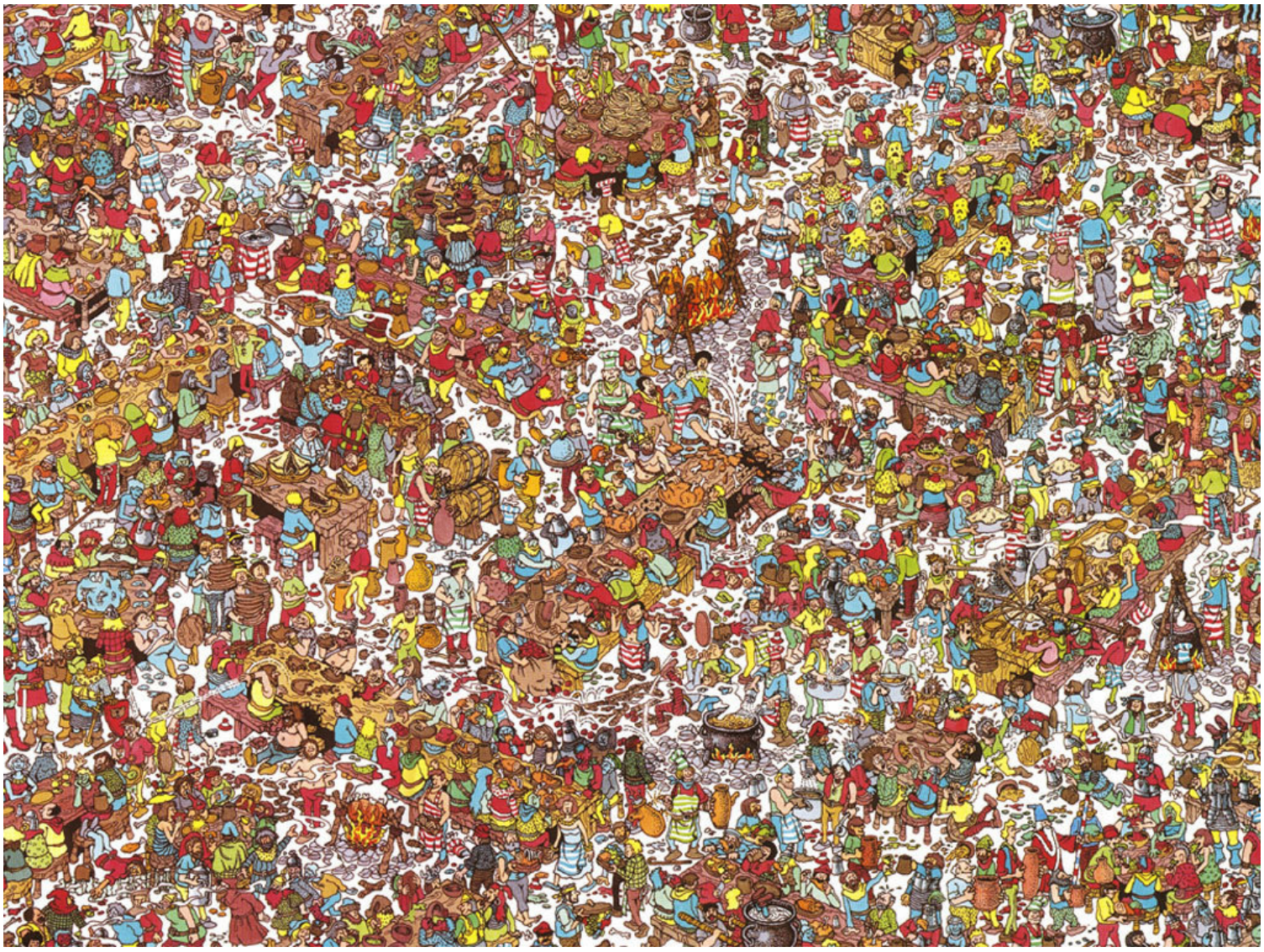
Intro to States & Search

We perform search in our daily lives all the time...

We have **problems** that have a certain issue and resolution, and generally some **strategies** to use **actions** to resolve those problems.



The familar Where's Waldo is a classic visual search paradigm where we start in some **state** of not knowing where Waldo is, and perform serial search to try to find Waldo.

á **Search** is exactly what it sounds like: the process of moving from a starting state of a problem to a goal state of that problem.

- **States** are a given instantiation of a search problem's variables of interest (possibly an incomplete instantiation).

Search is therefore problem dependent, such that we need some constraints upon the domain of what we're searching for.

In the case of Where's Waldo, it might be difficult to quantify some notion of states (perhaps we start in the "haven't found Waldo" state and want to move to the "found Waldo" one)

- A **search domain** is the set of all possible instantiation of variables of interest, i.e. all possible states, for a given problem.

If we didn't have a search domain, then the concept of search is intractable for both humans and computers.

For Where's Waldo, the search domain is simply a given picture that we know has Waldo somewhere inside of it; we don't go looking in a dictionary for Waldo (the dictionary is not in the search domain!), unless we're really bad at the game...

So, to perform search, we have a couple of rules about states:

- An **initial state** is the state that you begin your search operation within.

- A **goal state** is a state that satisfies your search criteria, i.e., that completes your search from an initial to a goal state.

- **Actions** are the legal manipulations of a current state that either (1) return us to the same state, or (2) transition to a new state.

Actions can also have costs associated with them, e.g., the action of driving through a 10 mile road might have a higher cost than moving through a 1 mile road.

- Therefore, the **score** of a particular goal state might be a sum of the costs encountered on the path of actions from the initial to goal state.

So, in summary, search is the process of moving from some initial state to some goal state through successive applications of legal actions.

Classical Search

"I found Beethoven!" you exclaimed excitedly.

Not that kind of classical search!

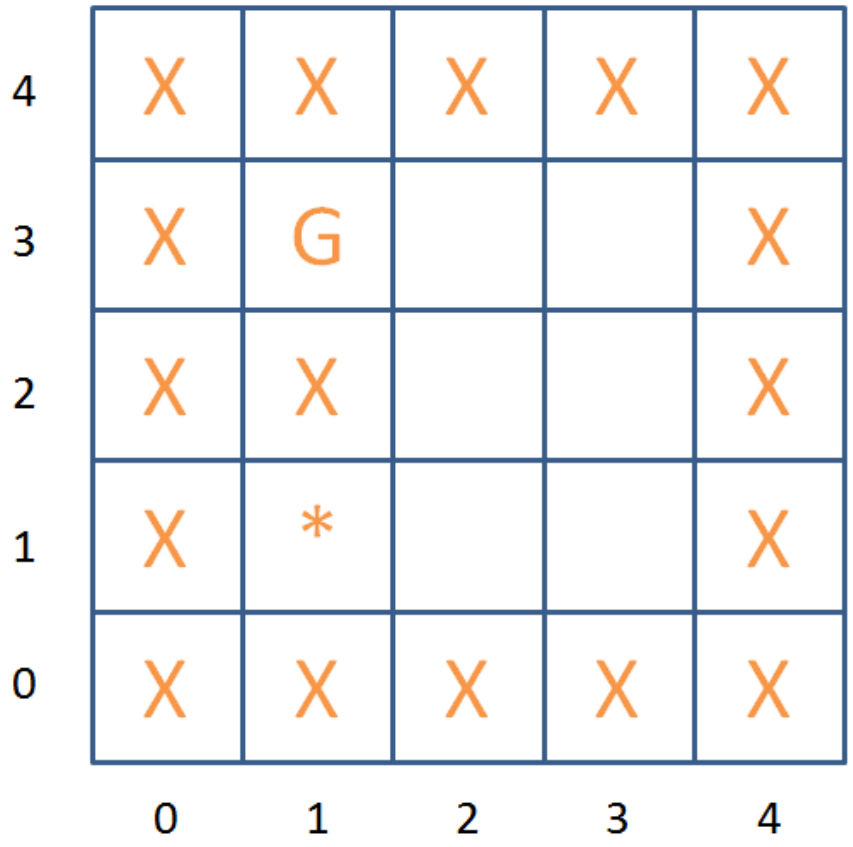
> **Classical search** is the systematic, goal / test oriented search of moving from an initial state to a goal state.

Motivating Example

Let's talk about the problem of maze pathfinding for our motivating example!

Here's the problem we're going to discuss:

Legend:
X = Barrier (impassable)
* = Player
G = Goal



Our search goal will be to determine if a path exists in the grid from the player's position to the goal state.

- What will a state look like in this problem?

- What will the initial state look like in our example of this problem?

- What will the goal state look like in our example of this problem?

- What will actions look like in this problem?

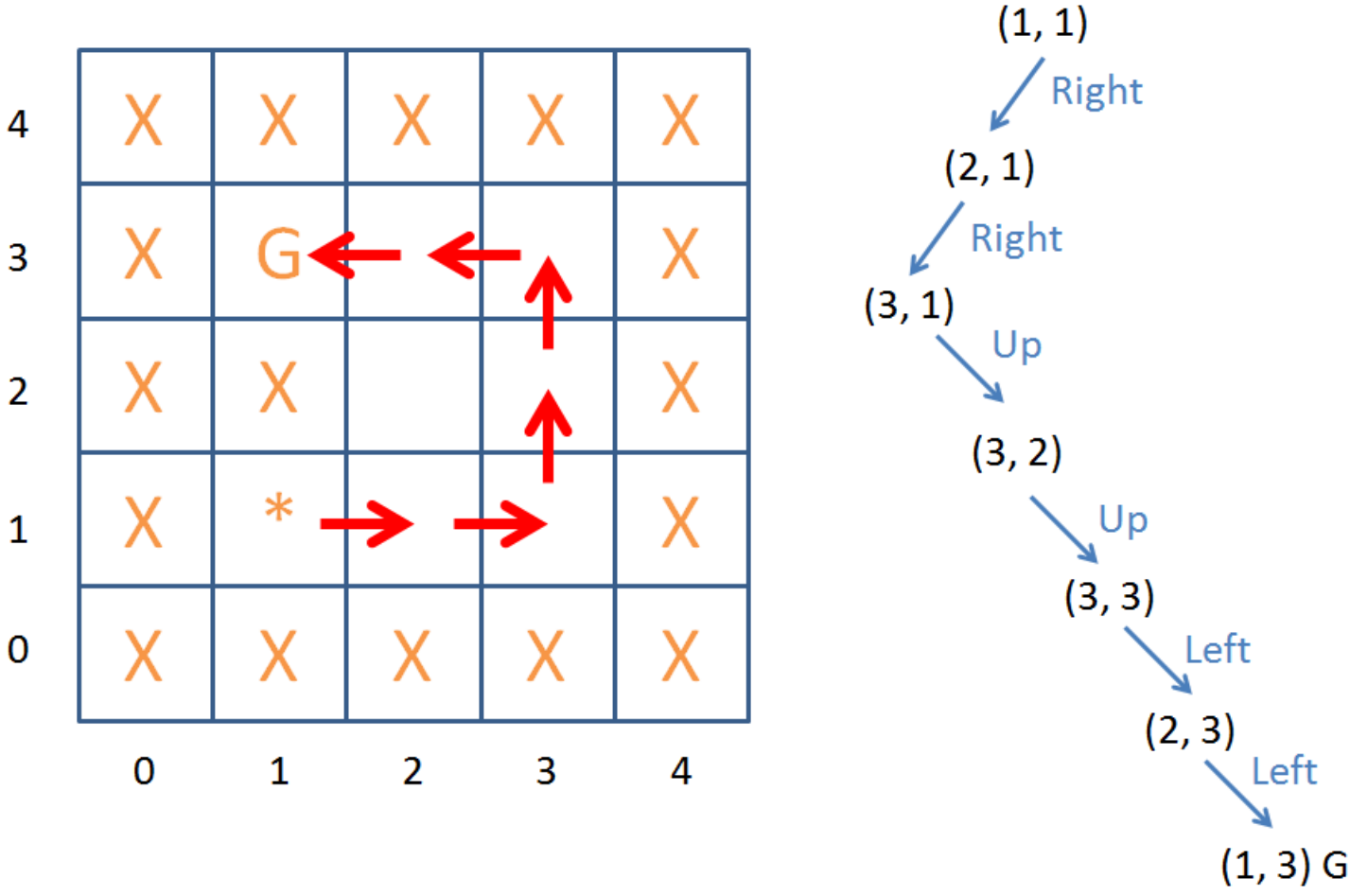
Building a Search Tree

So now that we have our concept of states and transitions in the maze pathfinding example, how do we go about searching for the answer?

