Fast exact graph matching using adjacency matrices

Marlon Etheredge
Amsterdam University of Applied Sciences
Duivendrechtsekade 36
Amsterdam, The Netherlands
marlon.etheredge@hva.nl

ABSTRACT
This paper introduces a technique of graph subgraph searching, that allows for varied complex subgraphs to be matched in directed or undirected target graphs in a fast and flexible manner. Along with a discussion on the contrast with other known algorithms, benchmarks are presented that compare these known algorithms to the algorithm that is presented in this paper.

Categories and Subject Descriptors
G.2.2 [**Discrete Mathematics**]: Graph Theory—Graph algorithms

General Terms
Algorithms

Keywords
Graph grammars, graph subsets, graph rewrite rules, procedural generation

1. INTRODUCTION
The demand for a fast and exact graph subset search algorithm comes from development of the game DroidRacers. DroidRacers is a multiplayer racing game with infinite procedurally generated race tracks that are constructed from graphs. The game is played on a public screen where many potential players will pass and allows players to play the game using their Android device as a controller. Players will join and leave the game while it is running. With the introduction of this feature within the game, the need for a fast mechanism of procedural content generation is required since the set of players within the game is adjusted when players enter or leave the game. The gamestate is adjusted by the experience level of the group as a whole. To tackle this problem the game makes use of so called mission graphs as described by Joris Dormans [3]. With the mission graphs, graph transformation is used to allow for adaptive gameplay, e.g. changing track layout depending on player skill level and constructing new track geometry upon completion of mission goals in the game. To achieve this altering of game content search and replace operations need to be performed on the game content, these known patterns are represented as subgraphs that are searched within a target graph (the gamestate, or part of the gamestate). This requires a fast and flexible technique for graph searching that does not affect performance in a realtime game environment. Many existing methods of subgraph searching do not meet the requirements for processing time of realtime game environments, causing poor performance or, when approximating, suboptimal matching. Also, existing methods that rely on evolutionary (genetic) algorithms do not meet the standard for the quick and interactive way of graph transformation in this game; changes should be introduced directly, without the need of evolution. Therefore a robust and fast technique of exact graph subgraph matching is indispensable.

1.1 Other related work
For a graph matching method we refer the reader to [3], where multiple appliances of graph grammars are described in the context of game design. An example of a graph search algorithm is discussed in [5]. Graph grammars and rewrite rules are related to querying graphs in graph databases. There is a long history of research and approaches concerning graph databases and querying graphs. Approximation is used for querying graphs in various approaches, like TALE [7]. Other approaches including GraphGrep[4] and GiS [6] make use of an index phase and filter phase, but due to the nature of realtime game environments we cannot afford to have to run these additional phases. As well as these existing tools many algorithms exist for solving the graph isomorphism problem, a well-known solution for this problem was introduced by Ullmann [9]. Ullmann's approach was later used in VF, as well as the VF algorithm for solving isomorphism in graphs [2], VF is used throughout this paper to test our algorithm against, both in the description of the algorithm as the benchmark (VF and Ullmann). There are known solutions for the generation of personalised race tracks as described in [1] and [8].

2. ADAPTIVE GAMEPLAY
DroidRacers allows players to enter and leave an ongoing game. This requires that the content of a live game is altered according to the state of the players that enter and leave the game. Race tracks may need to be altered and objectives within the game may need to be changed according to the new set of players. Previous research like [1] and [8] that fo-

cus on the generation of personalised racetracks make use of evolutionary (genetic) algorithms to achieve this. Since the states of DroidRacers need to be adjusted in realtime and we require that the game is modified without many evolutions, the evolutionary approach does not seem suitable for DroidRacers. In addition to this insuitability caused by the evolutions in genetic algorithms, the graph-based approach allows us to alter any form of content within the game; not only race tracks may be altered at runtime, but we also may introduce new objectives or we may adjust the game space. Practically every structure in the game may be represented by graphs and transformed according to certain events within the game. In other words, implementing a fast graph matching algorithm makes it possible to create an endless race game that constantly adopts its content to the changing set of players.

3. MATCHING

Let G be a graph that consists of the following properties:

- G_n as a set of nodes
- $G_e = (n_1, n_2)$ a set of 2-tuples elements as e edge
- $n_1 \in G_n$, $n_2 \in G_n$

The set G_e might be stored in different ways in an implementation, e.g. sets of nodes per node to store edges or, like proposed, adjacency matrices.

