Back to computation as search

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

Back to computation as search

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

More than one `Next` may satisfy `arc(Node,Next)`
$\rightsquigarrow$ non-determinism

Back to computation as search

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

More than one `Next` may satisfy `arc(Node,Next)`
$\rightsquigarrow$ non-determinism

Choose `Next` closest to goal (heuristic h, Q)
perhaps keeping track of costs (min cost, A*)

Back to computation as search

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

More than one `Next` may satisfy `arc(Node,Next)`

$\leadsto$ non-determinism

Choose `Next` closest to goal (heuristic h, Q)

perhaps keeping track of costs (min cost, A*)

Available choices depend on `arc`

- actions specified by Turing machine (graph)

Back to computation as search

```
search(Node) :- goal(Node).

search(Node) :- arc(Node,Next), search(Next).
```

More than one `Next` may satisfy `arc(Node,Next)`

$\rightsquigarrow$ non-determinism

Choose `Next` closest to goal (heuristic h, Q)

perhaps keeping track of costs (min cost, A^*)

Available choices depend on `arc`

- actions specified by Turing machine (graph)

Computation eliminates non-determinism (determinization)

Back to computation as search

```
search(Node) :- goal(Node).
search(Node) :- arc(Node,Next), search(Next).
```

More than one `Next` may satisfy `arc(Node,Next)`
$\rightsquigarrow$ non-determinism

Choose `Next` closest to goal (heuristic h, Q)
perhaps keeping track of costs (min cost, A*)

Available choices depend on `arc`
- actions specified by Turing machine (graph)

Computation eliminates non-determinism (determinization)

Bound number of calls to `arc`   (iterations of `search`)

Terminating search

Search times out after too many `ticks`

```
bSearch(Node,_) :- goal(Node).
bSearch(Node,Bound) :- arc(Node,Next),
                       tick(Bound,Less),
                       bSearch(Next,Less).
```

Terminating search

Search times out after too many `ticks`

```
bSearch(Node,_) :- goal(Node).
bSearch(Node,Bound) :- arc(Node,Next),
                       tick(Bound,Less),
                       bSearch(Next,Less).
```

Design `tick` to be *terminating*

there is **no** infinite sequence $x_1, x_2, \ldots$ s.t.

$$\texttt{tick}(x_i, x_{i+1}) \text{ for every integer } i > 0$$

Terminating search

Search times out after too many `ticks`

```
bSearch(Node,_) :- goal(Node).
bSearch(Node,Bound) :- arc(Node,Next),
                       tick(Bound,Less),
                       bSearch(Next,Less).
```

Design `tick` to be *terminating*

there is **no** infinite sequence $x_1, x_2, \ldots$ s.t.

$$\texttt{tick}(x_i, x_{i+1}) \text{ for every integer } i > 0$$

and set `Bound` based on `Start`

```
search(Start) :- bound(Start,Bound),
                 bSearch(Start,Bound).
```

Feasibility and non-determinism: P vs NP

Cobham's Thesis

A problem is feasibly solvable iff some deterministic Turing machine (dTm) solves it in polynomial time.

$$P \;=\; \{\text{problems a dTm solves in polynomial time}\}$$

Feasibility and non-determinism: P vs NP

Cobham's Thesis

A problem is feasibly solvable iff some deterministic Turing machine (dTm) solves it in polynomial time.

$$P \ = \ \{\text{problems a dTm solves in polynomial time}\}$$

$$NP \ = \ \{\text{problems a non-deterministic Tm solves in polynomial time}\}$$

Feasibility and non-determinism: P vs NP

Cobham's Thesis

A problem is feasibly solvable iff some deterministic Turing machine (dTm) solves it in polynomial time.

$$P = \{\text{problems a dTm solves in polynomial time}\}$$

$$NP = \{\text{problems a non-deterministic Tm solves in polynomial time}\}$$

Clearly, $P \subseteq NP$.