In the **classical search** methodology, we want a systematic way of exploring states so that we can find an answer; to do this, we build a search tree:

> á A **search tree** is a tree with the initial state at the root, transitions along the edges, and nodes corresponding to states that can be reached from one action applied from the parent.

So, here is a *non-systematic* (search strategy omitted) search path that corresponds to a successfully discovered goal from our previous problem:



I say it's non-systematic because we didn't really apply any search-strategy; I just kinda picked a path that I saw was available!

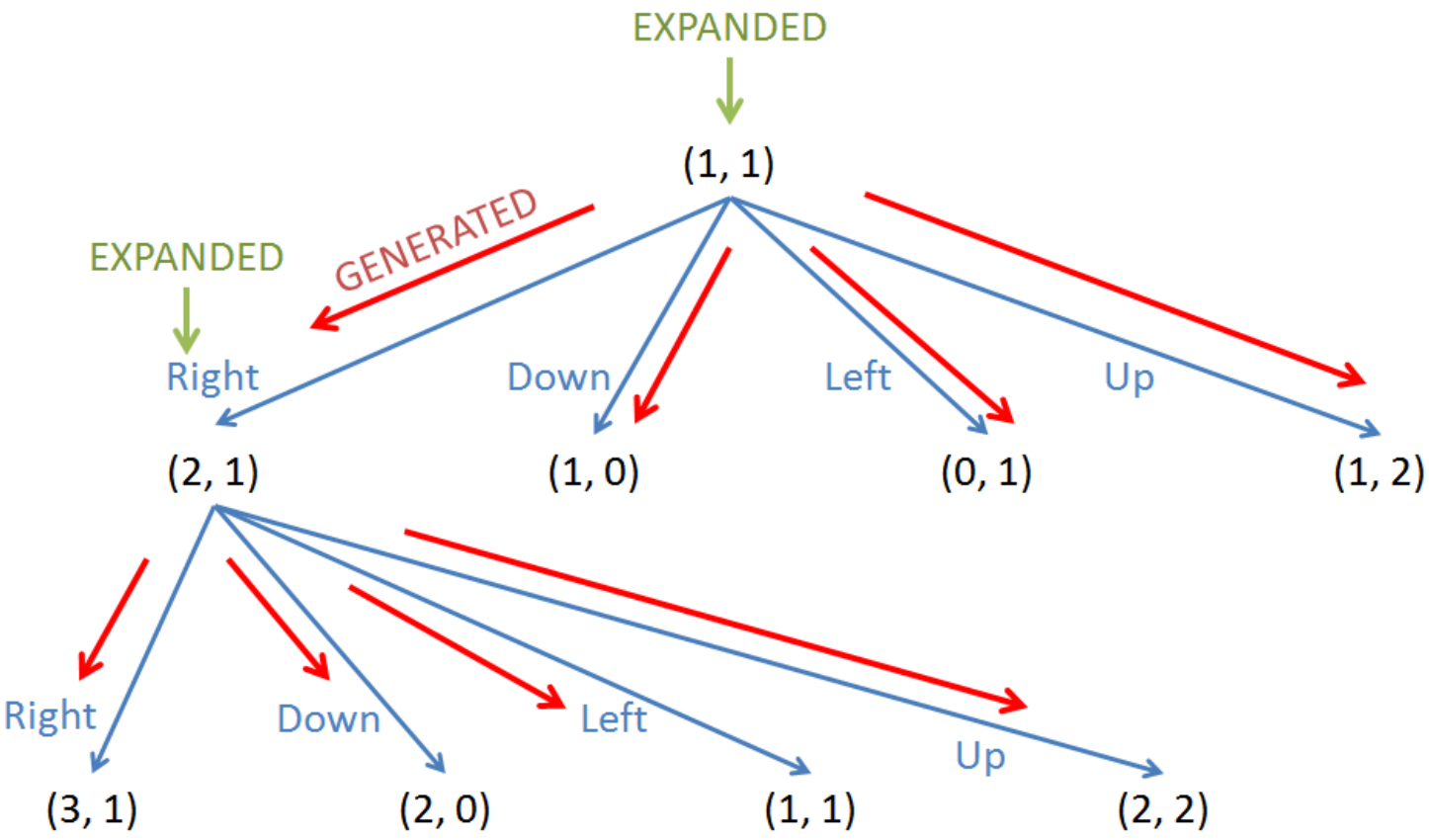
á A **search strategy** is a systematic means of exploring a search tree, and decides which state we will expand next in the course of our search.

In this conception, whenever we expand a node, we generate all of the possible moves from that node and then use our search strategy to choose which of the generated nodes to expand next.

á **Expanding** a node is akin to visiting it on a search traversal, and generating all plausible child nodes.

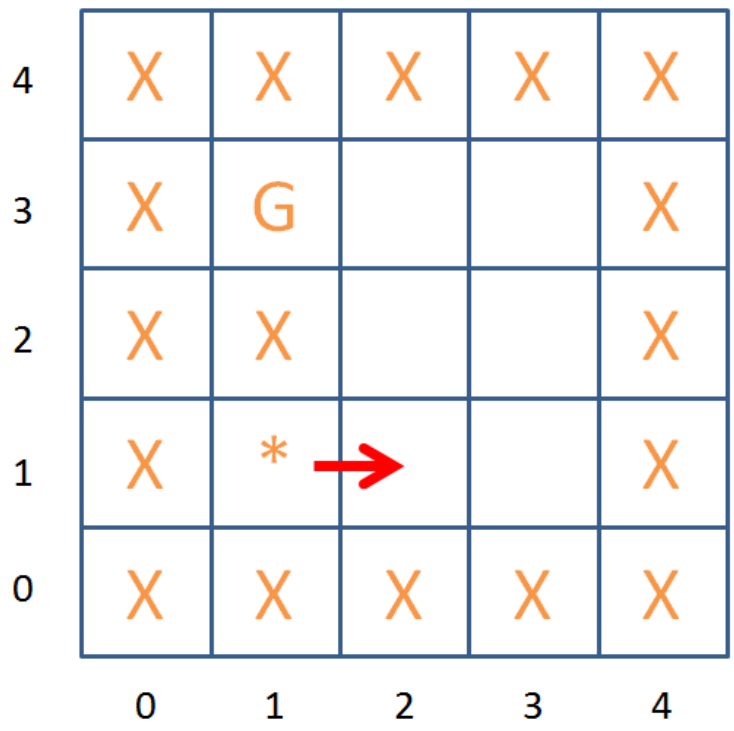
á We **generate** nodes by first expanding a parent node, and then generating all child nodes that could be reached from the parent by taking a legal action.

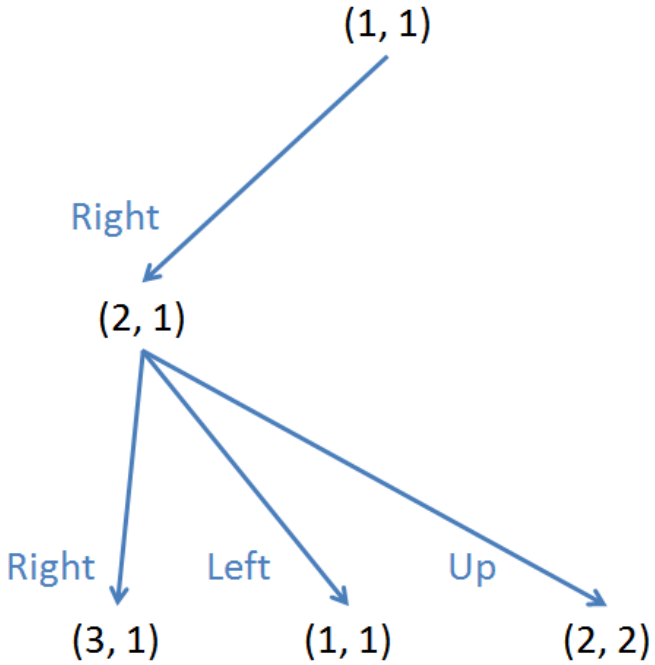
Here is a search tree where we expand the root, generating its children (assuming each movement is legal), and then expand the child node corresponding to a taken "right" action:



à Will this be a valid expansion / generation structure for our maze example at the start of this section?

So, to rectify this, we would clean the tree up as follows, generating only children who were the product of legal moves:





á The set of all leaf nodes available for expansion at any given point of a search is called the **frontier (AKA open-list)**

Search strategies give us systematic means of exploring the frontier.

So, if we wanted some "black-box" (don't care how it's accomplished but know what it's accomplishing) pseudocode for the general idea of search, we might say:

```
; ...for some expand-node function defined
(defun BB-SEARCH (frontier)
    (cond
        ; Base case: ran out of nodes, return nil
        ; (failed to find goal state)
        ((null frontier) nil)
        ; Otherwise, more nodes on frontier, so keep
        ; expanding them
        (t (expand-node frontier))
    )
)

; ...for some goal-state check and
; choose-node functions defined
(defun EXPAND-NODE (frontier)
    (let* ((next (choose-node frontier)))
        (cond
            ; Base case: found goal; done!
            ((goal-state next) next)
            ; Otherwise, expand and continue
            (t BB-SEARCH (append (remove next frontier))
                                 (successors next)
            )
        )
    )
)
; Pseudo-code credit to Evan Lloyd
```

Search Strategies

We'll start by going over two of the most commonly known search strategies that are known as uninformed searches.

