The Busy Beaver Problem

Computability and Logic

The Busy Beaver Problem

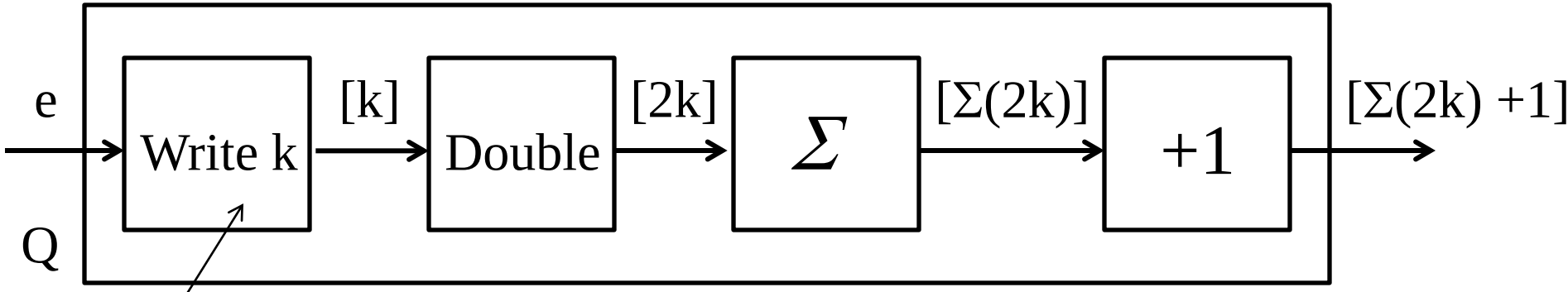
- *n:* number of states
- *M(n)*: set of TM's with *n* states and binary alphabet (only *0*'s and *1*'s)
- *[k]*: Configuration of having a block of *k* consecutive *1*'s on an otherwise blank (all *0*) tape, and with head at leftmost *1*.
- *e*: empty tape (all 0's)
- $\Sigma(M)$: $\Sigma(M) = k$ if machine *M*, when started on *e*, halts with *[k]*. Otherwise, $\Sigma(M) = 0$.
- *Busy Beaver Problem*: Find $\Sigma(n) = \max \{\Sigma(M) \mid M \in M(n)\}$

Variations

- We can define a variety of Busy Beaver problems:
  - Do we use the quadruple or quintuple formalization?
  - Do we use a binary alphabet or more than 2 symbols?
  - How does the machine come to a halt?
    - *Explicit halting state*: machine halts by a transition to an explicit halting (in which case halting state does not get counted towards n)
    - *Implicit halting state*: machine halts by the lack of a transition for current state and symbol
  - Are there any restrictions on the output configuration?
    - *Standard configuration*: head positioned at leftmost 1 (or other non-blank symbol) of consecutive string of 1's on otherwise empty tape
    - *Anything goes* (head does not need to be at leftmost 1, and 1's may be scattered all over tape)
- For now, let's stick to a binary alphabet and require a standard configuration. But all proofs on the next slides can be modified to accommodate all other types.

$\Sigma(n)$ is Turing-Uncomputable

- Proof by Contradiction: Suppose there is some Turing-Machine Σ that computes $\Sigma(n)$. Then consider the following Turing-machine Q, where k is the number of states of the last 3 components:



