Chapter 11

Graph Neural Networks

Neural networks have been successful in handling various forms of data, but they are strongest when they work with highly structured and ordered input data: simple scalars, vectors, matrices (images, temporal data), tensors (multi-dimensional matrices).

Since some of the world's most interesting data is represented by graphs, it would be great to be able to perform (deep) learning on graph-structured inputs as well. This is an area which has only recently started developing, mostly in the form of Graph Neural Networks (GNNs). Since the study of GNNs overlaps quite substantially with the study of distributed graph algorithms, GNNs fit nicely with our other material.

11.1 What are GNNs?

Graph Neural Networks (GNNs) are one of the success stories of machine learning in the past few years: they have achieved remarkable results in various applications, such as chemistry, physical systems, social science or recommendation systems.

Remarks:

- Similarly to our previous algorithms, GNNs operate on graphs. For simplicity, we will assume that these graphs have a *bounded degree*, i.e. Δ is a constant.
- In GNNs, nodes do not have identifiers; instead they have *features*, which describe some properties of the node. We can assume that features are represented as real numbers. Unlike IDs, these features might not be unique.
- To make our examples simpler, we assume that a node only has a single feature. However, in most cases, nodes actually have d different features, so the features of each node are stored as a vector $v \in \mathbb{R}^d$.
- A GNN is similar to our synchronous message passing model (Definition 1.8): the GNN operates in rounds, and every node communicates with their neighbors in each round.

Definition 11.1. *A* Graph Neural Network (GNN) *operates in synchronous rounds. In each round, every node v independently computes a new state; we denote the state of v after time step t by $h_v^{(t)}$. The initial state $h_v^{(0)}$ is the node feature(s) of v.*

The state in time step t is always computed from (i) the node's own state $h_v^{(t-1)}$ in the previous time step, and (ii) the state $h_u^{(t-1)}$ of the nodes' neighbors $u \in N(v)$ in the previous time step. More specifically, GNNs are described in terms of two functions:

$$a_v^{(t)} = \text{AGGREGATE}(\{\{h_u^{(t-1)} \mid u \in N(v)\}\})$$

and

$$h_v^{(t)} = \text{UPDATE}(h_v^{(t-1)}, a_v^{(t)}),$$

where AGGREGATE *is a permutation-invariant function, which can handle any number of inputs.*

Remarks:

- The double brackets $\{\{\cdot\}\}$ denote a *multiset*, where the same element can also appear multiple times.
- Permutation-invariance means that if the same multiset of states was distributed in any other way among the neighbors, then the function would still return the same value.
- In the machine learning literature, the states of nodes are also called embeddings.
- In the context of GNNs, the rounds are also called *layers*. We will denote the number of rounds/layers by r, where r is usually a small constant.
- GNNs can also operate on directed graphs, by only taking in-neighbors into account when aggregating.
- In more sophisticated models, the AGGREGATE and UPDATE functions may also depend on the time step t. However, they cannot depend on the node v, since each node uses the same function (i.e. executes the same program).
- How can we make sure that the AGGREGATE function is permutation-invariant, but still expressive?

Example 11.2. *Aggregation is often implemented as*

$$\text{AGGREGATE}(M) := \mathcal{A}(\{\{f(i) \mid i \in M\}\})$$

where f is some transformation $f : \mathbb{R} \to \mathbb{R}$, and $\mathcal{A}$ is one of the following permutation-invariant functions: MAX, MEAN, SUM. *In the simplest case, the transformation f is implemented as*

$$f(x) := \sigma(w \cdot x),$$

where $w \in \mathbb{R}$ is a learnable weight, and σ is a simple non-linearity.

Remarks:

- Executing a transformation f before $\mathcal{A}$ ensures that the GNN can represent a large class of functions. The non-linearity σ is usually a simple function such as the step function ($\sigma(x) = 0$ for $x < 0, \sigma(x) = 1$ for $x \geq 0$), a ReLU ($\sigma(x) = 0$ for $x < 0$, $\sigma(x) = x$ for $x \geq 0$), or some kind of a sigmoid.
- If our states are in $\mathbb{R}^d$ instead of just $\mathbb{R}$, then the transformation is a function $f : \mathbb{R}^d \to \mathbb{R}^d$, $W \in \mathbb{R}^{d \times d}$ is a matrix, and σ is applied element-wise on each cell of Wx.
- The main idea behind GNNs is that these functions are not all hard-coded. That is, while $\mathcal{A}$ and σ are usually decided in advance, the weight w is *learnt* by the neural network on a large set of data; that is, they are adjusted repeatedly during training, until the output of the GNNs (the final state) is of good enough quality.
- Some applications use a more sophisticated function f; for example, a so-called multi-layer perceptron (MLP), which is essentially a small network formed from a repeated application of the transformation $f(x) = \sigma(w \cdot x)$, with a different choice of weights on different levels. It is known that MLPs can approximate any continuous $\mathbb{R} \to \mathbb{R}$ function, which hints that with such a powerful f, GNNs can compute many different things.
- Let us see an example for a GNN!

Example 11.3. *Let us construct a GNN to decide if a node v is adjacent to a leaf node. Assume for convenience that the initial node features are the degrees: $h_v^{(0)} = \delta(v)$. Then we can easily solve this task by a* 1*-layer GNN that has*

$$\text{AGGREGATE}(N(v)) = \text{MIN}(\{\{h_u^{(t-1)} \mid u \in N(v)\}\})$$

and

$$\text{UPDATE}(h_v^{(t-1)}, a_v^{(t)}) = a_v^{(t)}.$$

This ensures that if $h_v^{(1)} = 1$, then v has an adjacent leaf neighbor, whereas if $h_v^{(1)} \neq 1$, then this is not the case.

