

- Patrick Winston
  - a: 6.034.mit.edu
  - No laptops / cell phones
    - insults instructor
  - "Most important MIT class"
    - learn to be real smart
- 

def Thinking  
Perception  
Action

Models thereof

- needed to understand
- predict what's happening

Need <sup>right</sup> representation to figure out

See constraints exposed by

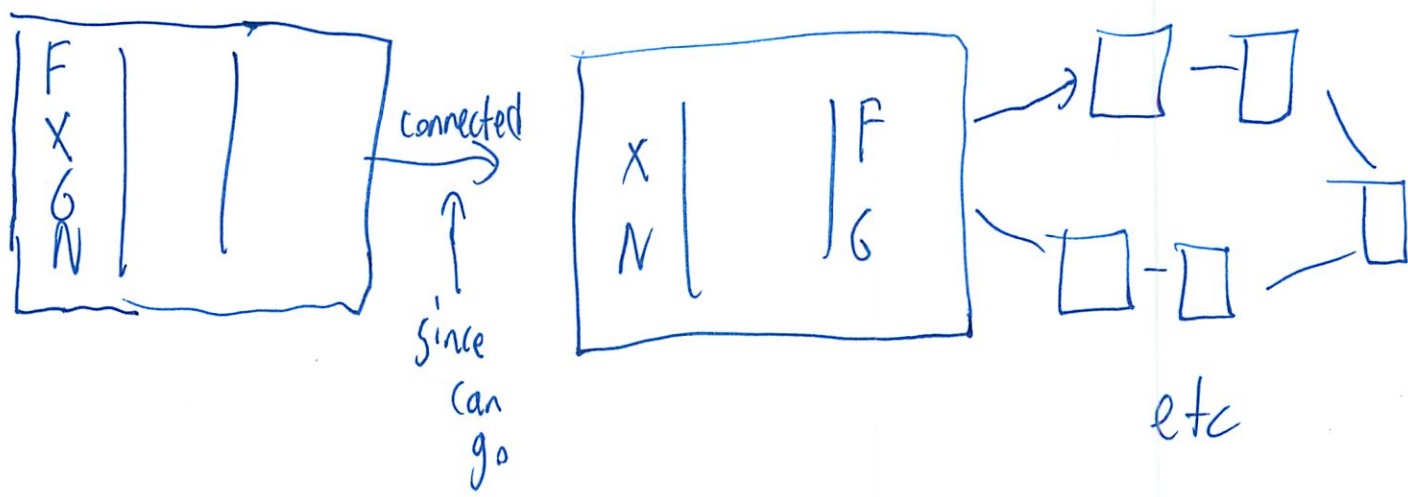
---

Goose + Brain + Farmer problem

Start w/ right representation

②

1. List all possible combos - and next possible legal rules  
 $2^4$



~~2. Well in this representation~~

2. Now just simple ~~graph~~ graph problem

def get methods that are enabled by

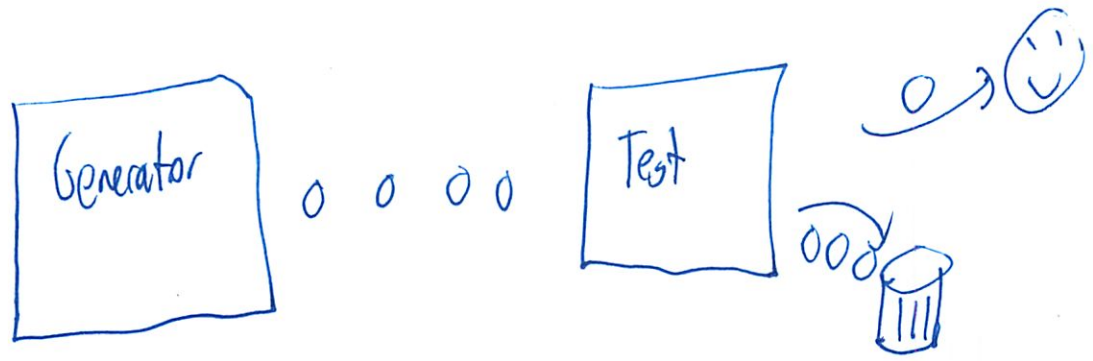
← what if found tree leaf?

- go through tree booklet

Is there an easier way?

Or way we can write it in a model

3



~~all possible~~

- complete = all possible solutions
- non redundant
- in formable

↑ Good to name simple things

- get power over it

Why AI?

Building intelligent systems

And what's special about humans

Deep Blue + Watson can't go to a conference on themselves  
 (History of field) - no model of itself

9

Symbolic integration (mor)

Pattern matching by building description

Short description ~~answer~~ answer

Answering about arches

- small # of examples

Blood cultures checking

Big engineering impact

- flights
- brain tumor surgery

Thinking about thinking

- Plato

60,000 years ago humans started cave ~~paint~~ paintings

Put <sup>2</sup> concepts together w/o limit

- learning stories

Reading summary of Macbeth

2 diff ways of drinking

- human vs cat
- dogs can't make that distinction

3  
No mega + regular recitations this week

Friday  
10AM

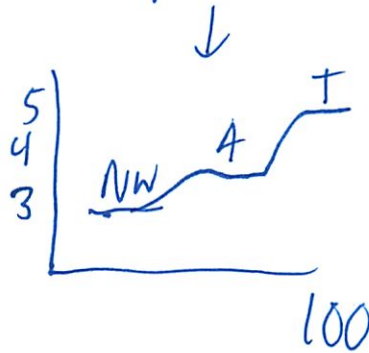
quiz  
problems

↑  
enrichment

Tutorial - help w/ hw

4 quizzes

- for each quiz step function



→ Max

F<sub>1</sub> - final part 1  
F<sub>2</sub>  
F<sub>3</sub>  
F<sub>4</sub>  
F<sub>5</sub>

Hw →  
Grade ← etc

Just show know stuff before semester is over

On board

- Right + representation → almost done
- tutorial ≠ simple
- Rumpelstiltskin principle

# 6.034 Models of Problem Solving

9/12

Song "You can get it if really want it"

---

Programs to integrate expression

- need problem solving that would work

- Generate + Tests?

↳ last semester

- ~~the~~ Today: Problem Reduction

- have a too hard problem

- so you simplify it

- to something hopefully easy

- to build you need to load w/ knowledge

- like from integral table, basic

$$\int \frac{1}{x} dx = \ln x$$

$$\int x^n dx = \frac{x^{n+1}}{n+1}$$

$$\int e^x dx = e^x$$

$$\int \cos x dx = \sin x$$

... 26 total nodes

- then need to represent in program

- then know how it is used

②

Next sure fire transformations

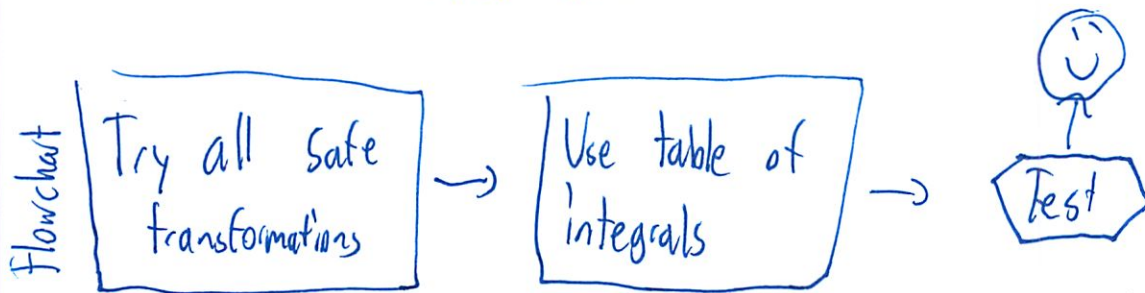
① -  $\int -f(x) dx = -\int f(x) dx$

② -  $\int \sum_i f_i(x) = \sum \int f_i(x) dx$  generally a good idea

③ -  $\int f(x) \cdot c dx = c \cdot \int f(x) dx$

④ -  $\frac{P(x)}{Q(x)}$  Divide

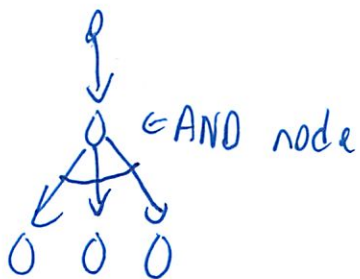
- 12 total transformations



But the program did not get an A in class

Needs Heuristic / risky Transformations

- Makes a whole set of new problems to be solved



3

(A) -  $f(\sin x, \cos x, \tan x, \cot x, \sec, \csc x)$

$\rightarrow g_1(\sin x, \cos x)$

$g_2(\tan x, \csc x)$

$g_3(\cot x, \sec x)$

(B) -  $\int f(\tan x) dx \xrightarrow{y=\tan x} \int \frac{f(y)}{1+y} dy$

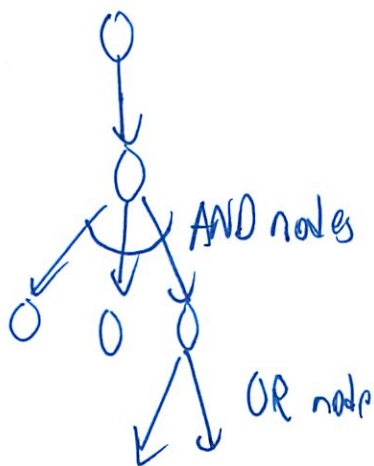
(C) -  $1-x^2 \xrightarrow{x=\sin y}$

$1+x^2 \xrightarrow{x=\tan y}$

↑ big hairy algebra

- 12 in total

So we not only have <sup>nodes</sup> AND, but OR nodes



← AND/OR tree  
aka Goal tree

← Heuristics are  
OR nodes



(4)

Only use heuristic on easy problems

- may be multiple branches

- ~~for~~ just use easiest

Look for depth of function composition

So lets do it for  $\int \frac{x^4}{(1-x^2)^{5/2}} dx$

1. No safe transformations

2. Not in my table

3. Look for easy problem

- found (C)

$$\frac{x = \sin y}{dx = \cos y dy} \rightarrow \int \frac{\sin^4 y}{\cos^4 y} dy$$

1. No safe

2. Not in table

3. Look for easy

- found (A)

(5)

$$\begin{array}{l} \nearrow \\ \text{OR} \\ \searrow \end{array} \int \frac{1}{\cot y} dy$$

$$\int \tan^4 y dy \quad \leftarrow \text{continue trying here}$$

- 1. No easy
- 2. Not on table
- 3. Use heuristic (B)

$$\underbrace{z = \tan y}_{\rightarrow} \int \frac{z^4}{1+z^2} dz$$

- 1. ~~is~~ an easy (C)

$$\rightarrow \int (z^2 - 1 + \frac{1}{1+z^2}) dz$$

Divide up into 3 integrals

$$\begin{array}{l} \nearrow \\ \searrow \\ \rightarrow \end{array} \begin{array}{l} \int z^2 dz \\ \int -1 dz \\ \int \frac{1}{1+z^2} dz \end{array}$$

6) Now try each of the branches

→  $\int z^2 dz$  all

1. No easy

2. Is in tree. Stop

→  ~~$\int dz$~~   $\int -1 dz$

1. ~~No~~ easy

~~2.~~

→  $\int dz$

1. No easy

2. In table

→  $\int \frac{1}{1+z^2} dz$

1. No easy

2. In table

3. Is  $\odot$

$\xrightarrow{z = \tan w}$   $\int dw$  ← Is in table, Done

7

Ok done

Program had branch earlier

- but solved it this way

- so forget other branch

## Performance Characteristics

Got 52/54 qv ~~was~~ right on freshmen physics finals

Well what did it get all wrong?

- not interesting here - but usually interesting

Max depth = 7

- ours was 6

Avg depth = 3

Not much is 'integratable' anyway

- only what they give you on finals

Unused rules  $\approx 1$

8

# Conclusion

\* Cat echism

What kind?

How represented?

How used?

How much?

What specifically

\* knowledge is power

About knowledge

\* Intelligence  $\hat{=}$  Mystery

Once you understand how a program works - intelligence vanishes

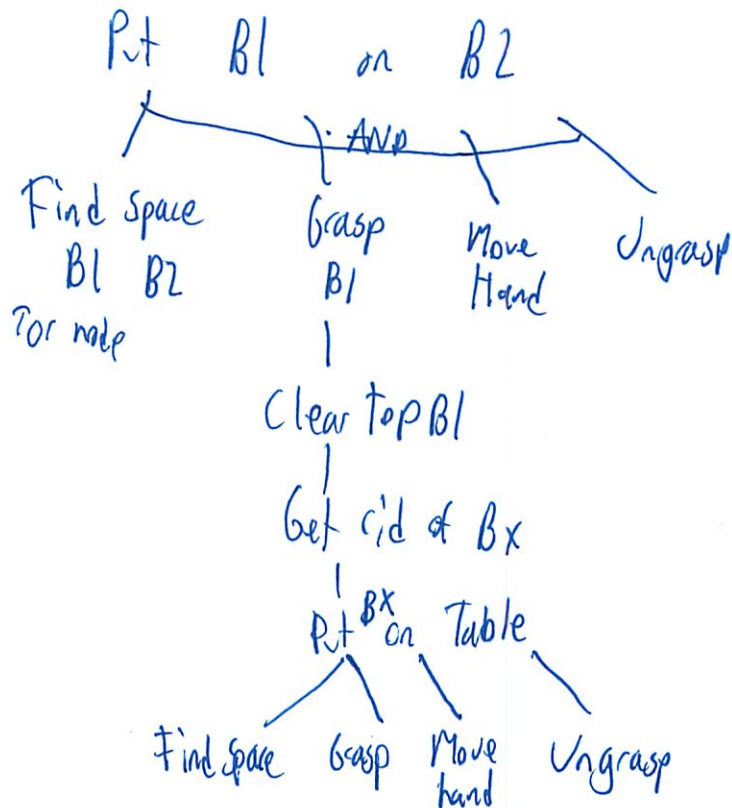
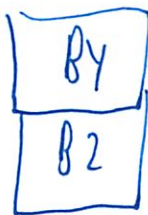
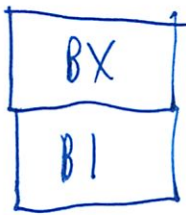
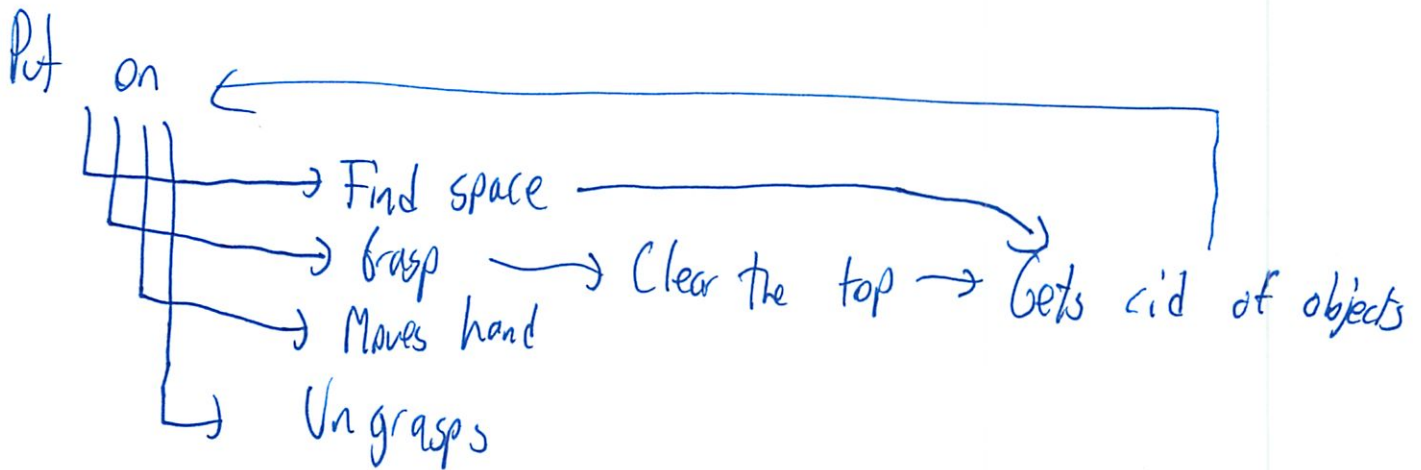
# 6.034 Models of Problem Solving

9/14

(Song: MIT Engineer drinking Song)

Program <sup>he</sup> <sup>grad</sup> <sup>sketch</sup> write that deals w/ blocks stacking

It also answers natural lang qv about ~~the~~ why/how it works



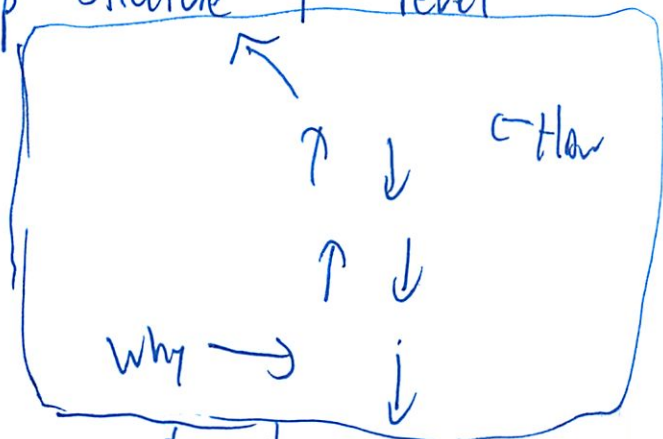
②

It traces its structure

- like we did on board

Then it can answer question

Go up structure | level



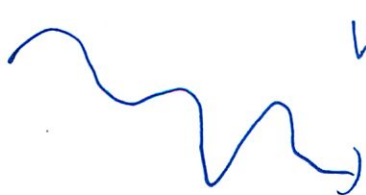
Flow goes the other way - down tree

"That structure" = goal tree  
- has and/or nodes

So because is and/or / goal tree it can answer what it does

$$\text{Complexity (Behavior)} = f(\text{Complexity (program), Complexity (environment)})$$

\* Simon-Zant metaphor of the ant



Watching ant on beach

goes a complex path

but all its doing is avoiding pebbles

3

Maxcin - program at Stanford  
dialog about antibiotics

We don't know about medicine  
So lets do petting zoo.

- has hair
- tells you if mammal

- Sharp teeth
- sharp claws
- forward pointing eyes



Connecting antecedents to consequences, two ways to tell

- eat Meat
- fast
- Spots

Well might be more of a zoo

□ → cheeta



4

Uses rules to reach conclusions

- just like a forward-chaining rule-based deduction system

What to be sure you are correct

- Run the rules backwards

- ask questions about it

Backward-chaining rule-based deduction system

or expert system

Could build a programming system around that

IF A and B are C  
then its d

Program designed housing for planned community

Architect could not tell his vs program's designs

Does it have intelligence?

Not a human

But smart in some way

5

Watch people do what they do

- hard to just ask them

- case might not have come up otherwise

Peas - different if cans or ~~peas~~ frozen

- Now can add rules for frozen

System would not know why put potato chips on bottom

So not as smart as human?

Do any of these rules apply to human intelligence?

Program of understanding Macbeth

Compiled knowledge

- deduce from examples

- or told

- like 'if someone kills someone then that person is dead'

**Recitation 1, Thursday September 15**

---

**Rule-based systems: Forward and Backward Chaining Notes** Prof. Bob Berwick, 32D-728

**Forward Chaining**

You can think of the forward chaining process as that of filtering a (finite) set of *rules* to find the one that is applicable, then firing the rule, i.e., carrying out that rule's consequent, to change the state of the world. The state of the world is represented as a set of database *assertions* (DB), which are statements about what is true in the world. You might want to think about how 'deeply' the rules actually encode the state of knowledge about a particular situation, for example, the grocery bagging rules. How robust or fragile are they? How could you improve them?

General Forward Chaining Pseudo-code

1. For all rules and assertions, find all *matches*, that is, Rule+Assertion combinations. (To do this, you check a rule's variables against all assertions in the DB to find bindings that make all the antecedents of a rule match one or more assertions in the database, thereby making the rule true).
2. Check if any of the matches are *defunct*.  
A *defunct* match is one where the consequents from the match are already in the assertion database.
3. Fire the first non-defunct match & add its consequents to the assertion database. (Note that if there is more than one non-defunct rule, one *must* have some *conflict resolution* method for picking a single rule to fire. Here we have used the order in which rules have matched, following the way they are listed. But one might pick the *last* rule, or the *most recently used* rule, or... what else?)
4. Go to Step 1 and repeat until no more matches fire.

Note 1: If you always choose the first matched rule instance, it's terribly inefficient to compute the entire set of matched rule instances. (Suppose the system has many thousands of rules and assertions. Most of the time, only a few of these will be in play.) If you use a different conflict resolution technique, however, you might very well need to look at the entire list. Here we assume that all the matched rule instances are computed.

Note 2: If rules contain DELETE actions, then assertions may be removed from the DB, so matches in Step 2 may become 'un'-defuncted. This adds greatly to the power of such a system, as we remark below.

**Deduction Rule Systems vs. Production Rule Systems**

Note 3: A deduction system only adds assertions; a production system can both add and delete assertions. (So in class Winston called production systems 'general programming languages' – meaning that they can do anything that a general-purpose computer can do. Why do you think this is?)

A deduction rule system will always converge to the same result except in the case of STOP assertions and infinite loop situations. (STOP assertions generally are not considered part of deduction systems.) What this means is that in a deduction system, you could fire any number of matched rules before re-matching rules with assertions, as long as you have a mechanism for making sure that the same assertion isn't added to the database.

A production rule system will not always converge to the same result if the conflict resolution technique introduces randomness into the order of rule execution. Production rule systems can add explicit STOP assertions, which stop execution of the chainer, or DELETE assertions that would cause other rules to be matched.

A rule with a consequent that stops the chaining is an example of representing knowledge about the problem-solving process itself, a kind of 'meta-knowledge.' You might contrast this sort of rule with ones that represents knowledge about the state of the world. How would you add *learning* to such systems?

## Backward Chaining

For this class we will always assume that our backward chainer is trying to prove the truth of a conclusion, also called a goal or hypothesis. In the process, they construct a so-called AND-OR *goal-tree*.

### General Backward Chaining Pseudo-code

1. First try to find an assertion in the DB that matches the current goal; if one exists, exit with *true*.
2. Otherwise, we want to see what rules can produce the goal, by matching the consequents of those rules against the goal. All the consequents that match are possible options, so we collect the results together in an OR node. If there are no matches, the current goal is a leaf that is *false*. (We might have the system ask the user as to the truth or falsity of the goal in this case, but we don't do that here.)
3. For each matching consequent, keep track of the variables that are bound. Look up the antecedent of that rule, and instantiate those same variables in the antecedent, resulting in a new goal. The antecedent may have AND or OR expressions, and so form an AND-OR node subgoal tree as appropriate.
4. Recursively backward chain on each subtree. (Note that we can short-circuit the evaluation of AND or OR nodes: if any AND subgoal is false, the entire tree of subgoals it is a part of must be false, and the other subgoals do not need to be evaluated; we can return false immediately from this subgoal tree; conversely, if any OR subgoal is true, the entire tree of subgoals it is a part of must be true, and the other subgoals do not need to be evaluated; we can return true immediately from this subgoal tree.)

Note that:

- The backward chainer tries rules in the order they appear in the database of rules.
- Antecedents are tried in the order the antecedents appear in each rule.
- When backward chaining, a NOT clause matches if and only if there is NO matching assertion in the list of assertions, and rules that connect assertions in the list of assertions to the NOT clause. Question: what about rule conflict resolution in this situation?

## Forward vs. Backward Chaining

Many rule systems can chain either forward or backward, but which direction is better? There are several rules-of-thumb.

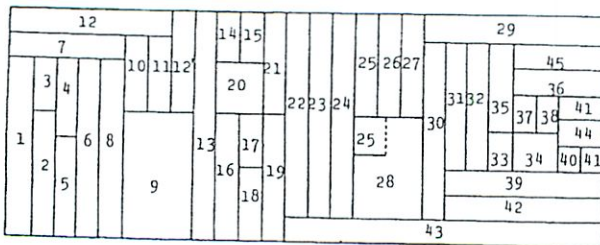
Most importantly, you want to think about how the rules relate facts to conclusions. Whenever the rules are such that a typical set of facts can lead to many conclusions, your rule system exhibits a high degree of "fan-out," and argues for backward chaining. Whenever the rules are such that a typical hypothesis (i.e., a goal to be proven) can lead to many questions, your rule system exhibits a high degree of "fan-in" and argues for forward chaining. However, there are other considerations:

1. If the facts that you have or may establish may lead to a large number of conclusions, but the number of ways to reach the particular conclusion in which you are interested is small, then there is more fan-out than fan-in, and you should typically use backward chaining.
2. If the number of ways to reach the particular conclusion in which you are interested in is large, but the number of conclusions that you are likely to reach is small, then there is more fan-in than fan-out, and you should typically use forward chaining.

These are just rules-of-thumb. In many situations, however, neither fan-out nor fan-in dominates, leading to other considerations:

3. If you have not yet gathered any facts, and you are interested in only whether one of many possible conclusions is true, use backward chaining.
4. If you already have in hand all the facts you are ever going to get, and you want to know everything that can be concluded from those facts, use forward chaining.

## Archeology: the first modern production rule system (circa 1949):



$$S7. \left\{ \begin{array}{l} La_k \\ Lb_k \end{array} \right\} LC_1 (V_1 + LC_1) \rightarrow \left\{ \begin{array}{l} \emptyset \\ La_k + Lb_k + LC_1 \end{array} \right\} (V_1 + La_k + LC_1)$$

## 9/15/11 Rule systems

Duncan Structivist is a protégé of world-renown architect Frank O. Gehry. In order to win over patrons, Duncan designed a rule-based system which takes descriptions of ordinary buildings and generates new descriptions of similar buildings designed with Gehry's distinctive style. Through careful knowledge engineering, Duncan has provided the system with a way to identify candidates for new Gehry masterpieces.

He created the following rule set:

```
P0:
  (IF ((? X) has non-reflective surfaces)
   THEN (Gehrys (? X) cooks pigeons on the sidewalk))
P1:
  (IF ((? X) has many straight lines)
   THEN ((? X) is functional)
         (Gehrys (? X) has twisted tortured forms))
P2:
  (IF (AND ((? X) is functional)
           ((? X) is not inspiring))
   THEN ((? X) is boring))
P3:
  (IF ((? X) is functional)
   THEN (Gehrys (? X) is inspiring))
P4:
  (IF (Gehrys (? X) has twisted tortured forms)
   THEN (Gehrys (? X) is inspiring))
P5:
  (IF (AND (Gehrys (? X) is inspiring)
           (Gehrys (? X) cooks pigeons on the sidewalk))
   THEN (Gehrys (? X) is a masterpiece))
```

A potential patron approached Duncan anonymously with tentative plans for two new structures. Duncan captured the essentials of the plans in the following data set:

```
A1: (Strata-Center has many straight lines)
A2: (Strata-Center has non-reflective surfaces)
A3: (Pickler-Institute has shiny surfaces)
A4: (Pickler-Institute has many straight lines)
```

## Part A: Forward Chaining

### Essential assumptions for forward chaining:

- Assume rule-ordering conflict resolution.
- New assertions are added to the bottom of the data set.
- If a particular rule matches assertions in the database in more than one way, the matches are considered in the order corresponding to the top-to-bottom order of the matched assertions. Thus, if a particular rule's has an antecedent that matches both A1 and A2, the match with A1 is considered first.

Part A.1. Duncan runs a forward chainer on the rules and assertions for 4 steps. (Hand simulation).

Part A.2 In order not to distract would-be patrons with unnecessary details, Duncan makes the following change to rule P3:

```
P3:  
  (IF ((? x) is functional)  
    THEN (Gehrys (? x) is inspiring)  
    DELETE ((? x) is functional))
```

Duncan runs a forward chainer on the rules and data set for 1000 steps. In this part of the problem, however, you consider only the first two steps.

**The data set is repeated on the next page for your convenience.**

In the first step, circle the rules whose antecedents match the initial database:

P0    P1    P2    P3    P4    P5

Also in the first step, circle the rule or rules whose antecedents match the initial database in more than one way.

P0    P1    P2    P3    P4    P5

What new assertion(s) is first added to or deleted from the database in the first step:

In the second step, again circle the rules whose antecedents match the database:

P0      P1    P2    P3    P4    P5

In the second step, circle the rule that actually adds an assertion or deletes an assertion from the database.

P0      P1    P2    P3    P4    P5

What new assertion(s) is added to or deleted from the database in the second step:

### Part A.3

After the forward chaining process started in A.2 has finished, has the assertion:

(Gehrys (? x) is a masterpiece))

been added to the data set for any binding(s) of the variable x? Provide a brief explanation for your answer. If your answer is yes, be sure to include the bindings(s) of the variable x.

The initial data set (repeated for your convenience):

A1: (Strata-Center has many straight lines)

A2: (Strata-Center has non-reflective surfaces)

A3: (Pickler-Institute has shiny surfaces)

A4: (Pickler-Institute has many straight lines)

## Part B: Backward chaining

### Essential assumptions for backward chaining:

- When working on a hypothesis, the backward chainer first tries to find a matching assertion in the database. If no matching assertion is found, the backward chainer tries to find a rule with a matching consequent. If neither a matching assertion nor a matching consequent is found, the backward chainer assumes the hypothesis is false.
- The backward chainer never alters the database, so it can derive the same result multiple times.
- Rules are tried in the order they appear.
- Antecedents are tried in the order they appear.

Simulate backward chaining with the hypothesis

(Gehrys Strata-Center is a masterpiece)

Write all the hypotheses the backward chainer looks for in the database in the order that the hypotheses are looked for. The table has more lines than you need. We recommend that you use the space provided on the next page to draw the goal tree that would be created by backward chaining from this hypothesis. **The goal tree will help us to assign partial credit** in the event you have mistakes in the list.

1
2
3
4
5
6
7
8
9
10

The data set (repeated for your convenience):

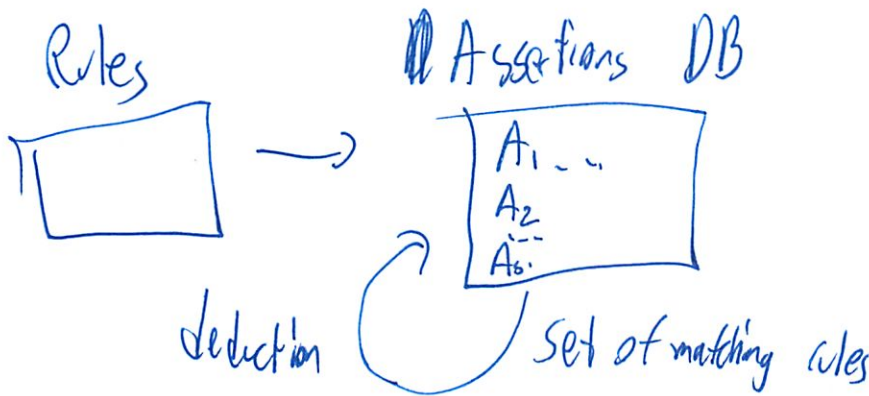
- A1: (Strata-Center has many straight lines)
- A2: (Strata-Center has non-reflective surfaces)
- A3: (Pickler-Institute has shiny surfaces)
- A4: (Pickler-Institute has many straight lines)



(Gehrys Strata-Center is a masterpiece)

Bob Berwick berwick@csail.mit.edu

- What is a RBS?
- Anatomy
- Forward chaining
- Backward chaining



Pf

If  $(L \rightarrow R)$  has non reflective surfaces  $\in$  antecedent  
then Gary cooks pigeons on the sidewalk  $\in$  consequent

Adding Facts - deduction system

2

Step 1 P0 matches b/c assertion A2

since  $x \in SC$   
     $\uparrow$   
    x bound to  
    state center

keep looking if more assertions found

P1 matches (A1)

$x \in SC$

P1 matches (A4)

$x \in PT$

P2 does not match ~~any~~ anything in assertion db

The rest of the rules not true

---

Master 1. Match rules againsts db

Steps 2. Remove all defunct rules

- when the conclusion is already in the db

- want to avoid work

- don't need to run rule

3. Fire the first non-defunct rule.

---

(I missed the point of all this - was 2 min late)

③

G's SC cooks Pidgeons on sidewalk

↳ add as A5 to assertion db

4. Repeat

- match rules again w/ new db - w/ assertion added

P0

P1  $x \in SC$

P1  $x \in PI$

Now select rule to fire

P0 is defunct

Already in db

So continue to next one

Fire next one P1  $x \in SC$

Fire rule means to take conclusions and add to assertion db

Add assertion

A6 SC is functional

A7 G's SC has twisted tailfeathers

4

Step 3 ~~RO~~

~~defunct~~ ~~GR~~  $X \rightarrow SC$   
P1  $X \rightarrow PI$

= Can be optimized  
w/ Rete algorithm

P3  $X \rightarrow SC$   
P4  $X \rightarrow SC$

Now fire P1  $X \rightarrow PI$

A8 PI is functional

A9 PI has twisted tortured forms

So deduced new facts from given ones  
Not really going anywhere

What if we modified a rule?

P3': If (x) is functional

Then Gehrys (?x) is inspiring

Delete (?x) is functional

New: ?  
Can delete  
assertions

5

Why would want to delete facts?

IF conditional, when you get new info, might want to withdraw statements

Can also ~~add~~<sup>have</sup> contradictions easily this way

But a rule can also become undetected  
a rule that used to function on that assertion  
Rule can be undetected  
Now that ~~add~~ can add loops

### Backward Chaining

Forward chaining    If  $\rightarrow$  Then  
Backward chaining    Then  $\rightarrow$  If

So ~~if~~ if we know ~~the~~ Gehery's SC is a masterpiece

Only one rule leads to that conclusion  $\rightarrow$  P5

6

Now look at 'it' part

So we also know

~~G's~~ G's SC is inspiring

G's SC looks ~~like~~ like pigeons on sidewalk  
both since AND A tree

Which rules conclude G's SC is 'inspiring'

P3 - is functional

P4 - has ~~twisted~~ twisted faulted forms

It's one or the other since OR branch

IS functional

→ P1 - G's SC has many straight lines  
↳ this is A1

→ P3 is functional

we already did this

Now ~~go~~ go up and do other half of AND branch

→ P0 - SC has non reflective surfaces

①

So

Masterpiece

← goal condition



G's SC is inspiring

Cooks pickpockets on sidewalk

Emotional

has TTF

has non reflective surfaces

many straight lies

---

2 things if rule not found

1. Always assume false

↳ Closed World Assumption

- not always true

2. Ask the question to the user.

3. Go to a more general theorem system



# Labs

From 6.034 Fall 2011

## Currently released labs

- Lab 0 -- due Friday, September 16 (at 11:59pm)

## The online grader

You will be submitting all of your labs to an online grader. Every lab comes with a file, `tester.py`, that contains the machinery to test your code and to submit it when you're done.

In order for this to work, you need to securely download a "key" that identifies who you are to the grader.

Make sure you have an MIT certificate (<http://ca.mit.edu/>), and go to <https://ai6034.mit.edu:444/fall11/tester/>. This will give you a file called `key.py`. Keep this file secure; for example, don't put it in a publicly-readable Athena directory. ✓

The only thing the grader cares about is whether you pass the tests. It does not care if your code is pretty or well-commented. However, commenting your code can still be important: if you want a TA to help you with your lab, he will be able to give you more help if your code is understandable.

The grader also submits the code to your lab, so that it can be reviewed later by a human. It should go without saying that you should not try to fool or work around the grader, and that the code you submit must be the code you tested. See our grading and collaboration policy, which also explains how your problem set grade is calculated.

Retrieved from "<http://ai6034.mit.edu/fall11/index.php?title=Labs>"

---

- This page was last modified on 10 September 2011, at 20:36.
- *Forsan et haec olim meminisse iuvabit.*

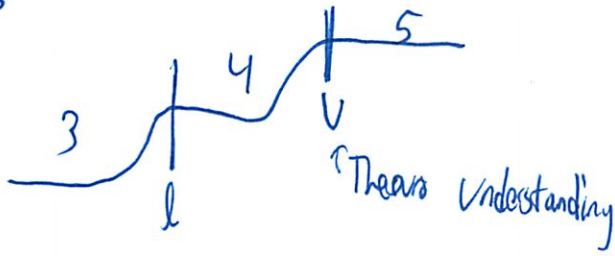
Can resubmit

Mark Seifter

Q <sub>1</sub>	Q <sub>2</sub>	Q <sub>3</sub>	Q <sub>4</sub>	-	Lab + Participation
				X	
					X
F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	F <sub>4</sub>	F <sub>5</sub>	

Take a max of each column, average all columns

Assignments



Not a pure step function

Can get a 4.77

Silver Star Ideas

- not as important as gold star ideas from lecture
- The system is dumber than you
- Rules in order, Assertions in order
- Show your work
- Come to Mega Recitation
- If you want to know, ask

②

## Easy System

Forward chaining is easy - just follow rules,

Backward chaining is hard - have to draw tree

System of 5 rules w/ 3 assertions

When 2 assertions for a rule

Then \_\_\_\_\_ has low taxes

\_\_\_\_\_ gets out of jail

One is added after the other

The backwards chaining if only one is true →

you can use it

Gottlieb's

R<sub>1</sub> If \_\_\_\_\_ business man

Then \_\_\_\_\_ makes million

R<sub>2</sub> If \_\_\_\_\_ in the mob

Then \_\_\_\_\_ low taxes

R<sub>3</sub> If \_\_\_\_\_ low taxes AND \_\_\_\_\_ makes million

Then \_\_\_\_\_ embezzles money

R<sub>4</sub> If \_\_\_\_\_ embezzles \$

Then \_\_\_\_\_ is criminal

③

RS If  $x$  knows  $y$  AND  $x$  makes millions AND  $y$  is in mob  
 Then  $x$  has low taxes  
 $x$  gets out of jail

A1 Gotti is in mob      A3 Cagani knows Gotti  
 A2 Cagani is business man

Back

Gotti is a criminal

First, check if statement is in assertions

No

Next, look in all then cases, are there any that could prove it

Yes R4 - only possible

Still not sure if criminal, need to find if embezzles

Look in assertions

No

Look in rules

R3 - only rule

Now AND

Gotti is criminal

R4

Gotti embezzles

R3

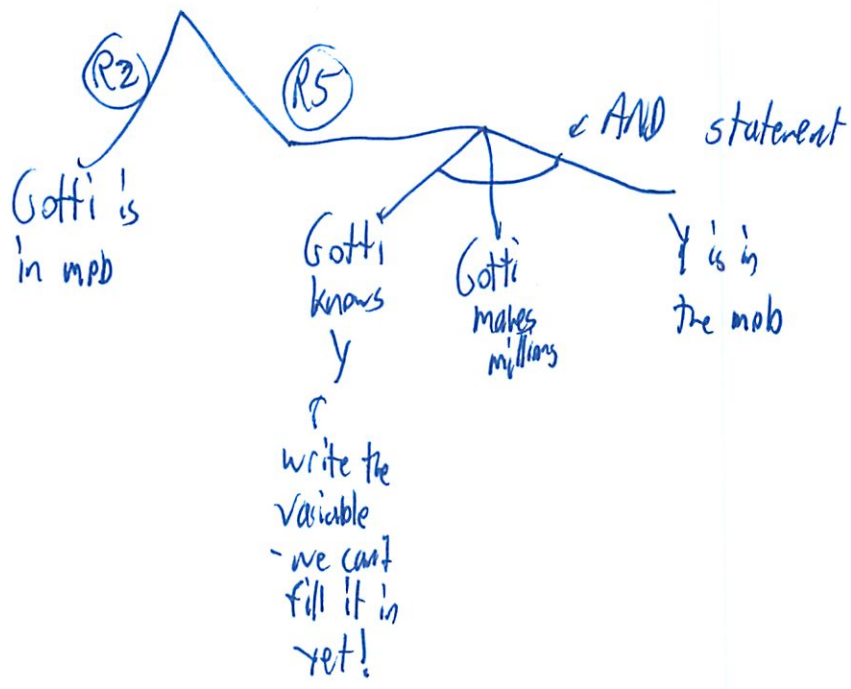
Gotti has low taxes

Gotti makes millions

4

For low taxes 2 possible choices

Gotti has low taxes



Depth first search

So Gotti is in mob is in AI

So don't need to explore R5 since we proved this branch (which is OR)

But still need to resolve Gotti makes millions via AND



(5)

Check assertions  
Check ~~prop~~ rules -No  
-No

Not in there so dead

Then propagate up dead

So we ~~do~~ don't have enough data to 'invite Gotti'

So we should add assertion

AY Gotti is in business man

And then he is invited

For the ~~xx~~ y - looks for assertion 'Gotti knows Lagoni'  
then fills in y w/ 1st possible

Note G knows C  $\neq$  C knows G and makes or branch w/ all people and tries rest  
- on quiz usually only 1

If similar 2nd assertion

- Some systems use memoisation
- assume isn't

6

If Gotti embezzles \$ - we would stop  
and Ry says is criminal (forward)

But not proven - so hard to do the tree

Can't do tree w/ Cagoni and prove Gotti

- since its Cagoni makes millions  
- here no

- I don't know otherwise

∞ loops can happen

- and be on quiz

- what can you determine? nothing!

(Its' a basic in-depth study of procedure/system

- like 6.01, 6.02 ~~was not~~

- unlike 6.005

- was not expecting that)

System will accept

A1 Gotti is not a biz man

A2 Gotti is a biz man

- Uses first one ~~for~~ for forward-chaining (silver star idea)

①

Depth first on large systems works faster

(This makes more sense now

- weird since no lecture to introduce it)
- or did not do reading

Forward-chaining

- always start from top



# Lab 0

## From 6.034 Fall 2011

- **Released:** Saturday, September 10
- **Due:** Friday, September 16

### Contents

- 1 Python
  - 1.1 Getting the lab code
  - 1.2 Getting the Submit Key
  - 1.3 Answering questions
  - 1.4 Run the tester
    - 1.4.1 Using IDLE
    - 1.4.2 Using the command line
- 2 Python programming
  - 2.1 Warm-up stretch
  - 2.2 Expression depth
  - 2.3 Tree reference
- 3 Symbolic algebra
- 4 Survey
- 5 When you're done
- 6 Questions? Issues?

The purpose of this lab is to familiarize you with this term's lab system and to serve as a diagnostic for programming ability and facility with Python. 6.034 uses Python for all of its labs, and you will be called on to understand the functioning of large systems, as well as to write significant pieces of code yourself.

While coding is not, in itself, a focus of this class, artificial intelligence is a hard subject full of subtleties. As such, it is important that you be able to focus on the problems you are solving, rather than the mechanical code necessary to implement the solution.

If Python doesn't come back to you by the end of this lab, we recommend that you seek extra help through the Course 6/HKN tutoring program (<http://hkn.mit.edu/tutor.php>), which matches students who want help with students who've taken and done well in a class. The department pays the tutor, and the program comes highly recommended.

Python resources

Some resources to help you knock the rust off of your Python:

- Any of the many good Python handbooks out there, such as:
  - Dive Into Python (<http://diveintopython.org>), for experienced programmers

- O'Reilly's Learning Python (<http://proquest.safaribooksonline.com/9780596513986/>)
- Think Python (<http://www.greenteapress.com/thinkpython/>) , for beginning programmers
- The standard Python documentation, at [1] (<http://docs.python.org/>) (the Library Reference and the Language Reference are particularly useful, if you know what you're looking for)
- Course 6/HKN tutoring program (<https://hkn.mit.edu/tutoring>)

## Python

There are a number of versions of Python available. This course will use standard Python ("CPython") from <http://www.python.org/>. If you are running Python on your own computer, you should download and install Python 2.5 or Python 2.6 from <http://www.python.org/download/> . All the lab code will require at least version 2.3.

Mac OS X comes with Python 2.3 pre-installed, but the version you can download from python.org has better support for external libraries and a better version of IDLE.

You can run the Python interpreter on Athena like this:

```
add python
idle &
```

You can, of course, edit Python files in a plain-text editor, and run them on Athena like this:

```
add python
python filename.py
```

## Getting the lab code

If you are working on Athena

The code for the labs is in the 6.034 locker. You can get lab 0 like this:

```
attach 6.034
mkdir -p ~/6.034-labs/lab0/
cp -R /mit/6.034/www/labs/lab0/* ~/6.034-labs/lab0/
```

Then, you can edit the code in your ~/6.034-labs/lab0 directory. That way, you won't need to do anything to submit the code - it will already be in the right place.

You can ssh into linux.mit.edu to work on Athena from a different computer (thank you SIPB)

If you are working on another computer with Python

Create a folder for the lab.

Download this file and extract it: <http://web.mit.edu/6.034/www/labs/lab0/lab0.zip>

*ch that is what the locker*

You can also view the code without downloading it: <http://web.mit.edu/6.034/www/labs/lab0/>

## Getting the Submit Key

In order to submit your labs, you must download a "key.py" file and place it in the same directory as your labs. The "key.py" file contains login information used by the tester to identify you personally to the testing server.

You can download a key from <https://ai6034.mit.edu/fall11/tester/>. Make sure that you have an up-to-date MIT Certificate (<http://ca.mit.edu>) before going to this page. Note that the page doesn't currently work in Apple's Safari Web browser (because of a bug in Safari regarding certificates); use Firefox/Chrome instead, or download the file on Athena.

## Answering questions

The main file of this lab is called `lab0.py`. Open that file in IDLE. The file contains a lot of incomplete statements that you need to fill in with your solutions.

The first thing to fill in is a multiple choice question. The answer should be extremely easy. Many labs will begin with some simple multiple choice questions to make sure you're on the right track.

## Run the tester

*More testing knowledge language*

Every lab comes with a file called `tester.py`. This file checks your answers to the lab. For problems that ask you to provide a function, the tester will test your function with several different inputs and see if the output is correct. For multiple choice questions, the tester will tell you if your answer was right. Yes, that means that you never need to submit wrong answers to multiple choice questions.

The tester has two modes: "offline" mode (the default), and "online" or "submit" mode. The former runs some basic, self-contained internal tests on your code. It can be run as many times as you would like. The latter runs some more tests, some of which may be randomly generated, and uploads your code to the 6.034 grader for grading.

You can run the online tester as many times as you want. If your code fails a test, you can submit it and try again. Because you always have the opportunity to fix your bugs, you can only get a 5 on a problem set if it passes all the tests. If your code fails a test, your grade will be 4 or below.

## Using IDLE

If you are using IDLE, or if you do not have easy access to a command line (as on Windows), IDLE can run the tester.

