Suboptimal and Linear Space Search

Suboptimal Searches

- **Suboptimal search:** Just find any solution fast: Speedy vs. Greedy.
- **Bounded suboptimal search:** Given a bound W we want the solution to be at most W x C* This is called *w-admissible.*
- **Anytime search:** improve the quality as time passes
- **Bounded cost search:** Give B, find a solution below B
- PAC search: given e and d, find a solution that will be of quality e with probability b
- Target-Value Search: Find a solution of exactly X

Bounded Suboptimal Search

- Given a bound W we want the solution to be at most W x C* This is called *w-admissible.*

Weighted A* (WA*)

- A generalized version of A* Where:

$$f(n) = W_g\, g(n) + W_h\, h(n)$$

Rewrite $W=W_h/W_g$ $1<=W=<\infty$

We get

$$f(n) = g(n) + Wh(n)$$

When W=1 it is A*

When $q=\infty$ it is Pure Heuristic Search

WA* is w-admissible

- Proof:

1. At all points of time there is a member of the shortest path in OPEN. Let's call this node node x. So $f(x) = g(x) + h(x) \leq C^*$.
2. Therefore, $f_W(x) = g(x) + Wh(x) \leq WC^*$.
3. Now, let n be the node chosen for expansion. In this case $f_W(n) \leq f_W(x) \leq WC^*$.
4. Now assume that goal is chosen for expansion,. Based on the above claim we get $f_W(goal) \leq WC^*$.
5. But, $f_W(goal) = g(goal)$. So, $g(goal) \leq WC^*$.

WA*

$$f(n) = g(n) + Wh(n)$$

- This table presents results of research of WA* for different values of W.
- We can clearly see that while a path length changes linearly, the number of expanded nodes grows exponentially
- There is a tradeoff.
  Small W – more time better solution
  Large W – less time worse solution
- Theorem → Given W, WA* will find a solution which is no more than W times the optimal solution

W	Number of nodes	Length of path
1	500 000 000	53
3	22 891	78.41
7	12 772	112.55
99	6 972	145.22

Focal Search

- Sort OPEN according to the f-value
- Keep a FOCAL list of nodes from OPEN
  - Each node n in FOCAL has
    - $Fmin \leq f(n) \leq W * Fmin$
- Then, expand a node from Focal via a secondary function.

Any node in FOCAL is guaranteed to be W-admissible

Focal Search

Algorithm 1: Focal Search: main procedure

```
1  focal-search(start state S)
2      OPEN ← {S};
3      FOCAL ← {S};
4      while FOCAL ≠ ∅ do
5          best ← ChooseNode(FOCAL)
6          Remove best from FOCAL and OPEN
7          if best is a goal then return best;
8          if f_min increased then FixFocal() ;
9          for n ∈ neighbors(best) do
10             Add n to OPEN
11             if f(n) ≤ B × f_min then add n to FOCAL ;
12         end
13     end
14 end
```

Examples of Focal Search

- 1) A*e: Choose to expand the node with minimal h
- 2) Dynamic Potential Search. Choose to expand a node with maximal

$$ud(n) = \frac{B \times f_{min} - g(n)}{h(n)}$$

-

- 3) Explicit Estimation Search (EES): has a sophisticated way of doing this.

Bounded Cost Search

Potential Search (PTS)

Choose to always expand the node n in open with maximal:

$$u(n) = \frac{C - g(n)}{h(n)}$$

Bounded Cost Search vs Bounded Suboptimal Search

- It turns out that any focal search can be turned to be a bounded cost search

	FOCAL	EES	PS	WA*
BCS	$f \leq C$	BEES	**PS**	N/A
		⇑	⇓	
BSS	$f \leq B \times f_{min}$	EES	**DPS**	WA*

Table 1: The different algorithms

Anytime algorithms

- In general, anytime algorithms find a solution but continue and search for better and better solutions

- They can halt at ANYTIME.