Remarks:

- GNNs are similar to our distributed algorithms in the message passing model: for a given number of rounds r, they aggregate messages from their neighbors, do a computation, and then they pass on the new states to their neighbors again. However, there are some key differences.
- Most obviously, GNNs do not have infinite computational power: they can only compute the next node state with the formulas above.
- Furthermore, they do not have the ability to send different, specialized messages to different neighbors: their current node state is essentially their entire view of the world, and they can only pass on this node state to each of their neighbors in each round.

- Another important difference is that nodes do not have unique identifiers in this setting. Hence symmetry breaking and distinguishing specific structures are some of the main challenges.
- Finally, nodes do not have a *port numbering*, i.e. they cannot distinguish their different neighbors. This means that if the neighbor states form a multiset $M = \{\{h_u^{(t)} \mid u \in N(v)\}\}$, then any other permutation of M along the neighbors would also produce the same next state $h_v^{(t+1)}$. In this framework, a node might not even be able to execute simple tasks that were fundamental steps in our distributed algorithms, e.g. a node cannot decide if it received the same message from a specific neighbor in two consecutive rounds.
- So what kind of graph-related questions do we want to answer with a GNN? We know that GNNs compute a final state $h_v^{(r)}$ for each node v. In the simplest case, node labels are already the thing we are looking for in the first place.

Definition 11.4 (Node Regression/Classification)**.** *In a* node regression *task, we want to compute (or approximate) a function $V \to \mathbb{R}$. For example, we want to compute some property of the nodes (represented as a real number or vector). In* node classification*, we want to partition the nodes into different classes.*

Remarks:

- If nodes are people and edges are family relations, we might classify the nodes by their nationality. (Node regression is uncommon, but we might for instance guess their height).
- The final state of the node can be interpreted as the value of this property or the class.
- For more sophisticated classification, we can also apply a function $g(h_v^{(0)}, h_v^{(1)}, \ldots, h_v^{(r)})$ which predicts the final label of v from all the states it had during the different rounds.
- In other applications, however, node states are not even the end of the story.

Definition 11.5 (Graph Classification)**.** *In a* graph classification *task, we want to estimate a function that is dependent on the entire graph; i.e. a function $\mathcal{G} \to \mathbb{R}$, where $\mathcal{G}$ denotes the set of all possible input graphs.*

Remarks:

- For example, given the structure of a specific molecule, we want to estimate some physical or chemical property of the molecule.
- In this case, we can collect the set of states of all of the nodes into a multiset M with $|M| = n$, and learn a so-called READOUT function $\mathcal{M} \to \mathbb{R}$ for the graph classification task, where $\mathcal{M}$ denotes the set of all multisets of size n.

- e.g. if each state $h_u^{(r)}$ describes how faulty node u is, then we could simply compute $\sum_{u \in V} h_u^{(r)}$ as the total "faultiness" of the entire network. For a classification task, we could also introduce a threshold θ, e.g. $\theta = 3$ or $\theta = n/5$, and classify a network as too faulty if $\sum_{u \in V} h_u^{(r)} \geq \theta$.

- For another example, if we compute a state where $h_u^{(r)} = 1$ if node u satisfies some (local) constraint and $h_u^{(r)} = 0$ otherwise, then an aggregation with $\text{AND}_{u \in V}$ allows us to decide if the constraint is fulfilled at every node in the graph.

- Finally, another popular task is link prediction.

Definition 11.6 (Link Prediction)**.** *In a* link prediction *task, there is an original graph G, but we only see a subset of the edges, i.e. we see another graph G' where some of the edges of G are missing. Given a specific pair of nodes u and v, our task is to predict whether the edge (u, v) is present in the original graph G.*

Remarks:

- In this case, we can learn a function $\mathbb{R} \times \mathbb{R} \to [0, 1]$ which takes the states $h_u^{(r)}$ and $h_v^{(r)}$ of u and v as inputs, and outputs the estimated probability of the edge (u, v) being present in the graph.

- So what exactly can GNNs compute, and what not?

- Recall from Example 11.2 that some of the popular permutation-invariant aggregation functions for $\mathcal{A}$ are MAX, MEAN and SUM.

- So which of these three functions is the best choice? It depends on the application. The example below shows that their expressive power is different.

Example 11.7. *In the example graphs of Figure 11.8, consider the state of v after using a GNN with a single layer ($r = 1$).*

The different values for the node feature are shown by colors. For simplicity, assume that the orange feature corresponds to 1*, and the green feature corresponds to* 2*. Also, assume that we execute no transformation before aggregating the values, i.e. $f(x) = x$ in the formula of Example 11.2.*

MAX *aggregation cannot even distinguish the left and middle graphs, since* MAX *returns* 2 *in both cases. However,* MEAN *can distinguish them: it returns* 1.66 *for the first graph, and* 1.5 *for the second.*

However, even MEAN *aggregation cannot distinguish the middle and right examples: it returns* 1.5 *in both cases. On the other hand,* SUM *can distinguish these too: it returns* 3 *for the middle graph, and* 6 *for the right-hand graph.*

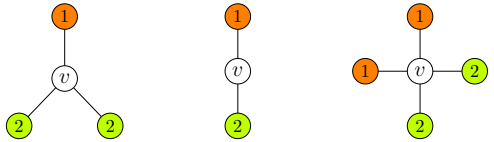


Figure 11.8: An example for the limits of different aggregators.

Remarks:

- Note that the choice of f does not matter for the example above: for two nodes with the same feature $h_v^{(0)}$, the value of $f(h_v^{(0)})$ is the same as well.

- This example suggests that MEAN aggregation is more expressive than MAX, and SUM is more expressive than MEAN.