Open the `tester.py` file and run it using Run Module or F5. This will run the offline tests for you. When the offline tests pass (or when you're up against a deadline, or when you have questions for the staff) you can

```
test_online()
```

to submit your code and run the online tests.

In fact, it will run the submission and online test just as soon as you pass the offline tests, saving you a few keystrokes.

You should run the tester (and submit!) early and often. Think of it as being like the "Check" button from 6.01. It makes sure you're not losing points unnecessarily. Submitting your code makes it easy for the staff to look at it and help you.

## Using the command line

If you realize just how much emacs and/or the command line rock, then you can open your operating system's Terminal or Command Prompt, and `cd` to the directory containing the files for Lab 0. Then, run:

```
python tester.py
```

to run the offline tester, and

```
python tester.py submit
```

to submit your code and run the online tester.

You should run the tester (and submit!) early and often. Think of it as being like the "Check" button from 6.01. It makes sure you're not losing points unnecessarily. Submitting your code makes it easy for the staff to look at it and help you.

## Python programming

Now it's time to write some Python.

### Warm-up stretch

Write the following functions:

*I like being back on Python - focus on code implementation - not memory silliness*

- `cube(n)`, which takes in a number and returns its cube. For example, `cube(3) => 27`.
- `factorial(n)`, which takes in a non-negative integer  $n$  and returns  $n!$ , which is the product of the integers from 1 to  $n$ . ( $0! = 1$  by definition.)

We suggest that you should write your functions so that they raise nice clean errors instead of dying messily when the input is invalid. For example, it would be nice if `factorial` rejected negative inputs right away; otherwise, you might loop forever. You can signal an error like this: `raise Exception, "factorial: input must not be negative"`

Error handling doesn't affect your lab grade, but on later problems it might save you some angst when you're trying to track down a bug.

- `count_pattern(pattern, lst)`, which counts the number of times a certain pattern of symbols appears in a list, including overlaps. So `count_pattern(('a', 'b'), ('a', 'b', 'c', 'e', 'b', 'a', 'b', 'f'))` should return 2, and `count_pattern(('a', 'b', 'a'), ('g', 'a', 'b', 'a', 'b', 'a'))` should return 3.

### Expression depth

*So go through letter by letter*

One way to measure the complexity of a mathematical expression is the depth of the expression describing it in Python lists. Write a program that finds the depth of an expression.

For example:

- `depth('x') => 0`
- `depth(('expt', 'x', 2)) => 1`
- `depth(('+', ('expt', 'x', 2), ('expt', 'y', 2))) => 2`
- `depth('/', ('expt', 'x', 5), ('expt', ('-', ('expt', 'x', 2), 1), ('/', 5, 2))) => 4`

*So # lists down*

Note that you can use the built-in Python "`isinstance()`" function to determine whether a variable points to a list of some sort. "`isinstance()`" takes two arguments: the variable to test, and the type (or list of types) to compare it to. For example:

```

>>> x = [1, 2, 3]
>>> y = "hi!"
>>> isinstance(x, (list, tuple))
True
>>> isinstance(y, (list, tuple))
False

```

*← good they give*

### Tree reference

Your job is to write a procedure that is analogous to list referencing, but for trees. This "`tree_ref`" procedure will take a tree and an index, and return the part of the tree (a leaf or a subtree) at that index. For trees, indices will have to be lists of integers. Consider the tree in Figure 1, represented by this Python tuple: `((1, 2), 3), (4, (5, 6)), 7, (8, 9, 10))`

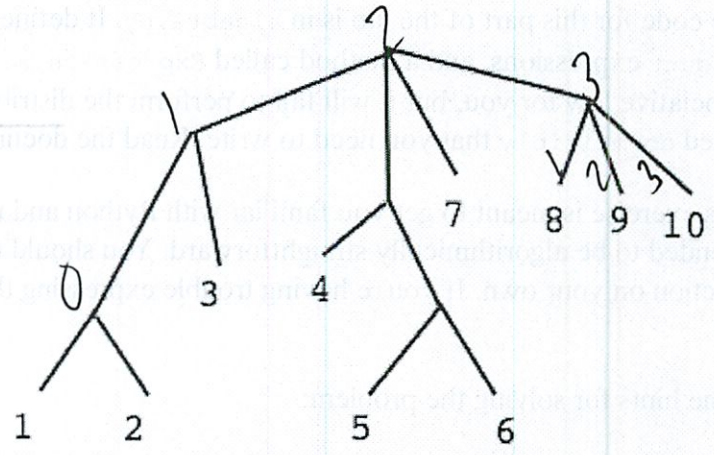


Figure 1: Example Tree

To select the element 9 out of it, we'd normally need to do something like `tree[3][1]`. Instead, we'd prefer to do `tree_ref(tree, (3, 1))` (note that we're using zero-based indexing, as in list-ref, and that the indices come in top-down order; so an index of `(3, 1)` means you should take the fourth branch of the main tree, and then the second branch of that subtree). As another example, the element 6 could be selected by `tree_ref(tree, (1, 1, 1))`.

Note that it's okay for the result to be a subtree, rather than a leaf. So `tree_ref(tree, (0,))` should return `((1, 2), 3)`.

## Symbolic algebra

Throughout the semester, you will need to understand, manipulate, and extend complex algorithms implemented in Python. You may also want to write more functions than we provide in the skeleton file for a lab.

In this problem, you will complete a simple computer algebra system that reduces nested expressions made of sums and products into a **single sum of products**. For example, it turns the expression  $(2 * (x + 1) * (y + 3))$  into  $((2 * x * y) + (2 * x * 3) + (2 * 1 * y) + (2 * 1 * 3))$ . You could choose to simplify further, such as to  $((2 * x * y) + (6 * x) + (2 * y) + 6)$ , but it is not necessary.

This procedure would be one small part of a symbolic math system, such as the integrator presented in Wednesday's lecture. You may want to keep in mind the principle of reducing a complex problem to a simpler one.

An algebraic expression can be simplified in this way by repeatedly applying the associative law and the distributive law.

Associative law

$$((a + b) + c) = (a + (b + c)) = (a + b + c)$$

$$((a * b) * c) = (a * (b * c)) = (a * b * c)$$

Distributive law

$$((a + b) * (c + d)) = ((a * c) + (a * d) + (b * c) + (b * d))$$

The code for this part of the lab is in `algebra.py`. It defines an abstract `Expression` class, `Sum` and `Product` expressions, and a method called `Expression.simplify()`. This method starts by applying the associative law for you, but it will fail to perform the distributive law. For that it delegates to a function called `do_multiply` that you need to write. Read the documentation in the code for more details.

This exercise is meant to get you familiar with Python and using it to solve an interesting problem. It is intended to be algorithmically straightforward. You should try to reason out the logic that you need for this function on your own. If you're having trouble expressing that logic in Python, though, don't hesitate to ask a TA.

Some hints for solving the problem:

- How do you use recursion to make sure that your result is thoroughly simplified?
- In which case should you not recursively call `simplify()`?

## Survey

We are always working to improve the class. Most labs will have at least one survey question at the end to help us with this. Your answers to these questions are purely informational, and will have no impact on your grade.

Please fill in the answers to the following questions in the definitions at the end of `lab0.py`:

- When did you take 6.01?
- How many hours did 6.01 projects take you?

- How well do you feel you learned the material in 6.01?
- How many hours did this lab take you?

## When you're done

Remember to run the tester! The tester will automatically upload your code to the 6.034 server for grading and collection.

## Questions? Issues?

It's quite possible that this lab -- or, in particular, the grader system -- will have issues that need to be fixed or things that need to be clarified.

If you have a question or a bug report, send an e-mail to [6.034tas@csail.mit.edu](mailto:6.034tas@csail.mit.edu) (<mailto:6.034tas@csail.mit.edu>).

Retrieved from "[http://ai6034.mit.edu/fall11/index.php?title=Lab\\_0](http://ai6034.mit.edu/fall11/index.php?title=Lab_0)"

---

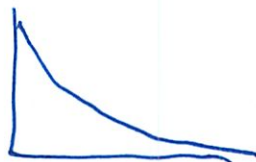
- This page was last modified on 10 September 2011, at 20:41.
- *Forsan et haec olim meminisse iuvabit.*

# Automatic grading

- only cares if correct
- diff test cases
- can always resubmit
- but random test cases

I like their half life ~~late~~ late table / policy

grad. .5  $\rightarrow$   $\frac{1}{7}$

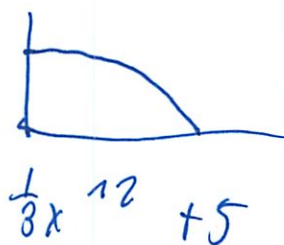


(I need to be more clever 'is knowing what eq represent what)

Always some credit

Or smoother then fall

$\rightarrow$   
approx



What I can't say where  $\frac{1}{3}$  comes from to get it to cross x axis  $\approx 7$

How to make it cross at 7

- 7 would need to be a root

~~to be~~ - current root  $x = 3\sqrt{5}$  since  $\frac{1}{3}$



②

Work backwards when  $x=0$  want eq = 7

$$\cancel{7 = Cx^2 + 5}$$

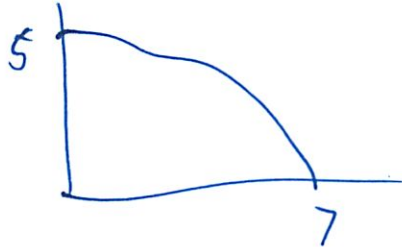
when  $x=7$  when  $y=0$

$$0 = C(7)^2 + 5$$

$$-5 = 49C$$

$$C = \frac{-5}{49}$$

$$\text{So } -\frac{5}{49}x^2 + 5$$



Ok 10 min diversion over

---

~~No test cases~~

? No test cases on cube?

Well in py I can manually try

Also its always cube?

### ③ Factorial

So kinda need to handle errors

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$\text{or } 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$$

↑ range(1, x)

↑ well 2

Be careful!

Good to test!

Can't pattern

↑ only two lists?

---

Oh they don't reload ()

- Sent email that its a good idea

---

test w/ verbosity - 0 hides passed tests

---

depth

recursive: if list tuple - return 1

or otherwise unwrap

- but may not be even - need each branch

4

Need to return max  
always need to think a bit on recursion  
Cool did that fairly fast

---

Oh my patterns thing does not work on ints  
Tells you test cases in verbosity 0 not 1 :)  
Fixed

---

Next Tree reference

returns tree at index

0 based ref

Can do recursive here

again need to iterate over

All I'm getting this!

- starting to think like comp scientist

(remember recursive needs a return/stop phase  
Cool!

Grade up to 4!

5

How do you run it?

I tried running tests - did not get

Focus

So all we need to do is do - multiply

then why is everything explained

can only have two items in

- No!

So simplify does go through one by one ...

Hard to see what doing w/o print access

BUT what does `Product([3])` mean: 3?

Yeah I have no clue here what they want us to do

Somehow they want us to return an expression

- Sometimes not ...

Geon - emailed them

- last time 1.5 hr response time ...

6

I have abs no clue!!!

- So stupid ~~to~~

- they don't tell you what to do

- what they want ...

What should `sum([1, 3]).simplify()` return?

Go online test, says list of fraction

`[*2*, *1*, *3*]` etc

```
# Section 3: Algebraic simplification
```

```
# This code implements a simple computer algebra system, which takes in an
# expression made of nested sums and products, and simplifies it into a
# single sum of products. The goal is described in more detail in the
# problem set writeup.
```

? which referred me here!

```
# Much of this code is already implemented. We provide you with a
# representation for sums and products, and a top-level simplify() function
# which applies the associative law in obvious cases. For example, it
# turns both (a + (b + c)) and ((a + b) + c) into the simpler expression
# (a + b + c).
```

```
# However, the code has a gap in it: it cannot simplify expressions that are
# multiplied together. In interesting cases of this, you will need to apply
# the distributive law.
```

```
# Your goal is to fill in the do_multiply() function so that multiplication
# can be simplified as intended.
```

```
# Testing will be mathematical: If you return a flat list that
# evaluates to the same value as the original expression, you will
# get full credit.
```

I have  
one value returned!

```
# We've already defined the data structures that you'll use to symbolically
# represent these expressions, as two classes called Sum and Product,
# defined below. These classes both descend from the abstract Expression class.
```

```
# The top level function that will be called is the .simplify() method of an
# Expression.
```

```
# >>> expr = Sum([1, Sum([2, 3]])
# >>> expr.simplify()
# Sum([1, 2, 3])
```

```
### Expression classes
```

```
# Expressions will be represented as "Sum()" and "Product()" objects.
# These objects can be treated just like lists (they inherit from the
# "list" class), but you can test for their type using the "isinstance()"
# function. For example:
```

```
# >>> isinstance(Sum([1,2,3]), Sum)
# True
# >>> isinstance(Product([1,2,3]), Product)
# True
# >>> isinstance(Sum([1,2,3]), Expression) # Sums and Products are both Expressions
# True
```

```
class Expression:
```

```
"This abstract class does nothing on its own."
```

```
pass
```

```
class Sum(list, Expression):
```

```
    """
```

```
A Sum acts just like a list in almost all regards, except that this code
can tell it is a Sum using isinstance(), and we add useful methods
such as simplify().
```

```
Because of this:
```

- \* You can index into a sum like a list, as in `term = sum[0]`.
- \* You can iterate over a sum with `"for term in sum:"`.
- \* You can convert a sum to an ordinary list with the `list()` constructor:

```
    the_list = list(the_sum)
```
- \* You can convert an ordinary list to a sum with the `Sum()` constructor:

```
    the_sum = Sum(the_list)
```

```
    """
```

```
def __repr__(self):
```

```
    return "Sum(%s)" % list.__repr__(self)
```

```
def simplify(self):
```

```
    """
```

```
This is the starting point for the task you need to perform. It
removes unnecessary nesting and applies the associative law.
```

```
    """
```

```
    terms = self.flatten()
```

```
    if len(terms) == 1:
```

```
        return simplify_if_possible(terms[0])
```

```
    else:
```

```
        return Sum([simplify_if_possible(term) for term in terms]).flatten()
```

```
def flatten(self):
```

```
    """Simplifies nested sums."""
```

```
    terms = []
```

```
    for term in self:
```

```
        if isinstance(term, Sum):
```

```
            terms += list(term)
```

```
        else:
```

```
            terms.append(term)
```

```
    return Sum(terms)
```

*So is this done?*

```
class Product(list, Expression):
```

```
    """
```

```
See the documentation above for Sum. A Product acts almost exactly
like a list, and can be converted to and from a list when necessary.
```

```
    """
```

```
def __repr__(self):
```

```
    return "Product(%s)" % list.__repr__(self)
```

```
def simplify(self):
```

```
    """
```

To simplify a product, we need to multiply all its factors together while taking things like the distributive law into account. This method calls multiply() repeatedly, leading to the code you will need to write.

```
"""
```

```

factors = []
for factor in self:
    if isinstance(factor, Product):
        factors += list(factor)
    else:
        factors.append(factor)
result = Product([1])
for factor in factors:
    result = multiply(result, simplify_if_possible(factor))
return result.flatten()

```

```

def flatten(self):
    """Simplifies nested products."""
    factors = []
    for factor in self:
        if isinstance(factor, Product):
            factors += list(factor)
        else:
            factors.append(factor)
    return Product(factors)

```

```

def simplify_if_possible(expr):
    """
    A helper function that guards against trying to simplify a non-Expression.
    """
    if isinstance(expr, Expression):
        return expr.simplify()
    else:
        return expr

```

*well parts*

# You may find the following helper functions to be useful.  
 # "multiply" is provided for you; but you will need to write "do\_multiply"  
 # if you would like to use it.

```

def multiply(expr1, expr2):
    """
    This function makes sure that its arguments are represented as either a
    Sum or a Product, and then passes the hard work onto do_multiply.
    """
    # Simple expressions that are not sums or products can be handled
    # in exactly the same way as products -- they just have one thing in them.
    if not isinstance(expr1, Expression): expr1 = Product([expr1])
    if not isinstance(expr2, Expression): expr2 = Product([expr2])
    return do_multiply(expr1, expr2)

```

```
def do_multiply(expr1, expr2):
```



"""

You have two Expressions, and you need to make a simplified expression representing their product. They are guaranteed to be of type Expression -- that is, either Sums or Products -- by the multiply() function that calls this one.

So, you have four cases to deal with:

- \* expr1 is a Sum, and expr2 is a Sum
- \* expr1 is a Sum, and expr2 is a Product
- \* expr1 is a Product, and expr2 is a Sum
- \* expr1 is a Product, and expr2 is a Product

You need to create Sums or Products that represent what you get by applying the algebraic rules of multiplication to these expressions, and simplifying.

Look above for details on the Sum and Product classes. The Python operator '\*' will not help you.

"""

# Replace this with your solution.

raise NotImplementedError

So Sum \* Sum  
 Ok so sum([1, product([3, 1])])

Is 1 + (3 \* 1)

So return what

Sum (1 + 3)

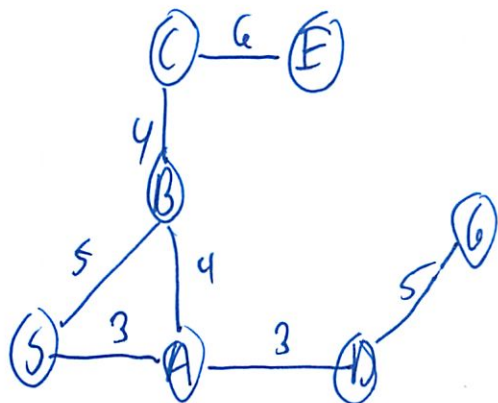
Sum (4) ???

# 6.034 Basic Search

9/12

- One type of intelligence
  - Chess
  - Class scheduling
  - it very good/fast
  - Search is not about maps
  - Search is about choice
- 

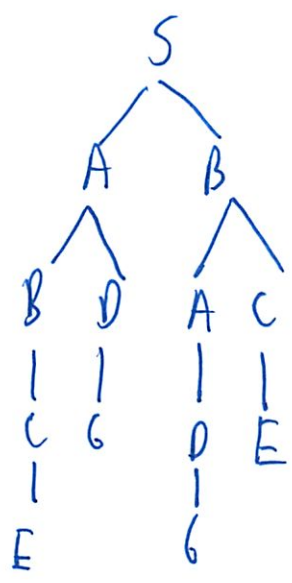
- Can use search on a map
- finding the shortest/optimal path
- don't always want - if just want to find answer once
  - not spend a lot of time searching



②

A human can look at this + quickly think about  
Computers can't

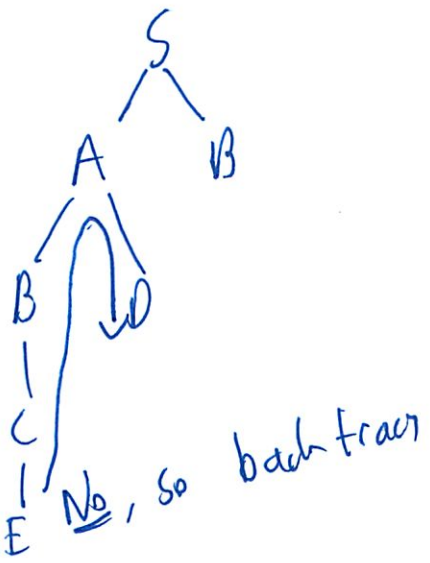
So build a tree, starting at starting pt  
- With all legal, non-repeating/looping paths  
↳ exhaustive



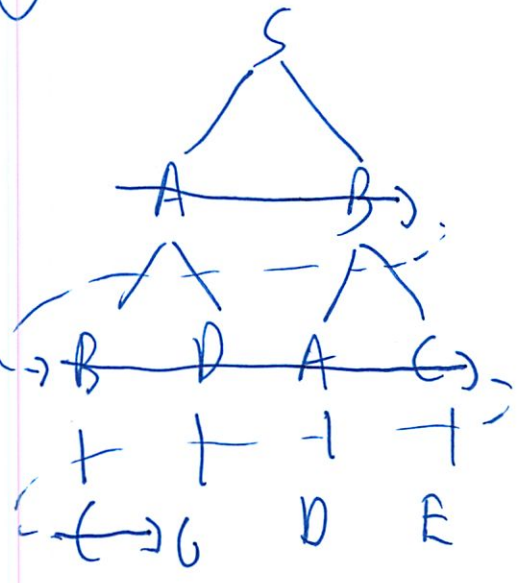
↳ British museum par algorithm

always put lower letter on left

Can do depth-first search



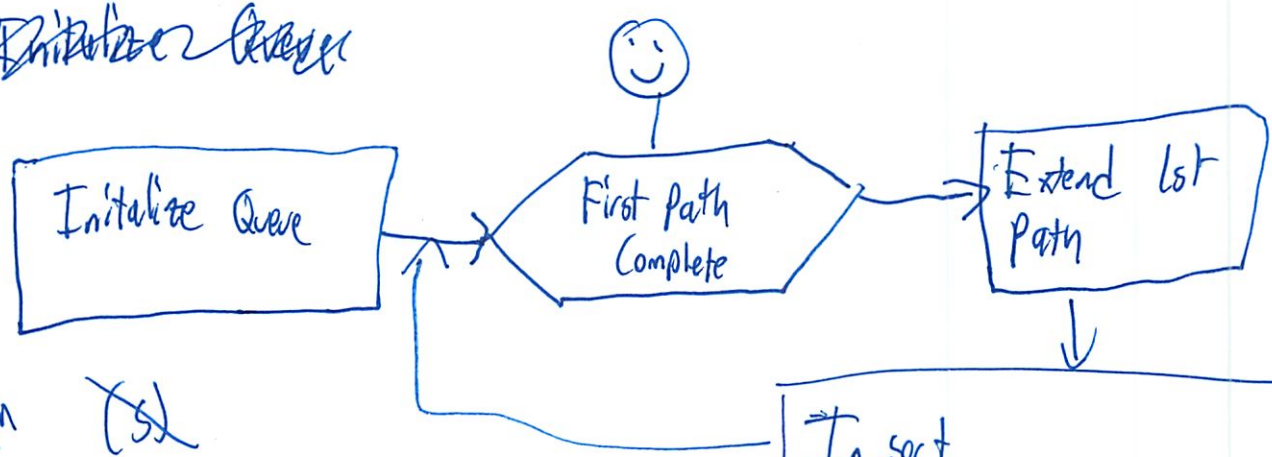
3



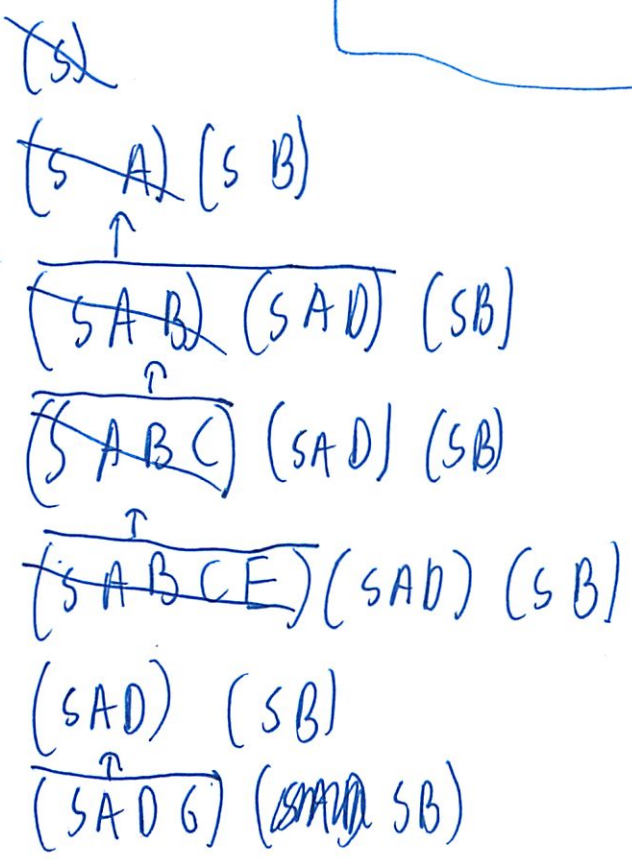
Breadth First  
 - expand everything to uniform level

keep a queue of possible paths

~~Initialize Queue~~



Depth First



In sort

Depth-First - new items in front  
 Breadth-First - new items in back  
 Hill climbing: front, sorted  
 Beam: ... need to sort

4

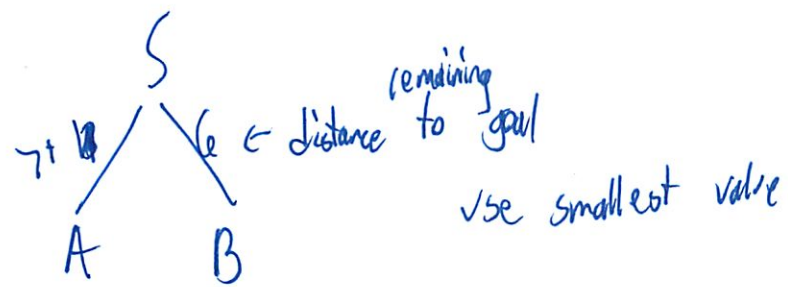
Usually have some idea of how you are doing

- So can make an intelligent choice
- not just a random case

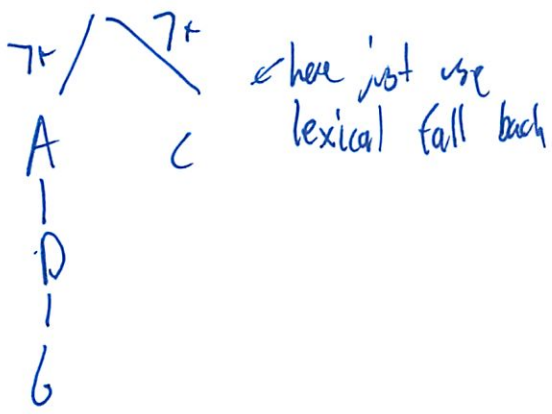
Can measure bird distance from picture  
- a heuristic

No road there, but makes depth-first more interesting

Called hill climbing



Take best from local Agenda



- But,
- might not have a heuristic
  - might be a local maximum
  - get stuck



5

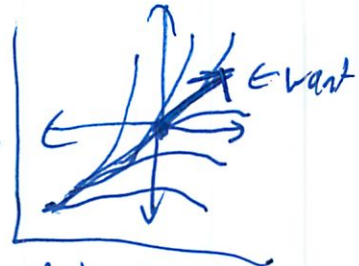
Telephone problem

- wander around can't find type



Ridge problem

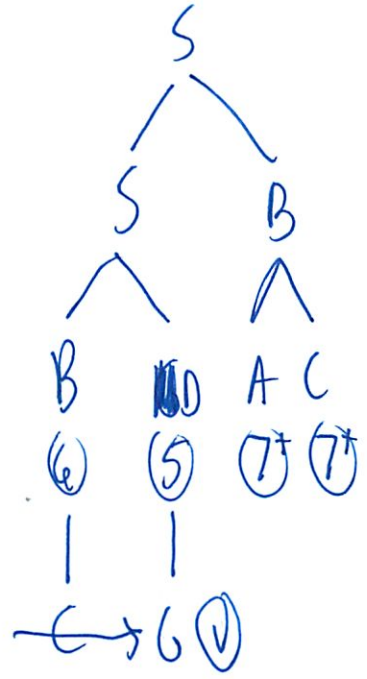
- each direction takes ya down hill  
- need to sample more directions



but try 4  
Cardinal directions  
only

So can we do ~~depth~~<sup>breadth</sup> 1st w/ heuristic too

width = 2



keep only 2 at each level  
(the beam) that get us closest  
to goal

from that level only

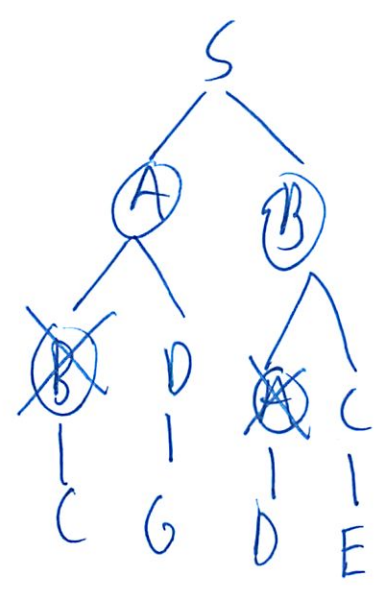
when adding sort within level

6

But notice we've done stupid stuff

Expanded B, A twice

don't



So add to flow chart

If final node, not already extended

	X	✓	Extended list filtering
Depth - First	31	31	
Hill Climbing	12	12	

not worth trouble here

Breadth - First	386	34
-----------------	-----	----

much better!

# Summary

	Backtracking option	Informed not random	Extended list
British Museum	X	X	X
Depth list	✓	X	✓
Breadth list	X	X	✓
Hill Climbing	✓	✓	✓
Beam	✓	✓	✓

Always try to use

Fill out EECS survey

## Gold-star ideas

- \* Search = choice
- \* More k means less S
- \* Why computers make us stupid
  - ↑ Skipped due to time



# 6.034 Tutorial 1

9/18/19

Lab 1 due Friday

Quiz 1 Wednesday

Read chap 7 for description of system

Peter

6 people in this class

---

To do well in ~~the~~ Quizzes just memorize algorithm

Will do practice problems ~~in~~ in tutorial

Started w/ Zoo keeper rule based system

- large group of rules
  - small # of assumptions
- 

## Forward Chaining Pseudo Code

- Until no rule produces a new assertion:
  - For each rule  $i$ 
    - For each set of possible variable bindings
      - instantiate the consequent
      - determine whether the instantiated consequent is already asserted, if not assert it

2

So to do well in 6.034 pretend to be a computer

When asking which rules match look ahead

R0, R3, R4

Before firing any rules

~~RM~~ - which is done in order

After firing rule still matches

(Silly difference)

When 2 variables

When find first one, bind that

A4 Y = Jeremy

X = Hermoine

Then continue looking if true - restart assertion list

If it is not, continue down assertion) <sup>so for each part of if statement,</sup> restart assertion search

A7 Hermine fancies Jeremy

-rebind variables

-research

Example

3

## Backward chaining

look at consequence only  
'it match add hypothesis

If multiple statements lead to same consequent  
then OR free

(This class is easy - half running recitation  
- got hw fairly easily)

---

## Depth First

Will tell you if anything fancy like de-dupe.

## Quiz 1, Problem 1, Rules (50 points)

The administration, worried about the social habits of its students, agrees to finance cross-school-mixers. The 034 TA's decide to fly to England and mix with the students at Hogwarts School of Witchcraft and Wizardry. A merry old time ensues, but the morning after, due to an accidental confundo charm (and perhaps also a large consumption of butterbeer), no one can remember the events that transpired. The 034 staff, in an attempt to show off the power of Muggle logic, promise they can piece together the important events with a rule based system.

Using their keen sense of logic, Matt, Erica, and Mark are able to piece together the following rules:

### **RULES:**

- R0 : IF (?X) goes to MIT,  
THEN (?X) is a muggle,  
      (?X) consumed butterbeer
- R1: IF (?X) made math jokes AND  
      (?X) consumed butterbeer  
THEN (?X) was transfigured into a porcupine
- R2: IF (?Y) fancies (?X) AND  
      (?X) fancies (?Y) AND  
      (?Y) is a muggle  
THEN (?X) snogged (?Y)
- R3: IF (?X) fancies (?Y) AND  
      (?X) made math jokes,  
THEN (?Y) fancies (?X)
- R4: IF (?X) made math jokes  
THEN (?X) goes to MIT

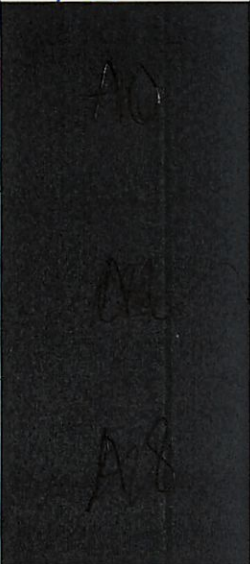
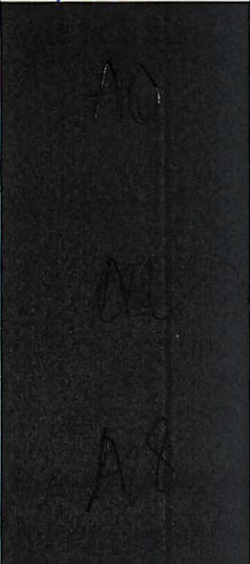
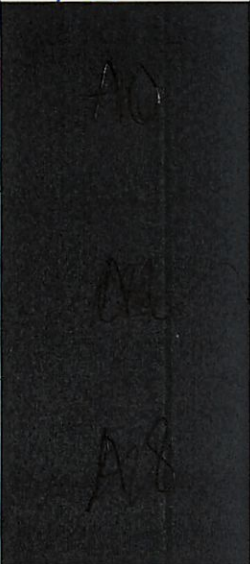
You start with the following list of assertions which is **all you have to go on**.

### **ASSERTIONS:**

- A0: Olga made math jokes  
A1: Yuan goes to MIT  
A2: Jeremy made math jokes  
A3: Hermione consumed butterbeer  
A4: Jeremy fancies Hermione

## Part A: Forward Chaining (24 points)

Run forward chaining on the rules and assertions provided for the first 5 iterations. For the first two iterations, fill out the first two rows in the table below, noting the rules whose antecedents match the data, the rule that fires, and the new assertions that are added by the rule. For the remainder, supply only the fired rules and new assertions. As usual, break ties using the earliest rule on the list that matches. If the earliest rule matches more than once, break ties by assertion order.

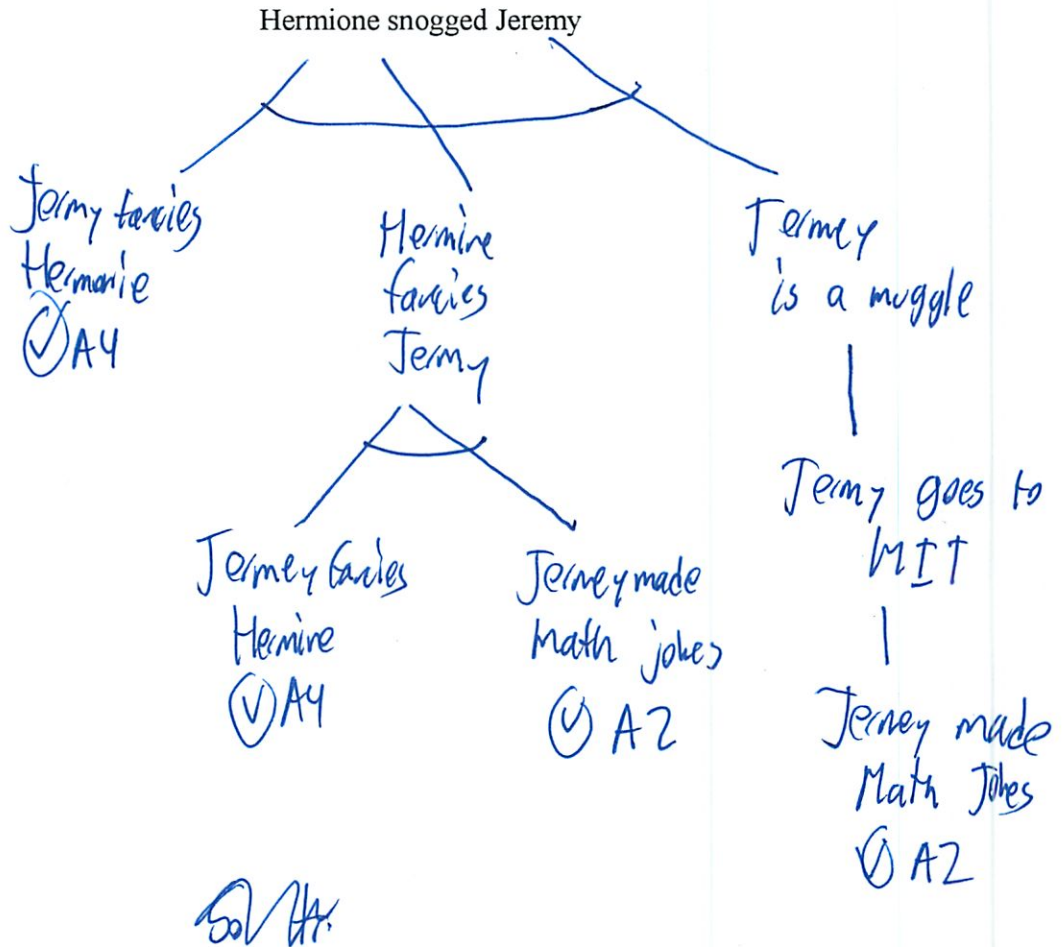
	Matched	Fired	New assertions added to database
1	fill all <del>R0</del> R0, R3, R4	<del>R0</del>	5 Yvan is a muggle ✓ 6 Yvan consumed butter beer
2	<del>R0</del> R0, R3, R4	R3	7 Hermione faces Jeremy ✓
3		R4	8 Olga goes to MIT ✓
4		R4	Jeremy goes to MIT <sup>restart</sup>
5		R0	Olga is a muggle Olga consumed butter beer

R1 Olga was transformed in a Porcupine

stop

## Part B: Backward Chaining (26 points)

Ron Weasley claims that Hermione snogged Jeremy. Use backward chaining to determine if this event occurred. **Draw the goal tree for this statement.** Partial credit will be given for partial completion of the goal tree.



Is the claim that Hermione snogged Jeremy true?

✓ Yes

# Lab 1

9/17 pre review

From 6.034 Fall 2011

## Contents

- 1 Forward chaining
  - 1.1 Explanation
    - 1.1.1 Rule expressions
    - 1.1.2 Running the system
  - 1.2 Multiple choice
  - 1.3 Rule systems
    - 1.3.1 Poker hands
    - 1.3.2 Family relations
- 2 Backward chaining and goal trees
  - 2.1 Goal trees
  - 2.2 Backward chaining
    - 2.2.1 The backward chaining process
    - 2.2.2 Some hints from production.py
- 3 Survey
- 4 Errata

This problem set is due Friday, September 23 at 11:59pm. If you have questions about it, ask the list [6034tas@csail.mit.edu](mailto:6034tas@csail.mit.edu).

To work on this problem set, you will need to get the code, much like you did for Lab 0.

- You can view it at: <http://web.mit.edu/6.034/www/labs/lab1/>
- Download it as a ZIP file: <http://web.mit.edu/6.034/www/labs/lab1/lab1.zip>
- Or, on Athena, attach 6.034 and copy it from `/mit/6.034/www/labs/lab1/`.

Most of your answers belong in the main file `lab1.py`. However, the more involved coding problems in section 2 have their own separate files.

If you successfully submitted `lab0`, then you should copy your `key.py` into the directory containing `lab1` code. You should not need to download a new `key.py`.

You will probably want to use the Python feature called "list comprehensions" at some point in this lab, to apply a function to everything in a list. You can read about them in the official Python tutorial (<http://docs.python.org/tutorial/datastructures.html#list-comprehensions>).

Those who think recursively may also benefit from the Python function `reduce` (the equivalent of Scheme's "fold-left" or "accumulate"), documented in the Python library reference (<http://docs.python.org/lib/built-in-funcs.html#reduce>).

## Forward chaining

### Explanation

This section is an explanation of the system you'll be working with. There aren't any problems to solve. Read it carefully anyway.

This problem set will make use of a production rule system. The system is given a list of rules and a list of data. The rules look for certain things in the data -- these things are the antecedents of the rules -- and usually produce a new piece of data, called the consequent. Rules can also delete existing data.

Importantly, rules can contain variables. This allows a rule to match more than one possible datum. The consequent can contain variables that were bound in the antecedent.

A rule is an expression that contains certain keywords, like IF, THEN, AND, OR, and NOT. An example of a rule looks like this:

```
IF( AND( 'parent (?x) (?y)',
        'parent (?x) (?z)' ),
    THEN( 'sibling (?y) (?z)' ) )
```

This could be taken to mean:

If  $x$  is the parent of  $y$ , and  $x$  is the parent of  $z$ , then  $y$  is the sibling of  $z$ .

Given data that look like 'parent marge bart' and 'parent marge lisa', then, it will produce further data like 'sibling bart lisa'. (It will also produce 'sibling bart bart', which is something that will need to be dealt with.)

Of course, the rule system doesn't know what these arbitrary words "parent" and "sibling" mean! It doesn't even care that they're at the beginning of the expression. The rule could also be written like this:

```
IF (AND( '(?x) is a parent of (?y)',
        '(?x) is a parent of (?z)' ),
    THEN( '(?y) is a sibling of (?z)' ) )
```

*So text is important*

Then it will expect its data to look like 'marge is a parent of lisa'. This gets wordy and involves some unnecessary matching of symbols like 'is' and 'a', and it doesn't help anything for this problem, but we'll write some later rule systems in this English-like way for clarity.

Just remember that the English is for you to understand, not the computer.

**Rule expressions**

*if you could write it*

Here's a more complete description of how the system works.

The rules are given in a specified order, and the system will check each rule in turn: for each rule, it will go through all the data searching for matches to that rule's antecedent, before moving on to the next rule.

*assumptions so antecedents re-searched for each rule*

A rule is an expression that can have an IF antecedent and a THEN consequent. Both of these parts are required. Optionally, a rule can also have a DELETE clause, which specifies some data to delete.

The IF antecedent can contain AND, OR, and NOT expressions. AND requires that multiple statements are matched in the dataset, OR requires that one of multiple statements are matched in the dataset, and NOT requires that a statement is *not* matched in the dataset. AND, OR, and NOT expressions can be nested within each other. When nested like this, these expressions form an AND/OR tree (or really an AND/OR/NOT tree). At the bottom of this tree are strings, possibly with variables in them.

The data are searched for items that match the requirements of the antecedent. Data items that appear earlier in the data take precedence. Each pattern in an AND clause will match the data in order, so that later ones have the variables of the earlier ones.

*does in order*

If there is a NOT clause in the antecedent, the data are searched to make sure that *no* items in the data match the pattern. A NOT clause should not introduce new variables - the matcher won't know what to do with them. Generally, NOT clauses will appear inside an AND clause, and earlier parts of the AND clause will introduce the variables. For example, this clause will match objects that are asserted to be birds, but are not asserted to be penguins:

*in order of and clauses matters*

*show how backward chaining?*

```
AND( '(?x) is a bird',
    NOT( '(?x) is a penguin' ) )
```

The other way around won't work:

```
AND( NOT( '(?x) is a penguin' ), # don't do this!
    '(?x) is a bird' )
```

The terms **match** and **fire** are important. A rule **matches** if its antecedent matches the existing data. A rule that matches can **fire** if its

*fire*



THEN or DELETE clauses change the data. (Otherwise, it fails to fire.)

Only one rule can fire at a time. When a rule successfully fires, the system changes the data appropriately, and then starts again from the first rule. This lets earlier rules take precedence over later ones. (In other systems, the precedence order of rules can be defined differently.)

*60 read rule, scan assumptions, go back to rule 1*  
*→ read rule, scan, match, fire,*

### Running the system

If you from production import forward\_chain, you get a procedure forward\_chain(rules, data, verbose=False) that will make inferences as described above. It returns the final state of its input data.

Here's an example of using it with a very simple rule system:

```
from production import IF, AND, OR, NOT, THEN, DELETE, forward_chain
theft_rule = IF( 'you have (?x)',
                THEN( 'i have (?x)' ),
                DELETE( 'you have (?x)' ) )
data = ( 'you have apple',
         'you have orange',
         'you have pear' )
print forward_chain([theft_rule], data, verbose=True)
```

We provide the system with a list containing a single rule, called theft\_rule, which replaces a datum like 'you have apple' with 'i have apple'. Given the three items of data, it will replace each of them in turn.

Here is the output if you copy-and-pasted the code above and ran it as a python script:

```
Rule: IF(you have (?x), THEN('i have (?x)'))
Added: i have pear
Rule: IF(you have (?x), THEN('i have (?x)'))
Added: i have apple
Rule: IF(you have (?x), THEN('i have (?x)'))
Added: i have orange
('i have apple', 'i have orange', 'i have pear')
```

*out of order*

*e here ships you have pear since old*

NOTE: The Rule: and Added: lines come from the verbose printing. The final output is the set of assertions after applying the forward chaining procedure.

You can look at a much larger example in zookeeper.py, which classifies animals based on their characteristics.

### Multiple choice

Bear the following in mind as you answer the multiple choice questions in lab1.py:

- that the computer doesn't know English, and anything that reads like English is for the user's benefit only
- the difference between a rule having an antecedent that matches, and a rule actually firing

### Rule systems

*? if obsolete*

#### Poker hands

We can use a production system to rank types of poker hands against each other. If we tell it the basic things like 'three-of-a-kind beats two-pair' and 'two-pair beats pair', it should be able to deduce by transitivity that 'three-of-a-kind beats pair'.

Write a one-rule system that ranks poker hands (or anything else, really) transitively, given some of the rankings already. The rankings will all be provided in the form '(?x) beats (?y)'.

Call the one rule you write transitive-rule, so that your list of rules is [ transitive-rule ].

Just for this problem, it is okay if your transitive rule adds 'x beats x', even though in real-life transitivity may not always imply reflexivity.

## Family relations

You will be given data that includes three kinds of statements:

- 'male  $x$ ':  $x$  is male
- 'female  $x$ ':  $x$  is female
- 'parent  $x$   $y$ ':  $x$  is a parent of  $y$

Every person in the data set will be defined to be either male or female.

Your task is to deduce, wherever you can, the following relations:

- 'brother  $x$   $y$ ':  $x$  is the brother of  $y$  (sharing at least one parent)
- 'sister  $x$   $y$ ':  $x$  is the sister of  $y$  (sharing at least one parent)
- 'mother  $x$   $y$ ':  $x$  is the mother of  $y$
- 'father  $x$   $y$ ':  $x$  is the father of  $y$
- 'son  $x$   $y$ ':  $x$  is the son of  $y$
- 'daughter  $x$   $y$ ':  $x$  is the daughter of  $y$
- 'cousin  $x$   $y$ ':  $x$  and  $y$  are cousins (a parent of  $x$  and a parent of  $y$  are siblings)
- 'grandparent  $x$   $y$ ':  $x$  is the grandparent of  $y$
- 'grandchild  $x$   $y$ ':  $x$  is the grandchild of  $y$

You will probably run into the problem that the system wants to conclude that everyone is his or her own sibling. To avoid this, you will probably want to write a rule that adds 'same-identity ( $?x$ ) ( $?x$ )' for every person, and make sure that potential siblings don't have same-identity. (Hint: You can assume that every person will be mentioned in a clause stating his gender (either male or female)). The order of the rules will matter, of course.

*It's not given*

Some relationships are symmetrical, and you need to include them both ways. For example, if  $a$  is a cousin of  $b$ , then  $b$  is a cousin of  $a$ .

