Basic Search Algorithms

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 terms of time, space, and cost of the solution paths.
  - Systematic brute-force search
    - Breadth-first search (BFS)
    - Depth-first search (DFS)
    - Depth-first Iterative-deepening (DFID)
    - Bi-directional search
  - Heuristic search: best-first search
    - A*
    - IDA*
- The issue of storing information in DISK instead of main memory.
- Solving 15-puzzle.
Definitions

- **Node branching factor** $b$: the number of different new states generated from a state.
  - Average node branching factor.
  - Assumed to be a constant here.

- **Edge branching factor** $e$: the number of possible new, maybe duplicated, states generated from a state.
  - Average node branching factor.
  - Assumed to be a constant here.

- **Depth of a solution** $d$: the shortest length from the initial state to one of the goal states
  - The depth of the root is 0.

- **A search program finds a goal state starting from the initial state by exploring states in the state space.**
  - Brute-force search
  - Heuristic search
Brute-force search

- A brute-force search is a search algorithm that uses information about
  - the initial state,
  - operators on finding the states adjacent to a state,
  - and a test function whether a goal is reached.

- A “pure” brute-force search program.
  - A state maybe re-visited many times.

- An “intelligent” brute-force search algorithm.
  - Make sure a state will be visited a limited number of times.
    - Make sure a state will be eventually visited.
A “pure” brute-force search is a brute-force search algorithm that does not care whether a state has been visited before or not.

Algorithm Brute-force($N_0$)

\{ * do brute-force search from the starting state $N_0$ * \}

* current $\leftarrow N_0$

* While true do

  ▶ * If current is a goal, then return success *

  ▶ current $\leftarrow$ a state that can reach current in one step

Comments

* Very easy to code and use very little memory.
* May take infinite time because there is no guarantee that a state will be eventually visited.
Intelligent brute-force search

- An “intelligent” brute-force search algorithm.
  - Assume $S$ is the set of all possible states
  - Use a systematic way to examine each state in $S$ one by one so that
    - A state is not examined too many times — does not have too many duplications.
    - It is efficient to find an unvisited state in $S$.

- Need to know whether a state has been visited before efficiently.

- Some notable algorithms.
  - Breadth-first search (BFS).
  - Depth-first search (DFS) and its variations.
  - Depth-first Iterative deepening (DFID).
  - Bi-directional search.
Breadth-first search (BFS)

- deeper\((N)\): gives the set of all possible states that can be reached from the state \(N\).
  - It takes at least \(O(e)\) time to compute deeper\((N)\).
  - The number of distinct elements in deeper\((N)\) is \(b\).

- **Algorithm BFS**\(\(N_0\)\) \{ /* do BFS from the starting state \(N_0\) */
  - If the starting state \(N_0\) is a goal, then return success
  - Initialize a Queue \(Q\)
  - Add \(N_0\) to \(Q\);
  - While \(Q\) is not empty do
    - Remove a state \(N\) from \(Q\)
    - If one of the states in deeper\((N)\) is goal, then return success
    - Add states in deeper\((N)\) to \(Q\)
  - Return fail
BFS: analysis (1/2)

- **Space complexity:**
  - $O(b^d)$
    - The average number of distinct elements at depth $d$ is $b^d$.
    - We may need to store all distinct elements at depth $d$ in the Queue.

- **Time complexity:**
  - $1 \cdot e + b \cdot e + b^2 \cdot e + b^3 \cdot e + \cdots + b^{d-1} \cdot e = (b^d - 1) \cdot e / (b - 1) = O(b^{d-1} \cdot e)$, if $b$ is a constant.
    - For each element $N$ in the Queue, it takes at least $O(e)$ time to find deeper($N$).
    - It is always true that $e \geq b$. 

A smart mechanism is needed if you want to make sure each node is visited at most once.

