

Depth-First Interactive-Deepening: An Optimal Admissible Tree Search

by R. E. Korf

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

<http://www.iis.sinica.edu.tw/~tshsu>

Abstract

- **The complexities of various search algorithms are considered in term of time, space, and cost of solution path.**
 - Breadth-first search (BFS)
 - Depth-first search (DFS)
 - Iterative-deepening DFS
 - Bi-directional search
 - Heuristic search

Definitions

- Node branching factor b : the number of different new states generated from a state, assumed to be constant here
- Edge branching factor e : the number of possible new states generated from a state (duplications are counted twice), assumed to be constant here
- Depth of a solution d : the shortest length from the initial state to one of the goal states
- A **brute-force search** is a search algorithm that uses only information about
 - the initial state,
 - operators on finding the states adjacent to a state,
 - and a test function whether a goal is reached.

Breadth-first search (BFS)

- $deeper(N)$: gives the set of all possible states that can be reached from the state N
- **Algorithm BFS**
 - Initialize a queue Q
 - $N = N_0$ is the initial state
 - **while** $deeper(N)$ is not empty **do**
 - ▷ *If one of the states in $deeper(N)$ is goal, then return succeed*
 - ▷ *Add states in $deeper(N)$ to the queue Q*
 - ▷ *Remove a state from Q and let it be N*
 - **return fail**

BFS: analysis

■ Time complexity:

- $b + b^2 + b^3 + \dots + b^d = O(b^d)$

■ Space complexity:

- $O(b^d)$

■ Comments:

- Always finds an optimal solution, i.e., one with the smallest possible d .
- Most critical drawback: huge space requirement.
 - ▷ *It is tolerable for an algorithm to be 100 times slower, but not so for one that is 100 times larger.*

Depth-first search (DFS)

- $next(current, S)$ returns the state next to “current” in $deeper(N)$
 - Assume we can generate $next(current, S)$ based on information stored in “current” and S .
- **Algorithm**
 - Initialize a stack S
 - Push $(null, N_0)$ to S where N_0 is the initial state
 - while S is not empty do
 - ▷ Pop $(current, N)$ from S
 - ▷ $R = next(current, N)$
 - ▷ Push (R, N) to S
 - ▷ If R is a goal, then return succeed
 - ▷ Push $(null, R)$ to S
 - ▷ **Can introduce some cut-off depth here to not go too deep**
 - return fail

DFS: analysis

- **Time complexity:**
 - $O(e^d)$
- **Space complexity:**
 - $O(d)$
- **Comments:**
 - May not find a solution in time without good cut-off depth.
 - May not find an optimal solution.
 - Heavily depends on the **move ordering**.
 - ▷ *Which one to search first when you have multiple choices for your next move?*
 - A node can be searched many times.
 - ▷ *Need to do something, e.g., hashing, to avoid researching too much.*
 - ▷ *Need to consider the effort to memorize and the effort to research.*
 - **Most critical drawback: huge and unpredictable time complexity**

Depth-first iterative-deepening (DFID)

- $DFS(i)$: DFS with a depth cut off at i
- **Algorithm**
 - $depth = 1$
 - **while** $depth < cut_off_depth$ **do**
 - ▷ *if $DFS(depth)$ returns succeed than return succeed*
 - ▷ $depth = depth + 1$
 - **return fail**
- **Space complexity:**
 - $O(d)$

Time complexity of DFID

- The nodes at depth i are generated $d - i + 1$ times.
 - There are b^{i-1} nodes at depth i .
- Total number of nodes visited $M(b, d)$ is
 - $db^0 + (d - 1)b + (d - 2)b^2 + \dots + 2b^{d-1} + b^d$
 - $= b^d(1 + 2b^{-1} + 3b^{-2} + \dots + b^{-d})$
 - $\leq b^d(1 - 1/b)^{-2}$ if $b > 1$
- Examples:
 - When $b = 2$, $M(b, d) \leq 4b^d$.
 - When $b = 3$, $M(b, d) \leq 9/4b^d$.
 - When $b = 4$, $M(b, d) \leq 16/9b^d$.
 - When $b = 5$, $M(b, d) \leq 25/16b^d$.
 - $M(b, d) = O(b^d)$ with a small constant factor.
- Comments:
 - No need to worry about a good cut-off depth as in DFS.
 - Good for a tournament situation where each move has a limited amount of time.

Bi-directional search

- Combined with iterative-deepening
- $DFS(S, G, successor, i)$: DFS with the set of starting states S , goal states G , $successor$ function and depth limit i .
 - $successor$ is *deeper* for forward searching
 - $successor$ is *prev* for backward searching
 - ▷ Given a state S_i , $prev(S_i)$ gives all states that can reach S_i in one step.
- Algorithm
 - $depth = 1$
 - **while** $depth < cut_off_depth$ **do**
 - ▷ if $DFS(\{S_0\}, G, deeper, depth)$ returns succeed then return succeed
Stores all states at depth i in an area H
 - ▷ if $DFS(G, H, prev, depth)$ returns succeed then return succeed
 - ▷ if $DFS(G, H, prev, depth + 1)$ returns succeed then return succeed
 - ▷ $depth = depth + 1$
 - return fail
- Backward searching at depth $depth + 1$ is needed to find odd-length solutions.

