Gridlock

Group 8-21

Benson Lin Zhan Li (4S101)

Galen Lee Qixiu (4S115)

Timothy Loh Yan Xun (4S119)

Contents

1. Introduction

Combinatorial games are often fascinating; complex games can be solved using simple algorithms or strategies. Here, we will consider the following combinatorial game, similar to that of Vertex Geography:

2 players, named P1 and P2 for simplicity, play a game on an $a \times b \times c$ cuboid. P1 starts off by writing the number "1" in a cell of their choice, P2 then chooses one empty cell adjacent to P1's choice, where 2 adjacent cells share a common face and writes "2" in it. This continues and alternates between players; each written number being greater than the previous by 1. When a player runs out of legal moves, the game ends and his opponent wins the game.

We will also consider multiple variations of this game with different graphs and work out the winning strategies.

1.1 Objectives

The objectives are as follows:

1. To determine the winner and the winning strategy of the proposed game
2. To consider other types of moves and their effect on the result and strategy
3. To generalise the game to special graphs and determine the winning strategy

1.2 Research Problems

The research problems are as follows:

1. In a cuboid of dimensions $a \times b \times c$, which player will win, and what is the winning strategy?
2. If we write consecutive numbers in a pattern similar to knight moves, what will the result be, and, if applicable, what is the winning strategy?
3. If the game is played on special graphs such as the generalised grid graph, rook graph or complete graph, what are the winning strategies?

1.3 Terminology

Term	Definition
Vertex	A vertex is the endpoint of an edge in a graph.
Edge	An edge defined as a pair of vertices (u, v).
Graph	A graph G made of a pair of sets (V, E), where V is the set of vertices in the graph while E is the set of edges.
Simple Path	A simple path $P = (e_1, e_2, \cdots, e_k), e_i \in E \; \forall i$ in a graph $G(V, E)$ is a sequence of edges in G such that the path drawn does not contain a cycle.
Matching	A matching in a graph $G(V, E)$ is a set of edges such that no 2 edges in M share a common endpoint.
Perfect Matching	A perfect matching M in a graph $G(V, E)$ is a matching in G such that every vertex is incident to an edge in M.
Augmenting Path	An augmenting path is a simple path $P = (e_1, e_2, \cdots, e_{2k}, e_{2k+1})$ in a graph G with a predefined matching M such that $e_{even} \in M$ and $e_{odd} \notin M$ with the first and last vertices of P not incident to any edge in M.

A more complete terminology can be found in Appendix 6.1.

1.4 Literature Review

In 2004, the Singapore Math Society magazine, Mathematical Medley, discussed a similar problem with a cube of side length N. In this variant, the first player selects the starting cell and moves immediately after that; it was proven with a checkerboard colouring that the first player always wins. Such colourings and invariants may be useful in the solving of our problem. However, our version only allows the first player to choose the starting point; they are not allowed to move immediately, affecting the optimal strategy for each player.

A 1996 research paper titled “Geography Played on Products of Directed Cycles” by Nowakowski and Poole showed an explicit strategy to solving a specific case of vertex geography on products of certain directed cycle graphs. The proof constructs the optimal strategy using special moves dubbed “closing off sequences”. It is possible to use this result on grids which are in fact products of path graphs. However, this only solves for special directed graphs, while our problem considers undirected graphs.

A 1993 research paper by Fraenkel, Scheinerman and Ullman, “Undirected Edge Geography”, gave a necessary and sufficient condition for a win by either player in vertex geography using maximal matchings in undirected graphs. This condition may assist in the proving of certain optimal strategies; however, the majority of the paper looks at edge geography, which is vastly different from vertex geography.

2. Methodology

Firstly, we conducted research on the techniques and ideas potentially applicable to our project, such as graph theory. Next, we use those techniques to devise solutions to our research questions. By applying our solutions to our first two Research Questions, we evaluated their reproducibility in more general graphs in the third Research Question.

3. Results

To solve our research problems, we will utilize the following 3 lemmas.

Lemma 1 (Berge's Theorem)

Let M be a maximal matching in a graph $G(V,E)$. Then there does not exist an augmenting path for G and M. (Berge, 1957)

Lemma 2

Suppose vertex geography is played on a graph $G(V,E)$. The second player wins iff there exists a perfect matching in G.

Lemma 3

Let G and H be 2 graphs such that there exists a subgraph S of H that is isomorphic to G and contains all the vertices of H. If there exists a perfect matching in G, there exists a perfect matching in H.

The proofs of these 3 lemmas can be found in Appendix 8.1.

The general strategy of the first player is to find a maximal matching and follow the edges in the maximal matching, while the general strategy for the second player is to find a perfect matching and follow the edges in the perfect matching.

3.1 Research Question 1

The first player's strategy on a cuboid with an odd number of cells is to split the cuboid into disjoint 1×2 dominos, leaving 1 cell not in any domino. The first player starts on this cell and moves from 1 cell in a domino to the other cell in the domino.

The second player's strategy on a cuboid with an even number of cells is similar; split the cuboid into disjoint 1×2 dominos. The second player can then move from 1 cell in a domino to the other cell in the same domino.

The proofs for the optimality of each strategy can be found in Appendix 6.3.