- It needs to keep track of all nodes visited so far.
  \[1 + b + b^2 + b^3 + \ldots + b^d = \frac{(b^{d+1} - 1)}{(b - 1)} = O(b^d)\]

- Need a good algorithm to check for states in \(\text{deeper}(N)\) are visited or not.
  - Hash
  - Binary search
  - \ldots

- This is not really needed since it won’t guarantee to improve the performance because of the extra cost to maintain and compare states in the pool of visited states!
BFS: comments

- Always finds an optimal solution, i.e., one with the smallest possible depth $d$.
  - Do not need to worry about falling into loops if there is always a goal.
    - Need to store nodes that are visited before if it is possible to have no solution.

- 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.
BFS: ideas when there is little memory

- What can be done when you do not have enough main memory?

  - DISK
    - Store states that has been visited before into DISK and maintain them as sorted.
    - Store the QUEUE into DISK.
  
  - Memory: buffers
    - Most recently visited nodes.
    - Candidates of possible newly explored nodes.
  
  - Merge the buffer of visited nodes with the one in DISK when memory is full.
    - We only need to know when a newly explored node has been visited or not when it is about to be removed from the QUEUE.
    - The decision of whether it has been visited or not can be delayed.
  
  - Append the buffer of newly explored nodes to the QUEUE in DISK when memory is full or it is empty.
BFS: disk based (1/2)

- **Algorithm** BFS$_{disk}(N_0)$
  
  ```
  \{ * do disk based BFS from the starting state $N_0$ *
  
  * If the starting state $N_0$ is a goal, then return success
  * Initialize a Queue $Q_d$ of nodes to visited using DISK
  * Initialize a buffer $Q_m$ of nodes to visit using main memory
  * Add $N_0$ to $Q_d$;
  * While $Q_d$ and $Q_m$ are not both empty do
    
    - If $Q_d$ is empty, then {
      
      Sort $Q_m$;
      Write $Q_m$ to $Q_d$;
      Empty $Q_m$
    }
    
    - Remove a state $N$ from $Q_d$
    
    - If one of the states in deeper($N$) is goal, then return success
    
    - Add states in deeper($N$) to $Q_m$;
    
    - If $Q_m$ is full, then {
      
      Sort $Q_m$;
      Append states in $Q_m$ to $Q_d$;
      Empty $Q_m$
    }

  * Return fail
  ```

TCG: Basic Search, 20131101, Tsan-sheng Hsu ©
BFS: disk based (2/2)

- States to be visited are already sorted using their depths in ascending order.
  - No extra work is needed.
  - The states are appended according to their depths.
Disk based algorithms

- When data cannot be loaded into the memory, you need to re-invent algorithms even for tasks that may look simple.
  - Batched processing.
    - Accumulate tasks and then try to perform these tasks when they need to.
    - Combine tasks into one to save disk I/O time.
    - Order disk accessing patterns.

- Main ideas:
  - It is not too slow to read all records of a large file in sequence.
  - It is very slow to read every record in a large file in a random order.
  - Sorting of data stored on the DISK can be done relatively efficient.
  - When two files are sorted, it is cost effective to
    - compare the difference of them;
    - merge them.
Implementation of the QUEUE.
- QUEUE can be stored in one disk file.
- Newly explored ones are appended at the end of the file.
- Always retrieve the one at the head of the file.

A newly explored node will be explored after the current QUEUE is empty.
- property of BFS.
Disk based BFS (2/2)

- How to find out a newly explored node has been visited before or not if this is desired?
  - Maintain the list of visited nodes on DISK sorted according to some index function on ID’s of the nodes.
    - When the member buffer is full, sort it according to their indexes.
    - Merge the sorted list of newly visited nodes in buffer into the one stored on DISK.
  - We can easily compare two sorted lists and find out the intersection or difference of the two.
    - We can easily remove the ones that are already visited before once $Q_m$ is sorted.
    - To revert items in $Q_m$ back to its the original BFS order, which is needed for persevering the BFS search order, we need to sort again using the original BFS ordering.