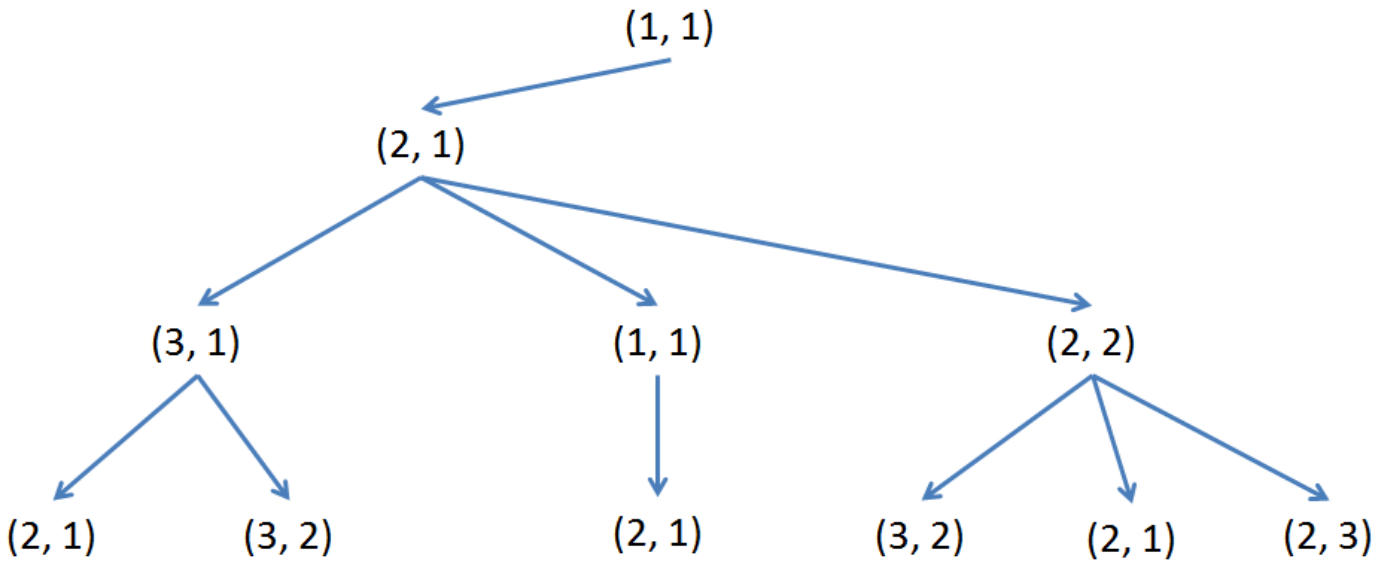
á An **uninformed search** is a search strategy that *only* knows how to expand and generate nodes, and detect a goal state.

á Uninformed search strategies are therefore distinguished by the **order** in which nodes are expanded along the frontier.

We'll begin by looking at breadth-first search.

- **Breadth-first search** expands nodes level-by-level, always expanding the *shallowest* node on the fronteir (i.e., always expanding nodes at depth D before expanding *any* nodes at depth (D + 1).

So, take this partial search tree for example, and determine its breadth-first search expansion order:



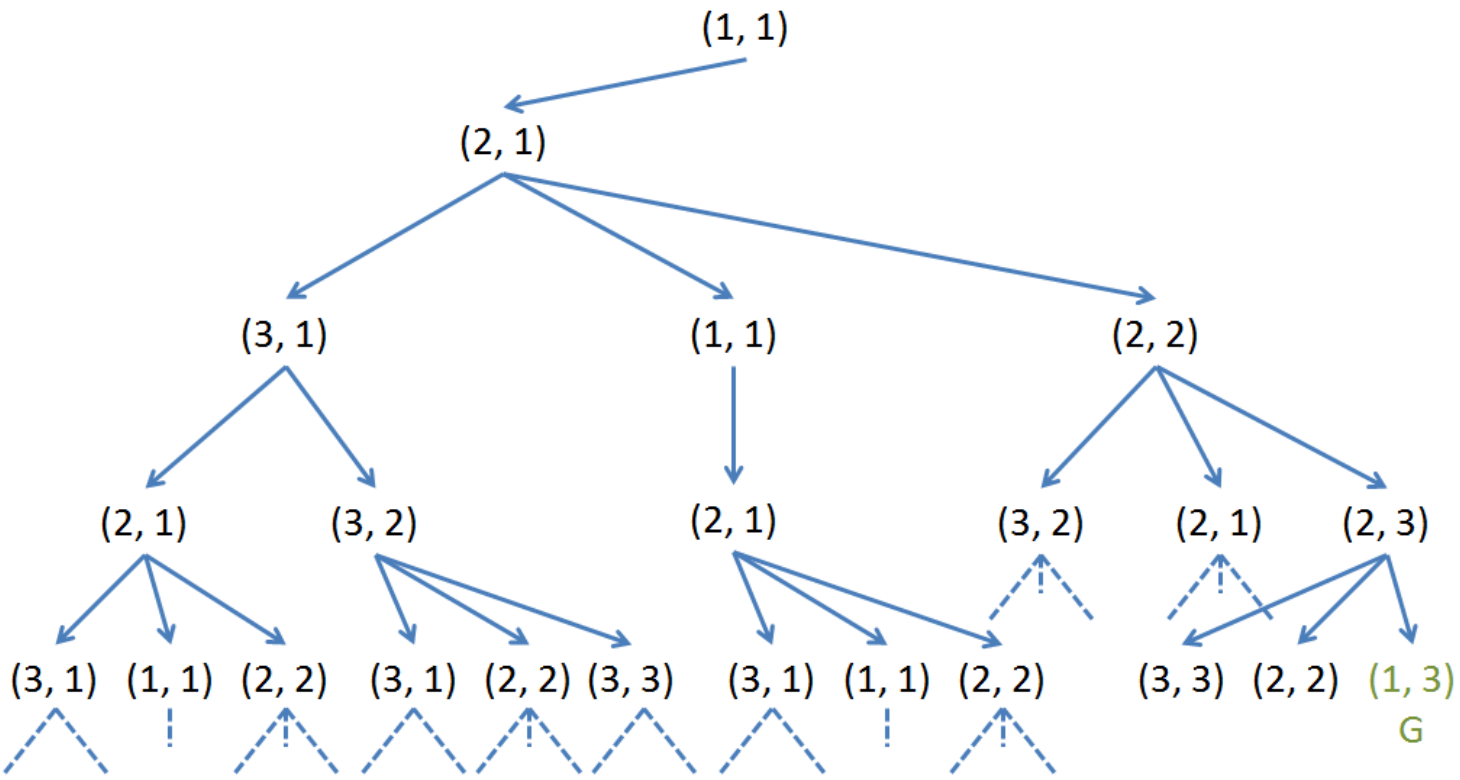
- What is the breadth-first expansion order for this search tree (assume an expansion order of Right, Down, Left, Up preference per level)?

We'll need some tools for evaluating a particular search strategy, and we can start by defining some common characteristics of a search tree:

- The **branching factor (b)** of a search strategy is the maximum number of children a node can generate when expanded.

- The **depth (d)** of a search strategy is the number of nodes expanded from the root to the shallowest goal node.

So if we expand a few more nodes on our frontier from the previous breadth-first search and then reach a goal (as seen below):



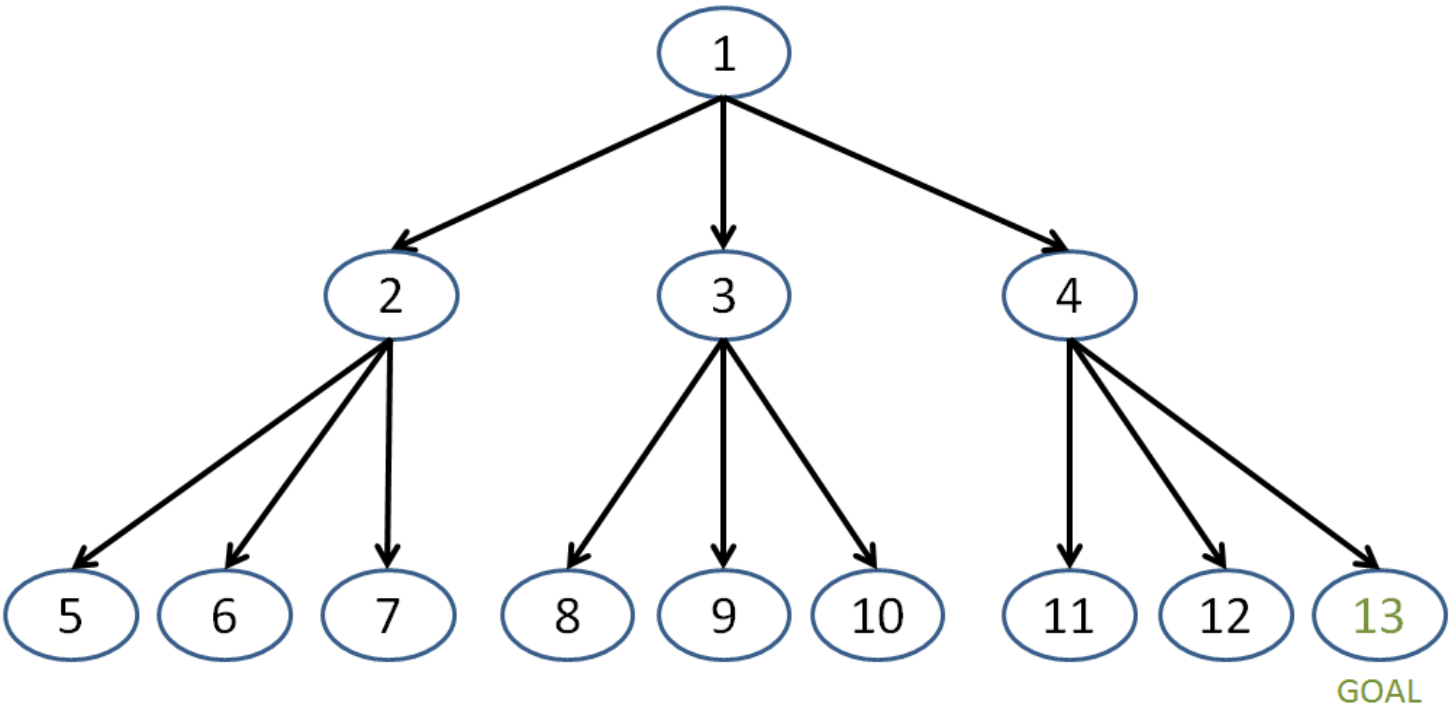
à What are the values for b and d in the above tree?

Using the notions of branching factor and depth, we can assess four metrics for search strategy performance:

á **Time complexity** judges how long our search will take to find a solution based on some assymptotic analysis of a problem's search space (often in terms of b and d).

In other words, we can think of how many nodes need to be expanded before we find a solution.

Let's consider the following abstract representation of a complete search tree:



How many nodes have to be expanded at depth 0?

How many nodes have to be expanded at depth 1?

How many nodes have to be expanded at depth 2?

How many nodes have to be expanded at depth d?

à What is the time complexity for breadth first search in big-O notation?

á **Space complexity** is a measure of how many states we need to keep in memory at any given time for our search strategy.

So, if we look at our abstract breadth-first search representation again, are we allowed to delete any states / branches from memory while we're exploring the frontier?

How many nodes have to be generated at depth 0?

How many nodes have to be generated at depth 1?

How many nodes have to be generated at depth 2?

How many nodes have to be generated at depth d?

à What is the space complexity for breadth first search in big-O notation?

á **Completeness** is a property of a search strategy that asks: "If a solution exists in the search space, is this search strategy guaranteed to find it?"

Now consider breadth-first search and how it expands level-by-level...

à Is breadth-first search complete?

