Problem and Search Spaces

Chess has approximately 10^{120} game paths. These positions comprise the *problem search space*. Typically, AI problems will have a very large space, too large to search or enumerate exhaustively.

Our approach is to search the space for a path to some goal. The problem may be formulated in terms of:

- States - describe the current state of the problem (or solution)
- Initial state
- Successor function – the "moves" we can make from one state to another
- Goal test – some way to know if we reached the goal
- Path cost – cost for each step from the initial state to the goal state

Consider the state space for the Cannibals and Missionaries problem. You have 3 cannibals and 3 missionaries, all who have agreed they want to get to the other side of the river. You don't want to ever have more cannibals than missionaries on one side alone or the cannibals may eat the missionaries. The boat available only holds two people.

State: number of cannibals and missionaries on each side of the river, location of boat

Successor Function, or Move-Generator:

For side the boat is on:

if c>=2 send 2 cannibals

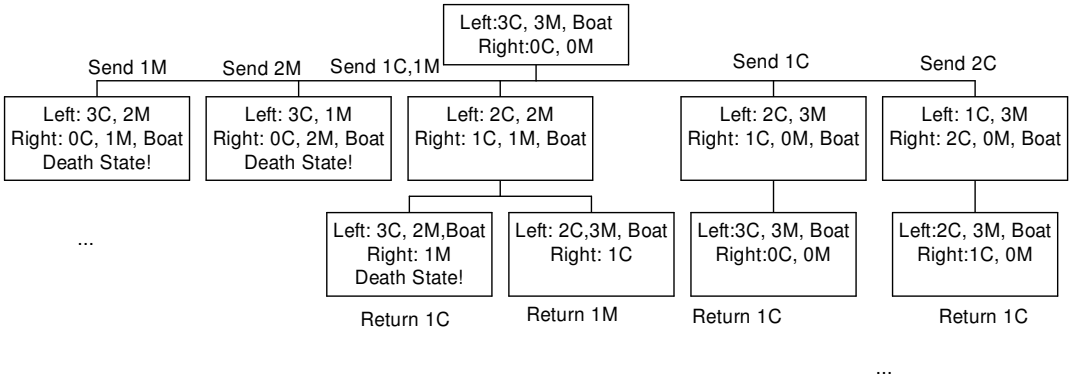
if c>=1 send 1 cannibals

if c>=1 and m>=1 send 1 cannibal and 1 missionary

if m>=1 send 1 missionary

if m>=2 send 2 missionaries

Searching for a state: apply all applicable moves to the current state to generate a new state, and repeat.



Note the space grows exponentially; difficult and almost impossible to enumerate for large spaces, at least to find a goal state (in this case, the goal state is when all 6 people are on the other side of the river).

One solution:

Initial State:	3m3cb	0m0c
Send 2c:	3m1c	0m2cb
Return 1c	3m2cb	0m1c
Send 2c	3m0c	0m3cb
Return 1c	3m1cb	0m2c
Send 2m	1m1c	2m2cb
Return 1m1c	2m2cb	1m1c
Send 2m	0m2c	3m1cb
Return 1c	0m3cb	3m0c
Send 2c	0m1c	3m2cb
Return 1c	0m2cb	3m1c
Send 2c	0m0c	3m3cb

Note that this problem is somewhat difficult for people to solve because it involved "backwards" moves where we take away people from the other side of the river. People tend to think in terms of heuristics, which are essentially rules of thumb for progress, and these types of move violate the heuristic of "the more people on the other side of the river, the better". We will also have the problem of returning back to states we've already been in, which can raise the potential for endless loops in searching for a solution.

Searching Problem Space – Uninformed Search Algorithms

The problem space is a theoretical construct; the entire space "exists". However, only a portion of this space need (or can) be represented in the computer. The rest will need to be generated. Question: best way to search/generate the nodes in the problem space?

Breadth First Search (BFS)

```
function BFS(state)
        node-list =Apply-Moves(state)          ; return all valid "moves" from the current state
        while (node-list != empty)
                node=node-list.Remove-Front()
                if Goal-Test(node) then return true
                node-list.Enqueue-At-End(Apply-Moves(node))
```

By expanding new states at the end of the node-list, BFS systematically explores all children of a given state first. The nodes on the list but not yet expanded are called the Fringe. (Show order of node generation on Missionary/Cannibal problem). This is different from "normal" BFS that is taught in algorithms classes because here we are not attempting to explore all states, but we are attempting to find a particular state. Consequently, not all states will be explicitly enumerated.