3.2 Research Question 2

We will assume without loss of generality that $b \geq c$ and that a is the smallest even number among the 3 sides if at least 1 of the 3 dimensions is even. The problem can then be split into 10 cases, which are summarised in the following table:

Case Description	**Winner**
a, b, c are all odd	P1
$b = 1$	P1
$a = 4, b \geq 2$	P2
$a = 2, b \leq 3$	P1
$a = 2, 4 \mid b, c = 1$ or 2	P2
$a = 2, 4 \nmid b, c = 1$ or 2	P1
$a = 2, b \geq 5, c = 3$	P2
$a = 2, b \geq 5, c \geq 5$	P2
$a \geq 6, b = 3$	P2
$a \geq 6, b \geq 5$	P2

The proofs of these results can be found in Appendix 6.4.

3.3 Research Question 3

We will consider the following graphs:

- Generalised Grid Graph
- Generalised Rook Graph
- Complete Graph
- Trees
- Bipartite Graph

3.3.1 Generalised Grid Graphs, Generalised Rook Graphs and Complete Graphs

If the number of vertices is odd, then it is clear than a perfect matching cannot exist. By Lemma 2, the first player can force a win.

If the number of vertices is even, then it can also be shown that all 3 graphs have a perfect matching using Lemma 3. By Lemma 2, the second player always wins.

The full proof can be found in Appendix 6.5.

3.3.2 Trees and Bipartite Graphs

As all trees are bipartite graphs, we may work out the solutions for bipartite graphs to solve for the case of a tree.

Finding the size of a maximal matching in a bipartite graph is a well-known problem with known polynomial time and space algorithms. Using a modified Augmenting Path algorithm, we were able to create a program that takes in a bipartite graph $G(S_1, S_2, E)$ as input and output all winning starting positions for P1 or determines that P2 always wins. The code and results can be found in Appendix 6.6 and 6.7.

4. Conclusion and Future Work

The second player wins iff there exists a perfect matching.

For the game variant in an $a \times b \times c$ cuboid in which players use either moves to adjacent unit cubes or moves similar to knight moves, we have shown whether or not there exists a perfect matching, for all values of a, b and c. We also designed a C++ program to find winning positions for bipartite graphs.

Possible areas for further research include:

- Investigating the winning strategies for a general simple graph
- Extending the winning condition to non-simple graphs with multi-edges and self-loops.
- Allowing players to be able to use multiple types of moves, instead of just one type per game

5. References

Berge, C. (1957). TWO THEOREMS IN GRAPH THEORY. Proceedings of the National Academy of Sciences of the United States of America, 43(9), 842–844.

Fraenkel, A. S., Scheinerman, E. R., & Ullman, D. (1993). Undirected edge geography. Theoretical Computer Science, 112(2), 371-381. doi:10.1016/0304-3975(93)90026-p

Nowakowski, R. J., & Poole, D. G. (n.d.). Geography Played on Products of Directed Cycles. Retrieved March 21, 2018, from http://library.msri.org/books/Book29/files/nowak.pdf

Prized Problems: Problem A. (2004). Mathematical Medley, 31.

Cull, P., & De Curtins, J. (n.d.). KNIGHT'S TOUR REVISITED. Retrieved June 23, 2018, from https://www.fq.math.ca/Scanned/16-3/cull.pdf

6. Appendices

6.1 Full Terminology

Term	Definition
Vertex	A vertex is the fundamental unit of a graph and is the endpoint of edges in a graph.
Edge	An edge defined as a pair of vertices (u, v). If the edge (u, v) exists, the vertices u and v are said to be neighbours.
Graph	A graph G made of a pair of sets (V, E), where V is the set of vertices in the graph while E is the set of edges (u, v) with $u, v \in E$ in G.
Simple Path	A simple path $P = (e_1, e_2, \cdots, e_k), e_i \in E \; \forall i$ in a graph $G(V, E)$ is a sequence of edges in G such that the path drawn does not contain a cycle.
Matching	A matching M in a graph $G(V, E)$ is a set of edges such that no 2 edges in M share a common endpoint.
Maximal Matching	A maximal matching M in a graph $G(V, E)$ is a matching in G with the maximum possible cardinality. There can exist more than 1 maximal matching in a graph.
Perfect Matching	A perfect matching M in a graph $G(V, E)$ is a matching in G such that the cardinality of M is equal to half the number of vertices of G. It is obvious that a perfect matching cannot exist in a graph with an odd number of vertices.
Augmenting Path	An augmenting path is a simple path $P = (e_1 , e_2 , \cdots, e_{2k} , e_{2k+1})$ with an odd number of edges in a graph G with a predefined matching M such that $e_{even} \in M$ and $e_{odd} \notin M$ with the first and last vertices of P not incident to any edge in M.

