Michael Park
Sachin Chitta
Alex Teichman
Mark Yim
GRASP Laboratory, University of Pennsylvania, USA
E-mail: {parkmich,sachinc,ateich,yim}@grasp.upenn.edu

Automatic Configuration Recognition Methods in Modular Robots

Abstract

Recognizing useful modular robot configurations composed of hundreds of modules is a significant challenge. Matching a new modular robot configuration to a library of known configurations is essential in identifying and applying control schemes. We present three different algorithms to address the problem of (a) matching and (b) mapping new robot configurations onto a library of known configurations. The first method solves the problem using graph isomorphisms and can identify configurations that share the same underlying graph structure, but have different port connections amongst the modules. The second approach compares graph spectra of configuration matrices to find a permutation matrix that maps a given configuration to a known one. The third algorithm exploits the unique structure of the problem for the particular robots used in our research to achieve impressive gains in performance and speed over existing techniques, especially for larger configurations. With these three algorithms, this paper presents novel solutions to the problem of configuration recognition and sheds light on theoretical and practical issues for long-term advances in this important area of modular robotics. Results and examples are provided to compare the performance of the three algorithms and discuss their relative advantages.

KEY WORDS—modular robots, configuration recognition, graph isomorphism

1. Introduction

A modular robotic system composed of reconfigurable units has been an active area of research for the past two decades. Various groups have created an array of different models with diverse control schemes (master/slave, entirely local, genetic algorithms) (Fukada and Kawauchi 1990; Murata et al. 1998; Kotay et al. 1998; Chirikjian 1994; Rus and Vona 1999; Ünsal et al. 1999; Murata et al. 2000; Yim et al. 2000; Castano et al. 2000; Fitch et al. 2000; Jorgensen et al. 2004; Shen et al. 2006). These systems are made with a number of repeated units that can be rearranged to form different configurations, each of which can be used for different applications. For example, a snake-like configuration may be good at going through small holes, while a monkey-like configuration may be better at climbing. As the number of modules increases, the number of possible configurations rapidly increases.

Automatic configuration recognition is the process by which a modular system can determine its own configuration without having it explicitly programmed. This has a variety of uses including:

- *Function follows form* can be implemented with preprogrammed behaviors that can be called up based on the configuration. For example, if modules are assembled into the shape of a dog, the system is controlled to behave like a dog, if the modules are in the configuration of a snake, then the system is controlled to behave like a snake, etc.
- *Rapid manual repair* for broken labeled modules can mean that automatic configuration detection allows any module to replace any other without reprogramming. The important characteristic is the module's connectivity relationship to other modules (e.g. location in a graph) and not an ID number.
- *Self-repair* with many repeated modules can include the reconfiguration of a system to move broken modules to non-critical locations in the configuration. Identifying isomorphic configurations can be one step.

In all three cases listed above, it would be more interesting to not only find the isomorphic configurations, but find the functionally similar configurations. For example, a four-limbed configuration with ten modules in each limb is similar to a four-limbed configuration with nine modules. Finding the set of all functionally similar configurations is difficult as an

The International Journal of Robotics Research
Vol. 27, No. 3–4, March/April 2008, pp. 403–421
DOI: 10.1177/0278364907089350
©SAGE Publications 2008 Los Angeles, London, New Delhi and Singapore
Figures 1–6, 12, 13 appear in color online: http://ijr.sagepub.com

understanding and breakdown of the function of possible tasks would be required as well as reasoning about failure modes. It is easier (but not easy) to find the set of all kinematically similar or isomorphic configurations which would be a step towards finding functionally similar configurations. This is the main focus of this paper.

This work presents a comparison of three methods of matching a new configuration to a set of known configurations and mapping their physical labeled modules to their logical position in a control scheme. The three methods are:

1. an automorphism grouping method using *nauty*;
2. a spectral decomposition approach;
3. a heuristic graph search called 3DLL.

An analysis of these approaches is given with insights into understanding the physical relationships between the scalability and execution times of the different approaches.

Chen and Burdick (1996) published related work on enumeration of non-isomorphic configurations and identified kinematically similar structures in this set. The work presented here is a departure from enumeration and instead focuses on algorithmic approaches to configuration recognition and mapping to configurations in a known database. A positive match and mapping provides the network structure needed for isomorphic control.

Castano and Will (2001) introduce the matching of a physical arrangement of modules to a known configuration as *configuration discovery* on CONRO. While they address hardware and software processes for building a representation of the configuration, they do not address the matching problem, instead referencing the graph automorphism and *nauty* as a possible approach. CONRO uses unlabeled modules, so mapping of labels is not required. Indeed, the unlabeled system is a subset of the problem in this paper. Arbitrary labeling can be applied to two unlabeled configurations and checked to see if they match. In addition, we present complete implementation details for three different algorithms each of which has its own relative advantages.

We include a new algorithm that exploits the special structure of our problem and is able to achieve impressive improvements in speed and performance for certain cases. The implementation of these algorithms and analysis of the results achieved with them provides *new* insights into the problem and the feasibility of employing the different techniques we present here.

This use of the special structure is also used by the Butler et al. (2002) goal recognition algorithm. However, their work is designed for distributed systems and solves the matching problem, whereas the methods presented here are centralized systems (exploiting global knowledge) and solve both the matching *and mapping* problems. In fact, mapping solutions may not be necessary in a distributed system. Distributed versions of the methods presented here are possible, but that is left for future work.

1.1. Automatic Configuration Detection

A configuration of a modular self-reconfigurable robot is defined as the arrangement (connectivity) of modules into a single connected component. There are many ways of representing this connectivity, but it is typically done with a graph where the nodes of the graph are the modules and the edges represent the connection between modules.

In a homogenous system, all modules are identical having a number of connection ports c with each pair of ports having w multiple ways (e.g. orientations) of being connected. The number of different ways that n modules can connect into one connected component is very large, approaching $(cw)^n$ since each module added to a configuration can be added to any of $c(n-1)$ connection ports in w different orientations. However, many of these configurations are morphologically identical (isomorphic). In addition, many modules have symmetries such that attaching a module results in functionally identical morphologies, although the control may need to be modified (e.g. mirrored modules might need control to have an opposite sense).

Automatic configuration recognition has two functions: (1) identifying a configuration (e.g. checking to see if the configuration is isomorphic to one in a library of configurations); (2) mapping the labeled modules in the physical configuration with the logical arrangement of the known configurations to which it is isomorphic. We call the first part configuration *matching* and the second part configuration *mapping*.

1.2. Graph Representation of Modular Robots

Graphs are concise representations for modular robots and readily allow application of techniques from graph theory. A few principal tools derived from ideas in graph theory are employed in our methods of configuration recognition and mapping. In this section, we introduce the notation and definitions related to work in later sections.

A modular robot configuration is often represented as a graph $G = (V, E)$. The vertices $(V = (v_0, v_1, \ldots, v_n))$ of the graph represent the modules while the edges $(E = (e_1, e_2, \ldots, e_c))$ of the graph represent the connections between the modules. Here n is the number of modules in the robot while c is the number of connections between the modules. The labels for the vertices are the unique node IDs corresponding to each module. Furthermore, a mapping $f_E : E \rightarrow E$ is used to assign a particular edge type to each edge. The edge type represents the type of connection between the two modules (which is based on the ports that are connected to each other). The graph representation can be converted to a matrix representation using a *adjacency matrix* (M). The adjacency matrix for a graph with n vertices and c edges is a $n \times n$ matrix with $M_{ij} = 1$, implying that vertex v_i is adjacent to vertex v_j and is 0 otherwise. A special version of the adjacency matrix called the *port-adjacency matrix* is defined as a $n \times n$ matrix A where A_{ij} represents the port number on module i that module j connects to. Instead of 1 as in an adjacency matrix the element holds an integer from 0 to c.

A graph G_1 is isomorphic to another graph G_2 if the adjacency matrices for the graphs are related by a set of row and column permutations. If γ_{12} is the isomorphism between the two graphs, then $M_{\gamma_{12}}(G_1) = G_2$. A graph may also be isomorphic to itself, i.e. there exists an isomorphism γ such that $M_\gamma(G_1) = M(G_1)$. The set of graph automorphisms is especially important to test against when trying to identify module configurations. The problem of modular robot configuration recognition and mapping can now be formally defined as follows: given a set of robot configurations $G = G_1, G_2, \ldots, G_n$ and a new modular robot configuration G_{new}, identify a robot configuration $G_i \in G$ that is isomorphic to G_{new}.

1.3. Organization

The paper is organized as follows. In Section 2, we introduce a modular robotic system which serves as a good example system for connectivity analysis. Sections 3–5 discuss the three implemented solutions with Section 3 serving to introduce the problem in more detail. These implementations were compared by testing on different sets of configurations including a set of randomly generated configurations. These tests and results are shown in each section and discussed in Section 6. Some experimental results of an embedded application are also shown in this section. Finally, Section 7 talks about the implications of these results and describes future work.

2. CKbot

While many of the current incarnations of modular robots are intended for self-reconfiguration, the work here applies to both self-reconfigured and human reconfigured modules. In both cases, the connection mechanisms, whether autonomous or manual, define the possible set of configurations.

A connector physically attaches two modules together and often connects power and/or communications as well. The connectors may be gendered (male and female) or they may be hermaphroditic (containing both male and female components). Gendered connectors connect only to the opposite gender whereas hermaphroditic connectors can be homogeneous, where all connectors are the same and can connect to each other. For this paper we will consider only homogeneous hermaphroditic connectors.

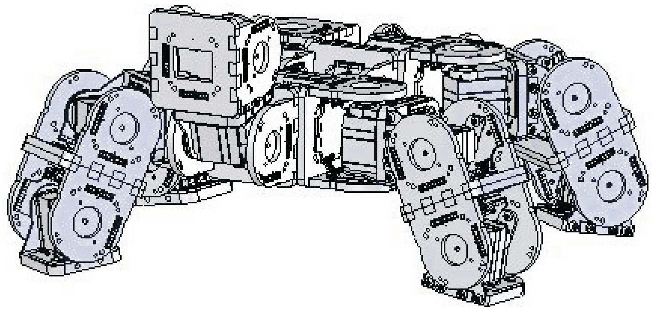
Fig. 1. *Surveyer Dog* with 17 modules in a four-legged configuration.