- In terms of representative power, one can theoretically show that SUM indeed is the strongest one of these options. However, SUM might not always be the simplest choice in practice, since MEAN and MAX have the useful property that the output does not scale with the number of neighbors.

- So let us now revisit our original, general question: How can we describe the functions that GNNs can compute? In order to answer this, we first have to make a brief detour to the Weisfeiler-Lehman algorithm on graphs, also known as the state refinement algorithm.

11.2 WL Algorithm

Algorithm 11.9 State Refinement

$t \leftarrow 0$
for all $v \in V$ **do**
  $s_v^{(0)} \leftarrow 0$
end for
while True **do**
  for all $v \in V$ **do**
    $s_v^{(t+1)} = \text{RELABEL}(s_v^{(t)}, \{\{s_u^{(t)} \mid u \in N(v)\}\})$
  end for
  $t \leftarrow t + 1$
end while

Remarks:

- In the beginning of this algorithm, each node v has the same initial state $s_v^{(0)}$, and these states are refined through multiple rounds.

- Throughout the algorithm, the state of a given node essentially represents its current knowledge about its local neighborhood.

- The RELABEL function is essentially a hash function which assigns a different state to each possible configuration in the neighborhood of the node. More formally, RELABEL is an injective function $\mathbb{R} \times \mathcal{M} \to \mathbb{R}$, where $\mathcal{M}$ denotes all possible finite multisets of $\mathbb{R}$.
- Intuitively speaking, this means that if two nodes u and v have the same state in iteration t, then they will receive a different state in the next time step if and only if there exists a state s such that u and v have a different number of neighbors that are in state s at time step t.
- After the first round of the algorithm, two nodes have the same state if and only if they have the same degree.
- Note that there is no termination condition in the pseudocode: the algorithm runs forever. However, in practice, we are only interested in further refinement steps as long as the algorithm actually makes progress, i.e. it distinguishes more nodes from each other.

Definition 11.10 (WL Termination)**.** *We say that state refinement has* finished *at time step t if there exist no pair of vertices u, v such that $s_u^{(t)} = s_v^{(t)}$, but $s_u^{(t+1)} \neq s_v^{(t+1)}$.*

Theorem 11.11. *Algorithm 11.9 finishes in at most n rounds.*

Proof. If the algorithm has not terminated after round t, then this means that there are two vertices u and v such that $s_u^{(t-1)} = s_v^{(t-1)}$, but $s_u^{(t)} \neq s_v^{(t)}$. On the other hand, if nodes u and v have different states in round t, then they also have different states in all subsequent rounds $t' > t$.

Hence the number of different states that are present in the graph strictly increases in each round. Since this can be at most n, the algorithm finishes after at most n rounds. □

Remarks:

- The state refinement algorithm is also known as the Weisfeiler-Lehman (WL) algorithm. It was originally developed to test the isomorphism of graphs.

Definition 11.12 (Graph Isomorphism)**.** *Two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are* isomorphic *if there exists a bijection $\pi : V_1 \to V_2$ such that for any $u, v \in V_1$, we have $(u, v) \in E_1$ if and only if $(\pi(u), \pi(v)) \in E_2$.*

Remarks:

- That is, if two graphs are isomorphic, then they are "essentially the same", but the nodes are (possibly) presented in a different order.
- Deciding if G_1 and G_2 are isomorphic is a fundamental problem, but it is rather difficult: we do not know how to solve it in polynomial time, even in a centralized setting. Think about a naive algorithm that checks all possible bijections π: this would take $\Omega(n!)$ time.

Algorithm 11.13 Isomorphism testing with the output of Algorithm 11.9

```
M_1 = {{s_v | v ∈ V_1}} and M_2 = {{s_v | v ∈ V_2}}
if M_1 = M_2 then
    return "maybe isomorphic"
else
    return "not isomorphic"
end if
```

- After running the state refinement algorithm on both G_1 and G_2, we can test isomorphism with the following method.

Remarks:

- The equality of multisets can be checked, for example, by sorting both multisets, and then comparing the corresponding elements.
- If these so-called *canonical forms* of two graphs are not equivalent, then the two graphs are certainly not isomorphic.
- However, if the canonical forms are equivalent, then it is still possible that the graphs are not isomorphic, but the test was unable to detect this. In other words, the algorithm is a one-sided isomorphism test.
- Some examples for non-isomorphic graphs that are not distinguished by the Weisfeiler-Lehman algorithm are as follows.

Example 11.14. *The two 8-node graphs in Figure 11.15 are non-isomorphic: one of them consists of two 4-cycles, the other consists of a single 8-cycle.*

Since each node begins with the same state s and each node has exactly two neighbors in state s, RELABEL *assigns the same new state to each node. Hence the algorithm already terminates in the first round on both graphs, and the multisets M_1 and M_2 are identical; the algorithm outputs "maybe isomorphic".*

(a) (b)

Figure 11.15: Two graphs on 8 nodes, consisting of cycles of different length.

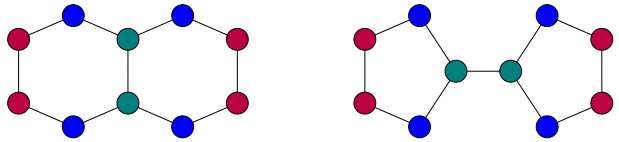
Example 11.16. *The two graphs in Figure 11.17 are again non-isomorphic; for example, one of them has a triangle, but the other does not.*

In the first state refinement step, nodes receive different states based on their degree. In the second step, however, nodes in the same state already all have the same multiset of states in their neighborhoods, so the algorithm terminates. The multiset of states is the same in the two graphs.

(a) (b)

Figure 11.17: Two graphs that cannot be distinguished by the WL algorithm.