The WA* family of algorithms

- Pure WA*
  - Determine a value for W.
  - Run WA*.
  - Halt once the first solution was found.

Iterative weighting A*

- Set W=large value
- While (you have time){
  - Run WA* with the current W until a solution is found.
  - Decrease W

  }
- Each iteration will run more time but will return a better solution
- Higher quality solutions while time passes.

Anytime weighted A* (AWA*)

- Set W=large value
- After a solution has been found, continue the search
- Higher quality solutions while time passes.

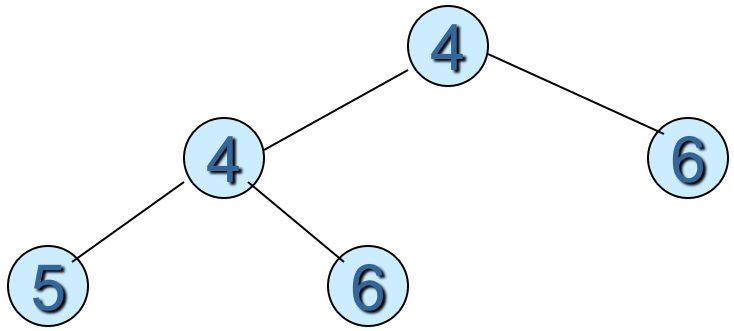
Space Complexity of A*

- **The main drawback of A* is its space complexity.**
- **Like all best-first search algorithms' it stores all the nodes it generates in either the Open list or the Closed list. Thus, its space complexity is the same as its time complexity, assuming that a node can be stored in a constant amount of space.**
- **On current computers, it will typically exhaust the available memory in a number of minutes .**
- **This algorithm is memory- limited.**

Iterative Deepening A* (IDA*)

- IDA* is to A* as DFID is to BFS
- How does it work:
  - A cost *threshold l* is set.
  - $f(n) = g(n) + h(n)$ is computed in each iteration.
  - If $f(n) <= l$ we expand the node.
  - Else the branch is pruned (we don't expand it).
  - If a goal node is reached with a cost lower then the goal it is returned.
  - Else if a whole iteration has ended without reaching the goal, then another iteration is begun with a greater cost threshold.
  - The new cost threshold is set to the minimum cost of al nodes that were pruned on the previous iteration.
  - The cost Threshold for the first Iteration is set to the cost of the initial state.

IDA* - Example

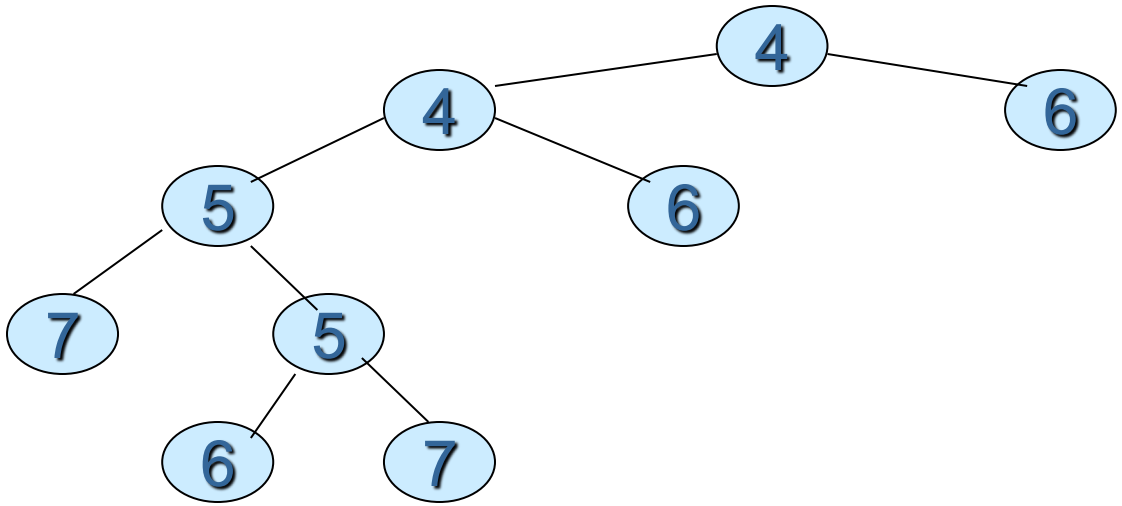


This is the First Iteration of IDA*.