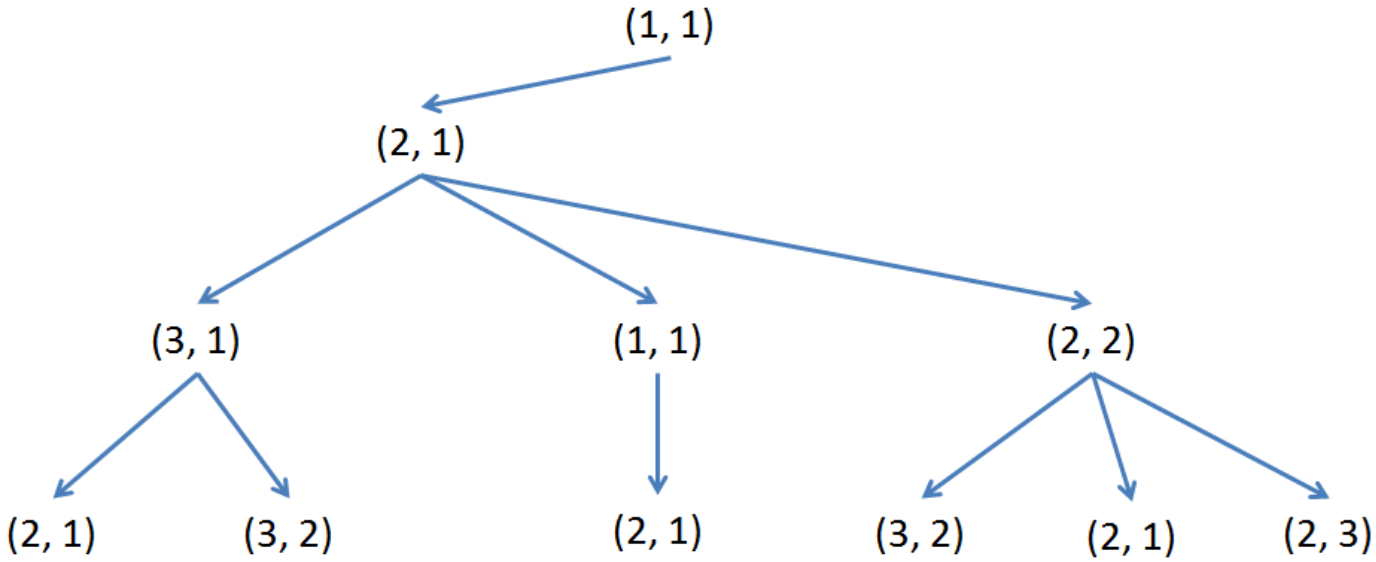
Example
t Prove that breadth-first search is complete.

á **Optimality** is a measure for a search that asks if it will always find the *optimal* solution to a problem (i.e., the solution with the least cost).

Let's look at this in a bit...

á **Depth-first search** is a search strategy that expands the deepest node in the current frontier of a search tree.

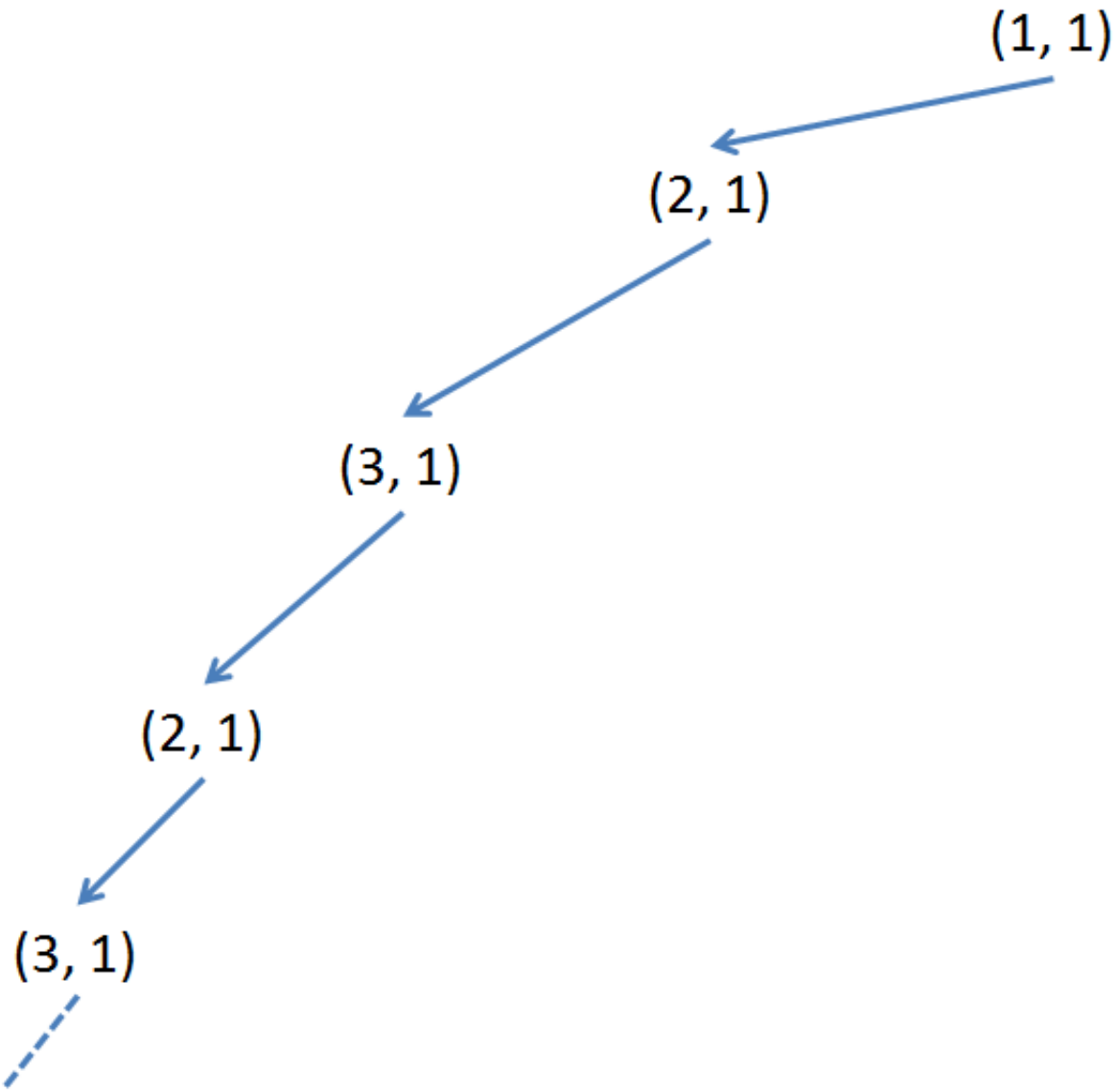
So, take this partial search tree for example, and determine its depth-first search expansion order:



à What is the depth-first expansion order for this search tree (assume an expansion order of Right, Down, Left, Up preference per level)?

Let's return to our maze pathfinding example...

ň What will happen with standard breadth-first search in our partial search tree, pictured below?

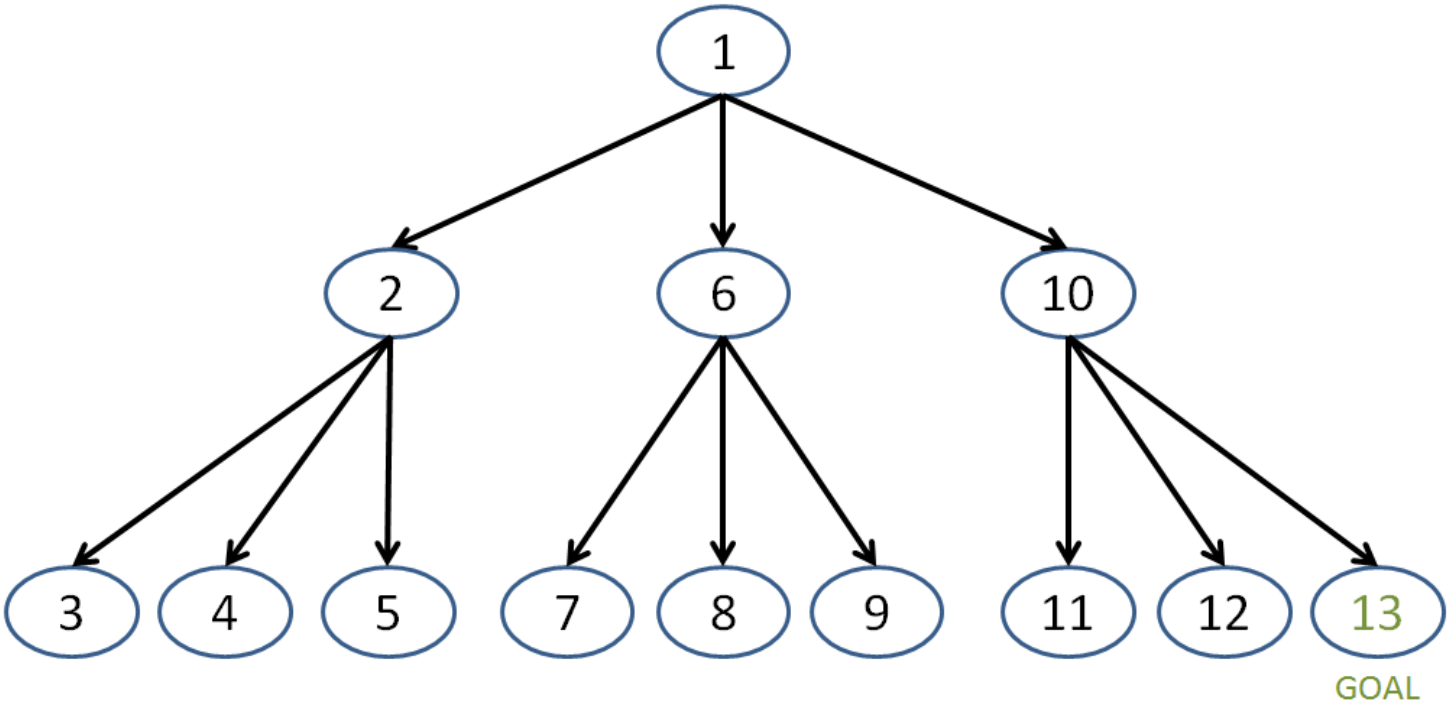


...errr, that's kind of an issue... and rather useless in most cases!

We could keep a list of our repeated states and then remember not to repeat them, but that would grow quickly for large search problems...

à What constaint could we place on our depth-first search such that it won't have the infinite expansion?

Using this strategy, consider our abstract limited-depth-first search tree representation again:



What is the value of m in this tree?

What are the number of nodes expanded at depth 0?

What are the number of nodes expanded at depth 1?

What are the number of nodes expanded at depth 2?

What are the number of nodes expanded at depth m?