Advantages:

Will never get trapped exploring a blind alley.

Guaranteed to find solution, if it exists. Good if solution is only a few steps away.

Disadvantages:

Time and memory may be poor.

Call b the branching factor - the number of paths possible to take at a node. (show binary tree, branching factor=2). At depth=0, nodes=2^0=1. At depth=1, nodes=2^1=2. At depth=2, nodes=2^2=4… at depth=10 nodes=2^{10}=1024. In general, nodes = b^d. This is fine for b=2,but consider a game like GO where b=50. Then, if we want to look ahead to a depth of 6, we need to generate 50^6=15600000000 states. If each state required 100 bytes, this would occupy 1.56 terabytes (1560 gigabytes). Too large to do and even store in memory, let alone most hard drives!

BFS requires $O(b^d)$ space to store the results. Exponential time to generate! Not feasible for large problems, these could take years to compute.

Depth-First Search (DFS)

```
function DFS(state)
    stack = Apply-Moves (state)  // Push all the moves from the state onto a stack
    while (stack != empty)
        node= stack.Pop()
        if Goal-Test(node) then return true
        stack.Push(Apply-Moves (node))
    return false
```

DFS always expands one of the nodes at the deepest level of the tree, and only when a dead end is hit, does search go back to expand nodes at shallower levels. (Show example on Missionary/Cannibals problem).

Advantages:

Requires space only in storing the path traversed. For maximum depth m, requires bm nodes to be expanded, as opposed to b^d for BFS.

Might get lucky and find solution right away.

Disadvantages:

Still has $O(b^m)$ time complexity.

May get stuck following an unfruitful path, or never find a solution if stuck in loop.

Simple variation: Limit the depth that we can search (this is Depth-Limited DFS).

Iterative Deepening Search

Later we will combine both DFS and BFS together to get something called A*. For now we will discuss DFID, Depth-First Iterative Deepening.

Tries to combine DFS and BFS by trying all possible depths, starting with a very short depth. Algorithm is to simply perform a DFS to depth 0, then to depth 1, then to depth 2, etc. This is similar to BFS, but uses the DFS algorithm instead. (Show example on Missionary/Cannibal problem).

Advantages:

Optimal and complete; will find a solution if one exists.

Same memory requirements as DFS.

Disadvantages:

Some states will be expanded multiple times, especially initial moves and moves up in the search tree. BUT the computation in an exponential search tree is dominated by the number of leaves; i.e. $O(b^{d+1})$ dominates $O(b^d)$. So this does not add significantly to runtime.

Ex: Tree with b=10, d=5. Number nodes = 1 + 10 + 100 + 1,000 + 10,000 + 100,000

Using DFID: 1st pass we examine 1 node

2nd pass we examine 1 + 10 = 11 nodes

3d pass we examine 1 + 10 + 100 = 111 nodes

4th pass we examine 1 + 10 + 100 + 1000 = 1111 nodes

5th pass we examine 1 + 10 + 100 +1000 + 10000 = 11,111 nodes

6th pass we examine 1+10 +100+ 1000+10000+100000 = 111,111 nodes

Total nodes = 123,456. All previous passes contribute little, dominated by the leaves on the last level of computation.

In general, DFID is the preferred search method when there is a large search space and the depth of the solution is not known.

Summary of BFS, DFS, DFID

b=branch factor, d=depth of solution, m=maximum depth of search tree

Criteria	**BFS**	**DFS**	**DFID**
Time	b^d	b^d	b^d
Space	b^d	bm	bd
Optimal? (Find best solution)	Yes	No	Yes
Complete? (Guaranteed to find solution if exists)	Yes	No	Yes

Duplicate States

So far we've been ignoring the possibility of making a cycle and returning to repeated states. In general, we don't want to return to a duplicate state. Typical solution is to either ignore the problem or use a hash table to store visited states, and then check for them before visiting. O(1) time, although O(s) storage space required (using a good hash function).