A graph will contain a set of node types, these node types are used to check similarity within graph and subgraph, and are independent of implementation. As long as the node types are comparable in an implementation, any definition for this node type would be correct. Node types may even be stored inside a node itself.

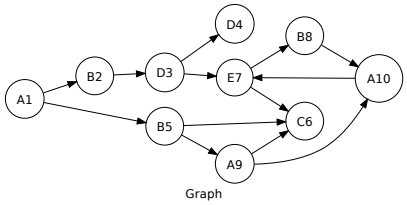

Figure 1: A target graph example

3.1 Search operation

Consider the target graph that is laid out in figure 1, one would want to search for a subgraph of node types in an arbitrary position in the graph and match all subgraphs that match in the target graph. The matching of this subgraph is done by selecting nodes that are accepted by the search operation and marking these nodes as such. An exact search algorithm must ensure that only nodes that are part of a subgraph used as a search criterium are selected. Another requirement for a successful search operation is to have exactly the same connections as the targeted graph.

Complexity of these requirements increase whenever the subgraph gets more complex, especially when the subgraph contains many deep branches. This is due to the fact that whenever deep branches are traversed, it becomes more complex over time to step back into nodes of the same type at the same level of a branch.

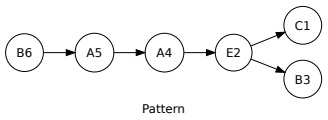

Figure 2: A subgraph example

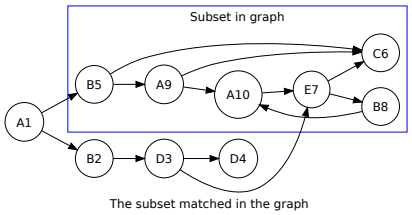

Figure 3: The subgraph matched in the targeted graph, showing the accepted nodes

It is clear that the subgraph that is matched that is shown in figure 3 contains all types that were required to match the subgraph in the target graph.

4. TRADITIONAL SEARCHING

An algorithm as described in [5] or as used in [3] will store the edges as sets of nodes. Describing a subgraph A by the following properties:

- A_n as a set of nodes existing in A
- A_{out} as a set of outgoing edges
- A_{in} as a set of incoming edges
- $A_{out} \subseteq A_n$, $A_{in} \subseteq A_n$

And a targeted graph G by the following properties and the previously described structure of edges:

- G_{out} as a set of outgoing edges
- G_{in} as a set of incoming edges
- $G_{out} \subseteq G_n$, $G_{in} \subseteq G_n$

An implementation of this traditional search algorithm would then recursively search every node in G_{out} for a node in A_{out} starting from every node in G_{out}. This would cause exponential processing time, unacceptable in a realtime environment as well as undesired levels of recursion that make it harder to search for more complex subgraphs.

5. OUR ALGORITHM

In contrast with the previously mentioned search algorithm, our algorithm uses adjacency matrices as a way to store and quickly search through edges. An adjacency matrix

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{bmatrix}$$

is constructed by a simple function (as seen in figure 4, having a as the adjacency matrix for the graph, C as a set of connections and l as the size of the matrix).

```
for (Connection connection : C) {
  pA = connection.A.Position;
  pB = connection.B.Position;

  a[pA][pB] = 1;

  ++a[pA][l - 1];
  ++a[l - 1][pB];
}
```

Figure 4: Simple adjacency construction function

In an implementation there should be storage for row and column size, it is trivial to store this data as the last element of a row or column, to provide fast access when iterating over rows and columns. These elements are later used to perform sorting operations on required sets prior to any search operations. The following adjacency matrix m is constructed from the graph as presented in figure 1. Rows can be seen as outgoing nodes, while columns are describing incoming nodes. 1 as an element states a connection, 0 states a non-existent connection.

$$\begin{array}{c} \\ A_1 \\ B_2 \\ D_3 \\ D_4 \\ B_5 \\ C_6 \\ E_7 \\ B_8 \\ A_9 \\ A_{10} \\ l \end{array} \begin{array}{c} \begin{array}{cccccccccc} A_1 & B_2 & D_3 & D_4 & B_5 & C_6 & E_7 & B_8 & A_9 & A_{10} & l \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 3 & 2 & 1 & 1 & 2 & 0 \end{pmatrix} \end{array}$$