Isomorphism	2 graphs, G and H, with vertex sets V and W respectively are considered isomorphic if there exists a bijective function $f: V \to W$ such that vertices u and v are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.
Subgraph	A subgraph S of a graph G is another graph formed from a subset of the vertices and edges of G. The vertex subset must include all endpoints of the edge subset.
Generalised Grid Graph	The generalised grid graph $G_{c_1,c_2,\cdots,c_d}$ is a graph of $c_1 c_2 \cdots c_d$ vertices where each vertex can be mapped to a d-dimensional (d $\geq$ 2) lattice point $(x_1, x_2, \cdots, x_d)$ where $1 \leq x_i \leq c_i$ for constants $c_1, c_2, \cdots, c_d$ ($c_i \geq 2 \ \ \forall i$) with 2 vertices are connected by an edge if the coordinates they represent, $(a_1, a_2, \cdots, a_d)$ and $(b_1, b_2, \cdots, b_d)$, differ by exactly 1 coordinate and the difference in that coordinate is 1. For example, vertices representing $(1, 2, 3, 4)$ and $(1, 3, 3, 4)$ are adjacent but vertices representing $(1, 2, 3, 4)$ and $(1, 4, 3, 4)$ are not adjacent.
Generalised Rook Graph	The generalised rook graph (which will be denoted here as $R_{c_1,c_2,\cdots,c_d}$) is a graph of $c_1 c_2 \cdots c_d$ vertices where each vertex can be mapped to a d-dimensional (d $\geq$ 2) lattice point $(x_1, x_2, \cdots, x_d)$ where $1 \leq x_i \leq c_i$ for constants $c_1, c_2, \cdots, c_d$ ($c_i \geq 2 \ \ \forall i$) with 2 vertices are connected by an edge if the coordinates they represent, $(a_1, a_2, \cdots, a_d)$ and $(b_1, b_2, \cdots, b_d)$, differ by exactly 1 coordinate. For example, vertices representing $(1, 2, 3, 4)$ and $(4, 2, 3, 4)$ are adjacent but vertices representing $(1, 2, 3, 4)$ and $(1, 3, 2, 4)$ are not adjacent.
Complete Graph	The complete graph of n vertices, denoted as K_n is the graph $G(V, E)$ such that for any 2 vertices u and v, the edge (u, v) exists in E..

Tree	A tree is a connected acyclic graph $G(V,E)$ such that the number of edges is exactly 1 less than the number of vertices.
Bipartite Graph	A bipartite graph is a graph $G(V,E)$ whose vertices can be partitioned into 2 sets S_1 and S_2 such that for 2 vertices u and v in the same set, the edge (u,v) does not exist in E. We shall denote such a graph as $G(S_1,S_2,E)$ from here onwards.

6.2 Proof of the 3 Lemmas

Proof of Lemma 1

Suppose there exists an augmenting path $P=(e_1\ ,\ e_2\ ,\cdots,\ e_{2k}\ ,\ e_{2k+1})$. Let e_1 be (u_1,v_1) and e_{2k+1} be (u_2,v_2) with u_1 and v_2 being the first and last vertices of P respectively. Since neither u_1 nor v_2 are part of an edge in M, we consider the 2 set of edges that form P, $S_1=\{e_1\ ,e_3\ ,\cdots,\ e_{2k+1})$ and $S_2=\{e_2\ ,e_4\ ,\cdots,\ e_{2k})$. Evidently, these 2 sets of edges are disjoint, the cardinality of S_1 is 1 more than that of S_2 and 2 edges within the same set do not share a vertex.

Now consider the set of vertices $M'=(M\setminus S_2)\cup S_1$. Since S_2 is a subset of M, S_1 is disjoint with M and the cardinality of S_1 is 1 more than that of S_2, the size of M' is 1 more than that of M. Since M is a matching and S_1 is a subset of M, all vertices along the augmenting path P are not incident to an edge in $M\setminus S_2$, hence M' forms a matching as well. However, this contradicts the assumption that M is a maximal matching since M' is a matching of greater cardinality than M. This means that our assumption that P exists is false, hence Lemma 1 is proven.

Proof of Lemma 2

Suppose there exists a perfect matching M in $G(V, E)$. We claim that if the second player always plays moves along edges in M, the second player cannot lose.

Let u be the starting vertex that P1 chooses. Since M is a perfect matching in G, there exists a vertex v in G such that the edge (u,v) is in M. P2 moves to v and now both u and v are visited vertices. As such, they can be ignored for the rest of the game as no player can ever move to those 2 vertices. If P1 cannot move, P2 clearly wins. Now suppose P1 moves to a vertex w incident to v.

The game is now equivalent to playing on a graph G' defined as G without u and v and all of the edges incident to u or v as well as the perfect matching M' on G' defined as M without the edge (u, v). P1 started on w and P2 can keep moving on the edge in M' containing w. This process will terminate as G is a finite graph, and since P2 can always move after P1, P2 never loses. As this is a zero-sum game, P2 must win.

Now suppose that there does not exist a perfect matching in $G(V, E)$. Let M be an arbitrary maximal matching in G. We claim that if P1 chooses a vertex that is not incident to any edge in M and plays along edges in the matching M, P1 cannot lose.

Let the sequence of vertices marked in order of visit is $(u_1, u_2, \cdots, u_{2k})$ where u_1 is the vertex that the first player chooses and u_{2k} is the most recently visited vertex. Evidently, P2 was the one who moved to u_{2k}. Note that whenever P1 moves along an edge in M, P2 cannot move along an edge in M as no 2 edges in M can be incident to the same vertex. If there is no edge in M that is incident to u_{2k}, i.e. P1's strategy falls apart, then we have found an augmenting path in M. However, this is impossible by Lemma 1 as we had assumed that M was a maximal matching. Thus P1 can always make a move and never lose. As such, P1 will win. Hence, Lemma 2 is proven.

Proof of Lemma 3

Let M be the perfect matching in G. Since S is isomorphic to G, there exists a perfect matching M' in S since there exists a perfect matching in G. Since M' is a subset of the edges in S and S is a subgraph of H, M' is a subset of the edges in H. As S and H have the same number of vertices, M' is a perfect matching in H and thus Lemma 3 is proven.