As the answer to this problem, you should provide a list called `family-rules` that contains the rules you wrote in order, so it can be plugged into the rule system. We've given you two sets of test data: one for the Simpsons family, and one for the Black family from Harry Potter.

`lab1.py` will automatically define `black_family_cousins` to include all the 'cousin  $x$   $y$ ' relationships you find in the Black family. There should be 14 of them.

NOTE: Make sure you implement all the relationships defined above. In this lab, the online tester will be stricter, and will test some relationships not tested offline.

## Backward chaining and goal trees

### Goal trees

For the next problem, we're going to need a representation of goal trees. Specifically, we want to make trees out of AND and OR nodes, much like the ones that can be in the antecedents of rules. (There won't be any NOT nodes.) They will be represented as `AND()` and `OR()` objects. Note that both 'AND' and 'OR' inherit from the built-in Python type 'list', so you can treat them just like lists.

*like last problem*

Strings will be the leaves of the goal tree. For this problem, the leaf goals will simply be arbitrary symbols or numbers like  $g_1$  or 3.

An **AND node** represents a list of subgoals that are required to complete a particular goal. If all the branches of an AND node succeed, the AND node succeeds. `AND( $g_1$ ,  $g_2$ ,  $g_3$ )` describes a goal that is completed by completing  $g_1$ ,  $g_2$ , and  $g_3$  in order.

An **OR node** is a list of options for how to complete a goal. If any one of the branches of an OR node succeeds, the OR node succeeds. `OR( $g_1$ ,  $g_2$ ,  $g_3$ )` is a goal that you complete by first trying  $g_1$ , then  $g_2$ , then  $g_3$ .

Unconditional success is represented by an AND node with no requirements: `AND()`. Unconditional failure is represented by an OR node with no options: `OR()`.

A problem with goal trees is that you can end up with trees that are described differently but mean exactly the same thing. For example, `AND( $g_1$ , AND( $g_2$ , AND(AND(),  $g_3$ ,  $g_4$ )))` is more reasonably expressed as `AND( $g_1$ ,  $g_2$ ,  $g_3$ ,  $g_4$ )`. So, we've provided you a function that reduces some of these cases to the same tree. We won't change the order of any nodes, but we will prune some

nodes that it is fruitless to check.

We have provided this code for you. You should still understand what it's doing, because you can benefit from its effects. You may want to write code that produces "messy", unsimplified goal trees, because it's easier, and then simplify them with the `simplify` function.

This is how we simplify goal trees:

1. If a node contains another node of the same type, absorb it into the parent node. So `OR(g1, OR(g2, g3), g4)` becomes `OR(g1, g2, g3, g4)`.
2. Any AND node that contains an unconditional failure (OR) has no way to succeed, so replace it with unconditional failure.
3. Any OR node that contains an unconditional success (AND) will always succeed, so replace it with unconditional success.
4. If a node has only one branch, replace it with that branch. `AND(g1)`, `OR(g1)`, and `g1` all represent the same goal.
5. If a node has multiple instances of a variable, replace these with only one instance. `AND(g1, g1, g2)` is the same as `AND(g1, g2)`.

We've provided an abstraction for AND and OR nodes, and a function that simplifies them, in `production.py`. There is nothing for you to code in this section, but please make sure to understand this representation, because you're going to be building goal trees in the next section. Some examples:

```
simplify(OR(1, 2, AND()))           => AND()
simplify(OR(1, 2, AND(3, AND(4)), AND(5))) => OR(1, 2, AND(3, 4), 5)
simplify(AND('g1', AND('g2', AND('g3', AND('g4', AND()))))) => AND('g1', 'g2', 'g3', 'g4')
simplify(AND('g'))                  => 'g'
simplify(AND('g1', 'g1', 'g2'))     => AND('g1', 'g2')
```

## Backward chaining

*Backward chaining* is running a production rule system in reverse. You start with a conclusion, and then you see what statements would lead to it, and test to see if those statements are true.

In this problem, we will do backward chaining by starting from a conclusion, and generating a goal tree of all the statements we may need to test. The leaves of the goal tree will be statements like 'opus swims', meaning that at that point we would need to find out whether we know that Opus swims or not.

We'll run this backward chainer on the ZOOKEEPER system of rules, a simple set of production rules for classifying animals, which you will find in `zookeeper.py`. As an example, here is the goal tree generated for the hypothesis 'opus is a penguin':

```
OR(
  'opus is a penguin',
  AND(
    OR('opus is a bird', 'opus has feathers', AND('opus flies', 'opus lays eggs'))
    'opus does not fly',
    'opus swims',
    'opus has black and white color' ))
```

You will write a procedure, `backchain_to_goal_tree(rules, hypothesis)`, which outputs the goal tree.

The rules you work with will be limited in scope, because general-purpose backward chainer are difficult to write. In particular:

- You will never have to test a hypothesis with unknown variables. All variables that appear in the antecedent will also appear in the consequent.
- All assertions are positive: no rules will have DELETE parts or NOT clauses.
- Antecedents are not nested. Something like `(OR (AND x y) (AND z w))` will not appear in the antecedent parts of rules.

## The backward chaining process

Here's the general idea of backward chaining:

- Given a hypothesis, you want to see what rules can produce it, by matching the consequents of those rules against your hypothesis. All the consequents that match are possible options, so you'll collect their results together in an OR node. If there are no matches, this statement is a leaf, so output it as a leaf of the goal tree.
- If a consequent matches, keep track of the variables that are bound. Look up the antecedent of that rule, and instantiate those same variables in the antecedent (that is, replace the variables with their values). This instantiated antecedent is a new

hypothesis.

*then check it*

- The antecedent may have AND or OR expressions. This means that the goal tree for the antecedent is already partially formed. But you need to check the leaves of that AND-OR tree, and recursively backward chain on them.

Other requirements:

- The branches of the goal tree should be in order: the goal trees for earlier rules should appear before (to the left of) the goal trees for later rules. Intermediate nodes should appear before their expansions.
- The output should be simplified as in the previous problem (you can use the `simplify` function). This way, you can create the goal trees using an unnecessary number of OR nodes, and they will be conglomerated together nicely in the end.
- If two different rules tell you to check the same hypothesis, the goal tree for that hypothesis should be included both times, even though it seems a bit redundant.

*well easier to code*

Some hints from `production.py`

- `match(pattern, datum)` - This attempts to assign values to variables so that *pattern* and *datum* are the same. You can `match(leaf_a, leaf_b)`, and that returns either `None` if `leaf_a` didn't match `leaf_b`, or a set of bindings if it did (even empty bindings: `{}`).
  - Examples:
    - `match("(?x) is a (?y)", "John is a student") => { x: "John", y: "student" }`
    - `match("foo", "bar") => None`
    - `match("foo", "foo") => {}`
    - Note: `{}` and `None` are both false expressions in python, so you should explicitly check if `match`'s return value is None.
- `populate(exp, bindings)` - given an expression with variables in it, look up the values of those variables in *bindings* and replace the variables with their values. You can use the bindings from `match(leaf_a, leaf_b)` with `populate(leaf, bindings)`, which will fill in any free variables using the bindings.
  - Example: `populate("(?x) is a (?y)", { x: "John", y: "student" }) => "John is a student"`
- `rule.antecedent()`: returns the IF part of a rule, which is either a leaf or a `RuleExpression`. `RuleExpressions` act like lists, so you'll need to iterate over them.
- `rule.consequent()`: returns the THEN part of a rule, which is either a leaf or a `RuleExpression`.

## Survey

Please answer these questions at the bottom of your `lab1.py` file:

- How many hours did this problem set take?
- Which parts of this problem set, if any, did you find interesting?
- Which parts of this problem set, if any, did you find boring or tedious?

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

When you're done, run the online tester to submit your code.

*The well parts that were hard but eventually figured out on own*

## Errata

Retrieved from "[http://ai6034.mit.edu/fall11/index.php?title=Lab\\_1](http://ai6034.mit.edu/fall11/index.php?title=Lab_1)"

- This page was last modified on 14 September 2011, at 15:44.
- *Forsan et haec olim meminisse iuvabit.*

data = assumptions

Consequent changes

For # 2 I think yes

Since no line "\_\_\_ is dead"

So statement is true

I see what they are trying to do

- but statement is not there

emailed in

↑  
Oh tester.py has answer

Nothing to instantiate variable w/

Would work for backwards chaining

---

#5: If consequent is already present, rule does  
NOT fire

## ② 1.3.1 Poler

transitive if  $a \rightarrow b$  AND  $b \rightarrow c$   
Then  
 $a \rightarrow c$

So write rule

IF ( AND ( ) , THEN ( ) )

To use in forward chaining

If ( (x beats y) AND (y beats z) )  
Then (x beats z)

Weird syntax ✓

---

## 1.3.2 Family Relationships

So like if ( ~~is~~ parent x y ) And  
parent x z then brother x y

- well add in gender and  
not self

Does it auto parse x, y?

③ It does not seem to dedupe  
Order matters

Grandparent

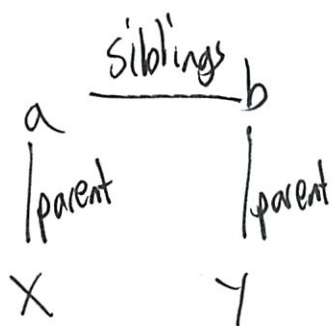
x parent y y parent z

then x grand parent z

~~do not~~  
z grand child x

same time

Cousins



Cousins not wrong

Oh brother twice since both parents

- ignore

Oh no cousins in Simpson

have 14 in black

So code did work

Passes tests offline

4

## Section 7 Backchain

Here is the hard core coding

Good they give how to test examples

~~Make trees~~

Make trees out of AND() OR() objects

Strings are leaves

Have simplification ~~test~~ case function

1. Generate a goal tree w/ all items

• a) See what rules can produce hyp

b) Add all possible to tree w/ OR() branch

c) If consequents match - keep track of variables bound

Must be in order

Include redundant rules

Some functions given to us



(5)

Could it be ~~even~~ recursive?

Where are starting assumptions?

Need to iterate over

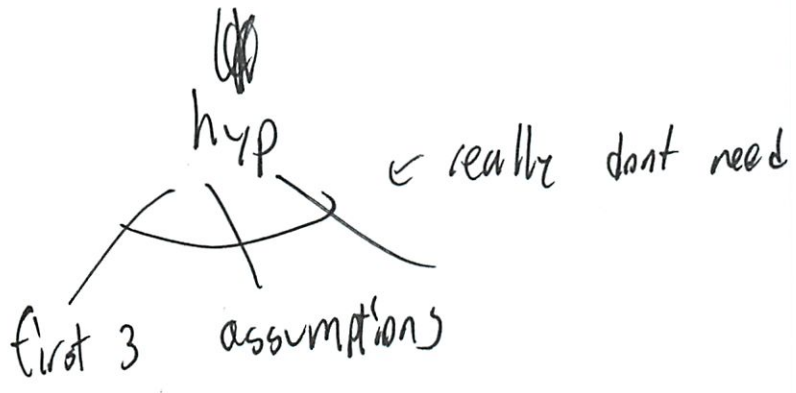
Now need to add item to ~~assumptions~~ tree

sub function

no keep as list

We should have test data for one we did on the

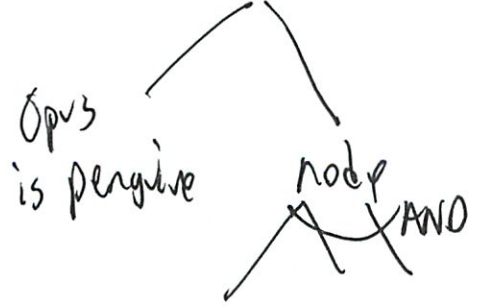
So how would hyp fit on tree



Now

Yeah gets replaced

Can't have text and Node →



(6)

I am putting a lot of extra work in trying to make recursive  
How have rules been modified permanently???

---

Next ~~the~~ which item to take

- be depth first?
- but first unanswered one?

No so this is just build tree

How do we know when to stop?

When item not matched?

? Need to save state we are on?

---

Spent a lot of time doing deep copy

don't know how to make new object  
when object name is a variable

Oh I think this is good now

---

Back to which item to take

? feed in that as hyp

⑧ Think I got - but need to check  
example does not go depth 2/

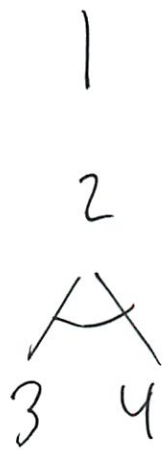
---

A Fixed string issue

---

Not returning correctly  
not inserting the trees correctly when returning

Well for



I return AND (3,4)

So what do they want

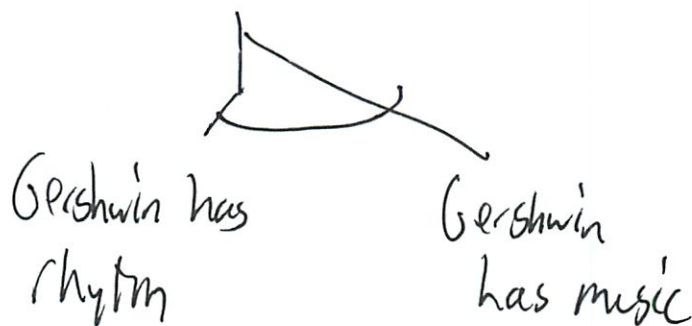
OR(1, 2, AND(3,4))

Try out doing one

~~App~~

Gershwin could not ask for more

Gershwin has rhythm + music



So how do they want us to return

Ok - so I that is what I had before?

Let me think of best way to do

- keep tree

Attach tree part better

So tried adding current hyp to or

- Now we have too many ORs

9

Could not ask

|  
|

So basic notes OR when should be simple

---

Now simplify does not seem to work

Or (or ( --- )) does not simplify

'i was it since it was a string'

(I bet other people solved so much easily

---

Now close but why is simplify not working!

Oh does it mutate or need to save??

And what is Table vs single quote in python

- not really in python

10

So simplify <sup>was not</sup> worked

- That might have some of my earlier problems

- yeah it would

Now for last one just single vs double quote  
in ANDs

Or when return hyp string

So it comes from deep copy!

Oh calling repr

10-15 min

Passed a bunch more!!!

- 1 test offline

- 2 tests online

- Now need to clear printing.

- well fix my fun

(11)

So it is a very abstract one

Why did initial not match?

Oh when not a list?

- no works

- but match failing?

So just say it == or match

Fixed!

14/14 offline!

16/16 online!

```
def backchain_to_goal_tree_4_getargs():
    return [ [ IF( AND( '(?x) has (?y)',
                      '(?x) has (?z)' ),
                THEN( '(?x) has (?y) and (?z)' ) ),
            IF( '(?x) has rhythm and music',
                THEN( '(?x) could not ask for anything more' ) ) ] ],
    Start -> 'gershwin could not ask for anything more' ]
```

```
result_bc_4 = OR('gershwin could not ask for anything more',
                 'gershwin has rhythm and music',
                 AND('gershwin has rhythm',
                     'gershwin has music'))
```

> > > > >

XX

```
let us expand:
gershwin could not ask for anything more
THEN('(?x) has (?y) and (?z)')
THEN('(?x) could not ask for anything more')
match
{'x': 'gershwin'}
(?x) has rhythm and music
-----
```

```
let us expand:
gershwin has rhythm and music
THEN('(?x) has (?y) and (?z)')
match
{'y': 'rhythm', 'x': 'gershwin', 'z': 'music'}
AND('(?x) has (?y)', '(?x) has (?z)')
-----
```

```
let us expand:
'gershwin has rhythm'
THEN('(?x) has (?y) and (?z)')
THEN('(?x) could not ask for anything more')
not found, returning
+++++++
AND("'gershwin has rhythm'", "'(?x) has (?z)'")
+++++++
-----
```

```
let us expand:
'gershwin has music'
THEN('(?x) has (?y) and (?z)')
THEN('(?x) could not ask for anything more')
not found, returning
+++++++
AND("'gershwin has rhythm'", "'gershwin has music'")
+++++++
OR(AND("'gershwin has rhythm'", "'gershwin has music'"))
THEN('(?x) could not ask for anything more')
not found, returning
simplify
OR(AND("'gershwin has rhythm'", "'gershwin has music'"))
```



++++++

OR(AND("'gershwin has rhythm'", "'gershwin has music'"))

++++++

OR(OR(OR(AND("'gershwin has rhythm'", "'gershwin has music'"))))

not found, returning

simplify

OR(OR(OR(AND("'gershwin has rhythm'", "'gershwin has music'"))))

Test 13/14: Incorrect.

—&gt;backchain\_to\_goal\_tree

Got: OR(OR(OR(AND("'gershwin has rhythm'", "'gershwin has music'"))))

Expected: OR('gershwin could not ask for anything more', 'gershwin has rhythm and music',

AND('gershwin has rhythm', 'gershwin has music'))

```

from production import AND, OR, NOT, PASS, FAIL, IF, THEN, match, populate, \
    simplify, variables
from zookeeper import ZOOKEEPER_RULES

# This function, which you need to write, takes in a hypothesis that can be
# determined using a set of rules, and outputs a goal tree of which statements
# it would need to test to prove that hypothesis. Refer to the problem set
# (section 2) for more detailed specifications and examples.

def backchain_to_goal_tree(rules, hypothesis):
    #print '-----'

    #print 'let us expand: '
    #print hypothesis

    #scan through rules for our hypothesis
    newTreeNode = OR()
    #append current hyp
    newTreeNode.append(hypothesis)
    for rule in rules:
        #print rule.consequent()
        for subrule in rule.consequent():
            #print subrule
            if (subrule == hypothesis or match(subrule, hypothesis)):
                #print 'match'
                bindings = match(subrule, hypothesis)
                #print bindings
                #print rule.antecedent()
                #for each rule antecedent, populate it
                i = 0; #can't modify array from rulepart

                #for some reason this is a string in an example
                if (type(rule.antecedent()) == str):
                    ruleAntecedent = OR(rule.antecedent())
                    treePart = OR(rule.antecedent())
                else:
                    #duplicate so can modify
                    ruleAntecedent = rule.antecedent().deepcopy()
                    treePart = rule.antecedent().deepcopy() # each part will be replaced
later

                for rulepart in ruleAntecedent:
                    ruleAntecedent[i] = populate(rulepart, bindings)
                    #try to get further
                    treePart[i] = backchain_to_goal_tree(rules, ruleAntecedent[i])
                    #print '+++++++'
                    #print 'original ' + str(hypothesis)
                    #print treePart
                    #print '+++++++'
                    i = i + 1
                newTreeNode.append(treePart)
            #print newTreeNode

```

```
260 1619 101
# if nothing matches, return hyp back or tree at end
#print 'not found, returning'
# test if just it or more of a node
justHypothesis = OR(hypothesis)
if newTreeNode != justHypothesis:
    #print 'simplify'
    newTreeNode = simplify(newTreeNode)
    #print newTreeNode
    return newTreeNode
else:
    return hypothesis

# Here's an example of running the backward chainer - uncomment it to see it
# work:
#print backchain_to_goal_tree(ZOOKEEPER_RULES, 'opus is a penguin')
```

# 6.034 Recitation 2 Search

9/22

Reviewing search pseudo code

- Use for all problems in class

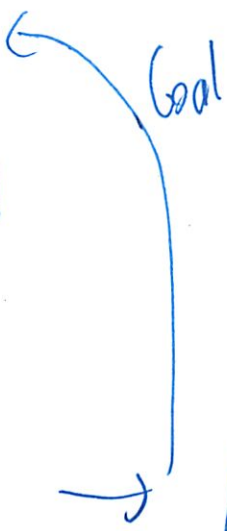
Various types of methods and heuristics

- depth vs breadth-first

Search is not just about graphs

Can be those little ~~puzzles~~ puzzles

1	2	3	4
5	6	7	8
4	10	11	12
13	14	15	



If start w/

1	2	3	4
6	7	8	9
12	13	14	10
15	5	5	14

↑ blank

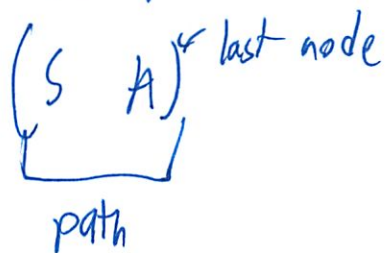


②

So represent as tree where we can move the blank space around ~~then~~ as a tree



All the possible paths to the goal



Agenda manipulating is difference b/w the methods  
(did in 6.01)

See ~~prev~~ pseudo code on handout

- can fail

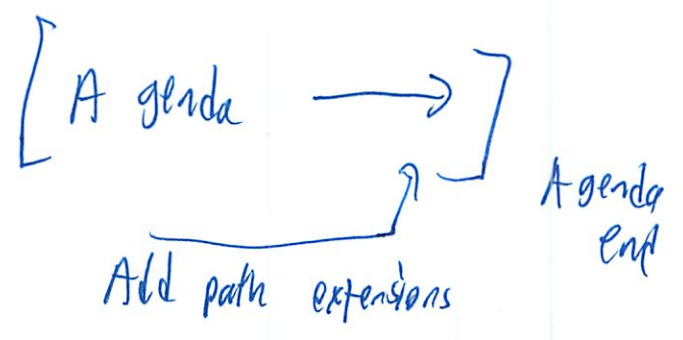
Never add loops in step 3

- not written

Assume bidirectional graph

3

So for example  
- BFS  
- no extended list



(S)

(S A) (S B)

(S B) (S A C) (S A D)

(S A C) (S A D) (S B D) (S B G)

(S A D) (S B D) (S B G) (S A C D)

(S ~~A B D~~) (S B G) (S A C D) (S A D B) (S A D C) (S A D G)

agenda gets big - memory intensive

(S B G) (S A C D) (S A D B) (S A D C) (S A D G) (S B D A) (S B D C)

(S B G) ✓ Done

<sup>24</sup> (S B D G)

BFS



fan-out next states  
"branching factor"

4)

w/ extended list  
it is less work (avoid work)

extended list  
don't expand nodes  
we've tracked  
before

(S) → S  
(SA) (SB) → S, A

(SA C) (SAD) (SBD) (SBC) → S, A, B  
(SA D) (SBD) (SBC) (SACD)

~~(SBD) (SBC)~~, ~~(SACD) (SADG)~~  
? don't expand  
we already did  
(SACD) (SADG)

(SBC) (✓) Done

### Depth First Search

- no extended list

Add paths to front

[ Agenda ]  
↑  
new paths added here

5

Or even smarter, use heuristic

rough guides of how far from goal

can be distances

Or ~~know~~ costs given to you

- guide search so goes on ~~longer~~ <sup>smarter</sup> path

otherwise just pick random

Or use alpha order

Put sorted expansion at front

DFS + Heuristics

(0 S)

(2 SA) (3 SB)

↑ This better to pursue

(1 SAD) (2 SAC) (3 SB)

(0 SADG) (2 SADC) (3 SADB) (3 SB)

(SADG) ✓

Called hill climbing

- at each point decide on next best A

[ Agenda ]

add path expansions sorted!

- so always sorted least cost to most

- pull next cheapest path



6

Heuristic might be bad + lead us astray

Can do for BFS

- by sorting adjacently

- and expanding next best path

L = best first search

So heuristics helps a bit

- Will do more tomorrow w/ heuristic values

- add them at each node

Best search is  $A^*$  find lowest cost/least length

Need to find right heuristic

- all 0s don't help

- too large lead astray

---

15 - Puzzle

- Normal laptop runs out of memory for BFS



$10^{25}$  possibilities

7

Chess  $10^{176}$

Go  $10^{287}$

Measure heuristic by how much it prunes search space

15-puzzle  $10^{25}$

$10^{25}$

↓ heuristic

9,825 nodes

↓ another heuristic

~2000 nodes

What are possible heuristics?

- Manhattan distance

← ~~the~~ very good

- Substitue # ~~14~~ 14 - 5 = 9 ← difference

↑      ↘  
supposed value      current value

- # tiles at of place

- # of pairwise at of place ← best

Could ya build a PC that finds heuristic?

**Recitation 2, Thursday September 22**

Search Me!

Prof. Bob Berwick

**0. Notation, algorithm, terminology**

**graph** is a data structure containing node-to-node connectivity information

**start** is the starting node

**goal** is the target node

Here is a simple generic search algorithm:

```
function search(graph, start, goal)
```

```
0. Initialize
```

```
  agenda = [ [start] ]           # a list of paths
```

```
  extended list = [ ]           # a list of nodes
```

```
  while agenda is not empty:
```

```
    1. path = agenda.pop(0) # remove first element from agenda
```

```
    2. if path is-a-path-to-goal then return path # we've reached the goal
```

```
    3. otherwise extend the current path if last node in this path  
       is not in the extended list:
```

```
      3a. add the last node of the current path to the extended list
```

```
      3b. for each node connected to this last node # look-up using graph
```

```
      3c.   make a new path                               # don't add paths with loops!
```

```
    4. add new paths from step 3c to agenda and reorganize agenda
```

```
  end while
```

```
return nil path # failure
```

**Notes**

- Search returns either a successful path from start to goal or a nil path indicating a failure to find such a path.
- **Agenda** keeps track of all the paths under consideration, and the way it is maintained is *the key* to the difference between most of the search algorithms.
- **Loops in paths:** *Thou shall not create or consider paths with cycles in step 3.*
- **Exiting the search:** Non-optimal searches may actually exit when they find or add a path with a goal node to the agenda (at step 3). But optimal searches *must only* exit when the path is the first removed from the agenda (steps 1, 2).
- **Backtracking:** When we talk about depth-first search (DFS) or DFS variants (like Hill Climbing) we talk about with or without “backtracking”. You can think of backtracking in terms of the agenda. If we make our agenda size 1, then this is equivalent to having no backtracking. Having agenda size  $> 1$  means we have some partial path to go back on, and hence we can backtrack.
- **Extended list (or set):** the list of nodes that have undergone “extension” (step 3). Using an extended list/set is an *optional* optimization that could be applied to all algorithms (some with implications, see A\* search). In the literature the extended list is also referred to as the “closed” or “visited” list, and the agenda the “open” list.
- If we **do not** use an extended list, then the underlined parts above are **not** used.

**Terminology**

Informed vs. uninformed search: A search algorithm is informed if is some evaluation function  $f(x)$  that helps guide the search. Except for breadth-first search (BFS), DFS, and the British Museum algorithm, all the other searches we studied in this class are informed in some way.

Complete vs. incomplete: A search algorithm is complete if, whenever there exists a solution (path from start to goal), then the algorithm will find it.

Optimal vs. Non-optimal: A search algorithm is optimal if the solution found is also the best one (as determined by the path cost)

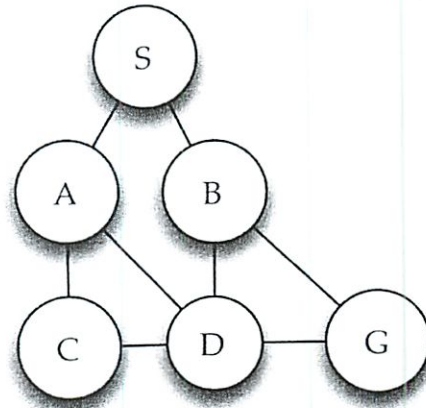
1. Now let's see how this works with the uninformed searches....

Search Algorithm	Properties	Required Parameters	How the agenda is managed in step 4.
Breadth-First Search (BFS)	Uninformed, Non-optimal (Exception: Optimal only if you are counting total path length), Complete		Add all new path extensions to the BACK of the agenda, like a <b>queue</b> (FIFO)
Depth-First Search (DFS)	Uninformed, Non-optimal, Incomplete		Add all new path extensions to the FRONT of the agenda, like a <b>stack</b> (FILO)
British Museum	Brutally exhaustive, Uninformed, Complete		Most likely implemented using a breadth-first enumeration of all paths

Let's try this out with the graph on the right, S= Start node; G= Goal node, for both BFS and DFS....

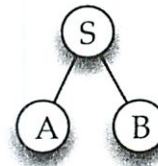
**BFS:** add path extensions to *back* of agenda

	agenda
1	[(S)]
2	[(SA), (SB)]
3	[(SB), (SAC), (SAD)]
4	[(SAC), (SAD), (SBD), (SBG)]
5	[(SAD), (SBD), (SBG), (SACD)]
6	[(SBD), (SBG), (SACD), (SADG)]
7	[(SBG), (SACD), (SADG), (SBDG)]
8	Success - agenda.pop(0) has goal in path, (S B G)



(Note: we could have exited at round 5 here, for non-optimal case.)

Your turn – **DFS:** add extensions to *front* of agenda. You should start by expanding the graph as a tree...



Step	agenda
1	[(S)]
2	[(SA), (SB)]
3	[(SAC), (SAD), (SB)]
4	[(SACD), (SAD), (SB)]
5	[(SACDB), (SACDG), (SAD), (SB)]
6	[(SACDBG), (SACDG), (SAD), (SB)]
7	Success - agenda.pop(0) has goal in path, (SACDBG) [but it is not optimal!]

## 2. Informed search definitions – moving towards optimal search

$f(x)$  = the total cost of the path that your algorithm uses to rank paths.

$g(x)$  = the cost of the path so far.

$h(x)$  = the (under)estimate of the remaining cost to the goal  $g$  node (Use  $h$  for ‘heuristic’)

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

$c(x, y)$  is the *actual* cost to go from node  $x$  to node  $y$ .

“Heuristics, Patient rules of thumb,  
So often scorned: Sloppy, Dumb!  
Yet, Slowly, common sense come” – Ode to AI

Search Algorithm	Properties	Required Parameters	How the agenda is managed in step 4.
Hill Climbing	Non-optimal, Incomplete Like DFS with a heuristic	$f(x)$ to sort newly added paths (usually, this is just $h$ )	1. Keep only <u>newly-added</u> path extensions sorted by $f(x)$ 2. Add sorted <u>new</u> paths to the FRONT of agenda
Best-First Search	Depends on definition of $f(x)$  If $f(x) = h(x)$ (estimated distance to goal) then likely not optimal, and potentially incomplete. However, A* is a type of best-first search that is complete and optimal because of its choice of $f(x)$ which combines $g(x)$ and $h(x)$ (see below)	$f(x)$ to sort the entire agenda by.	Keep <u>entire</u> agenda sorted by $f(x)$

### 2.1 Hill-climbing (with backup)

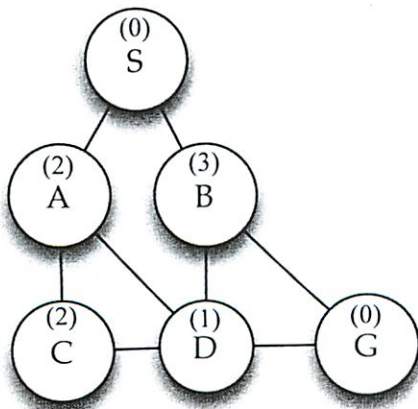
Add path extensions (*sorted by heuristic value*) to the *front* of agenda.

Heuristic value is a measure of the ‘goodness’ of the path, e.g., an *estimate* of how far remaining to go, as the crow flies; or in some other terms if not a map. (We will see this a bit later how to work this into optimal search.)

Note that hill-climbing only looks at the next *locally* best step (*not over all paths!*).

(Below we tack the heuristic value to the front of the list, to keep track; note sorting.)

Our graph now has heuristic values that label each node, in parentheses inside the node:



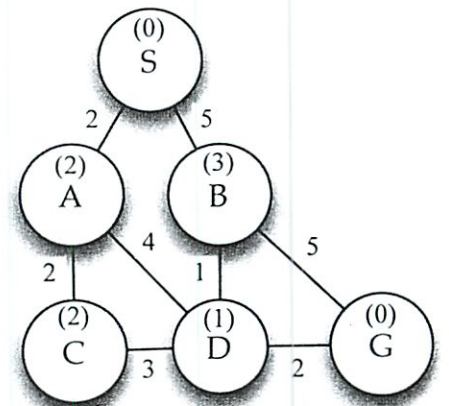
	agenda
1	[(0 S)]
2	[(2 SA), (3 SB)]
3	[(1 SAD), (2 SAC), (3 SB)]
4	[(0 SADG), (2 SADC), (3 SADB), (2 SAC), (3 SB)]
5	Success - agenda.pop(0) has goal in path, (SADG)

## 2.2 Optimal search methods

Search Algorithm	Properties	Required Parameters	How the agenda is managed in step 4.
Branch & Bound (B&B)	Optimal Like best-first except uses actual path costs	$g(x) = c(s, x)$ = the cost of path from $s$ to node $x$ . $f(x) = g(x) + 0$	Sort paths by $f(x) = g(x)$ = total path cost so far)
A* w/o extended list  (or B&B w/o extended list + admissible heuristic)	Optimal if $h$ is <b>admissible</b>	$f(x) = g(x) + h(x, g)$ $h(x, g)$ is the estimate of the cost from $x$ to $g$ . $h(x)$ must be an <b>admissible</b> heuristic	Sort paths by $f(x)$
A* with extended list	Optimal if $h$ is <b>consistent</b>	$f(x) = g(x) + h(x)$ $h(x)$ must be a <b>consistent</b> heuristic	Sort paths by $f(x)$

2.3 Now let's try Branch & Bound, using  $f(x) = g(x) + 0 = g(x)$  = cost of path so far to sort the agenda. We just pay attention to the numbers on the path links, *not* the 'heuristic' numbers in parentheses at each node. Let us also use an extended list.

Step	agenda	Extended
1	[(0 S)]	{ }
2	[(2 S A), (5 S B)]	{S}
3	[(4 S A C), (5 S B), (6 S A D)]	{S, A}
4	[(5 S B), (6 S A D), (7 S A C D)]	{S, A, C}
5	[(6 S A D), (6 S B D), (7 S A C D), (10 S B G)]	{S, A, B, C}
6	[(6 S B D), (7 S A C D), (8 S A D G), (10 S B G)]	{S, A, B, C, D}
7	[(8 S A D G), (10 S B G)]	{S, A, B, C, D, G}
8	Success - agenda.pop(0) has goal in path, (8 S A D G), optimal	{S, A, B, C, D, G}



You can see here how B&B characteristically explores paths in order of *monotonically increasing* path length so far. Note that B&B is really finding the optimal path to each node in the graph. It is not 'biased' in the direction of the goal node. We will need to add in a heuristic function  $h$  to do that.

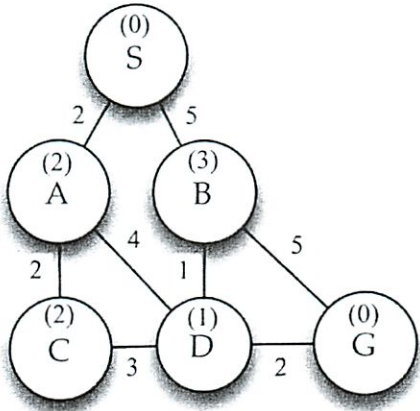
Note also how the extended list comes into play at Step 6 (how?).

(Note also the 'tie' in step 5...what about that? In quizzes you will always be instructed about how to break such ties.)

### 2.4 A\* search = B&B + admissible heuristic

The main idea of A\* is to avoid expanding paths that are already expensive. We use the evaluation function  $f(n)=g(n)+h(n)$ . We sort the entire agenda by this value, and pick the best path to work on next.

OK, let's try this. Now for  $f$  at a node we compute the **sum** of the path-length-so-far **plus** the  $h$  value at that node, the value in parentheses. For instance, the  $f$  value at node  $A$ , given that we start from  $S$ , is  $2+2 = 4$ ; for node  $B$  it is  $5+3=8$ . We also use an extended list.

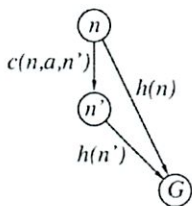


Step	Agenda	Extended
1	[(0 S)]	{ }
2	[(4 S A), (8 S B)]	{S}
3	[(6 S A C), (7 S A D), (8 S B)]	{S, A}
4	[(7 S A D), (8 S B), (8 S A C D)]	{S, A, C}
5	[(8 S B), (8 S A C D), (8 S A D G), (10 S A D B)]	{S, A, C, D}
6	[(7 S B D), (8 S A C D), (8 S A D G), (10 S A D B), (10 S B G)]	{S, A, B, C, D}
7	[(8 S A D G), (10 S A D B), (10 S B G)] Success! (8 S A D G)	{S, A, B, C, D, G}

Note that if the heuristic values at  $S$  and  $D$  were  $S=10$  and  $D = 4$ , these would be inadmissible because, e.g., 4 is an overestimate of the remaining distance to the goal, 2. So the  $h$  value at  $D$  must be  $\leq 2$ , and similarly the  $h$  value at  $S$  must be  $\leq 8$  to be admissible (in fact, for all node values,  $h \leq 8$ ). Why is this important? Suppose an  $h$  value is inadmissible, say,  $10^6$  at some node. Then A\* could fail: a path through that node will never get worked on, even though the *actual* path length through that node might be the optimal one. Admissibility is a constraint that must hold between *every* node and the goal node. There is another, stronger constraint that is sometimes easier to check, that implies admissibility, namely, *consistency*, which amounts to the triangle inequality. This ensures that  $f(n)$  is non-decreasing along *any* path, and it *must* hold if we are using A\* with an extended list, as we will see below. (However, admissibility does **not** imply consistency, so this is not a bi-conditional.)

**Definition:** A heuristic is *consistent* if, for every node  $n$ , every successor node  $n'$  of  $n$  satisfies the following condition:

$$h(n) \leq c(n,a,n') + h(n')$$



So if  $h$  is *consistent*, we have:

$$\begin{aligned} f(n') &= g(n') + h(n') && \text{[by dfn of } f\text{]} \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) = f(n) && \text{[substituting for dfn of consistent } h \text{ \& dfn of } f\text{]} \end{aligned}$$

So  $f(n)$  is **non-decreasing along any path**. This is the same condition that B&B obeyed (since it uses actual costs or path values it holds for B&B).

**Question:** is the search graph above *consistent*? (Hint: Look at paths from  $C$  to node  $D$  and calculate the  $f$  values, to see if they are non-decreasing, or look at what happened between Steps 5 and 6 above.) If a graph is *inconsistent* and we are using A\* with an extended list, then A\* might fail: consider what would happen above if the  $S$ - $B$  link were of length 4 instead of 5.

### Properties of A\*

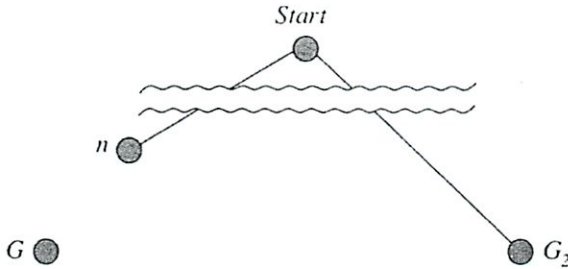
1. A\* is complete unless there are infinitely many nodes s.t.  $f \leq f(G)$
2. Time: Exponential in [relative error in  $h$  x length of solution path]
3. Space: Keeps all nodes in memory (the dark side of A\*, usually runs out of memory)
4. Optimal: Yes, cannot expand  $f_{i+1}$  until  $f_i$  is finished

A\* expands all nodes with  $f(n) < C^*$ , where  $C^*$  is the optimum cost/distance

A\* expands some nodes with  $f(n) = C^*$

A\* expands no nodes with  $f(n) > C^*$

### 3.1 Enrichment portion 1: Optimality of A\*



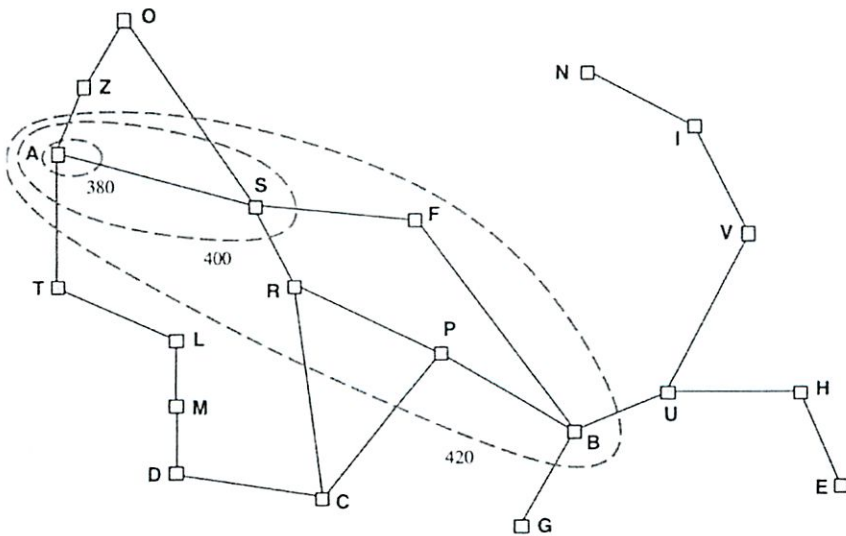
Suppose the algorithm generates some suboptimal goal  $G_2$  and is in the fringe, as in the picture. Let  $n$  be an unexpanded node in the fringe such that  $n$  is on a shortest path to the optimal goal  $G$ . Then:

- (1)  $f(G_2) = g(G_2)$  since  $h(G_2) = 0$
- (2)  $g(G_2) > g(G)$  since  $G_2$  is suboptimal
- (3)  $f(G) = g(G)$  since  $h(G) = 0$
- (4)  $f(G_2) > f(G)$  from (1), (2), (3)
- (5)  $h(n) \leq h^*(n)$  since  $h$  is admissible
- (6)  $g(n) + h(n) \leq g(n) + h^*(n)$
- (7)  $f(n) \leq f(G)$  by defn of  $f(G)$  as  $g(n) + h^*(n)$

But then:

- (8)  $f(G_2) > f(n)$  by (4) and (7), so A\* will never select  $G_2$  for expansion. QED.

Another picture, possibly more helpful (see properties of A\*). A\* expands in terms of increasing  $f$  values (like B&B), directed along contours 'pointing' towards the goal.





### 3.2 Enrichment 2: Cost and Performance of Various Search Strategies; Iterative Deepening Search

(branching factor =  $b$ , depth =  $d$ )

Worst case time = proportional to # nodes visited

Worst case space = proportional to maximum length of  $Q$

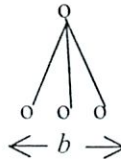
Search Strategy	Worst Time	Worst Space	Fewest Nodes?	Guaranteed to find path?
Depth-first (with backup)	$b^{d+1}$	$bd$	No	Yes
Breadth-first	$b^{d+1}$	$b^d$	Yes	Yes
Hill-Climbing (no backup)	$d$	$b$	No	No
Hill-Climbing (with backup)	$b^{d+1}$	$bd$	No	Yes
Best-first	$b^{d+1}$	$b^d$	No	Yes
Beam (beam width $k$ , no backup)	$kd$	$kb$	No	No

How could we combine the space efficiency of DFS with BFS? (BFS guaranteed to find path to goal with minimum number of nodes.) Answer: Iterative Deepening Search (IDS) – search DFS, level by level, until we run out of time. Let's see.

#### Counting Nodes in a Tree

Why is  $(b^{d+1} - 1)/(b - 1)$  the number of nodes in a tree? (branching factor =  $b$ , depth =  $d$ )

If each node has  $b$  immediate descendents:



Then Level 0 (the root) has 1 node.

Level 1 has  $b$  nodes.

Level 2 has  $b * b = b^2$  nodes.

Level 3 has  $b^2 * b = b^3$  nodes.

...

Level  $d$  has  $b^{d-1} * b = b^d$  nodes.

So the total number of nodes is:

$$N = 1 + b + b^2 + b^3 + b^4 + \dots + b^d$$

$$bN = b + b^2 + b^3 + b^4 + \dots + b^d + b^{d+1}$$

Subtracting:

$$(b - 1)N = b^{d+1} - 1$$

$$N = \frac{b^{d+1} - 1}{b - 1}$$

So we could do this to implement Iterative Deepening Search (IDS):

1: Initialize  $D_{max} = 1$ . (The goal node is of unknown depth  $d$ )

2: **Do**

3: DFS from  $S$  for fixed depth  $D_{max}$

4: **If** found a goal node, depth  $d \leq D_{max}$  **then exit**

5:  $D_{max} = D_{max} + 1$

Cost is:  $O(b^1 + b^2 + \dots + b^L) = O(b^L)$  where  $L$  = length to goal. But isn't IDS wasteful because we repeat searches on the different iterations? No. For example, suppose  $b=10$  and  $d=5$ . Then the total # number of nodes  $N$  we look at for in each case is:

$N(\text{IDS}) \approx db + (d-1)b^2 + \dots + b^5 = 123,450$ , while for BFS the # of nodes is approximately,

$N(\text{BFS}) \approx b + b^2 + \dots + b^d = 111,110$ , or only about 10% less. Most of the time is spent at depth  $d$ . So, IDS is asymptotically optimal; because 'most' of the time is spent in the fringe of the search tree. It is the preferred method over BFS, DFS when the goal depth is unknown.

Similarly, for A\* search, in order to avoid HUGE memory costs, one will often use IDA\*, i.e., iterative deepening A\*.

#### 4. The value of good heuristics: the 8 puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

What is a 'legal move;?

What would be a good heuristic  $h$  for this puzzle? Note that even IDS search is costly; if # tiles is 14, then IDS typically searches 3,473,941 nodes. If # tiles is 24, then this is about 54,000,000,000 nodes.