The following adjacency matrix a can be constructed from the graph as displayed in figure 2. This matrix uses the same mapping as the previous matrix.

$$\begin{array}{c} \\ C_1 \\ E_2 \\ B_3 \\ A_4 \\ A_5 \\ B_6 \\ l \end{array} \begin{array}{c} \begin{array}{ccccccc} C_1 & E_2 & B_3 & A_4 & A_5 & B_6 & l \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

The matrix a is used to extract patterns of nodes in nodes that are used to scan m. When speaking of these patterns we define a pattern as a map containing a key and one or more value entries. The key of such a pattern 'pair' will describe the nodetype that requires the entries in the values of the map for the particular pattern. And will be extracted by a function similar to the pseudocode that is presented in figure 5, having P as a set of patterns, N as a set of nodes within a graph and m as the adjacency matrix of the graph. Please note that in the pseudocode the requirements set is implemented using bitmasking.

The adjacency matrices are used to test if edges exist for a specific node, iteration is over nodes and a lookup for node types in the collection of nodes is needed to test for requirements defined in the subgraph graph. This approach allows for quick searching since no traversal into node lists is needed whenever node types do not meet. Another benefit from this approach is that the pattern collection condenses while more matches are found, since these patterns are invalidated upon a match, search requirements in the form of patterns become more narrow.

```
P.clear();

for (int i = 0 ; i < N.size() ; ++i) {
  Pattern p;

  p.Key = N[i].Type;

  for (int j = 0 ; j < N.size() ; ++j) {
    if (D[i][j] == 1) {
      p.Requirements |= 1 << N[j].Type.GetId();

      ++p.RequirementsLength;
    }
  }

  P.push_back(p);
}

P.sort();
```

Figure 5: Pattern extraction pseudocode

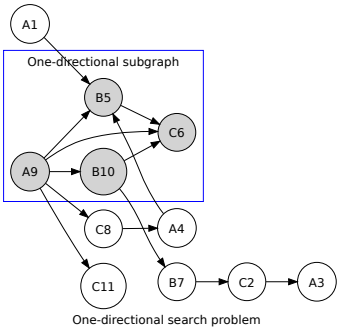

Figure 6: A one-directional subgraph in a target graph

5.1 Directional searching

Figure 6 shows an example of one-directional searching, where the direction is only outgoing seen from the entry-node A_9. This is a case of simple searching, where only the outgoing edges set should be searched for patterns.

Where figure 3 shows a case of bi-directional searching, where seen from entry-node E_7, the direction is outgoing to node C_6 and B_8 as well as incoming from node A_{10} through B_8. Required for such a case is that both incoming as outgoing edge sets are searched for patterns. This requires the pattern set to contain both a set for input edges as output edges. Since this case adds to the polynomial search complexity of this algorithm, additional optimizations might be used to pre-evaluate such a search condition.

An entry point for this algorithm with a graph G should be the first row $r = 0$ in the adjacency matrix of the graph that is targeted. It should continue iterating over every row

```
void Graph::SearchInMatches(O, M, P, A) {
  for (Node match : M) {
    for (Pattern pattern : P) {
      if (!A.Exists(match)) {
        if (pattern.key == match.Type) {
          SearchState s = SearchForPatterns(o,
          match.position, pattern, P, A);

          if (s == LEAF) {
            break;
          }
        }
      }
    }
  }
}
```

Figure 7: Search in matches function

```
void Graph::Search(O, R) {
  R.clear();

  P = O.GetPatterns();

  int i = 0;

  for (Node node : N) {
    SearchForPatterns(O, i++, P.front(), P, R);
  }
}
```

Figure 8: Search entry-point

until the last row $r = |G_n|$. The algorithm can be broken up in three parts as seen in figure 7, figure 9 and figure 8. Where the pseudocode presented in figure 8 can be seen as the entry-point to a search operation, supplying the subgraph as O and an empty set R used for search result storage.

6. FLEXIBILITY

A traditional search technique recursively traversing into branches is limited in flexibility. When sticking to the case of simple subgraph matching alone, there are cases where this method does not comply with the definition of exact graph matching.