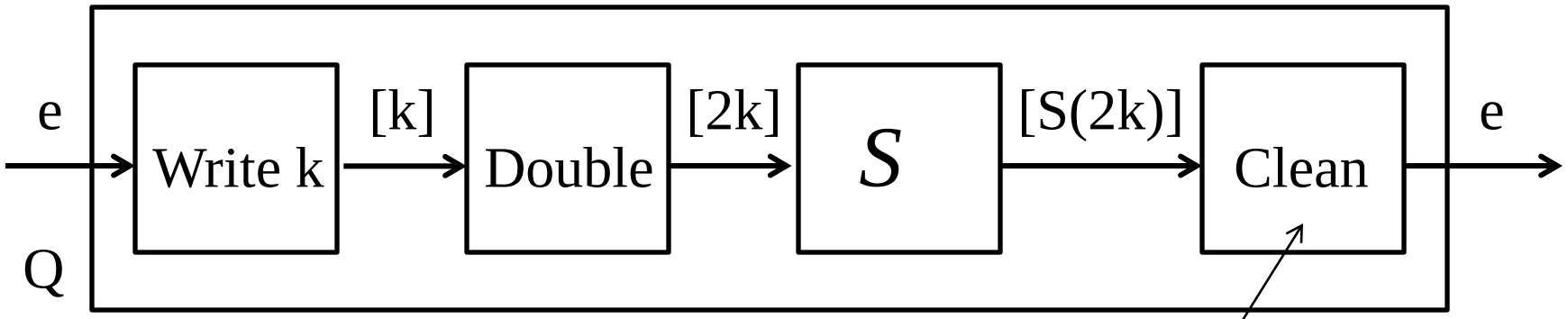
Can be implemented using k states. So, Q has 2k states …
and outputs Σ(2k) +1 when started on an empty tape! Whoopsie!

The Shifting Problem

- *S(M)*: *S(M) = m* if machine *M*, when started on *e*, takes *m* steps before it halts with *[k]* for some *k*. Otherwise, *S(M) = 0*.
- *Shifting Problem*: Find $S(n) = \max \{S(M) \mid M \in M(n)\}$

S(n) is Turing-Uncomputable

- Proof by Contradiction: Suppose there is some Turing-Machine *S* that computes *S(n)*. Then consider the following Turing-machine Q, where k is the number of states of the last 3 components:



Cleaning the tape will take at least S(2k) steps. So, Q has 2k states …but takes more than S(2k) steps before halting! Whoopsie!

S(n) is Uncomputable: Alternative Proof

- The uncomputability of *S(n)* can also be derived from the uncomputability of *Σ(n)*:
- Suppose *S(n)* is computable. Then *Σ(n)* can be determined simply by running all of the finitely many Turing-machines M with n states, starting on e. If a Turing-machine is still running after *S(n)* steps, you know it is a non-halter. For all the others *Σ(M)* can be determined, and now simply take the max.

Computability, Uncomputability, and the Church-Turing Thesis

- Notice that the proof on the previous slide establishes that *S(n)* is uncomputable, rather than Turing-uncomputable.
- In particular, the proof assumed that *Σ(n)* is uncomputable, rather than just Turing-uncomputable.
- So, the proof appeals to the Church-Turing Thesis, which states that anything that is computable is Turing-computable (so: since we know that *Σ(n)* is Turing-uncomputable, *Σ(n)* is not computable).
- Many of the other proofs you'll find in the rest of this presentation make similar use of the Church-Turing Thesis, i.e. from now on we'll simply equate computability with Turing-computability.

Problem Reductions

- The proof from 2 slides ago is a good example of reducing one problem into another.
- Problem A reduces to problem B if being to solve problem B allows you to solve problem A.
- So, if problem A reduces to problem B, then if we know that problem A cannot be solved, then we know that B cannot be solved either.
- In the proof, we reduced the computability of $\Sigma(n)$ to the computability of $S(n)$.

The Busy Beaver Problem: General Version

- In general, the busy beaver problem is to find the ‘most productive’ Turing machine with n states and m symbols.
- The ‘productivity’ of a Turing machine can be defined in many ways:
  - The number of steps taken (‘time’)
  - The number of symbols written (‘f(n)’)
  - The number of cells moved away from the starting cell (‘space’)
  - Etc.
- Any of these kinds of functions can be found to be uncomputable.
- For any particular problem you can show this either by a direct proof, or by reducing it into another problem that you already established to be uncomputable.

