**Report on Search Algorithm Implementations**

Group Members: Shane McKelvey, Julian Fletcher

Date: September 18, 2025

Programming Language: C++

Hardware: MacBook Pro, 36 GB Memory

AI Tools Used: Gemini 2.5 Pro

**1. Overview of Implementation**

The provided C++ code implements a solver for a variation of the vacuum world problem. The agent must navigate a grid and clean all dirty locations. The problem is modeled as a state-space search, where each state is defined by the agent's current location and the set of remaining dirty squares. The code implements three uninformed search algorithms to find a sequence of actions leading to a goal state.

* **Core Data Structures:** The implementation uses structs for Location, State, and Node. The frontier (Fringe) is a std::priority\_queue ordered by ascending path cost, with tie-breaking by row and then column.

**2. Algorithm Implementations**

**2.1. Uniform Cost Tree Search (UCTS)**

This algorithm finds a path to a goal state with the minimum possible cost.

* **Methodology:** UCTS explores the state space by always expanding the node with the lowest path cost from the root. It uses the Fringe (priority queue) to manage nodes, ensuring the lowest-cost node is always selected for expansion.
* **Implementation Details:** This is a **tree search**, meaning it does **not** maintain a closed\_set of visited states. Consequently, if the search space contains multiple paths to the same state, this algorithm may expand the same state multiple times. This can lead to significantly more expanded nodes compared to a graph search, especially in densely connected state spaces

**2.2. Uniform Cost Graph Search (UCGS)**

This algorithm is an optimization of UCTS for graphs.

* **Methodology:** Like UCTS, this algorithm expands the node with the lowest path cost.
* **Implementation Details:** The key difference is the use of a std::set<State> closed\_set. Before a node is expanded, its state is checked against this set. If the state has been seen before (i.e., is in the closed\_set), the node is discarded and not expanded. This prevents redundant exploration of already-visited states and avoids infinite loops in graphs with cycles, making it significantly more efficient in terms of nodes expanded.

**2.3. Iterative Deepening Tree Search (IDTS)**

This algorithm combines the space-efficiency of depth-first search with the completeness of breadth-first search.

* **Methodology:** IDTS performs a series of depth-limited searches (DLS) with a progressively increasing depth limit, starting from 0. The search is repeated from the root node in each iteration.
* **Implementation Details:** A recursive helper function, depth\_limited\_search, is used to explore paths up to the current depth\_limit. If a solution is found, the algorithm terminates. Otherwise, the depth limit is incremented, and the search restarts. Because the code defines varying action costs, the first solution found by IDTS is guaranteed to be optimal in terms of the **number of moves (depth)**, but **not** necessarily in terms of **total path cost**. As noted in the source code, tracking the first five expanded nodes is impractical, as nodes are repeatedly expanded in each iteration.

**3. Experimental Results**

**Instance 1**

* **Initial State:** [V(2,2), d(1,2), d(2,4), d(3,5)]
* **Grid Size:** 4x5

**a. Uniform Cost Graph Search**

* **First 5 Expanded Node States:**

1. [V(2,2), d(1,2), d(2,4), d(3,5)]
2. [V(3,2), d(1,2), d(2,4), d(3,5)]
3. [V(1,2), d(1,2), d(2,4), d(3,5)]
4. [V(2,3), d(1,2), d(2,4), d(3,5)]
5. [V(2,1), d(1,2), d(2,4), d(3,5)]

* **Performance Metrics:**
* Nodes Expanded: **89**
* Nodes Generated: **370**
* CPU Time: **0.0030 seconds**
* **Solution Details:**
* Solution Path: Up, Suck, Down, Right, Right, Suck, Down, Right, Suck
* Number of Moves: **9**
* Solution Cost: **6.7**

**b. Iterative Deepening Tree Search**

* **First 5 Expanded Node States:**

1. [V(2,2), d(1,2), d(2,4), d(3,5)]

2. [V(1,2), d(1,2), d(2,4), d(3,5)]

3. [V(2,2), d(1,2), d(2,4), d(3,5)]

4. [V(1,2), d(1,2), d(2,4), d(3,5)]

5. [V(2,2), d(1,2), d(2,4), d(3,5)]

* **Performance Metrics:**
* Nodes Expanded: **87,399**
* Nodes Generated: **380,350**
* CPU Time: **0.3652 seconds**
* **Solution Details:**
* Solution Path: Up, Suck, Down, Right, Right, Suck, Down, Right, Suck
* Number of Moves: **9**
* Solution Cost: **6.7**