As an example, a graph containing one or multiple branches containing equal node types as in figure 10 would be complex to search using the traditional search algorithm. The traditional algorithm would need to push both node 2 and 3 in a set S. Set S then needs to be traversed to make sure that both branches $A_1 \rightarrow B_2$ and $A_1 \rightarrow B_3$ are searched instead of only one of the two branches. In the example above, the loss in performance would be negligible, but imagining a case where this patterns is repeated several times, the complexity of the search algorithm would increase significantly over time. In any such case in a targeted graph, S will need to be constructed, stored and searched for a particular pattern at that level of depth.

When implementing this case within the traditional method, we must implement a maximum recursion depth. Not only

```
SearchState Graph::SearchForPatterns(O, r, p, P, A)
{
  Node *node = n[r];

  if (pattern.Key == node.Type) {
    list<Node> M;

    for (Node node : N) {
      if (!A.Exists(node)) {
        if (D[row][i]) {
          if (pattern.Requirements.Exits(node)) {
            M.push_back(node);
          }
        }
      }
    }

    if (M.size() >= pattern.ReqLength && M.size() > 0) {
      pattern.Key = NullNodeType();

      A.push_back(N[r]);

      SearchInMatches(O, M, P, A);

      return TRAVERSE;
    }
    else if (M.size() == 0 && pattern.ReqLength == 0) {
      pattern.Key = NullNodeType();

      A.push_back(N[r]);

      return LEAF;
    }
  }

  return NONE;
}
```

Figure 9: Search for patterns function

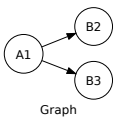

Figure 10: A graph example

to make sure that searching is always clamped in an implementation dependent processing frame, but also to make sure that the search operation is not getting stuck in an infinite loop.

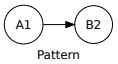

Figure 11: A subgraph example

When searching for a pattern like displayed in figure 11, every time a node of type A is discovered, every outgoing connection will need to be stored in S. S will then in turn need to be searched for pattern $A \rightarrow B$, prior to searching for other patterns; in this particular example none.

The nature of our algorithm allows for the matching of both branches by default. Figure 6 shows an example of one-

directional searching, to support bi-directional searching we only need to invalidate the pattern after processing every match.

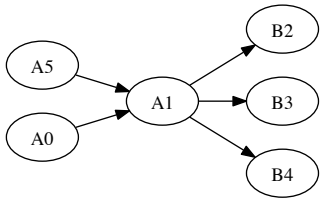

Figure 12: A subgraph example

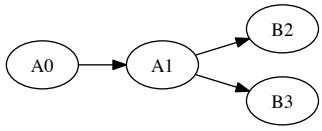

Figure 13: A subgraph example

Another example of a more complex matching case is given in figure 12 that should match the subgraph presented in figure 13. When running a search operation on this scenario the search algorithm should return two sets of affected nodes: $\{A5, A1, B2, B3, B4\}$ and $\{A0, A1, B2, B3, B4\}$. Our algorithm covers this case naturaly, as long as the set of affected nodes is stored with a relationship between the entry-node ($A5$) and all other nodes.

6.1 Advanced node types

When implementing more advanced node types like for instance:

Wildcard Node type that allows for any graph structure positioned at the wildcard node in a subgraph to be accepted by the search operation.

Exclusion Node type that accepts nodes that are not of the not-node type positioned at the not-node in a subgraph.

Times Node type that allows for any graph structure described by the structure in the subgraph positioned at the time node in a subgraph to be accepted by the search operation one or n times.

Using the traditional search algorithm to match these kinds of structures would be very complex and inefficient, for the same reason as described above.

Implementing these structures would be trivial and a case of implementation, only a specific equals-operator should be implemented for different node types.

7. BENCHMARKS

To test the processing time of both algorithms, an implementation of our algorithm is compared to both the VF2 algorithm [2] and Ullmann's algorithm [9], by using the VFLib with implementations of both algorithms. The benchmarking method that is chosen is to generate graphs from random adjacency matrices and match these graphs against a full subset of that target graph. The complexity of the graphs is then increased by introducing more nodes to the graphs. For all search algorithms only the search operation is timed, timing is done by using the standard C++ timing functions.

