Lecture 6 — Reduction: Use A to Solve B

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

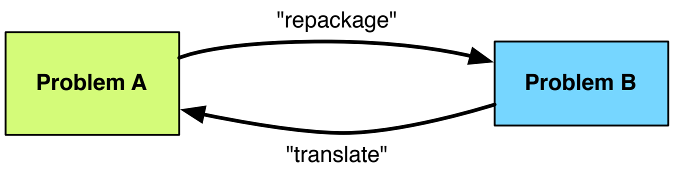
Lectured by Kanat Tangwongsan — February 2, 2012

Material in this lecture: Today's lecture is about *reduction*.

- Reduction
- Genome sequencing, or actually finding shortest superstring.
- (Asymmetric) Traveling Salesperson
- Graphs

1 Reduction: How to Deligate Your Work

Often, the easiest way to solve a problem is to make someone else do it for you. This is the key idea behind reduction techniques. More seriously, if we want to solve a problem A, we could reduce the problem—i.e., we repackage it—to another problem B with hopes that we have known algorithms for B or it is easier to reason about B. The challenge is usually in recognizing similarities between problems that on the surface seem very different—and being able to come up with the translation. This will come with practice.



You probably have encountered this simple but powerful idea in many places already.

Example I: In the MCSS problem, we're given a sequence S of numbers, find a contiguous subsequence of S that has the largest sum. We said

$$\texttt{mcss}(S) = \max_j \left\{ X_j - \min_{i \le j} X_i \right\},$$

where the X_j are the partial sums of S (i.e., $X_j = sum_{i=1}^{j} S_i$). But it is not hard to see that

$$\max_j \left\{ X_j - \min_{i \le j} X_i \right\} = \max_{j \ge i} \{X_j - X_i\},$$

which is exactly the stocks market problem. Therefore, we can use stocks to solve MCSS:

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

```
fun mcss S =
  let val X = scan_incl plus 0 S    <--- repackaing
  in case (stocks X)
       of JUMP(x) => x              <--- translation
        | _ => 0
  end
```

Example II: We sometimes want to show that a problem is not easy. One way of doing it is to show that it is as hard as some other problem. If that problem is known to be hard, then we can conclude that our problem, too, is hard. More concretely, to prove that A is hard: Say we know that B is super hard (e.g., NP-hard). If we could reduce it to A (i.e. you could both repackage it and translate it back fast, say $O(n)$ cost), then, it must be the case A is hard, too. We're going to see more of these soon when we talk about lower bounds.

Let's look at a more sophisticated example.

2 Sequencing the Genome

Sequencing of a complete human genome represents one of the greatest scientific achievements of the century. It is also one of the most important contributions of algorithms to date. For a brief history, the efforts started a few decades ago with the following major landmarks:

1996	sequencing of first living species
2001	draft sequence of the human genome
2007	full human genome diploid sequence

Interestingly, all these achievements rely on efficient parallel algorithms. In this lecture, we will take a look at some algorithms behind the recent results—and the power of problem abstraction which will reveal surprising connections between seemingly unrelated problems.

2.1 What makes sequencing the genome hard?

There is currently no way to read long strands with accuracy. Current DNA sequencing machines are only capable of efficiently reading relatively short strands (e.g. 1000 base pairs). Therefore, we resort to cutting strands into shorter fragments and then reassembling the pieces. The "primer walking" process cuts the DNA strands into consecutive fragments. But the process is slow because you need the result of one fragment to "build" in the wet lab the molecule needed to find the following fragment (inherently sequential process). Alternatively, there are fast methods to cut the strand at random positions. But this process mixes up the short fragments, so the order of the fragments is unknown. For example, the strand cattaggagtat might turn into, say, ag, gag, catt, tat, destroying the original ordering.