2.1. CKBot Hardware

CKBot (Connector Kinetic roBot) is a new modular reconfigurable robot that is used as the experimental platform in this work. The kinematics and connector strategy are typical for many chain style reconfigurable modular robots (Yim et al. 2000; Castano et al. 2000; Murata et al. 2000). Figure 1 shows 17 modules in the configuration of a dog. Each module in the system consists of:

1. a laser cut plastic (ABS) body with a hobby servo actuator to control one rotational degree of freedom;
2. a controller (PIC18F2680) and associated hardware for implementing a controller area network (CAN) communications protocol;
3. four connector faces that pass the communications bus and power bus with an option of attaching at 90° rotations. If power and communications are ignored, the faces have rotational symmetry order four.

Two modules are attached together by using screws (four pairs of holes are available, four threaded, four non-threaded). An electrical header is included in between the modules to facilitate the communications and electrical power bus. With this header the connection is homogeneous and hermaphroditic. Power can be supplied either from an external power supply or on-board Li-poly batteries that plug into the power/data ports on the module.

The module can be considered as a cube with connectors on the top, bottom, left and right faces as in Figure 2. The top, left and right faces are rigidly mounted together, whilst the bottom face is actuated to rotate up to 90° to form the front or rear face of a perfect cube. Functionally the module has one symmetry where the module is rotated 180° so left and right sides are swapped, although the actuator would need to be controlled in an opposite sense to be equivalent. The top and bottom connectors almost have the same symmetry; however, the left and right faces are rigidly attached to the top, so a 180°

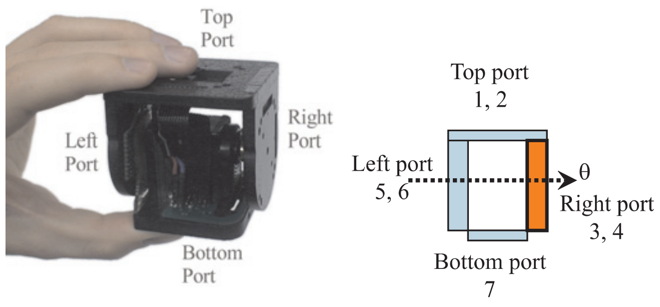

Fig. 2. One CKBot module with a schematic representation. The arrow indicates the rotational axis and the numbers are the port identification numbers assigned to each port.

rotation swapping top and bottom results in a kinematic change (although not positional) in the left and right faces.

The system has a global communications bus allowing each module to talk to and discover which other modules are available. However, this bus does not indicate neighbor connectivity. Local communication allows each module to talk to its neighbors which inherently indicates the connectivity (presence or absence) of its neighbors. CKBot uses an infrared emitter/detector-based local communications mechanism.

Figure 2 shows a CKBot module with four connection ports and the two-dimensional schematic representation of a module. Each port except the bottom port has two infra-red (IR) transmitter/receiver pairs. Figure 3 show the layout of the seven IR pairs. Note that, when two faces are attached together, the transmitter LED (TX) faces directly on to the receiver photodiode (RX) on the opposing face and *vice versa*. Currently, 40 CKBot modules have been constructed and a variety of tasks have been demonstrated including moving like a snake, rolling like a tread, digging in sand and walking like a slinky toy. Section 6.1 also describes a demo combining multiple gaits.

2.2. Low-level Software

The software used in CKBot builds upon the Robotics Bus (Gomez-Ibanez et al. 2004). The Robotics Bus was designed to be adaptable to a variety of applications based on the CAN bus communication protocol. One feature of the Robotics Bus architecture is a broadcast *heartbeat* signal of each module about once every second. This signal contains a unique identifier allowing other modules on the bus to periodically identify the other modules present on the bus. Note that this necessitates the assignment of a unique node ID number to each module. The Robotics Bus also promotes *browseability*, i.e. the ability of a central controller to query and identify the important attributes of a module. This includes the number and type of modules, important parameters such as gain and scaling factors that can be set externally, the module's current state and

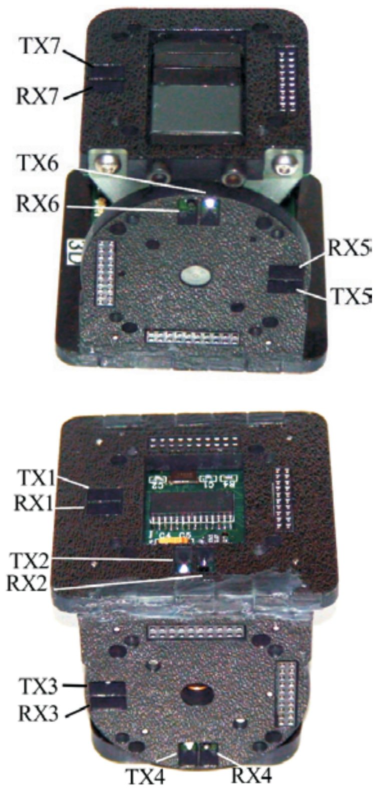

Fig. 3. Two IR transmitter and receiver pairs are on each side except the bottom port which has one pair.

sensor data associated with the module. For CKBot, the important information available on the bus includes the joint angle sensor for the module.

The unique node ID on CKBot means that the modules are labeled. Modular systems may either be labeled or unlabeled (Castano et al. 2000). For many control schemes, module labels and configurations are coupled in a manner critical to the control. For example, the methods rely on mapping of physical modules to logical configuration dependent locations. In one such method individual modules perform prescripted motions in a *gait control table* (Yim et al. 2001). These control methods are applied only to specific configurations or configurations that can be extended in a regular pattern (e.g. making a snake longer, or increasing the number of legs in a centipede.) In most of these cases, the configuration of the robot and the position of each module in the robot is specified either manually, or determined using a set of rules for a small class of configurations.

Each CKBot module functions as an independent entity. Distributed control is possible with CKBot, although for this paper centralized control will be the focus as it is easier to implement and explain.

When power is applied to a CKBot system, the central controller logs *heartbeats* and additional information including positional feedback values and neighbor information of each module. The central controller can use this information to output a desired gait. Since the robots are sometimes hand assembled, the same configuration can be formed by putting together disparate sets of modules. The central controller must then recognize the manually assembled configuration and output the correct set of gait values to different individual modules. The *heartbeat* allows a central controller to recognize the addition of new modules or the removal (or failure) of modules in the robot while running.

2.3. Neighbor Discovery Subroutine

Neighbor discovery is the process where each module detects the presence or absence of neighboring modules and a representation of the connectivity of the whole system is generated. This was the central contribution in Castano et al. (2001). On CKBot, detection is done by the seven IR ports on each connection as shown in Figure 3.

There are 11 possible ways that the top, left or right ports can be attached to another module and seven possible ways for the bottom port. Each module has seven IR pairs to determine each of these possibilities but one asynchronous serial communication device (on the PIC, a USART). The USART is multiplexed to each IR port.

The neighbor detection procedure can be distributed as in Castano et al. (2001) or centralized. For CKBot a centralized controller is used. The controller designates each module (as discovered from heartbeats), one at a time as a transmitter and the others as receivers. The transmitter cycles through its seven IR ports, waiting for a certain time on each port and transmitting its node ID on that port. The receivers cycle through checking their seven ports at a rate eight times faster than the transmitter. Any contact with the transmitter is logged in a neighbor table along with the node ID specified by the transmitter. Each module stores information about its own neighbors. A complete neighbor map can be constructed by the central controller by querying this information from the individual modules.

In this section we looked at some of the conventions and implementation details necessary to formulate the problem of configuration recognition and mapping. In the next three sections, we will present the three algorithms used to solve this problem.

3. Identifying Graph Isomorphisms

Traditional techniques from graph theory form the basis of the first approach to solving the configuration recognition problem. The process involves two steps: computing the graph isomorphism that relates the two configurations and then mapping the two configurations onto each other. The first problem is tackled by comparing a *canonical* representation for the underlying graph structure (the graph nodes without port information, i.e. the adjacency matrix, not the port-adjacency matrix) of the configurations. This pares the matching problem down to considering a fraction of configurations in the library. The second problem is tackled by comparing the complete automorphism group for the new configuration with the canonical representations in the library.

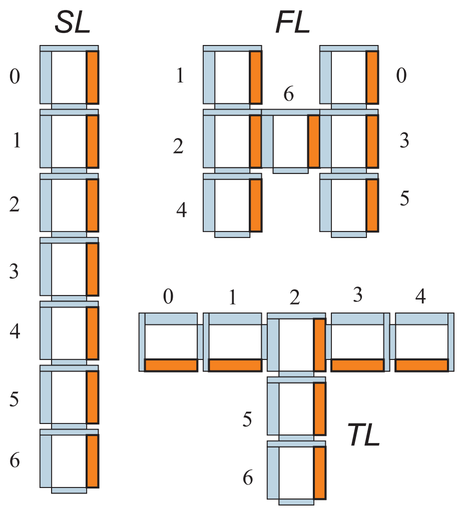

Fig. 4. A database of known configurations.

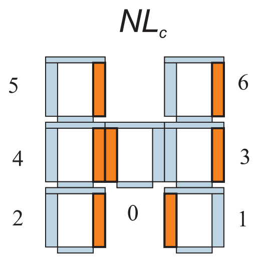

Fig. 5. A new configuration.

3.1. Example

Consider the set of known configurations in Figure 4. All the configurations have seven modules and include a snake-like (*SL*), a four-limb (*FL*) and a three-limb configuration (*TL*). The goal is to match the new configuration *NL* in Figure 5 to one of the configurations in the database.