- Why we can delay the decision of whether a newly explored node has been visited or not?
  - We only need to know when a newly explored node has been visited or not when it is about to be removed from the QUEUE.
  - The decision of whether it has been visited or not can be delayed.
**Depth-first search (DFS)**

- \( \text{next}(\text{current}, N) \): returns the state next to the state “current” in \( \text{deeper}(N) \).
  - Assume states in \( \text{deeper}(N) \) are given a linear order with dummy first and last elements both being null, and assume \( \text{current} \in \text{deeper}(N) \).
  - Assume we can efficiently generate \( \text{next}(\text{current}, N) \) based on “current” and \( N \).

- **Algorithm DFS(** \( N_0 \) **) { * do DFS from the starting state \( N_0 \) * }
  - Initialize a Stack \( S \)
  - Push \((\text{null}, N_0)\) to \( S \)
  - While \( S \) is not empty do
    - \( \text{Pop} \ (\text{current}, N) \) from \( S \)
    - \( R \leftarrow \text{next}(\text{current}, N) \)
    - If \( R \) is null, then continue \{ * all children of \( N \) are searched * \}
    - If \( R \) is a goal, then return success
    - Push \((R, N)\) to \( S \)
    - If \( R \) is already in \( S \), then continue \{ * to avoid loops * \}
    - Can introduce some cut-off depth here in order not to go too deep
    - Push \((\text{null}, R)\) to \( S \) \{ * search deeper * \}
  - Return fail
DFS: analysis (1/2)

- **Time complexity:**
  - $O(e^d)$
    - *The number of possible branches at depth $d$ is $e^d$.*
  - This is only true when the game tree searched is not skewed.
    - *The leaves of the game tree are all of $O(d)$.*
  - It can be as bad of $O(e^D)$ where $D$ is the maximum depth of the tree.

- **Space complexity:**
  - $O(d)$
    - *Only need to store the current path in the Stack.*
  - This is also only true when the tree is not skewed.
  - It can be as bad of $O(D)$ where $D$ is the maximum depth of the tree.
DFS: analysis (2/2)

- May need to store the set of visited nodes in order not to visit a node too many times.
  - Methods:
    - Hash table
    - Sorted list and then use binary search
    - Balanced search tree
  - This is a real issue in order to get out of a long and wrong branch as fast as you can.
DFS: comments

- Without a good cut-off depth, it may not be able to find a solution in time.
- May not find an optimal solution at all.
- 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 balance the effort to memorize and the effort to research.
- Most critical drawback: huge and unpredictable time complexity.
DFS: when there is little memory

- Difficult to implement a STACK on a DISK so far if the STACK is too large to be fit into the main memory.
- We need to decide instantly whether a node is visited or not.
  - The decision of whether a node is visited or not cannot be delayed.
    - Batch processing is not working here.
    - It may take too much time to handle a disk based hash table.
- Use data compression and/or bit-operation techniques to store as many visited nodes as possible.
  - Some nodes maybe visit again and again.
  - Need a good heuristic to store the most frequently visited nodes.
    - Avoid swapping too often.
DFS with a depth limit

- Do DFS from the starting state \( N_0 \) without exceeding a given depth limit.
  - \( \text{length}(\text{root}, y) \): the number of edges visited from the root node \( \text{root} \) to the node \( y \) during DFS searching.

- Algorithm \( \text{DFS}_{\text{depth}}(N_0, \text{limit}) \)
  - Initialize a Stack \( S \)
  - Push \((\text{null}, N_0)\) to \( S \) where \( N_0 \) is the initial state
  - While \( S \) is not empty do
    - Pop \((\text{current}, N)\) from \( S \)
    - \( R \leftarrow \text{next}(\text{current}, N) \)
    - If \( R \) is a goal, then return success
    - If \( R \) is \text{null}, then continue \{\text{all children of } N \text{ are searched}\}
    - Push \((R, N)\) to \( S \)
    - If \( \text{length}(N_0, R) > \text{limit} \), then continue \{\text{cut off}\}
    - If \( R \) is already in \( S \), then continue \{\text{to avoid loops}\}
    - Push \((\text{null}, R)\) to \( S \) \{\text{search deeper}\}
  - Return fail
Depth-first iterative-deepening (DFID)

- **DFS\(_{\text{depth}}(N, \text{current\_limit})\)**: DFS from the starting state \(N\) and with a depth cut off at the depth \(\text{current\_limit}\).