If we could make copies of the original sequence, is there something we could do differently to order the pieces? Let's look at the shotgun method, which according to Wikipedia is the de facto standard for genome sequencing today. It works as follows:

1. Take a DNA sequence and make multiple copies. For example, if we are cattaggagtat, we produce many copies of it:

   cattaggagtat
   cattaggagtat
   cattaggagtat

2. Randomly cut up the sequences using a "shotgun", well, actually using radiation or chemicals. For instance, we could get

   $\boxed{\text{catt}}\ \boxed{\text{ag}}\ \boxed{\text{gagtat}}$
   $\boxed{\text{cat}}\ \boxed{\text{tagg}}\ \boxed{\text{ag}}\ \boxed{\text{tat}}$
   $\boxed{\text{ca}}\ \boxed{\text{tta}}\ \boxed{\text{gga}}\ \boxed{\text{gtat}}$

3. Sequence each of the short fragments, which can be done in parallel with multiple sequencing machines.

4. Reconstruct the original genome from the fragments.

Steps 1–3 are done in a wet lab, and **Step 4 is where algorithms come in.** In Step 4, we want to solve the following problem: *Given a set of overlapping genome subsequences, construct the "best" sequence that includes them all.* But what notion of quality are we talking about? What does it mean to be the "best" sequence? There are many possible candidates. Below is one way to define "best" objectively.

Definition 2.1 (The Shortest Superstring (SS) Problem)**.** Given an alphabet set Σ and a finite set of finite, non-empty strings $S \subseteq \Sigma^+$, the shortest superstring problem is find a shortest string r that contains every $s \in S$ as a substring of r.

Note that in this definition, we require each $s \in S$ to appear as a contiguous block in r. That is, "ag" is a substring of "ggag" but is *not* a substring of "attg".

That is, given sequence fragments, construct a string that contains all the fragments and that is the shortest string. The idea is that the simplest string is the best. Now that we have a concrete specification of the problem, we are ready to look into algorithms for solving it.

For starters, let's observe that we can ignore strings that are contained in other strings. That is, for example, if we have gagtat, ag, and gt, we can throw out ag and gt. Continuing with the example above, we are left with the following "snippets"

$$S = \left\{\text{tagg}, \text{catt}, \text{gga}, \text{tta}, \text{gagtat}\right\}.$$

Following this observation, we can assume that the "snippets" have been preprocessed so that none of the strings are contained in other strings. (How fast can we do this?)

2.2 Algorithm 1: Brute Force

The first algorithm we will look at is a brute force algorithm, because it tries all permutations of the set of strings and for each permutation, we remove the maximum overlap between each adjacent pair of strings. For example, the permutation

$$\text{catt}\ \ \underline{\text{tt}}\text{a}\ \ \underline{\text{ta}}\text{gg}\ \ \underline{\text{gg}}\text{a}\ \ \underline{\text{ga}}\text{gtat}$$

will give us cattaggagtat after removing the overlaps (the excised parts are underlined). Note that this result happens to be the original string and also the shortest superstring.

Does trying all permutations always give us the shortest string? As our intuition might suggest, the answer is yes and the proof of it, which we didn't go over in class, hints at an algorithm that we will look at in a moment.

Lemma 2.2. *Given a finite set of finite strings $S \subseteq \Sigma^+$, the brute force method finds the shortest superstring.*

Proof. Let r^* be any shortest superstring of S. We know that each string $s \in S$ appears in r^*. Let i_s denote the beginning position in r^* where s appears. Since we have eliminated duplicates, it must be the case that all i_s's are distinct numbers. Now let's look at all the strings in S, $s_1, s_2, \ldots, s_{|S|}$, where we number them such that $i_{s_1} < i_{s_2} < \cdots < i_{s_{|S|}}$. It is not hard to see that the ordering $s_1, s_2, \ldots, s_{|S|}$ gives us r^* after removing the overlaps. □

The problem with this approach is that, although highly parallel, it has to examine a large number of combinations, resulting in a super large work term. There are $n!$ permutations on a collection of n elements. This means that if the input consists of $n = 100$ strings, we'll need to consider $100! \approx 10^{158}$ combinations, which for a sense of scale, is more than the number of atoms in the universe. As such, the algorithm is not going to be feasible for large n.