The *adjacency matrices* for the set of known configurations and the new configuration are generated by labeling the vertices in increasing order of node IDs. Vertex v_0 corresponds to the lowest node ID while vertex v_6 corresponds to the highest node ID. The adjacency matrices for the set of known configurations in the database are given by

$$M_{SL} = \begin{bmatrix} 0&1&0&0&0&0&0\\ 1&0&1&0&0&0&0\\ 0&1&0&1&0&0&0\\ 0&0&1&0&1&0&0\\ 0&0&0&1&0&1&0\\ 0&0&0&0&1&0&1\\ 0&0&0&0&0&1&0 \end{bmatrix},$$

$$M_{FL} = \begin{bmatrix} 0&0&0&1&0&0&0\\ 0&0&1&0&0&0&0\\ 0&1&0&0&1&0&1\\ 1&0&0&0&0&1&1\\ 0&0&1&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&1&1&0&0&0 \end{bmatrix},$$

$$M_{TL} = \begin{bmatrix} 0&1&0&0&0&0&0\\ 0&0&1&0&0&0&0\\ 0&1&0&1&0&1&0\\ 0&0&1&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&1&0&0&0&0\\ 0&0&0&0&0&1&0 \end{bmatrix}.$$

The adjacency matrix for the new configuration is given by

$$M_{NL} = \begin{bmatrix} 0&0&0&1&1&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 1&1&0&0&0&0&1\\ 1&0&1&0&0&1&0\\ 0&0&0&0&1&0&0\\ 0&0&0&1&0&0&0 \end{bmatrix}.$$

The graph isomorphism problem is approached using a software program called *nauty* (McKay 1981). *nauty* (**no auto**morphisims, **y**es?), authored by Brendan McKay (and available electronically from `http://cs.anu.edu.au/bdm/nauty/`), is a software program that determines a set of generators and the size of the automorphism group of a graph. The input to *nauty* is the *adjacency matrix* corresponding to a particular robot configuration. *nauty* returns the automorphism group corresponding to the graph and a canonical representation for the input graph. Two graphs are isomorphic if they have the same canonical representation.

The canonical representations that are used here also represent only the connectivity of the underlying graphs corresponding to the different configuration, but do not contain any information about the port connections themselves. As we shall see later in this section, the port connections are essential in determining actual matches. However, the initial step of comparing canonical forms allows the search space for determining complete matches to be significantly narrowed down.

The first step, as mentioned earlier, is to find the canonical representations of known configurations. Let SL_c denote the canonical representation of a configuration SL.

The *canonical* representations for these configurations are computed using *nauty* and the mappings from the canonical representations to the configurations in the database are given as follows:

$$g_{SL} : \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 6 & 3 & 4 & 2 & 1 & 5 \end{pmatrix},$$

$$g_{FL} : \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 4 & 5 & 6 & 3 & 2 \end{pmatrix},$$

$$g_{TL} : \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 3 & 5 & 0 & 4 & 6 & 2 \end{pmatrix}.$$

The gait for the reference configuration is also mapped to the gait for the canonical configuration. Given $(\phi_0(t), \ldots, \phi_i(t), \ldots, \phi_n(t))$ as the gait for the reference configuration SL, the gait for the canonical configuration of SL is given by $(\theta_0(t), \ldots, \theta_i(t), \ldots, \theta_n(t)) = (\phi_{g_{SL}(0)}, \ldots, \phi_{g_{SL}(i)}, \ldots, \phi_{g_{SL}(n)})$.

On examining the canonical configurations, FL and NL have the same canonical representation given by

$$M_{FL_c} = \begin{bmatrix} 0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1\\ 0&0&0&0&0&1&0\\ 0&0&0&0&0&1&1\\ 1&0&0&1&1&0&0\\ 0&1&1&0&1&0&0 \end{bmatrix}.$$

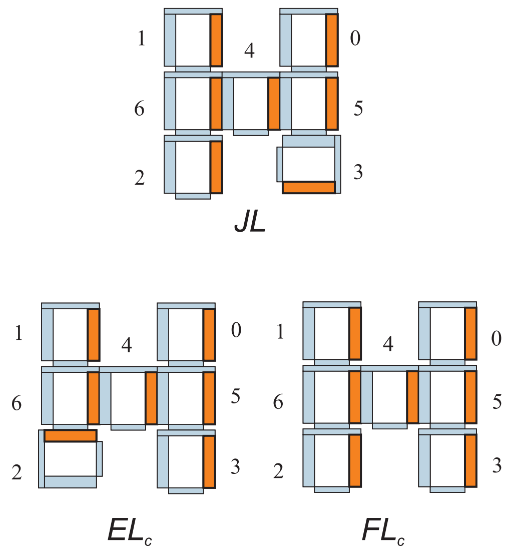

Fig. 6. Examining the automorphism group and the port-adjacency matrix.

It is easy to see that the new configuration is a four-limb configuration. In addition to the canonical representation, a mapping g from the vertices of the canonical configuration FL_c to the vertices (and thus node IDs) of NL is also calculated by the algorithm. Thus, this part of the solution gives two results: (a) a match between NL and FL_c and (b) a mapping from the nodes of NL and FL_c. However, note from Figure 5 that the connections between nodes 0, 3 and 4 in configuration NL are different from the connections between nodes 4, 5 and 6 in configuration FL_c. This necessitates an additional test for complete matching where the symmetries of the individual modules are taken into account, using the *port-adjacency matrix*.

3.2. Automorphisms and the Port-adjacency Matrix

In the previous section, the isomorphism of the underlying graph structures was used to reduce the size of the search space to be considered for the matching problem. However, there are two additional issues in completely matching the new configuration to the reduced database of configurations. Consider, for example, the set of configurations in Figure 6. Here, EL_c and FL_c are two canonical representations for configurations in the database while JL is a new configuration we are looking to identify. The graphs for all three configurations have the same canonical representation, yet only one pair match.

Looking at the figures it is clear that JL and FL_c are different configurations. The difference lies in the nature of the attachment between modules 3 and 5. Although this information is unavailable from just the adjacency matrices corresponding to these two configurations, it is available in the port-adjacency matrices corresponding to these two configurations:

$$A_{FL_c} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 1 & 0 & 0 & 7 & 6 & 0 & 0 \\ 0 & 1 & 7 & 0 & 4 & 0 & 0 \end{bmatrix},$$

$$A_{JL} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 1 & 0 & 0 & 7 & 6 & 0 & 0 \\ 0 & 1 & 7 & 0 & 4 & 0 & 0 \end{bmatrix}.$$

It is easy to see that the two configurations differ in row 4.

Now consider EL_c and JL. They have the same canonical representations, but different port-adjacency matrices. However, with the left/right symmetries, they represent the same configuration. This can be seen on examining the *automorphism group* for JL. An automorphism denotes the isomorphism of the graph to itself and in this case EL_c and JL belong to the automorphism group for the underlying graph structure. Thus, during comparison, two configurations must be compared against each other and against the whole automorphism group for one of the configurations.

The last property we must consider in comparing the two configurations is the symmetry of the modules themselves. Again, this can be seen by comparing EL_c and JL. Modules 5 and 3 in JL map to modules 6 and 2 in EL_c. However, on matching the two configurations and aligning them with respect to each other, all the modules of JL are rotated 180° such that the left and right ports are swapped. Thus, in addition to comparing the port-adjacency matrices, we must also take into account symmetries in the module itself. This is easily done by listing all the symmetries explicitly and comparing against them when carrying out the port-adjacency matrix comparison.

3.3. Final Mapping

Using the techniques detailed in the previous sections, we can now find the mapping from configuration FL_c to configuration NL in our original example:

$$g : \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \to \begin{pmatrix} 6 & 5 & 2 & 1 & 0 & 3 & 4 \end{pmatrix}.$$

Vertex v in the canonical configuration FL_c maps to $g(v)$ in the new configuration NL.

To find the correct mapping for the gait to the new configuration, align the two configurations using the graph isomorphism defined earlier and compute the direction of the joint axis of each module *with respect to* the joint axis of the module corresponding to vertex v_0 in the canonical configuration. Now, compare the direction of the joint axis for the vertex v_i (with respect to v_0) in the reference (database) configuration to the direction of the joint axis for the vertex $v_k = g(v_i)$ (with respect to $g(v_0)$) in the new configuration. The direction can either be parallel or anti-parallel. If the directions are parallel then θ_i in the gait maps to θ_k in the new configuration. However, if the directions are anti-parallel then θ_i in the gait maps to $-\theta_k$ in the new configuration. The desired goal of mapping the gait for the reference configuration $\begin{pmatrix} \theta_1(t) & \dots & \theta_n(t) \end{pmatrix}$ onto the joint angles for the new configuration has been achieved.

Applying this procedure to NL, given the gait $(\theta_0(t), \dots, \theta_i(t), \dots, \theta_6(t))$ for the canonical configuration FL_c, the gait for NL is found to be $(\phi_0(t), \dots, \phi_6(t)) = (-\theta_4(t), -\theta_3(t), \theta_2(t), \theta_5(t), \theta_6(t), \theta_1(t), \theta_0(t))$.

3.4. General Algorithm

The above procedure is presented as a general algorithm in Algorithm 1.

