discutidas:

Estrategia	**Completa**	**Óptima**	**Complejidad Espacial**
Profundidad	No	No	$O(bm)$
Anchura	Sí	Sí	$O(b^d)$
Coste Uniforme	Sí	Sí	$O(b^{C^*/\varepsilon})$
A*	Sí	Sí	$O(b^d)$

Cuadro 1.1: Comparación de estrategias de búsqueda

Donde b es el factor de ramificación, d es la profundidad de la solución más superficial, m es la profundidad máxima del espacio de estado y C* es el coste de la solución óptima.

1.7. Técnicas Complementarias

Existen varias técnicas complementarias para mejorar la eficiencia de la búsqueda:

- **Uso de movimientos de libro**: Utilizar conocimiento precomputado para ciertas fases del problema (por ejemplo, aperturas en ajedrez).
- **Espera del reposo**: Evitar el efecto horizonte continuando la búsqueda si se detectan cambios drásticos en la evaluación.
- **Técnica de bajada progresiva**: Recorrer nodos por niveles y devolver la mejor solución encontrada hasta el momento si se alcanza el límite de tiempo.
- **Poda heurística**: Reducir el factor de ramificación desarrollando solo los mejores movimientos de cada nivel.
- **Continuación heurística**: Seleccionar un subconjunto de nodos terminales para desarrollar búsquedas más profundas.

1.8. Conclusiones

Las estrategias de búsqueda son una herramienta fundamental en la IA para resolver problemas complejos. La elección de la estrategia adecuada depende de las características específicas del problema y de los recursos disponibles. El uso de heurísticas y técnicas complementarias puede mejorar significativamente la eficiencia de la búsqueda, permitiendo abordar problemas de gran escala en tiempos razonables.

Capítulo 2

Búsqueda en Juegos y Satisfacción de Restricciones

2.1. Introducción a la Búsqueda en Juegos

La búsqueda en juegos es un área fundamental de la Inteligencia Artificial (IA) que se centra en desarrollar algoritmos capaces de tomar decisiones óptimas en entornos competitivos. Los juegos proporcionan un marco ideal para estudiar la toma de decisiones en situaciones de incertidumbre y con información imperfecta.

2.1.1. Historia y Evolución

La historia de la IA en juegos está marcada por hitos significativos:

- **1950**: Alan Turing propone el "Test de Turingz desarrolla el primer programa de ajedrez.
- **1956**: Arthur Samuel crea un programa de damas capaz de aprender.
- **1997**: Deep Blue de IBM derrota al campeón mundial de ajedrez, Garry Kasparov.
- **2016**: AlphaGo de DeepMind vence al campeón mundial de Go, Lee Sedol.
- **2017**: AlphaZero domina ajedrez, go y shogi, aprendiendo solo mediante autojuego.

2.1.2. Complejidad Computacional en Juegos

Los juegos de estrategia presentan una complejidad computacional enorme:

- **Ajedrez**: Aproximadamente 10^{120} posiciones posibles.
- **Go**: Alrededor de 10^{170} posiciones posibles.
- Factor de ramificación medio en ajedrez: 35 movimientos por turno.
- Profundidad media de una partida de ajedrez: 80 movimientos.

Esta complejidad hace imposible una búsqueda exhaustiva, requiriendo estrategias más sofisticadas.

2.2. Representación de Juegos como Problemas de Búsqueda

Para abordar los juegos como problemas de búsqueda, se definen los siguientes elementos:

- **Estado**: Representación completa del tablero en un momento dado.
- **Jugadores**: Típicamente MAX (que busca maximizar la puntuación) y MIN (que busca minimizarla).
- **Acciones**: Movimientos legales desde un estado dado.
- **Modelo de Transición**: Cómo cambia el estado tras realizar una acción.
- **Test Terminal**: Determina si el juego ha terminado.
- **Función de Utilidad**: Asigna un valor numérico a los estados terminales.

2.3. Algoritmo MiniMax

El algoritmo MiniMax es la base de muchas estrategias de búsqueda en juegos. Su funcionamiento se basa en la alternancia entre los jugadores MAX y MIN.

2.3.1. Principio de Funcionamiento

1. Generar el árbol de juego hasta una profundidad predefinida.
2. Evaluar los nodos hoja usando una función de evaluación.
3. Propagar los valores hacia arriba en el árbol:
   - Nodos MAX: seleccionar el máximo valor de los hijos.
   - Nodos MIN: seleccionar el mínimo valor de los hijos.
4. La raíz elige la acción que lleva al mayor valor.