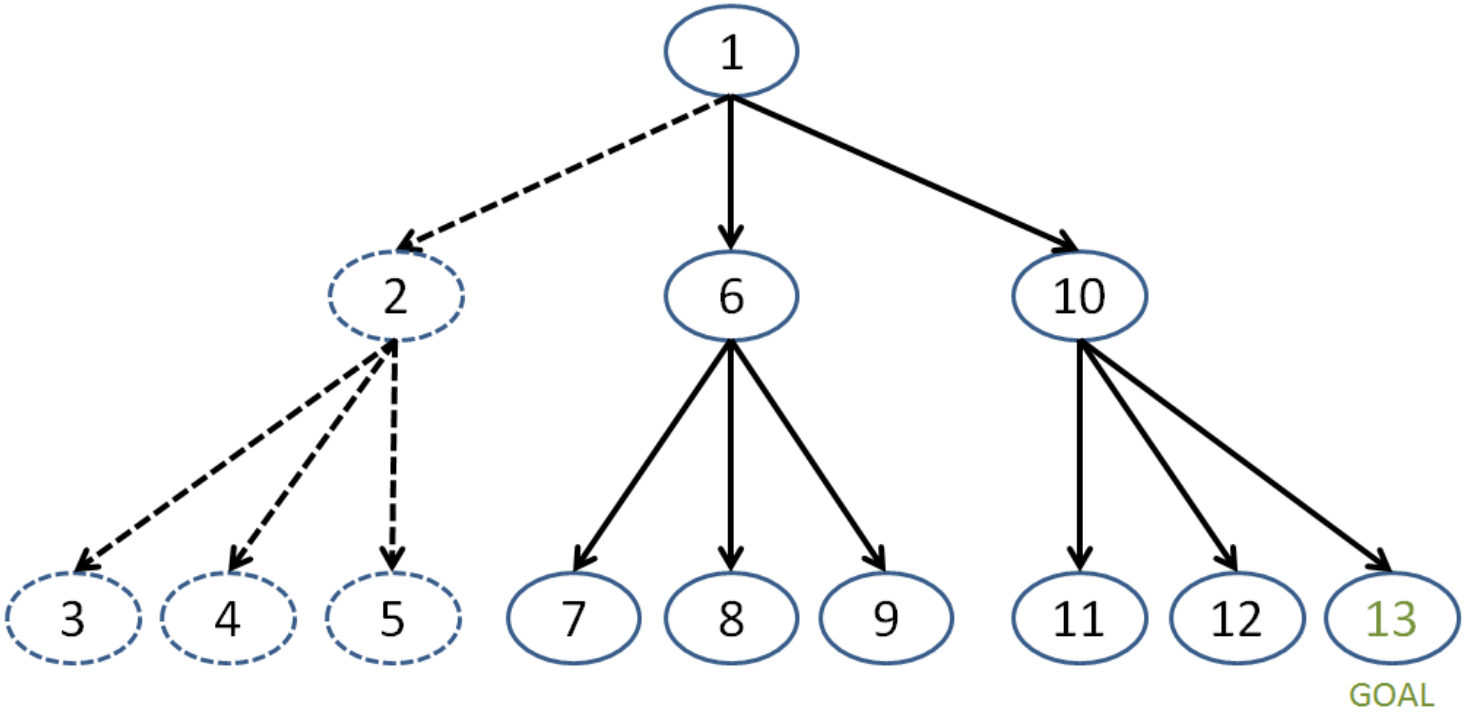
- à What is the time complexity for limited-depth-first search in terms of b and m?

Now, let's examine the space complexity...

With breadth-first search, we needed to keep our frontier in memory because we were expanding level-by-level... is the same true for depth-first search?

Example

- t What happens when we perform expansion #6 in our tree below?



Ah, so as soon as we've hit our depth limit m (a leaf) we needn't remember the branch that got us there!

So, let's think about an upper-bound for the number of nodes we'd need to keep in memory for any depth-first expansion:

à What is the space complexity for limited-depth-first search in terms of b and m?

So we actually save space by using depth-first search (compared to breadth-first)!

ň What happens when we don't know the depth (m) to limit our search to?

What if we chose to limit our search to depth 3 and a goal was at depth 4?

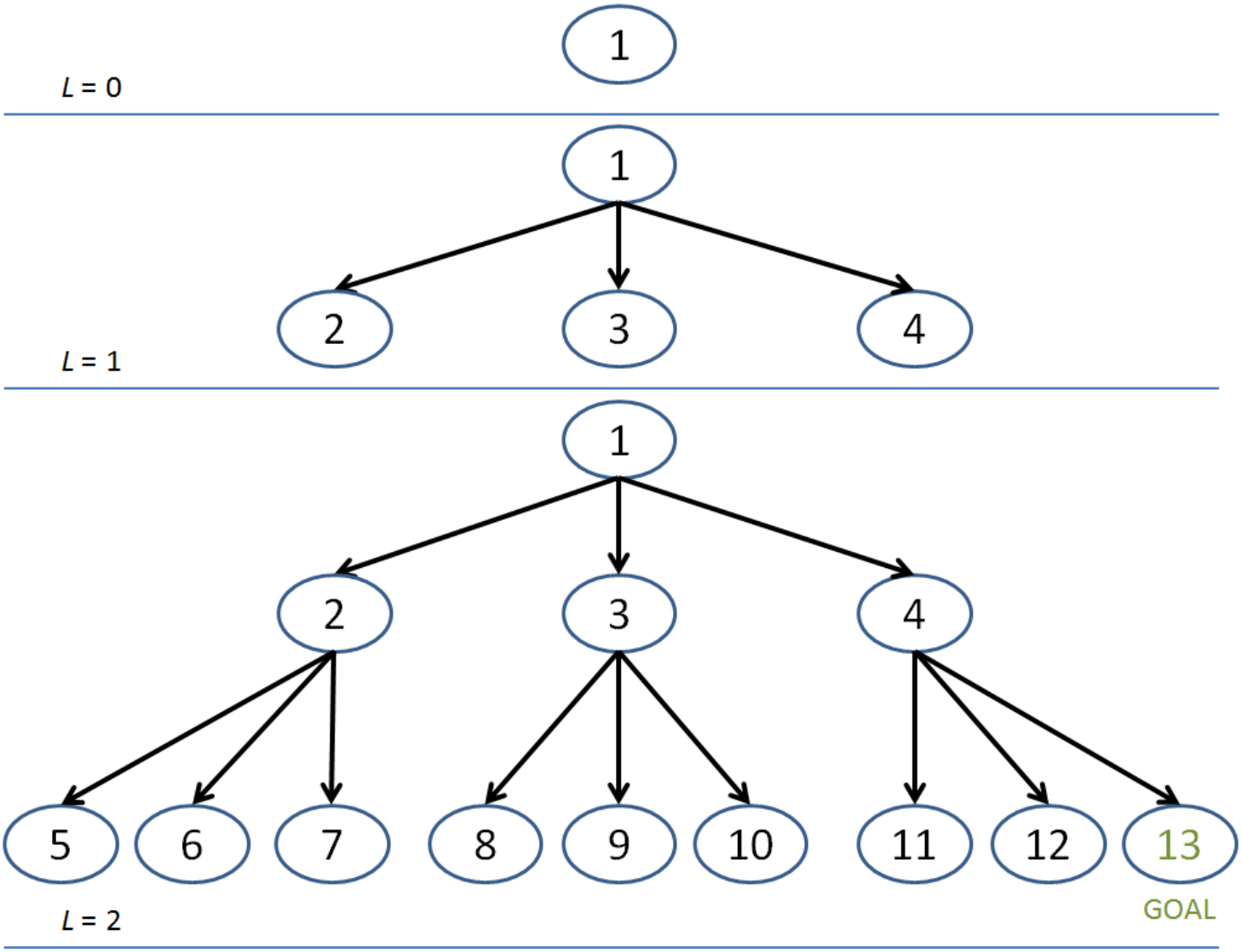
à Are either depth-first or limited-depth-first search complete?

Is there a variant of depth-first search to make it complete?

As it turns out, there is! It was even discovered by our own Dr. Richard Korf!

á **Iterative-deepening depth-first search** is simply repeated application limited-depth-first search where m = 0, then 1, then 2, etc. etc.

Pictorially, then, we can think of iterative-deepening depth-first search as looking like the following, with the current depth-limit (l) starting at 1 and incrementing each time:



à If L is the current maximum depth of an IDDFS iteration, then what are its time and space complexities?

As it turns out, there isn't a whole lot of overhead for using IDDFS over breadth-first, though there is a bit.

Notice that the two strategies are assymptotically equivalent in time complexity!

à Why would you bother ever using IDDFS over BFS?

So now that we see how IDDFS works, we can answer the following question:

à Is IDDFS complete?

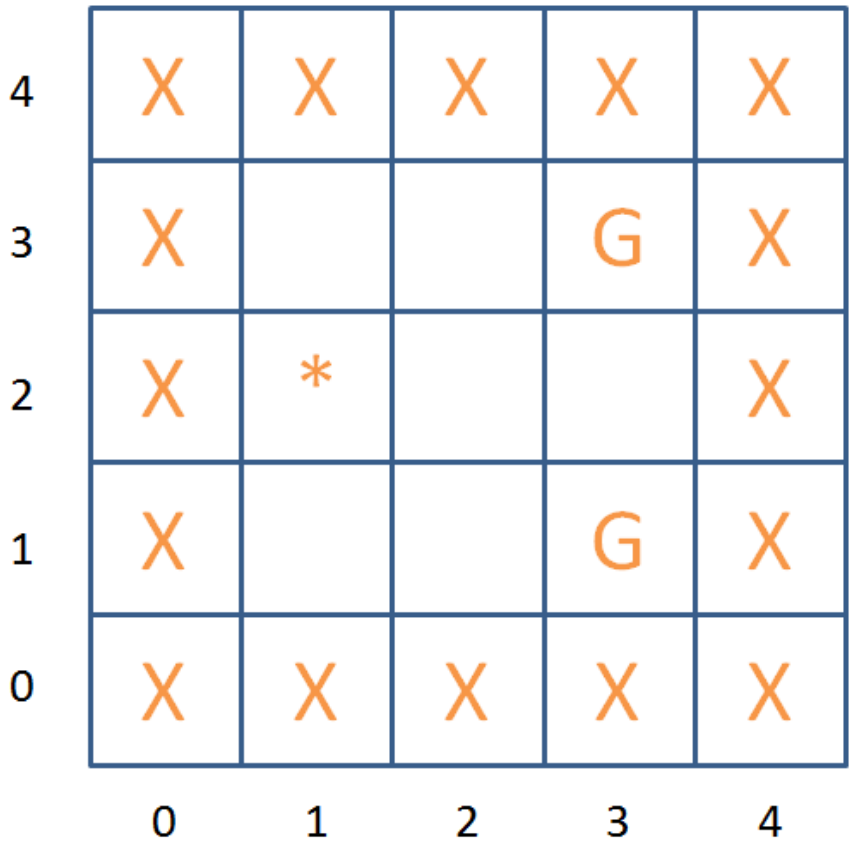
Cost & Optimality

Did you know that there's a difference between the words "optimal" and "optimum?"

á An **optimal** goal state is one with the lowest cost, that might not be the only best solution.

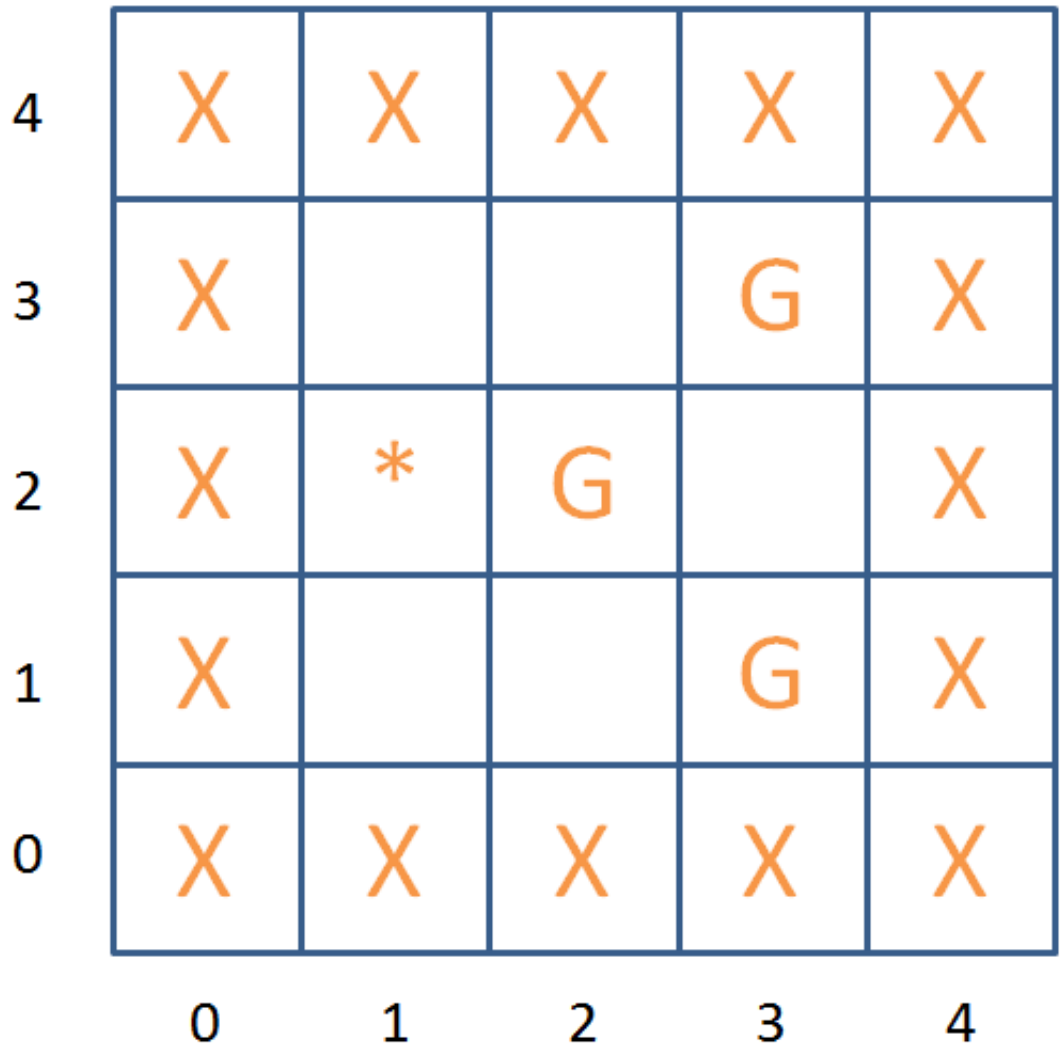
For example, observe below that our two goal states have the same distance from the initial state, so their cost is the same; they are both optimal.