For our example problem from Figure 5, given a gait control table (Yim et al. 2001) for the canonical form of configuration FL,

$$\begin{bmatrix} -45 & -45 & 60 & 60 & 0 & 15 & 15 \\ -30 & -30 & 45 & 45 & 0 & 30 & 30 \\ -15 & -15 & 30 & 30 & 0 & 45 & 45 \\ 0 & 0 & 15 & 15 & 0 & 60 & 60 \\ 15 & 15 & 0 & 0 & 0 & 45 & 45 \end{bmatrix},$$

the mapping found earlier can be used to write a similar gait table for the new configuration NL:

Algorithm 1 *nauty*-based robot configuration matching algorithm.

```
Given a database of configurations $\mathbb{S}$, find the corresponding canonical representation of the database $\mathbb{S}_c$.
For new configuration $E$, compare the canonical representation of $E$ (denoted by $E_c$) with all the elements of $\mathbb{S}_c$. Store any matches in $\mathbb{M}_c$.
if $\mathbb{M}_c \neq \phi$ then
    Construct $\Phi_{\mathbb{M}_c}$ - the set of assembly port matrices for all elements of $\mathbb{M}_c$ under the action of the respective automorphism group.
    Compare $\Phi_{E_c}$ to $\Phi_{\mathbb{M}_c}$ to find a match, $F_c \in \mathbb{S}_c$, for $E$.
    if Match found then
        Generate the map for the robot from $F$ to $F_c$.
        Align $F_c$ and $E$ and map orientation changes for $F_c$ to $E$.
    end if
end if
```

$$\begin{bmatrix} 0 & -60 & 60 & 15 & 15 & -45 & -45 \\ 0 & -45 & 45 & 30 & 30 & -30 & -30 \\ 0 & -30 & 30 & 45 & 45 & -15 & -15 \\ 0 & -15 & 15 & 60 & 60 & 0 & 0 \\ 0 & 0 & 0 & 45 & 45 & 15 & 15 \end{bmatrix}.$$

3.5. Implementation

The algorithm was implemented using Matlab and an interface to *nauty*. The database of reference configurations was generated using a combination of known configurations and randomly generated configurations. Configurations with up to 200 modules were considered. The input to the algorithm is the port-adjacency matrix for a new configuration and the output is the port-adjacency matrix for a matching configuration from the database of known configurations. This is used to determine a mapping from the new configuration to the matching configuration in the database and thus the mapping for any gaits generated for that configuration.

In addition to matching random configurations, tests were also performed with specific configurations, specifically robot configurations that have been constructed and used in our research. Such robot configurations have been constructed with a maximum of 10 modules; however, virtual configurations of the same form were created for testing for the higher number of modules. The configurations created included a snake-like serial line configuration, a centipede configuration, a loop configuration and a plane configuration.

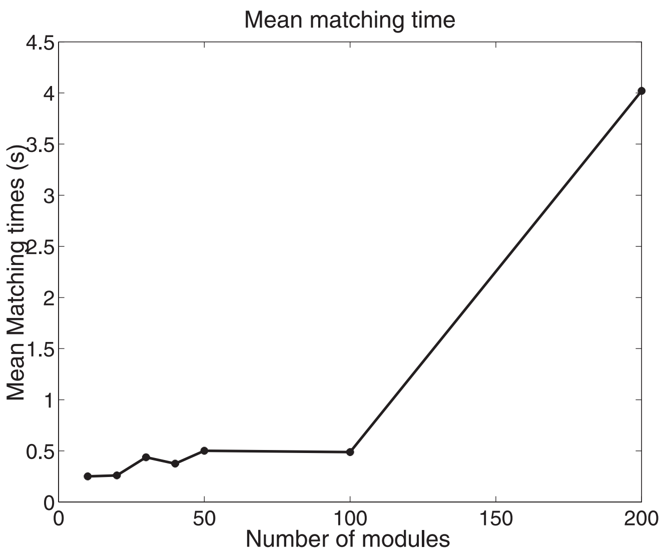

Fig. 7. Mean search times for a *nauty*-based algorithm versus number of modules in the robot configuration.

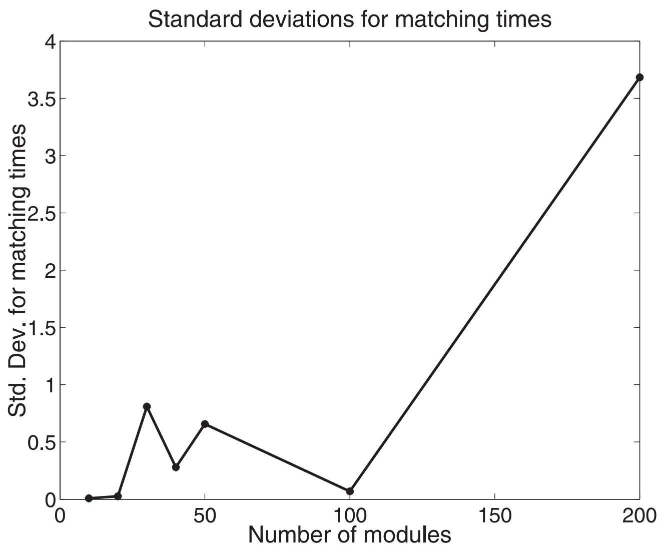

Fig. 8. Standard deviation in seconds for a *nauty*-based algorithm versus the number of modules in the robot configuration.

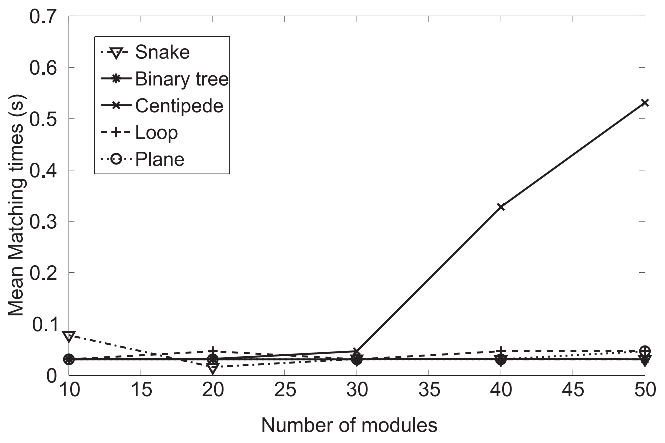

Fig. 9. Mapping times for specific configurations for a *nauty*-based algorithm versus the number of modules in the configuration.

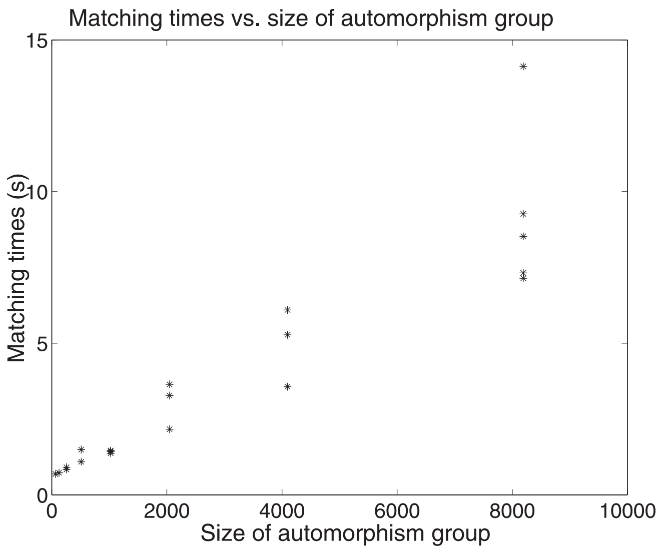

Fig. 10. Matching time versus the size of the automorphism group for a *nauty*-based algorithm.

3.6. Results

From the results in Figures 7–9, it is clear that, while the matching time increases with number of modules, there is considerable variation in matching times even for configurations with the same number of modules. The matching time is a function of the size of the automorphism group for the particular configuration to be matched. This can be clearly seen in Figure 10. Here, the matching times for configurations with 100 modules in each are plotted against the size of the automorphism group for that particular configuration. It is obvious that the relationship between matching time and size of automorphism group is almost linear.

The size of the automorphism group is a reflection of the symmetries in the underlying graph structure of the robot. If the number of symmetries in the structure of the robot is high, the size of the automorphism group will be higher and the algorithm needs to check through more permutations of the graph to perform the matching. Thus, the algorithm will be slower for configurations that are very symmetric and fastest for asymmetric configurations where the only member of the automorphism group is the current configuration itself.

The results also bring out another advantage of using this particular method for lower numbers of modules (< 100). The canonical forms for the graphs are pre-computed and stored beforehand. This ability to preprocess the library greatly reduces the time required for matching since the canonical forms have to be computed only for the new configuration to be matched. The matching then only involves a matrix comparison followed by a permute and match operation for the isomorphic graphs. However, the method scales badly with increase in the number of modules in the configurations. In fact, it was often difficult to obtain meaningful data with configurations of more than 200 modules. Since most hardware implementations so far have been less than 100 modules, this approach will still be useful for the near term.

The other advantage of this method is its ability to detect configurations whose port-adjacency matrices may not match although they are isomorphic. Thus, it can detect kinematically equivalent configurations where the port numbers corresponding to connection between a pair of modules in the two configurations may not be the same. This is advantageous even in situations where the robots are put together simply manually since it spares the designer the tedious task of specifying the correct orientations for each module and modifying the gaits accordingly.

4. Spectral Decomposition

The next approach we consider applies basic concepts from spectral graph theory as a means for determining configuration isomorphism and label mapping. A known method for checking for isomorphism between two graphs is through adjacency matrix spectral decomposition (Chung 1997). That is, if two graphs G_1 and G_2 are isomorphic, then the eigenvalues of the corresponding port-adjacency matrices A_1 and A_2 are equal. The inverse, however, is not always true: adjacency matrices with identical arrays of eigenvalues are not necessarily isomorphic. To deal with this issue, permutations of eigenvector elements are employed as a confirmation of isomorphism and as a basis for finding the permutation mapping of module IDs.

The other methods considered in this paper use heuristics to search for matches between two graphs. This approach differs in its use of well-established ideas in spectral graph theory and its applicability to generate approximate methods (Zavlanos and Pappas 2006) where the other techniques may fail.

Cospectral graphs such as the pair shown in Figure 11 (Hogben 2005) are rare and interesting cases that may arise in modular robotic configurations. The characteristic polynomial for these graphs are the same despite non-isomorphism. In these scenarios, although the eigenvalues are the same, the structures are not isomorphic since no relabeling of nodes maps one configuration to the other. A comparison of the eigenvectors for these graphs is required to find that permutation does not exist and confirm non-isomorphism.

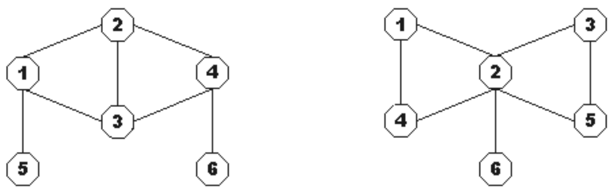

Fig. 11. Example of cospectral graphs with the same characteristic polynomial: $-1 + 4\lambda + 7\lambda^2 - 4\lambda^3 - 7\lambda^4 + 6\lambda^6$.