Bi-Directional Search:

Not used too often; idea is to search simultaneously both forward from the initial state, and backward from the goal, and stop when the two searches meet in the middle.

Ex: With Missionaries and Cannibals, search backwards from everyone on the right side of the river, along with searching everyone from the left.

If solution found at depth d, the solution will be found in $O(2b^{d/2}) = O(b^{d/2})$ steps since the algorithm will only have to search halfway. Can make a big difference!
If b=10 and d=6, BFS generates 1,111,111 nodes, and bidirectional search only 2,222 nodes.

Problem: One search needs to retain in memory all nodes examined so it is known if they meet. Requires $O(b^{d/2})$ memory to store. Also, you may not know the goal state or there may be multiple goal states (e.g. chess).

Constraints

Constraints are one way to limit the search space. In the missionaries/cannibals problem, we could instead view the problem as having the constraint that we can't make a move that results in more cannibals than missionaries, rather than create the death state. As a result, there are fewer moves that can be generated. Constraints are inherent in many search problems (e.g., crytpoarithmetic).

Example: CryptoArithmetic

```
                S       E       N       D
+               M       O       R       E
-----------------------------------------------------
=       M       O       N       E       Y
```

Here, no two letters have the same value. The sums of the digits must be as shown in the problem.

We could use normal DFS, BFS, or DFID. In this case, we'd assign a value to each letter until we find a solution that works. But the constraints of arithmetic help us solve the problem faster.

Steps:

1. Propagate constraints using the rules of the problem domain. In this case, the rules of arithmetic.
2. Guess value for some variable
3. Repeat process by propagating constraints based on the guess
4. If we reach a dead end, back up and take a different guess

How the constraints help:

Let's rewrite the problem with carry's:

```
                C3      C2      C1
                S       E       N       D
+               M       O       R       E
-----------------------------------------------------
=       M       O       N       E       Y
```

We know that M=1 because two single digits + a carry can't be more than 19

If M=1, then S=8 or 9, has to be big enough to generate the carry

If M=1 and S=8 or 9, then the sum of M+S+C3 will be either 10 or 11. This means that O=0 or 1. But since no two letters can be the same, and M is already 1, then O=0.

If O=0, then then N=E+C2. This means that N=E or N=E+1. But since N can't be E due to the constraint that no two letters be the same, then N=E+1.

If N=E+1, then C2=1

If C2=1, then N+R+C1>=10

Without doing any search we've already identified several variables!

When we can't find any more constraints, we just need to guess a value for a variable.

Guess that E=2.

Then N=3

R=8 or 9

Guess that C1=1

Etc… until we find some set of variables that satisfy the equation. At this point we're doing a normal DFS search, propagating constraints each time, to narrow what moves we can make next.

It's left as an exercise to the reader to find the rest of the numbers that satisfy the equation.

Generate and Test:

Algorithm:

1) Generate possible solution. This might be a entire path from the start state to the end state (as in Cannibals/Missionaries) or it might be a single state (trying to find the right values to optimize some function, say, the cryptarithmetic problem (SEND+MORE=MONEY)).
2) Test to see if solution works.
3) If found quit, otherwise return to step 1.

Can be done systematically as in DFS. May also be performed randomly to perform a more random search ; this has been called the British Museum Algorithm, named after visitors that randomly wander through the museum.

The algorithm is typically implemented through DFS with backtracking.

Best-First Search Methods

So called Best-First Search methods should really be named "Possibly Best-First Search" since if we actually made the best move possible at every move, we wouldn't be searching at all. We'd just find a direct path to the goal.

Most searches are based on heuristics, which makes them **informed** search techniques (as opposed to the previous techniques, which were **uninformed**):

Heuristic h(n) = estimated cost of the cheapest path from the state at node n to a goal state
i.e. a guess as to the "goodness" of a particular state. A heuristic only applies to a particular state.

History: Heuristic comes from the Greek word "heuriskein" that means "to find". Archimedes is supposed to have said "Heureka", for 'I have found it!' not "Eureka".
Sample heuristics:

Consider navigation through Anchorage in a car. You want to get to UAA from the convention center. There are buildings in the way, and also one way streets, so the problem is not trivial. A heuristic for how close you are to the goal might just be the air distance from your current location to the goal.

A second heuristic might be the Manhattan Distance, which is the sum of the horizontal and vertical distances to the goal.