Can we come up with a smarter algorithm that solves the problem faster? Unfortunately, it is unlikely. As it happens, this problem is NP-hard. But we should not fear NP-hard problems. In general, NP-hardness only suggests that there are families of instances on which the problem is hard in the worst-case. It doesn't rule out the possibility of algorithms that compute near optimal answers or algorithms that perform well on real world instances.

For this particular problem, we know efficient approximation algorithms that (1) give theoretical bounds that guarantee that the answer (i.e., the length) is within a constant factor of the optimal answer, and (2) in practice do even better than the bounds suggest.

2.3 Algorithm 2: Reducing to Another Problem

We now consider converting the shortest superstring problem to another seemingly unrelated problem: the traveling salesperson[1] (TSP) problem. This is a canonical NP-hard problem dating back to the 1930s and has been extensively studied, e.g. see Figure 1. The two major variants of the problem

[1]This is formerly known as the traveling salesman problem.

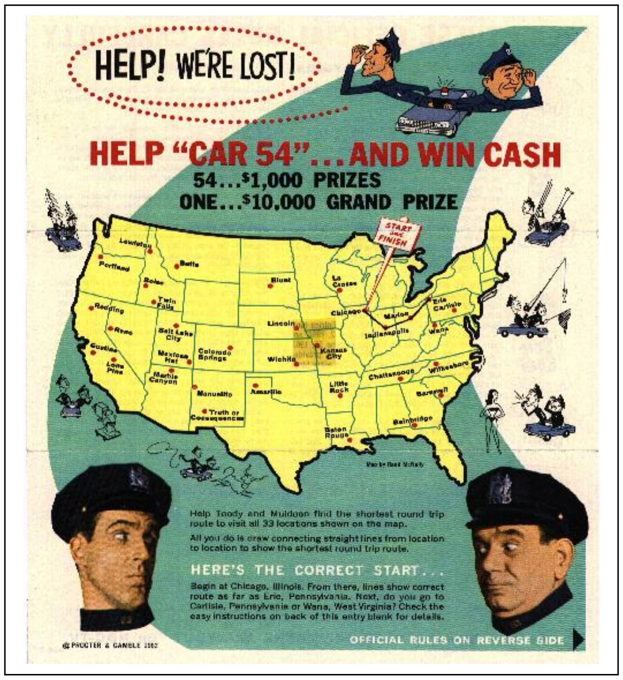

HELP! WE'RE LOST!
HELP "CAR 54"...AND WIN CASH
54...$1,000 PRIZES
ONE...$10,000 GRAND PRIZE
START and FINISH
Lewiston
Portland
Butte
Boise
Twin Falls
Redding
Reno
Salt Lake City
Gustine
Lone Pine
Mexican Hat
Colorado Springs
Marble Canyon
Manuelito
Truth or Consequences
Amarillo
Wichita
Lincoln
Blunt
La Crosse
Chicago
Kansas City
Little Rock
Baton Rouge
Chattanooga
Indianapolis
Marion
Erie
Carlisle
Wana
Wilkesboro
Barnwell
Bainbridge
Help Toody and Muldoon find the shortest round trip route to visit all 33 locations shown on the map.
All you do is draw connecting straight lines from location to location to show the shortest round trip route.
HERE'S THE CORRECT START...
Begin at Chicago, Illinois. From there, lines show correct route as far as Erie, Pennsylvania. Next, do you go to Carlisle, Pennsylvania or Wana, West Virginia? Check the easy instructions on back of this entry blank for details.
OFFICIAL RULES ON REVERSE SIDE
© PROCTER & GAMBLE 1962

Figure 1: A poster from a contest run by Proctor and Gamble in 1962. The goal was to solve a 33 city instance of the TSP. Gerald Thompson, a Carnegie Mellon professor, was one of the winners.

are *symmetric* TSP and *asymmetric* TSP, depending on whether the graph has undirected or directed edges, respectively. The particular variant we're reducing to is the asymmetric version, which can be described as follows.