Upper Bounds

- Another interesting thing to note about the proof (from 4 slides ago) is that it uses the technique of using upper bounds: if we know an upper bound to the number of steps a machine can take before halting (such as S(n)!), then we can determine any machine with n states to be a halter or non-halter simply by running it: any machine that is still running after S(n) steps is a non-halter.
- So, once you have discarded all non-halters, you can simply run all halters to completion to figure out any kind of Busy Beaver function you want.

Connection Halting Problem and Busy Beaver Problem

- In fact, there seems to be an intimate connection between the Busy Beaver Problem and the Halting Problem.
- Indeed, one might be inclined to say that the Halting Problem is immediately be solvable if S(n) is computable … but that would be a mistake!
  - Remember that S(n) gives an upper-bound to the number of steps taken by all machines with n states … when started on an empty tape!
  - So, it does not give an upper-bound taken by all machines with n states given any kind of input tape, and the Halting function considers input tapes of any kind.
- Still, it does turn out that the two problems are intimately related! But we'll have to do a bit of work …

If Halting Problem is solvable, then Busy Beaver Problem is.

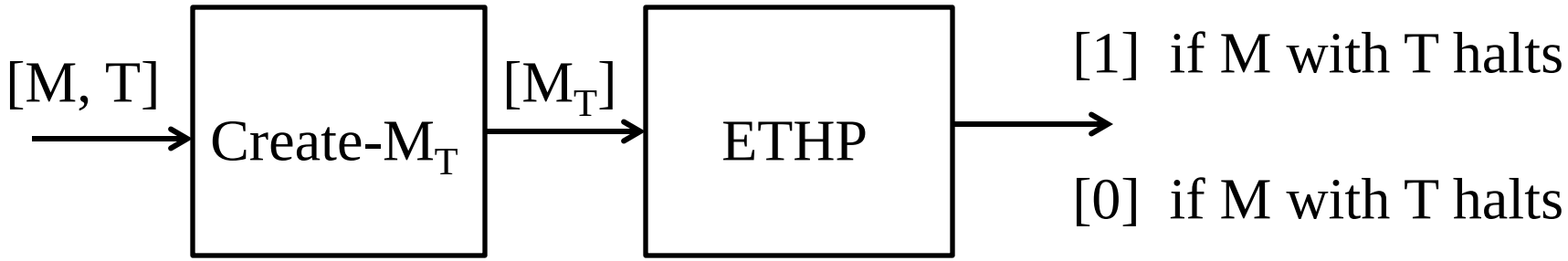
- One connection is obvious: If the Halting Problem is solvable, then (any) Busy Beaver Problem is solvable.
- That is, $\Sigma(n)$ (or $S(n)$, or what have you) can be computed if we can solve the halting problem: Simply go through all the finitely many machines with n states, use the halting solution to discard any non-halters, and simply run all others to completion to get the desired answer.
- OK, but what about the other way? That one is more difficult …

Empty Tape Halting Problem

- Let us define the Empty Tape Halting Problem (ETHP) as the problem of determining for any machine M whether or not it will halt when started on an empty tape.
- Now, it is clear that if the general Halting Problem would be solvable, then ETHP would be solvable as well.
- But, does the unsolvability of the Halting problem imply the unsolvability of ETHP?

Uncomputability of The Empty Tape Halting Problem

- Yes! First, we can devise a routine that, given any M and T, constructs a machine M_T that, when given an empty tape, first puts T on the tape, and then runs M on that tape. Let's call this the Create-M_T routine.
  - Note: This routine is far from easy to write as a Turing-machine routine, but it is intuitively obvious that we should be able to do this, i.e. that such a routine does exist.
- Now let us assume the Empty Tape Halting Problem is solvable. Then the Halting Problem is solvable as well:



Empty Tape Halting Problem and S(n)

- Claim: The Empty Tape Halting Problem is solvable iff the S(n) is computable.
- Proof:
  - If the ETHP is solvable, then we can figure out S(n) by discarding all non-halters, running all others to completion, and determine max.
  - If S(n) is computable, then ETHP is solvable: simply start running any machine on an empty tape, and if it is still running after S(n) steps, then it is a non-halter, otherwise it is a halter.