4.1. Linear Algebra of Adjacency and Permutation Matrices

Given two port-adjacency matrices A_1 and A_2 with the same graph spectrum, we wish to find the *permutation matrix* P that reorders the rows and columns of A_1 so that they are identical to those of A_2:

$$A_2 = PA_1P^{-1}. \quad (1)$$

Note that P swaps the rows and P^{-1} swaps the columns of A_1 so that gaits for A_1 can be mapped onto corresponding gaits for A_2. The permutation matrix is composed of only one 1 across any row and column with the remaining entries as 0's. This gives the property that all permutation matrices are orthogonal, satisfying $P^T = P^{-1}$. The identity matrix is a permutation matrix that maps a configuration onto itself.

If A_1 and A_2 are decomposed into their Jordan canonical forms

$$\begin{aligned} A_1 &= Q_1\Lambda Q_1^{-1}, \\ A_2 &= Q_2\Lambda Q_2^{-1}, \end{aligned}$$

where Λ is the diagonal eigenvalue matrix (note that they are the same for both since A_1 and A_2 correspond to isomorphic graphs as the rows and columns are just interchanged) and Q_1 and Q_2 are the associated eigenvector matrices. This gives

$$Q_2\Lambda Q_2^{-1} = PQ_1\Lambda Q_1^{-1}P^{-1} = (PQ_1)\Lambda(PQ_1)^{-1},$$

which reduces to

$$Q_2 = PQ_1. \quad (2)$$

This shows that the permutation matrix also relates eigenvector elements of adjacency matrices of isomorphic configurations. Therefore, by matching appropriate matrix elements in Q_2 to corresponding elements in Q_1, we can determine the permutation matrix that satisfies Equation 1. In the following section, we illustrate this procedure with an example.

4.2. Example of Finding the Permutation Matrix Between Isomorphic Configurations

Consider configuration *FL* from Figure 4. Recall that the port-adjacency matrix corresponding to this configuration is given by

$$A_{FL_c} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 6 \\ 1 & 0 & 0 & 7 & 6 & 0 & 0 \\ 0 & 1 & 7 & 0 & 4 & 0 & 0 \end{bmatrix}.$$

Now, consider another configuration with a port-adjacency matrix

$$A_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 1 & 7 & 0 & 4 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 7 & 1 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 6 & 0 & 0 & 4 & 0 \end{bmatrix}.$$

First, we note that A_{FL_c} and A_2 have the same characteristic polynomial:

$$\begin{aligned} \mathrm{Det}(A_{FL_c} - \lambda I) &= \mathrm{Det}(A_2 - \lambda I) \\ &= \lambda^7 - 76\lambda^5 + 868\lambda^3. \end{aligned}$$

This property suggests that these configurations are likely candidates for being isomorphic. To confirm this suggestion (and rule out that these structures are cospectral), we proceed further to find a permutation matrix that satisfies the property in Equation 2. We first compute the eigenvector matrices with columns ordered according to the roots of the characteristic polynomial (eigenvalue graph spectrum):

$$\lambda = \begin{bmatrix} 7.87 & 3.74 & -3.74 & -7.87 & 0 & 0 & 0 \end{bmatrix},$$

$$Q_1 = \begin{bmatrix} -0.47 & -0.72 & 0.72 & -0.47 & -0.97 & 0.32 & -0.58 \\ -0.31 & 0.48 & -0.48 & -0.31 & -0.18 & -0.93 & 0.12 \\ -0.04 & 0.06 & -0.69 & -0.04 & -0.02 & 0.11 & -0.35 \\ -0.06 & -0.10 & 0.10 & -0.06 & 0.06 & -0.07 & -0.42 \\ -0.53 & 0 & 0 & -0.53 & 0.09 & 0.02 & 0.58 \\ -0.52 & -0.38 & 0.38 & 0.52 & 0 & 0 & 0 \\ -0.34 & -0.25 & 0.25 & 0.34 & 0 & 0 & 0 \end{bmatrix},$$

$$Q_2 = \begin{bmatrix} -0.06 & -0.10 & 0.10 & -0.06 & 0.06 & -0.07 & -0.42 \\ -0.47 & 0.72 & -0.72 & 0.47 & -0.91 & -0.83 & 0.83 \\ -0.34 & -0.25 & -0.25 & -0.34 & 0 & 0 & 0 \\ -0.31 & -0.48 & 0.48 & 0.31 & 0.38 & 0.41+0.11i & 0.41+0.11i \\ -0.04 & -0.06 & 0.06 & 0.04 & -0.01 & -0.19+0.04i & -0.19+0.04i \\ -0.52 & 0.38 & 0.38 & -0.52 & 0 & 0 & 0 \\ -0.53 & 0 & 0 & 0.53 & -0.078 & 0.23-0.10i & 0.23-0.10i \end{bmatrix}.$$

Note that the columns of Q_1 and Q_2 (the eigenvectors of A_{FL_c} and A_2) are both ordered so that they correspond to the same eigenvalue elements. The columns have been normalized so that the sum of the squares down any column equals one. Also, note that the absolute value of each eigenvalue element is of interest, since eigenvectors, as a whole, can be scaled by a minus sign (a vector pointing in the opposite direction) with the eigenvalues and similarity properties of the matrix unchanged. To create the permutation matrix, note that Q_{2ij} (the element in the i^{th} row and the j^{th} column of Q_2) can be written as

$$Q_{2ij} = P_{i1}Q_{11j} + P_{i2}Q_{12j} + \cdots + P_{i7}Q_{17j}.$$

The property that P has a single 1 across any column or row (with all other elements zero) allows us to build P simply by comparing the permutation of elements down corresponding columns in Q_1 and Q_2.

For instance, we see that $|Q_{111}| = |Q_{221}| = 0.47$. Consequently, $P_{21} = 1$ with all other elements in the rank and file of P_{21} equal to zero. Next, observe that $|Q_{121}| = |Q_{241}| = 0.31$. This gives us $P_{42} = 1$ with all other elements in the rank and file of P_{42} equal to 0. Similarly, $|Q_{131}| = |Q_{251}| = 0.04$ giving $P_{53} = 1$ with all other elements in the rank and file of P_{53} equal to 0. Continuing down the first columns of Q_1 and Q_2, we can construct the following P that confirms isomorphism and gives the desired module labels:

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

The mapping is given by

$$\pi_{1\to 2} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 4 & 5 & 1 & 7 & 6 & 3 \end{pmatrix}.$$

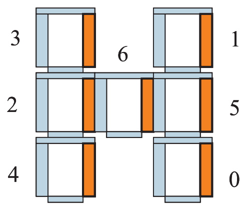

Fig. 12. Relabeling of A_{FL_c}.

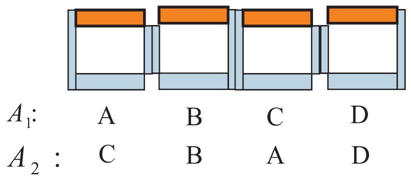

Fig. 13. Example of a symmetric configuration with two permutation matrices that relabel modules in the same way.

Generally, for a column k, when $|Q_{1jk}| = |Q_{2ik}|$, $P_{ij} = 1$ with all other elements across row i and down column j zero. The relabeled graph is shown in Figure 12.

4.3. Graph Symmetry and Configuration Mapping

It is evident that the choice of eigenvector for comparison is important. In the above example, if we had chosen any of the eigenvectors associated with the degenerate eigenvalue (zero), we would not have been able to build the permutation matrix. This redundancy in eigenvalues can be attributed to an algebraic regularity in the graph structure (Spielman 1996).

Structural symmetry creates interesting scenarios for this method to find the graph isomorphism mapping. Consider the configurations in Figure 13.

The two rows of labels give the adjacency matrices

$$A_1 = \begin{bmatrix} 0 & 7 & 0 & 0 \\ 7 & 0 & 1 & 0 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 7 & 0 \end{bmatrix},$$

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 7 \\ 1 & 0 & 7 & 0 \\ 0 & 7 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{bmatrix},$$

with the following eigensystem:

$$\lambda = \begin{bmatrix} -7.52 & -6.52 & 6.52 & 7.52 \end{bmatrix},$$

$$Q_1 = \begin{bmatrix} -0.48 & 0.52 & -0.52 & 0.48 \\ 0.52 & -0.48 & -0.48 & 0.52 \\ -0.52 & -0.48 & 0.48 & 0.52 \\ 0.48 & 0.52 & 0.52 & 0.48 \end{bmatrix},$$

$$Q_2 = \begin{bmatrix} 0.52 & -0.48 & 0.48 & -0.52 \\ -0.52 & -0.48 & -0.48 & -0.52 \\ 0.48 & 0.52 & -0.52 & -0.48 \\ -0.48 & 0.52 & 0.52 & -0.48 \end{bmatrix}.$$

Following the same approach as above, we end up with the following permutation matrix:

$$P^* = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}.$$

The multiple 1's across the rows and columns occur because of the redundant elements in the eigenvectors. This redundancy is because two distinct permutation matrices both satisfy Equation 1, namely:

$$A_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 7 & 0 & 0 \\ 7 & 0 & 1 & 0 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 7 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 7 & 0 & 0 \\ 7 & 0 & 1 & 0 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 7 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where P^* is the union of P_1 and P_2 given by

$$P_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$P_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

The reason two permutations occur for this configuration (and any isomorphic labeling of this graph) is because both matrices are in the graph's symmetry group. Also called the automorphic group (as discussed earlier in Section 3), we see that the symmetry is a 180° rotation about the center of mass of the system. For the purpose of the spectral decomposition approach, the algorithm tries valid combinations of P^* type unions (only one 1 in all rows and columns) until Equation 1 is satisfied. Therefore, this algorithm is slightly faster for asymmetric systems, as seen in Figure 16 which shows that the time to map random structures is, on average, less than the time for the more ordered structures (tree, snake, plane, centipede). The general algorithm is presented in Algorithm 2. Note that m is the size of the block redundancy in P^*; the snake example above has two blocks of two.