- **Algorithm DFID\((N_0, \text{cut\_off\_depth})\) { */ do DFID from the starting state \(N_0\) with a depth limit \(\text{cut\_off\_depth}\) */}
  - \(\text{current\_limit} \leftarrow 0\)
  - While \(\text{current\_limit} < \text{cut\_off\_depth}\) do
    - If **DFS\(_{\text{depth}}(N_0, \text{current\_limit})\)** finds a goal state \(g\), then return \(g\) as the found goal state
    - \(\text{current\_limit} \leftarrow \text{current\_limit} + 1\)
  - Return fail

- **Space complexity:**
  - \(O(d)\)
The branches at depth $i$ are generated $d - i + 1$ times.

- There are $e^i$ branches at depth $i$.

Total number of branches visited $M(e, d)$ is

$$(d + 1)e^0 + de^1 + (d - 1)e^2 + \cdots + 2e^{d-1} + e^d$$

$$= e^d(1 + 2e^{-1} + 3e^{-2} + \cdots + (d + 1)e^{-d})$$

$$\leq e^d(1 - 1/e)^{-2} \text{ if } e > 1$$

Analysis:

1. $(1 - x)^{-2} = 1/(1 - 2x + x^2) = 1 + 2x + 3x^2 + \cdots + kx^{k-1} + (k + 1)x^k - kx^{k+1}$.
2. Hence $1 + 2x + 3x^2 + \cdots + kx^{k-1} \leq (1 - x)^{-2}$, if $|x| < 1$.
3. Since $|x| < 1$,

$$\lim_{k \to \infty} ((k + 1)x^k - kx^{k+1}) = 0.$$

4. If $k$ is large enough and $|x| < 1$, then $(1 - x)^{-2} \approx 1 + 2x + 3x^2 + \cdots + kx^{k-1}.$
Let $M(e, d)$ be the total number of branches visited by DFID with an edge branching factor of $e$ and depth $d$.

Examples:
- When $e = 2$, $M(e, d) \leq 4e^d$.
- When $e = 3$, $M(e, d) \leq 9/4e^d$.
- When $e = 4$, $M(e, d) \leq 16/9e^d$.
- When $e = 5$, $M(e, d) \leq 25/16e^d < 1.57e^d$.
- ... 
- When $e = 30$, $M(e, d) \leq 900/841e^d < 1.071e^d$.

$M(e, d) = O(e^d)$ with a small constant factor when $e$ is sufficiently large.
DFID: comments

- No need to worry about a good cut-off depth as in DFS.
- Still need a mechanism to decide instantly whether a node has been visited before or not.
- Good for a tournament situation where each move needs to be made in a limited amount of time.

Q:

- Does DFID always find an optimal solution?
- How about BFID?
DFS with depth limit and direction (1/2)

- Two refined service routines when direction of the search is considered:
  - $\text{DFS}_{\text{dir}}(B, G, \text{successor}, i)$: DFS with the set of starting states $B$, goal states $G$, successor function and depth limit $i$.
  - $\text{next}_{\text{dir}}(\text{current}, \text{successor}, N)$: returns the state next to the state “current” in $\text{successor}(N)$.

- In the above two routines:
  - $\text{successor}$ is deeper for forward searching
  - $\text{successor}$ is $\text{prev}$ for backward searching

- Note:
  - Given a state $N$, $\text{prev}(N)$ gives all states that can reach $N$ in one step.
  - Given a state $N$, $\text{deeper}(N)$ gives the set of all possible states that can be reached from the state $N$ in one step.
DFS with depth limit and direction (2/2)

- **DFS$_{dir}(B, G, successor, i)$**: DFS with the set of starting states $B$, goal states $G$, successor function and depth limit $i$.