Two suggested heuristics,  $h_1=7$ ;  $h_2= ????$

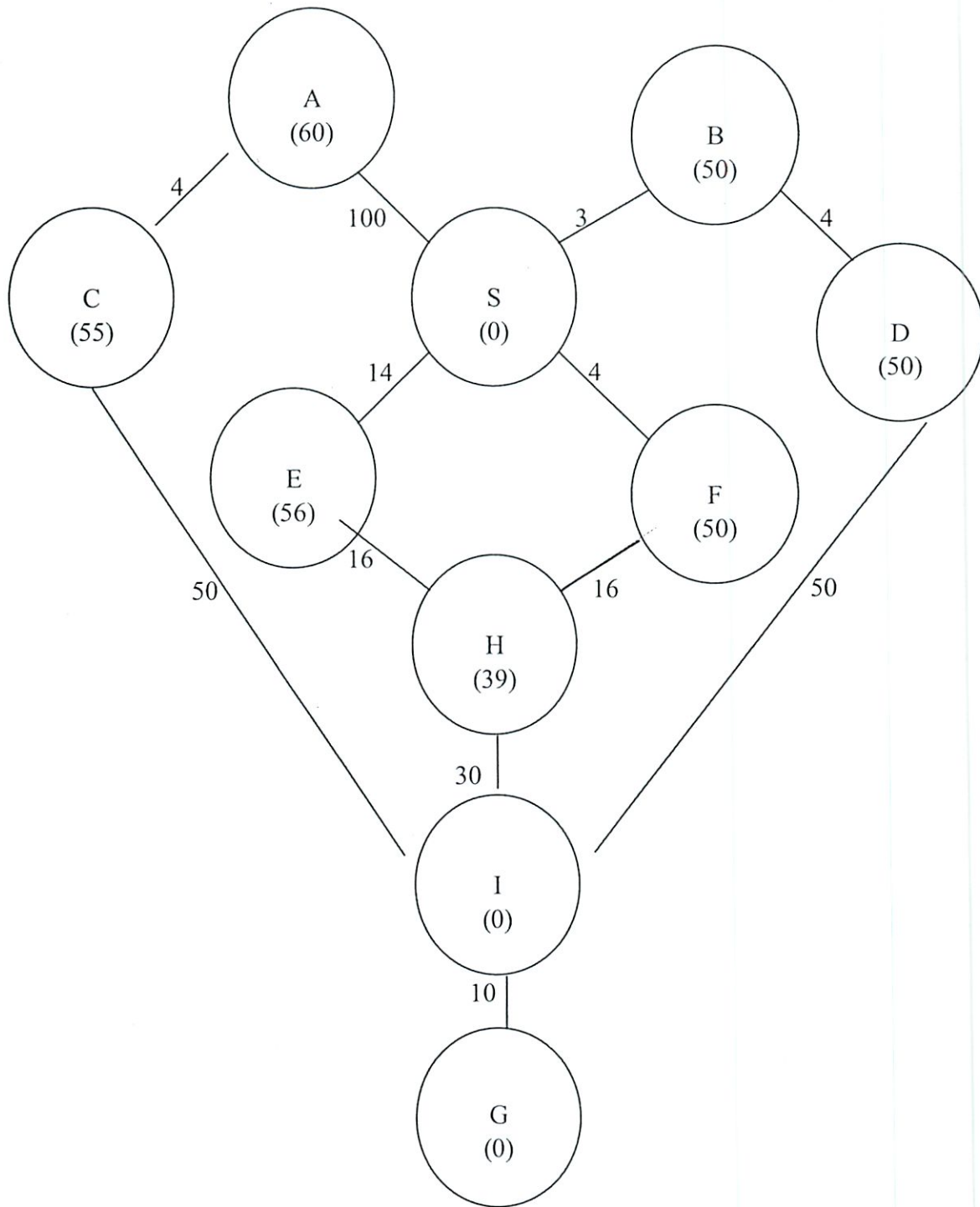
The first is called:

The second is called:

Question: can you guess what happens to the efficiency of search if it's always the case that  $h_2(n) \geq h_1(n)$  for all  $n$ ? (Both heuristics admissible). Why do you think this? What does this say about *how* to choose a heuristic?

## 9/22/11 Optimal Search

Now that Mark has his new stronghold, he wants to invade parallel universes. So Mark programs his evil supercomputer to find the shortest path of jumps from his starting universe S to his goal universe G.



## Part B1 Branch & Bound search

First, Mark programs a simple **branch-and-bound search with an extended list**. As usual, he breaks ties of equal length in lexicographic order. List the nodes Mark's computer adds to the extended list, in order. Distances are shown next to edges. Ignore the numbers in parentheses for this part of the problem. Extra space is provided below in case you want to show your work.

Remember: for B&B,  $f(n)=g(n)$  (total path length so far).

What path does Mark's computer find?

### Part B2 A\* search

Frustrated by branch-and-bound's speed, Mark reprograms his computer to use  $A^*$ . Mark counts the number of subspace anomalies between each universe and the goal and uses this count as the **heuristic** for  $A^*$  (**these are the numbers in parentheses**). List the nodes Mark's computer adds to the extended list, in order. Extra space is provided below in case you want to show your work.

Remember that now  $f(n) = g(n) + h(n)$ . We have done the calculation of the first node  $f$  values 1 step away from  $S$  for you, to start. Use the page after this one and tear it off.

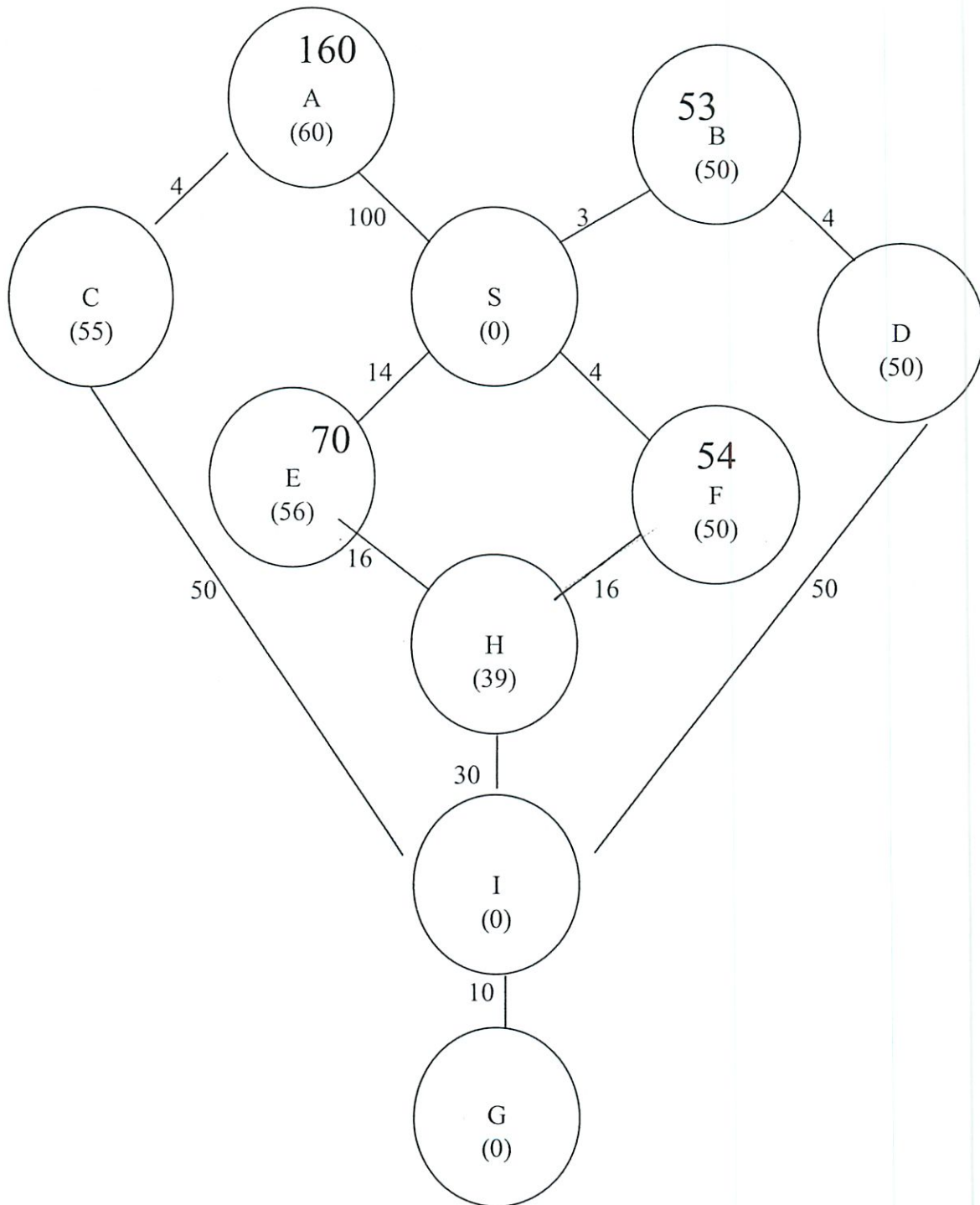
What path does Mark's computer find now?

### Part B3 (5 points)

Mark is confused. Give a **brief** but **specific** explanation of what happened and why.

## 9/22/11 Optimal Search

Now that Mark has his new stronghold, he wants to invade parallel universes. So Mark programs his evil supercomputer to find the shortest path of jumps from his starting universe S to his goal universe G.



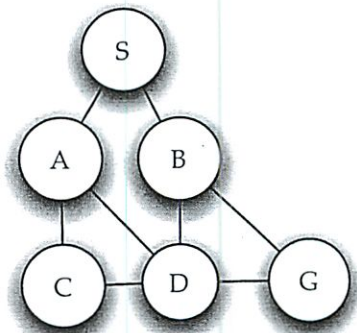
1. Now let's see how this works with the uninformed searches....

Search Algorithm	Properties	Required Parameters	How the agenda is managed in step 4.
Breadth-First Search (BFS)	Uninformed, Non-optimal (Exception: Optimal only if you are counting total path length), Complete		Add all new path extensions to the BACK of the agenda, like a queue (FIFO)
Depth-First Search (DFS)	Uninformed, Non-optimal, Incomplete		Add all new path extensions to the FRONT of the agenda, like a stack (FILO)
British Museum	Brutally exhaustive, Uninformed, Complete		Most likely implemented using a breadth-first enumeration of all paths

Let's try this out with the graph on the right, S= Start node; G= Goal node, for both BFS and DFS....

(NOTE REVISED) BFS: add path extensions to *back* of agenda

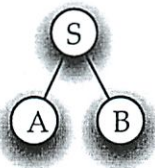
Step	agenda
1	[(S)]
2	[(SA), (SB)]
3	[(SB), (SAC), (SAD)]
4	[(SAC), (SAD), (SBD), (SBG)]
5	[(SAD), (SBD), (SBG), (SACD)]
6	[(SBD), (SBG), (SACD), (SADB), (SADC), (SADG)]
7	[(SBG), (SACD), (SADB), (SADC), (SADG), (SBDG), (SBDG)]
8	Success - agenda.pop(0) has goal in path, (S B G)



(Note: we could have exited at Step 5 here, for non-optimal case, but for uniformity, exit as per code.)

Does adding an extended\_list change anything in this example? (We will try it.)

Your turn – DFS: add extensions to *front* of agenda. You should start by expanding the graph as a tree...



Step	agenda
1	[(S)]
2	[(SA), (SB)]
3	[(SAC), (SAD), (SB)]
4	[(SACD), (SAD), (SB)]
5	[(SACDB), (SACDG), (SAD), (SB)]
6	[(SACDBG), (SACDG), (SAD), (SB)]
7	Success - agenda.pop(0) has goal in path, (SACDBG)

9/22/11 Revised pages

2. Informed search definitions – moving towards optimal search

$f(x)$  = the total cost of the path that your algorithm uses to rank paths.

$g(x)$  = the cost of the path so far.

$h(x)$  = the (under)estimate of the remaining cost to the goal  $g$  node (Use  $h$  for ‘heuristic’)

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

$c(x, y)$  is the actual cost to go from node  $x$  to node  $y$ .

“Heuristics, Patient rules of thumb,  
So often scorned: Sloppy, Dumb!  
Yet, Slowly, common sense come” – Ode to AI

Search Algorithm	Properties	Required Parameters	How the agenda is managed in step 4.
Hill Climbing	Non-optimal, Incomplete Like DFS with a heuristic	$f(x)$ to sort newly added paths (usually, this is just $h$ )	1. Keep only newly-added path extensions sorted by $f(x)$ 2. Add sorted new paths to the FRONT of agenda
Best-First Search	Depends on definition of $f(x)$  If $f(x) = h(x)$ (estimated distance to goal) then likely not optimal, and potentially incomplete. However, A* is a type of best-first search that is complete and optimal because of its choice of $f(x)$ which combines $g(x)$ and $h(x)$ (see below)	$f(x)$ to sort the entire agenda by.	Keep entire agenda sorted by $f(x)$

2.1 Hill-climbing (with backup)

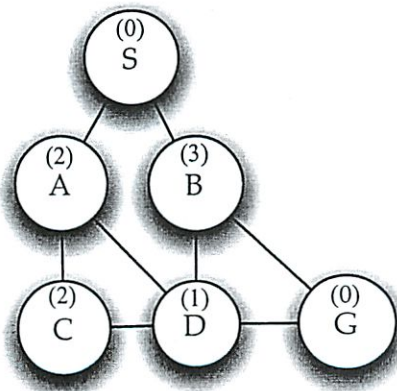
Add path extensions (sorted by heuristic value) to the front of agenda.

Heuristic value is a measure of the ‘goodness’ of the path, e.g., an estimate of how far remaining to go, as the crow flies; or in some other terms if not a map. (We will see this a bit later how to work this into optimal search.)

Note that hill-climbing only looks at the next locally best step (not over all paths!).

(Below we tack the heuristic value to the front of the list, to keep track; note sorting.)

Our graph now has heuristic values that label each node, in parentheses inside the node:



Step	agenda
1	[(0 S)]
2	[(2 SA), (3 SB)]
3	[(1 SAD), (2 SAC), (3 SB)]
4	[(0 SADG), (2 SADC), (3 SADB), (2 SAC), (3 SB)]
5	Success - agenda.pop(0) has goal in path, (SADG)



# 9/22/11 Basic search problems

Mark Vader is shopping for a new evil stronghold. Starting from his current stronghold, the Depth-First-Search Star, he can explore the available models by either subtracting or adding a single feature. Fortunately, Mark remembers how to perform the search techniques he learned in 6.034 from his mentor Emperor Patrickine.

## Part A: Basic Search

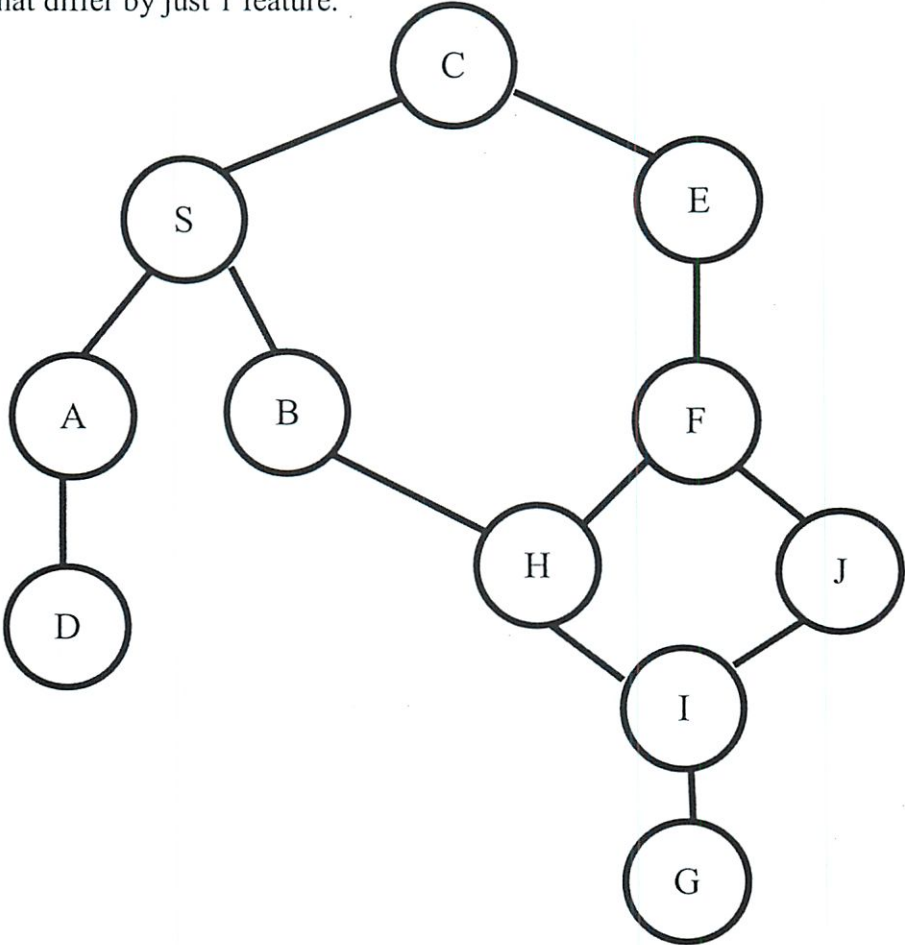
Mark is looking for a stronghold that has the following qualities:

		Exhaust Pipe Weakness	“That’s no Moon”	Race of Enslaved Minions	Secret Escape Route	Sharks with Laser Beams
G	6.03Fortress	-	+	+	+	+

Here is a table of 11 possible strongholds, in tie-breaking order.

		Exhaust Pipe Weakness	“That’s no Moon”	Race of Enslaved Minions	Secret Escape Route	Sharks with Laser Beams
S	DFS Star	+	+	-	-	-
A	Shayol Ghul	-	+	-	-	-
B	Dol Guldor	+	+	+	-	-
C	Moonraker	+	-	-	-	-
D	Zeal Underwater Palace	-	+	-	+	-
E	Core of Zeromus	+	-	+	-	-
F	Whalers of the Moon Ride	+	-	+	-	+
G	6.03Fortress	-	+	+	+	+
H	Atlantis	+	+	+	-	+
I	Willy Wonka's Factory	+	+	+	+	+
J	Dr. Evil Moon Base	+	-	+	+	+

Being a clever Overlord, Mark also produces this graph of exploration **choices** with edges joining the strongholds that differ by just 1 feature.



## Part A1

Mark uses **Depth-First Search with backtracking but NO extended list**. He breaks ties according to the order in the table. List the strongholds that Mark **extends**, in order, starting with the Depth-First-Search Star. Use the letters provided. If he extends a single stronghold more than once, list it multiple times. Extra space is provided below in case you want to show your work.

## Part A2

How many times did Mark hit a dead end?

### Part A3

Mark repeats the process with a **Breadth-First Search with an extended list**. What path does he find?

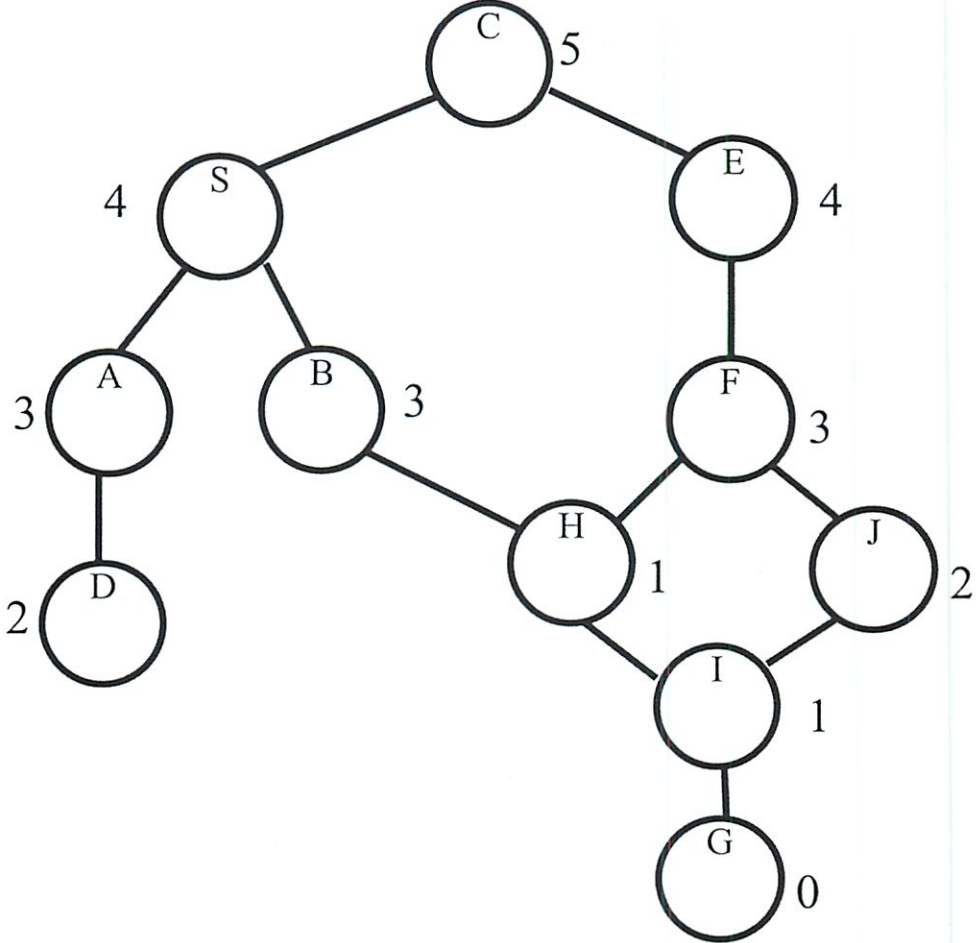


**Part A4: Please use graph on NEXT PAGE that has heuristic values for this!**

Mark considers using **Hill-Climbing with backtracking but NO extended list** using the number of features that do not match his ideal stronghold as a heuristic. Would this substantially help Mark's search through the strongholds given in this problem? Explain briefly.



This graph has the heuristic values at each node (# of features that do not match ideal stronghold)



# The Shortest Path

- Checking an Oracle
- Branch & Bound
- Extended List vs Engreved List
- Admissable Heuristic
- A\*

## Searching Through graph

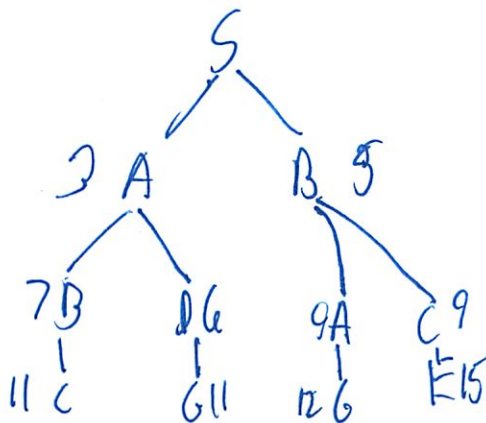
- like a map
- or about the story Macbeth

Not just find a path - but find shortest possible path

Interested in distances

- a heuristic
- pick path w/ shortest distances

British Museum w/  
distances

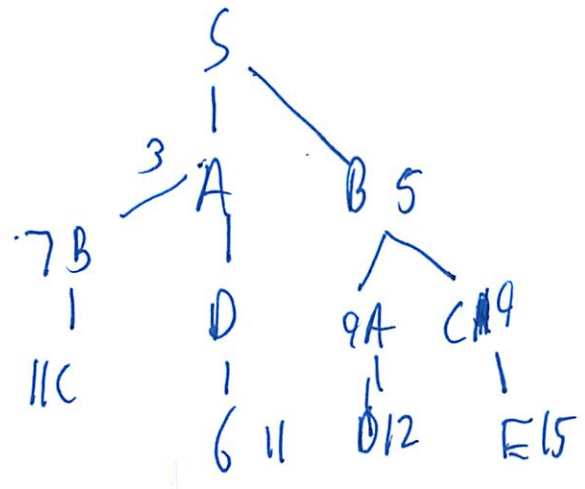


2

Problem Solving: Fastest way to solve many problems is to ask a friend

Can check other paths to confirm

Oracle Checking



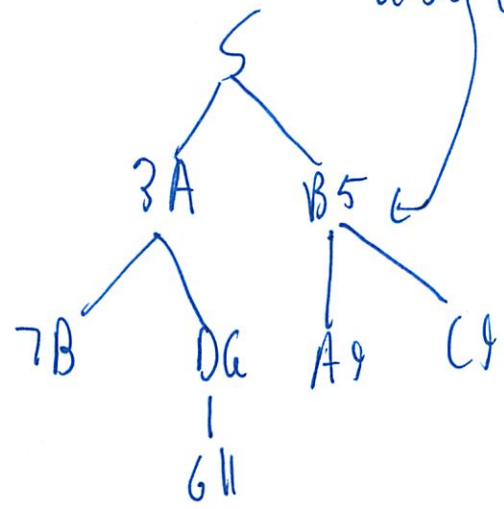
Extend each possible path  
fill its longer than Oracle

~~What if no oracle?~~

What if no oracle: Just extend shortest path

Look at all to find shortest path anywhere

- across every horizontal

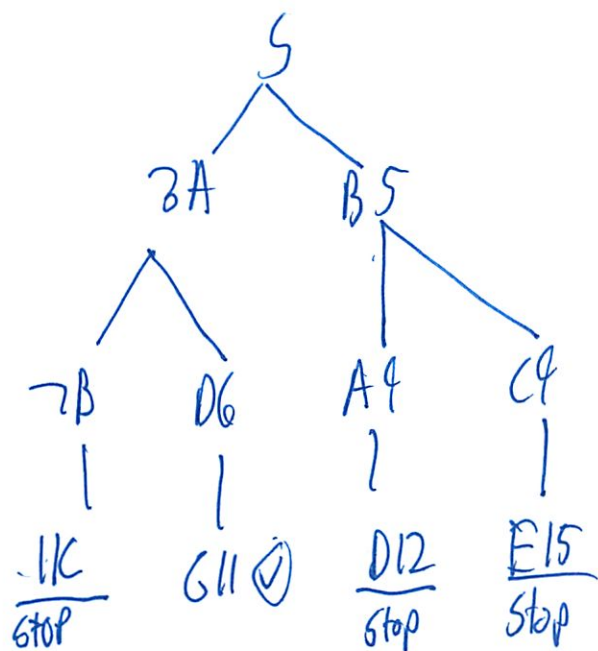


Branch & Bound

No expanded list allowed  
I believe

Even when find a path, must still look for shorter paths  
Extend rest till > 11 or reaches G when you've expanded goal

3

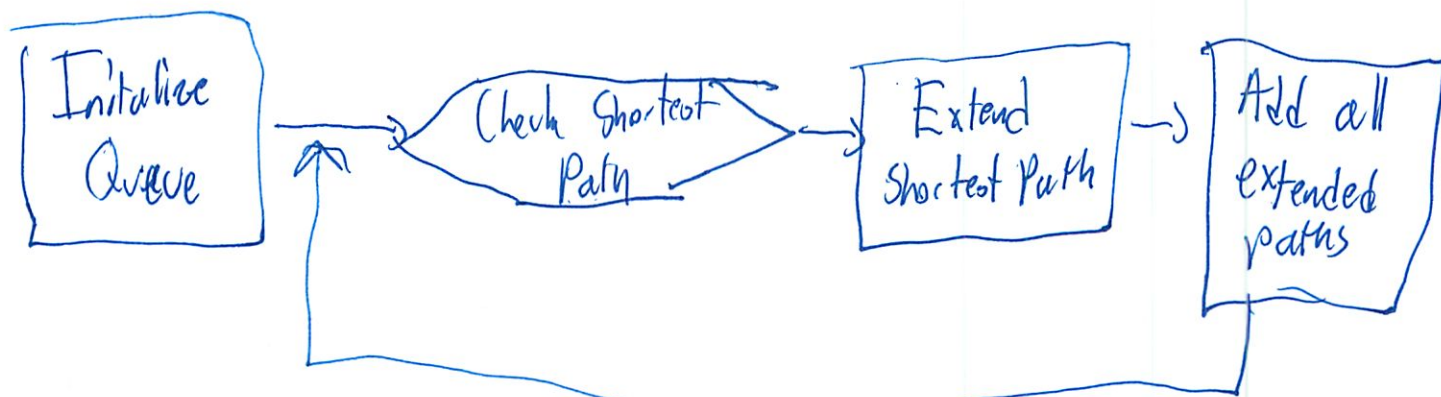


Branch + Band Full

Another ~~method~~ measure: how many paths extended

Do you have to do a whole lot more work to find shortest path?

Doing Branch + Band

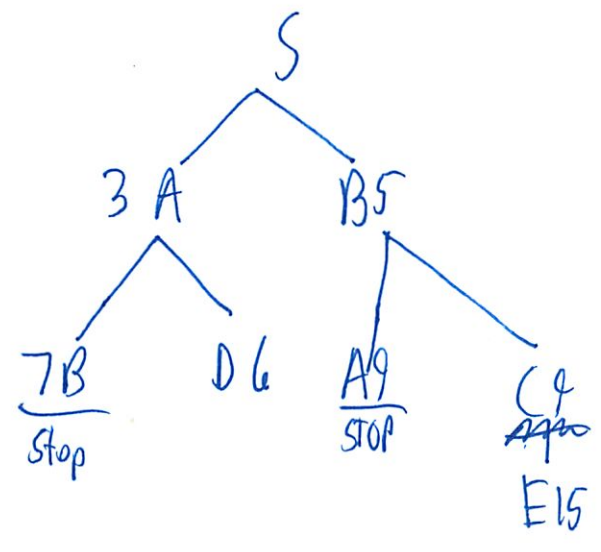


But it evolves into a lot of possible paths  
Very slow on large graphs

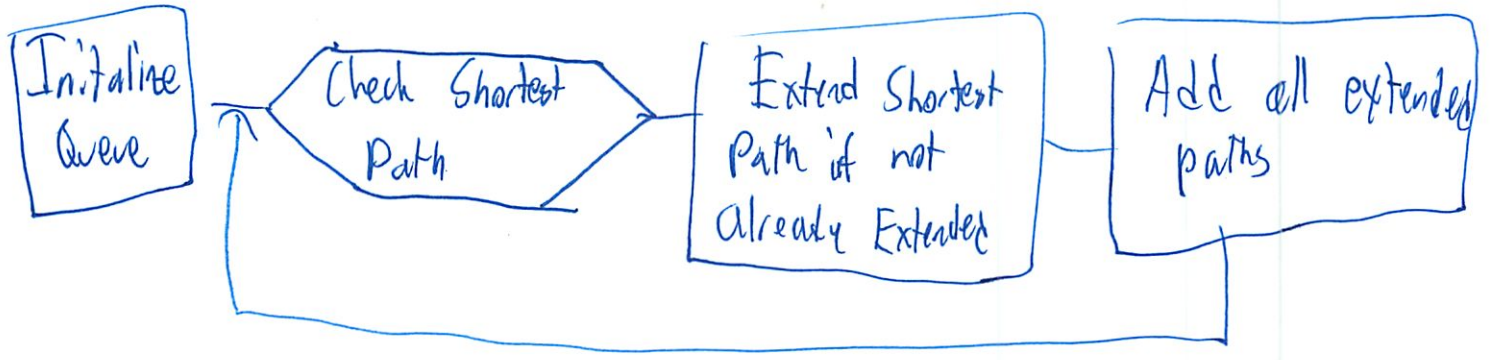


9

Well we can ignore duplicates  
 - pick one w/ lowest path  
 - so use an extended path



B+B + Extended ??



We can also not add it to the tree if already extended → an Enqueued list  
 - Only works on DFS  
 - won't work here since another path might be shorter that we add second

5

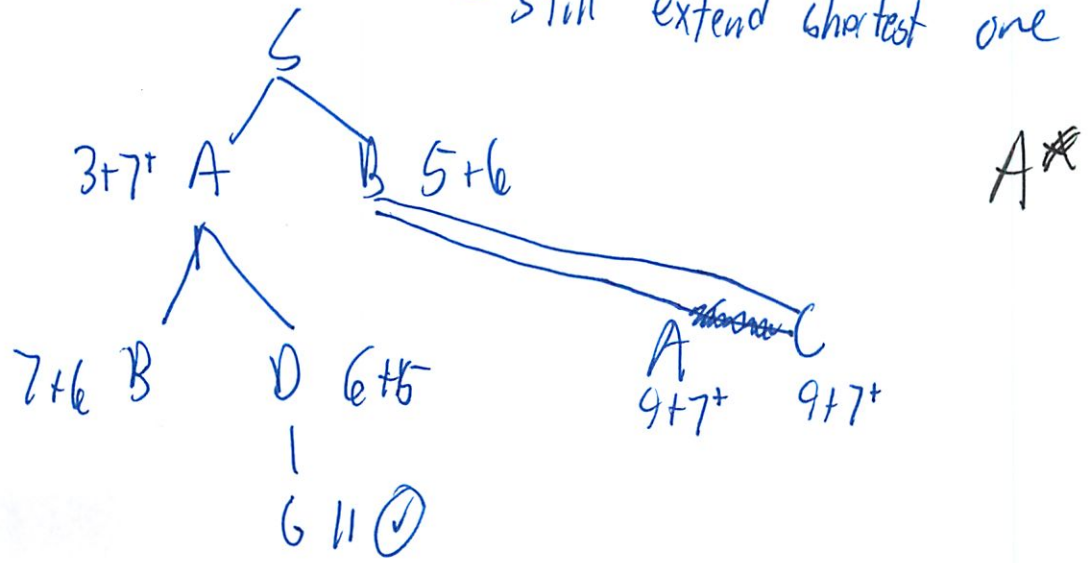
Remember extended list did not help much w/ DFS/hill climbing  
But does a lot of good for BFS

Think B+B + Extended is like BFS  
- really need that extended list

Now lets do something else

We could use a heuristic

- On a map can do crow flies
- don't have a path there - but can know about
- Use existing distance + crow flies to goal  
- still extend shortest one



All the others are longer, so don't need to expand here after finding goal. Actually I think prof said it will find best list always

(6)

This is an admissible heuristic since heuristic is always shorter than original

Does the straight line addition

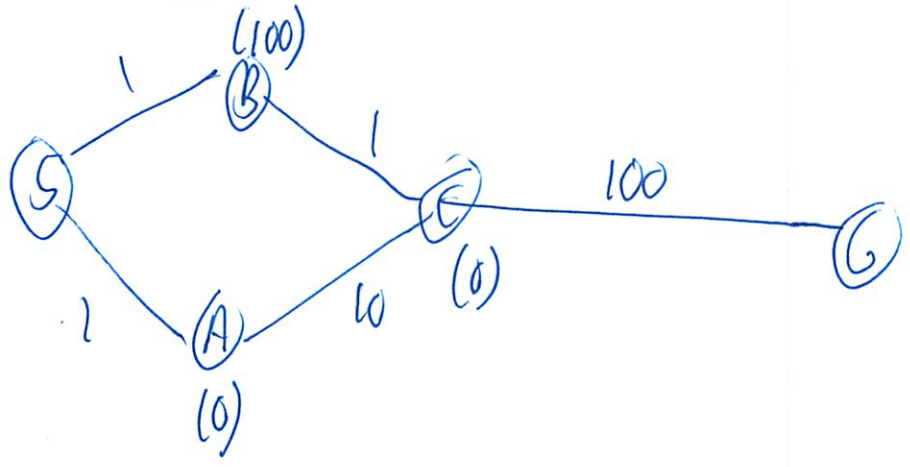
Can use them both - called  $A^*$

	B+A	+Extended	+Admissible	$A^*$	
E1	238	37	73	28	
E2	205	37	4	11	$\epsilon$ almost depth first
E3	60	36	5	5	$\epsilon$ here

$A^*$  is not guaranteed to find shortest ~~map~~ search

- because when its not a map

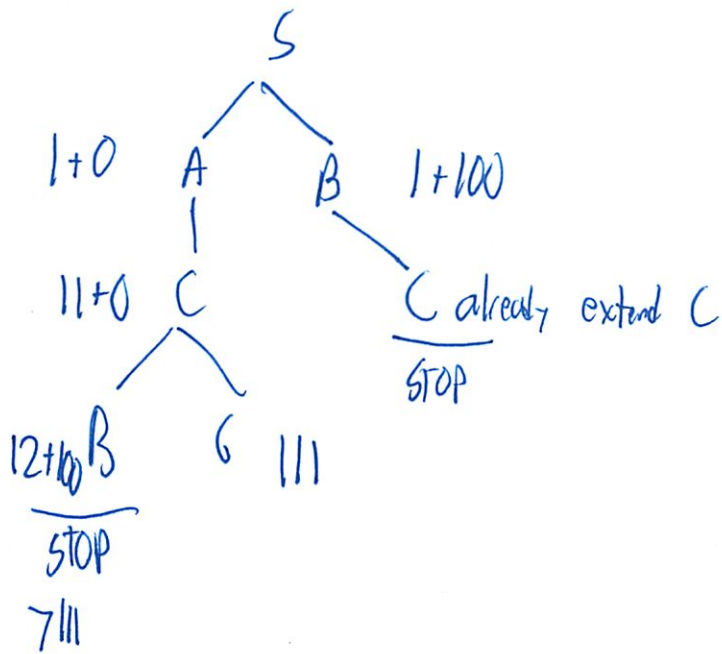
- screw case



( ) = Heuristic - still admissible since under the actual

7

Now do  $A^*$



Heuristic needs to not only be admissible

$$H(x, G) \leq D(x, G)$$

But also consistency

$$\hookrightarrow || H(x, G) - H(y, G) || \leq D(x, y)$$

So screw case b/w B, A is <sup>actual minimum distance</sup> non consistent  
Change (100) to (1)

On a map always consistent

⑧

What if interested in longest path?

- like tasks to do

- or if street view car

- PERT

- care if on critical path

1

9/24

# Lab 2

From 6.034 Fall 2011

## Contents

- 1 Search
  - 1.1 Explanation
    - 1.1.1 Helper functions
  - 1.2 The Agenda
    - 1.2.1 Extending a path in the agenda
  - 1.3 The Extended-Nodes Set
  - 1.4 Returning a Search Result
  - 1.5 Exiting the search
  - 1.6 Multiple choice
- 2 Basic Search
  - 2.1 Breadth-first Search and Depth-first Search
  - 2.2 Hill Climbing
  - 2.3 Beam Search
- 3 Optimal Search
  - 3.1 Branch and Bound
  - 3.2 A\*
- 4 Graph Heuristics
- 5 Survey
- 6 Checking your submission
- 7 Images of Graphs defined in search.py
- 8 Images of Graphs defined in tests.py
- 9 Questions?

This problem set will be due on Friday, September 30, at 11:59pm.

To work on this problem set, you will need to get the code, much like you did for the first two problem sets.

- You can view it at: <http://web.mit.edu/6.034/www/labs/lab2/>
- Download it as a ZIP file: <http://web.mit.edu/6.034/www/labs/lab2/lab2.zip>
- Or, on Athena, add 6.034 and copy it from `/mit/6.034/www/labs/lab2/`.

Your answers for the problem set belong in the main file `lab2.py`.

# Search

## Explanation

This section is an explanation of the system you'll be working with. There aren't any problems to solve. Read it carefully anyway.

We've learned a lot about different types of search in lecture the last week. This problem set will involve implementing several search techniques. For each type of search you are asked to write, you will get a graph (with a list of nodes and a list of edges and a heuristic), a start node, and a goal node.

A graph is a class, defined in `search.py`, that has lists `.nodes` and `.edges` and a dictionary `.heuristic`. Nodes are just string names, but edges are dictionaries that contain each edge's `"NAME"`, `"LENGTH"`, and two endpoints, specified as `"NODE1"` and `"NODE2"`.

The heuristic is a dictionary, for each possible goal node, mapping each possible start node to a heuristic value.

An example graph would be made like this:

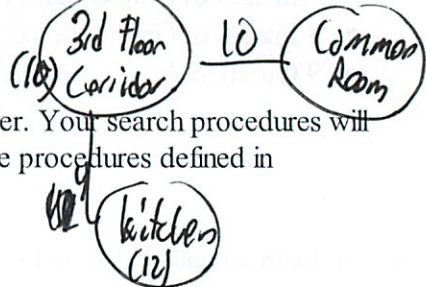
*What was it in 6.034?*

```

Graph(edges=[ { 'NAME': 'e1', 'LENGTH': 10, 'NODE1': 'Forbidden 3rd Floor Corridor', 'NODE2': 'Common Room' },
               { 'NAME': 'e2', 'LENGTH': 4, 'NODE1': 'Common Room', 'NODE2': 'Kitchens' } ],
       heuristic={'Common Room':{ 'Forbidden 3rd Floor Corridor': 10,
                                   'Common Room': 0,
                                   'Kitchens': 12 }
                })

```

In this graph representation, there are three nodes (Forbidden 3rd Floor Corridor, Common Room, and Kitchens), two edges, and heuristic values specified for getting to the Common Room or getting to the Kitchens. One of the edges connects the Forbidden 3rd Floor Corridor with the Common Room, and the other connects the Common Room with the Kitchens. See an image of this graph here ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/search.graph1.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/search.graph1.png)).



The representation for an entire graph, described above, is mainly used in the tester. Your search procedures will receive the graph object and you can access the data within the graph by using the procedures defined in `search.py`, and summarized below:

## Helper functions

These functions will help you work with the graph representation:

- `graph.get_connected_nodes(node)`: Given a node name, return a list of all node names that are connected to the specified node directly by an edge.
- `graph.get_edge(node1, node2)`: Given two node names, return the edge that connects those nodes, or None if there is no such edge.
- `graph.are_connected(node1, node2)`: Return True iff there is an edge running directly between node1 and node2; False otherwise

3

- `graph.is_valid_path(path)`: Given 'path' as an ordered list of node names, return True iff there is an edge between every two adjacent nodes in the list, False otherwise
- `graph.get_heuristic(start, goal)`: Given the name of a starting node in the graph and the name of a goal node, return the heuristic value from that start to that goal. If that heuristic value wasn't supplied when creating the graph, then return 0.

In addition, you're expected to know how to access elements in lists and dictionaries at this point. For some portions of this lab, you may want to use lists like either stacks or queues, as documented at <http://docs.python.org/tut/node7.html>.

You also may need to sort Python lists. Python has built-in sorting functionality, documented at <http://wiki.python.org/moin/HowTo/Sorting>. If you read that document, note that Solaris-Athena computers (the purple Athena computers, not the black ones) might still have an older version of Python prior to v2.4 installed.

## The Agenda

*no longer exist*

Different search techniques explore nodes in different orders, and we will keep track of the nodes remaining to explore in a list we will call the **agenda** (in class we called this the **queue**). Some techniques will add paths to the top of the agenda, treating it like a stack, while others will add to the back of the agenda, treating it like a queue. Some agendas are organized by heuristic value, others are ordered by path distance, and others by depth in the search tree. Your job will be to show your knowledge of search techniques by implementing different types of search and making slight modifications to how the agenda is accessed and updated.

*bring - did already*

## Extending a path in the agenda

In this problem set, a path consists of a list of node names. When it comes time to extend a new path, a path is selected from the agenda. The last node in the path is identified as the node to be extended. The nodes that connect to the extended node, the adjacent nodes, are the possible extensions to the path. Of the possible extensions, the following nodes are NOT added:

- nodes that already appear in the path.
- nodes that have already been extended (if an extended-nodes set is being used.)

As an example, if node *A* is connected to nodes *S*, *B*, and *C*, then the path ['S', 'A'] is extended to the new paths ['S', 'A', 'B'] and ['S', 'A', 'C'] (but not ['S', 'A', 'S']).

*← already in path*

The paths you create should be new objects. If you try to extend a path by *modifying* (or "mutating") the existing path, for example by using list `.append()`, you will probably find yourself in a world of hurt.

## The Extended-Nodes Set

An extended-set, sometimes called an "extended list" or "visited set" or "closed list", consists of nodes that have been extended, and lets the algorithm avoid extending the same node multiple times, sometimes significantly speeding up search. You will be implementing types of search that use extended-sets. Note that an extended-nodes set is a set, so if, e.g., you're using a list to represent it, then be careful that a maximum of one of each node name should appear in it. Python offers other options for representing sets, which may help you avoid this issue. The main

*What?*



4

point is to check that nodes are not in the set before you extend them, and to put nodes into the extended-set when you do choose to extend them.

## Returning a Search Result

A search result is a path which should consist of a list of node names, ordered from the start node, following existing edges, to the goal node.

## Exiting the search

Non-optimal searches such as DFS, BFS, Hill-Climbing and Beam **may** exit either:

- when it finds a path-to-goal in the agenda
- when a path-to-goal is first removed from the agenda. *either or*

Optimal searches such as branch and bound and A\* **must** always exit:

- When a path-to-goal is first removed from the agenda. *Since bes*

For the sake of consistency, you should implement all your searches to exit:

- When a path-to-goal is first removed from the agenda) *just do this*

## Multiple choice

This section contains the first graded questions for the problem set. The questions are located in lab2.py and let you check your knowledge of different types of search. You should, of course, try to answer them before checking the answers in the tester.

## Basic Search

The first two types of search to implement are breadth-first search and depth-first search. When necessary, use backtracking for the search.

### *going back to previous visited nodes - easy w/ agenda* Breadth-first Search and Depth-first Search

Your task is to implement the following functions:

```
def bfs(graph, start, goal):
def dfs(graph, start, goal):
```

The input to the functions are:

- graph: The graph
- start: The name of the node that you want to start traversing from

- end: The name of the node that you want to reach

When a path to the goal node has been found, return the result as explained in the section **Returning a Search Result** (above).

## Hill Climbing

So implement each five

Hill climbing is very similar to depth first search. There is only a slight modification to the ordering of paths that are added to the agenda. For this part, implement the following function:

```
def hill_climbing(graph, start, goal):
```

The hill-climbing procedure you define here *should* use backtracking, for consistency with the other methods, even though hill-climbing typically is not implemented with backtracking.

## Beam Search

which is by default

Beam search is very similar to breadth first search. There is modification to the ordering of paths in the agenda. The agenda at any time can have up to k paths of length n; k is also known as the beam width, and n corresponds to the level of the search graph. You will need to sort your paths by the graph heuristic to ensure that only the top k paths at each level are in your agenda. You may want to use an array or dictionary to keep track of paths of different lengths. You may choose to use an extended-set or not.

For this part, implement the following function:

```
def beam_search(graph, start, goal, beam_size):
```

## Optimal Search

guaranteed to return best solution

The search techniques you have implemented so far have not taken into account the edge distances. Instead we were just trying to find one possible solution of many. This part of the problem set involves finding the path with the shortest distance from the start node to the goal node. The search types that guarantee optimal solutions are branch and bound and A\*.

Since this type of problem requires knowledge of the length of a path, implement the following function that computes the length of a path:

```
def path_length(graph, node_names):
```

The function takes in a graph and a list of node names that make up a path in that graph, and it computes the length

6

of that path, according to the "LENGTH" values for each relevant edge. You can assume the path is valid (there are edges between each node in the graph), so you do not need to test to make sure there is actually an edge between consecutive nodes in the path. If there is only one node in the path, your function should return 0.