6.3 Proof for 3.1

We claim that P1 wins iff abc is odd. Let $G(V, E)$ be the graph representing the cuboid, where each cell in the cuboid is a vertex and 2 vertices in G are connected by an edge if the 2 cells they represent are adjacent (share a face).

Suppose abc is odd. Then G has an odd number of vertices, which means no perfect matching can exist. By Lemma 2, P1 wins.

Suppose abc is even. Without loss of generality, assume that a is even, Notice that an $a \times b \times 1$ cuboid can always be partitioned into disjoint 1×2 dominos since a is even. Thus, the whole grid can be partitioned into 1×2 dominos. For each domino, we let the edge connecting the 2 vertices representing the 2 cells in G belong to a set of edges K. Since the 1×2 dominos are disjoint, no 2 edges in K share a vertex. Also, since every cell belongs to a domino, K contains exactly $\frac{abc}{2}$ edges, which implies that K is a perfect matching in G. By Lemma 2, P2 wins.

6.4 Proof for 3.2

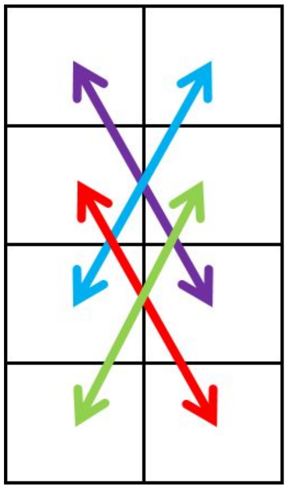
Again, we let $G(V, E)$ be the graph representing the cuboid, where each cell in the cuboid is a vertex and 2 vertices in G are connected by an edge if one can travel from 1 cell to the other via a knight move, which is defined as moving 2 cells in 1 axis-parallel direction and 1 cell in a direction perpendicular to the previous one.

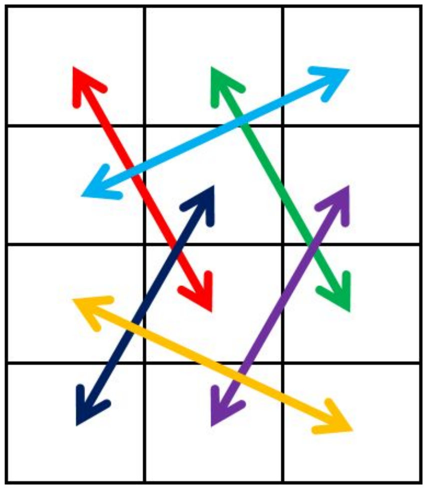
Suppose a, b and c are all odd, then the product abc is odd. As a result, G has an odd number of vertices and can never have a perfect matching. By Lemma 2, the first player wins.

From here on, we will assume without loss of generality that a is the smallest even number among a, b and c and that $b \geq c$.

Suppose that b is 1. Then c is 1 which means that our cuboid is simply a row of cells. Evidently, no matter where the first player starts at, the second player cannot move at all and the first player wins.

Suppose a is 4 and b is at least 2. The following 2 figures show perfect matchings for a $4\times2\times1$ and a $4\times3\times1$.



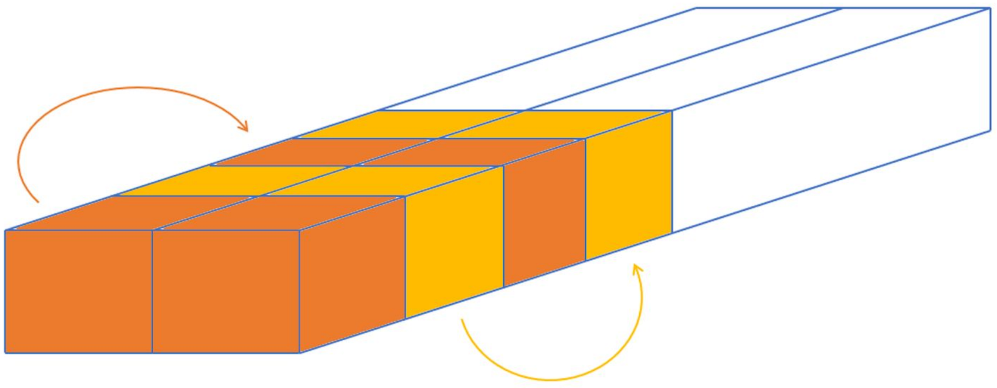


By the Chicken McNugget Theorem, the largest b that cannot be formed as a sum of 2s and 3s is $(2)(3) - 2 - 3 = 1$. Since b is at least 2, a perfect matching can be found in the $4 \times b \times 1$. This matching can be done for each of the c layers in the $4 \times b \times c$ cuboid, obtaining a perfect matching for the whole cuboid. By Lemma 2, P2 wins.

Suppose that $a = 2$ and b is at most 3. Then c is at most 3. Since all dimensions of the cuboid are at most 3, there exists a cell in the cuboid with distance at most $\lfloor \frac{3}{2} \rfloor = 1$ from each of the edges of the cuboid. If P1 plays in this cell, it is not possible for P2 to move at all since a knight move requires moving 2 cells in 1 direction, hence P1 wins.