- **Algorithm DFS$_{dir}(B, G, successor, limit)$**
  - Initialize a Stack $S$
  - For each possible starting state $t$ in $B$ do
    - Push $(null, t)$ to $S$
  - While $S$ is not empty do
    - Pop $(current, N)$ from $S$
    - $R \leftarrow$ next$_{dir}(current, successor, N)$
    - If $R$ is a goal in $G$, then return success
    - If $R$ is null, then continue {∗ all children of $N$ are searched ∗}
    - Push $(R, N)$ to $S$
    - If length($B, R$) $>$ limit, then continue {∗ cut off ∗}
    - If $R$ is already in $S$, then continue {∗ to avoid loops ∗}
    - Push $(null, R)$ to $S$ {∗ search deeper ∗}
  - Return fail

- Note length($B, x$) is the length of a shortest path between the state $x$ and a state in $B$.  

---

TCG: Basic Search, 20131101, Tsan-sheng Hsu © 2013
Bi-directional search

- Combined with iterative-deepening.
- \( \text{DFS}_{\text{dir}}(B, G, \text{successor}, i) \): DFS with the set of starting states \( B \), goal states \( G \), successor function and depth limit \( i \).
  - \text{successor} is deeper for forward searching
  - \text{successor} is \( \text{prev} \) for backward searching
  - Given a state \( S_i \), \( \text{prev}(S_i) \) gives all states that can reach \( S_i \) in one step.

- Algorithm BDS\((N_0, \text{cut off depth})\)
  - current\_limit \( \leftarrow 0 \)
  - while current\_limit < cut\_off\_depth do
    - if \( \text{DFS}_{\text{dir}}(\{N_0\}, G, \text{deeper, current\_limit}) \) returns success,
      then return success \{ * forward searching * \}
    - else store all states at depth = current\_limit in an area \( H \)
    - if \( \text{DFS}_{\text{dir}}(G, H, \text{prev, current\_limit}) \) returns success,
      then return success \{ * backward searching * \}
    - if \( \text{DFS}_{\text{dir}}(G, H, \text{prev, current\_limit + 1}) \) returns success,
      then return success \{ * in case the optimal solution is odd-lengthed * \}
    - current\_limit \( \leftarrow \) current\_limit + 1
  - return fail

- Backward searching at depth = current\_limit + 1 is needed to find odd-lengthed optimal solutions.
Bi-directional search: Example
Bi-directional search: analysis

- **Time complexity:**
  - $O(e^{d/2})$

- **Space complexity:**
  - $O(e^{d/2})$: needed to store the half-way meeting points $H$.

- **Comments:**
  - Run 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.
  - Need to know how to enumerate the set of goals.
  - Trade off between time and space.
    - What can be stored on DISK?
    - What operations can be batched?

- **Q:**
  - How about using BFS in forward searching?
  - How about using BFS in backward searching?
  - How about using BFS in both directions?
Heuristic search

- **Heuristics**: criteria, methods, or principles for deciding which among several alternative courses of actions promises to be the most effective in order to achieve some goal [Judea Pearl 1984].
  - Need to be simple and effective in discriminate correctly between good and bad choices.

- A **heuristic search** is a search algorithm that uses information about
  - the initial state,
  - operators on finding the states adjacent to a state,
  - a test function whether a goal is reached, and
  - heuristics to pick the next state to explore.

- A “good” heuristic search algorithm:
  - States that are not likely leading to the goals will not be explored further.
    - A state is cut or pruned.
  - States are explored in an order that are according to their likelihood of leading to the goals → good move ordering.
Heuristic search: A*

- Combining DFID with best-first heuristic search such as A*.
- A* search: branch and bound with a lower-bound estimation.
- Algorithm $A^*(N_0)$
  - Initialize a Priority Queue $PQ$ to store partial paths with keys being the costs of paths.
    - Initially, store only a path with the starting node $N_0$ only.
    - Paths in $PQ$ are sorted according to their current costs plus a lower bound on the remaining distances.
  - While $PQ$ is not empty do
    - Remove a path $P$ with the least cost from $PQ$
    - 11: If the goal is found, then return success
    - 12: Find extended paths from $P$ by extending one step
    - Insert all generated paths to $PQ$
    - Update $PQ$
    - 15: If two paths reach a common node then keep only one with the least cost
  - Return fail
A* algorithm

**Cost function:**
- Given a path $P$,
  - let $g(P)$ be the current cost of $P$;
  - let $h(P)$ be the estimation of remaining, or heuristic cost of $P$;
  - $f(P) = g(P) + h(P)$ is the cost function.