### Branch and Bound

example:

Now that you have a way to measure path distance, this part should be easy to complete. You might find the list procedure remove, and/or the Python 'del' keyword, useful (though not necessary). For this part, complete the following:

```
def branch_and_bound(graph, start, goal):
```

### A\*

You're almost there! You've used heuristic estimates to speed up search and edge lengths to compute optimal paths. Let's combine the two ideas now to get the A\* search method. In A\*, the path with the least (heuristic estimate + path length) is taken from the agenda to extend. A\* always uses an extended-set -- make sure to use one. (Note: If the heuristic is not consistent, then using an extended-set can sometimes prevent A\* from finding an optimal solution.)

```
def a_star(graph, start, goal):
```

### Graph Heuristics



A heuristic value gives an approximation from a node to a goal. You've learned that in order for the heuristic to be admissible, the heuristic value for every node in a graph must be less than or equal to the distance of the shortest path from the goal to that node. In order for a heuristic to be consistent, for each edge in the graph, the edge length must be greater than or equal to the absolute value of the difference between the two heuristic values of its nodes.

or will throw it off

In lecture and tutorials, you've seen examples of graphs that have admissible heuristics that were not consistent. Have you seen graphs with consistent heuristics that were not admissible? Why is this so? For this part, complete the following functions, which return True iff the heuristics for the given goal are admissible or consistent, respectively, and False otherwise:

So  
E=2/4-3  
So no shortcuts basically

```
def is_admissible(graph, goal):
def is_consistent(graph, goal):
```

got formula in class

### Survey

Please answer these questions at the bottom of your lab2.py file:

7

- How many hours did this problem set take? *5 hrs*
- Which parts of this problem set, if any, did you find interesting?
- Which parts of this problem set, if any, did you find boring or tedious?

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

## Checking your submission

When you're finished with your lab, don't forget to submit your code and run the online unit tests!

## Images of Graphs defined in search.py

For your manual tracing / debugging enjoyment:

- GRAPH1 ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/search.graph1.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/search.graph1.png))
- GRAPH2 (S->G) ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/search.graph2.sg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/search.graph2.sg.png))
- GRAPH3 (S->G) ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/search.graph3.sg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/search.graph3.sg.png))
- GRAPH4 (S->G) ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/search.graph4.sg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/search.graph4.sg.png))
- GRAPH5 (S->G) ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/search.graph5.sg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/search.graph5.sg.png))

## Images of Graphs defined in tests.py

- NEWGRAPH1 (F->G)  
([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.newgraph1.fg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.newgraph1.fg.png))
- NEWGRAPH1 (S->G)  
([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.newgraph1.sg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.newgraph1.sg.png))
- NEWGRAPH2 (S->G)  
([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.newgraph2.sg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.newgraph2.sg.png))
- NEWGRAPH2 (G->H)  
([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.newgraph2.gh.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.newgraph2.gh.png))
- NEWGRAPH3 (S->S)  
([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.newgraph3.ss.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.newgraph3.ss.png))
- NEWGRAPH4 (S->T) ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.newgraph4.st.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.newgraph4.st.png))
- AGRAPH (S->G) ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.newgraph5.fg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.newgraph5.fg.png))
- SAQG (S->G) ([http://web.mit.edu/6.034/www/labs/lab2\\_graphs/neato/png/tests.saqq.sg.png](http://web.mit.edu/6.034/www/labs/lab2_graphs/neato/png/tests.saqq.sg.png))

## Questions?

If you find what you think is an error in the problem set, tell [6034tas@csail.mit.edu](mailto:6034tas@csail.mit.edu) about it.

- This page was last modified on 22 September 2011, at 04:09.
- *Forsan et haec olim meminisse iuvabit.*

BFS

- look in recitation notes for Pseudo code
- or go back to 6.01 Files!

Got v1 Fast

But append mutates - does not return

And then other minor fix

- fell into place w/o much extra thought

Spent 15 min restarting after PC crashed b/c

did not check if node was on current path

Hill climbing = <sup>add</sup> in order of heuristic?

yes

or resort after each add so min list

should we implement extended list checking?

oh checking is for less than 1 sec

② So now some of my tests are correct

- Oh was added heuristic
- 'just searched'

Need to track heuristic count  
remaining distance to goal

Add insert helper method

'go through till  $> =$  what we want

- 'insert before'

---

Here testing would be good

- what if item is  $0$  - less than all

---

Why are ~~all~~ some tests failing?

- finding shorter paths to goal
- not adding correctly

Oh if no match add at end  
since if 4 on, add 12 ...

③ Now when adding needs to 'add after current level'  
Or test case wrong

So what is the actual rule  
- at each branch pick lowest?

Can't reset agenda since need backtracking

Made 5-min mistake where did not itt  
- grrr

---

## Beam Search

- So basically take ~~all~~ above
- Only add max  $m \times k$  paths for each node
- And search breadth (so add to pack)
- Quit after  $n$  rounds

- which is not a function input  
How are they added to local agenda?  
Same way I think - by order

Why is it returning none?



9  
Oh return none if no results ...

Wish would only show debug on failed tests ...

Ok is on extended list

- but that should be Ok

I can just remove - but why is it breaking

well it did add them to extended

Still wrong when removed ...

- No its a diff one ...

Oh k is max paths on agenda?

No then it says k at each level

So at the 3 level or something

at any one time

So need beam width enforcer

So ~~not~~ add items to main agenda

And then pare?

Q5

No its still BFS so should add new to end

But then what to prune

Only top k paths

But then what to explore next -> same level

So add to end!

But then sort to remove

Only 1 case ever where I prune

~~So it seems still do 1.~~

Oh I was not pruning right

- only if len = 2

- Being stupid! Did not add next go proper array division

Seems even worse now!

- pruning too aggressively

Do want to add to end of the agenda

Oh I'm having item deletion bug I think

copy instead

I thought about it but though python handle

~~2~~ (6) Then something I tried earlier was wrong!  
But still finds goal  
Seems like should do an array for each level  
So place sorted for each level?

~~Make~~ Make

Oh I was not adding 1 if diff length

Works

---

Branch + Bound

Path Length

So list of nodes provided

else do node path iteration

get\_edge

Ah perfect - finished fast

---

Branch + Bound

↓

7

So what are rules here

look at all to find shortest path anywhere

So like hill but no local agency

- think had earlier

- since confused them since hill was Mon, B+B yesterday

Let me try for old ~~path~~ code

Nope not right

- since even when find path must look for shorter

So add to list of possible goals?

And stop when cost  $> l$

No expanded!

---

Last are dies on file - 2 possible w/ same length

- what to do

- just return ~~at~~ the most recent?

8

# A\*

- least (heuristic + path length)
- and w/ extended set

But I was not using path in B+B

I was using heuristic

Need to fix!

Path = shortest path from here

Fixed B+B - and did want first NOT most recent goal

A\* existing path length + heuristic

Extended list

But return at 1st goal

Oh path length not proper inats

There we go works

9

## Heuristics

formula in class

(check whole graph)

to get shortest path - use  $A^*$

first need get nodes function

Two are wrong  
- actually false

No diff if switch to  $B+B$   
(since heuristic constant ...)

Actually need none heuristic function

$B+B$  is that

try is constant

That works!

New defn is admissible

Where does it fail?

They should tell you ...

Ohh oh I am returning a path + comparing w/ a #!

⑩ Oh ~~pics were available~~ online of test cases

A-star error seems to be file format.

-I mean char format

[S] not S

put in special case fix

Done! S.O

```
# Fall 2011 6.034 Lab 2: Search
# Name: [Your name]
# email: [mit mail]
#
# Your answers for the true and false questions will be in the following form.
# Your answers will look like one of the two below:
#ANSWER1 = True
#ANSWER1 = False

# 1: True or false - Hill Climbing search is guaranteed to find a solution if there is a
solution
ANSWER1 = False

# 2: True or false - Best-first search will give an optimal search result (shortest path
length).
ANSWER2 = False

# 3: True or false - Best-first search and hill climbing make use of heuristic values of
nodes.
ANSWER3 = True

# 4: True or false - A* uses an extended-nodes set.
ANSWER4 = True

# 5: True or false - Breadth first search is guaranteed to return a path with the shortest
number of nodes.
ANSWER5 = True #if extend all the way??

# 6: True or false - The regular branch and bound uses heuristic values to speed up the
search for an optimal path.
ANSWER6 = False #these questions just about do we know the names for the variations

# Import the Graph data structure from 'search.py'
# Refer to search.py for documentation
from search import Graph

## Problem 2.1
def bfs(graph, start, goal):
    agenda = [ [ start ] ]
    while len(agenda) is not 0:
        #explore next item
        path = agenda.pop(0) #remove first item
        if path[-1] == goal: #check for goal when full path-to-goal is removed from agenda
            #as opposed to checking graph.is_connected(path[-1], goal)
            return path
        #get all connected nodes
        connected = graph.get_connected_nodes(path[-1])
        for node in connected:
            #add to end of agenda since BFS
            if (path.count(node) == 0): #check if node already on this path
                newpath = list(path) #copy
                newpath.append(node)
```



```

        agenda.append(newpath)

#timeout
return []

## Once you have completed the breadth-first search, this part should be very simple to
complete.
def dfs(graph, start, goal):
    agenda = [ [ start ] ]
    while len(agenda) is not 0:
        #explore next item
        path = agenda.pop(0) #remove first item
        if path[-1] == goal: #check for goal when full path-to-goal is removed from agenda
            #as opposed to checking graph.is_connected(path[-1], goal)
            return path
        #get all connected nodes
        connected = graph.get_connected_nodes(path[-1])
        for node in connected:
            #add to start of agenda since DFS
            if (path.count(node) == 0): #check if node already on this path
                newpath = list(path) #copy
                newpath.append(node)
                agenda.insert(0,newpath)

#timeout
return []

## Now we're going to add some heuristics into the search.
## Remember that hill-climbing is a modified version of depth-first search.
## Search direction should be towards lower heuristic values to the goal.
def hill_climbing(graph, start, goal):
    agenda = [ [ start ] ]
    extended = []
    while len(agenda) is not 0:
        #explore next item
        path = agenda.pop(0)[0] #remove first item
        #check that we have not extended this before
        if (extended.count(path[-1]) == 0):
            if path[-1] == goal: #check for goal when full path-to-goal is removed from
agenda
                #as opposed to checking graph.is_connected(path[-1], goal)
                return path
        #get all connected nodes
        connected = graph.get_connected_nodes(path[-1])
        #make local agenda for this node
        #would reset entire agenda if no backtracking
        localAgenda = []
        for node in connected:
            #add to start of agenda since DFS
            if (path.count(node) == 0): #check if node already on this path
                newpath = list(path) #copy
                newpath.append(node)
                #check distance from this new node to goal
                heuristic = graph.get_heuristic(node, goal)

```

```

        localAgenda = addToAgendaSorted(localAgenda,[newpath, heuristic])
    #add all of localAgenda to agenda while keeping order!
    i = 0
    for item in localAgenda:
        agenda.insert(i, item) #keep order of localAgenda
        i = i+1
    #add to extended
    extended.append(path[-1])
#timeout
return []

def addToAgendaSorted (agenda, newpathItem):
    newpath = newpathItem[0]
    length = newpathItem[1]

    i=0
    for item in agenda:
        if item[1] >= length:
            agenda.insert(i, newpathItem)
            return agenda
        i = i + 1
    #if not found, insert at end
    agenda.append(newpathItem)
    return agenda

## Now we're going to implement beam search, a variation on BFS
## that caps the amount of memory used to store paths.
## Remember, beam search is a variant of breadth first search but where
## we maintain only k candidate paths of length n in our agenda at anytime.
## The k top candidates are to be determined using the
## graph get_heuristics function, with lower values being better values.
def beam_search(graph, start, goal, beam_width):
    agenda = [ [ start ] ]
    extended = []
    while len(agenda) is not 0:
        #explore next item
        path = agenda.pop(0)[0] #remove first item
        #check that we have not extended this before
        if (extended.count(path[-1]) == 0):
            if path[-1] == goal: #check for goal when full path-to-goal is removed from
agenda
                #as opposed to checking graph.is_connected(path[-1], goal)
                return path
            #get all connected nodes
            connected = graph.get_connected_nodes(path[-1])
            for node in connected:
                #add to start of agenda since DFS
                if (path.count(node) == 0): #check if node already on this path
                    newpath = list(path) #copy
                    newpath.append(node)
                    #check distance from this new node to goal
                    heuristic = graph.get_heuristic(node, goal)

```

```

        agenda = addToAgendaSortedForDepth(agenda, [newpath, heuristic])
        nodeDepth = len(newpath)
    prunedAgenda = [] #make temp copy to avoid changing loop under iterator
    count = 0
    for item in agenda:
        if len(item[0]) == nodeDepth:
            if count < beam_width:
                count = count + 1
                prunedAgenda.append(item)
            else:
                prunedAgenda.append(item)
    agenda = prunedAgenda
    #add to extended
    extended.append(path[-1])
#timeout
return []

def addToAgendaSortedForDepth(agenda, newpathItem):
    newpath = newpathItem[0]
    length = newpathItem[1]

    #if no items
    if len(agenda) == 0:
        agenda.append(newpathItem)
        return agenda

    i=0
    for item in agenda:
        if len(item[0]) == len(newpath):
            if item[1] >= length:
                agenda.insert(i, newpathItem)
                return agenda
            i = i + 1
    #if not found, insert at end
    agenda.append(newpathItem)
    return agenda

## Now we're going to try optimal search. The previous searches haven't used edge
distances in the calculation. In order to compute the optimal path, complete the helper
function below.
## The function you are to complete is below. It takes in a graph and a list of node
names, and returns the sum of edge lengths along the path -- the total distance in the path.
def path_length(graph, node_names):
    #test if just one node
    if len(node_names) == 1:
        return 0

    answer = 0
    i = 0
    for node in node_names:
        if i+1 < len(node_names):
            answer = answer + graph.get_edge(node, node_names[i+1])['LENGTH']

```

```

        i = i + 1
    return answer

def branch_and_bound(graph, start, goal):
    agenda = [ [ start ] ]
    goals = []
    bestGoal = 9999999999999999
    while len(agenda) is not 0:
        #explore next item
        path = agenda.pop(0)[0] #remove first item
        if path[-1] == goal:
            goals.append(path) #add as possible goal
            bestGoal = len(path)
        #check that not longer than current bestGoal
        if len(path) < bestGoal:
            #get all connected nodes
            connected = graph.get_connected_nodes(path[-1])
            for node in connected:
                #add to start of agenda since DFS
                if (path.count(node) == 0): #check if node already on this path
                    newpath = list(path) #copy
                    newpath.append(node)
                    #check distance from start to this new node
                    pathLength = path_length(start, node)
                    addToAgendaSorted(agenda, [newpath, pathLength])

    #timeout
    if bestGoal == 9999999999999999:
        return []
    #return first best goal
    for goal in goals:
        if len(goal) == bestGoal:
            return goal

def a_star(graph, start, goal):
    agenda = [ [ start ] ]
    extended = []
    while len(agenda) is not 0:
        #explore next item
        path = agenda.pop(0)[0] #remove first item
        if path[-1] == goal: #check for goal when full path-to-goal is removed from agenda
            #as opposed to checking graph.is_connected(path[-1], goal)
            if len(path) == 1: #weird special test case on online tests
                return [path]
            else:
                return path
        #check that we have not extended this before
        if (extended.count(path[-1]) == 0):
            #get all connected nodes
            connected = graph.get_connected_nodes(path[-1])
            for node in connected:
                #add to start of agenda since DFS

```

```

    if (path.count(node) == 0): #check if node already on this path
        newpath = list(path) #copy
        newpath.append(node)
        #check distance from start to this new node
        pathLength = path_length(graph, newpath)
        #and the remaining heuristic new node to goal
        heuristic = graph.get_heuristic(node, goal)
        addToAgendaSorted(agenda, [newpath, pathLength + heuristic])

#add to extended
extended.append(path[-1])

#timeout
if bestGoal == 9999999999999999:
    return []
#return first best goal
for goal in goals:
    if len(goal) == bestGoal:
        return goal

## It's useful to determine if a graph has consistent and admissible
## heuristics.  You've seen graphs with heuristics that are
## admissible, but not consistent.  Have you seen any graphs that are
## consistent, but not admissible?

def is_admissible(graph, goal):
    #for each node, the heuristic(node, goal) must be <=
    #distance of shortest path (node, goal)
    for node in graph.nodes:
        if graph.get_heuristic(node, goal) > path_length(graph, branch_and_bound(graph, node
, goal)):
            return False
    #timeout
    return True

def is_consistent(graph, goal):
    #each edge in the in the graph, the edge length must be >=
    #abs(diff between the 2 heuristic values of the nodes)
    for edge in graph.edges:
        if edge['LENGTH'] < abs(graph.get_heuristic(edge['NODE1'], goal) - graph.
get_heuristic(edge['NODE2'], goal)):
            return False
    return True

HOW_MANY_HOURS_THIS_PSET_TOOK = '5 hrs'
WHAT_I_FOUND_INTERESTING = 'The minute differences between the different formats.  Someone
should make a chart which options are used in which'
WHAT_I_FOUND_BORING = 'Early parts doing basic graph'
#You should ask which parts were hard, but we figured out

```

9/24

# 7

## Rules and Rule Chaining

quick review  
- Seems like distraction  
to read in  
this class

In this chapter, you learn about the use of easily-stated if-then rules to solve problems. In particular, you learn about forward chaining from assertions and backward chaining from hypotheses.

By way of illustration, you learn about two toy systems; one identifies zoo animals, the other bags groceries. These examples are analogous to influential, classic systems that diagnose diseases and configure computers.

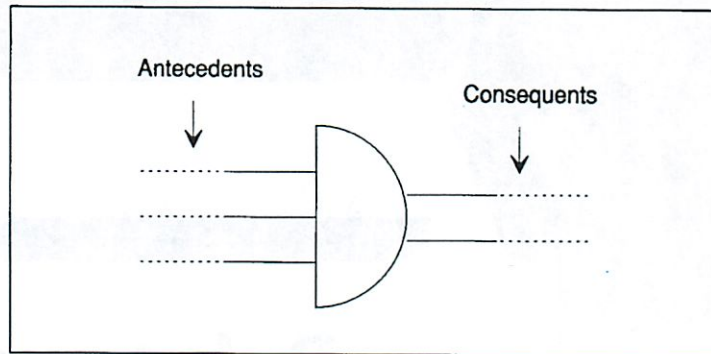
You also learn about how to implement rule-based systems. You learn, for example, how search methods can be deployed to determine which of many possible rules are applicable during backward chaining, and you learn how the rete procedure does efficient forward chaining.

When you have finished this chapter, you will understand the key ideas that support many of the useful applications of artificial intelligence. Such applications are often mislabeled **expert systems**, even though their problem-solving behavior seems more like that of human novices, rather than of human experts.

### RULE-BASED DEDUCTION SYSTEMS

Rule-based problem-solving systems are built using rules like the following, each of which contains several *if* patterns and one or more *then* patterns:

**Figure 7.1** A convenient graphical notation for antecedent–consequent rules. The symbol, appropriately, is the same as the one used in digital electronics for AND gates.



```

Rn  If    if1
      if2
      ⋮
     then then1
        then2
        ⋮
  
```

In this section, you learn how rule-based systems work.

### Many Rule-Based Systems Are Deduction Systems

A statement that something is true, such as “Stretch has long legs,” or “Stretch is a giraffe,” is an **assertion**.<sup>†</sup> In all rule-based systems, each *if* pattern is a pattern that may match one or more of the assertions in a collection of assertions. The collection of assertions is sometimes called **working memory**.

In many rule-based systems, the *then* patterns specify new assertions to be placed into working memory, and the rule-based system is said to be a **deduction system**. In deduction systems, the convention is to refer to each *if* pattern as an **antecedent** and to each *then* pattern as a **consequent**. Figure 7.1 shows a graphical notation for deduction-oriented antecedent–consequent rules.

Sometimes, however, the *then* patterns specify actions, rather than assertions—for example, “Put the item into the bag”—in which case the rule-based system is a **reaction system**.

In both deduction systems and reaction systems, **forward chaining** is the process of moving from the *if* patterns to the *then* patterns, using the *if* patterns to identify appropriate situations for the deduction of a new assertion or the performance of an action.

<sup>†</sup>Facts and assertions are subtly different: A *fact* is something known to be true; an assertion is a statement that something is a fact. Thus, assertions can be false, but facts cannot be false.

During forward chaining, whenever an *if* pattern is observed to match an assertion, the antecedent is **satisfied**. Whenever all the *if* patterns of a rule are satisfied, the rule is **triggered**. Whenever a triggered rule establishes a new assertion or performs an action, it is **fired**.

In deduction systems, all triggered rules generally fire. In reaction systems, however, when more than one rule is triggered at the same time, usually only one of the possible actions is desired, thus creating a need for some sort of *conflict-resolution procedure* to decide which rule should fire.

### A Toy Deduction System Identifies Animals

Suppose that Robbie, a robot, wants to spend a day at the zoo. Robbie can perceive basic features, such as color and size, and whether an animal has hair or gives milk, but his ability to identify objects using those features is limited. He can distinguish animals from other objects, but he cannot use the fact that a particular animal has a long neck to conclude that he is looking at a giraffe.

Plainly, Robbie will enjoy the visit more if he can identify the individual animals. Accordingly, Robbie decides to build ZOOKEEPER, an identification-oriented deduction system.

Robbie could build ZOOKEEPER by creating one if-then rule for each kind of animal in the zoo. The consequent side of each rule would be a simple assertion of animal identity, and the antecedent side would be a bulbous enumeration of characteristics sufficiently complete to reject all incorrect identifications.

Robbie decides, however, to build ZOOKEEPER by creating rules that produce intermediate assertions. The advantage is that the antecedent-consequent rules involved need have only a few antecedents, making them easier for Robbie to create and use. Using this approach, ZOOKEEPER produces chains of conclusions leading to the identification of the animal that Robbie is currently examining.

Now suppose that Robbie's local zoo contains only seven animals: a cheetah, a tiger, a giraffe, a zebra, an ostrich, a penguin, and an albatross. This assumption simplifies ZOOKEEPER, because only a few rules are needed to distinguish one type of animal from another. One such rule, rule Z1, determines that a particular animal is a mammal:

```
Z1      If      ?x has hair
        then    ?x is a mammal
```

Note that antecedents and consequents are patterns that contain *variables*, such as *x*, marked by question-mark prefixes. Whenever a rule is considered, its variables have no values initially, but they acquire values as antecedent patterns are matched to assertions.

Suppose that a particular animal, named *Stretch*, has hair. Then, if the working memory contains the assertion *Stretch has hair*, the antecedent



pattern, *?x has hair*, matches that assertion, and the value of *x* becomes *Stretch*. By convention, when variables become identified with values, they are said to be **bound** to those values and the values are sometimes called **bindings**. Thus, *x* is bound to *Stretch* and *Stretch* is *x*'s binding.

Once a variable is bound, that variable is replaced by its binding whenever the variable appears in the same or subsequently processed patterns. Whenever the variables in a pattern are replaced by variable bindings, the pattern is said to be **instantiated**. For example, the consequent pattern, *?x is a mammal* becomes *Stretch is a mammal* once instantiated by the variable binding acquired when the antecedent pattern was matched.

Now let us look at other ZOOKEEPER rules. Three others also determine biological class:

- Z2    If     ?*x* gives milk  
      then  ?*x* is a mammal
- Z3    If     ?*x* has feathers  
      then  ?*x* is a bird
- Z4    If     ?*x* flies  
          ?*x* lays eggs  
      then  ?*x* is a bird

The last of these rules, Z4, has two antecedents. Although it does not really matter for the small collection of animals in ZOOKEEPER's world, some mammals fly and some reptiles lay eggs, but no mammal or reptile does both.

Once ZOOKEEPER knows that an animal is a mammal, two rules determine whether that animal is carnivorous. The simpler rule has to do with catching the animal in the act of having its dinner:

- Z5    If     ?*x* is a mammal  
          ?*x* eats meat  
      then  ?*x* is a carnivore

If Robbie is not at the zoo at feeding time, various other factors, if available, provide conclusive evidence:

- Z6    If     ?*x* is a mammal  
          ?*x* has pointed teeth  
          ?*x* has claws  
          ?*x* has forward-pointing eyes  
      then  ?*x* is a carnivore

All hooved animals are ungulates:

Z7    If     ? $x$  is a mammal  
           ? $x$  has hoofs  
       then  ? $x$  is an ungulate

If Robbie has a hard time looking at the feet, ZOOKEEPER may still have a chance because all animals that chew cud are also ungulates:

Z8    If     ? $x$  is a mammal  
           ? $x$  chews cud  
       then  ? $x$  is an ungulate

Now that Robbie has rules that divide mammals into carnivores and ungulates, it is time to add rules that identify specific animal identities. For carnivores, there are two possibilities:

Z9    If     ? $x$  is a carnivore  
           ? $x$  has tawny color  
           ? $x$  has dark spots  
       then  ? $x$  is a cheetah

Z10   If     ? $x$  is a carnivore  
           ? $x$  has tawny color  
           ? $x$  has black strips  
       then  ? $x$  is a tiger

Strictly speaking, the basic color is not useful because both of the carnivores are tawny. However, there is no need for information in rules to be minimal. Moreover, antecedents that are superfluous now may become essential later as new rules are added to deal with other animals.

For the ungulates, other rules separate the total group into two possibilities:

Z11   If     ? $x$  is an ungulate  
           ? $x$  has long legs  
           ? $x$  has long neck  
           ? $x$  has tawny color  
           ? $x$  has dark spots  
       then  ? $x$  is a giraffe

Z12   If     ? $x$  is an ungulate  
           ? $x$  has white color  
           ? $x$  has black stripes  
       then  ? $x$  is a zebra

Three more rules are needed to handle the birds:

- Z13    If    ?x is a bird  
               ?x does not fly  
               ?x has long legs  
               ?x has long neck  
               ?x is black and white  
           then ?x is an ostrich
- Z14    If    ?x is a bird  
               ?x does not fly  
               ?x swims  
               ?x is black and white  
           then ?x is a penguin
- Z15    If    ?x is a bird  
               ?x is a good flyer  
           then ?x is an albatross

Now that you have seen all the rules in ZOOKEEPER, note that the animals evidently share many features. Zebras and tigers have black stripes; tigers, cheetahs, and giraffes have a tawny color; giraffes and ostriches have long legs and a long neck; and ostriches and penguins are black and white.

To learn about how forward chaining works, suppose that Robbie is at the zoo and is about to analyze an unknown animal, Stretch, using ZOOKEEPER. Further suppose that the following six assertions are in working memory:

Stretch has hair.  
 Stretch chews cud.  
 Stretch has long legs.  
 Stretch has a long neck.  
 Stretch has tawny color.  
 Stretch has dark spots.

Because Stretch has hair, rule Z1 fires, establishing that Stretch is a mammal. Because Stretch is a mammal and chews cud, rule Z8 establishes that Stretch is an ungulate.

At this point, all the antecedents for rule Z11 are satisfied. Evidently, Stretch is a giraffe.

### Rule-Based Systems Use a Working Memory and a Rule Base

As you have seen in the ZOOKEEPER system, one of the key representations in a rule-based system is the working memory:

---

A **working memory** is a representation

In which

- ▷ Lexically, there are application-specific symbols and pattern symbols.
- ▷ Structurally, assertions are lists of application-specific symbols, and patterns are lists of application-specific symbols and pattern symbols.
- ▷ Semantically, the assertions denote facts in some world.

With constructors that

- ▷ Add an assertion to working memory

With readers that

- ▷ Produce a list of the matching assertions in working memory, given a pattern
- 

Another key representation is the rule base:

---

A **rule base** is a representation

In which

- ▷ Lexically, there are application-specific symbols and pattern symbols.
- ▷ Structurally, patterns are lists of application-specific symbols and pattern symbols, and rules consist of patterns. Some of these patterns constitute the rule's *if* patterns; the others constitute the rule's *then* pattern.
- ▷ Semantically, rules denote constraints that enable procedures to seek new assertions or to validate a hypothesis.

With constructors that

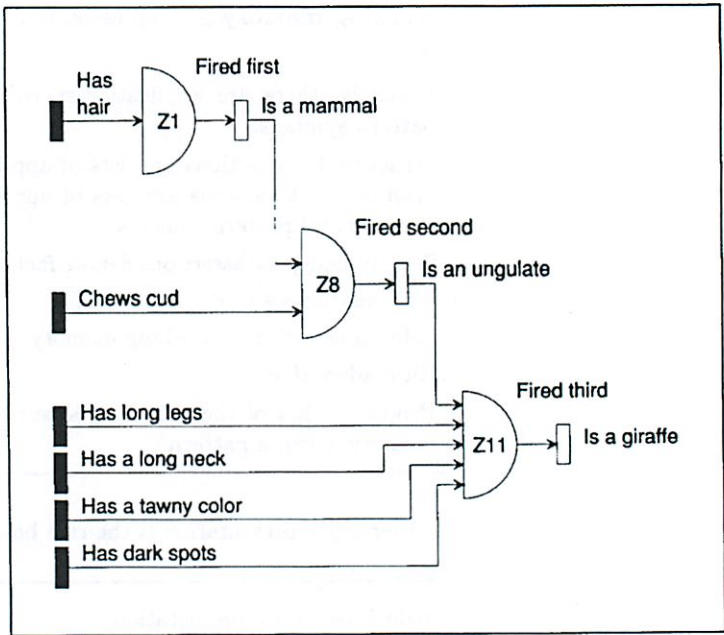
- ▷ Construct a rule, given an ordered list of *if* patterns and a *then* pattern

With readers that

- ▷ Produce a list of a given rule's *if* patterns
  - ▷ Produce a list of a given rule's *then* patterns
- 

Thus, ZOOKEEPER uses instances of these representations that are specialized to animal identification. ZOOKEEPER itself can be expressed in procedural English, as follows:

**Figure 7.2** Knowing something about an unknown animal enables identification via forward chaining. Here, the assertions on the left lead to the conclusion that the unknown animal is a giraffe.



To identify an animal with ZOOKEEPER (forward-chaining version),

- ▷ Until no rule produces a new assertion or the animal is identified,
- ▷ For each rule,
  - ▷ Try to support each of the rule's antecedents by matching it to known facts.
  - ▷ If all the rule's antecedents are supported, assert the consequent unless there is an identical assertion already.
  - ▷ Repeat for all matching and instantiation alternatives.

Thus, assertions flow through a series of antecedent-consequent rules from given assertions to conclusions, as shown in the history recorded in figure 7.2. In such diagrams, sometimes called **inference nets**, the D-shaped objects represent rules, whereas vertical bars denote given assertions and vertical boxes denote deduced assertions.

**Deduction Systems May Run Either Forward or Backward**

So far, you have learned about a deduction-oriented rule-based system that works from given assertions to new, deduced assertions. Running this

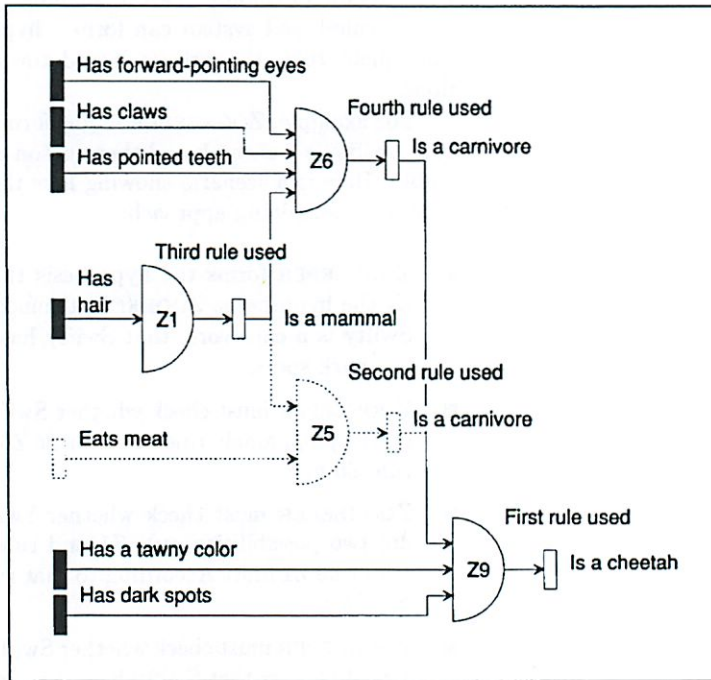
way, a system exhibits forward chaining. *Backward chaining* is also possible: A rule-based system can form a hypothesis and use the antecedent-consequent rules to work backward toward hypothesis-supporting assertions.

For example, ZOOKEEPER might form the hypothesis that a given animal, Swifty, is a cheetah and then reason about whether that hypothesis is viable. Here is a scenario showing how things work out according to such a backward-chaining approach:

- ZOOKEEPER forms the hypothesis that Swifty is a cheetah. To verify the hypothesis, ZOOKEEPER considers rule Z9, which requires that Swifty is a carnivore, that Swifty has a tawny color, and that Swifty has dark spots.
- ZOOKEEPER must check whether Swifty is a carnivore. Two rules may do the job, namely rule Z5 and rule Z6. Assume that ZOOKEEPER tries rule Z5 first.
- ZOOKEEPER must check whether Swifty is a mammal. Again, there are two possibilities, rule Z1 and rule Z2. Assume that ZOOKEEPER tries rule Z1 first. According to that rule, Swifty is a mammal if Swifty has hair.
- ZOOKEEPER must check whether Swifty has hair. Assume ZOOKEEPER already knows that Swifty has hair. So Swifty must be a mammal, and ZOOKEEPER can go back to working on rule Z5.
- ZOOKEEPER must check whether Swifty eats meat. Assume ZOOKEEPER cannot tell at the moment. ZOOKEEPER therefore must abandon rule Z5 and try to use rule Z6 to establish that Swifty is a carnivore.
- ZOOKEEPER must check whether Swifty is a mammal. Swifty is a mammal, because this was already established when trying to satisfy the antecedents in rule Z5.
- ZOOKEEPER must check whether Swifty has pointed teeth, has claws, and has forward-pointing eyes. Assume ZOOKEEPER knows that Swifty has all these features. Evidently, Swifty is a carnivore, so ZOOKEEPER can return to rule Z9, which started everything done so far.
- Now ZOOKEEPER must check whether Swifty has a tawny color and dark spots. Assume ZOOKEEPER knows that Swifty has both features. Rule Z9 thus supports the original hypothesis that Swifty is a cheetah, and ZOOKEEPER therefore concludes that Swifty is a cheetah.

Thus, ZOOKEEPER is able to work backward through the antecedent-consequent rules, using desired conclusions to decide for what assertions it should look. A backward-moving chain develops, as dictated by the following procedure:

**Figure 7.3** Knowing something about an unknown animal enables identification via backward chaining. Here, the hypothesis that Swifty is a cheetah leads to assertions that support that hypothesis.



To identify an animal with ZOOKEEPER (backward-chaining version),

- ▷ Until all hypotheses have been tried and none have been supported or until the animal is identified,
  - ▷ For each hypothesis,
    - ▷ For each rule whose consequent matches the current hypothesis,
      - ▷ Try to support each of the rule's antecedents by matching it to assertions in working memory or by backward chaining through another rule, creating new hypotheses. Be sure to check all matching and instantiation alternatives.
      - ▷ If all the rule's antecedents are supported, announce success and conclude that the hypothesis is true.

In the example, backward chaining ends successfully, verifying the hypothesis, as shown in figure 7.3. The chaining ends unsuccessfully if any required antecedent assertions cannot be supported.

### The Problem Determines Whether Chaining Should Be Forward or Backward

Many deduction-oriented antecedent-consequent rule systems can chain either forward or backward, but which direction is better? This subsection describes several rules of thumb that may help you to decide.

Most important, you want to think about how the rules relate facts to conclusions. Whenever the rules are such that a typical set of facts can lead to many conclusions, your rule system exhibits a high degree of **fan out**, and a high degree of fan out argues for backward chaining. On the other hand, whenever the rules are such that a typical hypothesis can lead to many questions, your rule system exhibits a high degree of **fan in**, and a high degree of fan in argues for forward chaining.

- If the facts that you have or may establish can lead to a large number of conclusions, but the number of ways to reach the particular conclusion in which you are interested is small, then there is more fan out than fan in, and you should use backward chaining.
- If the number of ways to reach the particular conclusion in which you are interested is large, but the number of conclusions that you are likely to reach using the facts is small, then there is more fan in than fan out, and you should use forward chaining.

Of course, in many situations, neither fan out nor fan in dominates, leading you to other considerations:

- If you have not yet gathered any facts, and you are interested in only whether one of many possible conclusions is true, use backward chaining.

Suppose, for example, that you do not care about the identity of an animal. All you care about is whether it is a carnivore. By backward chaining from the carnivore hypothesis, you ensure that all the facts you gather are properly focused. You may ask about the animal's teeth, but you will never ask about the animal's color.

- If you already have in hand all the facts you are ever going to get, and you want to know everything you can conclude from those facts, use forward chaining.

Suppose, for example, that you have had a fleeting glimpse of an animal that has subsequently disappeared. You want to know what you can deduce about the animal. If you were to backward chain, you would waste time pursuing hypotheses that lead back to questions you can no longer answer because the animal has disappeared. Accordingly, you are better off if you forward chain.

## RULE-BASED REACTION SYSTEMS

In deduction systems, the *if* parts of some if-then rules specify combinations of assertions, and the *then* part specifies a new assertion to be deduced



# APPLICATION

## **Mycin Diagnoses Bacterial Infections of the Blood**

ZOOKEEPER is based on MYCIN, a well-known rule-based deduction system. Historically, the MYCIN system played a major role in stimulating research interest in rule-based expert systems.

MYCIN's expertise lies in the domain of bacterial infections. Physicians usually must begin antibiotic treatment for patients who have bacterial infections without knowing exactly which organism is the culprit. There is no time to wait for definitive laboratory culture evidence, which accumulates too slowly. For the desperately sick, therapy must begin at once—not 2 days from at once. This requirement leaves two choices: the physician can either prescribe a broad-spectrum drug that covers all possibilities, or she can prescribe a better, disease-specific drug.

MYCIN helps the physician to prescribe disease-specific drugs. MYCIN informs itself about particular cases by requesting information from the physician about a patient's symptoms, general condition, history, and laboratory-test results that can be obtained easily and quickly. At each point, the question MYCIN asks is determined by MYCIN's current hypothesis and the answers to all previous questions. Thus, the questions start as though taken from a checklist, but the questions then vary as evidence builds. Here is an excerpt from a sample session:

- > What is the patient's name?  
John Doe.
- > Male or female?  
Male.
- > Age?  
He is 55.
- > Have you obtained positive cultures indicating general type?  
Yes.
- > What type of infection is it?  
Primary bacteremia.
- > When did symptoms first appear?  
May 5.
- > Let's call the most recent positive culture C1. From what site was C1 taken?  
From the blood.
- > When?  
May 9.
- > Let's call the first significant organism from this culture U1. Do you know the identity of U1?  
No.
- > Is U1 a rod or a coccus or something else?  
Rod.
- > What is the gram stain of U1?  
Gram-negative.
- > Have there been positive cultures from sites that are not associated with the infections about which you are seeking advice?  
No.
- > Did U1 grow aerobically?  
Yes.
- ⋮

When it is satisfied that it has a reasonably good grasp of the situation, MYCIN announces its diagnosis:

- > My recommendation will be based on the opinion that the identity of U1 may be
  1. Pseudomonas-aeruginosa
  2. Klebsiella-pneumoniae
  3. E. coli
  4. Bacteroides-fragilis
  5. Enterobacter
  6. Proteus-nonmirabilis.

After asking a few more questions about the patient, MYCIN announces its recommendation:

- > On a scale of 0 to 4, where higher numbers indicate greater severity, how would you rate the patient's degree of sickness?  
3.
- > Does the patient have a clinically significant allergic reaction to any antimicrobial agent?  
No.
- > How much does the patient weigh?  
70 kilograms.
- > My first recommendation is as follows: to cover for items 1, 2, 3, 5, and 6, give gentamycin using a dose of 119 mg (1.7 mg/kg) q8h IV [or IM] for 10 days. Modify dose in renal failure. Also, to cover for item 4, give clindamycin using a dose of 595 mg (8.5 mg/kg) q6h IV [or IM] for 14 days. If diarrhea or other gastrointestinal symptoms develop, check for pseudomembranous colitis.

MYCIN's pool of knowledge consists of approximately 500 antecedent-consequent rules, which give MYCIN the ability to recognize about 100 causes of bacterial infections. The following rule is typical:

```
M88   If      ?x's type is primary bacteremia
        the suspected entry point of ?x is the gastrointestinal tract
        the site of the culture of ?x is one of the sterile sites
    then there is evidence that ?x is bacteroides
```

MYCIN is a backward-chaining system, because physicians prefer to think about one hypothesis at a time. By sticking with the questions that are relevant to a particular hypothetical conclusion, the questioning is guaranteed to remain relevant to that hypothesis. A forward-running system can jump around, working first toward one conclusion and then toward another, seemingly at random.

Another reason why MYCIN was designed to be a backward-chaining system is that backward chaining simplifies the creation of an English-language interface. The interface needs to deal only with answers to specific questions, rather than with free-form, imaginative text.

directly from the triggering combination. In reaction systems, which are introduced in this section, the *if* parts specify the *conditions* that have to be satisfied and the *then* part specifies an *action* to be undertaken. Sometimes, the action is to *add* a new assertion; sometimes it is to *delete* an existing assertion; sometimes, it is to execute some procedure that does not involve assertions at all.

### A Toy Reaction System Bags Groceries

Suppose that Robbie has just been hired to bag groceries in a grocery store. Because he knows little about bagging groceries, he approaches his new job by creating BAGGER, a rule-based reaction system that decides where each item should go.

After a little study, Robbie decides that BAGGER should be designed to take four steps:

- 1 The check-order step: BAGGER analyzes what the customer has selected, looking over the groceries to see whether any items are missing, with a view toward suggesting additions to the customer.
- 2 The bag-large-items step: BAGGER bags the large items, taking care to put the big bottles in first.
- 3 The bag-medium-items step: BAGGER bags the medium items, taking care to put frozen ones in freezer bags.
- 4 The bag-small-items step: BAGGER bags the small items.

Now let us see how this knowledge can be captured in a rule-based reaction system. First, BAGGER needs a working memory. The working memory must contain assertions that capture information about the items to be bagged. Suppose that those items are the items listed in the following table:

Item	Container type	Size	Frozen?
Bread	plastic bag	medium	no
Glop	jar	small	no
Granola	cardboard box	large	no
Ice cream	cardboard carton	medium	yes
Potato chips	plastic bag	medium	no
Pepsi	bottle	large	no

Next, BAGGER needs to know which step is the current step, which bag is the current bag, and which items already have been placed in bags. In the following example, the first assertion identifies the current step as the check-order step, the second identifies the bag as Bag1, and the remainder indicate what items are yet to be bagged:

Step is check-order  
 Bag1 is a bag  
 Bread is to be bagged  
 Glop is to be bagged  
 Granola is to be bagged  
 Ice cream is to be bagged  
 Potato chips are to be bagged

Note that working memory contains an assertion that identifies the step. Each of the rules in BAGGER's rule base tests the step name. Rule B1, for example, is triggered only when the step is the check-order step:

```
B1    If      step is check-order
        potato chips are to be bagged
        there is no Pepsi to be bagged
    then ask the customer whether he would like a bottle of Pepsi
```

The purpose of rule B1 is to be sure the customer has something to drink to go along with potato chips, because potato chips are dry and salty. Note that rule B1's final condition checks that a particular pattern *does not* match any assertion in working memory.

Now let us move on to a rule that moves BAGGER from the check-order step to the bag-large-items step:

```
B2    If      step is check-order
        then   step is no longer check-order
        step is bag-large-items
```

Note that the first of rule B2's actions deletes an assertion from working memory. Deduction systems are assumed to deal with static worlds in which nothing that is shown to be true can ever become false. Reaction systems, however, are allowed more freedom. Sometimes, that extra freedom is reflected in the rule syntax through the breakup of the action part of the rule, marked by *then*, into two constituent parts, marked by *delete* and *add*. When you use this alternate syntax, rule B2 looks like this:

```
B2 (add-delete form)
    If      step is check-order
    delete  step is check-order
    add     step is bag-large-items
```

The remainder of BAGGER's rules are expressed in this more transparent **add-delete syntax**.

At first, rule B2 may seem dangerous, for it looks as though it could prevent rule B1 from doing its legitimate and necessary work. There is no problem, however. Whenever you are working with a reaction system, you adopt a suitable *conflict-resolution procedure* to determine which rule

to fire among many that may be triggered. BAGGER uses the simplest conflict-resolution strategy, *rule ordering*, which means that the rules are arranged in a list, and the first rule triggered is the one that is allowed to fire. By placing rule B2 after rule B1, you ensure that rule B1 does its job before rule B2 changes the step to bag-large-items. Thus, rule B2 changes the step only when nothing else can be done.

Use of the rule-ordering conflict resolution helps you out in other ways as well. Consider, for example, the first two rules for bagging large items:

```

B3   If      step is bag-large-items
        a large item is to be bagged
        the large item is a bottle
        the current bag contains < 6 large items
      delete the large item is to be bagged
      add    the large item is in the current bag

B4   If      step is bag-large-items
        a large item is to be bagged
        the current bag contains < 6 large items
      delete the large item is to be bagged
      add    the large item is in the current bag

```

Big items go into bags that do not have too many items already, but the bottles—being heavy—go in first. The placement of rule B3 before rule B4 ensures this ordering.

Note that rules B3 and B4 contain a condition that requires counting, so BAGGER must do more than assertion matching when looking for triggered rules. Most rule-based systems focus on assertion matching, but provide an escape hatch to a general-purpose programming language when you need to do more than just match an antecedent pattern to assertions in working memory.

Evidently, BAGGER is to add large items only when the current bag contains fewer than six items.<sup>†</sup> When the current bag contains six or more items, BAGGER uses rule B5 to change bags:

```

B5   If      step is bag-large-items
        a large item is to be bagged
        an empty bag is available
      delete the current bag is the current bag
      add    the empty bag is the current bag

```

Finally, another step-changing rule moves BAGGER to the next step:

<sup>†</sup>Perhaps a better BAGGER system would use volume to determine when bags are full; to deal with volume, however, would require general-purpose computation that would make the example unnecessarily complicated, albeit more realistic.

B6    If        step is bag-large-items  
       delete   step is bag-large-items  
       add       step is bag-medium-items

Let us simulate the result of using these rules on the given database. As we start, the step is check-order. The order to be checked contains potato chips, but no Pepsi. Accordingly, rule B1 fires, suggesting to the customer that perhaps a bottle of Pepsi would be nice. Let us assume that the customer goes along with the suggestion and fetches a bottle of Pepsi.

Inasmuch as there are no more check-order rules that can fire, other than rule B2, the one that changes the step to bag-large-items, the step becomes bag-large-items.

Now, because the Pepsi is a large item in a bottle, the conditions for rule B3 are satisfied, so rule B3 puts the Pepsi in the current bag. Once the Pepsi is in the current bag, the only other large item is the box of granola, which satisfies the conditions of rule B4, so it is bagged as well, leaving the working memory in the following condition:

Step is bag-medium-items  
 Bag1 contains Pepsi  
 Bag1 contains granola  
 Bread is to be bagged  
 Glop is to be bagged  
 Ice cream is to be bagged  
 Potato chips are to be bagged

Now it is time to look at rules for bagging medium items.