Example 11.18. *Figure 11.19 is a slight variation of Figure 11.17, where the process goes on for three iterations instead of two, again outputting "maybe isomorphic". The graphs in this case correspond to two different molecules, Decalin and Bicyclopentyl.*



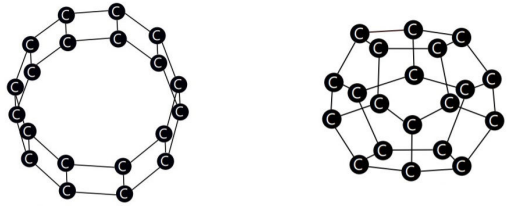
(a) (b)

Figure 11.19: Two graphs that cannot be distinguished by the WL algorithm.

Remarks:

- For the full power of the algorithm, it is important that we compare the actual states in M_1 and M_2, and not just the number of the occurrences of each state. For example, consider a clique on 4 nodes as G_1, and a cycle of length 4 as G_2. Both graphs are regular, so the refinement process will stop after one step in both cases, and both graphs will end up with 4 nodes of identical state. However, this state is different, since nodes in G_1 have degree 3, while nodes in G_2 have degree 2. As such, M_1 and M_2 are different, and the WL-test can distinguish the two graphs.
- In general, if both G_1 and G_2 are k-regular (every node has degree exactly k), then the algorithm cannot distinguish them, since it already terminates in the first iteration.
- We have already seen one example for non-isomorphic regular graphs in Example 11.14: the WL-test cannot distinguish cycles of different length. Another example is below.

Example 11.20. *The molecules Decaprismane and Dodecahedrane both form a 3-regular graph, as shown in Figure 11.21. However, they are not isomorphic: for example, Decaprismane contains a 4-cycle, while Dodecahedrane does not. The WL algorithm cannot distinguish the two graphs.*



(a) Decaprismane. (b) Dodecahedrane.

Figure 11.21: Two graphs that are both 3-regular.

Remarks:

- On the positive side, it was shown that the majority of graphs can indeed be distinguished by the algorithm. That is, if we look at the probability that two random graphs on n nodes can be distinguished by the algorithm, then this probability goes to 1 as n goes to infinity.
- One can also show stronger results for special classes of graphs such as trees.

Remarks:

- But how is the WL algorithm connected to GNNs?
- It turns out that they are closely related. Let us assume that a GNN begins with no information on the nodes, i.e. any two nodes u, v have the same features $h_u^{(0)} = h_v^{(0)}$.

Theorem 11.22 (GNN $\subseteq$ WL)**.** *If two nodes u and v in G have the same state $s_u^{(r)} = s_v^{(r)}$ after r rounds of the state refinement algorithm, then any r-layer GNN on G will compute the same final states for the nodes u and v after r rounds: $h_u^{(r)} = h_v^{(r)}$.*

Proof. We can prove this by induction, showing that the states $h_u^{(t)}$ and $h_v^{(t)}$ are identical after each time step $t \in \{0, \dots, r\}$. For $t = 0$, this is straightforward, since all nodes begin with the same feature.

For a general t, assume that the statement is already proven for $t-1$. Assume that u and v still have the same state in round t; due to the state refinement algorithm, this implies that they had the same state $s_u^{(t-1)} = s_v^{(t-1)}$ at time $t-1$, and the same multiset of states in their neighborhood: $\{\{s_{u'}^{(t-1)} \mid u' \in N(u)\}\} = \{\{s_{v'}^{(t-1)} \mid v' \in N(v)\}\}$. Due to the induction hypothesis, this means that $h_u^{(t-1)} = h_v^{(t-1)}$ and $\{\{h_{u'}^{(t-1)} \mid u' \in N(u)\}\} = \{\{h_{v'}^{(t-1)} \mid v' \in N(v)\}\}$ in our GNN. However, then according to the formulas in Definition 11.1, the nodes will also have the same state $h_u^{(t)} = h_v^{(t)}$. □

Remarks:

- This shows that the expressive power of GNNs is *at most as high* as that of the WL-test: if the WL algorithm fails two distinguish two graphs, then a GNN will also produce the same result on these graphs.
- For example, for several applications, it is important to know whether a node is contained in a triangle. However, as Example 11.16 shows, GNNs are unable to decide this from the final state of a node: the blue nodes in the left graph are contained in a triangle, while the blue nodes in the right graph are not. According to Theorem 11.22, these blue nodes will end up with the same final state in the two graphs.
- So what about the other direction: are GNNs indeed as powerful as the WL-test? Yes, it turns out!

Theorem 11.23 (WL $\subseteq$ "good" GNN)**.** *There exists an r-layer GNN that fulfills the following property: if two nodes u and v are in different states $s_u^{(r)} \neq s_v^{(r)}$ after r rounds of state refinement, then the GNN assigns different final states $h_u^{(r)} \neq h_v^{(r)}$ to u and v.*

Proof. We only outline the main idea of the proof here, which is quite natural: if both AGGREGATE and UPDATE are injective on their respective domains, then whenever two nodes are in different states in the state refinement procedure, then they are also in different states in the GNN. In particular, if this is the case, then, through another induction, one can show that if the final states of u and v are different in the state refinement procedure, then also $h_u^{(r)} \neq h_v^{(r)}$. It remains to show how to construct injective AGGREGATE and UPDATE functions.

To construct such functions, we first need to make a technical assumption. In particular, we assume that the initial (input) features are positive integers. Note how this is not too constraining, since the set of all possible input feature values is countable, as they can be represented inside computer memory. These being said, we will construct our GNN to only ever operate on positive integers during all rounds.