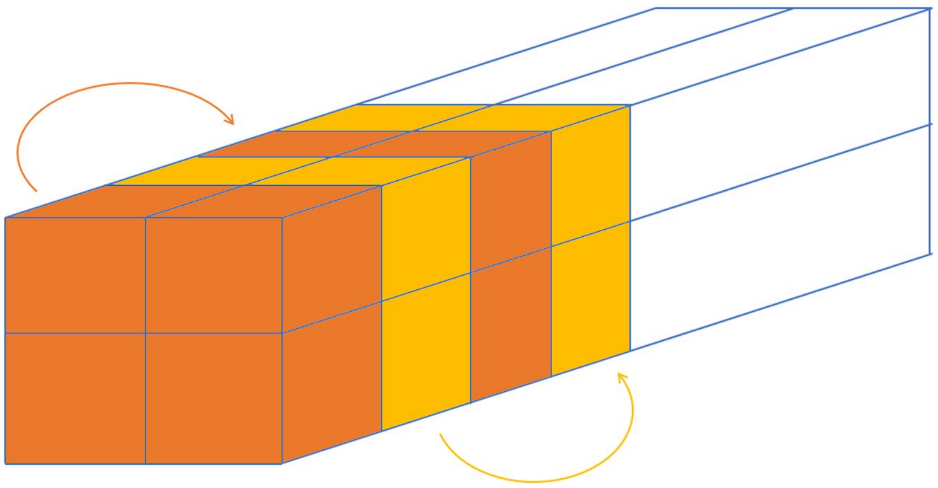
Suppose that $a = 2$, b is divisible by 4 and c is 1 or 2. Notice that an $2 \times b \times 1$ can be separated into disjoint $2 \times 4 \times 1$ cuboids as b is divisible by 4. Since a perfect matching for a $2 \times 4 \times 1$ exists, we can duplicate this matching for a $2 \times b \times 1$ and finally a $2 \times b \times c$. By Lemma 2, the second player wins.

Suppose that $a = 2$, $4 \nmid b$ and c is 1. Consider the 2 cells in the $2 \times 1 \times 1$ at one of the faces of the cuboid. These 2 cells must be matched to the 2 cells that are 2 cells away as they are not connected to any other cells. Similarly, the 2 cells 1 away from the edge must be matched to the vertices that are 2 away from them.



As such, if there exists a perfect matching, this $2\times4\times1$ block must have a perfect matching contained entirely within it. We can ignore this block when matching the rest of the vertices. By removing $2\times4\times1$ blocks, we will eventually end up with a $2\times3\times1$, $2\times2\times1$ or $2\times1\times1$, all of which do not have a perfect matching. Thus there exists no perfect matching in a $2\times b\times1$ with $4\nmid b$ and the first player wins.

Suppose that a = 2, $4\nmid b$ and c is 2. We may use a similar argument to that of the $2\times b\times1$ by considering $2\times4\times2$ blocks at the edge of $2\times b\times2$.



Removing these blocks gives us either a $2\times3\times2$, $2\times2\times2$ or a $2\times1\times2$, none of which have a perfect matching. Hence there is no perfect matching in a $2\times b\times2$ for $4\nmid b$ and P1 wins.

Suppose that $a=2$, b is at least 5 and c is 3. The following 3 diagrams will show perfect matchings for a $2\times5\times3$, a $2\times6\times3$ and a $2\times7\times3$. The matching for a $2\times4\times3$ can be done using the matchings for a $2\times4\times1$. (Cells with the same number are matched together)

1	2
3	4
5	11
12	13
14	15

Layer 1

5	6
7	8
9	10
8	7
10	9

Layer 2

2	1
4	3
11	6
13	12
15	14

Layer 3

Construction for 2x5x3

1	2
3	4
5	7
14	13
15	16
17	18

Layer 1

5	6
8	9
10	11
9	8
11	10
12	13

Layer 2

2	1
4	3
7	6
12	14
16	15
18	17

Layer 3

Construction for 2x6x3

1	2
3	4
13	14
15	12
16	17
18	19
20	21

Layer 1

5	6
7	8
6	5
8	7
9	10
11	12
10	9

Layer 2

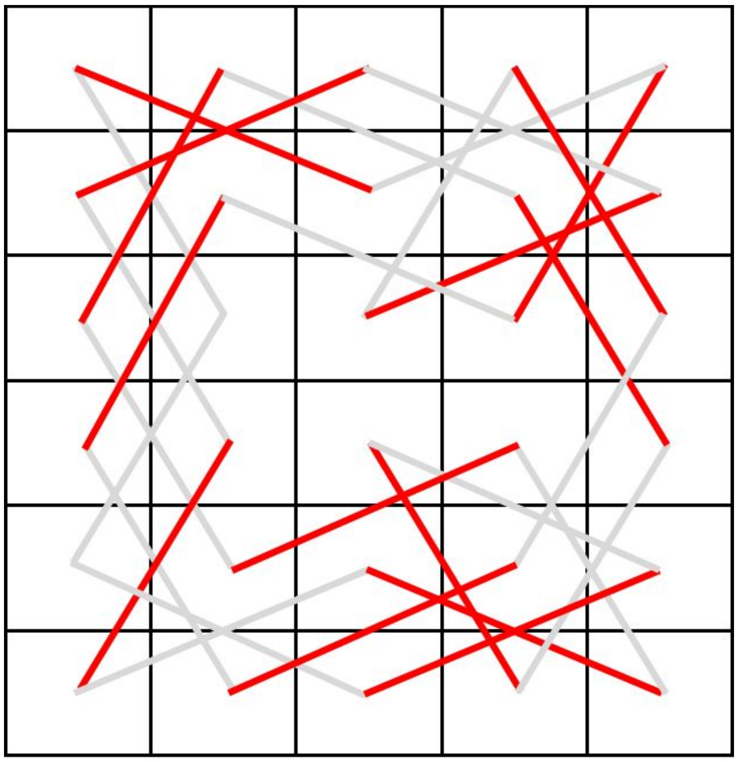
2	1
4	3
14	13
11	15
17	16
19	18
21	20

Layer 3

Construction for 2x7x3

Note that any positive integer greater than or equal to 5 can be represented as the sum of 4s, 5s, 6s and 7s. Thus any $2{\times}b{\times}3$ has a perfect matching and by Lemma 2, the second player wins.

