Reductions

Lecture 22

April 17, 2025

Course Outline

- Part I: models of computation (reg exps, DFA/NFA, CFGs, TMs)
- Part II: (efficient) algorithm design
- Part III: limits of (efficient) computation
    - Undecidablity: problems that have *no* algorithms
    - NP-Completeness: problems that (we think) have no *efficient* algorithms

Course Outline

- Part I: models of computation (reg exps, DFA/NFA, CFGs, TMs)
- Part II: (efficient) algorithm design
- Part III: limits of (efficient) computation
  - Undecidablity: problems that have *no* algorithms
  - NP-Completeness: problems that (we think) have no *efficient* algorithms

Key tool for proving intractability: reductions!

Part I

Reductions for Algorithms

Recall: Longest Sequences

Longest Increasing Subsequence: Find the longest subsequence of $A[1..n]$ such that each term is larger than the last.

- Can solve using DP in $O(n^2)$ time.

Recall: Longest Sequences

Longest Increasing Subsequence: Find the longest subsequence of $A[1..n]$ such that each term is larger than the last.

- Can solve using DP in $O(n^2)$ time.

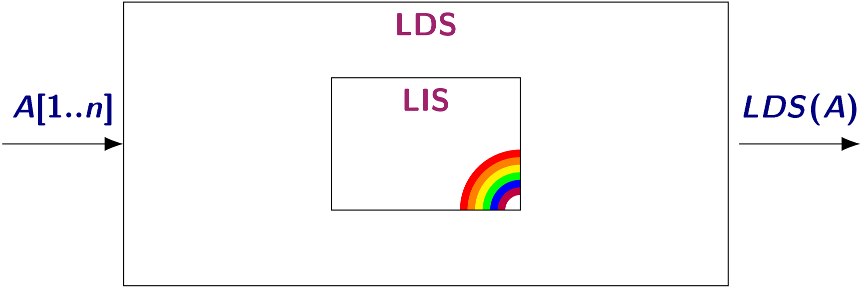
Longest *Decreasing* Subsequence: Find the longest subsequence of $A[1..n]$ such that each term is *smaller* than the last.

LDS Reduction

```
from magic import LIS
LDS(A[1..n]):
    Negate every element of A
    Compute seq = LIS(A)
    Negate every element of seq
    Return seq
```

LDS Reduction

```
from magic import LIS
LDS(A[1..n]):
    Negate every element of A
    Compute seq = LIS(A)
    Negate every element of seq
    Return seq
```

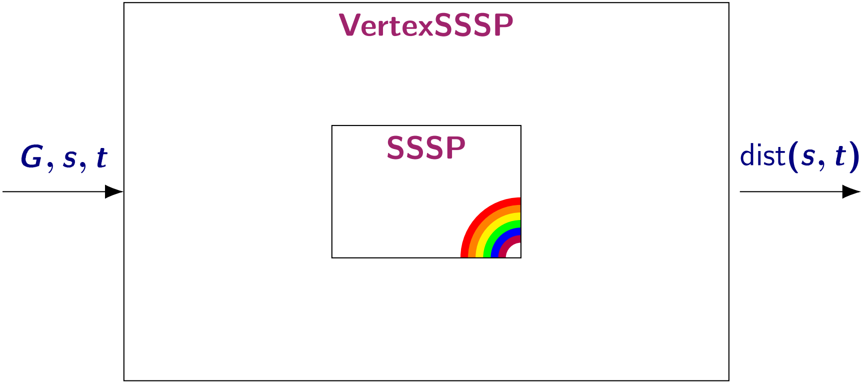


Vertex Weights

Say we have a graph $G = (V, E)$ with weights on the *vertices*. How do we find the length of the shortest path from s to t?

Vertex Weights

Say we have a graph $G = (V, E)$ with weights on the *vertices*. How do we find the length of the shortest path from s to t?



Why Reductions?

Reductions allow us to simplify algorithm design by making use of *abstraction barriers*.