Definition 2.3 (The Asymmetric Traveling Salesperson (aTSP) Problem)**.** Given a weighted directed graph, find the shortest path that starts at a vertex s and visits all vertices exactly once before returning to s.

That is, find a Hamiltonian cycle of the graph such that the sum of the edge weights along the cycle is the minimum of all such cycles.

Motivated by the observation that the shortest superstring problem can be solved exactly by trying all permutations, we'll make the TSP problem try all the combinations for us. For this, we will set up a graph so that each valid Hamiltonian cycle corresponds to a permutation. Note that in our case, there is always a Hamiltonian cycle because the graph is a complete graph—there are two directed edges between every pair of vertices.

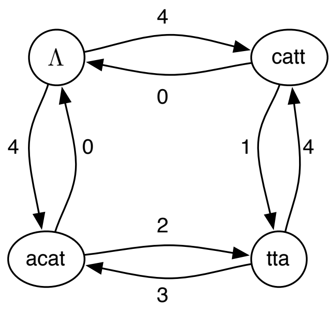
Let `overlap`(s_i, s_j) denote the maximum overlap for s_i followed by s_j. This would mean `overlap` ("tagg","gga") $= 2$.

The Reduction. Now we build a *directed* graph $D = (V, A)$.

- The vertex set V has one vertex per string and a special "source" vertex Λ where the cycle starts and ends.

- The arc (directed edge) from s_i to s_j has weight $w_{i,j} = |s_j| - \texttt{overlap}(s_i, s_j)$. This quantity represents the increase in the string's length if s_i is followed by s_j. As an example, if we have "tagg" followed by "gga", then we can generate "tagga" which only adds 1 character—indeed, $|\text{“gga”}| - \texttt{overlap}(\text{“tagg”}, \text{“gga”}) = 3 - 2 = 1$.
- The weights for arcs incident to Λ are set as follows: $(\Lambda, s_i) = |s_i|$ and $(s_i, \Lambda) = 0$. That is, if s_i is the first string in the permutation, then the arc (Λ, s_i) pays for the whole length s_i.

To see this reduction in action, the input {catt, acat, tta} results in the following graph (not all edges are shown).



As intended, in this graph, a cycle through the graph that visits each vertex once corresponds to a permutation in the brute force method. Furthermore, the sum of the edge weights in that cycle is equal to the length of the superstring produced by the permutation. Since TSP considers all Hamiltonian cycles, it also corresponds to considering all orderings in the brute force method. Since the TSP finds the min cost cycle, and assuming the brute force method is correct, then TSP finds the shortest superstring. Therefore, if we could solve TSP, we would be able to solve the shortest superstring problem.

But TSP is also NP-hard. What we have accomplished so far is that we have reduced one NP hard problem to another, but the advantage is that there is a lot known about TSP, so maybe this helps.

3 Graphs

As we just saw, graphs offer a great way of expressing relationships between pairs of items. Indeed, graphs are one of the most important abstractions in the study of algorithms. A graph is also known as a network. Graphs can be very important in modeling data, and a large number of problems can be reduced to known graph problems. A graph consists of a set of vertices with connections between them. Graphs can either be directed, with the directed edges (arcs) pointing from one vertex to another, or undirected, with the edges symmetrically connecting vertices. Graphs can have weights or other values associated with either the vertices and/or the edges.

For motivation, we'll outline some of the many applications of graphs:

1. *Road networks.* Vertices are intersections and edges are the road segments between them. Such networks are used by google maps, Bing maps and other map programs to find routes between

locations. They are also used for studying traffic patterns. Algorithmic questions we'll cover in this class include TSP, reachability, and shortest paths.

2. *Dependence graphs.* Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.

