MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence, Fall 2001
Recitation 5, October 5, 2001

I(D) Can't get no satisfaction

Prof. Robert C. Berwick

<u>Agenda</u>

1. **Administrivia**
2. **Deepening the search: iterative deepening to trade**
3. **ID for A* search**
4. **CSP *or* the Last Waltz**
5. **Waltz world example**
6. **FC and all that**
7. **Search for financial aid**

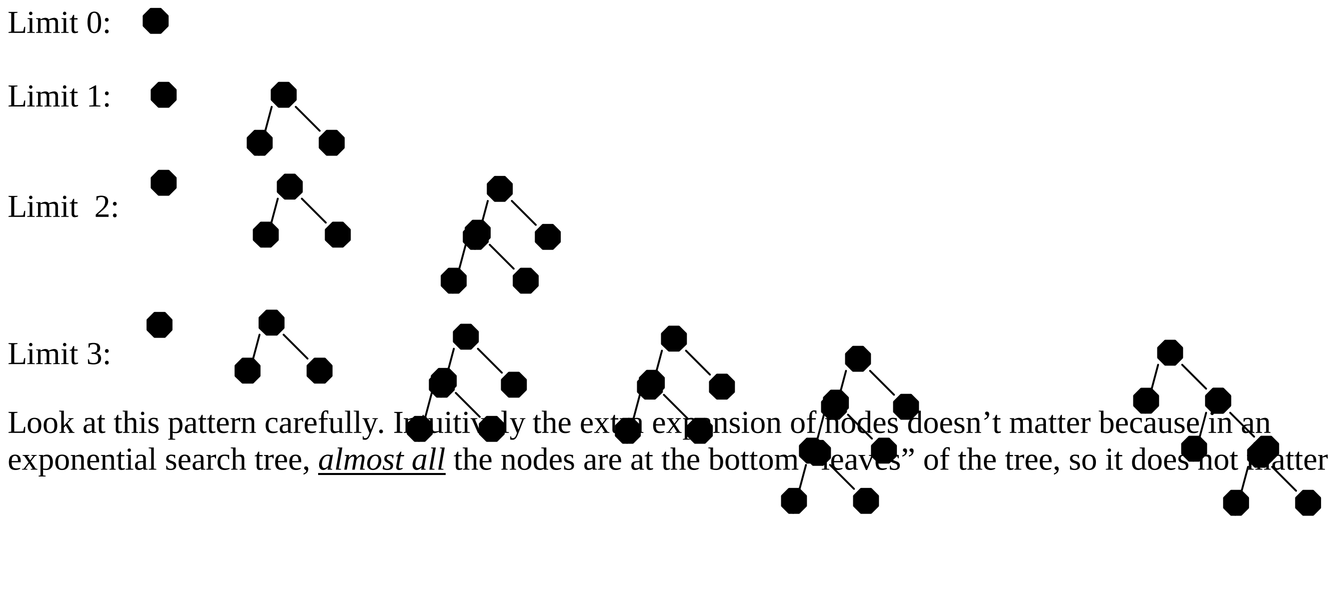
1. Administrivia: Holiday Monday; PS due Weds.

2. Iterative deepening (aka Progressive Deepening)

Iterative or progressive deepening sidesteps issue of choosing the best depth limit by trying all possible depths: first 0, then 1, etc. It combines the benefits of depth-first and breadth first search: It is *optimal* and *complete* like breadth-first search, but has only the memory requirements of depth-first search. Expansion order of nodes is similar to breadth-first, but some states are expanded multiple times. This might seem like a waste, but it is not, because the overhead is really small. Example: when $b=10$, then the number of nodes that ID expands going all the way from $d=1$ to $d=10$ is only about 11% more than single breadth first of depth-limited search to depth d. The higher the branching factor, the *lower* the overhead of repeatedly expanding states, but even with $b=2$, the overhead is only about twice as long as a complete breadth-first search. So, the time complexity is *still* $O(b^d)$ and the space complexity is $O(bd)$. Let's see why.

```
function iterative-deepening (problem) returns a solution sequence
    inputs problem
    for depth 0 to infinity do
        if depth-limited-search (problem, depth) succeeds then return its result
    end
    return failure
```