$S(n)$ and $\Sigma(n)$

- Earlier we saw that $\Sigma(n)$ is computable if $S(n)$ is computable, since $S(n)$ provides an upper-bound to the possible number of steps taken. Does the other way around also hold?
- Yes. For any machine M_1 that takes n steps before halting with [k] when started on an empty tape, you can construct a machine M_2 that simulates M_1, but also prints out a 1 for every step that M_1 makes, and where the number of states of M_2 is a linear (and thus computable!) function of the number of states of M_1
  - E.g. for the quintuple formalization, you can show that if M_1 has n states, then such a M_2 can be constructed with 20n states (see Julstrom)

S(n) and *Σ(n)* (Continued)

- So, supposing $S(M_1) = S(n)$ (i.e. supposing M_1 is a machine with n states that makes the most steps for any machine with n states) we thus have that $S(n) \leq \Sigma(f(n))$ for some computable f(n).
- So, if $\Sigma(n)$ is computable, then S(n) becomes computable too: simply start any machine with n states on an empty tape, and any machine that still runs after having taken $\Sigma(f(n))$ steps must be a non-halter. For all halters, simply determine the maximum number of steps taken. In short, $\Sigma(f(n))$ provides an upper-bound for S(n)!

Summing it all Up: Busy Beaver and Halting Problem

- We have shown that:
  - The Halting Problem is solvable iff the Empty Tape Haling Problem is solvable.
  - The Empty Tape Halting problem is solvable iff S(n) is computable.
  - S(n) is computable iff $\Sigma(n)$ is computable.
- So, these are all equivalent statements!
- In particular, all of these problems (functions) are unsolvable (uncomputable)!

More on Upper Bounds

- Suppose that there is some computable function f(n) that provides an upper-bound to the maximum number of steps that a machine with n states can take, starting on an empty tape. In short, suppose that for all n: $S(n) \leq f(n)$.
- Well, then S(n) would be computable too: all machines that still run after f(n) are non-halters, so S(n) can be determined by examining all halters.
- Since S(n) is not computable, we know that no such function exists: there is no computable upper-bound for S(n)!

So what?

- So this means that S(n) is a function that ‘grows faster’ than any computable function.
- And, you can easily come up with some computable functions that grow crazy fast.
- Well, S(n) will grow even faster than that!
- Moreover, since we found that $S(n) \leq \Sigma(f(n))$ for some computable f(n), we know that $\Sigma(n)$ also grows faster than any computable function! (otherwise, we’d once again have a computable upper-bound for S(n)).

Exhibit A: Established values for Σ(n) (for quintuple TM's)

n	Σ(n)
1	1 (trivial)
2	4 (easy)
3	6 (Lin and Rado)
4	13 (Brady)
5	>= 4098 (Marxen et al.)
6	$> 3.514 * 10^{18276}$ (Marxen et al.)
7	Huge!
8	Insane!!

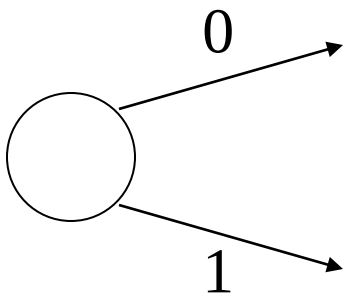
Problems in Determining $\Sigma(n)$

- First, search space is fairly big (see next slide):
  - Explicit halting state: $|M(n)| = (4n + 4)^{2n}$
  - Implicit halting state: $|M(n)| = (4n + 1)^{2n}$

- But most importantly, the behavior of even small machines is strange and hard to grasp, and we have a hard time figuring out whether they halt or not!