2.3.2. Implementación en Pseudocódigo

```
funcion minimax(nodo, profundidad, esMaximizador)
si profundidad = 0 o nodo es terminal entonces
retornar valor_heuristico(nodo)
si esMaximizador entonces
valor = -infinito
para cada hijo de nodo hacer
valor = max(valor, minimax(hijo, profundidad-1, falso))
retornar valor
sino
valor = +infinito
para cada hijo de nodo hacer
valor = min(valor, minimax(hijo, profundidad-1, verdadero))
retornar valor
```

2.4. Poda Alfa-Beta

La poda alfa-beta es una mejora significativa del algoritmo MiniMax, reduciendo el número de nodos evaluados sin afectar el resultado final.

2.4.1. Principio de Funcionamiento

- α: El mejor valor encontrado para MAX en el camino actual.
- β: El mejor valor encontrado para MIN en el camino actual.
- Se poda una rama cuando se determina que no puede influir en la decisión final.

2.4.2. Implementación en Pseudocódigo

```
función alfa_beta(nodo, profundidad, α, β,
   esMaximizador)
si profundidad = 0 o nodo es terminal entonces
retornar valor_heurístico(nodo)
si esMaximizador entonces
valor = -∞
para cada hijo de nodo hacer
valor = max(valor, alfa_beta(hijo, profundidad
   -1, α, β, falso))
α = max(α, valor)
si β ≤α entonces
romper  // Poda β
retornar valor
sino
valor = +∞
para cada hijo de nodo hacer
valor = min(valor, alfa_beta(hijo, profundidad
   -1, α, β, verdadero))
β = min(β, valor)
si β ≤α entonces
romper  // Poda α
retornar valor
```

2.5. Técnicas Avanzadas en Búsqueda de Juegos

2.5.1. Tablas de Transposición

Almacenan estados ya evaluados para evitar recálculos en posiciones repetidas.

2.5.2. Búsqueda con Profundización Iterativa

Incrementa gradualmente la profundidad de búsqueda, combinando las ventajas de la búsqueda en anchura y en profundidad.

2.5.3. Movimientos de Apertura

Utilizan bibliotecas de jugadas iniciales predefinidas para mejorar el rendimiento en las primeras etapas del juego.

2.5.4. Función de Evaluación

Diseñar una buena función de evaluación es crucial. Debe capturar aspectos importantes del juego como:

- Material (piezas en el tablero)
- Posición (control del centro, estructuras de peones)
- Movilidad (número de movimientos disponibles)
- Seguridad del rey

2.6. Juegos con Información Imperfecta

Algunos juegos, como el póker o el bridge, involucran información oculta o aleatoriedad. Estos requieren técnicas adicionales:

- **Árboles de Expectiminimax**: Incorporan nodos de azar para manejar eventos aleatorios.
- **Muestreo de Monte Carlo**: Simula múltiples partidas para estimar el valor de las acciones.

2.7. Satisfacción de Restricciones (CSP)

Los problemas de satisfacción de restricciones son una clase importante de problemas en IA, que a menudo se resuelven mediante técnicas de búsqueda.

2.7.1. Definición Formal

Un CSP se define por:

- Un conjunto de variables $X = \{X_1, ..., X_n\}$
- Dominios para cada variable $D_i = \{v_1, ..., v_k\}$
- Un conjunto de restricciones C que limitan los valores que las variables pueden tomar simultáneamente

2.7.2. Técnicas de Resolución

Backtracking

Asigna valores a las variables secuencialmente, retrocediendo cuando encuentra una inconsistencia.

Forward Checking

Mantiene una lista de valores legales para variables no asignadas, actualizándola con cada asignación.

Propagación de Restricciones

Utiliza las restricciones para reducir los dominios de las variables antes y durante la búsqueda.

Ordenación de Variables y Valores

Estrategias heurísticas para decidir qué variable asignar a continuación y en qué orden probar los valores.

2.8. Aplicaciones Prácticas

Las técnicas de búsqueda en juegos y CSP tienen numerosas aplicaciones:

- Planificación y scheduling
- Diseño de circuitos
- Asignación de recursos
- Configuración de productos
- Diagnóstico médico
- Procesamiento del lenguaje natural

2.9. Tendencias Actuales y Futuras

- Integración de técnicas de aprendizaje profundo con búsqueda tradicional
- Desarrollo de algoritmos más eficientes para juegos de gran escala

- Aplicación de técnicas de juegos a problemas del mundo real
- Exploración de juegos con múltiples agentes y objetivos complejos

2.10. Conclusiones

La búsqueda en juegos y la satisfacción de restricciones son áreas fundamentales de la IA que han impulsado avances significativos en algoritmos y heurísticas. Aunque los juegos como el ajedrez y el go han sido resueltos.[en] cierta medida por la IA, quedan desafíos importantes en juegos más complejos y en la aplicación de estas técnicas a problemas del mundo real. El futuro promete una integración más profunda de estas técnicas clásicas con métodos de aprendizaje automático modernos.

