Helwan University, Faculty of Engineering

Iterative Deepening DFS for Pacman

Presented to: Dr. Shahira, Eng. Abdelrahman, Eng. Nancy

Team Members:

Ziad Osama Mohamed El-Boshy — 222250166
Suhila Usama Ahmed — 222250175
Omar Ahmed Fouad Elsayed — 222250191
Ali Reda Salah Ali — 222250188
Mohamed Magdy Aly Ahmed — 222250203

December 17, 2025

Contents

1 Project Context

This repository is the UC Berkeley CS188 Pacman search project. Core logic lives in `pacman.py`, `game.py`, `util.py`, with search implementations in `search.py`, agents in `searchAgents.py`, and the CLI/game harness in `pacman.py`. Assets include layouts under `layouts/` and autograder fixtures in `test_cases/`.

2 Change Log and Rationale

All changes were made to support iterative deepening DFS (IDDFS) and standalone depth-limited DFS (DLDFS) with configurable depth limits and better usability.

2.1 search.py

- Added `time` import to timestamp runs for metrics.
- Implemented `depth_limited_dfs` to run DFS with a depth cap (or unbounded when `None`) while counting expansions. This isolates a single depth iteration for IDDFS.
- Implemented `depthLimitedSearch` as a CLI-facing wrapper that prints DLDFS metrics and returns only the path; aliased as `dldfs`.
- Implemented `iterativeDeepeningSearch` with optional `max_depth` (default 1000 or unbounded when `None`). It validates the argument, loops depth limits until a solution is found or the cap is reached, accumulates expansions, and prints timing plus node counts.
- Added shorthand alias `iddfs` for iterative deepening.

`depth_limited_dfs` (full code):

```
def depth_limited_dfs(problem: SearchProblem, limit: int):
    """
    Depth-limited DFS used by iterative deepening.

    Returns a tuple of (path, found, expanded) where expanded counts node
    expansions during this limited search.
    """
    from util import Stack

    # Treat None as unbounded for convenience when passed from CLI.
    depth_cap = float('inf') if limit is None else int(limit)

    stack = Stack()
    stack.push((problem.getStartState(), [], 0))
    visited = {}
    expanded = 0

```

```
    while not stack.isEmpty():
        state, path, depth = stack.pop()

        if problem.isGoalState(state):
            return path, True, expanded

        if depth >= depth_cap:
            continue

        if state in visited and visited[state] <= depth:
            continue
        visited[state] = depth

        successors = problem.getSuccessors(state)
        expanded += 1
        for successor, action, step_cost in successors:
            stack.push((successor, path + [action], depth + 1))

    return [], False, expanded
```

Step-by-step:

1. Normalize the limit: `None` becomes `inf`, else cast to `int`.
2. Initialize a LIFO stack with the start state, an empty action path, and depth 0.
3. Pop a node; if it is a goal, return its path, mark success, and report expansions so far.
4. If the current depth meets/exceeds the cap, skip expanding that node.
5. Avoid revisits at shallower or equal depths by tracking the best depth per state.
6. Expand successors, increment the expansion counter, and push each child with depth + 1 and an updated action list.
7. If the stack drains without a goal, report failure with the number of expanded nodes.

`depthLimitedSearch` **(full code):**

```
def depthLimitedSearch(problem: SearchProblem, max_depth: int = 1000):
    """
    Depth-limited DFS that mirrors CLI usage: returns only the action path and
    prints basic metrics. Uses 'max_depth' to align with SearchAgent's parsing.
    """
    try:
        depth_cap = int(max_depth) if max_depth is not None else None
    except (TypeError, ValueError):
        raise AttributeError('max_depth must be an integer or None')

    start_time = time.time()
    path, found, expanded = depth_limited_dfs(problem, depth_cap)
    elapsed = time.time() - start_time

```

```
    cap_label = depth_cap if depth_cap is not None else 'inf'
    if found:
        print(f"[DLDFS] solution within depth {cap_label} | expanded {expanded} nodes | {elapsed:.4f}s")
        return path
    print(f"[DLDFS] no solution within depth {cap_label} | expanded {expanded} nodes | {elapsed:.4f}s")
    return []
```

Step-by-step:

1. Parse `max_depth` (or allow `None` for unbounded); raise on invalid input.
2. Run `depth_limited_dfs` with that cap to get path, success flag, and expansion count.
3. Time the run and print DLDFS metrics (cap, expansions, elapsed).
4. Return the path if found; otherwise return an empty list after logging failure.