Manhattan distance has been extensively studied with solving 8-puzzles.
Initial state:

5	4	
6	1	8
7	3	2

Goal state:

1	2	3
4	5	6
7	8	

Heuristic function h=sum of Manhattan distance for each tile

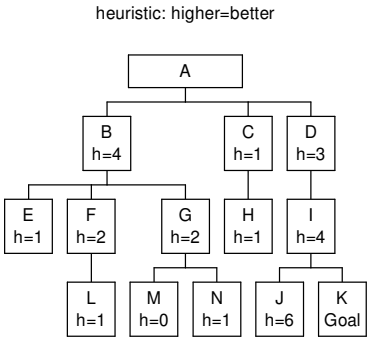
Hill Climbing

If you view the heuristic function applied to child states, then you can "climb the hill" that results:

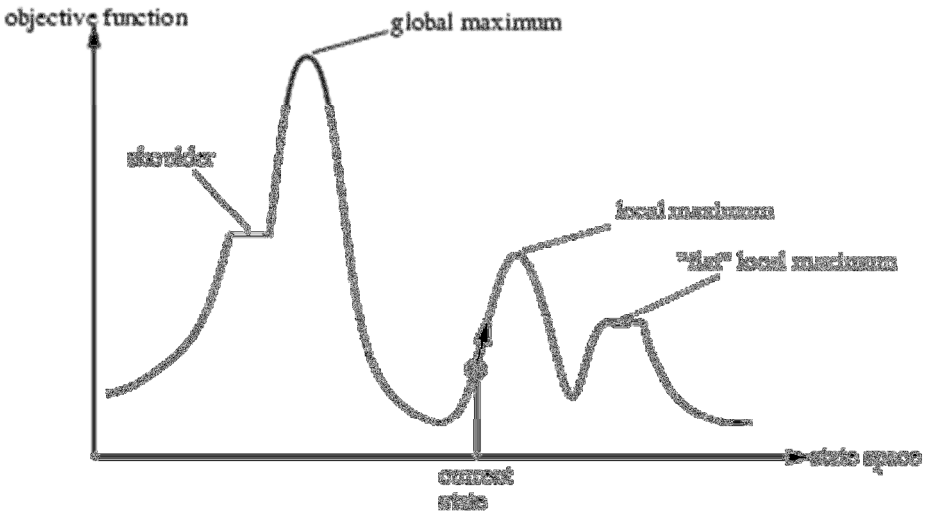
1) Evaluate initial state. If goal, then quit.
2) Loop until solution is found or no new operators left to apply to current state:
    3) Apply new operator to current state
    4) Evaluate new state. If goal, then quit. If not a goal but better than current state, then make it the current state. If worse that current state, continue looping.

Hill Climbing is often referred to as a “greedy” strategy since it follows whatever path looks best at the time. It also relies on an accurate heuristic to determine which operator to select.
The simple algorithm is most commonly used as *Steepest Ascent Hill Climbing*.
That is, apply all operators to the current state, and then pick the best resulting state as the new state. It has been called “"Like climbing Everest in thick fog with amnesia."

Example (show hill climbing for different goal=M and goal=K):



Problems:



Solutions: Backtrack to old solutions, then pick new direction
Make a random jump
Genetic Algorithms
Apply other rules before making tests

Simulated Annealing

Method intended to reduce risk of local maxima and ridges. Based upon annealing of materials, such as steel. In annealing, a material is heated to a high temperature and then cooled according to an annealing scheduling. Quick/slow annealing results in material that may be brittle, smooth, etc. Typically the material is cooled to produce a minimal energy state - i.e. energy-min is the goal.

Allows for transition from low energy to high energy by:

$$p = e^{-\Delta E/kT}$$

i.e. the probability of an uphill move decreases as the temperature decreases. This is analogous to initially allowing uphill moves, but as we zero in on the goal state, allow fewer and fewer uphill moves until a minimum is attained.

Algorithm:

1) If initial state is goal, quit
2) Set Best-So-Far to current state
3) Initialize T to annealing schedule (rate at which to cool)
4) Loop until solution is found or no new operators to apply
5) Apply new operator to current state
6) Compute change in energy; h(current) - h(new state)
7) If new state is goal, quit
8) If h(new state) > h (best-so-far) set best-so-far to new state and make it the current state
9) If not better than current state, make it the current state with probability $p = e^{-\Delta E/kT}$; this probability decreases as T is lowered. Right units must be found, typically k is a unit conversion factor.