Legend:
X = Barrier (impassable)
* = Player
G = Goal

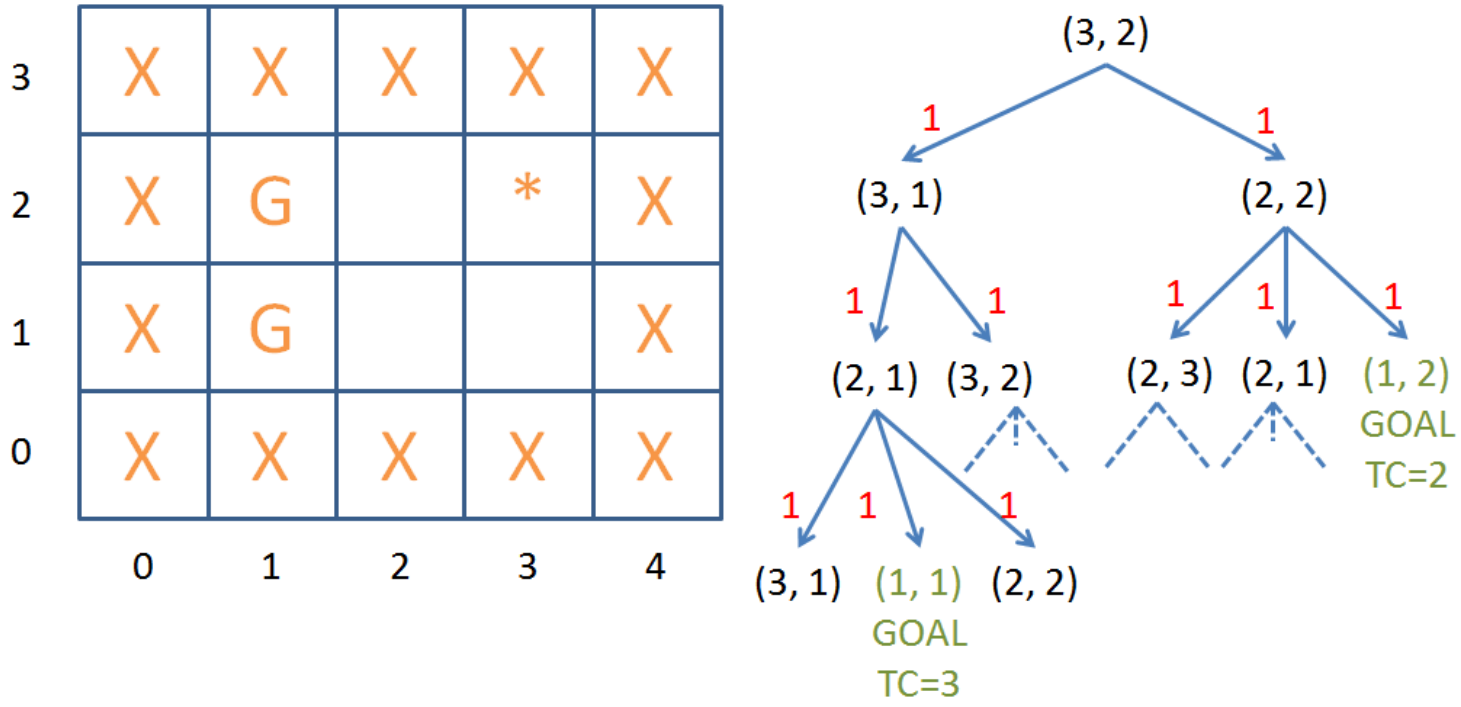


Contrast this with the following situation where we have 3 goal states, but only one of them can be reached with the lowest cost.

á An **optimum** goal state is *the* one with the lowest cost; it may have no peer goal states with an equal or lower cost.



Let's consider a maze where we're interested in reaching an optimum goal; we'll consider the cost incurred during movement as 1 unit per move.



So, this example allows us to ask a few questions about search strategy optimality (is a search strategy guaranteed to return the optimal / optimum solution):

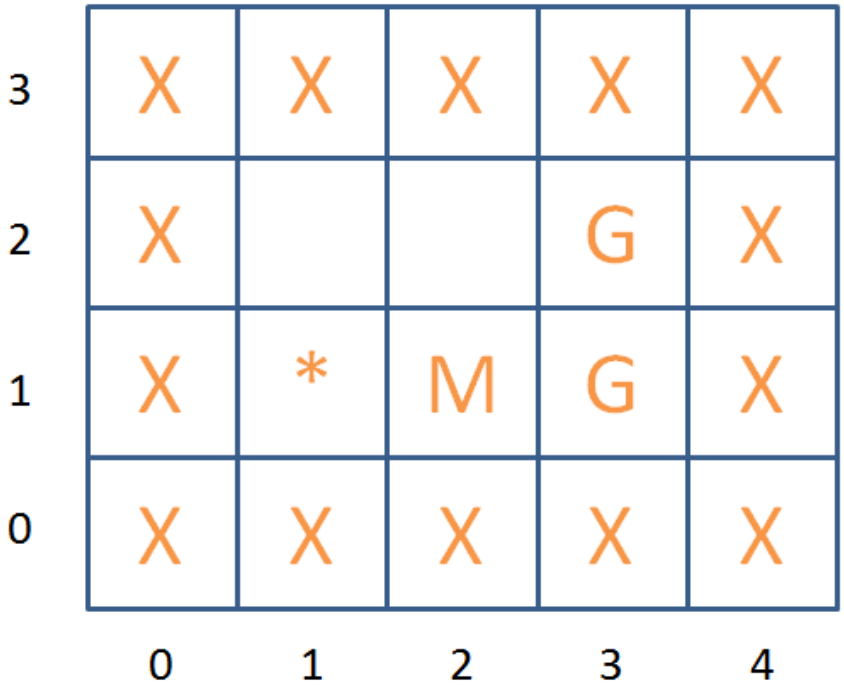
- Is DFS optimal? Is BFS? IDDFS?

That said, this whole time we've considered each of our movements in our maze pathfinder to have an equal cost, and so total cost is simply the sum of the number of movements we make.

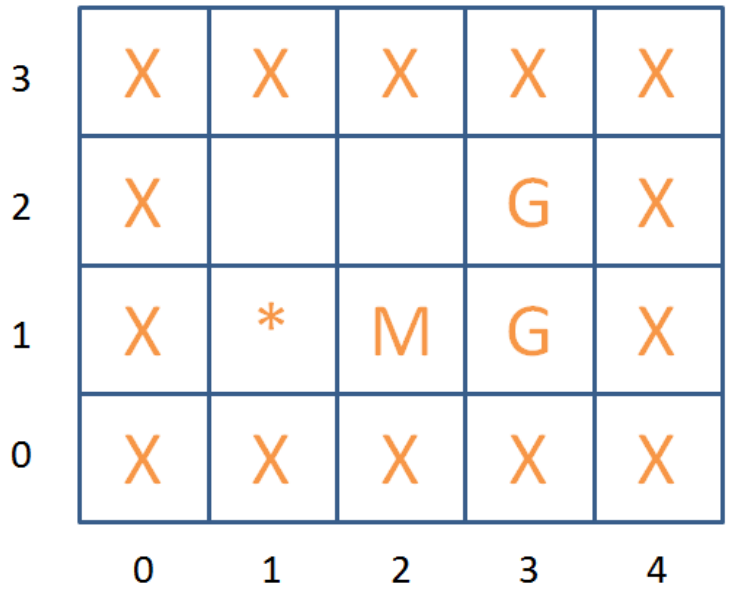
But what if we modified our costs by introducing a new element into our mazes?

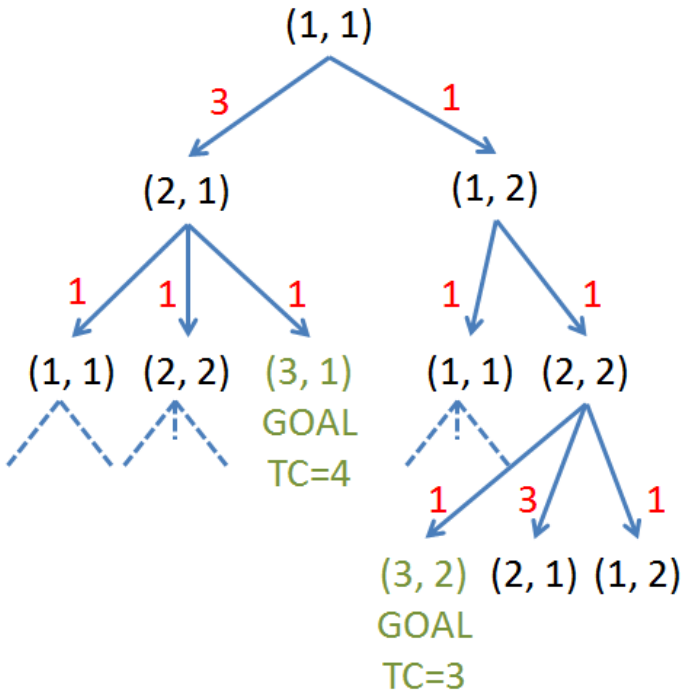
Let's consider clear tiles to have a travel cost of 1, and tiles with 'M' (mud) to have a movement cost of 3.

Legend:
X = Barrier (impassable)
* = Player
G = Goal
M = Mud (cost = 3)
<blank> = Clear (cost = 1)



The search tree for this scenario will look like the following, with costs along the edges:





A couple of observations to make here:

- Is the optimal / optimum solution always just the most shallow in the search tree? When is or isn't it?
- Will DFS find the optimum solution? Will BFS? IDDFS?

Hmm, so it looks like our uninformed searches aren't sufficiently empowered for optimality when we don't have uniform cost.

Let's look at some ways to empower our searches!