`iterativeDeepeningSearch` **(full code):**

```
def iterativeDeepeningSearch(problem: SearchProblem, max_depth: int = 1000):
    """
    Iterative deepening DFS: increase the depth limit until a path is found or
    max_depth is reached. Prints basic metrics for manual comparison.
    """
    try:
        max_depth = int(max_depth) if max_depth is not None else None
    except (TypeError, ValueError):
        raise AttributeError('max_depth must be an integer or None')

    total_expanded = 0
    start_time = time.time()

    depth = 1
    while True:
        path, found, expanded = depth_limited_dfs(problem, depth)
        total_expanded += expanded

        if found:
            elapsed = time.time() - start_time
            print(f"[IDDFS] solution at depth {depth} | expanded {total_expanded} nodes | {elapsed:.4f}s")
            return path
        if max_depth is not None and depth >= max_depth:
            print(f"[IDDFS] no solution within depth {max_depth} | expanded {total_expanded} nodes")
            return []
        depth += 1

    return []
```

Step-by-step:

1. Validate and coerce `max_depth` to an integer (or `None` for unbounded), raising an error on bad input.
2. Track total expansions and start a wall-clock timer to report performance.
3. Begin with a depth cap of 1 and loop, calling `depth_limited_dfs` for the current limit.
4. Accumulate expansions from each limited search to see the total work done across iterations.
5. On success, print the depth reached, total expansions, and elapsed seconds, then return the path.
6. If the configured `max_depth` is reached without a goal, print a no-solution message and return an empty path.
7. Otherwise increment the depth limit and continue; the terminal `return []` is defensive for completeness.

2.2 searchAgents.py

- `SearchAgent` now accepts `maxDepth` (default 1000) and safely casts it for searches that take a `max_depth` or `limit` parameter (covers IDDFS and DLDFS). This prevents string parsing errors when using CLI arguments.
- Added `IterativeDeepeningAgent`, a convenience wrapper that selects `iterativeDeepeningSearch` with the chosen problem type and depth cap.

Key snippet:

```
func = getattr(search, fn)
if func and ('max_depth' in func.__code__.co_varnames or 'limit' in func.__code__.
    co_varnames):
    depth_limit = int(maxDepth) if maxDepth is not None else None
...
elif 'max_depth' in func.__code__.co_varnames:
    self.searchFunction = lambda x: func(x, max_depth=depth_limit)
elif 'limit' in func.__code__.co_varnames:
    self.searchFunction = lambda x: func(x, limit=depth_limit)
```

2.3 pacman.py

- Added a new CLI flag `--maxDepth` to pass depth limits from the command line into agents. The flag is optional; leaving it unset uses the agent defaults.

Key snippet:

```
parser.add_option('--maxDepth', dest='maxDepth', type='int',
    help=default('Maximum search depth for depth-limited agents'), default=None)
...
if options.maxDepth is not None:
    agentOpts.setdefault('maxDepth', options.maxDepth)
```

3 Behavioral Impact

- Depth-limited DFS can be invoked directly via `fn=dldfs,maxDepth=#` for fixed-cap exploratory runs.
- Users can now run IDDFS with configurable depth caps directly from the CLI (e.g., `fn=iddfs,maxDepth=200`).
- Depth parsing is robust to string inputs from the CLI, eliminating type errors when composing agent arguments.
- Iterative deepening reports the depth at which the solution is found, total nodes expanded, and wall-clock time, aiding comparison against single-pass DFS or BFS.