In this example the initial I is 4 and every node with cost 4 is expanded.

We stop when we reach a cost larger than 4.

IDA* - Example

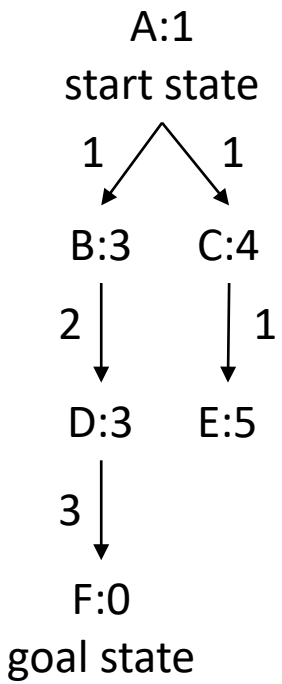


This is the Second Iteration of IDA*.

Limit I was 5 and every node with cost 5 or less was expanded.

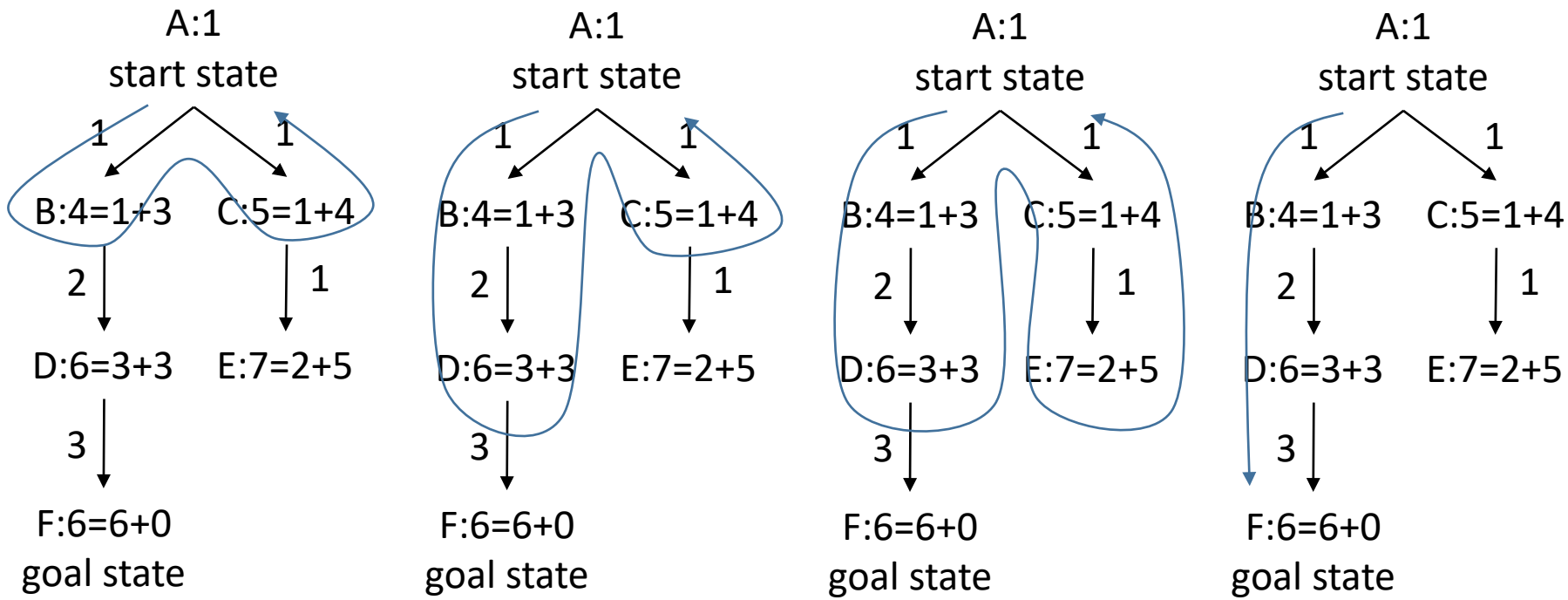
IDA* - Example

State space



Tree

depth-first search



Path: A B D F
(optimal)

Termination

- If a solution of finite cost exists, IDA* will find and return one.

Solution Optimality

- **Theorem 4.1** : ***In a graph where all edges have a minimum positive cost, and non-negative heuristic values that never overestimate actual cost, in which a finite-cost path exists to a goal state, IDA* will return an optimal path to a goal.***
- **Steps of proof (by induction):**
  - For the induction step, assume that at the iteration I , there is a node n on the frontier on an optimal solution path.
  - During iteration I+1, node n will be generated again, since the threshold for iteration I+1 is greater than for iteration I.

  In this way we can show that at the end of every iteration, there is at least one node n on the iteration that is on an optimal solution path.

Space Complexity

- The asymptotic space complexity is the maximum depth of the recursion stack .
- The optimal solution cost is *c*, and the minimum edge cost is *e*.
- The maximum length of any path with total cost less than or equal *c* is *c/e*.
- The maximum search depth is *c/e+1*.
- Since *e* is a constant, the asymptotic space complexity of IDA* is *O(c)*.

IDA* complexity eliminates the space constraint on A* in practice!

Time Complexity

- ***The last iteration***
  - The last iteration will expand all nodes connected to the root whose cost is less then or equal to *c*.
  - This is only true if the heuristic function is consistent.
  - In the worst case (when the heuristic function is not consistent) IDA* expands the same set of nodes as A* does.

Time Complexity

- ***The previous iterations***
  - The number of nodes generated by an iteration of IDA* with the cost threshold *x* is :

$$IDA(x) = \sum_{i=1}^{x} N(x)$$

  - If *N(x)* grows exponentially with *x*, with branching factor *b* then :