Size of Search Space

For every state:



For every transition:

Quintuple:
0 or 1 and L or R
$\Rightarrow$ 2*2 = 4 possibilities

Quadruple:
0 or 1 or L or R
$\Rightarrow$ 4 possibilities

In all cases: 2n state-symbol pairs

Explicit: n + 1 possible new states with full transition

Implicit: n possible new states with full transition + empty transition to halting state

n	$(4n + 4)^{2n}$	$(4n + 1)^{2n}$
1	64	25
2	20,736	6,561
3	16,777,216	4,826,809
4	$\sim 2.56*10^{10}$	$>10^{9}$
5	$>10^{13}$	$>10^{13}$
6	$> 10^{17}$	$> 10^{16}$

Strange Behavior

- For n = 5 (quintuple), some machines only come to a halt after over 47 million steps (this is the suspected shifting champion … but we haven't been able to prove that … yet?)
- For n = 6, some machines halt after more than 10^{36534} steps! (yes, you read that correctly!)
- In 1983, Brady conjectured that it's impossible to classify all machines with n = 5
- In 2012, we have indeed still not been able to settle the n=5 case … but people are getting close.
- Still, it's unlikely we'll ever settle the n = 6 case.
- And it's a virtual certainty we'll never settle the n = 7 case!

Attacking the Busy Beaver Function

- Still, people are trying to determine Busy Beaver values.
- This is done by:
  - Reducing the search space
  - Writing custom-made computer programs to simulate the behavior of Turing-machines and detecting knowable forms of non-halting behavior.

Reducing the Search Space

- The search space can be reduced in a number of ways (see Appendix):
  - Perform some initial analysis to rule out certain corners of the search space
  - Only consider one machine of a set of machines that can be shown to have equal productivity
  - Use Tree normalization method to create only ‘relevant’ machines.

Halters, Non-halters, and Holdouts

- Since the halting problem cannot be solved in general, any practical algorithm A that does try to figure out halting behavior is such that for any machine M:
  - A eventually declares that M halts (M is a halter)
  - A eventually declares that M does not halt (M is a non-halter)
  - A eventually declares that it doesn't know whether M halts or not (M is a *holdout*)

Taking Care of the Holdouts

- In order to take care of the holdouts, do the following:
  - Simulate a holdout for some number of steps
  - Observe the behavior, and see if there is some pattern present which one can use to prove that the machine will halt or not
  - Try and generalize the pattern of behavior, and incorporate it into the original algorithm to obtain a new and improved algorithm
  - If there are still holdouts left with the new algorithm, repeat this routine