Now, to construct an injective AGGREGATE function, first note that the degree of the graph is bounded by Δ, meaning that AGGREGATE takes multisets of cardinality at most Δ as its only argument. Then, let b be any integer such that $b > \Delta$ and define $\text{AGGREGATE}(M) = \sum_{x \in M} b^x$. This ensures that AGGREGATE is indeed injective: for any multiset M, the value $\text{AGGREGATE}(M)$ uniquely encodes M as a number in base b, where each position corresponds to a positive integer, and the base-b digit at that position gives the multiplicity of this integer in M. Another way to construct AGGREGATE that does not rely on Δ being bounded would be to take $\text{AGGREGATE}(M) = \prod_{i \in Z^+} p_i^{M(i)}$, where p_i denotes the i-th prime number and $M(i)$ denotes the multiplicity of i in multiset M.

An injective UPDATE function can be created in several ways, of which perhaps the simplest is to define it as $2^{h_v^{(t-1)}} \cdot 3^{a_v^{(t)}}$. A specific value $h_v^{(t)}$ then uniquely determines the corresponding $h_v^{(t-1)}$ and $a_v^{(t)}$.

□

Remarks:

- This means that the most powerful GNNs we can design are exactly as powerful as Algorithm 11.9!
- Intuitively speaking, this also shows that whenever we can describe an algorithm in the corresponding message passing model (constantly many rounds, not distinguishing a node's neighbors, no node IDs), then a GNN can indeed express the function computed by the algorithm. This allows us to discuss the expressiveness of GNNs on a higher abstraction level, and ignore the details of the actual implementation. Of course, this only describes the theoretical capabilities of a GNN — whether a GNN in practice can indeed *learn* such complicated functions is another question.
- The WL-algorithm is in fact a hierarchy of isomorphism-heuristics; what we discussed here is the 1-WL algorithm, but there are also more sophisticated variants called k-WL (for each $k \in \mathbb{Z}^+$). However, these higher-order WL algorithms require at least $\Omega(n^k)$ space and even higher time complexity, so they are often not applicable in practice.

11.3 Over and Under

Apart from WL, GNNs also suffer from other fundamental problems.

Definition 11.24 (Oversmoothing)**.** *Oversmoothing is a GNN problem where information quickly washes out because nodes aggregate all their neighbor information.*

Remarks:

- A problem that quickly emerged with GNNs is that we cannot have many GNN layers. Each layer averages and hence smoothens out the neighborhood information and the node's features. This effect leads to features converging after some layers, which is known as the oversmoothing problem.
- Several works address the oversmoothing problem, for example by sampling nodes and edges to use in message passing, leveraging skip connections, additional regularization terms, or by adopting an asynchronous model of communication.

Definition 11.25 (Underreaching)**.** *Using normal GNN layers, a GNN with r layers can only learn about nodes at most r hops away. A node cannot act correctly if it would need information that is $r + 1$ hops away.*

Remarks:

- In other words, underreaching is closely related to locality, i.e., that a distributed algorithm needs more rounds to gather all the necessary information. This is one of the main themes in distributed computing.

- There exist counter measures, for example, having a global exchange of features, or spreading information using diffusion processes.
- Methods that help against oversmoothing are usually also applied against underreaching, since we can use more layers and increase the neighborhood size.

Definition 11.26 (Oversquashing)**.** *In many graphs, the size of k-hop neighborhoods grows substantially with k. This requires squashing more and more information into a node state. Eventually this leads to the congestion problem (too much information having to pass through a bottleneck). This problem is known as oversquashing.*

Remarks:

- In distributed computing, oversquashing has a lot to do with congestion. We studied congestion in the context of global problems, i.e., in Chapter 6.
- One approach to solve oversquashing is introducing additional edges that function as shortcuts to non-direct neighbors.

11.4 More Expressive GNNs

The examples in Figures 11.15–11.19 show that some neighborhoods cannot be distinguished by the WL algorithm, and hence also not by GNNs. This raises a natural question: how can we make GNNs more expressive?

Remarks:

- One natural approach is to introduce port numbers into our model; this takes the setting closer to classical distributed algorithms.

Definition 11.27. *In a* GNN with port numbers*, each node v numbers its incident edges from* 1 *to $\delta(v)$, and the domain of* AGGREGATE *is not a multiset as before, but a vector $\mathbb{R}^\Delta$, which allows the result to depend on the different neighbors in a different way.*

More formally, the domain of the AGGREGATE *function is $\hat{\mathbb{R}}^\Delta$, where $\hat{\mathbb{R}} = \mathbb{R} \cup \{\bot\}$, and $\bot$ is a special symbol denoting that the node does not have a neighbor with the given port number.*

Remarks:

- Of course, there are multiple ways a node can number its incident edges. In these GNNs, the final state also depends on the port numbers assigned to the edges: for a different assignment of port numbers in the same graph, the nodes might compute a different state!
- Port numbers are still not always enough to distinguish some of our examples.

Theorem 11.28. *GNNs with port numbers are still unable to distinguish some of the previous examples in case of an unlucky port numbering.*

Proof. Consider Figure 11.29, which is an extension of the graph in Figure 11.15 with port numbers and some node features (colors). However, port numbers and features are chosen such that the two graphs still cannot be distinguished by a GNN. □

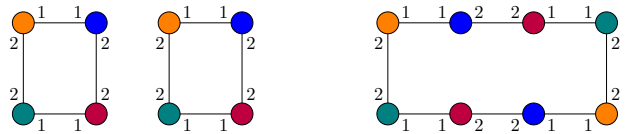


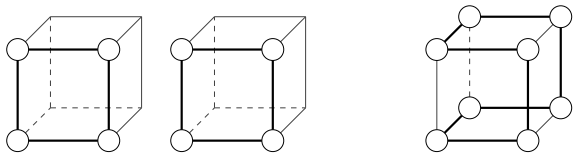
Figure 11.29: Graphs that cannot be distinguished even with port numbers. (Note that the node colors are not really necessary.)