Feasibility and non-determinism: P vs NP

Cobham's Thesis

A problem is feasibly solvable iff some deterministic Turing machine (dTm) solves it in polynomial time.

$$P = \{\text{problems a dTm solves in polynomial time}\}$$

$$NP = \{\text{problems a non-deterministic Tm solves in polynomial time}\}$$

Clearly, $P \subseteq NP$.

Whether $P = NP$ is the most celebrated open mathematical problem in computer science.

Feasibility and non-determinism: P vs NP

Cobham's Thesis

A problem is feasibly solvable iff some deterministic Turing machine (dTm) solves it in polynomial time.

$$P = \{\text{problems a dTm solves in polynomial time}\}$$

$$NP = \{\text{problems a non-deterministic Tm solves in polynomial time}\}$$

Clearly, $P \subseteq NP$.

Whether $P = NP$ is the most celebrated open mathematical problem in computer science.

$P \neq NP$ would mean non-determinism wrecks feasibility.

Feasibility and non-determinism: P vs NP

Cobham's Thesis

A problem is feasibly solvable iff some deterministic Turing machine (dTm) solves it in polynomial time.

$P = \{$problems a dTm solves in polynomial time$\}$

$NP = \{$problems a non-deterministic Tm solves in polynomial time$\}$

Clearly, $P \subseteq NP$.

Whether $P = NP$ is the most celebrated open mathematical problem in computer science.

$P \neq NP$ would mean non-determinism wrecks feasibility.

$P = NP$ says non-determinism makes no difference to feasibility.

A closer look

Given a set L of strings, and a Tm M.

A closer look

Given a set L of strings, and a Tm M.

M solves L in time n^k if there is a fixed integer $c > 0$ such that for every string s of size n,

$$s \in L \iff M \text{ accepts } s \text{ within } c \cdot n^k \text{ steps.}$$

A closer look

Given a set L of strings, and a Tm M.

M solves L in time n^k if there is a fixed integer $c > 0$ such that for every string s of size n,

$$s \in L \iff M \text{ accepts } s \text{ within } c \cdot n^k \text{ steps.}$$

$$\text{TIME}(n^k) := \{L \mid \text{some dTm solves } L \text{ in time } n^k\}$$

e.g. $\text{TIME}(n)$ includes every regular language

A closer look

Given a set L of strings, and a Tm M.

M solves L in time n^k if there is a fixed integer $c > 0$ such that for every string s of size n,

$$s \in L \iff M \text{ accepts } s \text{ within } c \cdot n^k \text{ steps.}$$

$$\text{TIME}(n^k) := \{L \mid \text{some dTm solves } L \text{ in time } n^k\}$$

e.g. $\text{TIME}(n)$ includes every regular language

$$P := \bigcup_{k \geq 1} \text{TIME}(n^k)$$

A closer look

Given a set L of strings, and a Tm M.

M solves L in time n^k if there is a fixed integer $c > 0$ such that for every string s of size n,

$$s \in L \iff M \text{ accepts } s \text{ within } c \cdot n^k \text{ steps.}$$

$$\text{TIME}(n^k) := \{L \mid \text{some dTm solves } L \text{ in time } n^k\}$$

e.g. $\text{TIME}(n)$ includes every regular language

$$P := \bigcup_{k \geq 1} \text{TIME}(n^k)$$

$$\text{NTIME}(n^k) := \{L \mid \text{some nTm solves } L \text{ in time } n^k\}$$

$$NP := \bigcup_{k \geq 1} \text{NTIME}(n^k)$$

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

e.g., $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ true is easy (P).

e.g., $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ true is easy (P). Non-determinism (guessing the assignment) puts SAT in NP.

e.g., $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ true is easy (P).
Non-determinism (guessing the assignment) puts SAT in NP.
But is SAT in P? There are 2^n assignments to try.

$$\text{e.g., } (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ true is easy (P).
Non-determinism (guessing the assignment) puts SAT in NP.
But is SAT in P? There are 2^n assignments to try.

Cook-Levin Theorem. *SAT is in P iff $P = NP$.*

$$\text{e.g., } (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ true is easy (P).
Non-determinism (guessing the assignment) puts SAT in NP.
But is SAT in P? There are 2^n assignments to try.

