Pretraining Graph State Encoders for microRTS using Graph Self-Supervised Learning

Pavan Kantharaju[1]

[1]Smart Information Flow Technologies, 319 1st Ave North, Suite 400, Minneapolis, MN 55401-1689

Abstract

Real-Time Strategy (RTS) games are a popular and successful genre of video games that also doubles as a research environment for evaluating challenging AI problems. μRTS is one such research environment that represents many of the AI research problems inherent in commercial RTS games in a low graphics fidelity grid-based game. Prior work studied the use of graph game states for Deep Reinforcement Learning (DRL)-based game-playing in μRTS. However, the state encoder used in the DRL agent was specifically trained for the task of DRL. Ideally, we want to train a single state encoder that can be transferred to a variety of downstream tasks (e.g., player modeling, game-playing, and content generation) for μRTS to minimize the costly training of state encoders for each possible task. Additionally, the state encoder used a Gated Recurrent Unit for processing graphs over Graph Neural Networks (GNNs) designed to process graphs. This paper provides an initial study on pretraining a transferrable GNN-based graph state encoder using a variant of the self-supervised learning algorithm Distillation with No Labels (DINO) for graph representation learning over a set of μRTS game replays. We provide a qualitative analysis of the latent graph states through a cluster analysis and begin to evaluate the transferability of the latent state representations starting with the task of action prediction. We show that the latent states contain information about μRTS game maps despite not being explicitly trained on spatial map features, and associations that hint at particular types of player behaviors. We also show that the latent states can be fine-tuned for action prediction.

Keywords

Real-Time Strategy Games, Self-Supervised Learning, Representation Learning, Graph Neural Networks

1. Introduction

Real-Time Strategy (RTS) games such as Starcraft and Age of Empires are a popular and successful genre of video game, which are also valuable for evaluating solutions to AI research problems. For example, RTS games have been used for evaluating methods in planning [1, 2, 3], plan and goal recognition [4, 5, 6], and reinforcement learning [7, 8, 9].

A core component of any RTS game is its game state, which is a snapshot of the game world at some given time. This game state includes characteristics about the player and their opponents, resources on the current game's map, and the terrain of the map. AI methods, particularly machine/deep learning methods, require a latent representation of the game state that encodes salient factors from the state, which allows them to perform their task. This representation is usually created using a neural encoder, which takes the game state and projects it into a latent space. Prior work by Jin [10] showed that graphs can be used as a game state representation that transfers across game maps from the RTS game μRTS [11]. A graph is a useful way to model game states in RTS games as graphs can represent relational and relative spatial information across units on a game map. Jin [10] then used a Gated Recurrent Unit (GRU) [12] encoder, contained within a Deep Reinforcement Learning (DRL) agent, to project the graph state into a latent space. However, the GRU encoder was specifically trained for the task of DRL-based game-playing, which limits its generality to other downstream tasks for RTS games. Additionally, GRUs are designed to work over sequential data (e.g., text [12]) instead of graph data, while Graph Neural Networks (GNNs) are designed to process graphs (e.g., knowledge graphs [13]).

Joint AIIDE Workshop on Experimental Artificial Intelligence in Games and Intelligent Narrative Technologies, November 10-11, 2025, Edmonton, Canada.

pkantharaju@sift.net (P. Kantharaju)

0000-0002-7599-8499 (P. Kantharaju)

This paper presents an initial study on pretraining a transferrable GNN graph state encoder over a set of μRTS game replays.[1] More specifically, we contribute a pretrained graph state encoder trained using a variant of the self-supervised learning (SSL) algorithm Distillation with No Labels (DINO) [14], used in visual representation learning, for learning latent graph game state representations from game replays. We provide a qualitative analysis of the pretrained latent graph states through a cluster analysis and a quantitative evaluation of the latent states for the task of action prediction. Our results show that the latent graph states contain information about μRTS game maps despite not being explicitly trained on spatial map features, and associations that hint at particular types of player behaviors. Our results also show that these latent states can be successfully transferred to the task of action prediction, providing a starting point for conducting evaluations across different RTS game tasks in future work.

This paper is structured as follows. First, we contextualize our work with respect to the literature. Next, we provide a brief background on μRTS, DINO, and the specific GNN used in this paper, Graph Attention Network V2 (GATv2) [15]. Next, we describe the μRTS graph game state and how we pretrain a GNN state encoder over these game states. We then detail our experiments, provide initial results on the pretrained GNN state encoder, and discuss the results. Finally, we conclude with next steps.

2. Related Work

Self-Supervised Graph Learning: There are three major categories of self-supervised learning over graphs. *Generative learning* [16] trains models to learn features that reconstruct its input data distribution. *Joint-embedding contrastive learning* [17] trains models to align positive (similar) inputs close together in latent space while pushing negative (dissimilar) inputs away from each other using *explicit* positive-positive and positive-negative input pairs. However, joint-embedding contrastive learning methods tend to be computationally expensive as they require a large number of positive-negative input pairs to perform well. *Joint-embedding non-contrastive learning* [18, 19, 20] addresses this limitation by training models to elicit the same behavior by only using positive-positive input pairs. The closest method to our approach, GraphDINO [19], falls under *joint-embedding non-contrastive learning*. GraphDINO is an SSL method that trains a modified Transformer [21] backbone using DINO [14] for learning representations of 3D neuronal morphologies. DINO trains models to place similar inputs close together in latent space by perturbing inputs and aligning perturbed data points generated from the same input in latent space. GraphDINO adapts DINO to spatial graph data structures by introducing stochastic graph perturbations relevant to spatial neuronal graphs and modifying the Transformer architecture for graph data while keeping the DINO learning algorithm unchanged. Our work also leaves the DINO algorithm unchanged, but, instead, uses a GNN backbone and graph perturbations relevant to game states, and is applied to learning game states for RTS games.

Bootstrapped Graph Latents (BGRL) [18] and Graph Barlow Twins (GBT) [20] are two other methods relevant to our work that falls under the *joint-embedding non-contrastive learning* category. BGRL, GBT, and GraphDINO conduct SSL by comparing outputs from two neural networks, but these methods differ in their training paradigms. GraphDINO and BGRL are based on the knowledge distillation learning paradigm, where a teacher network distills its knowledge into a student network. However, GraphDINO uses symmetric architectures for both student and teacher while BGRL uses asymmetric architectures. GBT is based on the redundancy-reduction principle [20], where models are trained to reduce the redundancy in their representations. Similar to GraphDINO, GBT uses symmetric architectures for its two networks, but GBT additionally uses the same model weights for the networks.

Game State Self-Supervised Learning: The goal of SSL for games is to construct general and informative representations of game states for AI game tasks, such as game-playing, content generation, and player modeling. Many of the SSL methods studied in the literature are contrastive-based methods. Trivedi et al. [22] provided a study of off-the-shelf SSL methods SimCLR, SwAV, and BYOL; to the best of our knowledge, BYOL is currently the only non-contrastive SSL method that has been studied.