B7    If        step is bag-medium-items  
               a medium item is frozen, but not in a freezer bag  
       delete   the medium item is not in a freezer bag  
       add       the medium item is in a freezer bag

B8    If        step is bag-medium-items  
               a medium item is to be bagged  
               the current bag is empty or contains only medium items  
               the current bag contains no large items  
               the current bag contains < 12 medium items  
       delete   the medium item is to be bagged  
       add       the medium item is in the current bag

B9    If        step is bag-medium-items  
               a medium item is to be bagged  
               an empty bag is available  
       delete   the current bag is the current bag  
       add       the empty bag is the current bag

Note that the fourth condition that appears in rule B8 prevents BAGGER from putting medium items in a bag that already contains a large item. If there is a bag that contains a large item, rule B9 starts a new bag.

Also note that rule B7 and rule B8 make use of the rule-ordering conflict-resolution procedure. If both rule B7 and rule B8 are triggered, rule B7 is the one that fires, ensuring that frozen things are placed in freezer bags before bagging.

Finally, when there are no more medium items to be bagged, neither rule B7 nor rule B8 is triggered; instead, rule B10 is triggered and fires, changing the step to bag-small-items:

```
B10  If      step is bag-medium-items
      delete  step is bag-medium-items
      add     step is bag-small-items
```

At this point, after execution of all appropriate bag-medium-item rules, the situation is as follows:

```
Step is bag-small-items
Bag1 contains Pepsi
Bag1 contains granola
Bag2 contains bread
Bag2 contains ice cream (in freezer bag)
Bag2 contains potato chips
Glop is to be bagged
```

Note that, according to simple rules used by BAGGER, medium items do not go into bags with large items. Similarly, conditions in rule B11 ensure that small items go in their own bag:

```
B11  If      step is bag-small-items
      a small item is to be bagged
      the current bag contains no large items
      the current bag contains no medium items
      the bag contains < 18 small items
      delete  the small item is to be bagged
      add     the small item is in the current bag
```

BAGGER needs a rule that starts a new bag:

```
B12  If      step is bag-small-items
      a small item is to be bagged
      an empty bag is available
      delete  the current bag is the current bag
      add     the empty bag is the current bag
```

Finally, BAGGER needs a rule that detects when bagging is complete:

B13    If        step is bag-small-items  
        delete    step is bag-small-items  
        add        step is done

After all rules have been used, everything is bagged:

Step is done  
 Bag1 contains Pepsi  
 Bag1 contains granola  
 Bag2 contains bread  
 Bag2 contains ice cream (in freezer bag)  
 Bag2 contains potato chips  
 Bag3 contains glop

### Reaction Systems Require Conflict Resolution Strategies

Forward-chaining deduction systems do not need strategies for conflict resolution because every rule presumably produces reasonable assertions, so there is no harm in firing all triggered rules. But in reaction systems, when more than one rule is triggered, you generally want to perform only one of the possible actions, thus requiring a **conflict-resolution strategy** to decide which rule actually fires. So far, you have learned about rule ordering:

- *Rule ordering.* Arrange all rules in one long prioritized list. Use the triggered rule that has the highest priority. Ignore the others.

Here are other possibilities:

- *Context limiting.* Reduce the likelihood of conflict by separating the rules into groups, only some of which are active at any time.
- *Specificity ordering.* Whenever the conditions of one triggered rule are a superset of the conditions of another triggered rule, use the superset rule on the ground that it deals with more specific situations.
- *Data ordering.* Arrange all possible assertions in one long prioritized list. Use the triggered rule that has the condition pattern that matches the highest priority assertion in the list.
- *Size ordering.* Use the triggered rule with the toughest requirements, where *toughest* means the longest list of conditions.
- *Recency ordering.* Use the least recently used rule.

Of course, the proper choice of a conflict resolution strategy for a reaction system depends on the situation, making it difficult or impossible to rely on a fixed conflict resolution strategy or combination of strategies. An alternative is to think about which rule to fire as another problem to be solved. An elegant example of such problem solving is described in Chapter 8 in the introduction of the SOAR problem solving architecture.



## PROCEDURES FOR FORWARD AND BACKWARD CHAINING

In this section, you learn more about rule-based systems. The focus is on how to do forward and backward chaining using well-known methods for exploring alternative variable bindings.

### Depth-First Search Can Supply Compatible Bindings for Forward Chaining

One simple way to do forward chaining is to cycle through the rules, looking for those that lead to new assertions once the consequents are instantiated with appropriate variable bindings:

---

To forward chain (coarse version),

- ▷ Until no rule produces a new assertion,
    - ▷ For each rule,
      - ▷ For each set of possible variable bindings determined by matching the antecedents to working memory,
      - ▷ Instantiate the consequent.
      - ▷ Determine whether the instantiated consequent is already asserted. If it is not, assert it.
- 

For an example, let us turn from the zoo to the track, assuming the following assertions are in working memory:

Comet	is-a	horse
Prancer	is-a	horse
Comet	is-a-parent-of	Dasher
Comet	is-a-parent-of	Prancer
Prancer	is	fast
Dasher	is-a-parent-of	Thunder
Thunder	is	fast
Thunder	is-a	horse
Dasher	is-a	horse

Next, let us agree that a horse who is the parent of something fast is valuable. Translating this knowledge into an if-then rule produces the following:

```

Parent Rule
  If    ?x is-a horse
        ?x is-a-parent-of ?y
        ?y is fast
  then ?x is valuable
  
```

# APPLICATION

## XCON Configures Computer Systems

BAGGER is based on XCON, a well-known rule-based deduction system. Historically, the XCON system played a major role in stimulating commercial interest in rule-based expert systems.

XCON's domain is computer-system components. When a company buys a big mainframe computer, it buys a central processor, memory, terminals, disk drives, tape drives, various peripheral controllers, and other paraphernalia. All these components must be arranged sensibly along input-output buses. Moreover, all the electronic modules must be placed in the proper kind of cabinet in a suitable slot of a suitable backplane.

Arranging all the components is a task called *configuration*. Doing configuration can be tedious, because a computer-component family may have hundreds of possible options that can be organized in an unthinkable number of combinations.

To do configuration, XCON uses rules such as the following:

- X1     If     the context is doing layout and assigning a power supply  
          an sbi module of any type has been put in a cabinet  
          the position that the sbi module occupies is known  
          there is space available for a power supply  
          there is no available power supply  
          the voltage and frequency of the components are known  
       then   add an appropriate power supply
- X2     If     the context is doing layout and assigning a power supply  
          an sbi module of any type has been put in a cabinet  
          the position the sbi module occupies is known  
          there is space available for a power supply  
          there is an available power supply  
       then   put the power supply in the cabinet in the available space

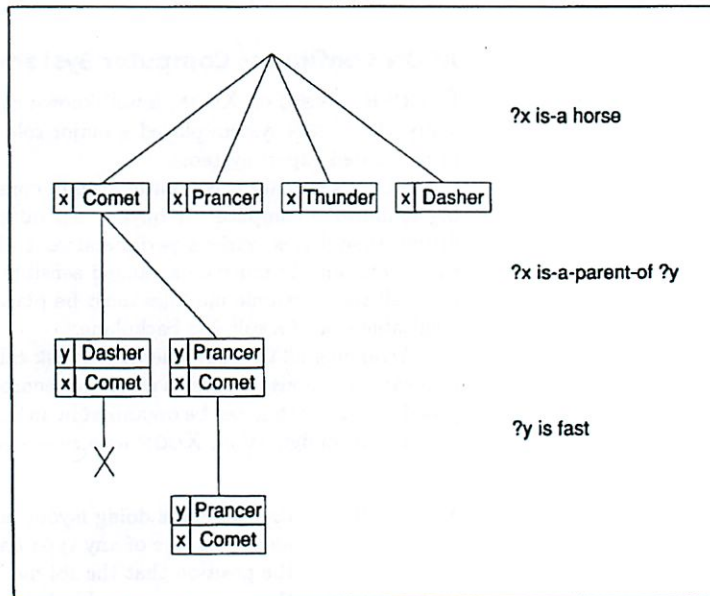
The first rule, X1, acts rather like the one in BAGGER that asks the customer whether he wants a bottle of Pepsi if the order contains potato chips but no beverage. The second rule, X2, is a typical insertion rule. The context mentioned in both rules is a combination of the top-level step and a substep. The context is changed by rules such as the following:

- X3     If     the current context is  $x$   
       then   deactivate the  $x$  context  
              activate the  $y$  context

Rule X3 has the effect of deleting one item from the context designation and adding another. It fires only if no other rule associated with the context triggers.

XCON has nearly 10,000 rules and knows the properties of several hundred component types for VAX computers, made by Digital Equipment Corporation. XCON routinely handles orders involving 100 to 200 components. It is representative of many similar systems for marketing and manufacturing.

**Figure 7.4** During forward chaining, binding commitments can be arranged in a tree, suggesting that ordinary search methods can be used to find one or all of the possible binding sets. Here, the parent rule's first antecedent leads to four possible bindings for  $x$ , and the rule's second antecedent, given that  $x$  is bound to Comet, leads to two possible bindings for  $y$ .



Now, if there is a binding for  $x$  and a binding for  $y$  such that each antecedent corresponds to an assertion when the variables are replaced by their bindings, then the rule justifies the conclusion that the thing bound to  $x$  is valuable. For example, if  $x$  is bound to Comet and  $y$  is bound to Prancer, then each of the antecedents corresponds to an assertion—namely *Comet is-a horse*, *Comet is-a-parent-of Prancer*, and *Prancer is fast*. Accordingly, Comet must be valuable.

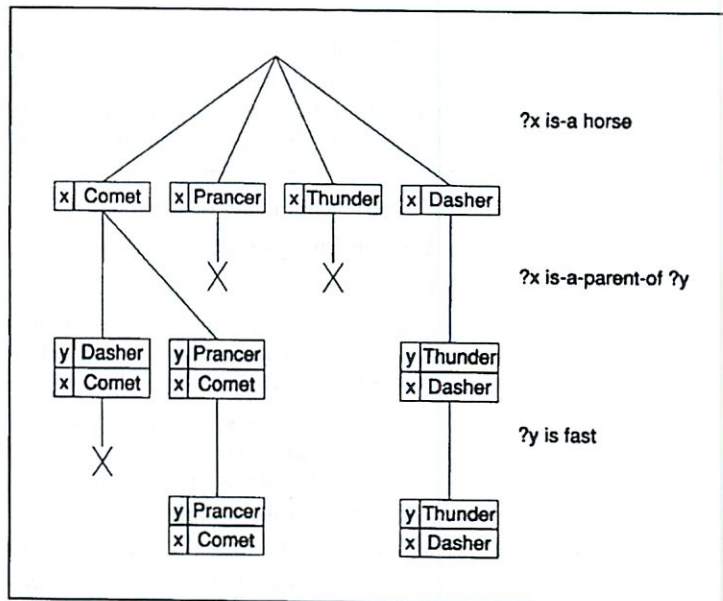
To conduct a search for binding pairs, you can start by matching the first antecedent against each assertion. As shown in figure 7.4, there are four matches and four corresponding binding choices for  $x$  in the antecedent *?x is-a horse*, because Comet, Prancer, Thunder, and Dasher are all horses.

Next, proceeding in the depth-first search style, assume that  $x$ 's binding should be Comet, which is the binding produced by the first match. Then, with  $x$  bound to Comet, the second assertion, after instantiation, is *Comet is-a-parent-of ?y*. Matching this second instantiated antecedent against each assertion produces two matches, because Comet is a parent of both Dasher and Prancer. Thus, there are two binding choices for  $y$  given that  $x$  is bound to Comet.

Figure 7.4 shows how the  $x$  and  $y$  choices fit together. Evidently, each of the two antecedents examined so far produces binding choices that can be arranged in a search tree.

Traveling along the leftmost branch, with  $x$  bound to Comet and  $y$  bound to Dasher, you proceed to the third antecedent, which becomes *Dasher is fast* when instantiated with  $y$ 's binding. This instantiated an-

**Figure 7.5** Two paths extend from the root down through the levels corresponding to three rule antecedents. Evidently, there are two binding sets that satisfy the rule.



tecedent fails to match any assertion, however, so you have to look farther for an acceptable combination. You do not have to look far, because the combination with  $x$  bound to Comet, as before, and  $y$  bound to Prancer leads to the instantiation of the third antecedent as *Prancer is fast*, which does match an assertion. Accordingly, you can conclude that the combination with  $x$  bound to Comet and  $y$  bound to Prancer is a combination that jumps over all the antecedent hurdles. You can use this combination to instantiate the consequent, producing *Comet is valuable*.

As shown in figure 7.4, there are three other choices for  $x$  bindings. Among these, if  $x$  is bound to Prancer or Thunder, then the second assertion, once instantiated, becomes *Prancer is-a-parent-of ?y* or *Thunder is-a-parent-of ?y*, both of which fail to match any assertion. If Dasher is the proper binding, then *Dasher is-a-parent-of ?y* matches just one assertion, *Dasher is-a-parent-of Thunder*, leaving only Thunder as a choice for  $y$ 's binding. With  $x$  bound to Dasher and  $y$  bound to Thunder, the third instantiated antecedent is *Thunder is fast*, which matches an assertion, leading to the conclusion, as shown in figure 7.5, that the Dasher-Thunder combination also jumps over all the hurdles, suggesting that Dasher is valuable too.

From this example, several points of interest emerge. First, you can see that each path in the search tree corresponds to a set of binding commitments. Second, each antecedent matches zero or more assertions given the bindings already accumulated along a path, and each successful match produces a branch. Third, the depth of the search tree is always equal

to the number of antecedents. Fourth, you have a choice, as usual, about how you search the tree. Exhaustive, depth-first, left-to-right search is the usual method when the objective is to find all possible ways that a rule can be deployed. This method is the one exhibited in the following procedure:

---

To forward chain (detailed version),

- ▷ Until no rule produces a new assertion,
  - ▷ For each rule,
    - ▷ Try to match the first antecedent with an existing assertion. Create a new binding set with variable bindings established by the match.
    - ▷ Using the existing variable bindings, try to match the next antecedent with an existing assertion. If any new variables appear in this antecedent, augment the existing variable bindings.
    - ▷ Repeat the previous step for each antecedent, accumulating variable bindings as you go, until,
      - ▷ There is no match with any existing assertion using the binding set established so far. In this case, back up to a previous match of an antecedent to an assertion, looking for an alternative match that produces an alternative, workable binding set.
      - ▷ There are no more antecedents to be matched. In this case,
        - ▷ Use the binding set in hand to instantiate the consequent.
        - ▷ Determine if the instantiated consequent is already asserted. If not, assert it.
        - ▷ Back up to the most recent match with unexplored bindings, looking for an alternative match that produces a workable binding set.
    - ▷ There are no more alternatives matches to be explored at any level.

---

### **Depth-First Search Can Supply Compatible Bindings for Backward Chaining**

You learned that forward chaining can be viewed as searching for variable-binding sets such that, for each set, all antecedents correspond to assertions once their variables are replaced by bindings from the set.

Backward chaining can be treated in the same general way, but there are a few important differences and complications. In particular, you start

by matching a hypothesis both against existing assertions and against rule consequents.

Suppose, for example, that you are still working with horses using the same rules and assertions in working memory as before. Next, suppose that you want to show that Comet is valuable; in other words, suppose that you want to verify the hypothesis, *Comet is valuable*. You fail to find a match for *Comet is valuable* among the assertions, but you succeed in matching the hypothesis with the rule consequent, *?x is valuable*. The success leads you to attempt to match the antecedents, presuming that *x* is bound to Comet.

Happily, the instantiated first antecedent, *Comet is-a horse*, matches an assertion, enabling a search for the instantiated second antecedent, *Comet is-a-parent-of y*. This second antecedent leads to two matches, one with the assertion *Comet is-a-parent-of Dasher* and one with the assertion *Comet is-a-parent-of Prancer*. Accordingly, the search branches, as shown in figure 7.6.

Along the left branch, *y* is bound to Dasher, leading to a futile attempt to match the third antecedent *Dasher is fast* to an assertion. Along the right branch, however, *y* is bound to Prancer, leading to a successful attempt to match *Prancer is fast* to an assertion.

Evidently, the hypothesis, *Comet is valuable*, is supported by the combination of the given rule and the given assertions because a binding set, discovered by search, connects the hypothesis with the assertions via the rule.

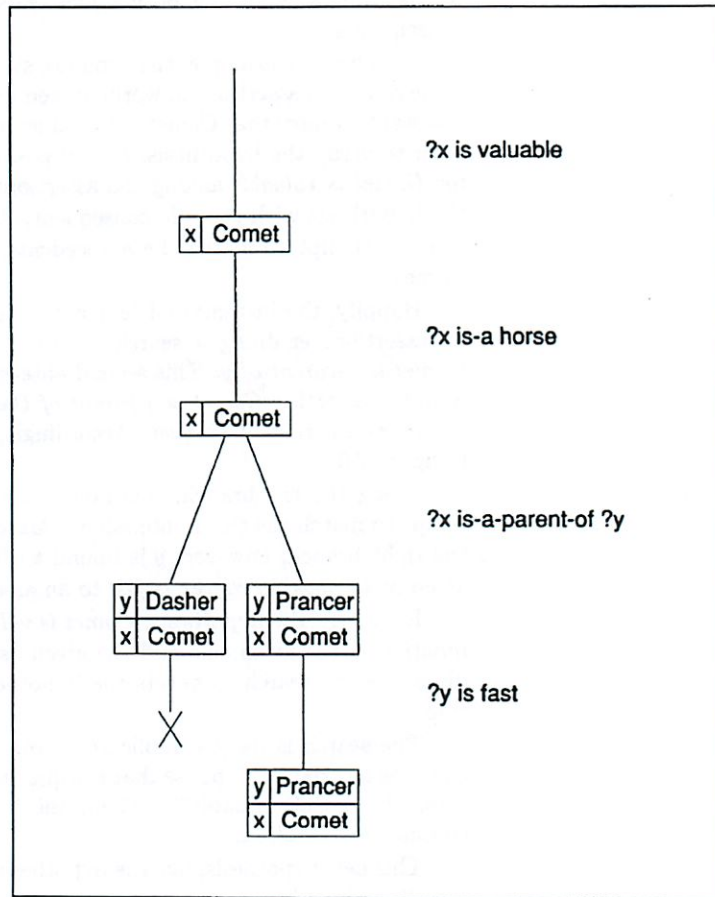
The search is more complicated, however, when the hypothesis itself contains a variable. Suppose that the question is "Who is valuable?" rather than "Is Comet valuable?" Then, the hypothesis itself, *?z is valuable*, contains a variable, *z*.

This new hypothesis, like the hypothesis *Comet is valuable*, matches no assertions but does match the consequent, *?x is valuable*. Now, however, you have a match between two variables, *z* and *x*, instead of a constant, Comet, and a variable, *x*.

Accordingly, now that it is time to match the first antecedent with the assertions, you go into the match with *z* bound to *x*. The variable *x* is not bound to anything, however, so the match of the first antecedent proceeds unfettered, as though the chaining were forward. There are four possible matches of the first antecedent to assertions, with *x* bound to any one of Comet, Prancer, Thunder, or Dasher. Then, assuming *x*'s binding should be Comet, and working through the bindings allowed by the next two assertions, you are led to one of the results shown in figure 7.7, with *z* bound to *x*, *x* bound to Comet, and *y* bound to Prancer.

The fact that *z*, the variable in the hypothesis, matches *x*, a variable in a rule, need cause you no serious pause. The only additional task you need to perform is to instantiate all the way to constants whenever you have an

**Figure 7.6** During backward chaining, as during forward chaining, binding commitments can be arranged in a tree, but the first binding commitments are established by the consequent, rather than by the first antecedent. Here, the consequent establishes one binding for *x*, and the second antecedent establishes two bindings for *y*. The binding for *x* and one of the two bindings for *y* establish that Comet is valuable.



option to continue instantiating. Thus, you first replace *z* by *x*, and then you replace *x* by Comet, producing an instantiated hypothesis of *Comet is valuable*.

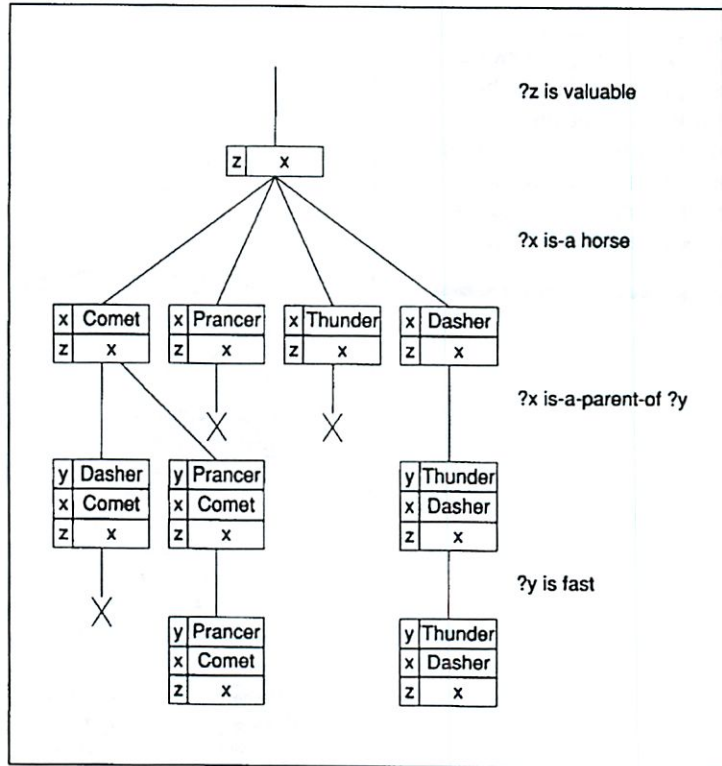
At this point, you could, if you wished, continue to look for other ways to bind variables so as to find other valuable horses.

The search is still more complicated when more than one rule can provide a variable binding. Suppose, for example, that you have the hypothesis with the variable, *?z is valuable*, but that you now add a new rule and two new assertions:

**Winner Rule**

If     ?*w* is-a winner  
 then  ?*w* is fast

**Figure 7.7** During backward chaining, hypothesis variables can be bound not only to assertion constants but also to consequent variables. Here, the hypothesis variable *z* is bound to the consequent variable *x*. Ultimately, two binding sets are found, establishing that two horses are valuable.



Dasher Is-a Winner  
Prancer Is-a Winner

Now, the search proceeds as before, as shown in figure 7.8, until it is time to find a match for the third antecedent, *?y is fast*. The first time, with *y* bound to Dasher, there is no matching assertion, but there is a match between the second rule's consequent *w is fast* and the instantiated first-rule antecedent, *Dasher is fast*. Consequently, *w* becomes bound to Dasher, and an effort is made to find a match between the second rule's instantiated antecedent, *Dasher is-a winner*, against an assertion. Because there is a match with one of the two new assertions, you can conclude that Dasher is indeed fast, which means that the original hypothesis, *?z is valuable*, can be connected via rules to assertions using the binding set with *w* and *y* bound to Dasher, *x* bound to Comet, and *z* bound to *x*. To instantiate the hypothesis with this binding set, you first replace *z* with *x*, and then replace *x* with Comet.

Note that you can gain nothing by trying to find a second match for the instantiated antecedent, *Comet is-a-parent-of y*, because the ultimate conclusion, that Comet is valuable, has already been reached. Nevertheless,





Similarly, when searching for evidence that Dasher is valuable, the instantiated antecedent, *Thunder is fast*, is not only matched to an assertion, it is matched to the consequent, *w is fast*, in the second rule. This time, however, the instantiated antecedent in the second rule, *w is-a winner*, does not match any assertion, so there remains just one way of showing that Dasher is valuable.

In summary, the backward-chaining procedure moves from the initial hypothesis, through rules, to known facts, establishing variable bindings in the process. When the initial hypothesis matches the consequent of a rule, you create a binding set. Additional bindings are added to the initial binding set as the backward-chaining procedure works on the antecedents, and still more bindings are added when the procedure chains through an antecedent to the consequent of another rule. The following procedure summarizes:

---

To backward chain,

- ▷ Find a rule whose consequent matches the hypothesis (or antecedent) and create a binding set (or augment the existing binding set).
  - ▷ Using the existing binding set, look for a way to deal with the first antecedent,
    - ▷ Try to match the antecedent with an existing assertion.
    - ▷ Treat the antecedent as an hypotheses and try to support it by backward chaining through other rules using the existing binding set.
  - ▷ Repeat the previous step for each antecedent, accumulating variable bindings, until,
    - ▷ There is no match with any existing assertion or rule consequent using the binding set established so far. In this case, back up to the most recent match with unexplored bindings, looking for an alternative match that produces a workable binding set.
    - ▷ There are no more antecedents to be matched. In this case, the binding set in hand supports the original hypothesis.
      - ▷ If all possible binding sets are desired, report the current binding set, and back up, as if there were no match.
      - ▷ If only one possible binding set is desired, report the current binding set and quit.
    - ▷ There are no more alternative matches to be explored at any level.
-

### Relational Operations Support Forward Chaining

Now it is time to look at another approach to forward chaining. First, you learn how relational database operations can handle the bookkeeping required for forward chaining. Then, you learn how the relational database operations can be arranged to produce high-speed operation.

All that you need to know about relational databases in this section is introduced as you need it. If you have not studied relational databases elsewhere, and find the introduction in this section to be too brief, read the appendix, which describes relational databases in more detail.

Now consider the Parent Rule and assertions previously used to demonstrate the search-oriented approach. Here, again, is the Parent Rule.

#### Parent Rule

```

If      ?x is-a horse
        ?x is-a-parent-of ?y
        ?y is fast
then   ?x is valuable
  
```

Now think of the assertions as though they were part of a table. In the language of relations, the assertions are recorded in a **relation**, named **Data**, whose columns are labeled with **field names**—namely **First**, **Second**, and **Third**:

First	Second	Third
Comet	is-a	horse
Prancer	is-a	horse
Comet	is-a-parent-of	Dasher
Comet	is-a-parent-of	Prancer
Prancer	is	fast
Dasher	is-a-parent-of	Thunder
Thunder	is	fast
Thunder	is-a	horse
Dasher	is-a	horse

To determine what values of  $x$  and  $y$  trigger the rule, you first determine which of the relation's records match the first antecedent in the rule, *?x is-a horse*. In the language of relations, you need to find those records whose Second field value is *is-a* and whose Third field value is *horse*. Conveniently, relational database systems include an access procedure, **SELECT**, that extracts records with specified field values from one relation to produce a new relation with fewer records. You can ask **SELECT** to pick the horses out of the **Data** relation, for example:

SELECT Data with Second = is-a and Third = horse

The result is the new relation:

First	Second	Third
Comet	is-a	horse
Prancer	is-a	horse
Thunder	is-a	horse
Dasher	is-a	horse

All you really want to know, however, is which bindings of  $x$  produce matches. Accordingly, you use another relational database access procedure, PROJECT, to isolate the appropriate field:

PROJECT Result over First

At this point, the field named First is renamed X to remind you that it consists of bindings for the  $x$  variable. The result, a single-field relation, is as follows:

A1

X
Comet
Prancer
Thunder
Dasher

Next, you determine which of the records in the data relation match the second antecedent in the rule, *?x is-a-parent-of ?y*. You need to select those records whose Second field value is *is-a-parent-of*. Then you project the results over the First and Third fields:

PROJECT [SELECT Data with Second = is-a-parent-of  
over First and Third

After renaming the field named First to X and the field named Third to Y, you have the following table:

A2

X	Y
Comet	Dasher
Comet	Prancer
Dasher	Thunder

Finally, you need to determine which of the records in the data relation match the third antecedent in the rule, *?y is fast*. Accordingly, you select those records whose Second field value is *is* and whose Third field value is *fast*, and you project the result over the First field:

```
PROJECT [SELECT Data with Second = is and Third = fast]
        over First
```

After renaming the field named First to Y, reflecting the fact that the field values are possible bindings for *y*, you have the following table:

A3

Y
Prancer
Thunder

You now have three new relations—A1, A2, and A3—corresponding to the three antecedents in the rule. The next question is, What bindings of *x* satisfy both the first and second antecedents? Or, What field values are found both in A1's X field and in A2's X field?

The JOIN operation builds a relation with records constructed by concatenating records, one from each of two source tables, such that the records match in prescribed fields. Thus, you can join A1 and A2 over their X fields to determine which values of *x* are shared. Here is the required JOIN operation:

```
JOIN A1 and A2 with X = X
```

The result is a relation in which field-name ambiguities are eliminated by concatenation of ambiguous field names with the names of the relations that contribute them:

B1 (preliminary)

X.A1	X.A2	Y
Comet	Comet	Dasher
Comet	Comet	Prancer
Dasher	Dasher	Thunder

All you really want, of course, is to find the pairs of bindings for *x* and *y* that satisfy the first two antecedents. Accordingly, you can project the preliminary B1 relation over, say, X.A1 and Y, with the following result, after renaming of the fields:

B1

X	Y
Comet	Dasher
Comet	Prancer
Dasher	Thunder

B1 now contains binding pairs that simultaneously satisfy the first two antecedents in the rule. Now you can repeat the analysis to see which of these binding pairs also satisfy the third antecedent.

To begin, you join A3 and B1 over their Y fields to determine which values of *y* are shared:

JOIN A3 and B1 with Y = Y

The result is as follows:

B2 (preliminary)

Y.A3	X	Y.B1
Prancer	Comet	Prancer
Thunder	Dasher	Thunder

Now you project to determine the pairs of bindings for *x* and *y* that satisfy not only the first two antecedents, but also the third:

B2

X	Y
Comet	Prancer
Dasher	Thunder

At this point, you know that there are two binding pairs that simultaneously satisfy all three antecedents. Inasmuch as the *then* part of the rule uses only the binding of *x*, you project B2 over the X field:

B2

X
Comet
Dasher

Thus, the parent rule is triggered in two ways: once with *x* bound to Comet, and once with *x* bound to Dasher. In a deduction system, both binding sets can be used. In a reaction system, a conflict-resolution procedure would be required to select the next action.

The only problem with the procedure that you just learned about is that it consumes a large amount of computation. If a rule has *n* antecedents, then it takes *n* SELECT and *n* PROJECT operations to produce

the A relations along with  $n - 1$  JOIN and  $n - 1$  PROJECT operations to produce the B relations. If there happen to be  $m$  rules, and if you check out each rule whenever a new assertion is added to the data relation, then you have to perform  $mn$  SELECTS,  $m(2n - 1)$  PROJECTS, and most alarmingly,  $m(n - 1)$  expensive JOINS each time a new assertion is added. Fortunately, there is another way to search for variable bindings that does not use so many operations.

### The Rete Approach Deploys Relational Operations Incrementally

You have just learned how to use relational operations to find binding sets, but the method described is an expensive way to do forward chaining, because a great deal of work has to be done to trigger a rule. Now you are ready to learn that the relational operations can be performed incrementally, as each new assertion is made, reducing both the total amount of work and the time it takes to trigger a rule once all the triggering assertions are in place.

Ordinarily, the word *rete* is an obscure synonym for net, found only in large dictionaries. In the context of forward chaining, however, the word **rete procedure** names a procedure that works by moving each new assertion, viewed as a relational record, through a rete of boxes, each of which performs a relational operation on one relation or on a few, but never on all the relations representing accumulated assertions.

The arrangement of the rete for the valuable-horse example is shown in figure 7.9.

As a new assertion is made, it becomes a single-record relation. That single-record relation is then examined by a family of SELECT operations, each of which corresponds to a rule antecedent.

In the example, the first assertion is *Comet is-a horse*. Accordingly, the following single-record relation is constructed:

New-assertion	First	Second	Third
	Comet	is-a	horse

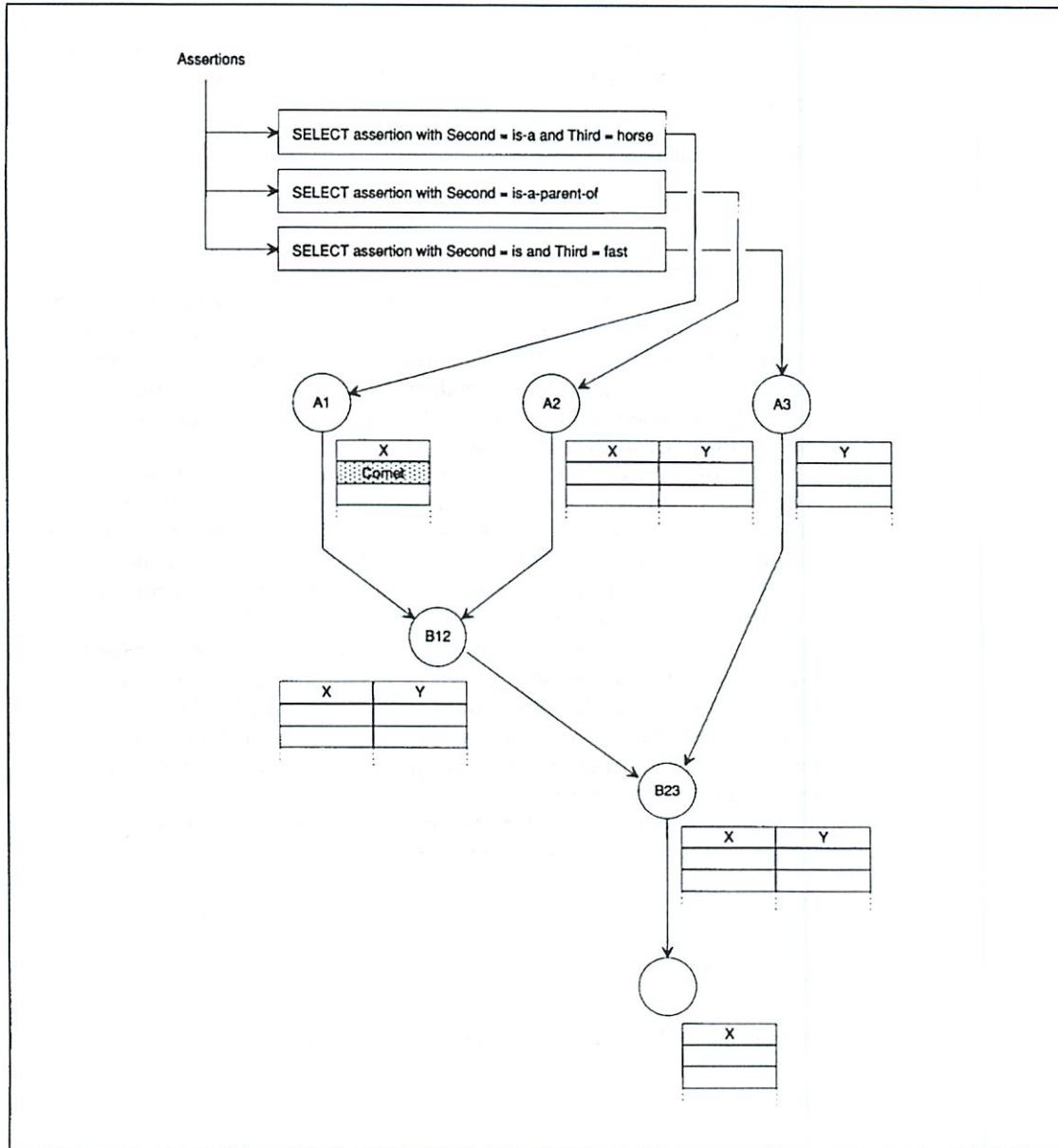
This relation is examined by three SELECT operations:

SELECT new-assertion with Second = is-a and Third = horse

SELECT new-assertion with Second = is-a-parent-of

SELECT new-assertion with Second = is and Third = fast

Next, whenever the record in the single-record relation makes it past a SELECT operation, the single-record relation endures a PROJECT operation that picks off the field or fields that contain bindings.



**Figure 7.9** The rete for a simple rule about horses with fast offspring. Here the state of the rete is captured just following the addition made in response to the first assertion. In this and other figures, the most recent changes are shown shaded.



In the example, the record makes it past the first of the three SELECT operations, whereupon the PROJECT and renaming operations produce the following:

New-assertion	X
	Comet

Once a record has gone past the SELECT, PROJECT, and renaming operations, it is added to a relation associated with a rule antecedent. Each antecedent-specific relation is located at an **alpha node** created specifically for the antecedent. Each alpha-node relation accumulates all the assertions that make it through the corresponding, filterlike SELECT operation. In the example, the selected, projected, and renamed record is added to the A1 node—the one attached to the first antecedent.

The second assertion, *Prancer is-a horse*, follows the first through the rete, and also ends up as a record in the relation attached to the A1 node. Then, the third assertion, *Comet is-a-parent-of Dasher*, following a different route, ends up as a record in the relation attached to the A2 node.

Each addition to an alpha node's relation inspires an attempt to join the added record, viewed as a single-record relation, with another relation. In particular, an addition to either A1's relation or A2's relation leads to joining of the added record, viewed as a single-record relation, with the relation attached to the other alpha node. Importantly, the JOIN operation is done with a view toward determining whether the variable binding expressed in the added record corresponds to a variable binding already established in the other relation.

In the example, the added record—the one added to A2's relation—produces the following single-record relation:

X	Y
Comet	Dasher

Meanwhile, A1's relation has accumulated two records:

A1	X
	Comet
	Prancer

Joining the two relations over the X field and projecting to eliminate one of the redundant X fields yields a one-record, two-field relation:

X	Y
Comet	Dasher

This new relation is then added to a relation attached to a **beta node**—the B12 node—so named because it is the JOIN of the A1 and A2 relations. B12's relation contains a single record that records a pair of bindings for  $x$  and  $y$  that satisfies the first and second antecedents.

Thus, an addition to either A1's relation or A2's relation leads to a JOIN operation that may add one or more records to B12's relation reflecting variable bindings that satisfy the first two rule antecedents simultaneously.

The next assertion—the fourth—*Comet is-a-parent-of Prancer*, produces the wave of activity in the rete shown by the shading in figure 7.10.

The wave starts with the addition of a second record to A2's relation. This new record, viewed as a single-record relation, is joined to A1's relation, producing a second record for B12's relation.

Because it is tiresome to append the phrase, *viewed as a relation*, each time a record, viewed as a relation, is joined with another relation, let us agree to speak of joining records with a relation, even though, strictly speaking, only relations are joined.

Next, the fifth assertion, *Prancer is fast*, initiates a wave of additions to the records in the rete and leads to a record in A3's relation. In general, an addition to A3's relation leads to joining the added record with B12's relation. This JOIN operation is done with a view toward determining whether the variable binding expressed in the added record corresponds to a variable binding already established in B12's relation.

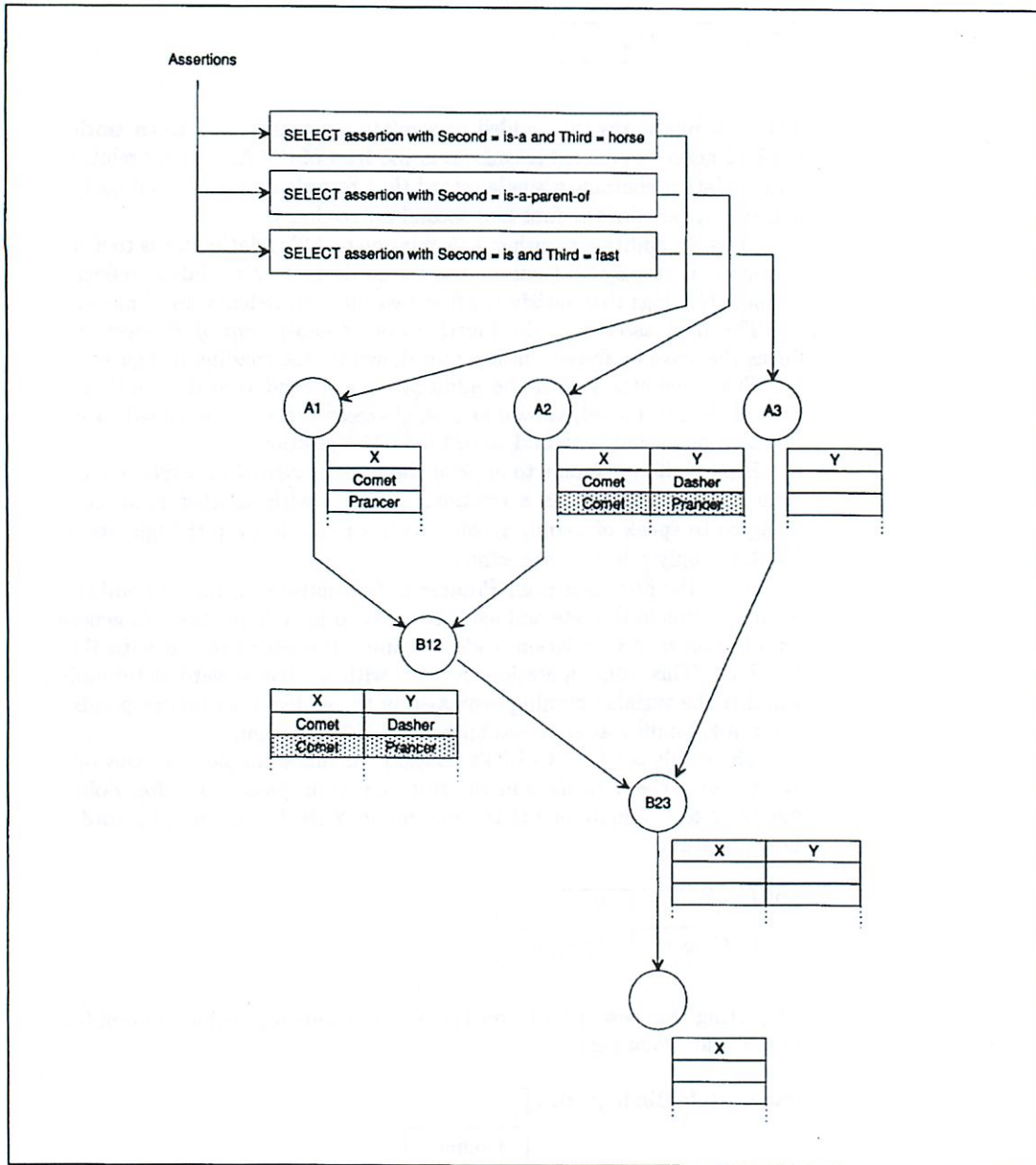
The result is added to B23's relation. In this example, the JOIN operation is over the Y fields, and the JOIN operation produces—after doing a PROJECT to eliminate one of the redundant Y fields—an initial record for B23's relation:

B23	X	Y
	Comet	Prancer

Projecting this new record over the X field yields a possible binding for  $x$  in the rule's *then* part:

Parent-Rule Bindings B23	X
	Comet

Thus, an addition to A3's relation has led to joining the added record with B12's relation. Symmetrically, of course, any new records added to B12's



**Figure 7.10** Here, the state of the rete is captured just following the additions made in response to the fourth assertion, *Comet is-a-parent-of Prancer*. The additions are shown shaded.

relation are joined to A3's relation. As before, the JOIN operation is done to determine whether the variable bindings expressed in the added records correspond to variable bindings already expressed in the other relation involved in the JOIN operation.

Now consider the state of the rete after you add three more assertions—*Dasher is-a-parent-of Thunder*, *Thunder is fast*, and *Thunder is-a horse*. A1's relation indicates that there are three horses:

A1	X
	Comet
	Prancer
	Thunder

A2's relation indicates that Comet is a parent of two children, and Dasher is a parent of one child:

A2	X	Y
	Comet	Dasher
	Comet	Prancer
	Dasher	Thunder

A3's relation indicates that Prancer and Thunder are fast:

A3	Y
	Prancer
	Thunder

Next, the information in the alpha-node relations is joined to form the beta-node relations:

B12	X	Y
	Comet	Dasher
	Comet	Prancer

B23	X	Y
	Comet	Prancer

Next, the ninth assertion, *Dasher is-a horse*, initiates another wave of additions to the records in the rete—the additions indicated by the shading in figure 7.11.

The first of these additions is the new record in A1's relation. This new record is joined to A2's relation, producing a new record in B12's relation:

B12 (increment)	X	Y
	Dasher	Thunder

But now this new B12 record is joined to A3's relation producing a new record for B23's relation:

B23 (increment)	X	Y
	Dasher	Thunder

Projection of this new record over the X field yields another possible binding for  $x$  in the rule's consequent:

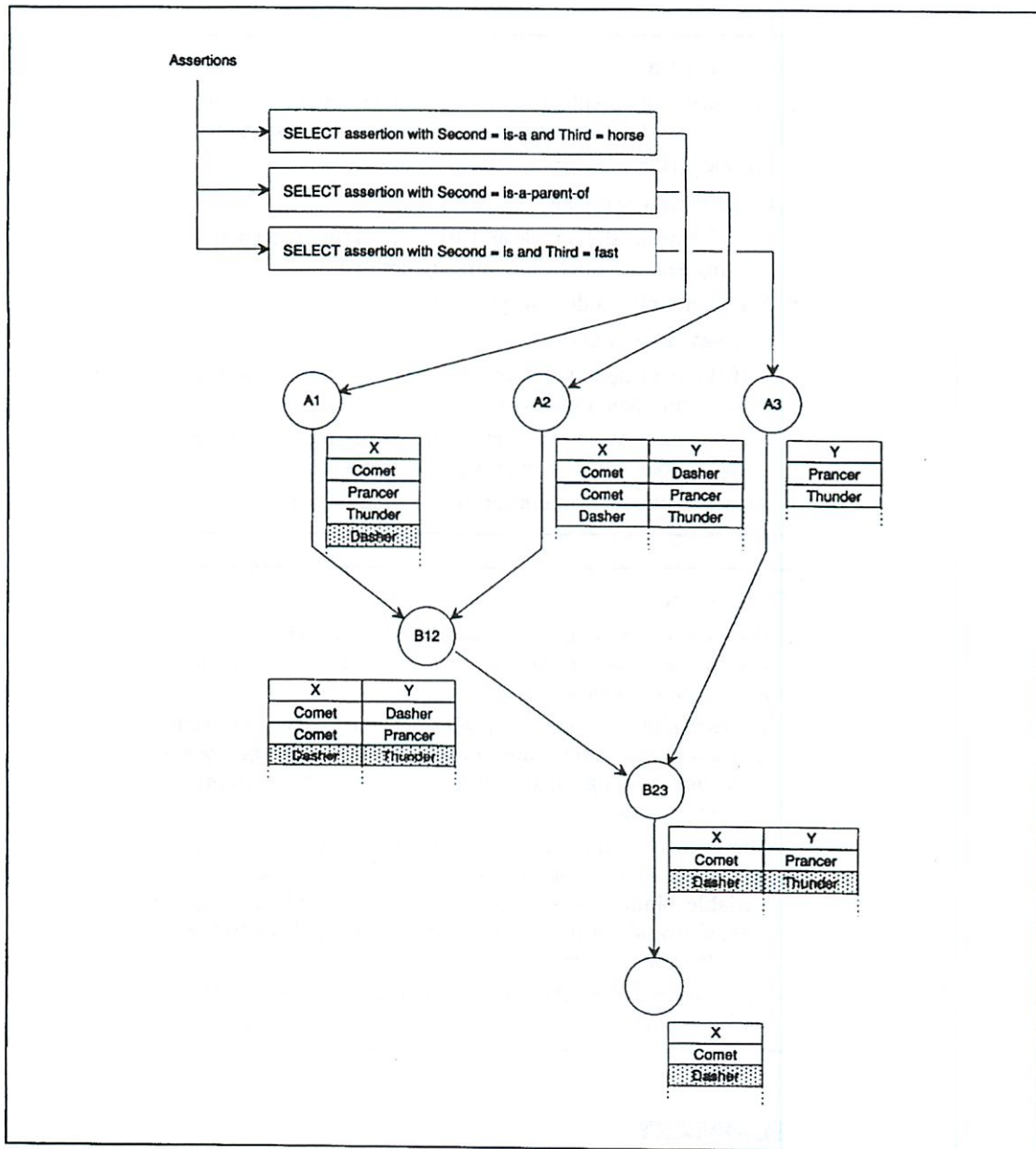
Parent-Rule Bindings (increment) B23	X
	Dasher

Thus, after all nine assertions are processed, the possible bindings for  $x$  in the rule's consequent are given by the following relation:

Parent-Rule Bindings B23	X
	Comet
	Dasher

As you have seen, adding a new relation produces a wavelike phenomenon that continues through the rete as long as JOIN operations produce new records. Note that all the relational operations involve only small relations containing a few assertions; they never involve the entire accumulated database of assertions.