Algorithm 2 Configuration matching using spectral decomposition.

```
for $i = 1$ to $library\ max$ do
  if Det($A_{\text{given}} - \lambda I$) = Det($A_i - \lambda I$), then
    Compute $Q_{\text{given}}$ and sort (in increasing order of corresponding eigenvalue) to match format of $Q_i$.
    for $j = 1$ to $n$ do
      Compare the elements of columns $j$ in $Q_{\text{given}}$ with $Q_i$.
      Record $w$ as the column that produces a $P$ with the minimum number of redundant ones.
      if Any $P$ satisfies $A_i = P A_{\text{given}} P^{-1}$, then
        RETURN $P$.
      end if
    end for
  else
    Construct $P^*$ using column $w$.
    for $j = 1$ to $m$ do
      if Any $P^*(j)$ satisfies $A_i = P^*(j) A_{\text{given}} P^{*-1}(j)$, then
        RETURN $P = P^*(j)$.
      end if
    end for
  end if
end for
```

4.4. Complexity of the Spectral Decomposition Method

Once a characteristic polynomial match is found, the complexity of determining P or P^* (the first if-statement in Algorithm 2) is $O(n^2)$ since the most computationally expensive loop in this step is the double loop of matching values in eigenvectors corresponding to the same eigenvector. For completely random and asymmetric structures (e.g. an arm-like robot, a head-to-tail snake, any linear or tree-like structure that is intended for forward and turning motion only), this determines P, and is consequently the expected running time to find the permutation mapping between isomorphic configurations.

For structures with at least one line of symmetry, the main if-block determines a P^* that has a P embedded within it. These structures, such as a dog with a head and tail, a head-to-head-tail-to-tail snake (Figure 13), a bipedal walking robot, etc., delve into the primary else-statement of Algorithm 2 to find the correct P amongst the $m!$ choices in P^*. For the one-line-of-symmetry structures, $m = 2$, and it is evident that the complexity to find P in these cases is $O(2n)$. In general, the complexity to find P is $O(m!n)$. In most cases, m is small or at most a moderate fraction of n. In the extreme case of each module having complete symmetry with respect to another module (imagine a torus composed of CKBot modules, with each module having exactly four neighbors), $m = n$ and the algorithm reduces to the naive case of trying all possible labels for each module. But this case is pathological; m is usually a fraction of n. For example, a centipede structure with four identical four-module segments, $m = 5$ (left/right symmetry plus four segment interchangeable symmetries) and $n = 20$.

It is certainly possible to divide the computation of reducing P^* to P amongst parallel processors. We are currently exploring a hierarchical architecture where one central processor supervises many others.

4.5. Results for Spectral Decomposition Method

Figure 14 shows the time to find a match using comparisons between graph spectra in a library of 200 random configurations (for each data point), for up to 1,000 modules. The Matlab function $eigs(A)$ was used to find the largest eigenvalues of the sparse matrices. In comparison with Figure 16, we see that the time to find the module mapping is more time-consuming. This figure compares the times to find the permutation matrix between two isomorphic configurations

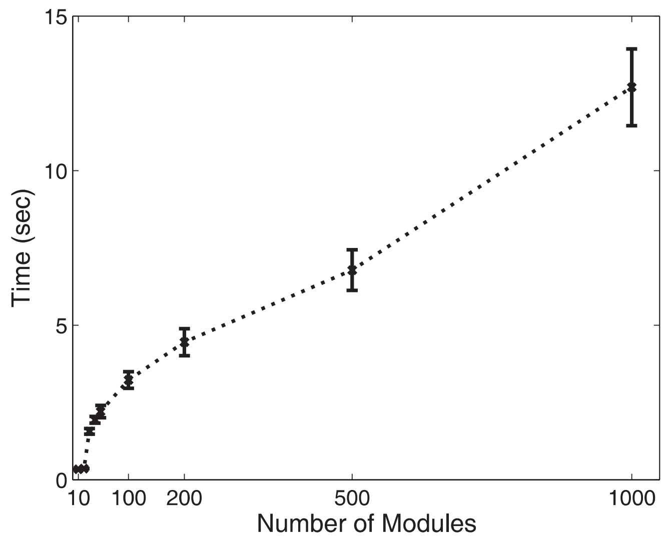

Fig. 14. Search times for the spectral-based algorithm versus number of modules in a random robot configuration.

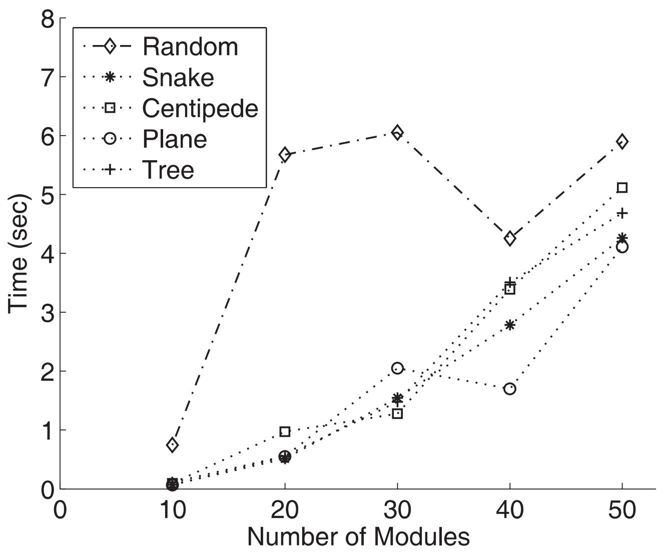

Fig. 15. Standard deviation of search times for the spectral-based algorithm versus number of modules in the robot configuration.

for up to 50 modules. Snake, centipede, plane, tree and random structures were tested. The random configuration times also include a negligible matching time to find the correct structure in a library of other random structures. Even so, the cumulative time to find the mapping is slightly less than the other structures. This can be attributed to the fact that there are less symmetries (on average) in random structures and this reduces the number of redundant permutation elements that the algorithm must choose from in the method described above. For comparison, the brute-force $n!$ time is included for up to 12 modules. The data point itself is off the scale of the graph.

Some limiting considerations in this approach include numerical stability and structural symmetry of configurations. In particular, the number of elements of each eigenvector equals the number of modules. For large matrices, these normalized eigenvectors are composed of small numbers that have accumulated rounding errors, attributed to the LU-decomposition approach (Matlab's LAPACK matrix algebra package). There are various methods that one can implement to deal with this issue (i.e. large normalizing factors, matrix balancing) but for numbers larger than 10^3 the problem becomes difficult to handle. Additionally, the time to compute eigenvectors becomes prohibitively large at that scale.

5. 3DLL Approach

The approach presented in Section 3 attempts to use the underlying graph structure of a modular robot to match new configurations to the database. However, it is evident that there is more information in our problem that could be used to further reduce the search space. In particular, the port-adjacency

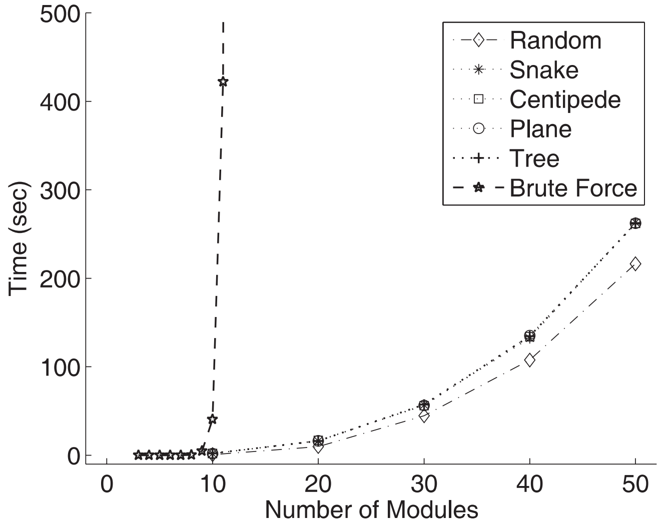

Fig. 16. Mapping times for specific configurations versus number of modules in the configuration.

matrix stores additional information that can be easily incorporated into a heuristic approach.

Just as in Section 3, where unique canonical forms of the automorphic group configurations in the library are precomputed, this method precomputes and stores a representation that exploits the physical nature of the modular robot configurations to generate nearly unique forms of the automorphic groups.

In this method, robots are represented by a three-dimensional linked list of module objects (thus the name 3DLL), each of which contains a position and orientation in 3-space with respect to an *expansion origin* module. The implementation presented here uses cubic modules to match the hardware, but could be extended to include non-cubic modules. Each configuration can have multiple representations based on which expansion origin module is chosen. Expansion origins are distinguished from *comparison origins*, which are a subset of the potential expansion origins used as starting points when comparing configurations module-by-module. Two representations of the same configuration starting from two different expansion origins are related by a constant translation and rotation relating the two origins in a global reference frame.

To compare two configurations, several heuristic measures are evaluated first. If these fail to distinguish the two configurations, the algorithm proceeds by comparing the positions, orientations and connection presences of each module in the configurations, starting from each comparison origin.

5.1. Library Creation

The first step in this approach is to create a representation of any new configuration. To do this, an expansion origin module must be selected. The selection process trims the number of possible expansion origin modules by allowing only modules with the degree that has the lowest multiplicity and then by giving priority to modules with the lowest combination of port values (i.e. an origin with connections on ports 1 and 3 takes priority over an origin with connections on ports 2 and 4). If more than one expansion origin remains after this process, a module is randomly chosen from this set and designated as the expansion origin. The goal here is to restrict the number of possible origin modules as much as possible to limit the amount of time that must be spent comparing configurations.

The next step assigns the origin a default position and orientation, and stores it in a three-dimensional linked list. New modules are added in a depth-first fashion by choosing the unplaced module connected to the lowest port number of the most recently placed module. At each step, the position and orientation of the next module is determined from the neighbor data, position and orientation of the current module. A global list of modules that have been added is maintained to prevent loops from running indefinitely and to end the expansion when all the modules have been added. This expansion process is deterministic: it will always generate a representation which stores the modules in the same order when the choice of origin module is constant.