Remarks:

- A more practice-oriented approach is to apply more insights from the given application area, e.g. by adding more node features which help us distinguish the different cases. For example, in molecule modeling, one possibility is to also measure the angles of the different edges between the graphs, and use these as an extra features.
- Note that this raises some difficult representation questions about how we store/use these angles: we either have to tie these features to pairs of edges, or we have to come up with a structured way to store all the $\binom{\delta(v)}{2}$ angles at each node v. For simplicity, we will only consider some specific graphs now where each node has degree two; this means that each node only needs to store exactly 1 angle parameter, i.e. a single extra feature.
- However, given an unlucky geometric embedding, there are graphs that still cannot be distinguished with this method.

Theorem 11.30. *Even if we add the angles of edges as extra features, there are non-isomorphic graphs G_1, G_2 that cannot be distinguished by GNNs.*

Proof. Consider the graphs in Figure 11.31, which are embeddings of the (non-isomorphic) graphs in Figure 11.15 in 3-dimensional space. For all nodes in both graphs, the two incident edges have an angle of 90° between them, so in any representation, all nodes begin with the same feature. This means that the upper bound in Theorem 11.22 still applies, and hence a GNN cannot distinguish the two graphs. □



(a) (b)

Figure 11.31: Two graphs that cannot be distinguished even with angle features.

Remarks:

- Another approach on the more theoretical side: let us do symmetry breaking by introducing random features to each node! That is, we increase the dimension d of the feature vector by 1, and into this final slot of the vector, we insert a uniform random integer in $\{1, \dots, L\}$ (for some constant L), which is chosen independently for each node.

Definition 11.32. *In a* GNN with random features*, we assign a new random feature to each node, i.e. an integer chosen uniformly at random from* $\{1, \dots, L\}$ *(for some constant limit* L*), independently from other nodes.*

Remarks:

- With this approach, an algorithm can now recognize that two nodes observed on two different paths actually correspond to the same node. That is, if the random features are more-or-less unique in the neighborhood (which becomes probable if the neighborhood has bounded size and L is large enough), then we can essentially use them as pseudoIDs, and we obtain a model that is again quite close to classical distributed algorithms.

Example 11.33. *Random IDs already allow us to separate cycles of different length. Consider a graph* G_1 *that consists of* 2 *disjoint* 3*-cycles, versus a graph* G_2 *which consists of a single* 6*-cycle, as in Figure 11.34. Both in* G_1 *and* G_2*, a standard GNN with* 3*-layers will observe the same structure.*

However, let us now add a random feature to each node, and assume that L *is high enough such that all the six nodes receive a different pseudoID with a decent probability. In this case, a node in* G_1 *will always see a node with the same pseudoID in a* 3*-hop distance, whereas in* G_2*, this will not happen when the pseudoIDs are unique. Based on this information, a sufficiently powerful GNN can distinguish the two graphs.*

Remarks:

- For this GNN variant, one can prove an even stronger result: in theory, it can distinguish any two distinct neighborhoods.

Theorem 11.35. *For any set of bounded-degree local neighborhoods* S*, there exists a GNN with weights such that for each node* v*,*

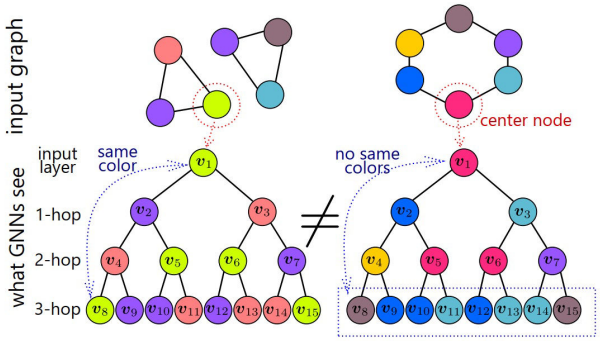


Figure 11.34: Distinguishing cycles of different length with random features.

- *if the* r*-hop neighborhood of* v *is isomorphic to a graph in* S*, then* $h_v^{(r)} > 0.5$ *with high probability;*
- *if the* r*-hop neighborhood of* v *is not isomorphic to any graph in* S*, then* $h_v^{(r)} < 0.5$ *with high probability.*

Remarks:

- However, while this sounds convincing in theory, the situation is not so simple in practice. If we train a GNN with random features, it often learns to distinguish different structures by learning how the different random features relate to each other. If we apply these learnt functions later in a situation where the random features happen to have a different relation to each other, then our learnt functions might return something that is not useful. Therefore, while random features often yield good results on the training data, the approach might not generalize so well to new test data.
- Another method of introducing randomization is to use dropouts.

Definition 11.36. *A* GNN with dropouts *executes the same* r*-round computation multiple times; each time is called a* run*. In each run, every node* v *is "dropped out" independently with a fixed probability* p*, which means that both* v *and all of its incident edges are removed from the graph. The GNN is then simulated in this smaller graph* G' *(remaining after dropouts) in the given run.*

The final state of v *in the* i^{th} *run is denoted by* $h_v^{(r)[i]}$*. After the* R *runs, each node* v *computes a final state* $\hat{h_v}$ *by aggregating its final states in these* R *runs through*

$$\hat{h_v} = \text{COMBINE}(\{\{h_v^{(r)[1]}, \dots, h_v^{(r)[R]}\}\},$$

where COMBINE *is a permutation-invariant function.*

Remarks:

- The GNN now executes the same computation multiple times, but on slightly different variants of the original graph where some nodes are always removed randomly.
- The COMBINE function has to be permutation-invariant because there is no ordering between the different runs: a given final state $h_v^{(r)[i]}$ is obtained in each run $i \in \{1, \dots, R\}$ with the same probability.
- But what if the final state $h_v^{(r)[i]}$ of node v if v is removed in the i^{th} round? This is just a technical question. We can conveniently assume that $h_v^{(r)[i]} = 0$ in this case, or that $h_v^{(r)[i]}$ is then left out from the multiset $\{\{h_v^{(r)[1]}, \dots, h_v^{(r)[R]}\}\}$ altogether.
- The main idea is that even if two neighborhoods cannot be distinguished by the WL-test, the modified neighborhoods (when some nodes are removed) are maybe still distinguishable.