Things to fiddle with to improve performance based on problem: Annealing schedule, mapping values, backtracking, etc.

Used to be a hot algorithm in AI, would help solve many search problems, but is now not used very widely.

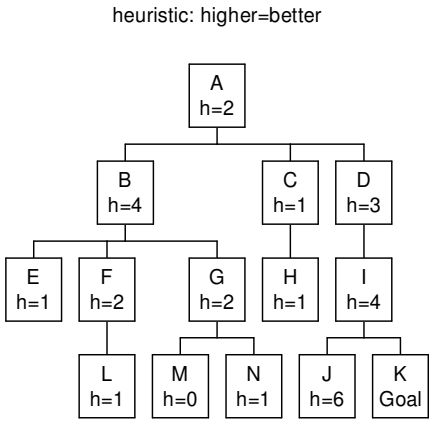
Beam Search

The idea in beam search is to keep track of k states rather than just one. Start with k randomly generated states. Then, at each iteration, all the successors of all k states are generated. If any one is a goal state, stop; else select the k best successors from the complete list and repeat.

Best-First Search

There is an actual algorithm named “Best-First Search” which is essentially a general search using a heuristic function (hopefully this specific algorithm is not to be confused with the general class of best-first search algorithms). Unlike DFS or BFS, Best-First Search remembers all previous moves and takes the next move that looks best, according to the heuristic function.

```
Best-First-Search(start)
    open ← start
    closed ← empty
    while open ≠ empty do
        x ← remove leftmost state from open
        if Goal(x) return x
        succ ← Production-Rules(x)
        for each child c of succ do
            if c is not on open or closed
                assign c a heuristic value
                add c to open
            if c is on open
                if c was reached with a shorter cost
                    assign shorter path to c
            if c is on closed
                if c was reached with a shorter cost
                    assign shorter path to c
                    remove from closed
                    add c to open
        Add x to closed
        Reorder states on open by heuristic (leftmost node=best)
    Return failure
```



Show example on graph below. Cost of move=1
For now we'll skip the case where we might have reached a node from a previous path. The cost in this case for each move is 1.

Start at A:

Open = [(A, 2)]
Generator successors for A: [(B, 4) (C, 1) (D, 3)]

Open = [(B,4) (D,3) (C,1)]
Closed = [(A, 2)]
Generate successors for B: [(E, 1) (F, 2) (G,2)]

Open = [(D, 3) (F, 2) (G, 2) (C, 1) (E,1)]
Closed = [(B, 4) (A, 2)]
Generate successors for D: [(I, 4)]

Open = [(I,4) (F, 2) (G, 2) (C, 1) (E,1)]
Closed = [(B, 4) (D, 3) (A, 2)]
Generate successors for I: [(J, 6) (K, Goal)]

Open = [(K,Goal) (J, 6) (F, 2) (G, 2) (C, 1) (E,1)]
Closed = [(B, 4) (D, 3) (I,4) (A, 2)]
K=Goal, quit

We didn't come across the case where a succ node was in open or closed. We'll look at this case next in A* search.

A Search: Minimize Total Cost*

h(n) attempts to address a way to minimize the cost to the goal state. Let's define:

g(n) as the cost of the path so far.

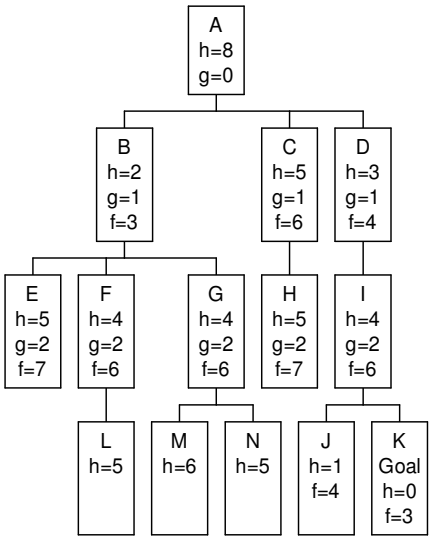
A natural method to approach search is to minimize the total cost: f(n)=g(n)+h(n). Consequently, f(n) should give us the estimated cost of the cheapest solution throughout the problem space. Note that if h(n)=0 then we essentially have BFS. Also note that here we are MINIMIZING the heuristic, not maximizing it. That is, a small heuristic value is better than a large one.