4 Usage Examples

- Single depth-limited DFS run:
  `python pacman.py -p SearchAgent -a fn=dldfs,maxDepth=200 --layout mediumMaze --frameTime 0`
- Full IDDFS run on a medium maze:
  `python pacman.py -p SearchAgent -a fn=iddfs,maxDepth=200 --layout mediumMaze --frameTime 0`
- Convenience agent:
  `python pacman.py -p IterativeDeepeningAgent -a maxDepth=150 --layout smallClassic --frameTime 0`

5 IDDFS Observed Metrics

Command: `python pacman.py -p SearchAgent -a fn=iddfs,maxDepth=200 --layout mediumMaze --frameTime 0 -q`
Run output:

```
[IDDFS] solution at depth 68 | expanded 10063 nodes | 0.0226s
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 10063
```

Pacman wins with score 442 using depth 68. Lower depth caps (e.g., 25) would yield no movement because the solution exceeds the limit.

6 Depth-Limited DFS (DLDFS) Output

Command: `python pacman.py -p SearchAgent -a fn=dldfs,maxDepth=200 --layout mediumMaze --frameTime 0 -q`
Run output:

```
[SearchAgent] using function dldfs with max depth 200
[SearchAgent] using problem type PositionSearchProblem
[DLDFS] solution within depth 200 | expanded 146 nodes | 0.0005s
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
```

DLDFS finishes quickly with far fewer expansions but returns a longer path (cost 130) because it follows the DFS order at the given limit rather than the shallowest goal.

7 Evaluation Metrics

Depth-Limited DFS (DLDFS)

- Command: `python pacman.py -p SearchAgent -a fn=dldfs,maxDepth=200 --layout mediumMaze --frameTime 0 -q`
- Metrics: ˜0.0005s, 146 nodes expanded, path cost 130.
- Upside: extremely fast and low memory; minimal re-expansion; good when any goal within the cap is acceptable.
- Downside: path follows DFS order, not necessarily shallowest or cheapest; sensitive to successor ordering and the chosen cap.

Iterative Deepening DFS (IDDFS)

- Command: `python pacman.py -p SearchAgent -a fn=iddfs,maxDepth=200 --layout mediumMaze --frameTime 0 -q`
- Metrics: ˜0.0226s, 10063 nodes expanded, path cost 68.
- Upside: finds the shallowest goal reachable within the cap; complete when unbounded; still uses DFS-like memory.
- Downside: heavy re-expansion across depth layers; slower than single-run DFS at the same cap.

Comparison

- On `mediumMaze` with cap 200: DLDFS is 45x faster and 69x fewer expansions but returns a longer path (cost 130) than IDDFS (cost 68).
- Choose DLDFS for quick, memory-light exploratory runs; choose IDDFS when you need the shallowest solution and can tolerate extra expansions.

8 Testing Notes

Practical checks were run headless with `--frameTime 0` to avoid graphics overhead. To validate locally:

- Depth-limited DFS sample: `python pacman.py -p SearchAgent -a fn=dldfs,maxDepth=200 --layout mediumMaze --frameTime 0 -q`
- Full IDDFS path on a medium maze: `python pacman.py -p SearchAgent -a fn=iddfs,maxDepth=200 --layout mediumMaze --frameTime 0`
- Verify depth parsing via a small cap (expected "no solution within depth" message) and a sufficient cap (solution found).
- Use `--maxDepth` with other agents if they consume depth limits.

9 Algorithm Comparison: DFS Variants

9.1 DFS

Advantages: Low memory; quickly descends into deep mazes.
Disadvantages: May miss solutions in infinite spaces; can return long, non-optimal paths; sensitive to successor order.

9.2 Depth-Limited DFS

Advantages: Prevents runaway depth; useful when a shallow bound is known.
Disadvantages: Incomplete if the limit is too small; still order-sensitive; requires picking a bound.

9.3 Iterative Deepening DFS

Advantages: Completeness similar to BFS with DFS-like memory; finds the shallowest goal without large queues; tunable via depth cap.
Disadvantages: Re-expands nodes at each depth layer; overhead grows if the optimal depth is large; needs a sensible cap to avoid long runs.

9.4 BFS (Reference)

Advantages: Finds shortest path in steps; complete on finite graphs.
Disadvantages: High memory footprint; slower on large branching factors; ignores edge costs.

Recommended for Pacman

IDDFS is the best balance when goal depth is unknown and memory is limited. Choose BFS when you need the shallowest path and can afford memory; use plain DFS for quick exploratory runs on small maps; pick depth-limited DFS when you trust a specific depth bound.