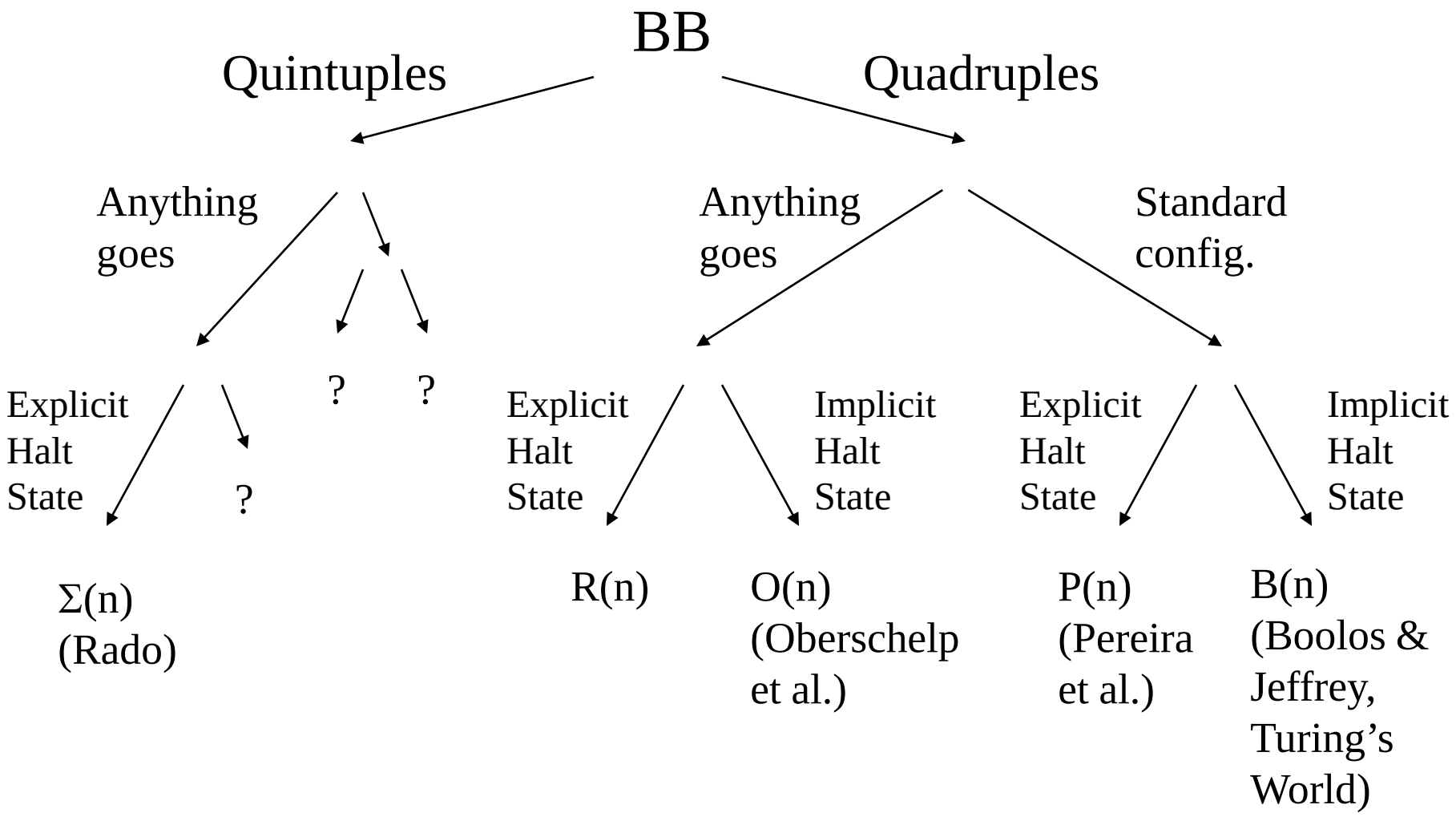
History of Σ(3)

- Rado was at first very pessimistic:
  - "… there is no evidence that any known approach will yield the answer, even if we avail ourselves of high-speed computers and elaborate programs." (Rado, 1963)
- Indeed, it turns out he was too pessimistic:
  - "The solution of this quite special problem was attempted by several competent mathematicians and programmers, by means of increasingly elaborate computer programs." (Lin and Rado, 1965)
- Lin and Rado reduced the 56871 holdouts from 1963 to 40 holdouts in 1965, and analyzed those last 40 by hand to determine Σ(3).

History of Σ(4)

- Still:
  - "As regards Σ(4), … the situation seems to be entirely hopeless at present." (Rado, 1963)
- However!
  - "More than 18 other programs were written for various housekeeping purposes, simulating and displaying machine behavior, exploring other reduction and filtering possibilities, etc. In all, at least 53 files were created and maintained for the project. Keeping track of what resembled a large scientific experiment became a major task in itself." (Brady, 1983)
- Brady obtained 0 holdouts for n = 3, and 218 holdouts for n = 4. These "were examined by means of voluminous printouts of their histories along with some program extracted features. It was determined to the author's satisfaction that none of these machines will ever stop."

Taxonomy of BB Problems



The Story of the Quadruples: R(n), O(n), P(n), and B(n)

n	R(n)	O(n)	P(n)	B(n)
1	1	1	1	1
2	2	2	2	2
3	4	3	4	3
4	8	8	7	5
5	16	15	16	11
6	>= 240	>=239	>= 21	>= 25
7	?	?	>= 102	>= 196
8	?	?	>= 384	>= 672

P(n) >= B(n)

O(n) >= B(n)

R(n) >= P(n)

R(n) = O(n) or
O(n) + 1

Values and records established by RPI research team in 2005!