Although the example may seem complicated, the procedures for building and using a rete are straightforward:



**Figure 7.11** Here, the state of the rete is captured just following the additions made in response to the ninth assertion, *Dasher is-a horse*. The most recent changes are shown shaded.

---

To construct a rete,

- ▷ For each antecedent pattern that appears in the rule set, create a SELECT operation that examines new assertions.
  - ▷ For each rule,
    - ▷ For each antecedent,
      - ▷ Create an alpha node and attach it to the corresponding SELECT operation, already created.
  - ▷ For each alpha node, except the first,
    - ▷ Create a beta node.
    - ▷ If the beta node is the first beta node, attach it to the first and second alpha nodes.
    - ▷ Otherwise, attach the beta node to the corresponding alpha node and to the previous beta node.
  - ▷ Attach a PROJECT operation to the final beta node.
- 

To use a rete,

- ▷ For each assertion, filter the assertion through the SELECT operations, passing the assertion along the rete to the appropriate alpha nodes.
  - ▷ For each alpha node receiving an assertion, use the PROJECT operation to isolate the appropriate variable bindings. Pass these new bindings, if any, along the rete to the appropriate beta nodes.
  - ▷ For each beta node receiving new variable bindings on one of its inputs, use the JOIN operation to create new variable binding sets. Pass these new variable binding sets, if any, along the rete to the next beta node or to the final PROJECT operation.
  - ▷ For each rule, use the PROJECT operation to isolate the variable bindings needed to instantiate the consequent.
- 

## SUMMARY

- Rule-based systems were developed to take advantage of the fact that a great deal of useful knowledge can be expressed in simple if-then rules.
- Many rule-based systems are deduction systems. In these systems, rules consist of antecedents and consequents. In one example, a toy deduction system identifies animals.

- Deduction systems may chain together rules in a forward direction, from assertions to conclusions, or backward, from hypotheses to questions. Whether chaining should be forward or backward depends on the problem.
- Many rule-based systems are reaction systems. In these systems, rules consist of conditions and actions. A toy reaction system bags groceries.
- Reaction systems require conflict-resolution strategies to determine which of many triggered rules should be allowed to fire.
- Depth-first search can supply compatible bindings for both forward chaining and backward chaining.
- Relational operations support breadth-first search for compatible bindings during forward chaining. The rete procedure performs relational operations incrementally as new assertions flow through a rule-defined rete.

## BACKGROUND

The rete procedure was developed by C. L. Forgy [1982].

MYCIN was developed by Edward Shortliffe and colleagues at Stanford University [1976].

XCON was developed to configure the Digital Equipment Corporation's VAX computers by John McDermott and other researchers working at Carnegie Mellon University, and by Arnold Kraft, Dennis O'Connor, and other developers at the Digital Equipment Corporation [McDermott 1982].



## 6.034 Tutorial 2

9/26

- Quiz 1: Rule based + Search Algorithms  
Systems

Lab 1 + Lab 2

Prior quizzes online for practice

- Lab 2 due Fri (✓) Done

- implement every search

- do it before quiz

- Quiz is open book, note, everything printed

- except PCs/phones

Print my code from  
p-sets

---

2 chaps on search

- both on online

- printed handout

---

Have basic DFS / BFS

- can add

- extended list

- distance heuristic

- dynamic programming

②

Optimal searches: B+B A\*

- use length of path looked at so far

---

B+B

- sort entire path w/ lowest cost

⋮

---

A\*

B+B but w/ estimate of remaining distance

- if correct, A\* has optimal solution  
↳ Under estimate

- And with dynamic programming



Start at S

(SA 1) (SBA 3)

↳ don't consider longer than SA 1

③

WP DP

Solve sub problems

like Dijkstra's algorithm

If  $A$  is node on min path  $P \rightarrow Q$

So we now min path  $P \rightarrow A$

If 2 or more paths meet - just use min path

Handout  
Search  
Algorithms

If the path through E had not worked, then the procedure would move still farther back up the tree, seeking another viable decision point from which to move forward. On reaching A, the procedure would go down again, reaching the goal through D.

Having learned about depth-first search by way of an example, you can see that the procedure, expressed in procedural English, is as follows:

---

To conduct a depth-first search,

- ▷ Form a one-element queue consisting of a zero-length path that contains only the root node.
  - ▷ Until the first path in the queue terminates at the goal node or the queue is empty,
    - ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - ▷ Reject all new paths with loops.
    - ▷ Add the new paths, if any, to the *front* of the queue.
  - ▷ If the goal node is found, announce success; otherwise, announce failure.
- 

#### **Breadth-First Search Pushes Uniformly into the Search Tree**

As shown in figure 4.4, **breadth-first search** checks all paths of a given length before moving on to any longer paths. In the example, breadth-first search discovers a complete path to node G on the fourth level down from the root level.

A procedure for breadth-first search resembles the one for depth-first search, differing only in where new elements are added to the queue:

---

To conduct a breadth-first search,

- ▷ Form a one-element queue consisting of a zero-length path that contains only the root node.
  - ▷ Until the first path in the queue terminates at the goal node or the queue is empty,
    - ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - ▷ Reject all new paths with loops.
    - ▷ Add the new paths, if any, to the *back* of the queue.
  - ▷ If the goal node is found, announce success; otherwise, announce failure.
-

## HEURISTICALLY INFORMED METHODS

Search efficiency may improve spectacularly if there is a way to order the choices so that the most promising are explored earliest. In many situations, you can make measurements to determine a reasonable ordering. In the rest of this section, you learn about search methods that take advantage of such measurements; they are called **heuristically informed methods**.

### Quality Measurements Turn Depth-First Search into Hill Climbing

To move through a tree of paths using **hill climbing**, you proceed as you would in depth-first search, except that you order your choices according to some heuristic measure of the remaining distance to the goal. The better the heuristic measure is, the better hill climbing will be relative to ordinary depth-first search.

Straight-line, as-the-crow-flies distance is an example of a heuristic measure of remaining distance. Figure 4.5 shows the straight-line distances from each city to the goal; Figure 4.6 shows what happens when hill climbing is used on the map-traversal problem using as-the-crow-flies distance to order choices. Because node D is closer to the goal than is node A, the children of D are examined first. Then, node E appears closer to the goal than is node A. Accordingly, node E's children are examined, leading to a move to node F, which is closer to the goal than node B. Below node F, there is only one child: the goal node G.

From a procedural point of view, hill climbing differs from depth-first search in only one detail; there is an added step, shown in italic type:

---

To conduct a hill-climbing search,

- ▷ Form a one-element queue consisting of a zero-length path that contains only the root node.
  - ▷ Until the first path in the queue terminates at the goal node or the queue is empty,
    - ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - ▷ Reject all new paths with loops.
    - ▷ *Sort the new paths, if any, by the estimated distances between their terminal nodes and the goal.*
    - ▷ Add the new paths, if any, to the *front* of the queue.
  - ▷ If the goal node is found, announce success; otherwise, announce failure.
-

Handout 2

### Beam Search Expands Several Partial Paths and Purges the Rest

**Beam search** is like breadth-first search in that it progresses level by level. Unlike breadth-first search, however, beam search moves downward only through the best  $w$  nodes at each level; the other nodes are ignored. Consequently, the number of nodes explored remains manageable, even if there is a great deal of branching and the search is deep. Whenever beam search is used, there are only  $w$  nodes under consideration at any depth, rather than the exponentially explosive number of nodes with which you must cope whenever you use breadth-first search. Figure 4.8 illustrates how beam search would handle the map-traversal problem.

### Best-First Search Expands the Best Partial Path

Recall that, when forward motion is blocked, hill climbing demands forward motion from the most recently created open node. In **best-first search**, forward motion is from the best open node so far, no matter where that node is in the partially developed tree.

In the example map-traversal problem, hill climbing and best-first search coincidentally explore the search tree in the same way.

The paths found by best-first search are likely to be shorter than those found with other methods, because best-first search always moves forward from the node that seems closest to the goal node. Note that *likely to be* does not mean *certainly are*, however.

### Search May Lead to Discovery

Finding physical paths and tuning parameters are only two applications for search methods. More generally, the nodes in a search tree may denote abstract entities, rather than physical places or parameter settings.

Suppose, for example, that you are wild about cooking, particularly about creating your own omelet recipes. Deciding to be more systematic about your discovery procedure, you make a list of *ingredient transformations* for varying your existing recipes:

- Replace an ingredient with a similar ingredient.
- Double the amount of an ingredient.
- Halve the amount of an ingredient.
- Add a new ingredient.
- Eliminate an ingredient.

Naturally, you speculate that most of the changes suggested by these ingredient transformations will turn out to taste awful, and thus to be unworthy of further development.

# Search

9/27

Name		Backtrack	New items	Heuristic	Optimal Path	Extended	Stopping
DFS	Go down path	✓	Front Stack	X	X	✓ + Enqueue	When expand goal node
BFS	Across all nodes at depth	X	Back Queue	X	X	Big Difference	
Hill climbing	DFS + Heuristic - dist to goal	✓	Front Sorted	✓	Telephone Problem X Ridge Problem	✓	
Beam	Breadth 1st, but lim # from each depth	✓	Back sort all from that level + but in back	✓	X	✓	
British Museum	Do all paths, at same time	X	Most likely BFS all paths	X	✓	X	
Branch + Bound	PFS, extend shortest so far	✓		X though path-length	✓	X	Not still look for shorter paths
B + B + Extended	Is like DFS	✓		X though path length	✓	✓	Stop when 711 or reach 6
A*	B + B + Extended + Heuristic Least cost in front - So DFS/BFS does not matter	✓		path length + heuristic	✓ if heuristic admissible		When expand goal will always be best

6034 trick qu

1/27

2009 q1 trick qu

When variable Milcent snags (7)

Look for assertions X knows —

anything and fill w/ lot possible

- Or make branch + try w/ all possible

So statement is false!

- they never say!



### Underestimates and Dynamic Programming Improve Branch-and-Bound Search

The A\* procedure is branch-and-bound search, with an estimate of remaining distance, combined with the dynamic-programming principle. If the estimate of remaining distance is a lower-bound on the actual distance, then A\* produces optimal solutions. Generally, the estimate may be assumed to be a lower bound estimate, unless specifically stated otherwise, implying that A\*'s solutions are normally optimal. Note the similarity between A\* and branch-and-bound search with dynamic programming:

---

To conduct A\* search,

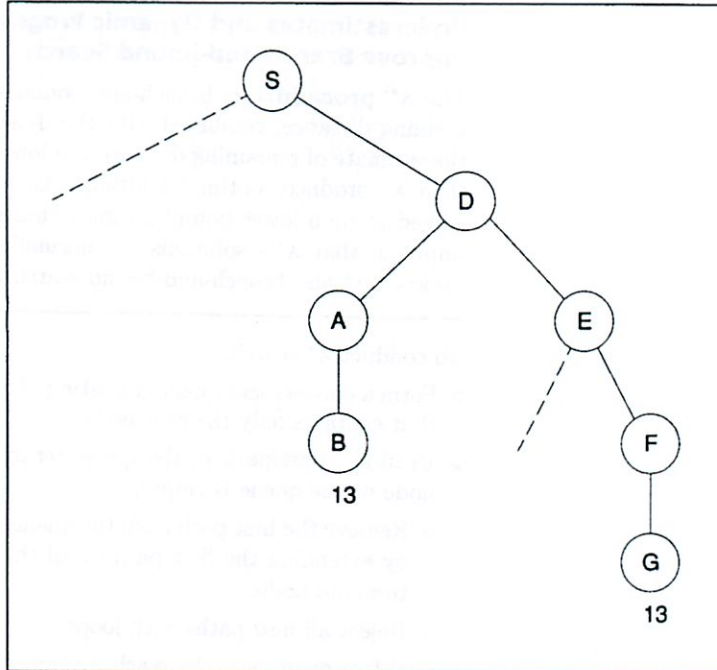
- ▷ Form a one-element queue consisting of a zero-length path that contains only the root node.
  - ▷ Until the first path in the queue terminates at the goal node or the queue is empty,
    - ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
    - ▷ Reject all new paths with loops.
    - ▷ If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
    - ▷ Sort the entire queue by the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front.
  - ▷ If the goal node is found, announce success; otherwise, announce failure.
- 

### Several Search Procedures Find the Optimal Path

You have seen that there are many ways to search for optimal paths, each of which has advantages:

- The British Museum procedure is good only when the search tree is small.
- Branch-and-bound search is good when the tree is big and bad paths turn distinctly bad quickly.
- Branch-and-bound search with a guess is good when there is a good lower-bound estimate of the distance remaining to the goal.
- Dynamic programming is good when many paths converge on the same place.
- The A\* procedure is good when both branch-and-bound search with a guess and dynamic programming are good.

**Figure 5.1** The length of the complete path from S to G, S-D-E-F-G is 13. Similarly, the length of the partial path S-D-A-B also is 13 and any additional movement along a branch will make it longer than 13. Accordingly, there is no need to pursue S-D-A-B any further because any complete path starting with S-D-A-B has to be longer than a complete path already known. Only the other paths emerging from S and from S-D-E have to be considered, as they may provide a shorter path.



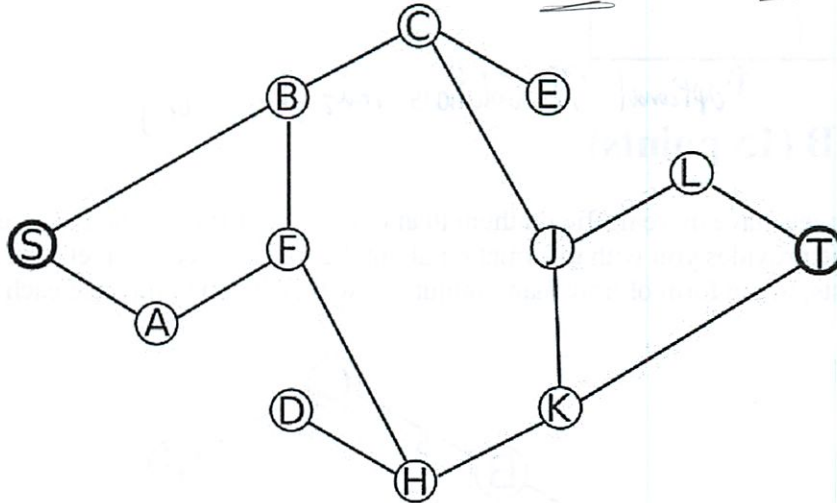
To conduct a branch-and-bound search,

- ▷ Form a one-element queue consisting of a zero-length path that contains only the root node.
- ▷ Until the first path in the queue terminates at the goal node or the queue is empty,
  - ▷ Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - ▷ Reject all new paths with loops.
  - ▷ Add the remaining new paths, if any, to the queue.
  - ▷ Sort the entire queue by path length with least-cost paths in front.
- ▷ If the goal node is found, announce success; otherwise, announce failure.

Now look again at the map-traversal problem, and note how branch-and-bound works when started with no partial paths. Figure 5.2 illustrates the exploration sequence. In the first step, the partial-path distance of S-A is found to be 3, and that of S-D is found to be 4; partial path S-A is

## Problem 2: Search (50 Points)

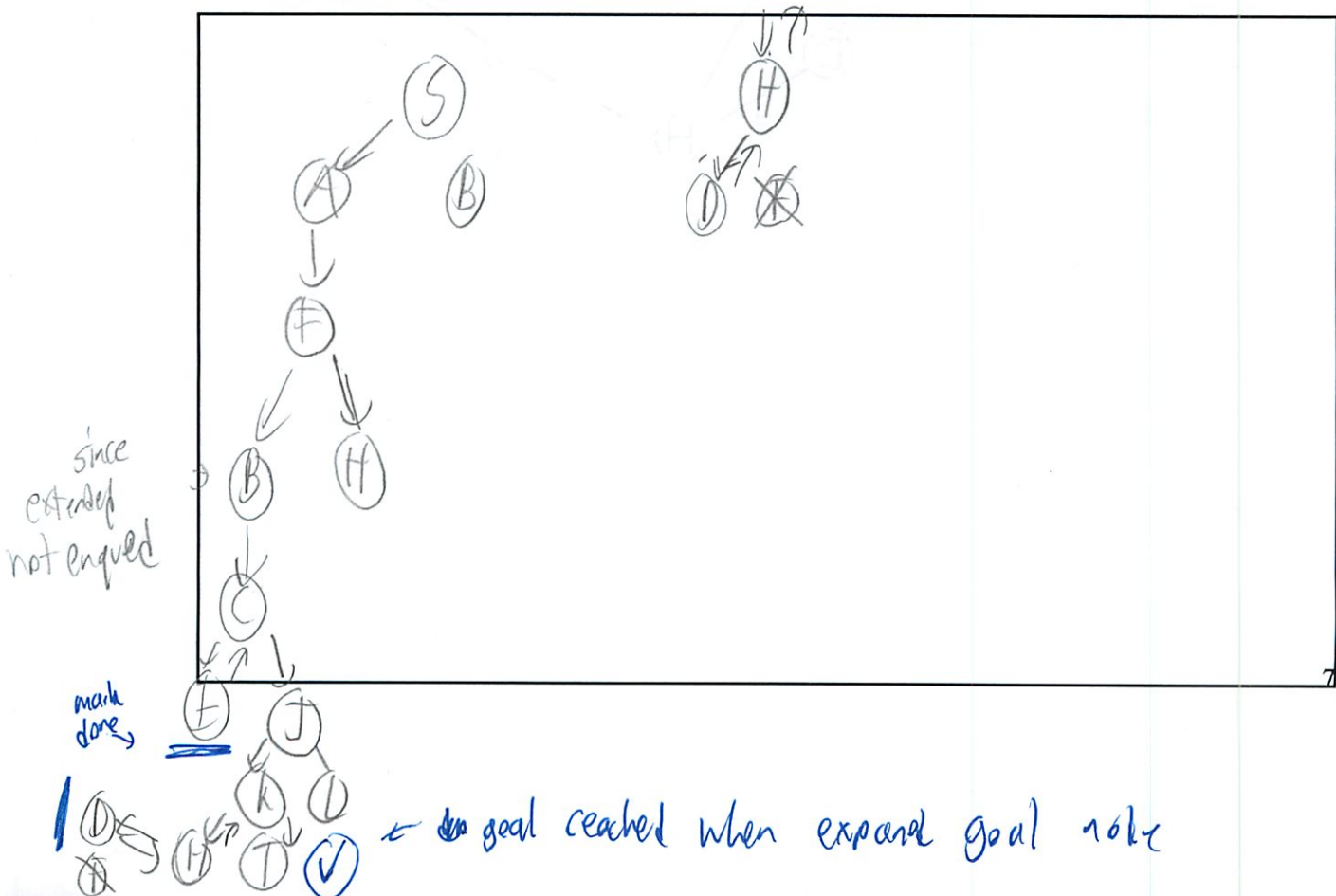
You have just moved to a strange new city, and you are trying to learn your way around. Most importantly, you want to learn how to get from your home at S to the subway at T.



In all search problems, use alphabetical order to break ties when deciding the priority to use for extending nodes.

### Part A (15 points)

Using depth-first search with backtracking and with an extended list, draw that part of the search tree that is explored by the search.



best

What is the final path found from the start (S) to the goal (T)?

S A F B C J K T ✓

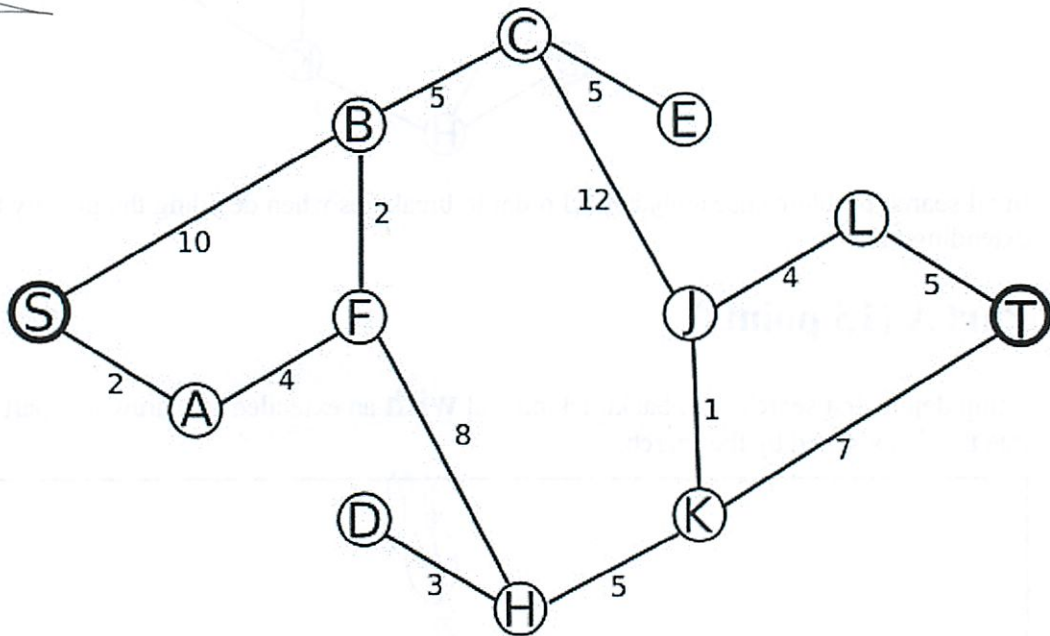
List the nodes at which you have to backtrack:

E O H

↑ optional ? solutions may be wrong

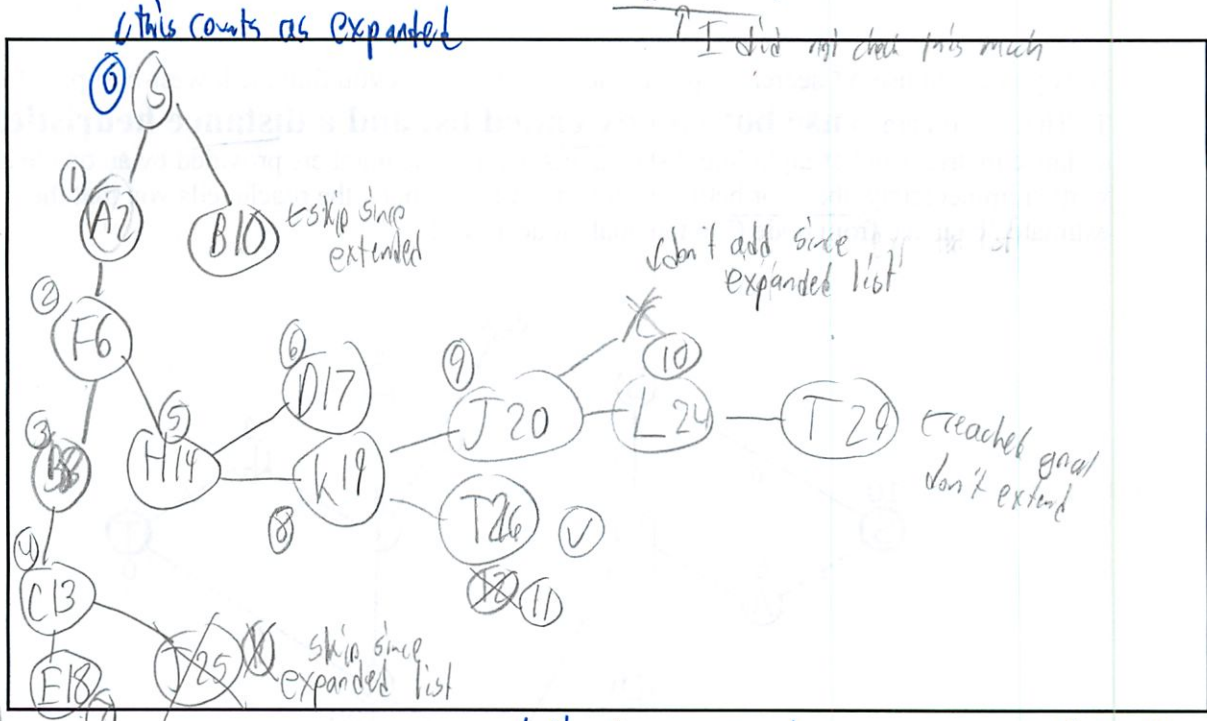
### Part B (15 points)

Some streets have more traffic on them than others. Your friend who has lived in this city for a long time provides you with information about the traffic on each street - the streets are labeled with costs, in the form of how many minutes it will take you to traverse each one.



Using these given path costs, you are to find the lowest-cost path from S to T using branch-and-bound **with an extended list but with no distance heuristic**. First draw the search tree. **Number each node as it is expanded, from 1 to n.**

- B+B is what
- path value from start
- expanded lowest path
- always expand lowest anywhere
- where to stop (should get notes)



*Write down nodes if on expanded list, just don't expand it*

Now identify the shortest path:

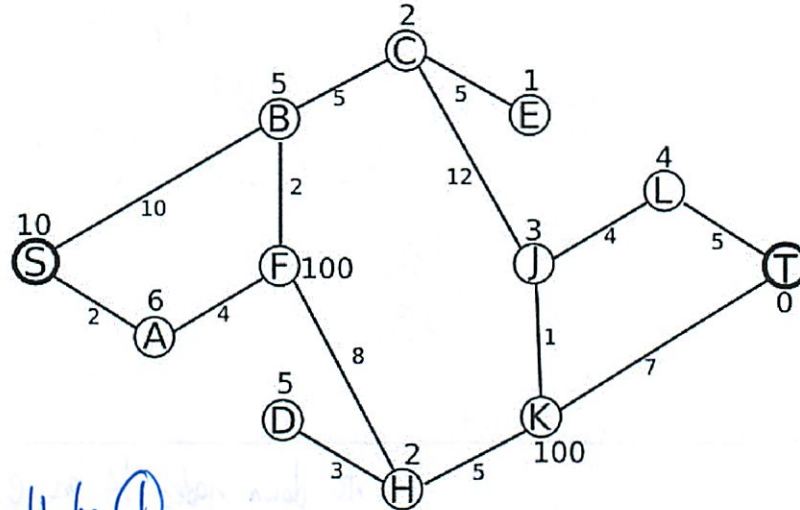
S A F H k T ✓

After you have found a path to T, which nodes must you still expand before you can be certain that you have found the shortest path to T?

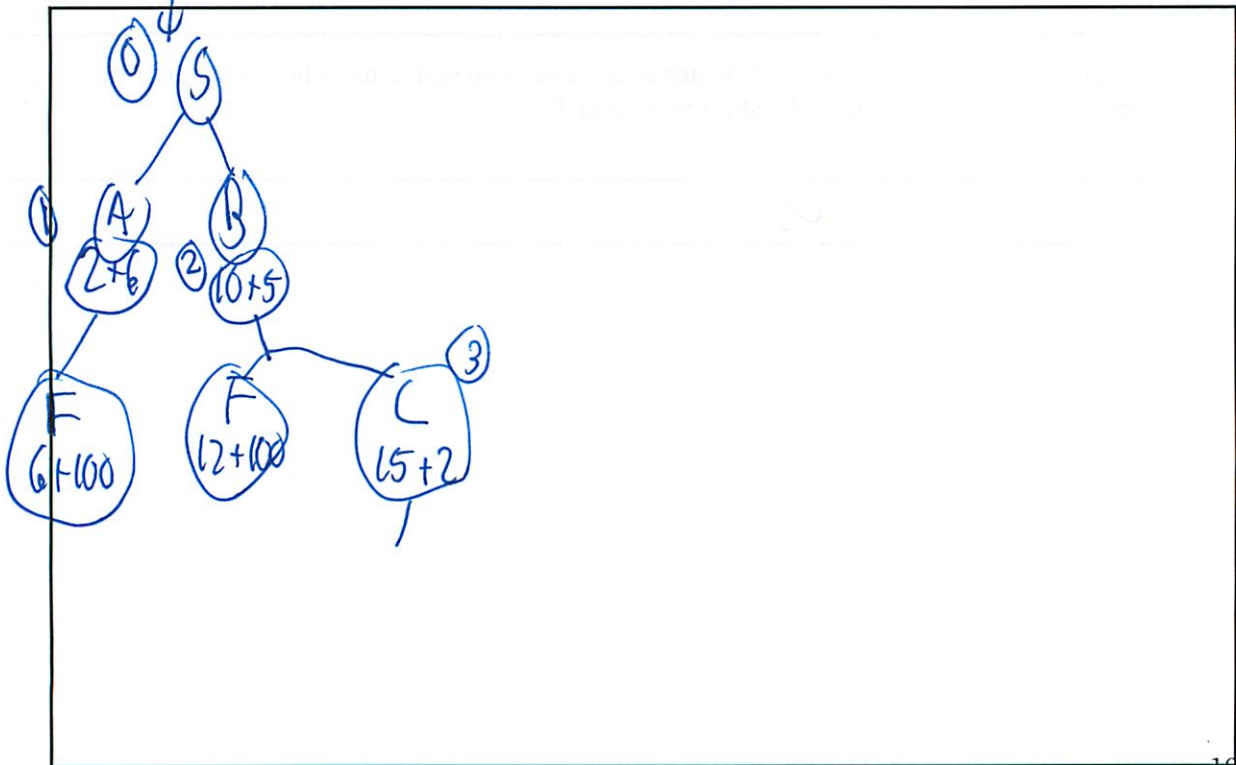
J L ✓

### Part C: (20 points)

Now you are to use A\* search, expecting to do less work as you find the lowest-cost path from S to T. That is, you are to **use both an extended list and a distance heuristic**. The distance metric is not straight line distance; instead use the numbers provided by an oracle and written immediately above or below each node. For example, the oracle tells you that the estimated distance from node C to the goal, node T, is 2.



First draw the search tree. <sup>Should be ①</sup> Number each node as it is expanded, from 1 to n.



Now, show the path you have found:

S B C J C T

If the path you found using A\* is the same as the path you found in Part B, explain in detail why it must be the same; if the path is not the same, explain why your answers are different.

Heuristic not admissible  
explain why:--

Represent problems by modeling



Humans smart since can put 2+2 together

Reduce hard problems to simpler ones

Like symbolic integrator - reduce problems

- safe transformations

- also risky

~~AND~~

- all must be true

OR

Forward chaining system

- (I should read/print formal description of)

- go down rules looking for assumptions

- when match, restart

What is 'intelligence'?

Once you understand how something works, is it less 'intelligent'?



②

Blocks stacking

- building goal tree

Complexity (behavior) =  $f$  (complexity (program), complexity (env))

Mycin program

Zoo animals

Expert system = Backwards chaining

Packing a grocery bag

deduction - only add assertion

production - add + delete

Backwards chaining

- go through assertions and build tree

(? do a practice exam?)

Firing - run + add assertion to db

Bubble up true in BC

- like for AND must all be true

③

### Search

(should make a table)

FC tutorial when 2 variables

- when find 1st, find that one
- if true, add, restart from top of rules

Search - assume bidirectional if not told

Admissible  $H(x, G) \leq D(x, G)$

Consistency  $|| H(x, G) - H(y, G) || \leq D(x, y)$   
 actual min distance

- map always consistent

# 6.034 Quiz 1

## 28 September 2011

Name	Michael Plasmeier
Email	theplaz@mit.edu

Circle your TA and recitation time (**for 1 extra credit point**), so that we can more easily enter your score in our records and return your quiz to you promptly.

TAs		Thu	Fri
		Time	Instructor
Avril Kenney	Adam Mustafa	1-2	Bob Berwick
Darryl Jones	Erek Speed	2-3	Bob Berwick
Gary Planthaber	Caryn Krakauer	3-4	Bob Berwick
Peter Brin	Tanya Kortz		

This semester I am taking 9 subjects with substantial final projects or papers.

define substantial  
I am in 7 subjects  
I would prob say 5

Problem number	Maximum	Score	Grader
1	45	43	P
2	45	41	AFK
3	10	10	AFK
Total	100	95	AFK

**There are 12 pages in this quiz, including this one, but not including blank pages and tear-off sheets. Tear-off sheets are provided at the end with duplicate drawings and data. As always, open book, open notes, open just about everything, including a calculator, but no computers.**

Advice read qv  
3 is easy

# Problem 1: Rule Systems (45 points)

After the first few weeks of 6.034, you realize that artificial intelligence is the most exciting topic you have ever studied, and you decide to start an AI club. To get more people to join the club, you organize an info session, which will (of course) have free food. You need to figure out how much food to order, so you write the following rule system to determine who will attend the info session.

## Rules:

P0	IF(OR('( ?x) lives in Burton Conner', '( ?x) lives in East Campus', '( ?x) lives in Random'), THEN('( ?x) does not buy a dining plan'))
P1	IF('( ?x) does not buy a dining plan', THEN('( ?x) wants free food'))
P2	IF('( ?x) has class in Stata', THEN('( ?x) saw a poster'))
P3	IF(OR('( ?x) saw a poster', '( ?x) is in 6.034', AND('( ?x) is friends with ( ?y)', '( ?y) is in 6.034')), THEN('( ?x) is interested'))
P4	IF(AND('( ?x) is interested', '( ?x) wants free food'), THEN('( ?x) will attend'))

## Assertions:

'Anna is in 6.034'  
'Anna lives in Maseeh'  
'Ben lives in East Campus'  
'Ben has class in Stata'  
'Chris is friends with Anna'  
'Chris wants free food'

## Part A: Backward Chaining (25 points)

Make the following assumptions about backward chaining:

- When working on a hypothesis, the backward chainer tries to find a matching assertion in the list of assertions. If no matching assertion is found, the backward chainer tries to find a rule with a matching consequent. If no matching consequent is found, then the backward chainer assumes the hypothesis is false.
- The backward chainer never alters the list of assertions, so it can derive the same result multiple times.
- Rules are tried in the order they appear.
- Antecedents are tried in the order they appear.
- Lazy evaluation/short circuiting is in effect (e.g., if the first part of an AND clause is false, the rest does not need to be evaluated to determine that the whole clause is false).

ohh

also for DR

**A1 (17 points)**

You want to know whether Anna will attend the info session. Use backward chaining to check the hypothesis 'Anna will attend'. Use the space provided **on the next page** to draw the goal tree that is created by backward chaining (we will use the goal tree to help us assign partial credit). Then, list, in the order that they are checked, all the hypotheses the backward chainer looks for in the course of checking the hypothesis 'Anna will attend'. The table may have more lines than you need.

Anna will attend
Anna is interested
Anna saw a poster
Anna has class in Statu
Anna is in 6.034
Anna wants free food
Anna does not buy a dining plan
Anna lives in BC
Anna lives in EC
Anna lives in Qanton

**A2 (8 points)**

If you were able to show 'Anna will attend', which assertions can you remove without changing the answer:

Also, identify which rules can you remove without changing the answer:

If you were not able to show 'Anna will attend', if it is possible to add a new *assertion* that would change your answer (other than adding the hypothesis itself), state such an assertion:

Anna lives in BC ← better: Anna wants free food

Also, if it is possible to add a new *rule* that would change your answer, state such a rule.

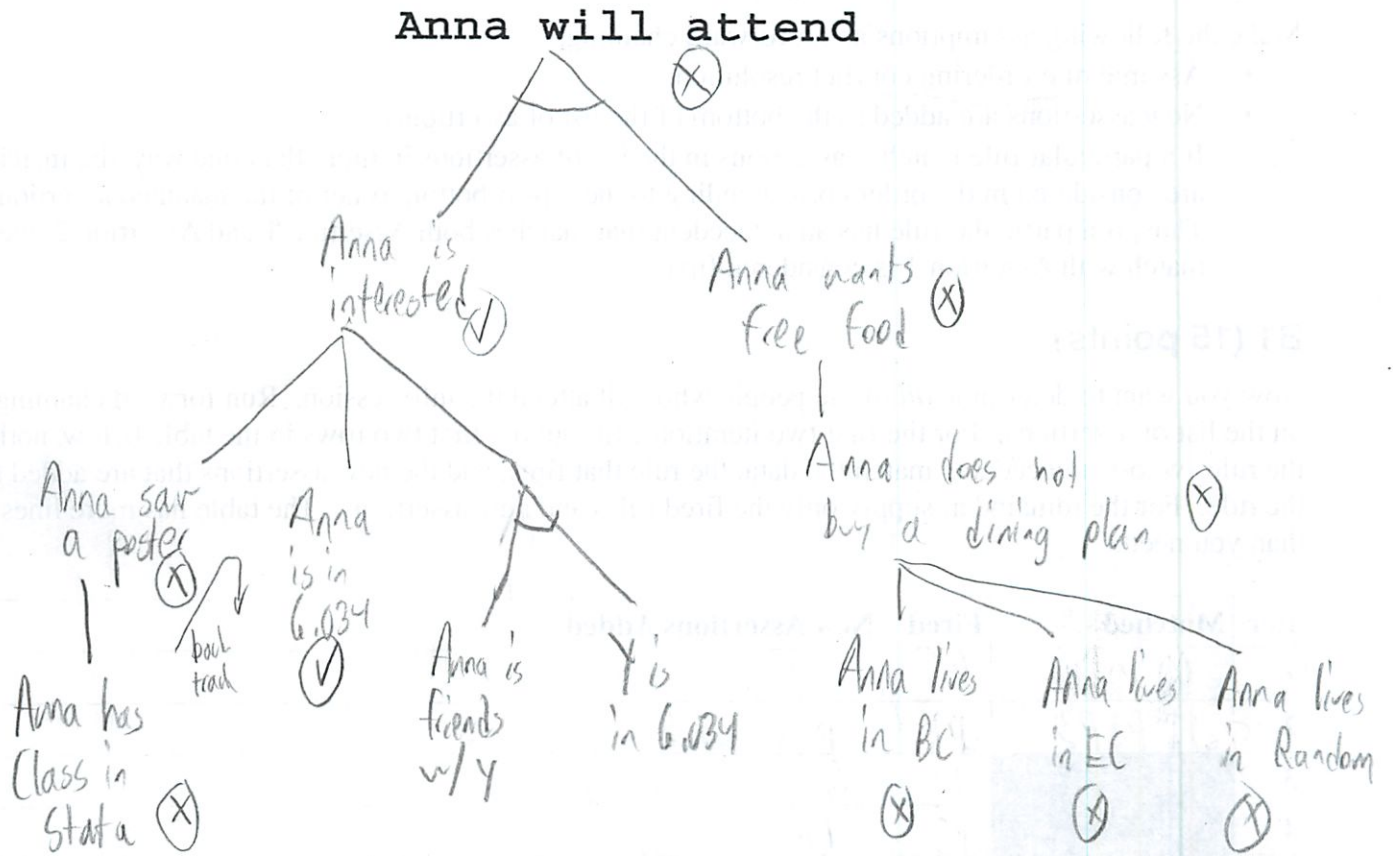
IF x lives in Maseeh

Then x wants free food

would conflict, but system would not know, TA said was fine → System would be true

DFS

Use this space to draw your goal tree.



## Part B: Forward Chaining (20 points)

Make the following assumptions about forward chaining:

- Assume rule-ordering conflict resolution.
- New assertions are added to the bottom of the list of assertions.
- If a particular rule matches assertions in the list of assertions in more than one way, the matches are considered in the order corresponding to the top-to-bottom order of the matched assertions. Thus, if a particular rule has an antecedent that matches both Assertion 1 and Assertion 2, the match with Assertion 1 is considered first.

### B1 (15 points)

Now you want to determine *all* of the people who will attend the info session. Run forward chaining on the list of assertions. For the first two iterations, fill out the first two rows in the table below, noting the rules whose antecedents match the data, the rule that fires, and the new assertions that are added by the rule. For the remainder, supply only the fired rules and new assertions. The table has more lines than you need.

Iter	Matched	Fired	New Assertions Added
1	P0, P2, P3	P0	Ben does not buy a dining plan
2	P1, P2, P3	P1	Ben wants free food
3		P2	Ben saw a poster
4		P3	Anna is interested
5		P3	Ben is interested
6		P3	Chris is interested
7		P4	Ben will attend
8		P4	Chris will attend
9			
10			

↳ ensure order of

How many people will attend the info session? 2

### B2 (5 points)

Suppose that you moved P1 to the end of the list of rules.

Then would forward-chaining conclude that *Ben* will attend the info session?

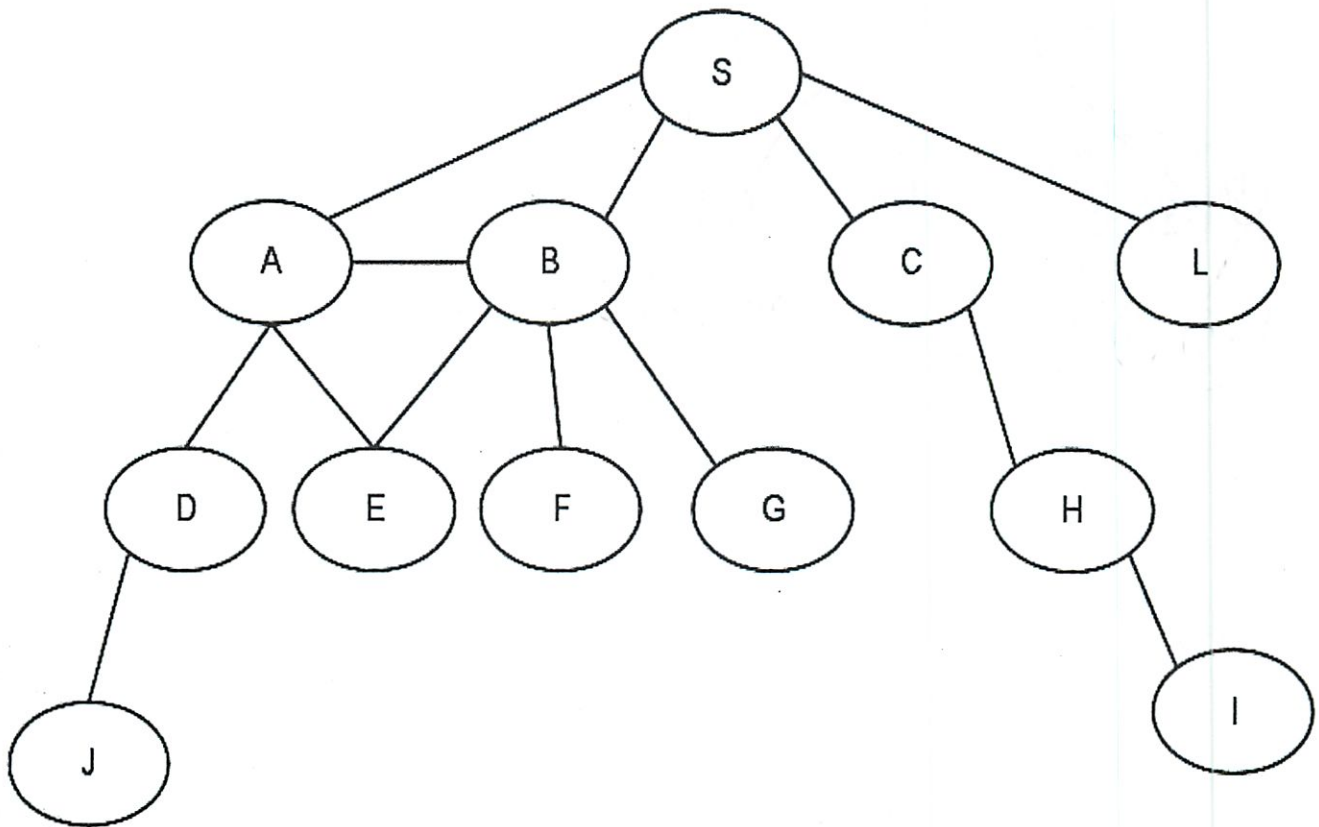
Circle one:  YES       NO

Yes but later

↳ you make it sound like it shouldn't have earlier

## Problem 2: Search (45 points)

### Part A: Basic search (30 points)





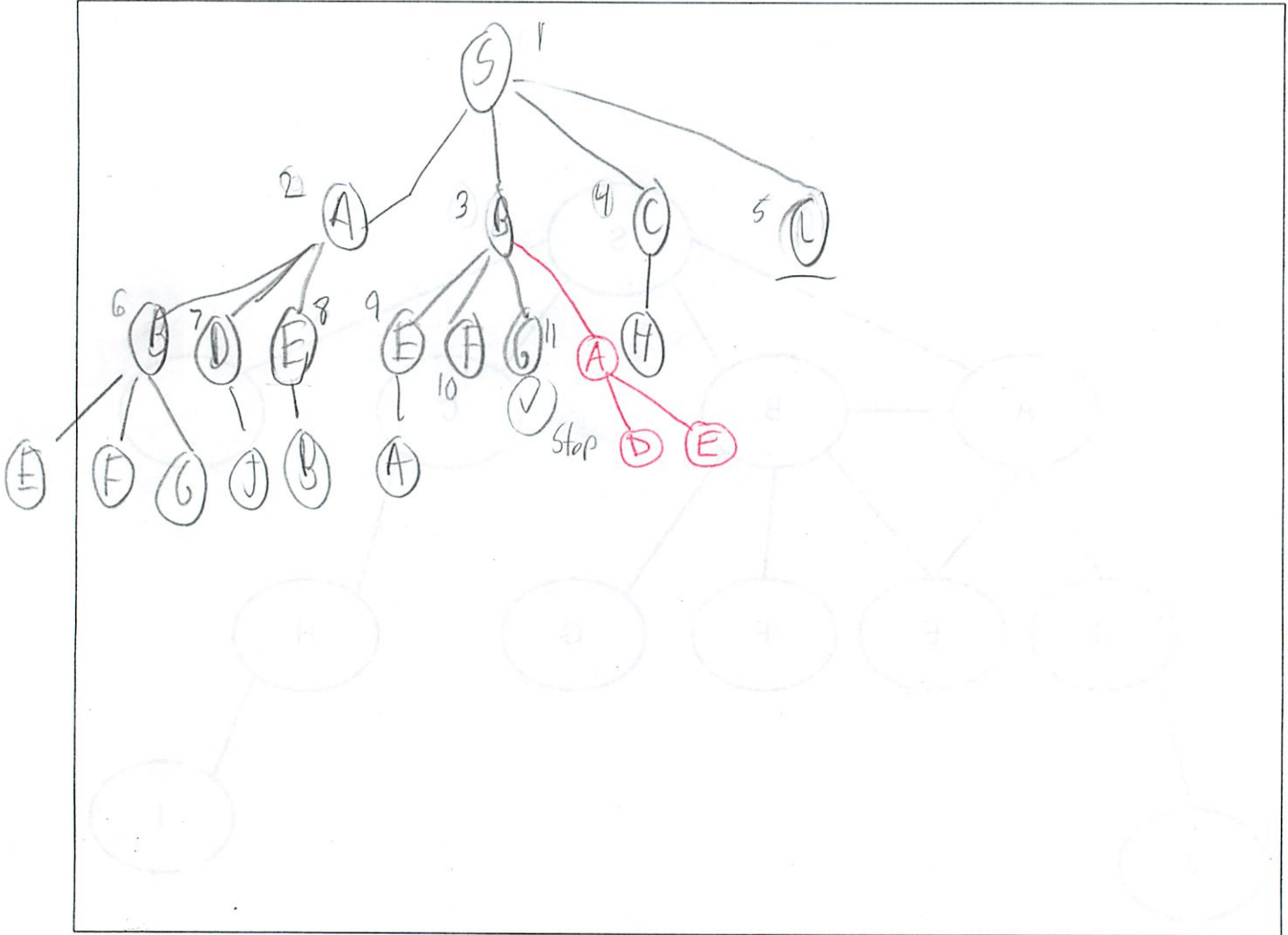
13/15

Prof: lexical order

### A1: Breadth First Search (15 points)

Starting at node S, find the BFS path to G, with **NO EXTENDED LIST**. Assume newly extended paths are placed on the end of a queue, but no path is checked to see if it is complete before it reaches the front of the queue. Draw the BFS tree and give the final path in the spaces below. In your tree drawing, number each node with the order it was extended.