Algorithm:

Run Best-First Search with a heuristic of **f(n) = g(n) + h(n)**. Small values are better than larger ones.

Example: Run on the same graph as before, but with heuristic values flipped so that smaller is better. The cost g(n) of making a move is 1.

heuristic: lower=better



Open = [(A, 8)]
Generator succesors for A: [(B, 3) (C, 6) (D, 4)]

Open = [(B,3) (D,4) (C,6)]
Generate successors for B: [(E, 7) (F, 6) (G,6)]

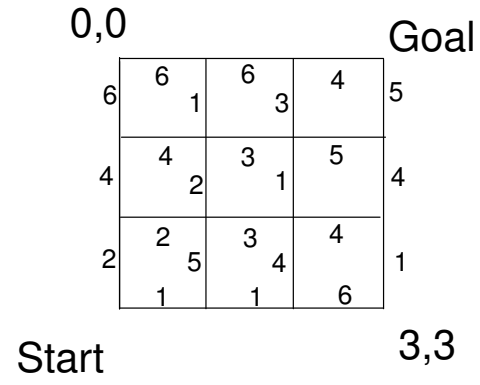
Open = [(D, 4) (C, 6) (F, 6) (G, 6) (E, 7)]
Closed = [(B, 3) (A, 8)]
Generate successors for D: [(I, 6)]

Open = [(I,6) (C, 6) (F, 6) (G, 6) (E, 7)]
Closed = [(B, 3) (D, 4) (A,8)]
Generate successors for C (tie, we could have done I,C,F, or G): [(H,7)]

Open = [(I,6)(F, 6) (G, 6) (E, 7) (H,7)]
Closed = [(B, 4) (D, 3) (C,6) (A,8)]
Generator successors for I (tie) : [(J, 4) (K, 3)]

Open = [(K,3) (J,4) (F, 6) (G, 6) (E, 7) (H,7)]
Closed = [(B, 4) (D, 3) (C,6) (I,6) (A,8)]
K=Goal, Quit

More complicated example: Finding shortest path to goal. Use Manhattan distance as the heuristic, where the Manhattan distance is just the number of edges to the goal.



Start in lower left corner.

Open = [(0,3 f=0+6)]
Generate succ for (0,3): [(0,2 f=2+5), (1,3 f=1+5)]

Open = [(0,2 f=2+5), (1,3 f=1+5)]
Closed = [(0,3 g=0)]
Generate succ for (1,3) : [(1,2 f=6+4), (2,3 f=2+4), (0,3 f=2+6)]
Since (0,3) is on closed but g=2 while old g=0, we don't update (and we never will for the start node!)

Open = [(2,3 f=6) (0,2 f=7) (1,2 f=10)]
Closed = [(1,3 g=1) (0,3 g=0)]
Generate succ for (2,3) : [(1,3 f=3+5) (2,2 f=6+3) (3,3 f=8+3)]
Since (1,3) is on closed but g=3 while old g=1, we don't update

Open = [(0,2 f=7) (2,2 f=9) (1,2 f=10) (3,3 f=11)]
Closed = [(1,3 g=1) (2,3 g=2) (0,3 g=0)]
Generate succ for (0,2) : [(0,1 f=6+4) (1,2 f=4+4)]
(1,2) is on Open. We found a shorter path, so update it.

Open = [(1,2 f=8) (2,2 f=9) (0,1 f=10) (3,3 f=11)]
Closed = [(1,3 g=1) (2,3 g=2) (0,2 g=2)]
Generate succ for (1,2) : [(0,2 f=6+5), (1,1 f=6+3) (2,2 f=7+3)]
Ignore (0,2) since g > old path
Ignore (2,2) since g > old path

Open = [(2,2 f=9) (1,1 f=9) (0,1 f=10) (3,3 f=11)]
Closed = [(1,2 g=4), (1,3 g=1) (2,3 g=2) (0,2 g=2)]
Generate succ for (2,2) : (2, 1 f=7+2) (1,2 f=9+4) (3,2 f=10+2) (2,3 f=10+4)]
Ignore (1,2) since g > old path
Ignore (2,3) since g > old path

Open = [(2, 1 f=9) (1,1 f=9) (0,1 f=10) (3,3 f=11)]
Closed = [(1,2 g=4), (1,3 g=1) (2,3 g=2) (0,2 g=2) (2,2 g=6)]
Generate succ for (2,1) : …. Etc. continue until reach the goal

Combines properties of breadth first search, but directs only along "good" paths.