Cook-Levin Theorem. *SAT is in P iff $P = NP$.*

$$\text{e.g., } (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

CSAT: φ is a conjunction of clauses, where a *clause* is an OR of literals, and a *literal* is a variable x_i or negated variable $\overline{x_i}$

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \ldots, x_n$, can we make φ true by assigning true/false to $x_1, \ldots, x_n$?

Checking that a particular assignment makes φ true is easy (P).
Non-determinism (guessing the assignment) puts SAT in NP.
But is SAT in P? There are 2^n assignments to try.

Cook-Levin Theorem. *SAT is in P iff $P = NP$.*

$$\text{e.g., } (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

CSAT: φ is a conjunction of clauses, where a *clause* is an OR of literals, and a *literal* is a variable x_i or negated variable $\overline{x_i}$

k-SAT: every clause has exactly k literals
3-SAT is as hard as SAT, 2-SAT is in P

Boolean satisfiability (SAT)

SAT. Given a Boolean expression φ with variables $x_1, \dots, x_n$, can we make φ true by assigning true/false to $x_1, \dots, x_n$?

Checking that a particular assignment makes φ true is easy (P).
Non-determinism (guessing the assignment) puts SAT in NP.
But is SAT in P? There are 2^n assignments to try.

Cook-Levin Theorem. *SAT is in P iff $P = NP$.*

$$\text{e.g., } (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3})$$

CSAT: φ is a conjunction of clauses, where a *clause* is an OR of literals, and a *literal* is a variable x_i or negated variable $\overline{x_i}$

k-SAT: every clause has exactly k literals
3-SAT is as hard as SAT, 2-SAT is in P

Horn-SAT: every clause has at most one positive literal — linear

Prolog and SAT

Prolog KB (definite clauses)

```
x1 :- x2,x4.
x2 :- x3.        ⇝    [[x1,x2,x4],[x2,x3],[x4]]
x4.
```

Prolog and SAT

Prolog KB (definite clauses)

```
x1 :- x2,x4.
x2 :- x3.        ⇝    [[x1,x2,x4],[x2,x3],[x4]]
x4.
```

CSAT-input

$\texttt{x1} \lor \overline{\texttt{x2}} \lor \overline{\texttt{x4}}$

$\texttt{x2} \lor \overline{\texttt{x3}}$ $\quad\rightsquigarrow\quad$ `[[1,-2,-4],[2,-3],[4]]`

`x4.`

Prolog and SAT

Prolog KB (definite clauses)

```
x1 :- x2,x4.
x2 :- x3.        ⇝   [[x1,x2,x4],[x2,x3],[x4]]
x4.
```

CSAT-input

$\texttt{x1} \vee \overline{\texttt{x2}} \vee \overline{\texttt{x4}}$

$\texttt{x2} \vee \overline{\texttt{x3}}$ $\quad \rightsquigarrow \quad$ `[[1,-2,-4],[2,-3],[4]]`

`x4.`

The assignment making all variables TRUE satisfies all CSAT-inputs in which every clause has a positive literal.
(All definite clause KBs are satisfiable.)

Prolog and SAT

Prolog KB (definite clauses)

```
x1 :- x2,x4.
x2 :- x3.        ⇝    [[x1,x2,x4],[x2,x3],[x4]]
x4.
```

CSAT-input

$$\mathtt{x1} \vee \overline{\mathtt{x2}} \vee \overline{\mathtt{x4}}$$
$$\mathtt{x2} \vee \overline{\mathtt{x3}} \qquad \rightsquigarrow \qquad \mathtt{[[1,-2,-4],[2,-3],[4]]}$$
$$\mathtt{x4.}$$

The assignment making all variables TRUE satisfies all CSAT-inputs in which every clause has a positive literal. (All definite clause KBs are satisfiable.)

From proofs to unsatisfiability:

$$\underbrace{KB \text{ proves } \varphi}_{\text{Prolog}} \iff \underbrace{KB, \overline{\varphi}}_{\text{Horn (linear SAT)}} \text{ is not satisfiable}$$