During the expansion process, several metrics are calculated for later use in comparing configurations. These include the center of mass of the system, degree multiplicity and a port count (the total number of connections on each port). The final step determines the comparison origins of the configuration, which include only those expansion origin modules with the minimum distance to the center of mass. This minimum distance is stored for use as another heuristic check.

5.2. Matching and Mapping

Two identical configurations (i.e. configurations having the same geometrical shape, but not necessarily the same module IDs) with the same choice of expansion origin module will have three-dimensional linked lists of modules with the same positions, orientations and connection presences. Therefore, comparing two representations is just a matter of stepping through the linked lists representing them and checking that these attributes match for each module in the list. The two representations are not the same if any two corresponding modules of the linked list do not match at any point. Note that this comparison has to be carried out in turn for each possible choice of origin module for one of the configurations until a match has been found or the set of origin modules is exhausted. Finding a match also finds the mapping between the two configurations since this is a simple assignment of node IDs stored in the linked lists for the two configurations.

To compare configurations, first several fast heuristics are evaluated in order of speed. These include graph invariants such as the number of modules in a configuration and the multiplicity of degrees of the configuration. Note that, because of the special nature of our robots, the maximum degree is four. If two configurations pass this first set of heuristic comparisons, then the linked-list-based representations of the two robots are compared starting from the comparison origin modules. The process of comparison of two configurations is formally presented in Algorithm 3.

5.3. Example

We illustrate this method further using the example considered earlier in Section 3 using configuration *FL* from Figure 4 with different module IDs. Module 4 is the only module with degree four and is thus chosen as the expansion origin. The next module for addition to the linked list is 5, since it is attached to port 2 of the origin. Now, module 5 is expanded to obtain, in turn, modules 0 and 3. At each step, the relative position and orientation of each module with respect to the previous module is determined by the ports that attach the two modules together. The expansion process terminates when modules 6, 1 and 2 have been added, in turn, to the linked list representing the configuration. This procedure is shown in Figure 17.

The end result is the three-dimensional linked list representation of the robot configuration. This object includes the positions and orientations of the modules as well as information used in the heuristics such as the center of mass, number of comparison origins, degree multiplicity (i.e. the number of connections on port *i* throughout the robot) and distance of comparison origins to center of mass. These robot statistics are trivial to compute while the three-dimensional linked list is built, and are then available for fast comparison between two representations. If any of the statistics between two representations do not match, then the two configurations are not the

Algorithm 3 3DLL configuration matching algorithm for a new configuration G_1 and a library representation G_2.

```
Match = 0
if $numModules(G_1)$ != $numModules(G_2)$ then
    RETURN
end if
$R_1 = buildRepresentation(G_1)$
$R_2$ is loaded from memory.
if portCount($R_1$) != portCount($R_2$) then
    RETURN
end if
if $degreeMultiplicity(R_1)$ != $degreeMultiplicity(R_2)$ then
    RETURN
end if
$n_{co1} = numComparisonOrigins(R_1)$
$n_{co2} = numComparisonOrigins(R_2)$
if $n_{co1}$ != $n_{co2}$ then
    RETURN
end if
if $distToCenter(n_{co1})$ - $distToCenter(n_{co2}) > \epsilon$ then
    RETURN
end if
Match = 1
for $i = 1$ to $n_{co1}$ do
    Starting with comparison origin $i$, step through the modules one by one and record the module ID mapping.
    if module positions, orientations or connection presences do not match at any point then
        Match = 0
        CONTINUE
    else
        BREAK
    end if
end for
RETURN
```

same (see Algorithm 3). Storage space required for representation objects is linear in the number of modules.

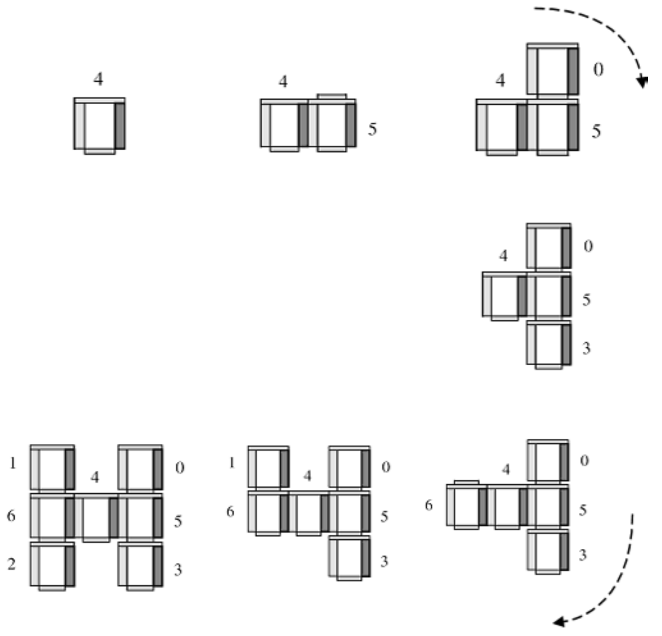

Fig. 17. Building a linked-list-based representation for a modular robot.

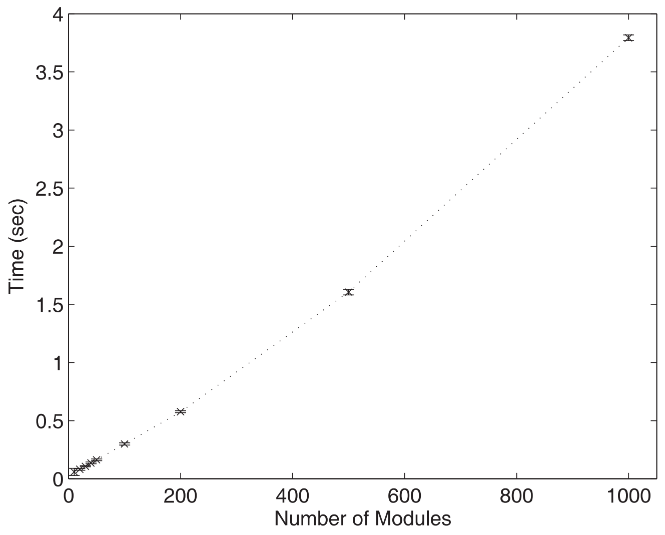

Fig. 18. Mean search times for 3DLL versus the number of modules in the robot configuration.

5.4. Results

Tests were carried out for this approach using exactly the same procedure as the one used for the previous two approaches in Sections 3 and 4. Results are presented here for both sets of tests, i.e. the test for configurations with different numbers of modules against a database of known configurations (Figures 18 and 19) and the mapping test for specific robot configurations where they are mapped onto the same configuration with a different set of node IDs (Figure 20).

Note that, for all five different configuration types, the scaling is nearly linear and the maximum amount of time is roughly four seconds. The time to search a library of 200 configurations and generate a mapping is nearly the same as the amount of time to just generate the mapping. While the standard deviation of the library test times is that of the times to find the 10th, 20th, ..., 200th configuration in the library, it is still similar to the mapping test standard deviations.

The mapping times for this algorithm are in general O($n \times n_{co}$) where n_{co} is the number of comparison origins. However,

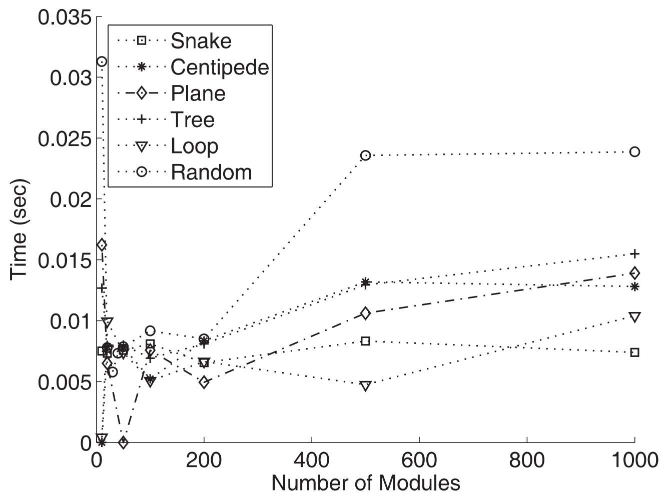

Fig. 19. Standard deviation of search times for 3DLL versus the number of modules in the robot configuration.

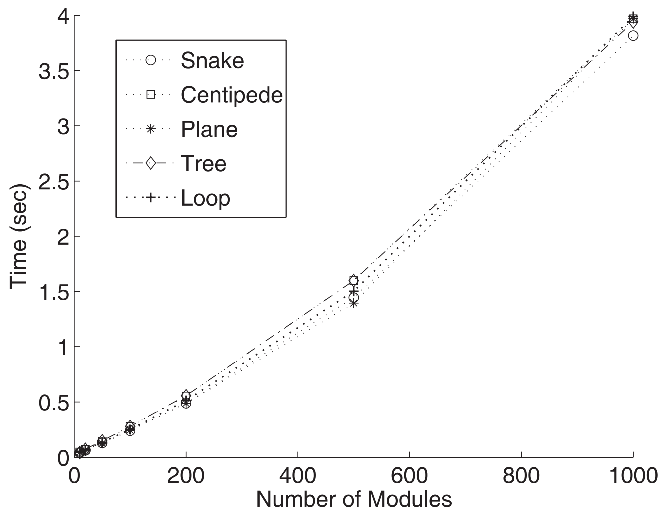

Fig. 20. Mapping times for specific configurations using 3DLL versus the number of modules in the configuration.

as can be seen from Figure 20, mapping times are in practice nearly O(n). Because of the constraint that origins must have the lowest multiplicity of degree, the number of origins is constant for the snakes, planes, and centipedes, and therefore the results scale linearly with the number of modules in the configuration. Similarly, because of the constraint that all comparison origins must have the minimum distance to the center of mass, the loops have only two possible comparison origins. The trees, while having anywhere from 1 to 63 modules with lowest multiplicity of degree, have only one possible comparison origin and therefore it too scales as O(n).