About the heuristic, h:

If h is a perfect estimation of the actual distance to the goal, A* will converge immediately to the goal along the best path without doing any search.
What about the properties of finding the optimal solutions and ensuring that we will find a solution if one exists?

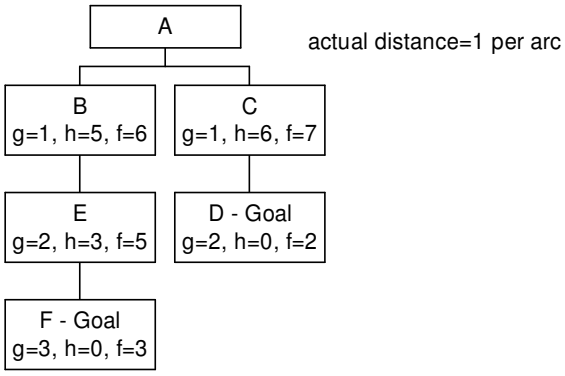
If h is possibly an *overestimate* of the distance to the goal, we could possibly be fooled into taking the wrong path and finding a non-optimal goal state. Consequently, we need the property that h always return an *underestimate* of the actual distance to the goal to have the guarantee that we find the optimal solution. This is called an *admissible* heuristic:

An admissible heuristic never overestimates the cost to reach the goal.

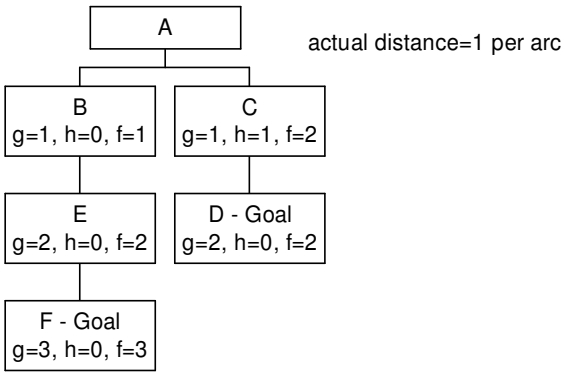
Note that the trivial admissible heuristic, h(n)=0, resorts to BFS which is optimal and complete.

If h(n) is admissible, then A* will find the optimal solution, and is complete on locally finite graphs (graphs with finite branching factor). The completeness follows from the *monotonicity* of h(n); i.e. h(b) <= h(a) + cost(a,b) if b is a successor of a. The h(n)-cost will generally decrease as search along the path progresses (and f(n) increase). Almost all admissible heuristics result in monotonically increasing f(n) functions. If not, it's possible to force an admissible heuristic to give monotonic results.

Here is an example illustrating admissible heuristics and monotonicity:
Consider the following tree:



Above is a tree with an inadmissible heuristic. Might find non-optimal answer.
A* search will go from A to B to E to F.



This is an admissible heuristic. Search does go the wrong path, to B first, and then it may go to E, but it will proceed to C and then D before going to F.

The property of *monotonicity* says that f will be *nondecreasing* as we move to child states if the heuristic is admissible. This is true in the second example; notice how f=1, 2, 3 or f=2,2 as we move along down the tree.

Computation Time: Depends on h(n), could result in same computation time as BFS ($O(b^d)$). The main problem is space: if the heuristic is not very good we could also generate exponential space to remember what nodes are on the OPEN list.

IDA - Iterative Deepening A**

Will only mention here; just as we did DFID, we can also perform Iterative Deepening on A* to remove the memory constraints. The idea is to perform a DFS using the f-cost as the limit rather than the depth-cost. Once again we will revisit some nodes, but the complexity remains the same, and we only use O(bd) storage for a solution at depth d to store each node along the solution path.