Informed / Heuristic Search

As we saw with our uninformed search strategies, we don't know about any descendants of a particular state (states reachable through some number of legal movements) without first expanding the states in between.

But, just because we don't know what states might be along a certain path, doesn't mean we can't make educated guesses.

á A **heuristic** is essentially an estimate of the cost that we might encounter expanding a search tree along a certain path.

The goal of heuristic search is to look at every node along the frontier, and then expand the one that it thinks will lead to an optimal goal.

To do this, we'll construct an evaluation function, f(n) that takes as input a node in our search tree (a state) and computes an estimated cost of reaching a goal state IF we decide to expand that node and follow a path from it.

á **Greedy / Best-first** search is a heuristic search strategy that, instead of having a fixed exploration order, will choose an evaluation function that *only* attempts to minimize cost to a goal.

In this way, we can think of greedy search as ONLY using a heuristic, h(n), where h(n) = the estimated cost of a goal state along the path of node n.

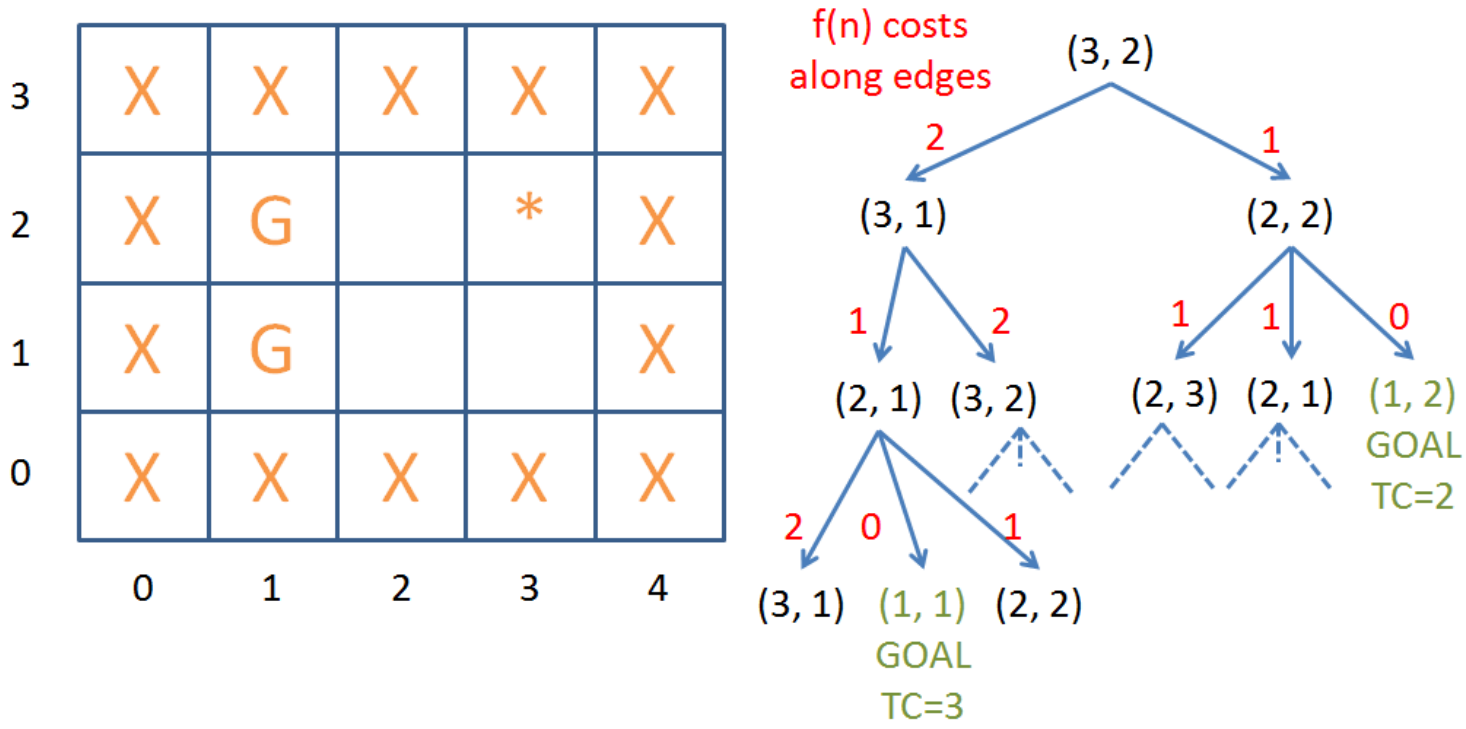
In other words, for greedy search, our evaluation function f(n) = h(n)

à What would be a good heuristic choice for our maze pathfinding example?

So, we might say:

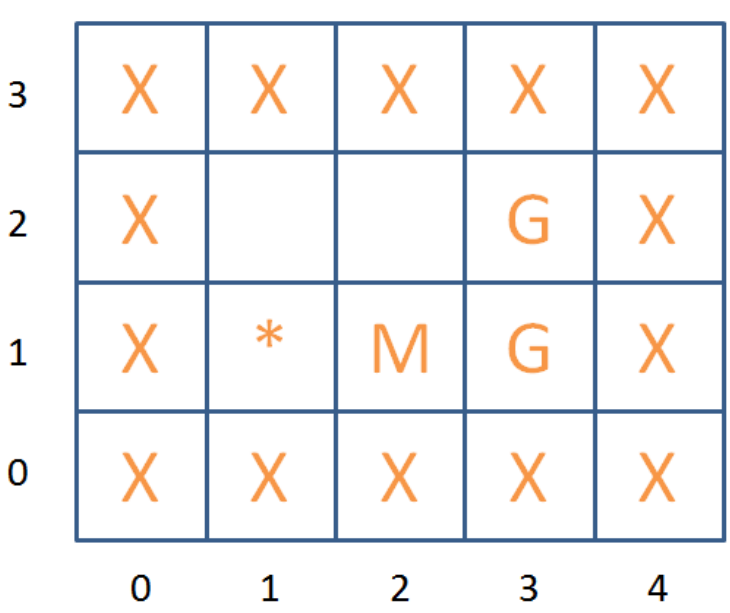
```
Mazefinding Manhattan Distance Heuristic:
f(n) = h(n)
     = argmin( abs(nX - iGoalX) + abs(nY - iGoalY) )
          i

(where n is the player position in a node in the search tree,
and i represents one of each goal tile in the maze)
```

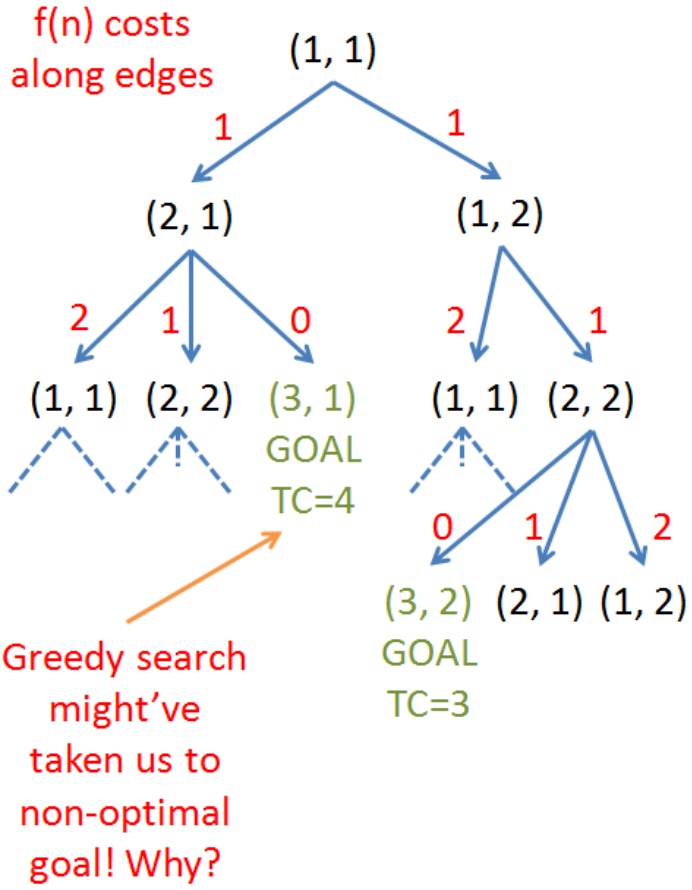
So, seeing that in action:



Notice, using best-first search, now we found not only the optimum solution, but also didn't bother expanding the left branch, and saved a lot of searching!

But now, let's throw in our Mud tiles and see how it does:





Why didn't greedy work for us there?

It didn't remember its history! It stepped on a mud tile (+3 cost) but was only interested in Manhattan distance.

So how do we remember our choices as well?

A* Search

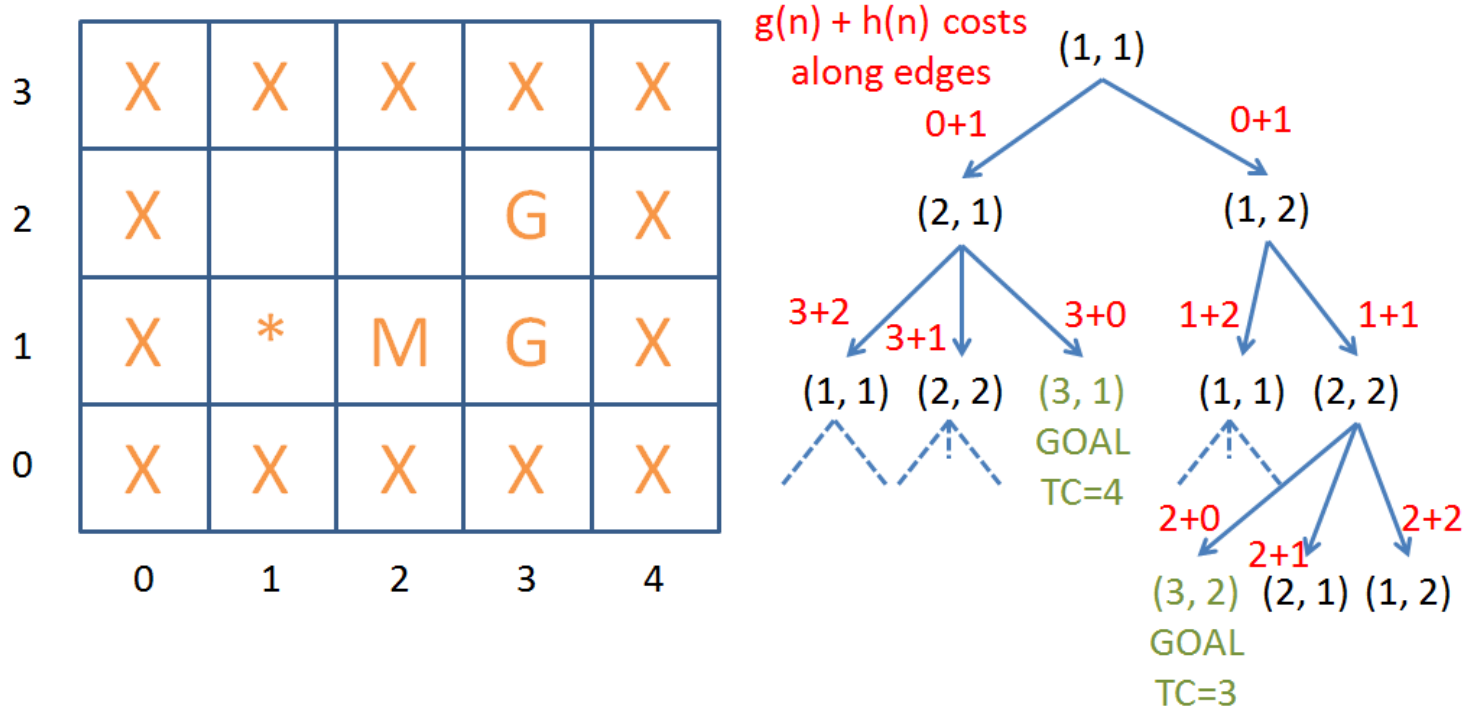
Greedy may have minimized our heuristic function but it didn't succeed with minimizing the *total* cost.