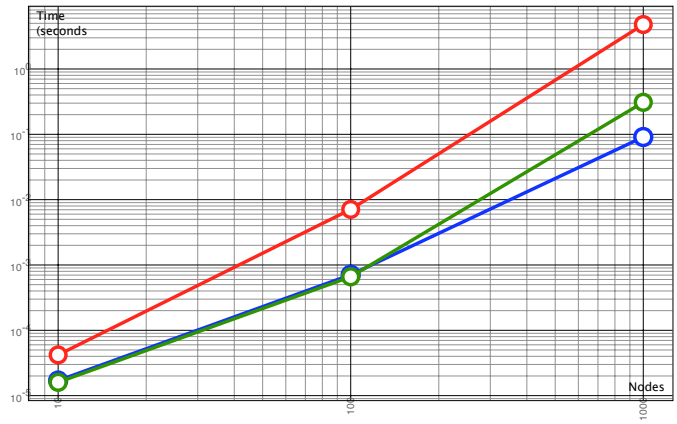

Figure 14: Graph showing the processing time of Ullmann's algorithm (red line), our algorithm (green line) and the VF2 algorithm (blue line).

Nodes	Time (our)	Time (VF2)	Time (Ullmann)
10	0.000016	0.000017	0.000042
100	0.000660	0.000707	0.007091
1000	0.309008	0.090885	4.792513

Along with the number of nodes, the number of connections are also increased with each increase of complexity, the following table shows the number of connections in relation with the number of nodes.

Nodes	Connections
10	35
100	3386
1000	370676

7.1 Test details

These benchmarks were performed on a system with a 2.4 GHz Intel Core 2 Duo CPU, 4GB 1067 MHz DD3 RAM, running Apple Mac OSX version 10.7. All implementations were written in C++, binaries were compiled using the Apple LLVM compiler version 3.0 without any optimizations. The VF2 and Ullmann algorithms were the optimized versions as presented in the VFLib library.

When we look at the results of the benchmark we can see that our algorithm performs better on low node counts and sticks close to the VF2 algorithm on high node counts, with a tipping point between graphs of 100 and 1000 nodes. Where the Ullmann implementation stays behind in processing time at low node counts as well as high node counts. Please note that both implementations used from the VFLib are highly

optimized, further optimization might be needed to our algorithm to reach even better results in sence of shortening processing time.

8. CONCLUSION

Subgraph searching in graphs allows to solve various problems in computer science in a neat and fast way and allows for search and replace operations to be performed. Traditional algorithms are limited in both efficiency and flexibility, causing suboptimal searching or performance. A search algorithm for subgraphs is required to allow for fast searching of such a subgraph in a target graph. Our algorithm offers a fast way of searching subgraphs in graphs, while still offering full flexibility. Opening new possibilities for graph grammars to allow for more complex node types to be searched for in graphs and thus allowing for more complex grammars to be created.

In our domain of game development this technique allows us to perform operations on graph-based game design structures in real time. This opens up possibilities to introduce new ways of procedurally generated games and even procedurally generated gameplay; game mechanics can be adjusted and introduced in-game. Allowing us to introduce new games with more dynamic gameplay.

9. ACKNOWLEDGMENTS

I would like to thank Joris Dormans, my mentor on the DroidRacers project, for his guidance throughout the project and being a true inspiration. As well as Jerrol Etheredge for his vision and making time for me to listen to my ideas on this paper and all other people that supported me throughout this research.

10. REFERENCES

[1] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *GECCO*, pages 395–402, 2011.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing*, ICIAP '99, pages 1172–, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Joris Dormans. *Engineering Emergence: Applied Theory for Game Design*. PhD thesis, Amsterdam University, 2012.

[4] R Giugno and D Shasha. Graphgrep: A fast and universal method for querying graphs. *Object recognition supported by user interaction for service robots*, 2(c):112–115, 2002.

[5] R. Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, February 2006.

[6] Praveen R Rao. A tool for fast indexing and querying of graphs. *Scenario*, pages 241–244, 2011.

[7] Yuanyuan Tian and Jignesh M. Patel. Tale: A tool for approximate large graph matching. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *ICDE*, pages 963–972. IEEE, 2008.

[8] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Towards automatic personalised content creation in racing games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007. *Togelius is working at IDSIA on SNF grant 21-113364 to J. Schmidhuber.*

[9] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.