[1] Code is available at https://github.com/Teravolt/microrts-graph-state-encoder-gssl

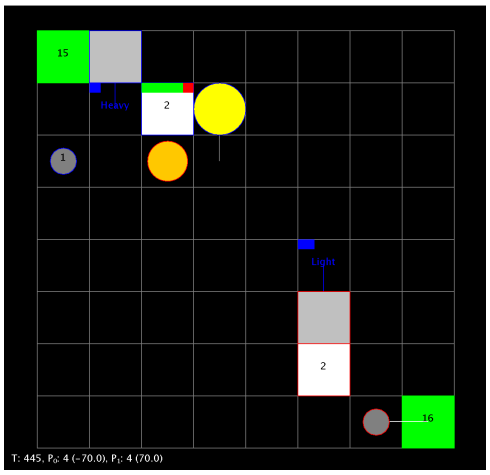

Figure 1: Screenshot of μRTS gameplay between two scripted game-playing agents

GameCLR [23] uses the SimCLR over images to learn game state representations while Trivedi et al. [24] uses supervised contrastive learning over labeled images. Our work differs in that (1) our work focuses on graph game states over image game states, and (2) we study learning representations for RTS games. The closest approach to our work is Knowledge-Enhanced Graph Contrastive Learning (KEGC) [25], which uses contrastive learning over graphs to address the task of game outcome prediction in Multiplayer Online Battle Arena games; our work instead uses non-contrastive SSL for RTS games.

AI in Real-Time Strategy Games: RTS games have been extensively used in prior research as a way to evaluate challenging AI research problems. A variety of different methods have been used for addressing AI tasks in RTS games, including planning [1, 3], plan and goal recognition [4, 6], and recently, reinforcement learning [7, 9]. Our pretrained graph state encoder can complement many of these methods and is one avenue for future work. The closest research to our work was done by Jin [10], who studied the combination of Convolutional Neural Networks (CNNs) and GRUs for extracting state information for DRL game-playing in μRTS. The CNNs were used to extract spatial information while the GRUs were used to extract relational information from graphs. Our work differs in that our graph state representations contains spatial information within the edge features instead of using a CNN. We also evaluate our GNN state encoder over the task of action prediction (vs DRL) and study how to pretrain a GNN over a large-scale dataset of μRTS game replays.

3. Background

This section describes the μRTS testbed and provides a brief background on GATv2 and DINO.

3.1. microRTS

μRTS is a minimalistic RTS game designed to evaluate AI research in an RTS setting [11, 3]. Figure 1 shows two scripted agents playing against each other on a game map represented as an $N \times N$ discrete grid. Despite its minimalism and low graphics fidelity, μRTS retains the properties of commercial RTS games, such as *StarCraft*, that make them complex from an AI point of view, such as durative and simultaneous action execution, real-time decision making, large state spaces, and full or partial observability. Since 2017, IEEE CoG (previously CIG) has hosted the μRTS competition to foster AI research in game-playing agents for RTS games.[2] This paper makes use of deterministic and fully-observable game replay data from the COG 2019 and 2020 μRTS competitions.

[2] https://sites.google.com/site/micrortsaicompetition/introduction?authuser=0

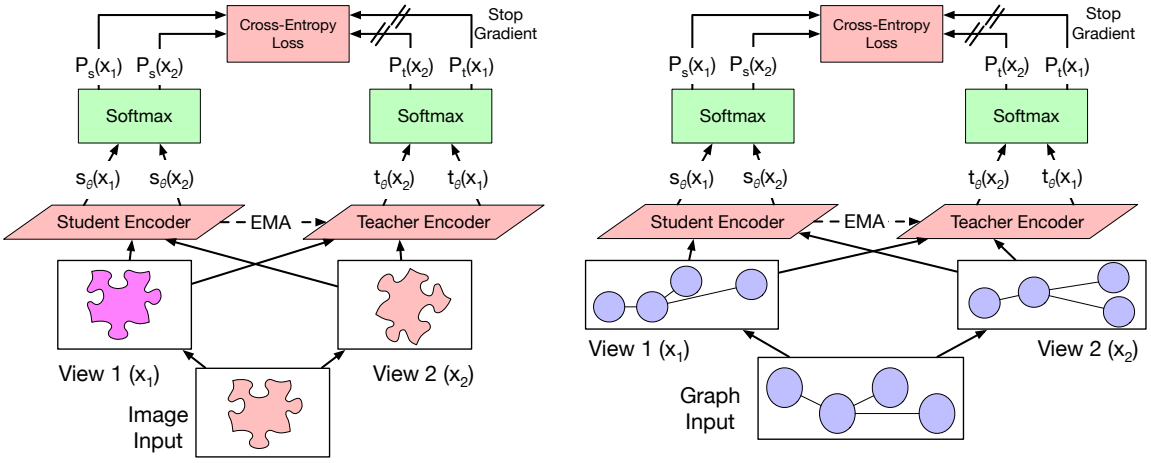

Figure 2: Illustration of DINO for image (left) and graph (right) data structures

μRTS has six core entities, specifically *light*, *heavy*, *ranged*, *barracks*, *base*, *worker*, and *resources*, the first six of which can be controlled by players. Resources are currency in μRTS, which are used to construct entities, and their amount and locations are predefined by a game map. The first three entities (light, heavy, ranged) are considered offensive units whose purpose is to attack enemy units, and are created using barracks. Bases are used to construct worker units that harvest resources, and build bases and barracks. This makes worker entities the backbone of μRTS gameplay, as they are indirectly or directly responsible constructing and interacting with the other core entities in the game.

The μRTS game state is an $N \times N$ or $N \times M$ grid-based game map containing the above entities, information pertaining to the entities (e.g., health, remaining resources, etc.), and terrain information. μRTS has six actions that can be done by player-controlled units: *idle*, *move*, *harvest*, *produce*, *return*, and *attack*. Each action is restricted to certain units, except the idle action, which can be done by all units. Specifically, the move and attack actions can only be done by worker, light, heavy, and ranged units. The harvest and return actions can only be done by worker units while produce can be done by worker, base, and barracks.

3.2. Distillation with No Labels

Distillation with **no** labels (DINO) is an SSL method from computer vision that trains vision neural networks to learn dense latent representations of images. Figure 2 (left) provides an illustration of DINO over images. DINO is based on the knowledge distillation learning paradigm, where a *Teacher Encoder distills knowledge* into a *Student Encoder*. More specifically, DINO uses representations from a previous training checkpoint of the model (Teacher Encoder) as the "label" for training the current version of model (Student Encoder). DINO takes an image x from the current batch and constructs two new images (known as views), x_1 and x_2, based on a set of domain-general perturbations (e.g., random crop, gaussian blur, etc). These views are provided to student and teacher siamese networks, s_θ and t_θ, to construct K-dimensional latent representations of the views, denoted by $s_\theta(x_j)$ and $t_\theta(x_j)$, where $j \in \{1, 2\}$ is the index of the image view. These representations are then passed through a temperature-scaled softmax to convert the representations into a K-dimensional probability distribution,

$$P_{z,i}(x_j) = \frac{e^{z_{\theta,i}(x_j)/\tau_z}}{\sum_{k=0}^{K} e^{z_{\theta,k}(x_j)/\tau_z}} \tag{1}$$

where z is either students s or teacher t, τ_z is the temperature, and $i \in \{1, 2, \ldots, K\}$. This will result in four probability distributions: two for the student ($P_s(x_1)$, $P_s(x_2)$) and two for the teacher ($P_t(x_1)$, $P_t(x_2)$). The student network is then trained through a cross-entropy loss between the student and teacher probability distributions while the teacher network is updated using an Exponential Moving Average (EMA) of the student network.