Tree:



Path:

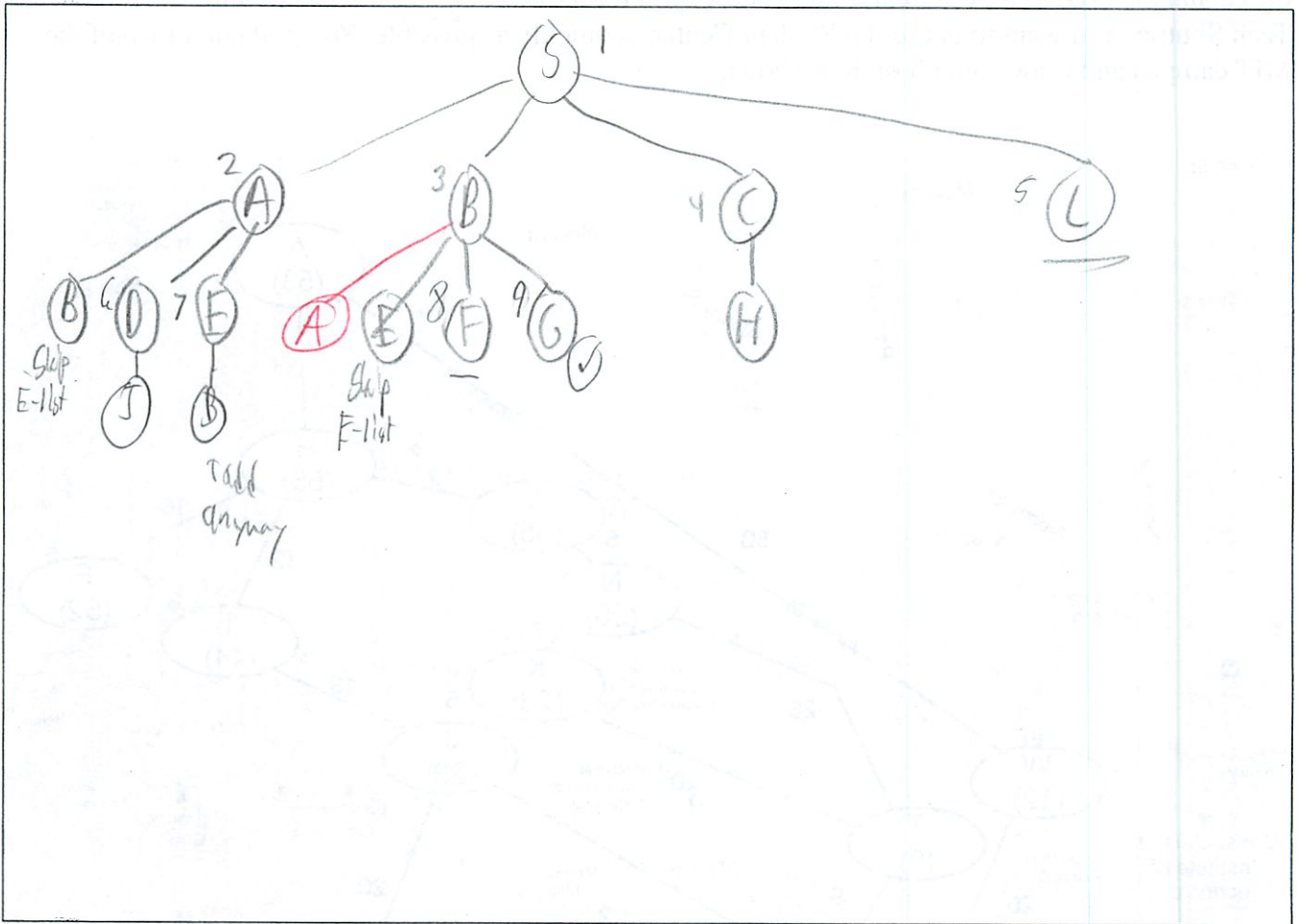
S B G

13/15

### A2: Breadth First Search + Extended List (15 points)

Starting at node S, find the BFS path to G USING AN EXTENDED LIST. Draw the BFS tree and give the final path in the spaces below. In your tree drawing, number each node with the order they were extended. Keep the extended list in the box provided below.

Tree:



Path:

S B G

Extended List:

S A B C L D E F G

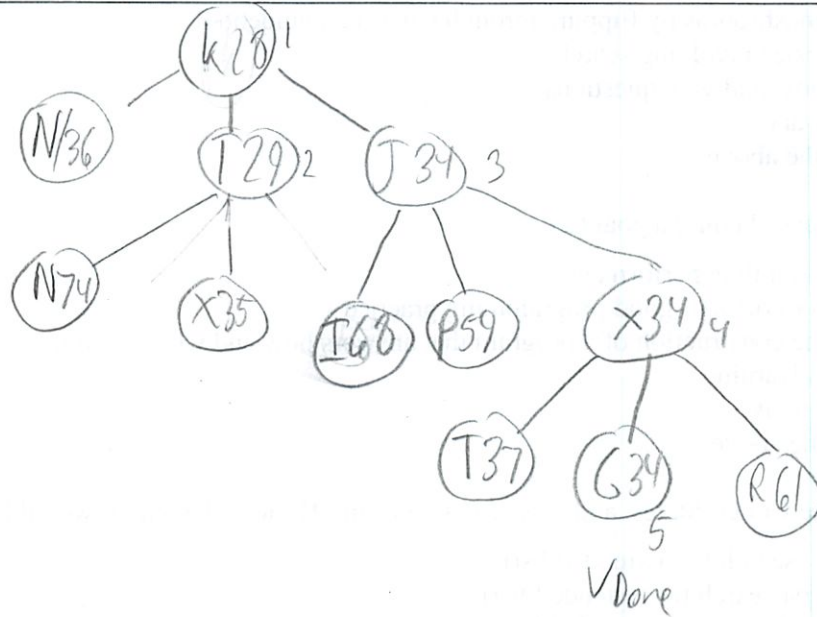


15/15

least cost any level

Starting at the **Building 26 (node K)**, find the **A\*** path to the **Student Center (G)** using the A\* algorithm you learned in class (i.e. use heuristic distance and an extended list). Draw the A\* extension tree and give the final path in the spaces below. In your tree drawing, show next to each node the number you use to determine the next path to extend (this way we can give partial credit for arithmetic errors). Keep the extended list in the box provided below (Alphabetical order breaks ties). Stop when a path that reaches the goal is the next path to be extended.

Tree:



Path:

K J X G

Extended List:

K T J X G

Handwritten notes on the left margin: 15/15, 28/15, 36, 29, 74, 35, 68, 59, 34, 37, 34, 61, 5, Done.

# Problem 3, Ideas (10 points)

10  
10

Circle the **best** answer for each of the following questions. There is no penalty for wrong answers, so it pays to guess in the absence of knowledge.

Generate and test is what we do when we

- 1. Identify mushrooms by flipping through pages in a guidebook *no generating*
  - 2. Solve puzzles involving search
  - 3. Answer how and why questions *no why*
  - 4. All of the above
  - 5. None of the above
- but counts I think*

A program that leaves behind a goal tree

- 1. Improves run time performance
  - 2. Conforms to rules of good programming practice
  - 3. Enables the construction of a program that answers how and why questions
  - 4. Facilitates learning
  - 5. All of the above
  - 6. None of the above
- Can show what assertions lead to what*

Given a map of the United States, and a desire to go from MIT to Cal Tech, it would be best to use

- 1. Depth first search (no extended list)
  - 2. Breadth first search (no extended list)
  - 3. Branch and bound (no extended list)
  - 4. Hill climbing (no extended list)
- admissible + consistent on a map*  
*heuristic*

When searching for a path, not necessarily the shortest, more knowledge

- 1. Can produce results faster
  - 2. Can produce shorter paths
  - 3. Can draw you into a dead end
  - 4. All of the above
  - 5. None of the above
- I don't really get this*  
*I can do all of these - depending on circumstance*

Beam search, with no extended list

- 1. Produces exponentially many paths with distance between start and goal
  - 2. Produces the shortest path
  - 3. Uses accumulated distance with no heuristic estimate of distance remaining
  - 4. All of the above
  - 5. None of the above
- limited*  
*there is a heuristic*

Intentionally blank

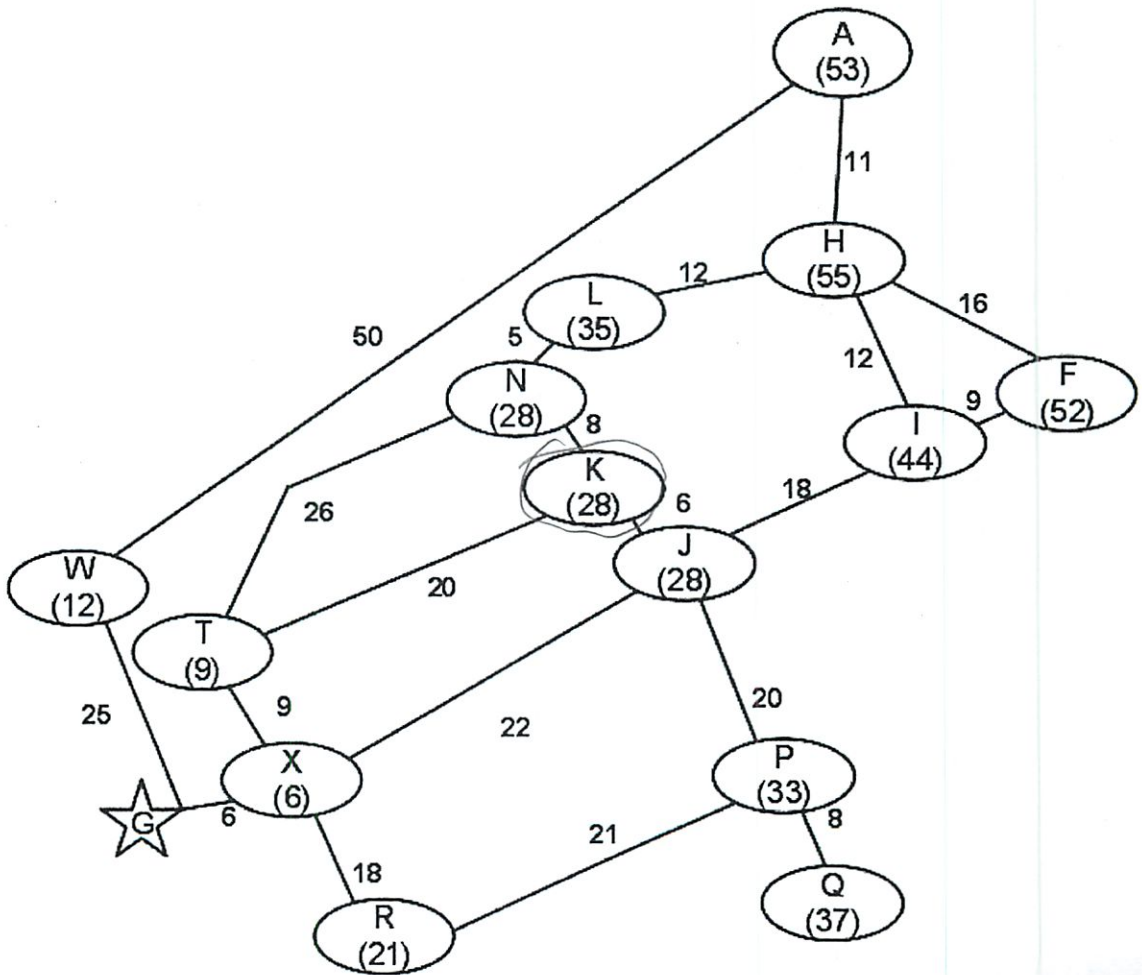
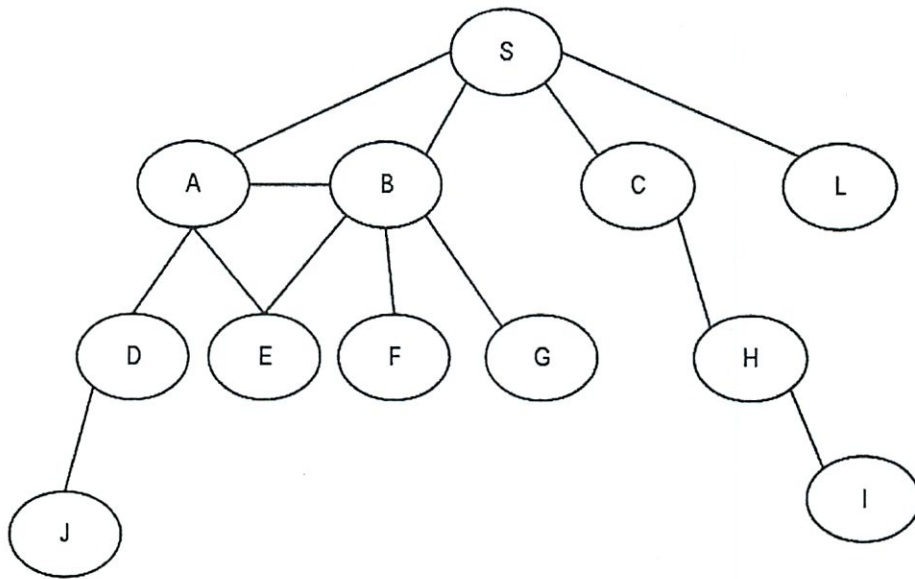
**Tear off sheets. You need not hand these in.**

**Rules:**

P0	IF(OR('( ?x) lives in Burton Conner', '( ?x) lives in East Campus', '( ?x) lives in Random'), THEN('( ?x) does not buy a dining plan'))
P1	IF('( ?x) does not buy a dining plan', THEN('( ?x) wants free food'))
P2	IF('( ?x) has class in Stata', THEN('( ?x) saw a poster'))
P3	IF(OR('( ?x) saw a poster', '( ?x) is in 6.034', AND('( ?x) is friends with (?y)', '(?y) is in 6.034')), THEN('( ?x) is interested'))
P4	IF(AND('( ?x) is interested', '( ?x) wants free food'), THEN('( ?x) will attend'))

**Assertions:**

'Anna is in 6.034'  
'Anna lives in Maseeh'  
'Ben lives in East Campus'  
'Ben has class in Stata'  
'Chris is friends with Anna'  
'Chris wants free food'





## 6.034 Quiz 1 28 September 2011

Name	
email	

Circle your TA and recitation time **(for 1 extra credit point)**, so that we can more easily enter your score in our records and return your quiz to you promptly.

TAs	
Avril Kenney	Adam Mustafa
Darryl Jones	Erek Speed
Gary Planthaber	Caryn Krakauer
Peter Brin	Tanya Kortz

Thu		Fri	
Time	Instructor	Time	Instructor
1-2	Bob Berwick	1-2	Randal Davis
2-3	Bob Berwick	2-3	Randall Davis
3-4	Bob Berwick	3-4	Randall Davis

This semester I am taking

subjects with substantial final projects or papers.

Problem number	Maximum	Score	Grader
1	45		
2	45		
3	10		
Total	100		

**There are 12 pages in this quiz, including this one, but not including blank pages and tear-off sheets. Tear-off sheets are provided at the end with duplicate drawings and data. As always, open book, open notes, open just about everything, including a calculator, but no computers.**

## Problem 1: Rule Systems (45 points)

After the first few weeks of 6.034, you realize that artificial intelligence is the most exciting topic you have ever studied, and you decide to start an AI club. To get more people to join the club, you organize an info session, which will (of course) have free food. You need to figure out how much food to order, so you write the following rule system to determine who will attend the info session.

### Rules:

P0	IF(OR('(?x) lives in Burton Conner', '(?x) lives in East Campus', '(?x) lives in Random'), THEN('(?x) does not buy a dining plan'))
P1	IF('(?x) does not buy a dining plan', THEN('(?x) wants free food'))
P2	IF('(?x) has class in Stata', THEN('(?x) saw a poster'))
P3	IF(OR('(?x) saw a poster', '(?x) is in 6.034', AND('(?x) is friends with (?y)', '(?y) is in 6.034')), THEN('(?x) is interested'))
P4	IF(AND('(?x) is interested', '(?x) wants free food'), THEN('(?x) will attend'))

### Assertions:

'Anna is in 6.034'  
'Anna lives in Maseeh'  
'Ben lives in East Campus'  
'Ben has class in Stata'  
'Chris is friends with Anna'  
'Chris wants free food'

## Part A: Backward Chaining (25 points)

Make the following assumptions about backward chaining:

- When working on a hypothesis, the backward chainer tries to find a matching assertion in the list of assertions. If no matching assertion is found, the backward chainer tries to find a rule with a matching consequent. If no matching consequent is found, then the backward chainer *assumes the hypothesis is false*.
- The backward chainer never alters the list of assertions, so it can derive the same result multiple times.
- Rules are tried in the order they appear.
- Antecedents are tried in the order they appear.
- Lazy evaluation/short circuiting is in effect (e.g., if the first part of an AND clause is false, the rest does not need to be evaluated to determine that the whole clause is false).

**A1 (17 points)**

You want to know whether Anna will attend the info session. Use backward chaining to check the hypothesis 'Anna will attend'. Use the space provided on the next page to draw the goal tree that is created by backward chaining (we will use the goal tree to help us assign partial credit). Then, list, in the order that they are checked, all the hypotheses the backward chainer looks for in the course of checking the hypothesis 'Anna will attend'. The table may have more lines than you need.

Anna will attend
Anna is interested
Anna saw a poster
Anna has class in Stata
Anna is in 6.034
Anna wants free food
Anna does not buy a dining plan
Anna lives in Burton Conner
Anna lives in East Campus
Anna lives in Random

**A2 (8 points)**

If you were able to show 'Anna will attend', which assertions can you remove without changing the answer:

N/A

Also, identify which rules can you remove without changing the answer:

N/A

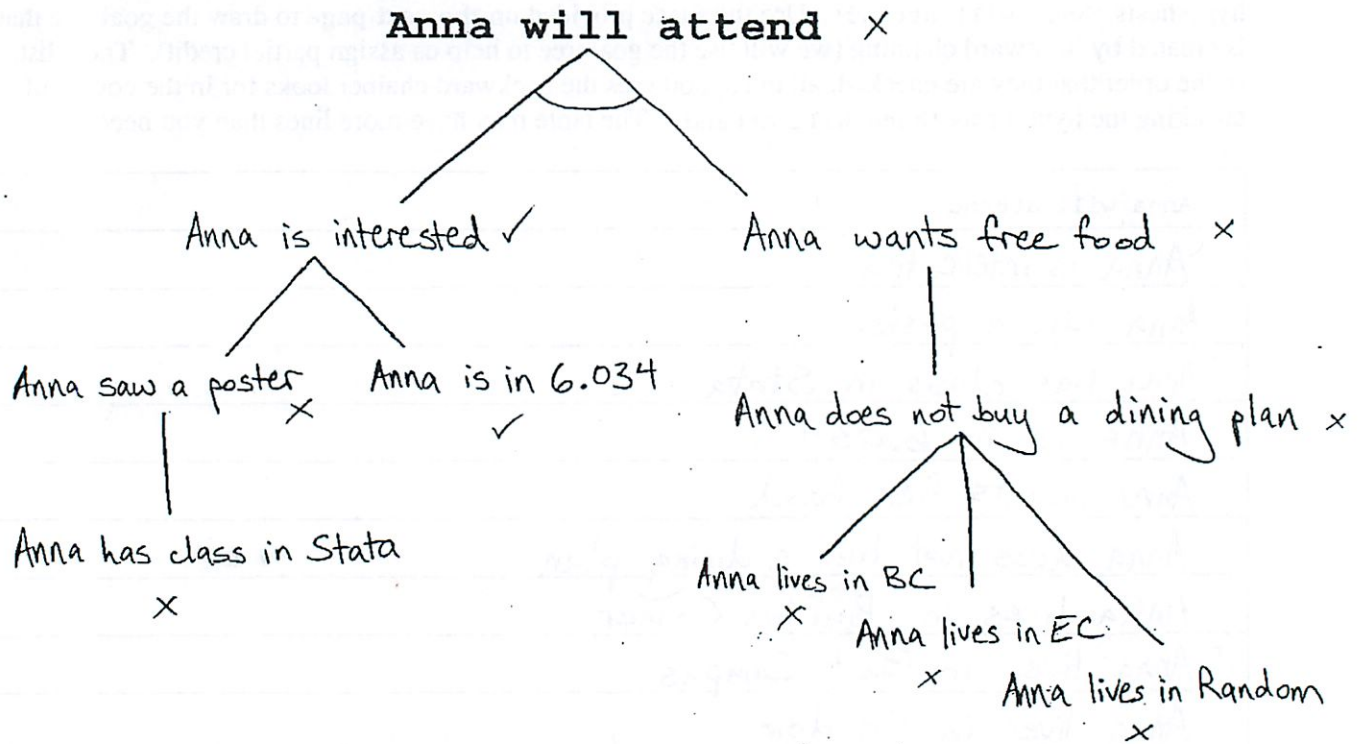
If you were not able to show 'Anna will attend', if it is possible to add a new *assertion* that would change your answer (other than adding the hypothesis itself), state such an assertion:

ONE OF:  
Anna wants free food      Anna lives in Burton Conner      Anna lives in Random  
Anna does not buy a dining plan      Anna lives in East Campus

Also, if it is possible to add a new *rule* that would change your answer, state such a rule.

Many possible answers. See next page.

Use this space to draw your goal tree.



Last part of A2: IF(P, THEN(Q)), where

P is any of the following (or any combination of them using AND/OR):

- (?x) is interested
- (?x) is in 6.034
- (?x) lives in Maseeh
- Chris is friends with (?x)
- AND('(?y) is friends with (?x)', '(?y) wants free food')
- AND('(?y) is friends with (?x)', '(?y) is interested')
- AND('(?y) is friends with (?x)', '(?y) will attend')

Q is any of the following:

- (?x) will attend
- (?x) wants free food
- (?x) does not buy a dining plan
- (?x) lives in Burton Corner
- (?x) lives in East Campus
- (?x) lives in Random

## Part B: Forward Chaining (20 points)

Make the following assumptions about forward chaining:

- Assume rule-ordering conflict resolution.
- New assertions are added to the bottom of the list of assertions.
- If a particular rule matches assertions in the list of assertions in more than one way, the matches are considered in the order corresponding to the top-to-bottom order of the matched assertions. Thus, if a particular rule has an antecedent that matches both Assertion 1 and Assertion 2, the match with Assertion 1 is considered first.

### B1 (15 points)

Now you want to determine *all* of the people who will attend the info session. Run forward chaining on the list of assertions. For the first two iterations, fill out the first two rows in the table below, noting the rules whose antecedents match the data, the rule that fires, and the new assertions that are added by the rule. For the remainder, supply only the fired rules and new assertions. The table has more lines than you need.

Iter	Matched	Fired	New Assertions Added
1	P0, P2, P3	P0	Ben does not buy a dining plan
2	P0, P1, P2, P3	P1	Ben wants free food
3		P2	Ben saw a poster
4		P3	Anna is interested
5		P3	Chris is interested
6		P3	Ben is interested
7		P4	Chris will attend
8		P4	Ben will attend
9			
10			

How many people will attend the info session? 2

### B2 (5 points)

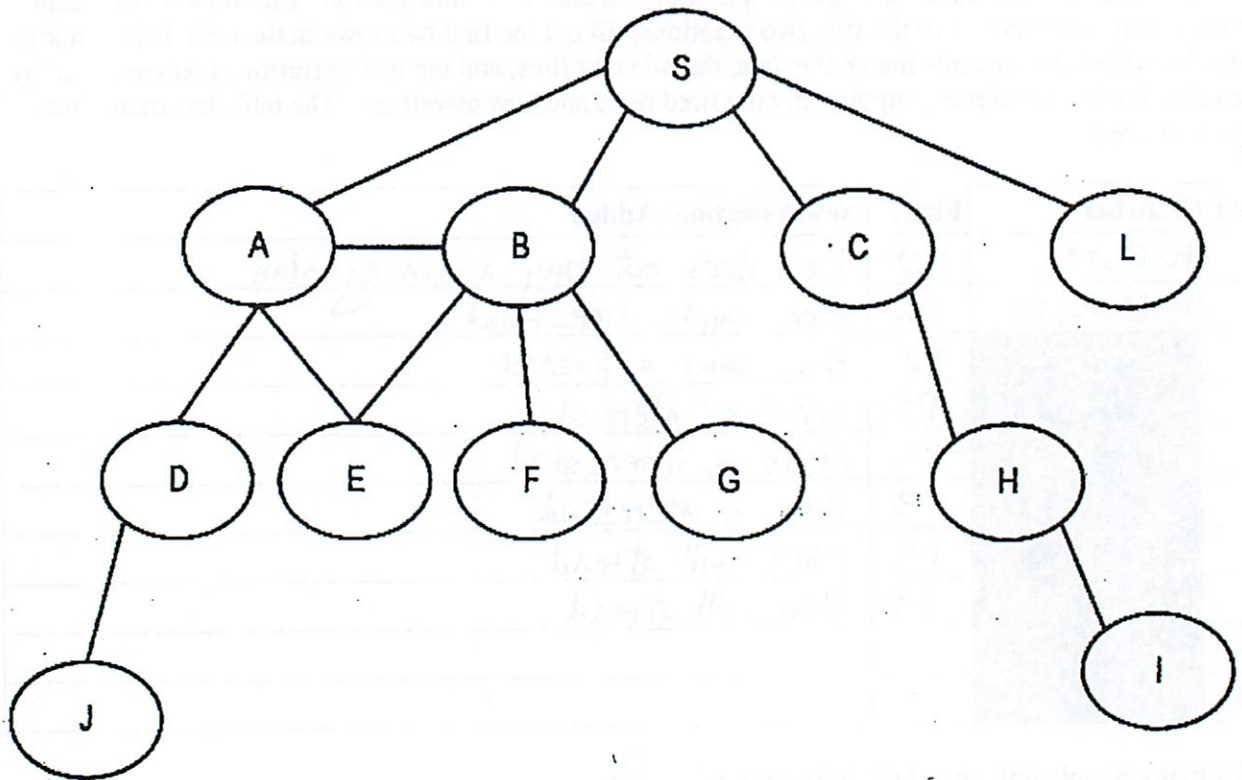
Suppose that you moved P1 to the end of the list of rules.

Then would forward-chaining conclude that *Ben* will attend the info session?

Circle one:  YES  NO

## Problem 2: Search (45 points)

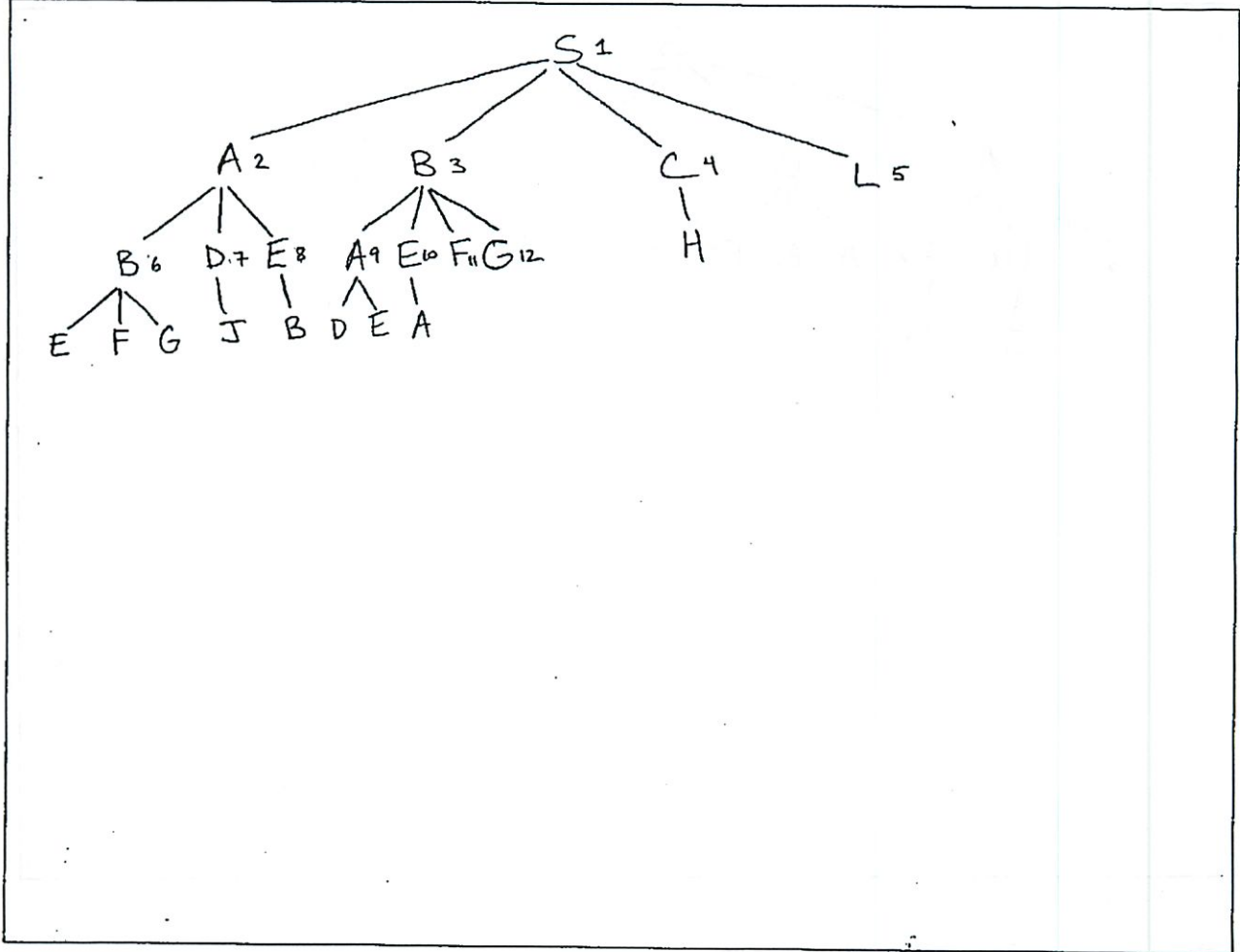
### Part A: Basic search (30 points)



**A1: Breadth First Search (15 points)**

Starting at node S, find the BFS path to G, with **NO EXTENDED LIST**. Assume newly extended paths are placed on the end of a queue, but no path is checked to see if it is complete before it reaches the front of the queue. Draw the BFS tree and give the final path in the spaces below. In your tree drawing, number each node with the order it was extended.

Tree:



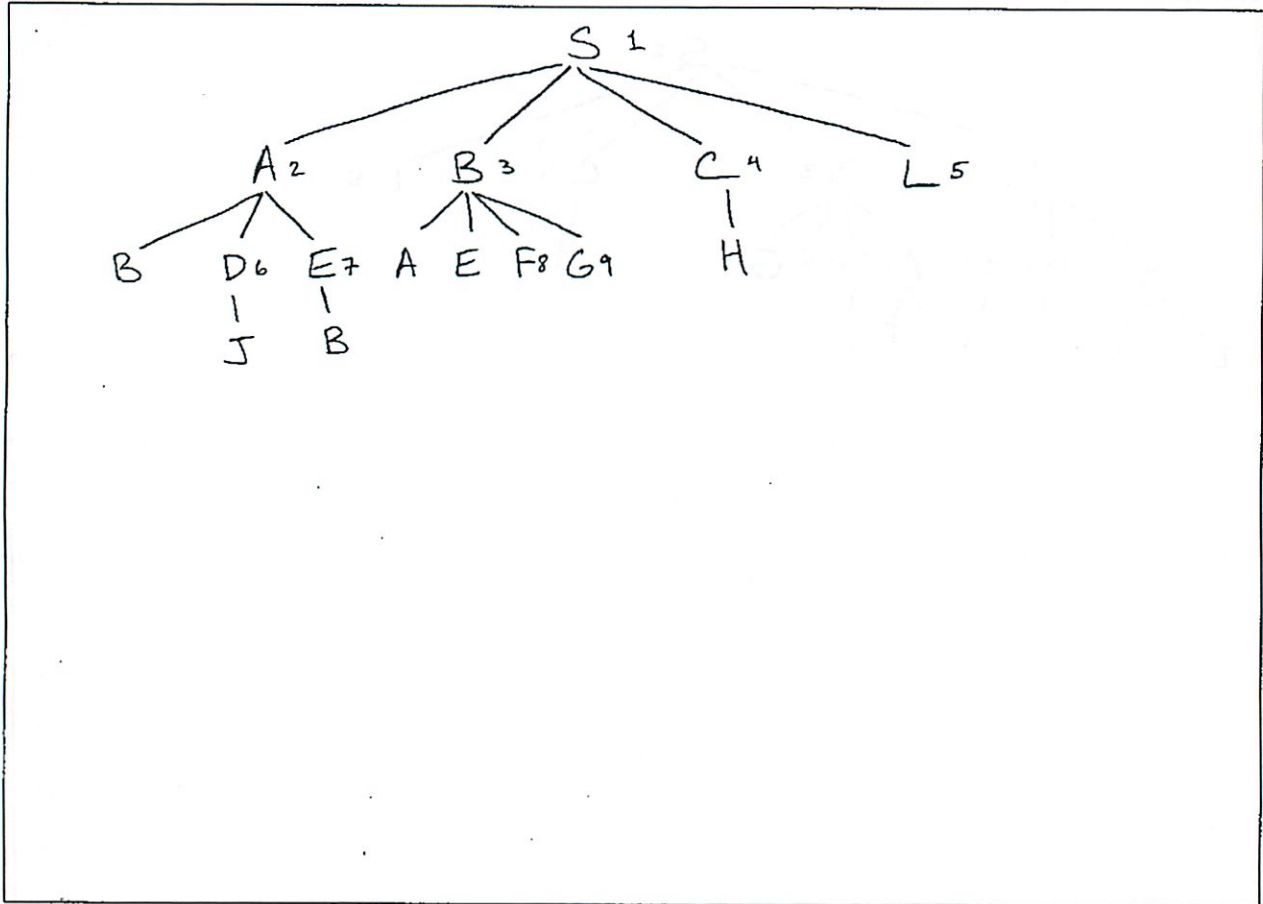
Path:

S B G

**A2: Breadth First Search + Extended List (15 points)**

Starting at node S, find the BFS path to G USING AN EXTENDED LIST. Draw the BFS tree and give the final path in the spaces below. In your tree drawing, number each node with the order they were extended. Keep the extended list in the box provided below.

Tree:



Path:

S B G

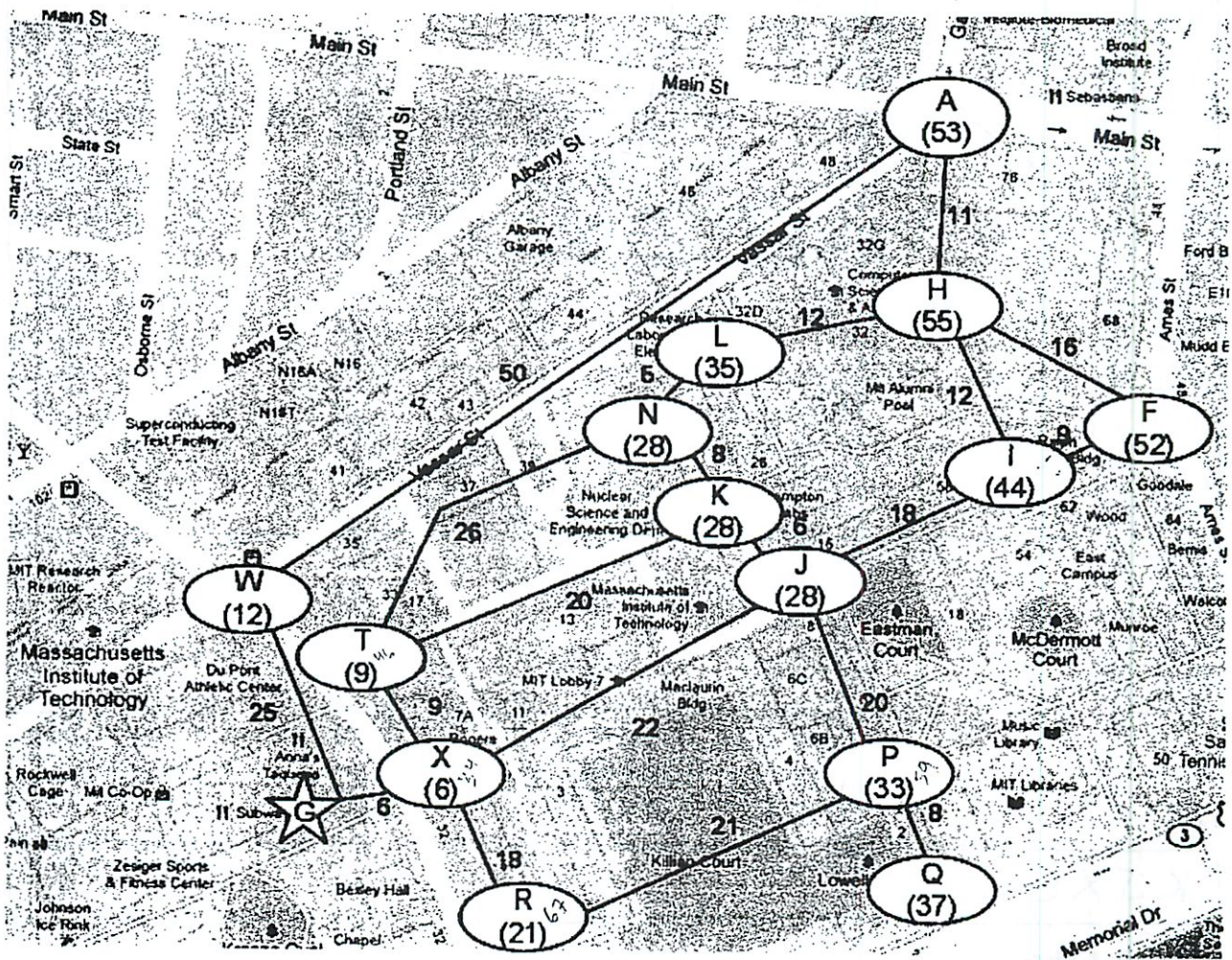
Extended List:

S A B C L D E F G



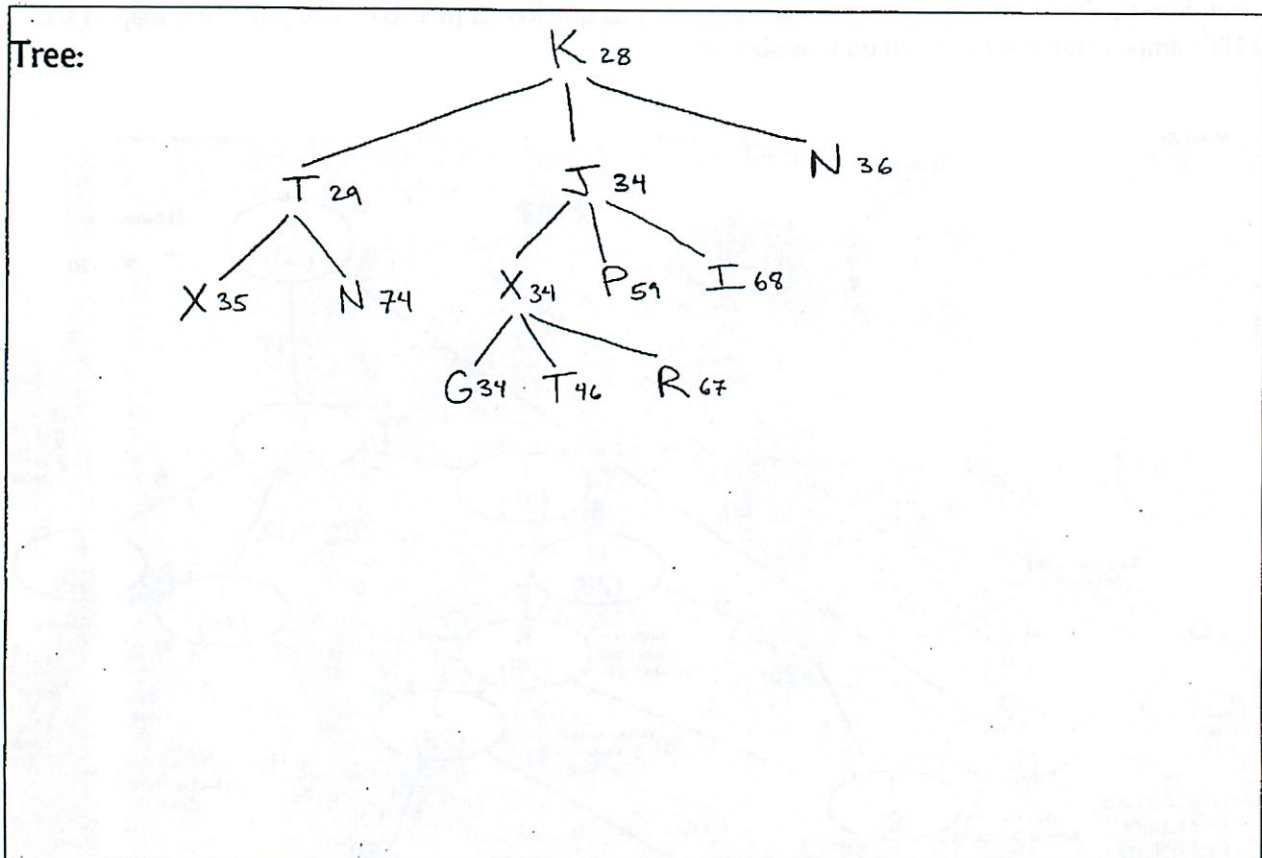
## Part B: Optimal-path search (15 points)

You are an environmentally conscious but greedy student. You have just learned that there is a recycling bin at the Student Center that converts glass into gold. You just left Professor Winston's office and the urge to recycle really hits you. As you reach Building 26, it starts raining and there is no Tech Shuttle. You want to get to the Student Center as quickly as possible. You pull out a map of the MIT campus and draw a graph on it as shown:



A copy of this graph without the map clutter is on the tear off sheet at the end of the quiz. The numbers on the edges correspond to the length of that edge and the numbers in the parentheses correspond to the heuristic distance to G.

Starting at the **Building 26 (node K)**, find the A\* path to the **Student Center (G)** using the A\* algorithm you learned in class (i.e. use heuristic distance and an extended list). Draw the A\* extension tree and give the final path in the spaces below. In your tree drawing, show next to each node the number you use to determine the next path to extend (this way we can give partial credit for arithmetic errors). Keep the extended list in the box provided below (Alphabetical order breaks ties). Stop when a path that reaches the goal is the next path to be extended.



Path:

K J X G

Extended List:

K T J X G

## Problem 3, Ideas (10 points)

Circle the **best** answer for each of the following questions. There is no penalty for wrong answers, so it pays to guess in the absence of knowledge.

Generate and test is what we do when we

1. Identify mushrooms by flipping through pages in a guidebook
2. Solve puzzles involving search
3. Answer how and why questions
4. All of the above
5. None of the above

A program that leaves behind a goal tree

1. Improves run time performance
2. Conforms to rules of good programming practice
3. Enables the construction of a program that answers how and why questions
4. Facilitates learning
5. All of the above
6. None of the above

Given a map of the United States, and a desire to go from MIT to Cal Tech, it would be best to use

1. Depth first search (no extended list)
2. Breadth first search (no extended list)
3. Branch and bound (no extended list)
4. Hill climbing (no extended list)

When searching for a path, not necessarily the shortest, more knowledge

1. Can produce results faster
2. Can produce shorter paths
3. Can draw you into a dead end
4. All of the above
5. None of the above

Beam search, with no extended list

1. Produces exponentially many paths with distance between start and goal
2. Produces the shortest path
3. Uses accumulated distance with no heuristic estimate of distance remaining
4. All of the above
5. None of the above