Four iterations of iterative/progressive deepening (ID) on a binary search tree – the nodes expanded in turn:



Look at this pattern carefully. Intuitively the extra expansion of nodes doesn't matter because in an exponential search tree, <u>*almost all*</u> the nodes are at the bottom "leaves" of the tree, so it does not matter that the upper levels are expanded multiple times. If the tree is searched to depth *d* and the branching factor is *b* then the number of node expansions is:

$$1+b+b^2+\cdots+b^{d-2}+b^{d-1}+b^d$$

For *b*=10 and *d*=5 this is 111,111.
In iterative deepening, the nodes on the bottom (leaves) are expanded only once, those next to the bottom are expanded twice, etc., up to the root of the search tree, which is expanded *d*+1 times. So the total # of expansions in a progressive deepening search is just the above sequence times the decreasing sequence *d*+1, *d*, …, 1:

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

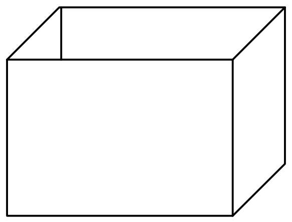
If we again calculate this value for *b*=10 and *d*=5, we get: 123,456 – an increase of only 11% over plain depth-limited search.

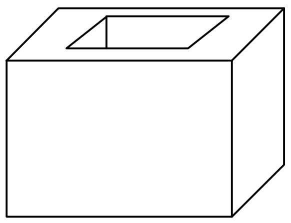
In general, iterative deepening is the preferred search method when there is a large search space and the depth of the solution is ***not*** *known.*

A* iterative deepening: use *f-cost* instead of *depth – put a contour on the search boundary*

3. Constraint satisfaction problems (CSPs): The Last Waltz

Warmup: try your hand at this…





What constraint about the physical world is *implicitly* involved in the Waltz labeling as revealed by attempts to label the two line drawings above?

What's in constraint propagation? Ans: **variables, values, constraints, domain (possible values), eliminating**

- The **key** point about constraint satisfaction problems is that there are some *additional* constraints that the problem has – the goal test specifies a set of constraints that the values of the problem variables must obey. For instance, in the Waltz line-labeling world, the 'values' at the junctions must obey the compatibility constraints with the assigned values at adjacent junctions (if a line is labeled + coming from one junction, it's + going to an adjacent junction on the same line- that's a property of the physical world projected into our 'toy' one).
- Constraint satisfaction problems consist of states that are defined by the values of a set of **variables**, each variable has a **domain** which is the set of possible values that the variables can take on (What are the variables in the Waltz world? What are the variable values?)
- CSPs also consist of a set of **constraints.** A **unary constraint** is just some allowable subset of the domain, e.g., from the set of all things, the 'foods' = consumable. A **binary constraint** between two variables specifies some subset of the cross-product of the two domains. For example, in the 8-Queens problem, let V_1 be the row occupied by the first Queen in the first column, and let V_2 be the row occupied by the second Queen in the second column. The domains of V_1 and V_2 are … (you tell me). The **no-attack** constraint between the two variables can be represented as a subset of the cross-product of these two domains that represent allowable pairs: {<1,3>,…}. How many combinations are possible *without* this constraint? How many are possible *with* it?
- Constraint satisfaction problems can **take advantage of the problem structure** better than plain search to eliminate a large fraction of the search space, in the best case. (In the very best case: no search remains at all – just singleton values for each variable. If multiple values – still search left to do.)