Example 11.37. *Recall that the graphs in Figure 11.15 cannot be distinguished by standard GNNs.*

However, consider a GNN with dropouts, $r = 2$ layers, and a simple choice of $a_v^{(t)} = \sum_{u \in N(v)} h_u^{(t-1)}$ and $h_v^{(t)} = h_v^{(t-1)} + a_v^{(t)}$. For example if all nodes start with $h_v^{(0)} = 1$, and if no node is removed in a run, then each node ends up with a final state of $h_v^{(2)} = 9$.

Now consider the probability of having a final state $h_v^{(r)} = 7$ in a run for any of the nodes v (in a run where v itself is not removed). In the left graph of Figure 11.15, this can already happen if the node at distance 2 from v is removed. Assume that nodes are removed with a small probability p, so this roughly happens with probability $p(1-p)^2$ in each run. However, in the 8-cycle, this only happens if both of the nodes at distance 2 from v are removed, which happens with probability $p^2(1-p)^2$. As such, the COMBINE *function can separate these cases based on the frequency of the value 7 in the multiset (with a decent probability).*

Example 11.38. *Recall that the middle and right graphs in Figure 11.8 cannot be distinguished with $\mathcal{A} =$ MEAN in case of $r = 1$.*

However, in case of dropouts, the right graph can produce a final $h_v^{(1)}$ value of 1.33 or 1.66 in case a neighbor of v is dropped out. These final states can never occur in the middle graph; as such, if the multiset $\{\{h_v^{(r)[1]}, \dots, h_v^{(r)[R]}\}\}$ contains either 1.33 or 1.66, then we know that v has the right-hand neighborhood instead of the one in the middle.

Remarks:

- In case of dropouts, v essentially observes a probability distribution of final states. If two such probability distributions are different, then a sophisticated COMBINE function can separate the two cases (if the number of runs R is high enough).
- Of course, this only holds if the number of runs R is large enough. But how large does R have to be?

- Note that, asymptotically speaking, if $p, \Delta, r \in \mathcal{O}(1)$, then the maximal size of the r-hop neighborhood is $\mathcal{O}(\Delta^r)$, and the number of possible dropout patterns is $2^{\mathcal{O}(\Delta^r)} = \mathcal{O}(1)$, each happening with a constant probability. However, ensuring that we observe all such configurations is usually not viable in practice.
- One simpler objective is to observe all the possible 1-dropouts.

Definition 11.39. *Given the r-hop neighborhood $N^r(v)$ of v, we say that a specific run is a* 1-dropout *if exactly 1 node is removed from $N^r(v)$, and v is not removed.*

Theorem 11.40. *Let $n_0 := |N^r(v)|$ for a specific node v. In order to maximize the probability of any 1-dropout for v, we should select $p = \frac{1}{n_0}$.*

Proof. The probability of any 1-dropout is $p \cdot (1-p)^{n_0-1}$. Differentiation shows that on $p \in [0, 1]$, this probability is maximized if $p = \frac{1}{n_0}$. □

Remarks:

- Note that different nodes have a different r-hop neighborhood size (i.e. different n_0), but p is a global parameter. However, if the neighborhood sizes are not so different, then this is not a problem in practice.
- How many runs does it take to ensure that we observe all the possible 1-dropouts?

Theorem 11.41. *Let $p = \frac{1}{n_0}$. Then with $R = \mathcal{O}(n_0 \log n_0)$ runs, v observes all the possible 1-dropouts in $N^r(v)$ with high probability.*

Proof. Let $R = c_0 \cdot n_0 \cdot \log n_0$ for some constant c_0. In R runs, the expected frequency μ of observing a specific 1-dropout is

$$\mu = p \cdot (1-p)^{n_0-1} \cdot R = \frac{1}{n_0} \cdot \left(1 - \frac{1}{n_0}\right)^{n_0-1} \cdot c_0 \cdot n_0 \cdot \log n_0 \geq \frac{1}{e} \cdot c_0 \cdot \log n_0.$$

This also shows that $\mu \geq 2$ as soon as $c_0 \geq 2e$. A Chernoff bound shows that the probability of the fact that this frequency is below $(1 - \frac{1}{2})\mu$ is at most

$$2e^{-\frac{1}{2^2} \cdot \frac{1}{3} \cdot \mu} \leq 2e^{-\frac{1}{12} \cdot \frac{c_0}{e} \cdot \log n_0} = 2n_0^{-\frac{c_0}{12e}}.$$

Note that this already ensures that we observe this 1-dropout at least once, since $(1 - \frac{1}{2})\mu \geq 1$ due to $\mu \geq 2$. If we select $c_0 \geq 24e$, then this probability of not observing a specific 1-dropout is below $2n_0^{-2}$.

Since there are $n_0 - 1$ different 1-dropouts in $N^r(v)$, we can take a union bound over these. The probability that *any* 1-dropout remains unobserved is at most $2n_0^{-1}$. □

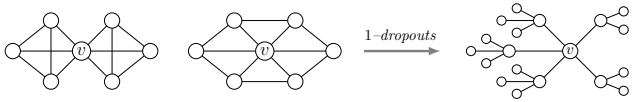


Figure 11.43: Example of two graphs that are not separable by 1-dropouts (left side). In both of the graphs, for any of the 1-dropouts, v observes the same tree structure for $r = 2$, shown on the right side.