- How to find a good $h()$ is the key of an A* algorithm?
- It is known that if $h()$ never overestimates the actual cost to the goal (this is called admissible), then A* always finds an optimal solution.
  - Q: How to prove this?
- Note: If $h()$ is admissible and $P$ reaches the goal, then $h(P) = 0$ and $f(P) = g(P)$.

**Checking of the termination condition:**
- We need to check for whether a goal is found only when a path is popped from the $PQ$, i.e., at Line 11.
- We cannot check for whether a goal is found when a path is generated and inserted into the $PQ$, i.e., at Line 12.
  - We will not be able find the optimal solution if we do the checking at Line 12.
When a path is inserted, namely at Line 15, check for whether it has reached some nodes that have been visited before.
- It may take a huge space and a clever algorithm to implement an efficient Priority Queue.
- It may need a clever data structure to efficiently check for possible duplications.

Cost function:
- Need an lower bound estimation that is as large as possible.
- Can design the cost function so that $A^*$ emulates the behavior of other search routines.

Q:
- What disk based techniques can be used?
- Why do we need a non-trivial $h(P)$ that is admissible?
- How to design an admissible cost function?
DFS with a threshold

- **DFS\textsubscript{cost}(N, f, threshold)** is a version of DFS with a starting state $N$ and a cost function $f$ that cuts off a path when its cost is more than a given threshold.
  - **DFS\textsubscript{depth}(N, cut off f_depth)** is a special version of DFS\textsubscript{cost}(N, f, threshold).

- **Algorithm DFS\textsubscript{cost}(N_0, f, threshold)**
  - 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 $S$
    - $R \leftarrow$ next(current, $N$) \{* pick a good move ordering here *\}
    - If $R = \text{null}$, then continue \{* all children of $N$ are searched *\}
    - Push ($R$, $N$) to $S$
    - Let $P$ be the path from $N_0$ to $R$
    - If $f(P) > \text{threshold}$, then continue \{* cut off *\}
    - If $R$ is a goal, then return success \{* Goal is found! *\}
    - If $R$ is already in $S$, then continue \{* to avoid loops *\}
    - Push (null, $R$) to $S$ \{* search deeper *\}
  - Return fail
How to pick a good move ordering (1/2)

Instead of just using \( \text{next}(\text{current}, N) \) to find the next unvisited neighbors of \( N \) with the information of the last visited node being \( \text{current} \), we do the followings.

- Use a routine to order the neighbors of \( N \) so that it is always the case the neighbors are visited from low cost to high cost.
- Let this routine be \( \text{next1}(\text{current}, N) \).
- Note we still need dummy first and last elements being \( \text{null} \).
How to pick a good move ordering (2/2)

- **Algorithm DFS1\(_{cost}(N_0,f,\text{threshold})\)**
  - 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 \(S\)
    - \(R \leftarrow \text{next1}(current,N)\)
    - **If** \(R = \text{null}, \text{then continue} \) \{ * all children of \(N\) are searched * \}
    - **Push** \((R,N)\) to \(S\)
    - **Let** \(P\) be the path from \(N_0\) to \(R\)
    - **If** \(f(P) > \text{threshold}, \text{then continue} \) \{ * cut off * \}
    - **If** \(R\) is a goal, then return success
    - **If** \(R\) is already in \(S\), then continue \{ * to avoid loops * \}
    - **Push** \((null,R)\) to \(S\) \{ * search deeper * \}
  - Return fail
How to incorporate ideas from $A^*$

- Instead of using a stack in $\text{DFS}_{\text{cost}}$, use a priority queue.