**c. Uniform Cost Tree Search**

* **First 5 Expanded Node States:**

1. [V(2,2), d(1,2), d(2,4), d(3,5)]
2. [V(2,2), d(1,2), d(2,4), d(3,5)]
3. [V(3,2), d(1,2), d(2,4), d(3,5)]
4. [V(1,2), d(1,2), d(2,4), d(3,5)]
5. [V(2,3), d(1,2), d(2,4), d(3,5)]

* **Performance Metrics:**
* Nodes Expanded: **331,475**
* Nodes Generated: **1,458,174**
* CPU Time: **1.6578 seconds**
* **Solution Details:**
* Solution Path: Up, Suck, Right, Right, Down, Suck, Down, Right, Suck
* Solution Cost: **6.7**
* Number of Moves: **9**

**Instance 2**

* **Initial State:** [V(3,2), d(1,2), d(2,1), d(2,4), d(3,3)]
* **Grid Size:** 4x5

**a. Uniform Cost Graph Search**

* **First 5 Expanded Node States:**

1. [V(3,2), d(1,2), d(2,1), d(2,4), d(3,3)]
2. [V(4,2), d(1,2), d(2,1), d(2,4), d(3,3)]
3. [V(2,2), d(1,2), d(2,1), d(2,4), d(3,3)]
4. [V(3,3), d(1,2), d(2,1), d(2,4), d(3,3)]
5. [V(3,1), d(1,2), d(2,1), d(2,4), d(3,3)]

* **Performance Metrics:**
* Nodes Expanded: **279**
* Nodes Generated: **1,154**
* CPU Time: **0.0027 seconds**
* **Solution Details:**
* Solution Path: Right, Suck, Right, Up, Suck, Up, Left, Left, Suck, Down, Left, Suck
* Number of Moves: **12**
* Solution Cost: **9.5**

**b. Iterative Deepening Tree Search**

* **First 5 Expanded Node States:**

1. [V(3,2), d(1,2), d(2,1), d(2,4), d(3,3)]

2. [V(2,2), d(1,2), d(2,1), d(2,4), d(3,3)]

3. [V(1,2), d(1,2), d(2,1), d(2,4), d(3,3)]

4. [V(2,2), d(1,2), d(2,1), d(2,4), d(3,3)]

5. [V(1,2), d(1,2), d(2,1), d(2,4), d(3,3)]

* **Performance Metrics:**
* Nodes Expanded: **17,597,803**
* Nodes Generated: **76,550,724**
* CPU Time: **75.9529 seconds**
* **Solution Details:**
* Solution Path: Right, Suck, Up, Right, Suck, Up, Left, Left, Suck, Down, Left, Suck
* Number of Moves: **12**
* Solution Cost: **9.5**

**c. Uniform Cost Tree Search**

* **First 5 Expanded Node States:**

1. [V(3,2), d(1,2), d(2,1), d(2,4), d(3,3)]
2. [V(3,2), d(1,2), d(2,1), d(2,4), d(3,3)]
3. [V(4,2), d(1,2), d(2,1), d(2,4), d(3,3)]
4. [V(2,2), d(1,2), d(2,1), d(2,4), d(3,3)]
5. [V(3,3), d(1,2), d(2,1), d(2,4), d(3,3)]

* **Performance Metrics:**
* Nodes Expanded: **55,223,888**
* Nodes Generated: **239,762,623**
* CPU Time: **441.5566 seconds**
* **Solution Details:**
* Solution Path: Right, Suck, Up, Right, Suck, Up, Left, Left, Suck, Down, Left, Suck
* Number of Moves: **12**
* Solution Cost: **9.5**

**4. Analysis and Conclusion**

The results clearly demonstrate the theoretical trade-offs between the three search algorithms.

* **Uniform Cost Graph Search (UCGS)** was overwhelmingly the **best performer**. By using a closed\_set to avoid re-expanding states, it solved both problems with minimal time and memory usage (very few nodes expanded). It is the ideal choice for this problem.
* **Iterative Deepening Tree Search (IDTS)** successfully found the optimal solution for both instances but was significantly slower than UCGS. The massive number of expanded nodes is a direct result of restarting the search at each depth, which is computationally expensive. However, its memory usage is low (like DFS), preventing it from crashing on the more complex instance.
* **Uniform Cost Tree Search (UCTS)** was highly inefficient. The lack of a closed\_set caused it to expand millions of redundant nodes. This led to extreme runtimes.