Remarks:

- Requiring $\mathcal{O}(n_0 \log n_0)$ distinct runs is not too bad from a theoretical perspective. However, this might already be too high to be feasible in practice.
- Note that with dropouts, we not only have to execute multiple runs while training the GNN, but also during testing, i.e. every time when we want to apply the GNN on a new graph.
- Are 1-dropouts already enough to distinguish any two distinct r-hop neighborhoods? Unfortunately not.

Theorem 11.42. *There is a pair of non-isomorphic neighborhoods that cannot be distinguished based on the* 1*-dropouts.*

Proof. Consider the left and middle graphs in Figure 11.43, obtained by connecting v to each node in two 3-cycles or a 6-cycle, respectively, and assume that $r = 2$. Note that in both cases, v can observe all the nodes and edges of the graph in 2 hops; a classical message passing algorithm could easily distinguish these cases.

However, a GNN without dropouts always computes the same state in both graphs. Furthermore, in case of any of the 6 possible 1-dropouts (in either of the graphs), v also observes the same tree structure (shown on the right side of the figure), so it also computes the same $h_v^{(2)}$. As such, the two graphs cannot be distinguished from the 1-dropouts. □

Chapter Notes

GNNs have first been developed in [GMS05, SGT+09], and have been the subject of intensive study in the last few years [Ham20]. They have also been successfully applied in a wide range of different application areas [GSR+17, YHC+18, PKD+19].

The Weisfeiler-Lehman algorithm is a graph isomorphism heuristic that has been studied for decades [CFI92, GKMS14, WL68]. It is in fact a hierarchy of isomorphism-heuristics with increasing power: for each k, there are non-isomorphic graphs that cannot be distinguished by the so-called k-WL algorithm, but they can be distinguished by $(k+1)$-WL [CFI92]. The WL algorithm is also one of the main ingredients of Babai's celebrated result on the complexity of the isomorphism problem [Bab16].

The fact that GNNs are as exactly as expressive as the WL-test has first been shown by [XHLJ19]. Since then, various GNN extensions have been suggested in order to overcome this limitation. GNNs with port numbers have been introduced by Sato and others [SYK19], who show that they are essentially as powerful as the so-called anonymous model of distributed computing. GNNs with angle parameters are studied in [KGG20]. GNNs equipped with random features was studied in [SYK21], which is also the source of our Figure 11.34. GNNs with dropouts was introduced and studied in [PMFW21]. GNN variants with the expressive power of higher-dimensional WL algorithms have also been built in [MBHSL19]; however, these are often not viable in practice due to higher space and time complexity. Other works, such as [GJJ20], shown examples for graphs that cannot be distinguished even by these more powerful GNN variants. A survey on different GNN variants is also available in [Sat20], which is also the source of Figure 11.21. A theoretical comparisons between different GNN variants revealed that strength of expressivity is depending on the problem [PW22]. Oversmoothing was for instance studied by [OS20]. Underreaching was named by [BKM+20]. And oversquashing was brought to the GNN context by [AY21].

This chapter was written in collaboration with Pál András Papp.

Bibliography

[AY21] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations (ICLR)*, 2021.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing (STOC)*, page 684–697, New York, NY, USA, 2016. ACM.

[BKM+20] Pablo Barceló, Egor Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan-Pablo Silva. The logical expressiveness of graph neural networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[CFI92] Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

[GJJ20] Vikas Garg, Stefanie Jegelka, and Tommi Jaakkola. Generalization and representational limits of graph neural networks. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119, pages 3419–3430. PMLR, 2020.

[GKMS14] Martin Grohe, Kristian Kersting, Martin Mladenov, and Erkal Selman. Dimension reduction via colour refinement. In *Algorithms - ESA 2014*, pages 505–516. Springer Berlin Heidelberg, 2014.

[GMS05] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, pages 729–734 vol. 2, 2005.

[GSR+17] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1263–1272, 2017.

[Ham20] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[KGG20] Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. In *8th International Conference on Learning Representations (ICLR)*, 2020.

[MBHSL19] Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32. Curran Associates, Inc., 2019.

[OS20] Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. In *International Conference on Learning Representations (ICLR)*, 2020.

[PKD+19] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. Estimating node importance in knowledge graphs using graph neural networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, page 596–606. Association for Computing Machinery, 2019.

[PMFW21] Pál András Papp, Karolis Martinkus, Lukas Faber, and Roger Wattenhofer. Dropgnn: Random dropouts increase the expressiveness of graph neural networks. In *35th Conference on Neural Information Processing Systems*. Curran Associates, Inc., 2021.

[PW22] Pál András Papp and Roger Wattenhofer. A Theoretical Comparison of Graph Neural Network Extensions. In *39th International Conference on Machine Learning (ICML), Baltimore, Maryland, USA*, July 2022.

[Sat20] Ryoma Sato. A survey on the expressive power of graph neural networks, 2020. arXiv preprint 2003.04078.

[SGT+09] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, pages 61–80, 2009.

[SYK19] Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Approximation ratios of graph neural networks for combinatorial problems. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32. Curran Associates, Inc., 2019.

[SYK21] Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Random features strengthen graph neural networks. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*, pages 333–341, 2021.

[WL68] Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *Nauchno-Technicheskaya Informatsi series*, 2(9):12–16, 1968.

[XHLJ19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *7th International Conference on Learning Representations (ICLR)*, 2019.

[YHC+18] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, page 974–983. Association for Computing Machinery, 2018.