References

- Rado, “On Non-Computable Functions”, The Bell System Technical Journal, 41(3), pp. 877-884, 1962
- Lin and Rado, “Computer Studies of Turing Machine Problems”, Journal of the ACM, 12(2), pp. 196-212, 1965
- Brady, “The Determination of the Value of Rado’s Noncomputable Function Σ(k) for Four-State Turing Machines”, Mathematics of Computation, 40(162), pp. 647-665, 1983
- Oberschelp et al., “Castor Quadruplurum”, Archive for Mathematical Logic, 27, pp. 35-44, 1988
- Marxen and Buntrock, “Attacking the Busy Beaver 5”, Bulletin of the EATCS, 40, pp. 247-251, 1990
- Julstrom, “A bound on the Shift Function in Terms of the Busy Beaver function”, ACM SIGACT, Vol. 23, Issue 3, 1992

Appendix A: Analyses of TM’s

Productively Equivalents

- Two machines M and M’ that have the same productivity are said to be productively equivalent. We write $M \equiv M'$.
- Obviously, one only needs to consider 1 machine out of each of the equivalence classes as defined by $\equiv$.
- Two important ways in which $M \equiv M'$:
  - M and M’ are structurally equivalent
  - M and M’ are functionally equivalent

Structural Equivalence

- Two machines M_1 and M_2 are structurally equivalent iff M_1's states can be renamed to form M_2. Example:

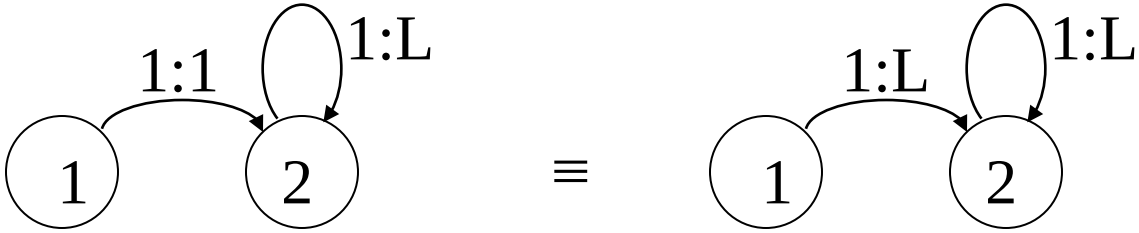
- With n states, there can be (n-1)! different but structurally equivalent graphs, so this becomes a significant reduction.

Further Useful Structural Considerations

- A machine is a ‘no path machine’ iff there exists no path from the start state to all other states
  - It is easily shown that BB(n) < BB(n+1) for any variant of BB. This can be used to show that any no path machine is not a Busy Beaver.
- A machine is a ‘fully connected machine’ iff it contains a fully connected subset of states S (not containing the halting state) such that every transition from a state in S goes to a state in S
  - Any fully connected machine is not a Busy Beaver

Functional Equivalence

- Two machines M_1 and M_2 are functionally equivalent iff $M_1(I) = M_2(I)$ for any input tape I. Example:



- With productivity being defined as the number of 1's left on the output tape when starting on an empty tape, $M_1 \equiv M_2$ if M_1 and M_2 are functionally equivalent (for many other definitions of productivity this is not the case!).