- I don't need to know *how* we solve SSSP—someone else already wrote the code for that!

Why Reductions?

Reductions allow us to simplify algorithm design by making use of *abstraction barriers*.

- I don't need to know *how* we solve SSSP—someone else already wrote the code for that!
- Easy to slot in a different algorithm for SSSP if needed.

Why Reductions?

Reductions allow us to simplify algorithm design by making use of *abstraction barriers*.

- I don't need to know *how* we solve SSSP—someone else already wrote the code for that!
- Easy to slot in a different algorithm for SSSP if needed.

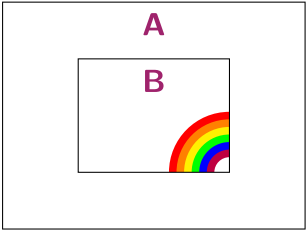
Tells us the relative "difficulty" of computational tasks.

Why Reductions?

Reductions allow us to simplify algorithm design by making use of *abstraction barriers*.

- I don't need to know *how* we solve SSSP—someone else already wrote the code for that!
- Easy to slot in a different algorithm for SSSP if needed.

Tells us the relative "difficulty" of computational tasks.



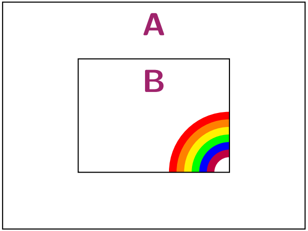
A is "no harder than" ***B***: any algorithm for ***B*** gives one for ***A***.

Why Reductions?

Reductions allow us to simplify algorithm design by making use of *abstraction barriers*.

- I don't need to know *how* we solve SSSP—someone else already wrote the code for that!
- Easy to slot in a different algorithm for SSSP if needed.

Tells us the relative "difficulty" of computational tasks.



A is "no harder than" ***B***: any algorithm for ***B*** gives one for ***A***.

B is "no easier than" ***A***: if ***A*** has no "good" algorithm, neither does ***B***!

Part II

Practice with Reductions

Autograding CS 124

Say we've asked a CS 124 student to write a "Hello World!" program.

Autograding CS 124

Say we've asked a CS 124 student to write a "Hello World!" program.

We're expecting:

```
main():
    print(``Hello World!'')
```

Autograding CS 124

Say we've asked a CS 124 student to write a "Hello World!" program.

We're expecting:

```
main():
    print(''Hello World!'')
```

Can we (algorithmically) check if a student's code works?

Autograding CS 124

Say we've asked a CS 124 student to write a "Hello World!" program.

We're expecting:

```
main():
    print(‘‘Hello World!’’)
```

Can we (algorithmically) check if a student's code works?

Intuitively simpler question: can we check if a student's code at least doesn't run forever?

Reducing “Hello World!” to Halting

We want to reduce checking if a student’s code prints “Hello World!” and halts to just checking if it (eventually) halts.

```
from magic import TestHalt
TestHW(StudentCode):
```

Reducing "Hello World!" to Halting

We want to reduce checking if a student's code prints "Hello World!" and halts to just checking if it (eventually) halts.

```
from magic import TestHalt
TestHW(StudentCode):
    Create program P that:
        (1) Runs StudentCode
        (2) If the terminal doesn't say ''Hello World!'',
            enters an infinite loop
    return TestHalt(P)
```

Note: ***P*** halts if and only if `StudentCode` is correct!

Reducing “Hello World!” to Halting

We want to reduce checking if a student’s code prints “Hello World!” and halts to just checking if it (eventually) halts.

```
from magic import TestHalt
TestHW(StudentCode):
    Create program P that:
        (1) Runs StudentCode
        (2) If the terminal doesn't say ``Hello World!'',
            enters an infinite loop
    return TestHalt(P)
```

Note: ***P*** halts if and only if `StudentCode` is correct!

This tells us that checking if a student’s code is correct is “no harder than” just checking if it runs forever.