Table 1
Node and edge features of the μRTS graph game state

Type	Description	Number of Features
Node	Unit type	7
	Resources	1
	Hit points	1
	Player indicator	1
Edge	Distance	1
	Directionality (radians)	1

DINO uses *sharpening* and *centering* of the teacher network's outputs to prevent the learned representations from collapsing to a single dominant dimension, or a uniform distribution regardless of the input. Sharpening is implemented by setting τ_t to a low value (in our experiments, $\tau_t = 0.0.4$), while centering is implemented by subtracting a center value c from the teacher's outputs. This center value is computed using first-order statistics of the teacher's outputs in a batch, and is updated every training iteration using EMA:

$$c = m * c + (1 - m) * \frac{1}{B} \sum_{b=1}^{B} [t_\theta(x_{b,1}) | t_\theta(x_{b,2})]$$

where $m \in [0, 1]$ is the EMA rate parameter, $|$ is a concatenation, B is the batch size, and $x_{b,1}$ and $x_{b,2}$ are the two views for an image at batch index b.

3.3. Graph Attention Network V2

The main contribution of our work is a transferrable pretrained graph neural network (GNN) state encoder. A GNN is a class of networks designed to process graphs, and has been applied to computer games [10], neuroscience [19], etc. GNNs can create latent node, edge, and graph representations, which can subsequently be used for a variety of downstream graph-oriented tasks, including node, edge, and graph classification and regression. A single-layer GNN can capture neighboring relationships for each of the nodes in a graph, while adding additional layers can help model deeper node relationships. The GNN that we focus on using in this paper is Graph Attention Network V2 (GATv2) [15].

GATv2 takes in a graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \ldots, N\}$ is a set of nodes, each containing a set of features $h_i^0 \in \mathcal{R}^k$, and $\mathcal{E}$ is the set of directed edges (i, j) from node i to node j, each optionally containing edge features $e_{i,j} \in \mathcal{R}^z$. GATv2 uses message passing to capture shallow and deep relationships between nodes in the graph. More specifically, for each node i in the graph, message passing aggregates (via summation) scaled features from neighboring nodes $\mathcal{N}_i$, and integrates them into the node. This scaling is a normalized score stating the importance of the neighboring node's features to i, where the normalization is done over all nodes in $\mathcal{N}_i$.

4. Learning Graph State Representation for microRTS

Graphs allow us to model the relational and relative spatial information across μRTS entities by representing them as nodes, with edges defining relations between them. Specifically, we represent an observed game state in μRTS as a fully-connected graph, where each node in the graph is an entity with a set of game features relevant to the entity. These node features are a one hot encoding of the unit's type (worker, base, barracks, heavy, light, ranged, resource), concatenated with the amount of resources, hit-points, and an indicator of whether this is a player's unit; if a feature is not available for the node's unit, that feature is 0.

The edges in the graph encode relative spatial information through euclidean distance and relative directionality between entities in the game state. The relative directionality is the angle (in radians) between a source and destination node, computed by (1) taking the inverse tangent between the destination and source node, and (2) adjusting the angle based on the positioning of the two nodes with

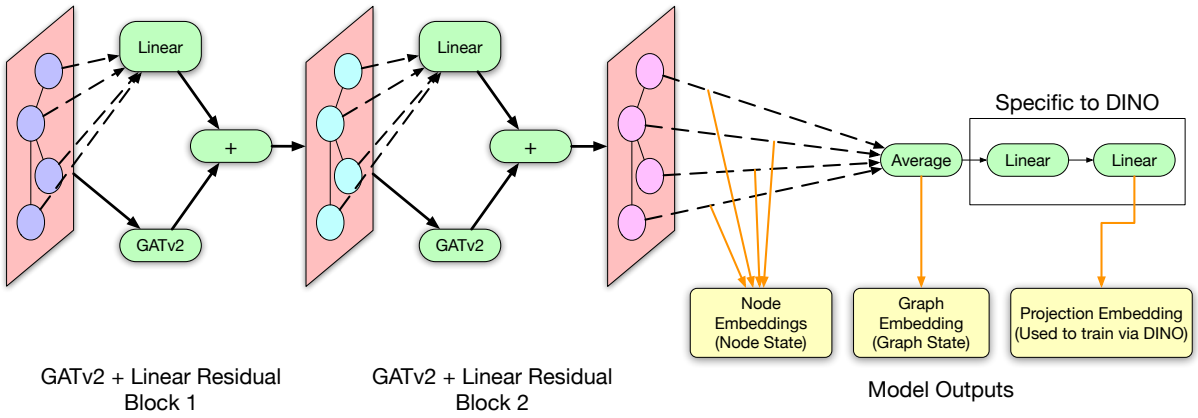

Figure 3: Graph state encoder model used for learning latent graph game state representations

respect to the orientation of the game map. For example, if the destination node is to the top-left of the source node, then the relative angle will fall under $[\pi/2, \pi]$ (second quadrant of an x,y grid). Table 1 provides a summary of the node and edge features used in the μRTS game state.

4.1. Graph Self-Supervised Learning

We want a GNN to learn to project semantically-similar game state graphs close together in latent space. Contrastive learning approaches [17] can fulfill this goal, but these approaches tend to be very computationally expensive. Instead, following previous work on GraphDINO [19], we use the self-supervised learning approach DINO to learn latent node and graph state representations. Figure 2 (right) provides an illustration of how graphs are processed by GraphDINO and our approach. Both approaches train an encoder model (*Student Encoder* in Figure 2) to place semantically-similar graphs close together in latent space by having the model predict the latent space it previously learned (*Teacher Encoder* in Figure 2). However, GraphDINO and our approach diverge in *what* and *how* graphs are processed by DINO. Thus, these methods can be viewed as two variations of graph SSL using DINO.

More specifically, there are two major differences between GraphDINO and our approach. The first difference is the type of encoder model architecture that is trained by DINO (the *how* difference). Since we are working with graphs, we use GNNs over Transformers as GNNs are designed specifically for graphs. Figure 3 provides a visualization of our graph state encoder, a network with two GATv2 [15] GNN and residual connection blocks (*GATv2 + Linear Residual Block 1 and 2* in Figure 3). The state encoder takes, as input, a graph $G_0 = (V, E)$ with node $h_i^0 \in \mathcal{R}^k$ and edge $e_{i,j} \in \mathcal{R}^z$ features $(i, j \in V)$, and passes the graph through a GATv2 layer while simultaneously passing the node embeddings through a linear residual layer. Both the GATv2 and linear residual layer output node embeddings, which are then added together to yield new node embeddings h_i^1 $(\forall i \in V)$ for graph $G_1 = (V, E)$ that is topologically the same as G_0. Next, G_1 and its node embeddings are passed into a second GATv2 and residual layer, respectively, and their node embeddings are added together resulting in G_2 with new node embeddings; these node embeddings h_i^2 $(\forall i \in V)$ are then averaged together to get a graph embedding. Next, this graph embedding is passed through two projection layers (linear layers) to get a final projection embedding that is used by DINO to learn node and graph representations (*Specific to DINO* in Figure 3). After training, the two projection layers are removed from the encoder and the node and graph embeddings are used for downstream tasks.