3. *Utility graphs.* The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

4. *Document link graphs.* The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

5. *Protein-protein interactions graphs.* Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

6. *Neural networks.* Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses.

7. *Network traffic graphs.* Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

8. *Social network graphs.* Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who tweeted whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

9. *Graphs in quantum field theory.* Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

10. *Semantic networks.* Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

11. *Scene graphs.* In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

12. *Finite element meshes.* In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.

13. *Robot planning.* Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

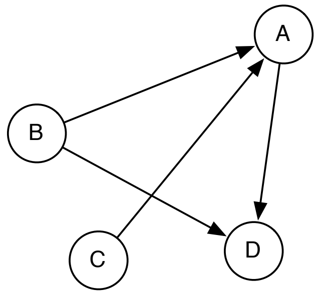
14. *Graphs in epidemiology.* Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

15. *Graphs in compilers.* Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

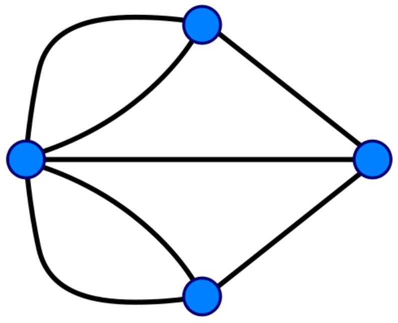
16. *Constraint graphs.* Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

There are many other applications of graphs.

3.1 Formalities



An example of a directed graph on 4 vertices.



An undirected graph on 4 vertices[2]

Formally, a *directed graph* or (*digraph*) is a pair $G = (V, A)$ where

- V is a set of *vertices* (or nodes), and
- $A \subseteq V \times V$ is a set of *directed edges* (or arcs).

Each arc is an ordered pair $e = (u, v)$ and a graph can have *self loops* (u, u). Directed graphs represent asymmetric relationships. An *undirected graph* is a pair $G = (V, E)$ where E is a set of unordered pairs over V (i.e., $E \subseteq \binom{V}{2}$). Each edge can be written as $e = \{u, v\}$, which is the same as $\{v, u\}$, and self loops are generally *not* allowed. Undirected graphs represent symmetric relationships.

Note that directed graphs are in some sense more general than undirected graphs since we can easily represent an undirected graph by a directed graph by placing an arc in each direction. Indeed, this is often the way we represent directed graphs in data structures.

Unfortunately, graphs come with a lot of terminology. Though, fortunately, most of it is intuitive once we understand the concept. At this point, we will just talk about graphs that do not have any data associated with edges, such as weights. We will later talk about weighted graphs.

- A vertex u is a *neighbor* of or equivalently *adjacent* to a vertex v if there is an edge between them. For a directed graph we use the terms *in-neighbor* (if the arc points to u) and *out-neighbor* (if the arc points from u).
- The *degree* of a vertex is its number of neighbors and will be denoted as $d_G(v)$. For directed graphs we use *in-degree* ($d_G^-(v)$) and *out-degree* ($d_G^+(v)$) with the presumed meanings. We will drop the subscript when it is clear from the context which graph we're talking about.
- For an undirected graph $G = (V, E)$, the *neighborhood* $N_G(v)$ of a vertex $v \in V$ is its set of neighbors, i.e. $N(v) = \{u \mid \{u, v\} \in E\}$. For a directed graph we use $N_G^+(v)$ to indicate the set of out-neighbors and $N_G^-(v)$ to indicate the set of in-neighbors of v. If we use $N_G(v)$ for a directed graph, we mean the out neighbors. The neighborhood of a set of vertices $U \subseteq V$ is the union of their neighborhoods, e.g. $N_G(V) = \cup_{v \in V} N_G(v)$, or $N_G^+(V) = \cup_{v \in V} N_G^+(v)$.
- A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, $\mathit{Paths}(G) = \left\{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\right\}$ is the set of all paths in G, where V^+ indicates all positive length sequences of vertices (allowing for repeats). The length of a path is one less than the number of vertices in the path—i.e., it is the number of edges in the path.
- A vertex v is *reachable* from a vertex u in G if there is a path starting at v and ending at u in G. An undirected graph is *connected* if all vertices are reachable from all other vertices. A directed graph is *strongly connected* if all vertices are reachable from all other vertices.
- A *simple path* is a path with no repeated vertices. Often, however, the term simple is dropped, making it sometimes unclear whether path means simple or not (sorry). In this course we will almost exclusively be talking about simple paths and so unless stated otherwise our use of path means simple path.
- A *cycle* is a path that starts and ends at the same vertex. In a directed graph a cycle can have length 1 (i.e. a *self loop*). In an undirected graph we require that a cycle must have length at least three. In particular going from v to u and back to v does not count. A *simple cycle* is a cycle that has no repeated vertices other than the start vertex being the same as the end. In this course we will exclusively talk about simple cycles and hence, as with paths, we will often drop simple.
- An undirected graph with no cycles is a *forest* and if it is connected it is called a *tree*. A directed graph is a forest (or tree) if when all edges are converted to undirected edges it is undirected forest (or tree). A *rooted tree* is a tree with one vertex designated as the root. For a directed graph the edges are typically all directed toward the root or away from the root.