Reducing Halting to "Hello World"

We can use these same ideas to reduce in the opposite direction!

```
from magic import TestHW
TestHalt(StudentCode):
    Create a program P that:
        (1) Runs StudentCode (supressing print statements)
        (2) prints ``Hello World!''
    return TestHW(P)
```

Note: ***P*** will halt and print "Hello World!" if and only if `StudentCode` halts.

Reducing Halting to "Hello World"

We can use these same ideas to reduce in the opposite direction!

```
from magic import TestHW
TestHalt(StudentCode):
    Create a program P that:
        (1) Runs StudentCode (supressing print statements)
        (2) prints ''Hello World!''
    return TestHW(P)
```

Note: ***P*** will halt and print "Hello World!" if and only if `StudentCode` halts.

This means that checking if a student's code halts is "no harder than" checking if it's correct—so the two tasks are the same "level of difficulty"!

Independent Set and Clique

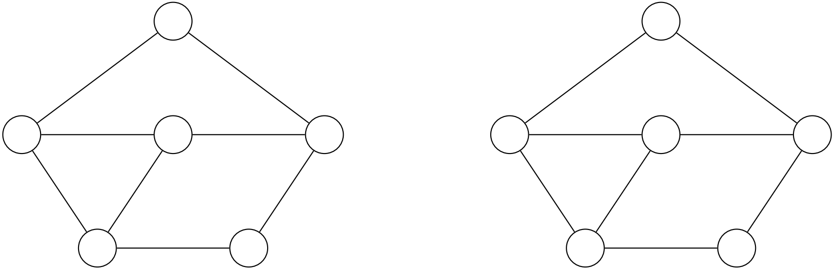
Given a graph $G = (V, E)$, we define

- A **clique** as $C \subseteq V$ such that each vertex in C has an edge to *every* other vertex in C.
- A **independent set** as $S \subseteq V$ such that no two vertices in S have an edge between them.

Independent Set and Clique

Given a graph $G = (V, E)$, we define

- A **clique** as $C \subseteq V$ such that each vertex in C has an edge to *every* other vertex in C.
- A **independent set** as $S \subseteq V$ such that no two vertices in S have an edge between them.



Cliques   Independent Sets

Independent Set and Clique

Given a graph $G = (V, E)$, we define

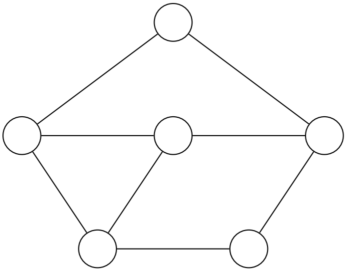
- A **clique** as $C \subseteq V$ such that each vertex in C has an edge to *every* other vertex in C.
- A **independent set** as $S \subseteq V$ such that no two vertices in S have an edge between them.

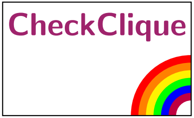
Problems of interest: given a graph G and an integer $1 \leq k \leq |V|$,

- Does G have a clique of size k?
- Does G have an independent set of size k?

Reducing Independent Set to Clique I

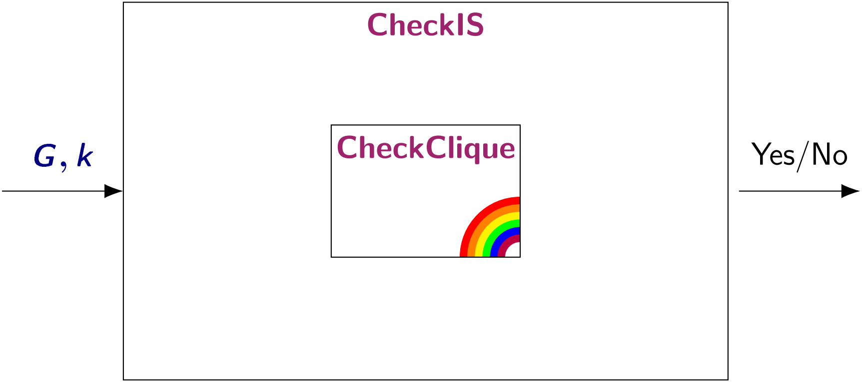
Say we wanted to check if there is an independent set of size 3 in this graph. How can we use `CheckClique` to help?