The second difference is the type of graph (graph representing game state) and stochastic perturbations conducted over these state graphs (the *what* difference). General graph perturbations, such as node feature masking and edge masking [18], etc., can result in changes to the semantics of a state graph. For example, if we mask the *resources* feature of a resource node in the graph game state, this would indicate that resource does not exist in the state, which changes what that state means (i.e., high $\rightarrow$ low resource state). We explore using three stochastic graph perturbations that maintain the semantics of

Table 2
Maps and agents from CoG 2019 and CoG 2020 μRTS competition used in our evaluation

CoG 2019		CoG 2020	
Maps	**Agents**	**Maps**	**Agents**
basesWorkers8x8A	RandomBiasedAI	basesWorkers8x8A	RandomBiasedAI
basesWorkers16x16A	POWorkerRush	basesWorkers16x16A	POWorkerRush
BWDistantResources32x32	POLightRush	BWDistantResources32x32	POLightRush
BroodWar/(4)BloodBath.scmB	NaiveMCTS	BroodWar/(4)BloodBath.scmB	NaiveMCTS
FourBasesWorkers8x8	Tiamat	FourBasesWorkers8x8	CoacAI
TwoBasesBarracks16x16	Droplet	TwoBasesBarracks16x16	UMSBot
NoWhereToRun9x8	Izanagi	NoWhereToRun9x8	GuidedRojoA3N
DoubleGame24x24	MixedBot	DoubleGame24x24	Rojo
-	Hybrid_HTN_MCTS_Planner_Bot	chambers32x32v2	MentalSeal
-	IDVRV_Bot	armyGeneration16x16	UTS_Imass_SocketAI
-	CompetitionSarsaSearch	caldera16x16	-
-	UTS_Imass_SocketAI	paths32x32	-

the original graph, *distance scaling*, *rotation offset*, and *edge removal*:

- **Distance Scaling Perturbation** (changes edge features): The relative distance between nodes is scaled by a factor $\alpha \in [a, b)$; in our experiments, $a = 0.5$ and $b = 1.5$.
- **Rotation Offset Perturbation** (changes edge features): The relative orientation of all nodes is offset by a factor $\beta \in [0, 2\pi)$.
- **Edge removal**: Edges are removed with a probability p following a Bernoulli distribution; in our experiments, $p = 0.5$.

The parameters for each perturbation is based on human judgement of what could significantly change the original graph state while not changing the semantics of the state.

5. Experiments

We now study the graph game state representations learned by DINO. First, we provide a qualitative cluster analysis which will allow us to understand what patterns and associations are contained in the representations that DINO learned, and the graph state encoder. Second, we conduct a quantitative evaluation of the learned representation's performance for the task of action prediction to understand the transferrability of the representation on a game AI task.

We use a series of μRTS replay datasets for these experiments generated from two-player, deterministic, and fully-observable games. Specifically, we use replay data from the CoG 2019 and 2020 μRTS competitions as these were readily available to the authors for use. Each replay in the datasets consists of a sequence of state, action pairs conducted by the two players. We parse each replay by skipping the first 200 state, action pairs and extracting every 10 state, action pairs for a maximum of 64 pairs. This is done so we can acquire diverse states for training and evaluation.

Training Dataset: The training dataset for the graph state encoder comes from the CoG 2019 μRTS competition replay data.[3] The competition consisted of three tracks, *standard* (deterministic and fully-observable), *non-deterministic*, and *partially-observable*. Each track involved a five-iteration round robin tournament where 12 agents played on eight open maps and four hidden maps; see Table 2 for the maps and agents. We use the first three iterations of the standard track for training (without the hidden maps), which contains 6336 replays, and ignore replays containing the *RandomBiasedAI* agent as its random actions will frequently yield states that are noise to the graph state encoder, resulting in 5808 replays and 38934 game states.

[3] https://sites.google.com/site/micrortsaicompetition/competition-results/2019-cog-results

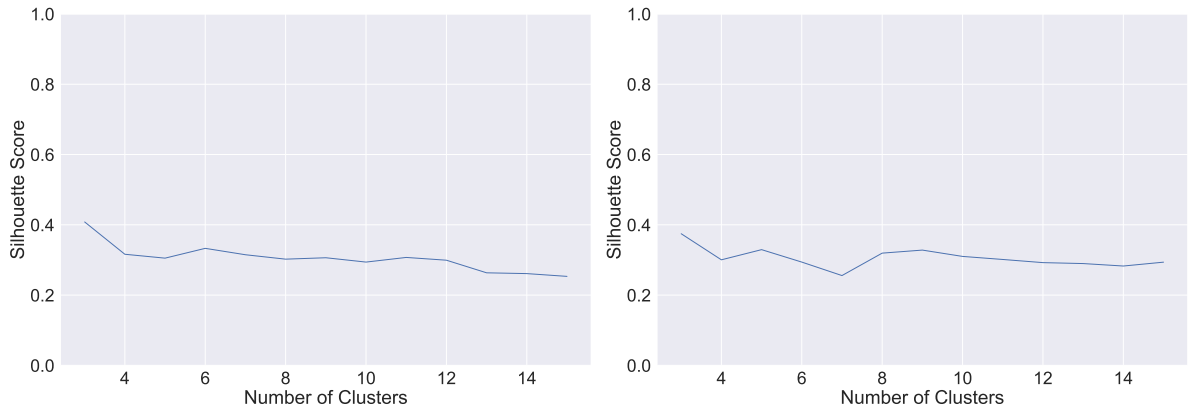

Figure 4: Silhouette score for Dataset 1 (left) and Dataset 2 (right) over 3 to 15 clusters

Evaluation Dataset 1: The first evaluation dataset consists of the last two iterations of the standard track of the CoG 2019 μRTS competition replay data, and contains 4224 replays (see Table 2 for the maps and agents). Similar to the training dataset, we ignore replays containing the *RandomBiasedAI*, resulting in 3872 replays and 17813 game states.

Evaluation Dataset 2: The second evaluation dataset consists of replay data from the last two iterations of the CoG 2020 μRTS competition.[4] Similar to the CoG 2019 competition, there are three tracks, *standard*, *non-deterministic*, and *partially-observable*. Each track involved a five-iteration round robin tournament where 10 agents played on eight open maps and four hidden maps; see Table 2 for the specific maps and agents. We run our evaluations on all maps from the last two iterations of the standard track. Similar to the first evaluation dataset, we ignore replays containing the *RandomBiasedAI*, resulting in 3888 replays and 26221 game states.

Pretraining Configuration: The graph state encoder is configured with a hidden dimension of 256 and is trained for 10 epochs with a batch size of 16. We use the AdamW optimizer [26] with a learning rate computed using a cosine scheduler with a linear warmup of 1000 steps and initial learning rate of 4e-7. Specific to DINO, the teacher network is updated using an EMA of the student network, where the EMA rate is $\alpha = 0.996$, the student and teacher temperature in Equation 1 is $\tau_s = 0.1$ and $\tau_t = 0.0.4$, and the EMA rate parameter for centering is $m = 0.9$ [14].

5.1. Cluster Analysis

We cluster the graph embeddings generated by our graph state encoder over Datasets 1 and 2 using k-medoids, an off-the-shelf clustering algorithm that clusters datapoints into k partitions and is less susceptible to outlier datapoints compared to k-means. To find the optimal number of clusters for each dataset, we compute the silhouette score for clusters ranging from 3 to 15, and use the elbow method to find an appropriate number of clusters. The elbow method is a visual heuristic that provides us with the largest number of clusters before the silhouette score starts to plateau. We use TSNE [27] to reduce the dimensionality of the graph embeddings for visualization.