Suppose that $a = 2$, b and c are at least 5 each. We know that there exists a knight tour in any grid whose smallest dimension is at least 5 (Cull, P., & De Curtins, 1978). Thus if either b or c is even, we may take alternate edges of the knight tour on a $1{\times}b{\times}c$ and obtain a perfect matching. An example of a $1{\times}6{\times}5$ is shown below, with the edges in the matching coloured red.



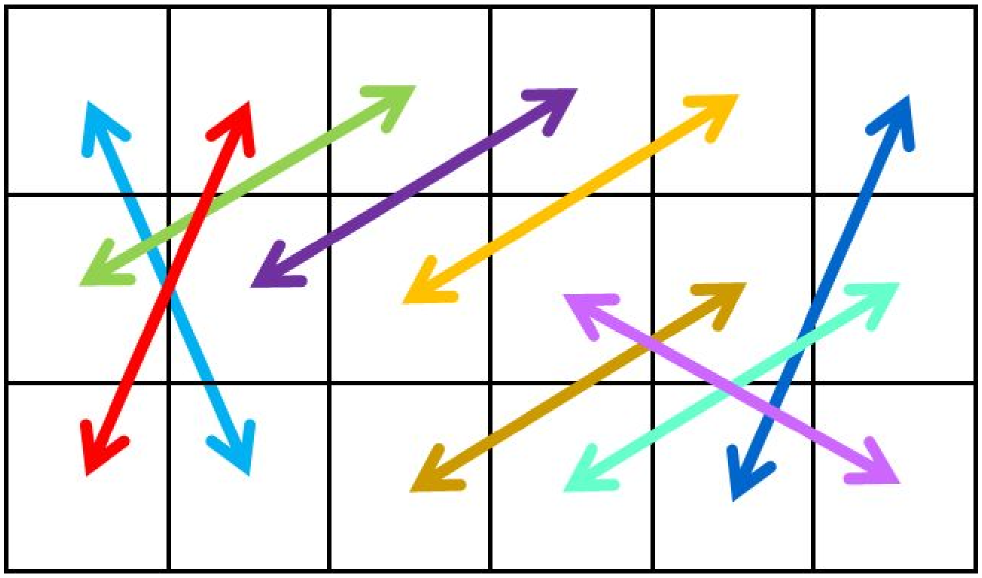
Knight Tour of 6x5

If b and c are both odd, then we shall consider each $1 \times b \times c$ as a separate layer. For each layer, colour it in a checkerboard fashion, with the corners coloured white. Note that a knight move will always change the colour of the square it is on, from black to white vice versa. Thus, the knight tour must start and end on a white square.

Suppose we block of one of the white squares on a layer. This breaks the knight tour into 2 disjoint paths, each of which as 1 black square and 1 white square as endpoints. This means each path has an even number of vertices, thus we may take alternate edges starting from one of the endpoints of each path to obtain a perfect matching in that layer excluding the blocked square.

Now, a knight move on a white square on one layer will always move to a white square on the next layer since it must move 1 cell to the next layer and 2 cells in another direction. Thus, we may block off the corresponding cell on the next layer. Since that cell is white, it breaks the knight tour in the second layer into 2 disjoint paths, which can form a perfect matching by taking alternate edges. Thus, we have obtained a perfect matching in the whole $2 \times b \times c$ cuboid and by Lemma 2, the second player wins.

Suppose that a is at least 6 and b is 3. The construction for a $6 \times 3 \times 1$ is shown below.



Notice that any even number that is at least 6 can be represented as a sum of 4s and 6s. Using the perfect matchings for a $4 \times 3 \times 1$ and a $6 \times 3 \times 1$, we can obtain a perfect matching for a $a \times 3 \times 1$ and use c layers of it to get a perfect matching for a $a \times 3 \times c$. By Lemma 2, the second player wins.

Suppose that a is at least 6 and b is at least 5. Then each $a \times b \times 1$ layer has a knight tour (Cull, P., & De Curtins, 1978) and since a is even, we may again take alternate edges of the knight tour to obtain a perfect matching for a $a \times b \times 1$, which can be duplicated for c layers to obtain a perfect matching in the whole $a \times b \times c$. By Lemma 2, the second player wins.

6.5 Proof for 3.3.1

Note that the generalised grid graph $G_{c_1,c_2,\cdots,c_d}$ is a subgraph of $R_{c_1,c_2,\cdots,c_d}$ and $R_{c_1,c_2,\cdots,c_d}$ is a subgraph of the complete graph $K_{c_1c_2\cdots c_d}$. By Lemma 3, if there exists a perfect matching in $G_{c_1,c_2,\cdots,c_d}$, then there exists a perfect matching in both $R_{c_1,c_2,\cdots,c_d}$ and $K_{c_1c_2\cdots c_d}$. Also, by the inverse of Lemma 3, if there does not exist a perfect matching in the complete graph $K_{c_1c_2\cdots c_d}$, then there does not exist perfect matchings in both $R_{c_1,c_2,\cdots,c_d}$ and $G_{c_1,c_2,\cdots,c_d}$.