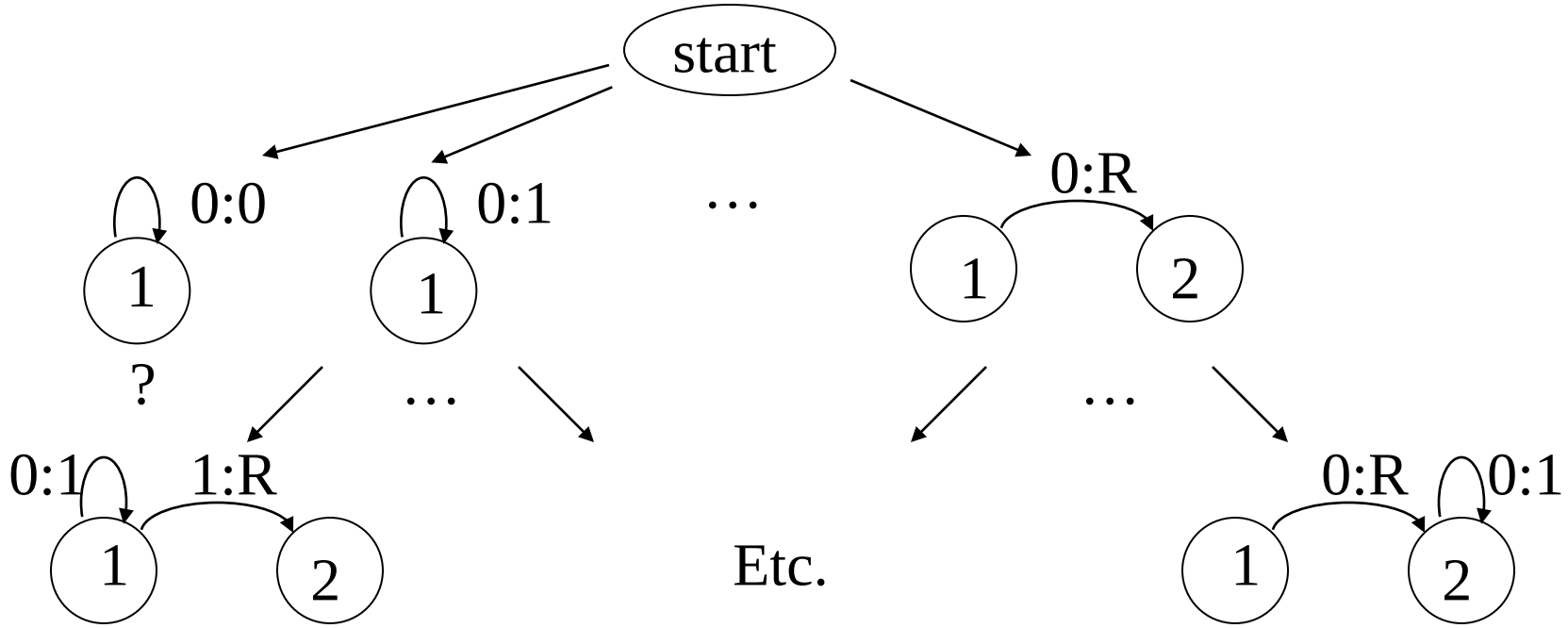
Useful Equivalences

Further Useful Functional Considerations

- Some types of machines:
  - A machine is a ‘empty tape machine’ iff starting on an empty tape, it reaches an empty tape after 1 or more steps
  - A machine is a ‘identical state machine’ iff it contains two states that have exactly the same transitions
  - A machine is a ‘unused transition machine’ iff it contains a transition that does not get used when starting on an empty tape
  - A machine is a ‘s:s machine’ iff it contains a s:s transition
  - A machine is a ‘s:s’+s’:s machine’ iff it contains a s:s’ transition to a state that has a s’:s transition
  - A machine M is a ‘mirror machine’ of a machine M’ iff swapping all L’s and R’s in the transitions of M results in M’

Tree Normalization: Step-by-Step Machine Construction

The tree normalization method generates candidate machines by adding transitions only as needed. This method inherently avoids considering machines that are different but structurally equivalent, unused transition machines, and no path machines.



Reducing the Number of Machines

- The number of machines generated during the tree normalization method can be further reduced (greatly) by using other structural and functional considerations:
  - Any fully connected machine is not a Busy Beaver
  - Any empty tape, s:s, or s:s'+s':s machine need not be considered, as any machine of any such type can be shown to either be a non-halter, or be functionally equivalent to a machine that is not of any such type
  - Only one of two mirror machines need be considered, as long as productivity is modified accordingly (i.e. it's ok to halt at the rightmost 1 of a consecutive string of 1's on an otherwise blank tape)