**Algorithm $\text{DFS2}_{\text{cost}}(N_0, f, \text{threshold})$**

- Initialize a priority queue $PQ$
- Insert $(null, N_0)$ to $PQ$ where $N_0$ is the initial state
- While $PQ$ is not empty do
  - Remove $(current, N)$ with the least cost $f(P)$ for the path $P$ from $N_0$ to $N$ from $PQ$
  - If current is a goal, then return success
  - $R \leftarrow \text{next1}(current, N)$
  - If $R = \text{null}$, then continue \{\text{* all children of $N$ are searched *}\}
  - Insert $(R, N)$ to $PQ$
  - Let $P$ be the path from $N_0$ to $R$
  - If $f(P) > \text{threshold}$, then continue \{\text{* cut off *}\}
  - If $R$ is already in $PQ$, then continue \{\text{* to avoid loops *}\}
  - Insert $(null, R)$ to $PQ$ \{\text{* search deeper *}\}

- Return fail

- It may be costly to maintain a priority queue as in the case of $A^*$. 
IDA* = DFID + A*

- \( \text{DFS}_{\text{cost}}(N, f, \text{threshold}) \) is a version of DFS with a starting state \( N \) and a cost function \( f \) that cuts off a path when its cost is more than a given \( \text{threshold} \).
- \( \text{IDA}^* \): iterative-deepening \( A^* \)
- Algorithm \( \text{IDA}^*(N_0, \text{threshold}) \)
  - \( \text{threshold} \leftarrow h(\text{null}) \)
  - While \( \text{threshold} \) is reasonable do
    - \( \text{DFS}_{\text{cost}}(N_0, g + h(), \text{threshold}) \)
      { * Can also use \( \text{DFS1}_{\text{cost}} \) or \( \text{DFS2}_{\text{cost}} \) here * }
    - If the goal is found, then return success
      - threshold \( \leftarrow \) the least \( g(P) + h(P) \) cost among all paths \( P \) being cut
  - Return fail
IDA*: comments

- **IDA**\(^*\) does not need to use a priority queue as in the case of **A**\(^*\).
  - **IDA**\(^*\) is optimal in terms of solution cost, time, and space over the class of admissible best-first searches on a tree.

- **Issues in updating** *threshold*.
  - Increase too little: re-search too often.
  - Increase too large: cut off too little.
  - Q: How to guarantee optimal solutions are not cut?
    - It can be proved, as in the case of **A**\(^*\), that given an admissible cost function, **IDA**\(^*\) will find an optimal solution, i.e., one with the 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.**
15 puzzle (1/2)

- **Introduction of the game:**
  - 15 tiles in a 4*4 square with numbers from 1 to 15.
  - One empty cell.
  - A tile can be slided horizontally or vertically into an empty cell.
  - From an initial position, slide 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 |
15 puzzle (2/2)

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

- It is feasible, in terms of computation time, to enumerate all possible positions, since 2007.
  - Can use DFS or DFID now.
  - Need to avoid falling into loops or re-visit a node too many times.

- It is still too large to store all possible positions in main memory now (2013).
  - Cannot use BFS efficiently even now.
  - Maybe difficult to find an optimal solution.
  - Maybe able to use disk based BFS.
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.

- Note: Intel Core i7 3960X (6 cores) is rated at 177,730 MIPS and ARM Cortex A7 is rated at 2,850 MIPS.

- Heuristics used:
  - $g(P)$: the number of moves made so far.
  - $h(P)$: the Manhattan distance between the current board and 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 every tile.
  ▶ $h(P)$ is admissible.
What else can be done?

- Bi-directional search and IDA*?
  - How to design a good and non-trivial heuristic function?
- How to find an **optimal** solution?
- How to get a better move ordering in DFS?
- Balancing in resource allocation:
  - The efforts to memorize past results versus the amount of efforts to search again.
  - The efforts to compute a better heuristic, i.e., the cost function.
  - The amount of resources spent in implementing a better heuristic and the amount of resources spent in searching.
- Search in parallel.
- More techniques for disk based algorithms.
- Q: Can these techniques be applied to two-person games?
References and further readings