á **A* search** defines its evaluation function as a sum of a heuristic function and a "history" function, g(n). f(n) = g(n) + h(n)

The history function g(n) keeps a total of the *actual* cost we've encountered along our path so far, and then adds it to our heuristic cost for what's to come.

This means that we now choose to expand the node on the frontier with the smallest cost that we've already encountered, and also the smallest cost of what's yet to come!

So, if we maintain our Manhattan distance heuristic from before, we see that A* now successfully navigates our maze into the optimum goal.



à In what order will A* expand nodes in the above example (assume a left-to-right child preference for equal cost evaluations)?

Nice! And we managed to cut off a bunch of branches from our search tree!

à Is A* complete? Is it optimal?

Heuristic Design

As we've seen, a heuristic function h(n) returns a problem-specific value that depends only on the input state, n.

However, in order to be a viable heuristic, it needs to adhere to one central property:

á **Admissibility** is a heuristic property that states that h(n) will never *overestimate* the cost of reaching a goal from node n.

If a heuristic is not admissible (for any case), then all of the nice assurances of optimality are lost for A*.

à Why would an inadmissible heuristic compromise optimality?

Example

t Which of the following are admissible heuristics for our maze pathfinder?

```
h1(n) = Manhattan distance away from the start

h2(n) = (total # of goal tiles)
      - (# of goal tiles in n's row)
      - (# of goal tiles in n's column)

h3(n) = Manhattan distance of n from closest goal

h4(n) = # of Mud tiles surrounding n
```

From the examples above we see that not all heuristics were created equal!

We can compare a heuristic (and therefore rank various ones) to the hypothetical *perfect* heuristic, h*(n), that will always return the exact cost of reaching a goal state from node n.

Through this comparison, we can compute the number of bad expansions our heuristics make in their attempt to guess the best course.

à Is it better for a heuristic to make very conservative cost estimates (smaller) or ones that are as large as possible while still being admissible?

Say we have three heuristics, h1, h2, and h3, and we're comparing the costs that they associate with various nodes:

```
        n1   n2   n3   n4   n5   n6
h1(n)    1    2    2    3    2    3
h2(n)    4    5    5    6    5    6
h3(n)    4    5    6    7    5    6
```

Are h1 and h2 going to have different expansion orders? Will one *not* expand anything that the other does?

What about h3? Will it expand anything that the others don't?

This gives us a simple illustration that:

á If h1(n) >= h2(n) for all n, then:
- Using h2, we expand at least as many (but usually more) nodes than had we used h1
- If we have some set of heuristics, {h1, h2, ..., hN}, then the heuristic h_max(n) = max(h1(n), h2(n), ..., hN(n)) is better than any of the heuristics individually!

So how much work does a good heuristic save us?

á The **effective branching factor (b*)** is simply an approximation of the branching factor that a uniform tree created by an A* solution would have needed to find a solution at depth d.

More formally, b* has the mathematical definition (no closed form) of:

```
Let N be the number of nodes generated by an A* problem solution
Let d be the depth at which a solution was found to said problem
Then, b* is computed from the quantity:

N + 1 = 1 + b* + (b*)^2 + ... + (b*)^d

So, for example, if A* expanded 52 nodes and found a solution at
depth d = 5, then:
N + 1 = 1 + b* + (b*)^2 + ... + (b*)^d
52 + 1 = 1 + b* + (b*)^2 + (b*)^3 + (b*)^4 + (b*)^5
                  |
                  | (maths)
                  v
b* = 1.92
```

A heuristic that's doing a decent job will have an effective branching factor b* close to 1.

Summary

Search strategy wrap-up:

- b = branching factor
- d = depth of solution
- m = max depth for DFS variant
- L = depth limit for limitted DFS

Strategy	Complete?	Optimal?	Time Complexity	Space Complexity
Breadth-first	Yes	Yes (for uniform cost)	Exponential: O(b^d)	Exponential: O(b^d)
Depth-(limited)-first	No	No	Exponential: O(b^m)	Linear: O(bm)
Iterative-Deepening-Depth-first	Yes	Yes (for uniform cost)	Exponential: O(b^L)	Linear: O(bL)
A*	Yes	Yes	Exponential (worst case), Linear (best case): Depends on heuristic	Exponential (worst case), Linear (best case): Depends on heuristic

---

Local Search

Sometimes, our search space is just too massive to tractably search exhaustively using classical search methods.

So, we can try something akin to intelligent dart-throwing that just kinda gives up on definitely finding an answer.

Like a lazy mechanic, local search says, "I'll try a few things, but no guarantees."

á **Local search** is a search strategy that attempts to guess a correct answer from a random initial state and then movements that attempt to find a goal from that start.

With Local Search, you need to have a few definitions for your problem:

- A **neighborhood** of a state is a set of all states that can be reached from performing one action on the current state.
- A **state score** is a function that evaluates how close a given state is to a goal state.

With these two elements defined, we'll try to move around from neighbor to neighbor until we find the state with the best score (if we don't need to be perfect) or meeting some threshold.

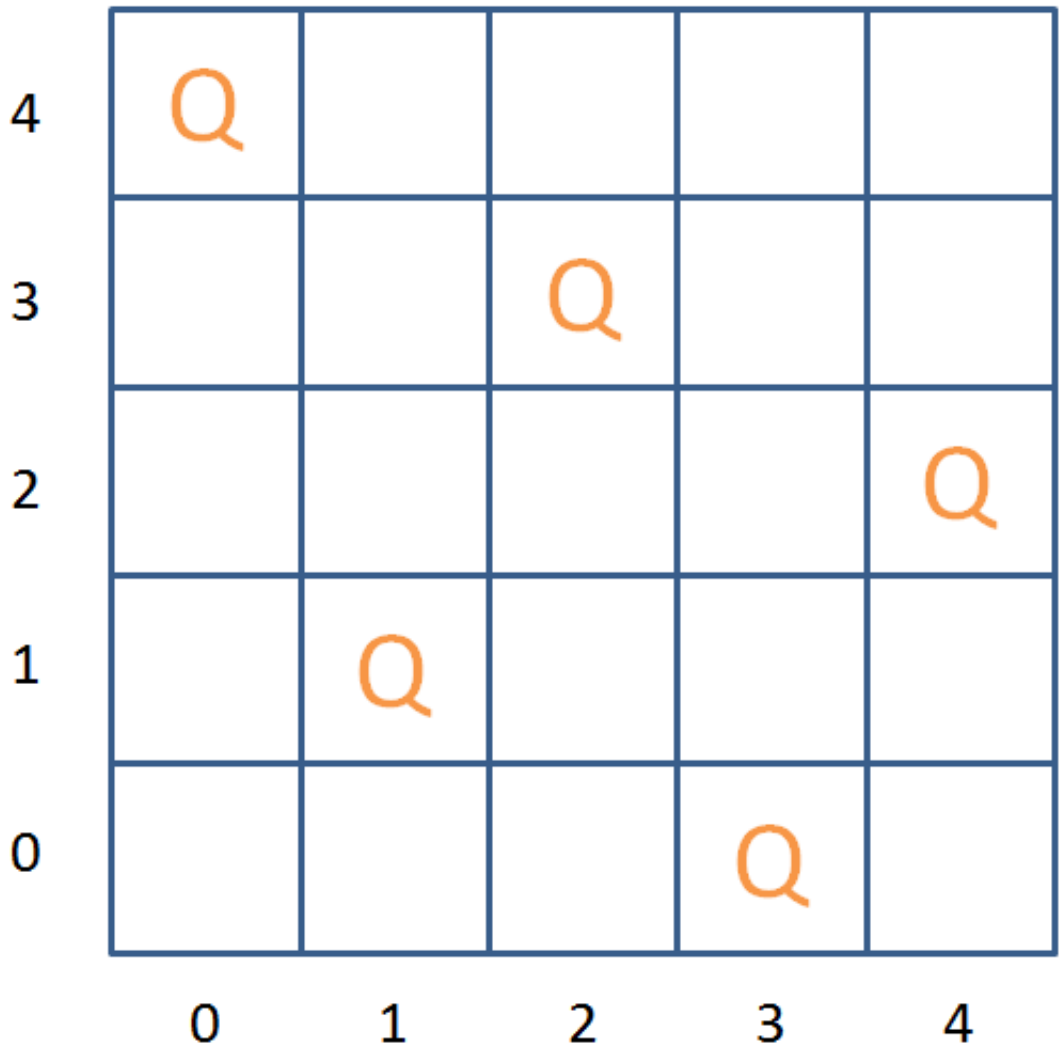
Local search, dependent upon the particular implementation, usually has the following general steps:

1. Try a random state (variables of interest randomly assigned values inside the search space)
2. Is it the goal state (or does it meet some threshold)? If yes, done!
3. Otherwise, try to massage that random starting position into the goal state.
4. Repeat step 3 some N number of times (don't want to keep going forever), and if you still haven't found a goal, we'll try some other things (discussed later)

Example
t The N-Queens (typically, the 8-Queens) problem asks for an instantiation of some N queens on a chessboard such that no queen is in position to capture another.

For those unfamiliar with chess, queens can move in any straight-line direction for as many spaces as they want (horizontally, vertically, or diagonally).

Here's a solution to the 5-queens problem:



These problems aren't so bad for low N, but what about the 100-Queens problem? Ugh...

So, why don't we try using this example for local search?

Let's make one improvement at the start and place the restriction that 1 column (or row) has to hold a single queen (or else we're immediately in trouble!)

à For our N-Queens local search, what will a state, neighborhood, and score look like?

So with those definitions in mind, we could go column by column and determine the number of queens that would be capturable by moving any queen within a given column to some other position.

For example if, in my 5-Queens problem, I got the following random assignment of queens to column positions, then I could view the scores of each neighbor state by determining how many queens would be capturable if I decided to move one into a given slot.

Then, we just try to minimize the cost until we (hopefully) find a solution!

á This sort of cost optimization local search is called **hill-climbing**, which attempts to find peak solutions.

á **Peaks** are a term in local search meaning that a solution with cost C has been found but for which all neighbors have some cost greater than C.

So, if we found a peak, then we might not be able to do better in that particular local exploration, and we don't want to get stuck!

à What are some strategies to avoid getting stuck with a bad random initial guess?

Some of these improvements may seem to be minor but can have a huge impact.

For example, allowing sideways movements in the 8-queens problem can change success rate from a lousy 14% to a respectable 94%!

ň Warning: Local search is not appropriate for all problems. Ask your doctor if local search is right for you.

à For example, would local search be appropriate for our maze pathfinder?

à Is local search complete? Optimal?

Homework 1 Questions

If you have any HW1 questions, now's the time to ask them!

Just to reiterate a forum post, let's look at how to picture frames from a tree perspective:

```
(STATE TYPE (EMOTION SENTIM (POS)
                     SCALE (>NORM))
       AGENT (HUMAN F-NAME (CHARLOTTE)
                    L-NAME (NEWTON)
                    GENDER (FEMALE))))
```