Truth in advertising. Often when abstracting a problem we can abstract away some key aspects of the underlying application that we want to solve. Indeed this is the case when using the Shortest Superstring problem for sequencing genomes. In actual genome sequencing there are two shortcomings with using the SS problem. The first is that when reading the base pairs using a DNA sequencer there can be errors. This means the overlaps on the strings that are supposed to overlap perfectly might not. Don't fret: this can be dealt with by generalizing the Shortest Superstring problem to deal with approximate matching. Describing such a generalization is beyond the scope of this course, but basically one can give a score to every overlap and then pick the best one for each pair of fragments. The nice thing is that the same algorithmic techniques we discussed for the SS problem still work for this generalization, only the "overlap" scores will be different.

The second shortcoming of using the SS problem by itself is that real genomes have long repeated sections, possibly much longer than the length of the fragments that are sequenced. The SS problem does not deal well with such repeats. In fact when the SS problem is applied to the fragments of an initial string with longer repeats that the fragment sizes, the repeats or parts of them are removed. One method that researchers have used to deal with this problem is the so-called *double-barrel shotgun method*. In this method strands of DNA are cut randomly into lengths that are long enough to span the repeated sections. After cutting it up one can read just the two ends of such a strand and also determine its length (approximately). By using the two ends and knowing how far apart they are it is possible to build a "scaffolding" and recognize repeats. This method can be used in conjunction with the generalization of the SS discussed in the previous paragraph. In particular the SS method allowing for errors can be used to generate strings up to the length of the repeats, and the double barreled method can put them together.

- For a graph $G = (V, E)$ the (unweighted) *shortest path length* between two vertices is the minimum length of a path between them : $SP_G(u, v) = \min\left\{|P| \mid P \in Paths(G), P_1 = u, P_{|P|} = v\right\}$.
- The *diameter* of a graph is the maximum shortest path length over all pairs of vertices: $\mathsf{diam}(G) = \max\left\{SP_G(u, v) : u, v \in V\right\}$.

Sometimes graphs allow multiple edges between the same pair of vertices, called *multi-edges*. Graphs with multi-edges are called *multi-graphs*. We will allow multi-edges in a couple algorithms just for convenience.

By convention we will use the following definitions:

$$\begin{aligned} n &= |V| \\ m &= |E| \end{aligned}$$

Note that a directed graph can have at most n^2 edges (including self loops) and an undirected graph at most $n(n-1)/2$. We informally say that a graph is *sparse* if $m \ll n^2$ and *dense* otherwise. In most applications graphs are very sparse, often with only a handful of neighbors per vertex when averaged across vertices, although some vertices could have high degree. Therefore, the emphasis in the design of graph algorithms, at least for this class, is typically on algorithms that work well for sparse graphs.