Optimum len $\Rightarrow L^*$

2-factor approx. soln $\Rightarrow$ 1236

$2L^* \geq 1236 \geq L^*$

$L^* \geq \frac{1236}{2}$

$L^* \geq$ ~~60~~ 618

Random algorithm

len	# times
1056	2
1076	18
1086	467
1098	120
1191	189
1317	99
1389	60
1537	45

~~2-approx~~. vertex cover (randomized)

while edges still uncovered:

take RANDOM edge

add one of endpoints RANDOMLY

If there's a ~~minimum~~ smallest vertex cover of size k, how long does the randomized algorithm take in the worst case?

↳ $n-1$ steps

(one vertex for each edge)

Statements about complexity classes

↳ From a computational view, NP-complete problems are the hardest ones out there. FALSE

↳ P = NP would imply all NP-complete problems are solvable in practice. FALSE (e.i.p. $\Rightarrow O(n^{1000})$)

↳ All NP-complete problems are solvable in polynomial time on non-deterministic machines TRUE

↳ All problems solveable in exponential time by a deterministic RAM take polynomial time on a non deterministic RAM (FALSE)

↳ It is likely that randomized algorithms can solve some NP complete problems in expected polynomial time FALSE

↳ If $P \neq NP$, then some NP-complete problems do not have constant factor approximation algorithms that run in polynomial time. TRUE

Limits of computation

There are certain problems that are provably impossible to solve.

Characteristics of a problem that's given to a computer:

1. Input is a finite string using a constant no. of symbols
2. Output is a finite string using a constant no. of symbols
3. Output is an objectively correct ~~and definitive~~ answer

There are problems that are characterized as above, and cannot be solved by a computer.

The Halting problem:

Input: A program P and i/p I for P.

Output: Does P ~~go into~~ will ever terminate on I?

Consider an algo. halt (P, I) that solves the above problem.

Only req: halt() solves problem in finite time.

Assume halting problem is decidable.

↳ Using the algo. halt (P, I)

Consider the program

<u>inverse_halt (program):</u>

if halt (program, program):
go into infinite loop
else :
return

→ doesn't halt if <u>program</u> halts
→ halts if <u>program</u> doesn't halt

Run inverse_halt (inverse_halt)

2 cases

↳ inverse_halt (inverse_halt) halts

⟹ halt (inverse_halt, inverse_halt) returns no

CONTRADICTION.

↳ inverse_halt (inverse_halt) doesn't halt

⟹ halt (inverse_halt, inverse_halt) returns YES

CONTRADICTION

⟹ Halting problem is undecidable

[Proof by contradiction]

<u>Implications</u>

An algorithm for this problem cannot exists.

For any real computer (finite memory), the halting problem is decidable.

① Simulate step 1 of program P
② If P, terminates then output YES. Else, record snapshot of machine
③ Compare snapshot against previous snapshots. If duplicate found, the output NO
④ Simulate next step of program and go to ②.

This argument is irrelevant in practice as in real computers, the memory is practically infinite (1 GB $\Rightarrow 2^{8,000,000,000}$ snapshots)

Another example of undecidability

Input : integer i

Output : # iterations to get to 1

$\Rightarrow$ i is even : $i \leftarrow i/2$

$\Rightarrow$ i is odd : $i \leftarrow 3i+1$

(collatz sequence)

} P

(P , i) $\Rightarrow$ not decidable

will P halt [redacted] for all i ?

---

using Halt ( ) . . . .

P : for $i = 1$ to 10^{10000}

run collatz rules for i

Halt (P[redacted]) $\Rightarrow$ Yes $\Rightarrow$ collatz conjecture stands true ~~from~~ upto 10^{10000}

↳ NO $\Rightarrow$ untrue

Halt ( ) is too powerful . . .