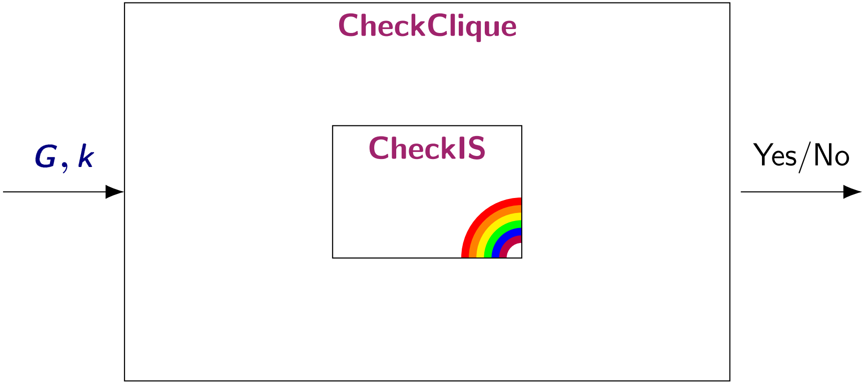
Reducing Independent Set to Clique II

We want to check if G has an independent set of size k, given the ability to check if a graph has a clique of some size.



Reducing Clique to Independent Set

We want to check if G has a clique of size k, given the ability to check if a graph has an independent set of some size.



So Independent Set and Clique are "as easy / difficult" as each other!

Not All Reductions Work Both Ways

Both reductions we've seen so far can work in either direction—but this isn't always the case!

Not All Reductions Work Both Ways

Both reductions we've seen so far can work in either direction—but this isn't always the case!

Can we reduce checking for a clique to checking if a program halts?

```
from magic import TestHalt
CheckClique(G, k):
```

Not All Reductions Work Both Ways

Both reductions we've seen so far can work in either direction—but this isn't always the case!

Can we reduce checking for a clique to checking if a program halts?

Can we reduce checking if a program halts to checking for a clique?

```
from magic import CheckClique
TestHalt(StudentCode):
```

Not All Reductions Work Both Ways

Both reductions we've seen so far can work in either direction—but this isn't always the case!

Can we reduce checking for a clique to checking if a program halts?

Can we reduce checking if a program halts to checking for a clique?

It turns out this is impossible! (We'll see why in the next lecture.)

Takeaway: It matters which direction your reduction goes—some problems really are "strictly harder" than others!

Search Versus Decision

So far, everything we've discussed have been *decision problems*—we just want to know if something is true. (eg, "is there a clique of size k?")

Search Versus Decision

So far, everything we've discussed have been *decision problems*—we just want to know if something is true. (eg, "is there a clique of size k?")

In practice, we often want to actually *find* an answer if one exists.

Search Versus Decision

So far, everything we've discussed have been *decision problems*—we just want to know if something is true. (eg, "is there a clique of size k?")

In practice, we often want to actually *find* an answer if one exists.

Concretely, how do the following problems compare?

- `CheckClique`(G, k): check if G has a clique of size k.
- `FindClique`(G, k): output a clique of size k if one exists, or say "Not possible".

Search Versus Decision

So far, everything we've discussed have been *decision problems*—we just want to know if something is true. (eg, "is there a clique of size k?")

In practice, we often want to actually *find* an answer if one exists.

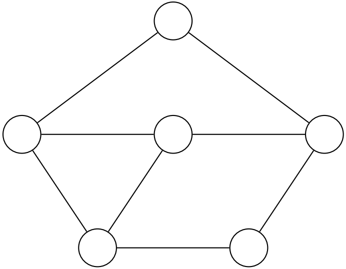
Concretely, how do the following problems compare?