Figure 4 provides the silhouette score for 3 to 15 clusters over Datasets 1 and 2; using the elbow method, we see that 7 clusters provides good fit over Dataset 1 while 8 clusters provides a good fit over Dataset 2. Figure 5 provides a visualization of the 7 clusters over Dataset 1. The visualization shows a good separation all clusters. To understand what could be contained in the clusters, we look at the distribution of various game features contained in each of the clusters. Figure 6 contains the average count of player units in each of the clusters. Overall, we see that heavy and ranged units are rarely

[4] https://sites.google.com/site/micrortsaicompetition/competition-results/2020-cog-results

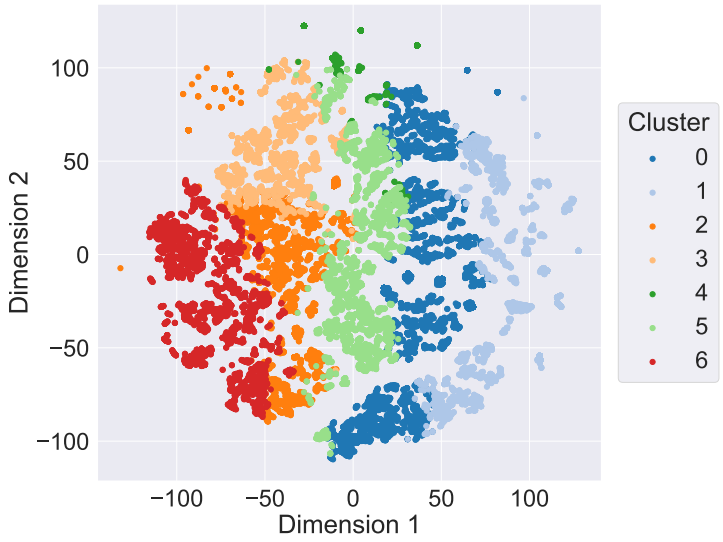

Figure 5: Cluster visualization of the learned latent space for Dataset 1. Dimensionality reduction done by TSNE

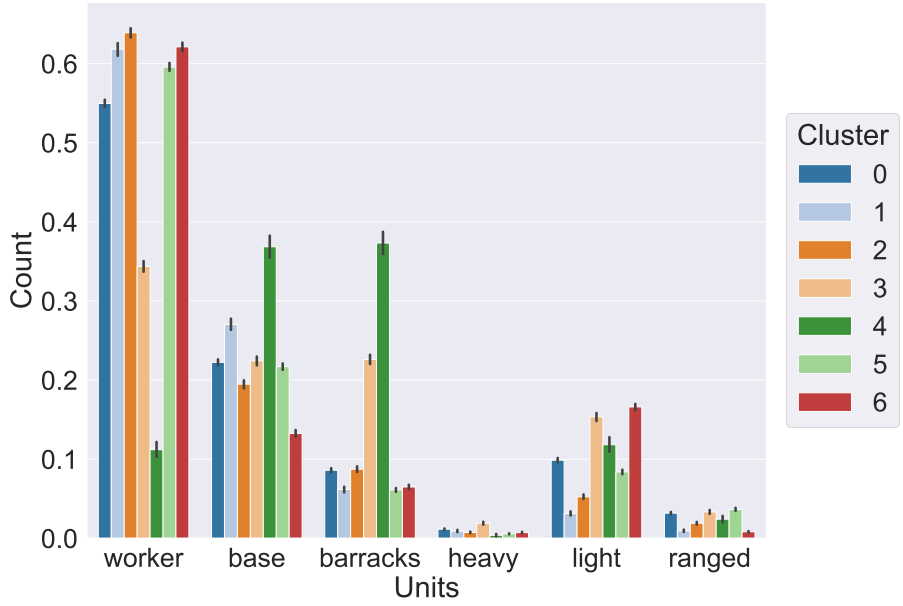

Figure 6: Player unit counts in the learned clusters for Dataset 1

contained in the clusters, while worker units are seen frequently in all clusters except for Cluster 4. The frequency of worker units makes sense as workers are directly and indirectly required for many of the core gameplay features in μRTS: harvesting and building bases, barracks, and offensive units.

While Cluster 4 lacks worker units, we do see that Cluster 4 has the highest number of bases and barracks. Figure 7 provides the distribution of game maps across the clusters and we see that Cluster 4 has a high concentration of game states from *BroodWar/(4)BloodBath.scmB* (64x64 game map), which is a large enough map to allow for expansion and construction of multiple bases and barracks. We also see that other clusters have a high concentration of particular game maps. For example, Figure 7 shows that *FourBasesWorkers8x8*, *BWDistantResources32x32* and *DoubleGame24x24* is most frequent in Cluster 1, Cluster 5, and Cluster 6, respectively. This result implies that the graph state encoder's latent space may contain information about game maps despite not being explicitly trained on spatial map features.

Figure 8 provides a cluster visualization of the learned latent space for Dataset 2. This visualization shows less clear separation of clusters in contrast to Figure 5, particularly for Clusters 0 and 7. However,

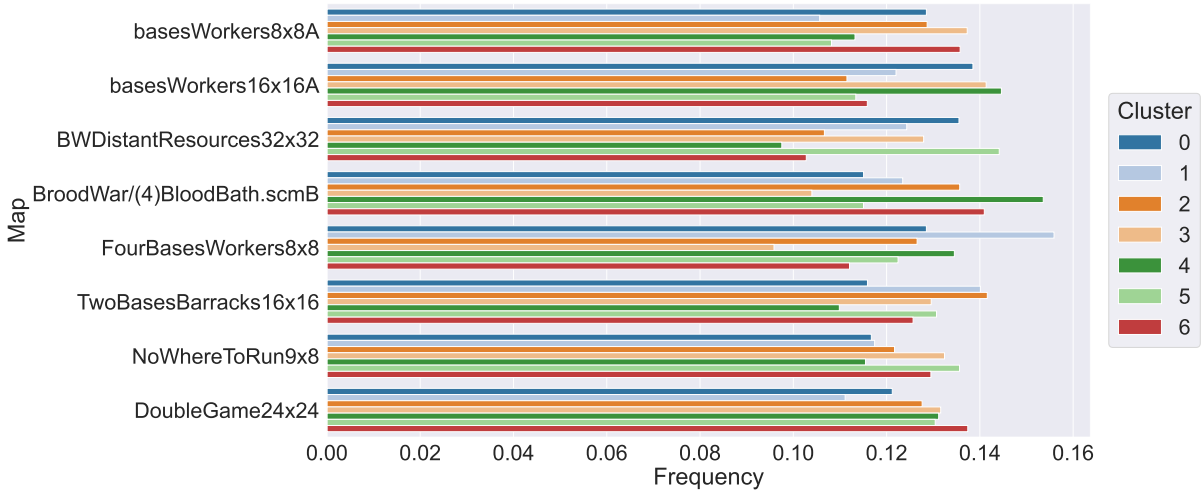

Figure 7: Map distribution in the learned clusters for Dataset 1

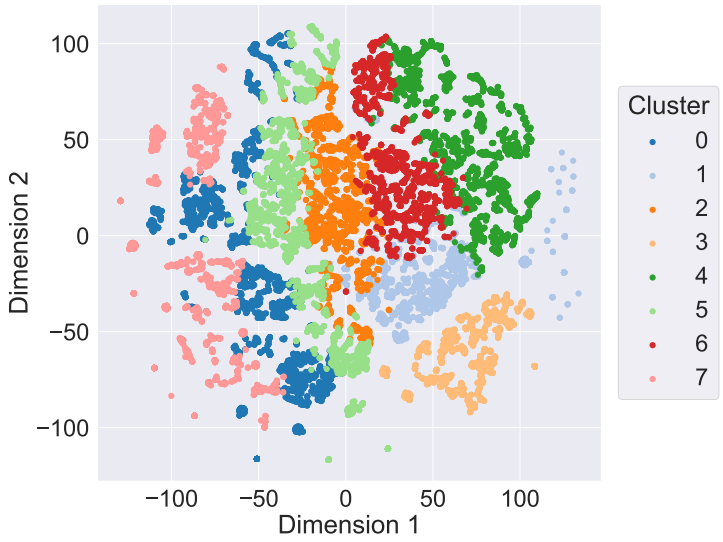

Figure 8: Cluster visualization of the learned latent space for Dataset 2. Dimensionality reduction done by TSNE

the clusters are still separate, indicating that the graph state encoder was able to discover and learn hidden patterns in the graph states from a completely different μRTS competition. Figure 9 contains the count of units in each of the clusters. Our results show that Cluster 3 contains mainly offensive units (light, heavy, ranged) and barracks, which could indicate offense-heavy game states or states from large game maps as heavy offensive units are rarely useful in small maps due to their cost and construction time. Looking at the map distribution in Figure 10, we see a high concentration of data points from Cluster 3 for the maps *armyGeneration16x16* (16x16 game map) and *BroodWar/(4)BloodBath.scmB*, confirming our hypothesis. We also see in Figure 10 that *NoWhereToRun9x8* and *DoubleGame24x24* is most frequently seen in Cluster 0 and Cluster 6, respectively, providing further evidence that the latent space learned by the graph state encoder may contain information about game maps. We also see, in Figure 9, that Cluster 1 has a low number of worker units in comparison to all other clusters except Cluster 3, but has the second-highest concentration of light and ranged units, and barracks. This could indicate that Cluster 1 contains states from agents using a rush strategy as these units are quick and cheap to construct.