$$IDA(x) \,/\, IDA(x-1) \;=\; b$$

  - This means that in each iteration the number of nodes developed also grows exponentially with the branching factor *b*.

Time Complexity

- So according to what we saw the asymptotic time complexity of IDA* is the same as that of A*.
- We have seen that, like DFID, most of the work is done in the final iteration. The time is not harmed by all the iterations we apply although we go over the nodes in the former levels a couple of times.

Time Complexity

- The overhead of DFID over Breadth-First Search (i.e. the percentage of additional node expansions) is often smaller than the overhead of IDA* over A*.
- The reason is that there are often more nodes with the same g-value [= all of them get expanded for the first time during the same Depth-First Search of DFID] when all action costs are one than there are nodes with the same f-value [= all of them get expanded for the first time during the same Depth-First Search of IDA*] (especially when all action costs are different).

IDA* - Conclusion

- IDA* keeps the optimal solution of A*.
- IDA* solves the space constraint that A* has without any sacrifice to the asymptotic time complexity.
- IDA* may run even faster than A* (Why?).
- IDA* is much easier to implement then A* because it’s a DFS algorithm and no open and closed lists have to be kept.

limitations of IDA*

- When all the node costs are different:
  - IDA* will develop a different iteration for each node and in each iteration only one new node will be expanded.
  - On such a tree the time complexity of A* will be $O(b^d)$ but for IDA* $O(b^{2d})$.
  - If the asymptotic complexity of A* is $O(N)$ - IDA*'s complexity can get in the worst case to $O(N^2)$.

limitations of IDA*

- The problem space for IDA* must be a tree because :
  - if a certain node can be reached via multiple paths it will be represented by more than 1 node in the search tree.
  - A* can avoid the duplicate nodes by storing them in the memory but IDA* is a DFS (no memory) and thus it can not detect most of the duplicates.
  - This can increase the time complexity of IDA* compared to A*.
  - Thus, if there are many short cycles in the graph and there is no memory problem - choose A*.