Capítulo 3

Búsqueda en problemas de Satisfacción de Restricciones

3.1. Búsqueda en problemas de Satisfacción de Restricciones

3.1.1. Definición y Conceptos Básicos

Los Problemas de Satisfacción de Restricciones (CSPs) son una clase fundamental de problemas en inteligencia artificial. Un CSP se define formalmente como una tripleta (V, D, ρ), donde:

- $V = \{V_1, V_2, ..., V_n\}$ es un conjunto de variables.
- $D = \{D_1, D_2, ..., D_n\}$ es un conjunto de dominios finitos para cada variable.
- $\rho = \{\rho_1, \rho_2, ..., \rho_m\}$ es un conjunto de restricciones definidas sobre subconjuntos de variables.

El objetivo de un CSP es encontrar una asignación de valores a todas las variables que satisfaga todas las restricciones.

Ejemplo: Coloreado de Mapas

Un ejemplo clásico de CSP es el problema de colorear un mapa utilizando un número limitado de colores, de manera que ningún par de regiones adyacentes tenga el mismo color.

- Variables: Las regiones del mapa.

- Dominios: Los colores disponibles (por ejemplo, {rojo, verde, azul}).
- Restricciones: Regiones adyacentes deben tener colores diferentes.

3.1.2. Representación de CSPs

Los CSPs se pueden representar gráficamente como redes de restricciones, donde:

- Los nodos representan variables.
- Las aristas representan las restricciones entre variables.

Esta representación permite visualizar la estructura del problema y aplicar técnicas de teoría de grafos para su resolución.

3.1.3. Métodos de Resolución

Generación y Prueba

Este es el método más simple pero menos eficiente.
Algoritmo:

1. Generar una asignación completa de valores a todas las variables.
2. Comprobar si la asignación satisface todas las restricciones.
3. Si satisface, devolver la solución. Si no, volver al paso 1.

Complejidad: $O(d^n)$, donde d es el tamaño del dominio más grande y n es el número de variables.

Backtracking

El algoritmo de backtracking mejora la eficiencia construyendo una solución de forma incremental.
Algoritmo:

1. Seleccionar una variable no asignada.
2. Asignar un valor del dominio de la variable.
3. Si la asignación es consistente, recurrir para la siguiente variable.
4. Si se detecta una inconsistencia o no quedan valores en el dominio, retroceder (backtrack) a la variable anterior.

Mejoras:

- Backjumping: Retroceder directamente a la variable causante del conflicto.
- Backmarking: Mantener un registro de los conflictos para evitar chequeos redundantes.

Forward Checking (FC)

El Forward Checking es una mejora sobre el backtracking que realiza cierta propagación de restricciones.

Algoritmo:

1. Al asignar un valor a una variable, actualizar los dominios de las variables futuras.
2. Eliminar valores incompatibles con la asignación actual.
3. Si algún dominio queda vacío, retroceder inmediatamente.

Ventaja: Detecta fallos antes que el backtracking puro, reduciendo el espacio de búsqueda.

Propagación de Restricciones

Esta técnica transforma el problema en otro equivalente pero más sencillo.

Algoritmo AC-3 (Arc Consistency):

1. Inicializar una cola con todos los arcos del grafo de restricciones.
2. Mientras la cola no esté vacía:
   - Seleccionar y eliminar un arco (Xi, Xj) de la cola.
   - Revisar el arco, eliminando valores inconsistentes del dominio de Xi.
   - Si se modifica el dominio de Xi, añadir a la cola todos los arcos (Xk, Xi) donde Xk Xj.

Complejidad: $O(ed^3)$, donde e es el número de arcos y d el tamaño del dominio más grande.

Algoritmos Híbridos

Combinan técnicas de búsqueda con inferencia.
Ejemplo: Maintaining Arc Consistency (MAC)

- Combina backtracking con AC-3.
- Después de cada asignación, aplica AC-3 para propagar las restricciones.
- Más efectivo que FC en problemas con restricciones densas.

3.1.4. Heurísticas

Las heurísticas pueden mejorar significativamente la eficiencia de la búsqueda en CSPs.

Ordenación de Variables

- MRV (Minimum Remaining Values): Elegir la variable con el dominio más pequeño.
- Degree heuristic: Elegir la variable que está involucrada en el mayor número de restricciones.

Ordenación de Valores

- Least Constraining Value: Elegir el valor que deja más opciones para las variables futuras.

3.1.5. Aplicaciones Prácticas

Los CSPs tienen numerosas aplicaciones en el mundo real:

- Planificación y programación de horarios
- Asignación de recursos
- Diseño de circuitos
- Diagnóstico médico
- Configuración de productos

3.1.6. Conclusión

La elección del método de resolución para un CSP depende de las características específicas del problema, como su tamaño, la densidad de las restricciones y la estructura de la red. En la práctica, los algoritmos más eficientes suelen combinar búsqueda con propagación de restricciones y utilizan heurísticas inteligentes para guiar la búsqueda.

Capítulo 4

Introducción Sistemas Expertos