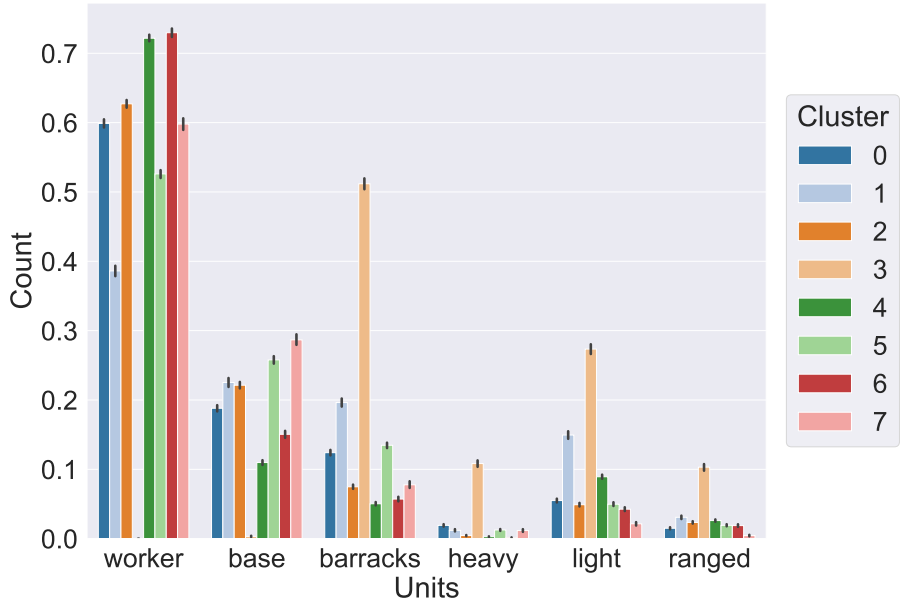

Figure 9: Unit counts in the learned clusters for Dataset 2

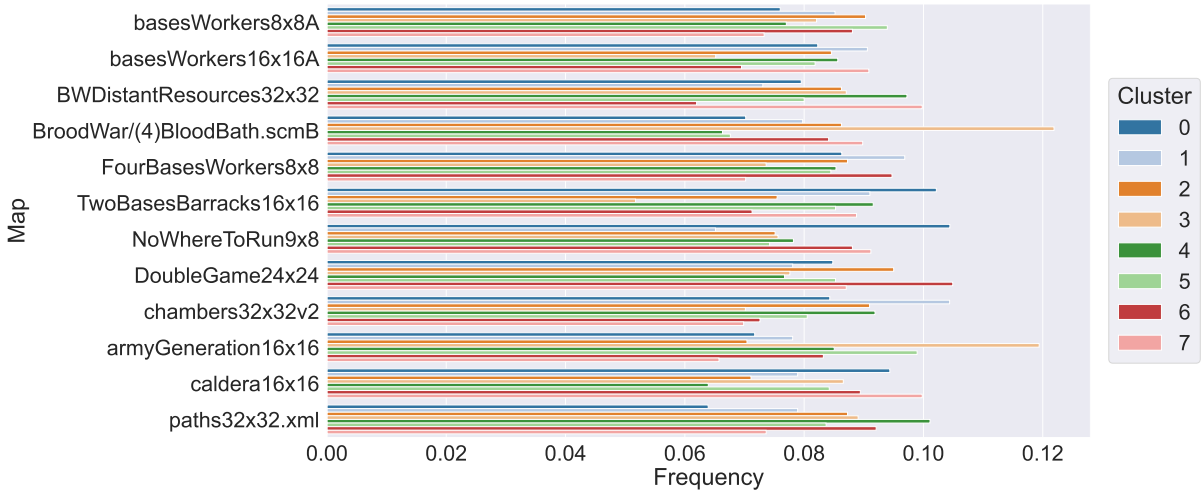

Figure 10: Map distribution in the learned clusters for Dataset 2

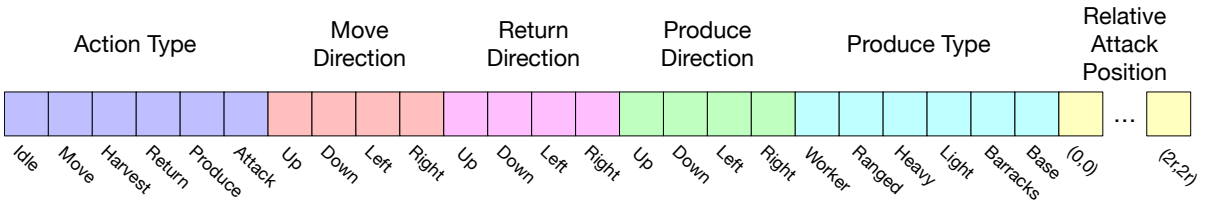

Figure 11: Visualization of the action feature vector used for action prediction

5.2. Action Prediction

We now turn to evaluating the pretrained graph state encoder on the task of action prediction. This task requires a method to determine the type of action (*idle*, *move*, *harvest*, *return*, *produce*, and *attack*) and its parameters (e.g., direction, attack position) for each observable unit in a given μRTS game state. For our experiments, we represent an action as a feature vector, similar to Huang et al. [8] and Goodfriend

Table 3
Action prediction evaluation results (precision, recall, F1 score) averaged over ten runs for different action prediction models ($\pm$ denotes sample standard deviation).

Method	Precision	Recall	F1 Score
action pred. (no GNN)	0.0000	0.0000	0.0000
action pred. random init	0.3135 ± 0.07497	0.1796 ± 0.04401	0.2239 ± 0.05382
action pred. linear-probe (BGRL)	0.007124 ± 0.01270	0.003564 ± 0.006347	0.004751 ± 0.008464
action pred. linear probe (GBT)	0.01370 ± 0.01848	0.006866 ± 0.009231	0.0091451 ± 0.01232
action pred. linear probe (DINO)	0.03947 ± 0.02883	0.019779 ± 0.014378	0.02634 ± 0.01919
action pred. fine-tune (BGRL)	0.3860 ± 0.04749	0.2183 ± 0.03100	0.2736 ± 0.03588
action pred. fine-tune (GBT)	0.3683 ± 0.06401	0.2089 ± 0.04167	0.2613 ± 0.04856
action pred. fine-tune (DINO)	$\mathbf{0.3951} \pm 0.02874$	$\mathbf{0.2281} \pm 0.02126$	$\mathbf{0.2829} \pm 0.02329$

[9]. This action feature vector is a binary vector, where different subsets of the vector represent different parts of an action, and is the same size across all μRTS unit types; Figure 11 provides an illustration of the action feature vector and its subsets. An action is represented using this vector by setting its action type and parameters to 1. For example, if we want to specify an action to *move up*, then we would have the following vector: $[0, 1, 0, 0, 0, 0, 1, 0, 0, 0, \ldots, 0]$. This makes the action prediction task a multilabel classification task.