Experiments with IDA*

- With the 8 tile puzzle, a good implementation of IDA* runs about 3 times faster per node generation then a good implementation of A* .
- IDA* with the Manhattan distance heuristic was the first algorithm to optimally solve random 15 puzzle instances. When trying to solve this problem with A* the memory quickly ran out because billions of nodes need to be expanded in this problem.
- IDA* was also used to solve the 24 puzzle (took a couple of weeks to solve).
- IDA* has also been used to solve the 3x3x3 Rubik's cube (This also takes about a week to solve).

ID as an Algorithm Schema

- ID can be viewed as an algorithm schema with different instantiations based on the cost function.
- There are several types :
  - DFID if *cost = depth*.
  - IDA* if *cost = g(n) + h(n)*
  - ID version of Uniform Cost Search

    if *cost = g(n)*

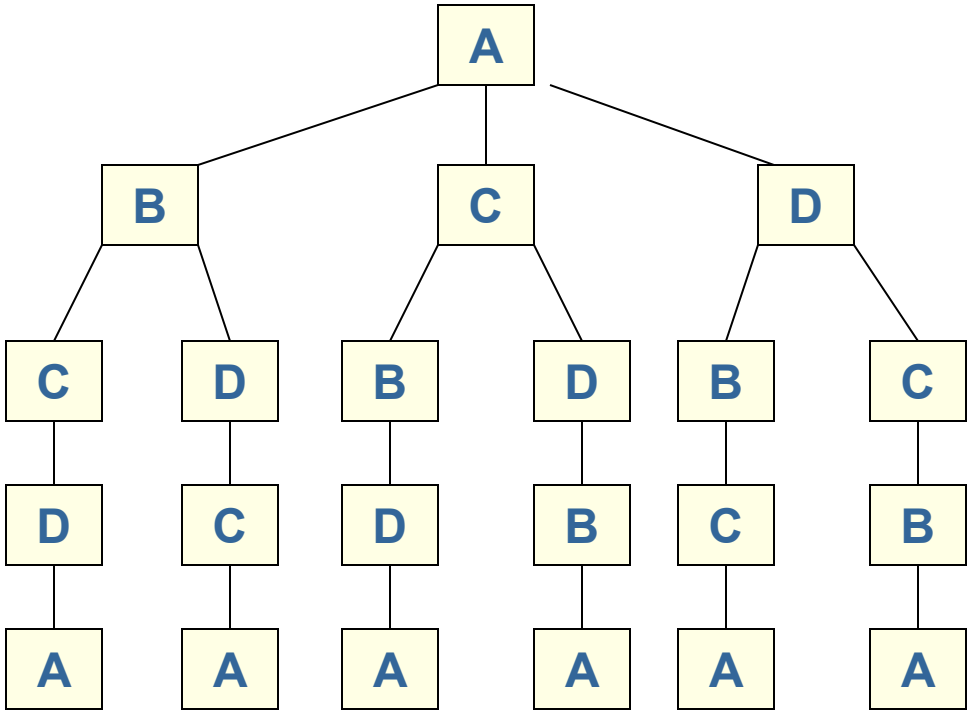
Depth First Branch and Bound

- IDA* isn’t so effective for all the problems.
- ***For Example***: TSP- the Traveling Salesman Problem deals with the possibilities of a salesman in visiting a finite number of cities in the cheapest way where every two cities have different price as their distance.

IDA* and TSP

- IDA* clearly doesn’t fit to deal with this problem in the best way because:
  - in the TSP problem there is a finite depth of the search tree, so Iterative Deepening is not useful in this case, because we know the depth of the solution and there is *n* use of searching different depths all the time when we know the correct depth from the beginning.

