CAP 4053 Artificial Intelligence for Computer Games: Search Algorithms

CAP 4053 Artificial Intelligence for Computer Games - Search Algorithms

Introduction

Search algorithms are a fundamental part of artificial intelligence, and are used to find paths to the goal in a variety of problem domains. Search algorithms can be applied to computer games to find the best move or path for a player or AI agent. In this lecture, we will discuss the different types of search algorithms, their advantages and disadvantages, and how they can be applied to computer games.

Types of Search Algorithms

There are several different types of search algorithms, each with its own advantages and disadvantages. The most common types of search algorithms include:

- Breadth-First Search (BFS): BFS is an algorithm that explores all of the nodes at the same level before moving on to the next level. It is a complete and optimal search algorithm, meaning it will always find the shortest path to the goal.

- Depth-First Search (DFS): DFS is an algorithm that explores one branch of the tree as deeply as possible before backtracking and exploring another branch. It is an incomplete search algorithm, meaning it may not always find the shortest path to the goal.

- Iterative Deepening Search (IDS): IDS is a combination of BFS and DFS. It starts with a shallow search, then gradually increases the depth of the search until the goal is found. It is a complete and optimal search algorithm.

- A* Search: A* is a heuristic search algorithm that uses an estimate of the cost of the remaining path to guide the search. It is a complete and optimal search algorithm, and is often used in computer games.

Coding Examples

Breadth-First Search
Start of Code
```
```
```
def bfs(start, goal):
        # Create a queue for BFS
        queue = []
        # Mark the start node as visited and enqueue it
        visited = set()
        queue.append(start)
        visited.add(start)
        # Loop until we find the goal
        while queue:
```

```
                # Dequeue a node from the queue
                node = queue.pop(0)
                # Check if the node is the goal
                if node == goal:
                        return True
                # Generate the children of the node
                children = generate_children(node)
                # Add the children to the queue
                for child in children:
                        if child not in visited:
                                queue.append(child)
                                visited.add(child)
        # We have not found the goal
        return False
```
End of Code

Depth-First Search
Start of Code
```
def dfs(start, goal):
        # Create a stack for DFS
        stack = []
        # Mark the start node as visited and push it onto the stack
        visited = set()
        stack.append(start)
        visited.add(start)
        # Loop until we find the goal
        while stack:
                # Pop a node from the stack
                node = stack.pop()
                # Check if the node is the goal
                if node == goal:
                        return True
                # Generate the children of the node
                children = generate_children(node)
                # Add the children to the stack
                for child in children:
                        if child not in visited:
                                stack.append(child)
                                visited.add(child)
        # We have not found the goal
        return False
```

```
End of Code

### Iterative Deepening Search
Start of Code
```
```
def ids(start, goal):
	# Set the initial depth to 0
	depth = 0
	# Loop until we find the goal
	while True:
		# Run DFS with the current depth
		found = dfs_limited(start, goal, depth)
		# If the goal was found, return True
		if found:
			return True
		# Increment the depth
		depth += 1
	# We have not found the goal
	return False
def dfs_limited(start, goal, depth):
	# Create a stack for DFS
	stack = []
	# Mark the start node as visited and push it onto the stack
	visited = set()
	stack.append((start, 0))
	visited.add(start)
	# Loop until we find the goal
	while stack:
		# Pop a node from the stack
		node, cur_depth = stack.pop()
		# Check if the node is the goal
		if node == goal:
			return True
		# Generate the children of the node
		children = generate_children(node)
		# Add the children to the stack
		for child in children:
			if child not in visited and cur_depth < depth:
				stack.append((child, cur_depth + 1))
				visited.add(child)
	# We have not found the goal
	return False
```

```
End of Code
```

A* Search
Start of Code
```
```
def a_star(start, goal):
	# Create a priority queue for A*
	queue = PriorityQueue()
	# Mark the start node as visited and enqueue it
	visited = set()
	queue.put((0, start))
	visited.add(start)
	# Loop until we find the goal
	while queue:
		# Dequeue a node from the queue
		cost, node = queue.get()
		# Check if the node is the goal
		if node == goal:
			return True
		# Generate the children of the node
		children = generate_children(node)
		# Add the children to the queue
		for child in children:
			if child not in visited:
				cost = calculate_cost(child)
				queue.put((cost, child))
				visited.add(child)
	# We have not found the goal
	return False
```
End of Code

## Practice Multiple Choice Questions

Q1. Which of the following is a complete and optimal search algorithm?

A. Breadth-First Search

B. Depth-First Search

C. Iterative Deepening Search

D. A* Search

Answer: D. A* Search
```