If the number of vertices is odd, i.e. the product $c_1c_2\cdots c_d$ is odd, there can never be a perfect matching. Thus by Lemma 2, the first player will win in $G_{c_1,c_2,\cdots,c_d}$, $R_{c_1,c_2,\cdots,c_d}$ and $K_{c_1c_2\cdots c_d}$.

If the number of vertices is even, we consider the generalised grid graph $G_{c_1,c_2,\cdots,c_d}$. Without loss of generality, we may assume that c_1 is even. We know that a 2D grid with 1 even side can be partitioned into disjoint 1×2 dominos. Thus, each c_1 by c_2 layer of vertices has a perfect matching. This can be duplicated $c_3c_4\cdots c_d$ times to obtain a perfect matching in $G_{c_1,c_2,\cdots,c_d}$, which implies the existence of a perfect matching in $R_{c_1,c_2,\cdots,c_d}$ and $K_{c_1c_2\cdots c_d}$. By Lemma 2, P2 wins.

6.6 Code for finding winning starting positions

Written in C++. Explanations for each of the functions can be found in comments in the code (lines between /* and */ or starting with //)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>
#include <cassert>

using namespace std;

/*
Input format:

First line contains 3 integers, the size of the first set of vertices S_1,
the size of the second set of vertices S_2 and the number of edges in the
bipartite graph E.

The next E lines contain 2 integers i and j, which represent the edge
between the ith vertex in the first set and the jth vertex in the second
set.

Note that i and j are 1-indexed, that means that i ranges from 1 to S_1
inclusive and j ranges from 1 to S_2 inclusive.


Output format:

The first line will be "The size of a maximal matching in the bipartite
graph is M" where M is the size of the maximal matching in the given bipartite
graph.

The next M lines will contain 2 integers X and Y each, where X lies within
1 and S_1, Y lies within 1 and S_2 and the edge (X,Y) exists in the given
bipartite graph. This represents the edge (X,Y) being chosen as an edge in
the maximal matching

The next line will be a blank line.

The first line will contain either "Player 2 always wins" or "Player 1
can win by starting at T different starting positions" where T is an
integer determined by the algorithm.

If the first line is "Player 1 can win by starting at T different
starting positions", the following T lines will print either "Vertex A
from the first set is a winning position" where A lies within the range
1 to S_1 or "Vertex A from the second set is a winning position" where
A lies within the range 1 to S_2.
*/

// Helpful typedef
typedef pair<int,int> pi;

// Adjacency List with directed edges from vertices in the first set to the
// vertices in the second set.
vector<vector<int> > AdjList;

// Adjacency List with directed edges from vertices in the second set to the
// vertices in the first set.
vector<vector<int> > BackList;
```

```
// Marks already visited nodes
vector<bool> visited;

// Used for the Augmenting Path algorithm
vector<int> P;

// For storing the maximal matchings
vector<pi> matches1, matches2;

int label1[100005];
int label2[100005];
int comp[100005];

int S1, S2, E, L, R, T, match1, match2;

// Finding Augmenting Paths (first set of vertices -> second set of vertices)
bool Aug1(int v) {

    // If the current vertex has already been processed, there is no point
    // reprocessing the vertex. Just return 0.
    if (visited[v]) return 0;

    // Set the current vertex as already processed. This is to deal with cycles.
    visited[v] = 1;

    // For each vertex u in the second set adjacent to vertex v in the first set,
    // check if u is incident to an edge in the matching. If not, match it with v
    // and recurse backwards along the augmenting path and add 1 to the number of
    // edges in the matching.
    for (auto u : AdjList[v]) {
        if (P[u] == -1) {
            P[u] = v;
            return 1;
        }
    }

    // For each vertex u in the second set adjacent to vertex v in the first set,
    // check if u is incident to an edge in the matching. If it is, try to create
    // an augmenting path by travelling along the edge matched to u. If an
    // augmenting path is found, set v as the vertex matched to u and add 1 to the
    // number of edges in the matching
    for (auto u : AdjList[v]) {
        if (Aug1(P[u])) {
            P[u] = v;
            return 1;
        }
    }

    // If nothing happens, there is no augmenting path, so return 0.
    return 0;
}
```

```
// Finding Augmenting Paths (second set of vertices -> first set of vertices)
// Basically the same as Aug1, but all the directed edges are reversed.
bool Aug2(int v) {
    if (visited[v]) return 0;
    visited[v] = 1;
    for (auto u : BackList[v]) {
        if (P[u] == -1) {
            P[u] = v;
            return 1;
        }
    }
    for (auto u : BackList[v]) {
        if (Aug2(P[u])) {
            P[u] = v;
            return 1;
        }
    }
    return 0;
}

// Starts on a vertex not in the maximal matching and finds an path that alternates
// between edges in the matching and not in the matching. Each vertex in first set
// along this path has a maximal matching in the bipartite graph that does not
// have an edge that is incident to that vertex, which means that it is a winning
// starting position for player 1.
void BackAug1(int v){

    // Sets the vertex as processed
    visited[v] = 1;
    label1[v] = 0;
    for (auto u : AdjList[v]){
        if (!visited[P[u]]) BackAug1(P[u]);
    }
    return;
}