Example: Search Tree for TSP on Four Cities



DFBnB assumes a cost function that can be applied to a partial solution, and is a lower bound on the cost of all completions of that partial solution.

This is clearly the case in the TSP.

How DFBnB Works

- DFBnB works like a simple DFS but
  - when finding the first solution the cost of that solution is stored in α .
  - from this point on each time the cost of the new path exceeds or equals α, that branch is pruned and we continue checking the next one.
  - each time we reach a path that costs less than α we change α to this cost and update the best solution.
  - The search ends when we finish checking the whole tree.
- α starts with infinity in the beginning of the search.

When DFBnB is Used

- Depth-First Branch-and-Bound is often used when the optimal solution is required.
- It can also be applied in an infinite tree if there is a good upper bound available on the optimal solution depth or solution cost.
- It is also better than IDA* when IDA* wastes much time when there are a lot of different costs, for example in the TSP each cost is different from the other so the IDA* will only generate few nodes in each iteration thus costing us a lot of wasted time.

Improving DFBnB

- ***Node Ordering***
  - We use node ordering to find a low cost solution as quickly as possible and this results in greater pruning in the remainder of the search. For example in the TSP we can order the nodes in increasing order of the distances between child cities and parent cities.
- ***Heuristic evaluation function***
  - We can use a heuristic evaluation function

$$f(n) = g(n) + h(n)$$

in order to improve the DFBnB Algorithm.

Summary: Improving DFBnB

- Follows the same search path as DFS
  - Let's make that heuristic DFS
- Can freely choose order to put children on the stack
  - Could e.g. use a separate "ordering" heuristic h' that is NOT admissible
- To compute a better lower bound
  - Need to compute f value using an admissible heuristic h
- This combination is used a lot in practice
  - Examples include solving Sudoku
  - But also integrates some logical reasoning at each node

Solution Quality and Complexity

- **DFBnB returns an optimal solution.**
- The asymptotic complexity is *O(bd)*, since all the children of each node on the current path must be generated and stored to order them. O(bd) = O(d) since we assume b is constant.

The asymptotic complexity is ***O(d)***

Time complexity is ***O(b^d)***

- A detailed analysis of the average time complexity of DFBnB has to model both the heuristic function and efficiency of node ordering. The existing analyses on this algorithm are all based on abstract analytic models.

An Analytic Model and Surprising Anomaly

- It is a model that assumed a tree with uniform branching factor and depth, were the edges are assigned costs randomly from some distribution .
- Example:
  - The edges cost zero or one with probability 0.5 each. - It can be shown that if the expected number of zero-cost children of a node is greater than 1, DFBnB with node ordering will run in time that is polynomial in the search depth.

Truncated Branch and Bound

- DFBnB is an any-time algorithm.
- It’s uniqueness is that at any given time the algorithm can stop and return a solution (not an optimal one).
- If you don’t know how much time you have to solve the problem you can use DFBnB and when you stop it, it will return the best path found so far.

ID vs. DFBnB

- Similarities:
  - Both guarantee optimal solutions given lower bound heuristics.
  - Both are DFS and have linear space complexity
  - Both use global cost bound.

ID vs. DFBnB

- Main differences:
  - In ID the cost threshold is always a lower bound of the best solution and it increases during the iterations.
  - DFBnB starts with an upper bound cost threshold and decreases through the search.
  - Both expand more nodes than A*. ID expands only nodes with lower cost than C but a couple of times, and DFBnB expands also nodes with a cost larger than C.

Non Monotonic Cost Functions

- WA* has a cost function:

$$f(n) = g(n) + wh(n)$$

- To avoid it's large space complexity (same as A*) we would like to use Iterative Deepening with this cost function.

Non Monotonic Cost Functions

- For W>1 the function is non-monotonic.
- IDA* with a non monotonic cost function does not search the tree in best first order.