In practice, the heuristics filter out the majority of the configurations in the library, and step-by-step module comparison occurs primarily for identical configurations, i.e. the heuristics reduce the search space in the library to exactly one. Furthermore, the number of potential comparison origins (those that must have the minimum distance to the center of mass) is nearly always one or two. These observations explain the relatively low standard deviation and the linearity of the library test results – the index of the matching configuration in the library is relatively unimportant because the majority of the time is spent on building the representation of the new configuration and stepping through the modules of the new and matching configuration.

This method is able to recognize configurations in an amount of time that is reasonable even for configurations that are currently physically unrealizable because of the number of modules required. Furthermore, it does so in a scalable way thanks to the extra information inherent in the port-adjacency matrix. Finally, because robot representations are stored in a form analogous to the actual robot, this method allows the inclusion of other robot features, such as different module types. They are simply added on as another check in the module-by-module comparisons in Algorithm 3.

6. Discussion

The configuration matching and mapping problem can be presented as a task involving four steps:

1. Match the underlying graph, independent of ports.
2. Match the ports.
3. Choose an origin with which to start the mapping process.
4. Generate a module ID mapping between the matched configurations.

In the *nauty*-based method Step 3 is not needed as the canonical form includes the same origin for both the new and matched configuration. The first and last steps are fast and linear in the number of modules. However, Step 2 (matching ports) depends on the number of configurations in the automorphism group of the underlying structure which can sometimes be very large. As a library becomes larger, the size of the automorphism group can be exponential in the number of modules in a configuration worst case. As can be seen comparing Figure 7 with Figures 18 and 14, the *nauty*-based method runs faster for lower numbers of modules ($\leq$ 50 modules). Some of this speed may be due to the implementation since the core *nauty* routines are compiled C rather than interpreted Matlab code for the other two tests.

In the 3DLL method Steps 1, 2 and 4 occur at the same time. Just as symmetries can cause the automorphic group to become large for the *nauty*-based methods, symmetries in configurations can cause ambiguities in finding an origin to start the matching process. However, in practice the center-of-mass heuristic works very well in reducing the candidate origins to one or two, making this method very scalable. It should be noted that this method cannot recognize functionally identical configurations – that is, configurations with different module orientations but identical orientations of the axis of rotation for each module. This is because the port count heuristic is not invariant under functionally similar configurations. In the future this could be replaced with a different heuristic, e.g. a count of the number of modules in each functional orientation rather than a count of the port connections.

While both the *nauty*-based method and the spectral decomposition method are very general and easily applied to any self-reconfiguring system, the 3DLL method is specific to CKBot and cube-oriented modules.

In the spectral decomposition method Steps 1 and 2 happen concurrently and quickly. For the Steps 3 and 4, symmetries in the configuration can lead to redundant eigenvector elements which require explicit disambiguation. This process can take a very long time.

Both the *nauty*-based and 3DLL approaches essentially precompute an approximate canonical form for the elements in the library. *nauty* computes a canonical form with out the port information, while the 3DLL method generates a representation that is very likely to be unique, exploiting the physical properties of configurations. This precomputation saves computation time with a nominal cost in memory – about half a megabyte for a 1,000-module robot in the 3DLL representation and roughly the same for the *nauty*-based method when stored as a sparse array. The spectral decomposition method does no precomputation, working directly with adjacency matrices and was thus easy to implement on embedded controllers.

6.1. Hardware Demonstration

As a preliminary test, isomorphic gait control using graph spectra has been implemented on CKBot configurations containing up to seven modules. A centralized controller containing 42 configurations with corresponding gaits ran a configuration detection scheme using communication architecture described in Section 2.

The centralized controller compares the graph spectra of port-adjacency matrices to see if a given structure is in the configuration library. The method incorporated to find the eigenvalues is based on the QR algorithm. If a match in the library is found, up to $n!$ permutations of node labeling schemes are tried until the exact mapping between structures is found. While the issue of label mapping was done by brute force, using graph spectra to find configuration matches reduced time noticeably over earlier schemes that did not use this information (this is to be expected since comparing an array of eigenvalues saves time, on average, compared with trying $n!$ permutations for all library configurations until an exact match is found). Prohibitively slow speeds for structures containing eight or more modules is a motivating factor for the work presented here. Configuration dependent gaits include those for snake, slinky, walking, rolling, lurching and turning motions.

While the simplicity of the graph spectra method allowed it to be implemented on the small embedded PIC controllers, converting the *nauty* and 3DLL-based methods to run on small memory (e.g. 32K) embedded systems is future work.

7. Conclusion

A comparison of the results for the three methods reveals the relative advantages and disadvantages of the three approaches. The spectral decomposition represents the most mathematically elegant of the three techniques but suffers from numerical issues. In addition, calculating the spectra, especially for larger numbers of modules, is a computationally expensive process. However, this technique works very well in the recognition problem, i.e. it can be used to quickly identify the configuration in the database. The majority of the computational time is spent finding the isomorphism *mapping*.

The traditional graph isomorphism comparison method benefits from the ability to create a canonical configuration which pares the size of the search space for further matching. In addition, since it initially compares the underlying graph structure, it can match robot configurations whose port-adjacency matrices may differ by symmetric rotations of the individual modules.

The 3DLL method using linked lists uses the extra information inherent in the port-adjacency matrix but suffers from the need to run through every configuration in the library at runtime.

The ideal choice of configuration recognition algorithm depends on the specific problem being solved. Where functionally identical configurations must be recognized, either the *nauty*-based approach or a modified 3DLL algorithm is best. Exact configuration recognition is fastest and most scalable using 3DLL. Spectral decomposition may be suitable for finding an approximate match to a new configuration. In the future, we plan to examine extensions to these three approaches and how best to combine the algorithms to achieve higher recognition speeds. Furthermore, we plan to modify these approaches to recognize heterogeneous modular robots and functionally identical configurations.

References

Butler, Z., Fitch, R., Rus, D. and Wang, Y. (2002). Distributed goal recognition algorithms for modular robots. *Proceeding of the IEEE International Conference on Robotics and Automation*, Washington, D.C., pp. 110–116.

Castano, A., Shen, W. M. and Will, P. (2000). CONRO: Towards deployable robots with inter-robots metamorphic capabilities. *Autonomous Robots*, **8**(3): 309–324.

Castano, A. and Will, P. (2001). Representing and discovering the configuration of CONRO robots. *Proceedings of the IEEE International Conference on Robotics and Automation*, Seoul, Korea, vol. 4, pp. 3503–3509.

Chen, I. M. and Burdick, J. (1996). Enumerating the non-isomorphic assembly configurations of a modular robotic system. *International Journal of Robotics Research*, **17**(7): 702–719.

Chirikjian, G. S. (1994). Kinematics of a metamorphic robotic system. *Proceedings of the IEEE International Conference on Robotics and Automation*, San Diego, CA, pp. 449–455.

Chung, F. R. K. (1997). *Spectral Graph Theory*. Providence, RI, American Mathematical Society.

Fitch, R., Rus, D. and Vona, M. (2000). A basis for self-repair robots using self-reconfiguring crystal modules. *Proceedings of the IEEE International Autonomous Systems 6 (IAS-6)*, Venice, Italy.

Fukuda, T. and Kawauchi, Y. (1990). Cellular robotic system (CEBOT) as one of the realization of self-organizing intelligent universal manipulator. *Proceedings of the IEEE International Conference on Robotics and Automation*, Cincinnati, OH, pp. 662–667.

Gomez-Ibanez, D. et al. (2004). The robotics bus: a local communications bus for robots. *Proceedings of the International Society of Optical Engineering*, vol. 5609.

Hogben, L. (2005). Spectral graph theory and the inverse eigenvalue problem of a graph. *Journal of the International Linear Algebra Society*, **14**:12–31.

Jorgensen, M. W., Ostergaard, E. H. and Lund, H. H. (2004). Modular ATRON: modules for a self-reconfigurable robot. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2068–2073.

Kotay, K., Rus, D., Vona, M. and McGray, C. (1998). The self-reconfiguring robotic molecule: design and control algorithms. *Workshop on Algorithmic Foundations of Robotics (WAFR)*, Houston, Texas, Agarwal, P. et al. (eds).

McKay, B. D. (1981). Practical graph isomorphism. *Congressus Numerantium*, **30**: 45–87, available at http://cs.anu.edu.au/bdm/nauty/PGI/.

Murata, S. et al. (1998). A 3-D self-reconfigurable structure. *Proceedings of the IEEE International Conference on Robotics and Automation*, Leuven, Belgium, pp. 432–439.

Murata, S. et al. (2000). Hardware design of modular robotic system. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Takamatsu, Japan, pp. 2210–2217.

Rus, D. and Vona, M. (1999). Self-reconfiguration planning with compressible unit modules. *Proceedings of the IEEE International Conference on Robotics and Automation*, Detroit, MI, pp. 2513–2519.

Shen, W. M. et al. (2006). Multimode locomotion via superbot robots. *Proceedings of the IEEE International Conference on Robotics and Automation*, Orlando, FL, pp. 2252–2257.

Ünsal, C., Kýlýççöte, H. and Khosla, P. K. (1999). I(CES)-cubes: a modular self-reconfigurable bipartite robotic system. *Proceedings of SPIE, Sensor Fusion and Decentralized Control in Robotic Systems II*, Boston, MA, vol. 3839, pp. 258–269.

Yim, M., Duff, D. and Roufas, K. (2000). Polybot: a modular reconfigurable robot. *Proceedings of the IEEE International Conference on Robotics and Automation*, San Francisco, IL, vol. 1, pp. 514–520.

Yim, M., Homans S. and Roufas, K. (2001). Climbing with snake-like robots. *(Proceedings of the IFAC Workshop on Mobile Robot Technology)*, Jejudo, Korea.

Zavlanos, M. and Pappas, G. (2006). A dynamical systems approach to weighted graph matching. *Proceedings of the 45th IEEE Conference on Decision and Control*, San Diego, CA.