Bi-directional search: analysis

■ Time complexity:

- $O(b^{d/2})$

■ Space complexity:

- $O(b^{d/2})$

■ Comments:

- Runs well in practice.
- Depth of the solution is expected to be the same for a normal uni-directional search, however the number of nodes visited is greatly reduced.
- Pay the price of storing solutions at half depth.
- Trade off between time and space.
- Q:
 - ▷ *How about using BFS in forward searching?*
 - ▷ *How about using BFS in backward searching?*
 - ▷ *How about using BFS in both directions?*

Heuristic search

- Combining DFID with best-first heuristic search such as A*.
- A* search: branch and bound with lower-bound estimation.
 - Initialize a priority queue Q to store partial paths.
 - ▷ Initially, store only a path with the starting node only.
 - ▷ Paths in Q are sorted according to their current cost plus a lower bound on the remaining distances.
 - while Q is not empty do
 - ▷ Remove the first partial path P from Q
 - ▷ Form paths from P by extending one step
 - ▷ If the goal is reached then return succeed
 - ▷ Insert all generated paths to Q
 - ▷ If two paths reach a common node, keep only one with the least cost
 - return fail

A*: analysis

■ Comments:

- When a path is inserted, check for whether it has reached a common node with some old paths.
- Given a path P ,
 - ▷ let $g(P)$ be the current cost of P
 - ▷ let $h(P)$ be the estimation of remaining cost of P
- How to find a good h is the key of A* algorithms.
- It is known that if h never overestimates the actual cost to the goal (this is called **admissible**), then A* always find an optimal solution.
 - ▷ Q: How to prove this?

$$\text{IDA}^* = \text{DFID} + \text{A}^*$$

■ **IDA^{*}: iterative-deepening A^{*}**

- *threshold = h(null)*
- **while true do**
 - ▷ *Perform a DFS with the constraint that cutting off a path P with $g(P) + h(P) > \text{threshold}$*
 - ▷ *If the goal is found, then return succeed*
 - ▷ *threshold = the least $g + h$ cost among all paths being cut*
- **return fail**

IDA*

■ Comments:

- A cost function h is **monotone** if for each node n and $s(n)$ where $s(n)$ is a successor of n , $h(n) \leq h(s(n))$.
- Given an admissible monotone cost function, IDA* will find a solution of least cost if one exists.
 - ▷ *Cost function is the knowledge used in searching.*
 - ▷ *Combine knowledge and search!*
 - ▷ *Need to balance the amount of time spent in realizing knowledge and the time used in searching.*
- IDA* is optimal in terms of solution cost, time, and space over the class of admissible best-first searches on a tree.

15 puzzle

■ Introduction of the game:

- 15 tiles in a 4*4 square with numbers from 1 to 15.
- One empty cell.
- A tile can slide horizontally or vertically into an empty cell.
- From an initial position, try to rearrange the tiles into a goal position.

■ Examples:

- Initial position:

10	8	*	12
3	7	6	2
1	14	4	11
15	13	9	5
- Goal position:

*	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

■ Total number of positions: $16! = 20,922,789,888,000 \leq 2.1 * 10^{13}$.

- It is feasible to enumerate all possible positions now (2007).

Solving 15 puzzles

- Using DEC 2060 a 1 MIPS machine: solved the 15 puzzle problem within 30 CPU minutes for all testing positions, generating over 1.5 million nodes per minute.
 - The average solution length was 53 moves.
 - The maximum was 66 moves.
 - IDA* generated more nodes than A*, but ran faster due to less overhead per node and requires a linear space.
- Heuristics used:
 - $g(P)$: the number of moves made so far
 - $h(P)$: the Manhattan distance between the current board to the goal position.
 - ▷ Suppose a tile is currently at (i, j) and its goal is at (i', j') , then the Manhattan distance for this tile is $|i - i'| + |j - j'|$.
 - ▷ The Manhattan distance between a position and a goal position is the sum of the Manhattan distance of each tile.
 - ▷ $h(P)$ is admissible.
 - ▷ $h(P)$ is monotone.

What else can be done?

- Bi-directional search and IDA*?
- How to get a better move ordering in DFS?
- Balancing in resource allocation:
 - The amount of effort to memorize past results vs. the amount of efforts to search again.
 - The amount of effort to get a better heuristic, e.g., the cost function.
 - The amount of resources spent in implementing a better heuristic and the amount of resources spent in searching.
- Can these techniques be applied to two-person games?

References and further readings

- * R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
- R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22, 2002.