- The principal source of structure in the problem space is that **the goal test is decomposed into a set of constraints on variables rather than being a unitary, black-box, 'goal'.** This is what allows us to do searching incrementally, and solve *part* of the problem first, like a jig-saw puzzle, before solving the *rest*. (In the Waltz world, we can label a pair of junctions perhaps uniquely before moving on – divide and conquer works.)
- We can represent variables as nodes and constraints between them as arcs (ie a graph)
- For instance, take the 8 Queens problem. We *could* do "Homer Simpson" depth-first search, and try out an assignment of rows to *all* 8 Queens at once – that is, assign *all* variable values in one fell swoop – say, (2, 3, 1, 4, 6, 5,7, 8). After all, the solution must be at depth 8. What branching factor does this result in for the search tree? (Ans:)
- So: suppose we realize that the **order of variable assignments makes no difference.** So, we need only pick the value for the first Queen, then the second at the next level of the search tree, and so on. So, how big is the search space now? (Ans: .)
- We can use the **no-attack** constraint incrementally to cut down on this search space. Straight DFS wastes time searching when constraints have already been violated. For instance, suppose we put the first two Queens in the top row (already a violation). DFS will examine all 262,144 possibilities for the remaining six Queens before discovering that this entire subtree is a loser. The **key** point is that once a variable assignment causes a violation, it can **never** be rescued by another variable assignment – local losing means global losing.
- We can thus improve this one-variable-at-a-time DFS by **checking** constraints as we go. Once we get a violation, we can abandon that line of variable assignment. *<u>All the types of CSP are defined by how this checking is done</u>*. So let's go through these a bit. (We've already just done the case of *no* checking: regular DFS.)

- The *simplest* checking is what we just described **-** just *one* variable at a time, and then backing up when we violate a constraint: **backtracking search.** Where would this go off the rails in our 8-Queens problem? Well, suppose we place the first 6 Queens so they attack all 8 squares in the 8th column – thus making it impossible to place the 8th Queen. Even so, backtracking search will try all the ways to place the 7th Queen first, even though this is wasted work.
- So, we add just the *smallest* bit of look ahead to try to detect unsolvability: check constraints from just the variable we have lately instantiated (=**backtrack-fc**). Each time a variable is instantiated (we assign a value to it), we *delete* from the domains of all the other not-yet-instantiated variables those values that conflict with the variables assigned so far. If any of the domains become empty, then we know that particular value assignment to the variable is not kosher. For instance, in the 6 Queens example, we already know that the assignment we have chosen for the 6th Queen will make the 8th impossible, because the domain for the 8th Queen goes to the null set. So, we must backup and choose another value for the 6th Queen (or beyond), and we don't mess with the 7th at all.

- For many problems, forward checking does as well as more sophisticated approaches – it is also easy to implement.

- If we insist on the *stronger* condition that, for a given state of the problem, **all** variables have **some** value that is consistent with **each** of the constraints on that variable, then this condition is dubbed **<u>arc-consistent.</u>**

This is Waltz labeling: we pile up variable values at a node (variable), and then propagate until we have eliminated as many values as possible (perhaps all). **Waltz labeling=arc consistency.**

- We can thus regard all these methods as some combination of *tree search*+ some fraction of *arc-consistency.* Zero arc-consistency= simple backtracking; Consistency just from singleton (single-valued) variables= forward checking; Consistency from *all* variables (even multiple valued) = arc-consistency.

One estimate is that backtracking does about 1/5 of full arc-consistency while forward checking (fc) does about ¼ of full arc-consistency (of course, this depends on the graph that describes the constraint system).

- **MORAL: good behavior in constraint propagation is a function of the <u>domain</u>, not the formulation of constraint satisfaction problems generally.**

4. Constraint propagation: what are the **variables?** The **values?** The **constraints?**

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

One formulation is this: Each domain is 0,1,…,9, for each of the variables s, e, n, d, m , o ,r , y. How may possibilities? Ans:

$$\begin{array}{rrrrr} & 1000s + & 100e + & 10n + & d= \\ & 1000m + & 100o + & 10r + & e= \\ \hline 10000m + & 1000o + & 100n + & 10e + & y \end{array}$$

Why is this not so good? What's another formulation where we can cut down the # of variable value combinations to a combination of 4 problems of $O(10^4)$ instead of the much larger number above? (Hint: think about the dual formulation – turn constraints into variables and variables into constraints.)