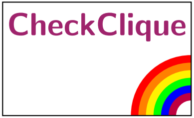
- `CheckClique`(G, k): check if G has a clique of size k.
- `FindClique`(G, k): output a clique of size k if one exists, or say "Not possible".

Immediate: if we can solve `FindClique`, we can solve `CheckClique`.

Clique Search to Decision Intuition

Say we wanted to find a clique of size 3 in this graph. How can we use `CheckClique` to help?





Clique Search to Decision Reduction

```
from magic import CheckClique
FindClique(G, k):
    if CheckClique(G, k) is false:
        Return ``Not possible''
    for each vertex v ∈ G:
        Construct G' by removing v (and its edges) from G
        if CheckClique(G', k) is true:
            Remove v (and its edges) from G
    Return remaining vertices
```

Correctness?

Clique Search to Decision Reduction

```
from magic import CheckClique
FindClique(G, k):
    if CheckClique(G, k) is false:
        Return ‘‘Not possible’’
    for each vertex v ∈ G:
        Construct G′ by removing v (and its edges) from G
        if CheckClique(G′, k) is true:
            Remove v (and its edges) from G
    Return remaining vertices
```

Correctness?

This means that checking if a clique exists and actually finding one "as easy / difficult" as each other!

Part III

Reductions for Decision Problems

Decision Problems

Similar to the first third of the class, we will be mostly interested in *decision* problems: our answer is either “Yes” or ”No”.

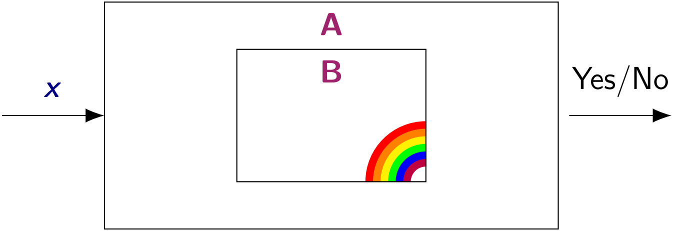
Decision Problems

Similar to the first third of the class, we will be mostly interested in *decision* problems: our answer is either "Yes" or "No".

Recall that we often refer to such problems as a "language" $L \subseteq \{0, 1\}^*$—strings in L are exactly those that we want to output "Yes" on (ie, accept).

Reducing Decision Problems

Say we have two decision problems A and B. We can reduce A to B by giving a function f such that $x \in L_A$ if and only if $f(x) \in L_B$.



Reducing Decision Problems

Say we have two decision problems A and B. We can reduce A to B by giving a function f such that $x \in L_A$ if and only if $f(x) \in L_B$.

Examples from before:

- Reducing Independent Set to Clique, we took $f(G, k) = (\overline{G}, k)$.
- Reducing "Hello World!" to halting, we took $f(\texttt{StudentCode})$ to be the program P we defined.

Reducing Decision Problems

Say we have two decision problems A and B. We can reduce A to B by giving a function f such that $x \in L_A$ if and only if $f(x) \in L_B$.

Examples from before:

- Reducing Independent Set to Clique, we took $f(G, k) = (\overline{G}, k)$.
- Reducing "Hello World!" to halting, we took $f(\texttt{StudentCode})$ to be the program P we defined.

When using this type of reduction, you just have to define f and prove that $x \in L_A$ iff $f(x) \in L_B$!

- This will be the most common type of reduction we use because it is the most simple.

Takeaway Points

Reductions are a powerful tool in CS

- Reducing A to a problem with a known algorithm gives us an algorithm for A.
- Reducing a "hard" problem to B tells us that B must also be "hard".

To reduce A to B, write an algorithm for A where we can use a subroutine that solves B.

- *Don't* worry about how the subroutine for B is implemented—just use that it solves B!
- For decision problems, it suffices to give a function f such that $x \in L_A$ if and only if $f(x) \in L_B$.