// Same as BackAug1 but from the second set to the first set instead.
void BackAug2(int v){
    visited[v] = 1;
    label2[v] = 0;
    for (auto u : BackList[v]){
        if (!visited[P[u]]) BackAug2(P[u]);
    }
    return;
}

int main(){

    // Fast I/O for debugging
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    // Input sizes and number of edges
    cin >> S1 >> S2 >> E;

    // Assertions to catch invalid graphs
    assert(S1 >= 0 && "Size of first set of vertices is negative\n");
    assert(S2 >= 0 && "Size of second set of vertices is negative\n");
    assert(E >= 0 && "Number of edges is negative\n");

    // Resize the vectors for input
    AdjList.resize(S1+1);
    BackList.resize(S2+1);
    // Input edges
    for (int i = 1; i <= E; ++i){
        cin >> L >> R;

        // Check for invalid edges
        assert(L >= 1 && L <= S1 && R >= 1 && R <= S2 && "Invalid Edge Detected\n");

        // Add the edges:
        // AdjList goes from first set to second set
        // BackList goes from second set to first set
        AdjList[L].push_back(R);
        BackList[R].push_back(L);

        // Note that duplicated edges are handles implicitly via checking for
        // already processed vertices in the graph
    }

    // First part: Finding a maximal matching from the via the first set

    P.resize(S2+1,-1);

    // Run the augmenting path algorithm to find the size of the maximal matching
    // along with 1 possible maximal matching
    for (int i = 1; i <= S1; ++i) {

        // Set all vertices to unvisited
        visited.resize(S1+1, 0);

        // Try to find an augmenting path starting at this vertex
        match1 += Aug1(i);

        // Reset the visited status of each vertex
        visited.clear();
    }

    // Stores the maximal matching
    for (int i = 1; i <= S2; ++i)
        if (P[i] != -1) matches1.emplace_back(P[i], i);

    for (int i = 1; i <= S1; ++i)
        label1[i] = 0;

    for (auto it : matches1)
        label1[it.first] = 1;

    // Run the BackAug to find all winning positions in the first set.
    for (int i = 1; i <= S1; ++i){

        // If the vertex is not part of the original maximal matching
        if (!label1[i]){

            visited.resize(S1+1, 0);

            // Run a BackAug starting from this vertex to find other
            // vertices not part of a maximal matching
            BackAug1(i);

            visited.clear();
        }
    }

    // Second part: Finding a maximal matching from the via the second set
    // The rest of the code here is essentially the same as the first part
    // except that the BackList is used instead of the AdjList.
    P.clear(); P.resize(S1+1,-1);
    for (int i = 1; i <= S2; ++i) {
        visited.resize(S2+1, 0);
        match2 += Aug2(i);
        visited.clear();
    }

    for (int i = 1; i <= S1; ++i)
        if (P[i] != -1) matches2.emplace_back(P[i], i);

    for (int i = 1; i <= S2; ++i)
        label2[i] = 0;

    for (auto it : matches2)
        label2[it.first] = 1;

    for (int i = 1; i <= S2; ++i){
        if (!label2[i]){
            visited.resize(S2+1, 0);
            BackAug2(i);
            visited.clear();
        }
    }

    // Find the number of vertices in the first set that are winning
    // positions for the first player
    for (int i = 1; i <= S1; ++i)
        if (!label1[i]) T++;

    // Find the number of vertices in the second set that are winning
    // positions for the first player
    for (int i = 1; i <= S2; ++i)
        if (!label2[i]) T++;

    // Output the maximal matching
    cout << "The size of a maximal matching in the bipartite graph is " << match1 << '\n';
    for (int i = 0; i < matches1.size(); ++i)
        cout << matches1[i].first << ' ' << matches1[i].second << '\n';

    cout << '\n';

    // If there exists no position where the first player can win
    if (T == 0){
        cout << "Player 2 always wins\n";
    }
    // Otherwise, print all winning positions
    else{
        cout << "Player 1 can win by starting at " << T << " different starting positions\n";
        for (int i = 1; i <= S1; ++i)
            if (!label1[i])
                cout << "Vertex " << i << " from the first set is a winning position\n";

        for (int i = 1; i <= S2; ++i)
            if (!label2[i])
                cout << "Vertex " << i << " from the second set is a winning position\n";

    }

    // Flush  output
    cout << flush;

    return 0;
}
```

6.7 Results of running the code in Appendix 6.6

Example Input

```
7 7 11
1 2
1 3
2 5
2 6
3 4
4 1
5 4
6 2
6 4
7 3
7 6
```

Example Output

```
The size of a maximal matching in the given bipartite graph is 6
4 1
6 2
1 3
3 4
2 5
7 6

Player 1 can win by starting at 3 different starting positions
Vertex 3 from the first set is a winning position
Vertex 5 from the first set is a winning position
Vertex 7 from the second set is a winning position
```

Benchmark Tests with randomly generated bipartite graphs with a given number of vertices.
(running on Ubuntu (64-bit) OS with a single Intel Core i7-6700HQ CPU @ 2.6GHz)

Number of vertices in first set	Number of vertices in second set	Number of edges	Time taken (in seconds)
100	100	1004	0.024
1000	1000	9969	0.059
5000	5000	25009	0.296
50000	49999	249731	21.398
50000	49999	2500140	4.335
50000	50000	250216	12.185
50000	50000	1249602	4.466
100000	99999	332943	79.786
100000	100000	289365	17.128