The action vector subset *relative attack position* (rightmost subset in Figure 11) is specifically used by the *attack* action to denote attack positions for a unit that can attack. This subset has a dimension of m^2, where $m = 2r + 1$ and r is the maximum attack range over all possible unit types, and represents an $m \times m$ attack grid around the unit. When a unit does an attack action, we use the distance of the offensive unit to the position being attacked to determine the attack grid position and assign that position to 1. For example, if the maximum attack range is 1 and the offensive unit is attacking to its upper-left position, then we would have a 3×3 attack grid around the offensive unit that will be attacking position $(0, 0)$, which corresponds to element 0 in the vector subset.

We construct an action prediction model for this task by connecting the pretrained graph state encoder to a single task-specific linear prediction layer. The graph state encoder provides node embeddings for each unit in the game state (*Node Embeddings* box in Figure 3), all of which are then passed into the linear prediction layer to predict the unit's action. We note that this linear prediction layer is not pretrained by DINO and is only trained for the task of action prediction. This action prediction model is trained for evaluation in two ways: *linear probing* and *fine-tuning*. Linear probing assesses the strength of the pretrained encoder's representation by freezing the parameters of the graph state encoder and only training the final linear prediction layer. Fine-tuning assesses the adaptability of the pretrained encoder's representation by jointly training both the encoder and linear prediction layer.

We use four baselines for our evaluation. The first baseline replaces the graph state encoder with three linear layers, where each layer is followed by ReLU activations. This baseline allows us to assess the benefit of using a graph state representation and GNNs, and is denoted as *action prediction (no GNN)*. The second baseline uses a randomly-initialized graph state encoder, which we denote by *action prediction random init*. The final two baselines use a graph state encoder pretrained by Bootstrapped Graph Latents (BGRL) [18] and Graph Barlow Twins (GBT) [20], which we use to compare our approach against previous graph self-supervised learning methods.

We train all action prediction models to conduct multilabel classification over the binary action vectors from Dataset 1. More specifically, for each game state in the replay dataset, we extract the binary action vectors for all units in the state, and have the action prediction models predict each unit's action vector. We then evaluate all models over Dataset 2. To accurately evaluate the models, we want to minimize the number of graph topologies in our evaluation dataset that were seen during training. The graph game state represents relations between units, whose topology directly depends on the gameplay behaviors of the agents. Thus, we remove *RandomBiasedAI*, *POWorkerRush*, *POLightRush*, *NaiveMCTS*, and *UTS_Imass_SocketAI* from Dataset 2 for evaluation as these agents were seen during training, resulting in an evaluation dataset containing 2160 replays and 9434 game states.

Table 3 provides precision, recall, and F1 scores over the Dataset 2 for different action prediction models across ten random seeds. We compare the models using the F1 score and all statistical significance tests for our comparisions are conducted using a one-sided Mann-Whitney U test ($\alpha = 0.05000$) and corrected using the Holm-Bonferroni method. We validate three confirmatory hypotheses using our experiment results. Our first hypothesis is that a GNN-based action prediction model that processes our graph state representation (*action prediction random init*) has a higher mean rank than an action prediction model that only processes unit information and not their relationships (*action prediction (no GNN)*). Table 3 shows that the mean F1 scores for *action prediction random init* and *action prediction (no GNN)* were 0.2239 and 0.0000, and their distributions differed significantly (Mann-Whitney U statistic = 100.0 and $p = 3.193 \times 10^{-5}$). Thus, we can reject the null hypothesis and state that there is a benefit to using a graph representation and GNN for action prediction under our corrected alpha level of 0.01667.

Our second confirmatory hypothesis is that an action prediction model using a randomly-initialized graph state encoder (*action prediction random init*) has a higher mean rank than a linear probe model that uses an encoder pretrained by DINO (*action prediction linear probe (DINO)*). This hypothesis is based on the fact that we do not have access to a diverse set of pretraining data for the state encoder, and thus we do not expect the pretrained latent space learned by DINO to directly be transferrable. Table 3 shows that the mean F1 scores for *action prediction random init* and *action prediction linear probe (DINO)* were 0.2239 and 0.02634 and their distributions differed significantly under our corrected alpha level of 0.02500 (Mann-Whitney U statistic = 100.0 and $p = 9.134 \times 10^{-5}$). We believe that incorporating training data from other μRTS competitions should help improve linear probing transferrability, which we aim to do for future work.

The third confirmatory hypothesis is that a fine-tuned model using an encoder pretrained by DINO (*action prediction fine-tune (DINO)*) has a higher mean rank than an action prediction model using a randomly-initialized graph state encoder (*action prediction random init*). This hypothesis is based on the fact that a model with prior knowledge of μRTS game states should outperform a model without prior knowledge. Table 3 shows that the mean F1 scores for *action prediction fine-tune (DINO)* and *action prediction random init* were 0.2829 and 0.2239 and their distributions differed significantly under our corrected alpha level of 0.05000 (Mann-Whitney U statistic = 81.00 and $p = 0.01057$). This suggests that the latent knowledge learned by the state encoder pretrained by DINO is beneficial when transferred to downstream tasks.

Table 3 also shows that DINO slightly outperformed GBT and BGRL under linear probing and fine-tuning. From this, we can hypothesize that pretraining the encoder using DINO from a larger and diverse set of replay data could outperform both GBT and BGRL under linear probing and fine-tuning.

6. Discussion

Our qualitative and quantitative results showed that the graph state encoder trained by DINO learns meaningful features that successfully transferred to the task of action prediction in μRTS. In particular, our qualitative cluster analysis showed that the latent graph states learned by the state encoder contain information about μRTS game maps. This is particularly interesting because our graph representation only provides map information directly through resources and their counts; any spatial information is only provided through unit formations defined by relative distance and orientation between units in the game state. This implies that the state encoder acquired information about the game maps without explicit spatial map features such as terrain or map geometry. This finding is very important for the applicability of game-playing and player modeling approaches for μRTS as spatial features are usually acquired from state encoders that directly process the spatial aspect of game maps (e.g., CNNs), but these models struggle to apply across the non-uniform map sizes (height and width) seen in μRTS [10]. Our graph state encoder is applicable across these maps, providing game-playing and player modeling approaches an alternative for encoding game states. One next step would be to understand any spatial features learned by our graph state encoder, and assess performance differences between spatial features learned by the encoder and models that directly process game maps.

Our quantitative results showed that the graph state encoder can be successfully transferred to the task of action prediction when fine-tuned, even when the encoder was pretrained on data from a single μRTS competition (CoG 2019 competition). The actions we used for the action prediction experiments are the same as those used in prior work for DRL in μRTS [8, 9], which provides a signal that the encoder could be transferrable to DRL. However, the encoder DINO learned struggled to perform well on the task of action prediction under linear probing. This could be due to two possible issues. First, DINO, which trains the encoder to learn general graph representations, may have been too general or not amicable for action prediction. While pretrained models should ideally learn task-general representations, these representations should also be helpful for any downstream task. Second, there may be a sizable distribution shift between the pretraining data and data used for action prediction. This second issue can be remediated by training on diverse replays from other, more recent μRTS competitions, which we are working on acquiring for future work.

Our results also demonstrated that graph state representations can be helpful for action prediction in μRTS. Graphs encode relationships through their nodes and edges, which allows us to represent team formations, but other modalities exist that encode different types of information. For example, images provide spatial and view information, which can be helpful for understanding geometry. We believe that graphs can encode one piece of the overall game state, but other modalities will be needed to fill in information gaps left by graph representations. Thus, another area of future work would be to learn multimodal state encoders that could build a “complete" understanding of the game state.

7. Conclusion

This paper presents initial work on pretraining latent graph state representations of game states in μRTS. Our results show that our graph representation and graph state encoder are beneficial for the task of action prediction, performing similarly or better than several baselines when the encoder was fine-tuned. Our immediate steps will be to conduct an in-depth evaluation of the graph state encoder and representations on tasks such as DRL and player modeling. We also would like to gather additional μRTS replay data, conduct systematic hyperparameter tuning of the graph state encoder, and scale the encoder for more representational power. Finally, a long term goal of this work is to study how state models can be pretrained for AI tasks that are generalizable and transferrable across a diverse set of tasks and video games.

Acknowledgments

The authors would like to give a special thanks to Dr. Santiago Ontañón for access to the CoG 2019 and 2020 μRTS competition data. The authors would also like to thank our reviewers for their insightful and helpful feedback.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] S. Ontañón, M. Buro, Adversarial hierarchical task network planning for complex real-time games, in: Proceedings of the 24th IJCAI, 2015, pp. 1652–1658.

[2] S. Ontañón, Combinatorial multi-armed bandits for real-time strategy games, Journal of Artificial Intelligence Research 58 (2017) 665–702.

[3] J. R. Marino, R. O. Moraes, C. Toledo, L. H. Lelis, Evolving action abstractions for real-time planning in extensive-form games, in: Proceedings of the 2018 AAAI Conference on Artificial Intelligence, 2018, pp. 2330–2337.
[4] G. Synnaeve, P. Bessiere, A bayesian model for plan recognition in rts games applied to starcraft, in: Proceedings of 7th AAAI Conference on AIIDE, 2011, pp. 79–84.
[5] G. Synnaeve, P. Bessiere, A bayesian model for opening prediction in rts games with application to starcraft, in: Proceedings of 2011 IEEE-CIG, 2011, pp. 281–288.
[6] P. Kantharaju, S. Ontañón, C. W. Geib, Scaling up ccg-based plan recognition via monte-carlo tree search, in: 2019 IEEE Conference on Games (CoG), 2019, pp. 1–8. doi:10.1109/CIG.2019.8848013.
[7] O. Vinyals, I. Babuschkin, W. M. Czarnecki, et. al., Grandmaster level in starcraft ii using multi-agent reinforcement learning, Nature 575 (2019) 350–354.
[8] S. Huang, S. Ontañón, C. Bamford, L. Grela, Gym-μrts: Toward affordable full game real-time strategy games research with deep reinforcement learning, in: 2021 IEEE Conference on Games (CoG), 2021, pp. 1–8. doi:10.1109/CoG52621.2021.9619076.
[9] S. Goodfriend, A competition winning deep reinforcement learning agent in microrts, in: 2024 IEEE Conference on Games (CoG), IEEE, 2024, pp. 1–8. doi:10.1109/cog60054.2024.10645610.
[10] Y. Jin, Advanced Deep Reinforcement Learning for Real-Time Strategy Games, Ph.D. thesis, School of Electronic Engineering and Computer Science Queen Mary University of London, 2024.
[11] S. Ontañón, The combinatorial multi-armed bandit problem and its application to real-time strategy games, in: Proceedings of the 9th AAAI Conference on AIIDE, 2013, pp. 58–64.
[12] K. Cho, B. van Merriënboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: Encoder–decoder approaches, in: D. Wu, M. Carpuat, X. Carreras, E. M. Vecchi (Eds.), Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Association for Computational Linguistics, Doha, Qatar, 2014, pp. 103–111. URL: https://aclanthology.org/W14-4012/. doi:10.3115/v1/W14-4012.
[13] Z. Ye, Y. J. Kumar, G. O. Sing, F. Song, J. Wang, A comprehensive survey of graph neural networks for knowledge graphs, IEEE Access 10 (2022) 75729–75741. doi:10.1109/ACCESS.2022.3191784.
[14] M. Caron, H. Touvron, I. Misra, H. Jegou, J. Mairal, P. Bojanowski, A. Joulin, Emerging properties in self-supervised vision transformers, in: 2021 IEEE/CVF International Conference on Computer Vision (ICCV), 2021, pp. 9630–9640. doi:10.1109/ICCV48922.2021.00951.
[15] S. Brody, U. Alon, E. Yahav, How attentive are graph attention networks?, in: International Conference on Learning Representations, 2022, pp. 1–26.
[16] J. Park, H. Jung, H. Park, Cimage: Exploiting the conditional independence in masked graph auto-encoders, in: Proceedings of the Eighteenth ACM International Conference on Web Search and Data Mining, WSDM '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 10–19. URL: https://doi.org/10.1145/3701551.3703515. doi:10.1145/3701551.3703515.
[17] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, Y. Shen, Graph contrastive learning with augmentations, in: Proceedings of the Thirty-Fifth Annual Conference on NeurIPS, 2021, pp. 1–12.
[18] S. Brody, U. Alon, E. Yahav, Large-scale representation learning on graphs via bootstrapping, in: International Conference on Learning Representations, 2022, pp. 1–21.
[19] M. A. Weis, L. Pede, T. Lüddecke, A. S. Ecker, Self-supervised graph representation learning for neuronal morphologies, TMLR (2023). URL: https://openreview.net/forum?id=ThhMzfrd6r.
[20] P. Bielak, T. Kajdanowicz, N. V. Chawla, Graph barlow twins: A self-supervised representation learning framework for graphs, Knowledge-Based Systems 256 (2022) 109631. URL: http://dx.doi.org/10.1016/j.knosys.2022.109631. doi:10.1016/j.knosys.2022.109631.
[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Łukasz Kaiser, I. Polosukhin, Attention is all you need, in: Advances in NeurIPS, 2017, pp. 5998–6008.
[22] C. Trivedi, K. Makantasis, A. Liapis, G. N. Yannakakis, Learning task-independent game state representations from unlabeled images, in: 2022 IEEE Conference on Games (CoG), IEEE Press, 2022, pp. 88–95. doi:10.1109/CoG51982.2022.9893648.
[23] C. Trivedi, K. Makantasis, A. Liapis, G. N. Yannakakis, Game state learning via game scene

augmentation, in: Proceedings of the 17th International Conference on the Foundations of Digital Games, FDG '22, ACM, New York, NY, USA, 2022, pp. 1–4. doi:10.1145/3555858.3555902.

[24] C. Trivedi, A. Liapis, G. N. Yannakakis, Contrastive learning of generalized game representations, 2021. URL: https://arxiv.org/abs/2106.10060. arXiv:2106.10060.

[25] J. Jiang, L. Wu, Z. Hu, R. Wu, X. Shen, H. Zhao, Knowledge enhanced graph contrastive learning for match outcome prediction, Information Processing and Management 62 (2025) 104010. doi:https://doi.org/10.1016/j.ipm.2024.104010.

[26] I. Loshchilov, F. Hutter, Decoupled weight decay regularization, 2019. URL: https://arxiv.org/abs/1711.05101. arXiv:1711.05101.

[27] L. Van der Maaten, G. Hinton, Visualizing data using t-sne, JMLR 9 (2008).