

The Knight's Tour

ECEN 2703 Final Project Report

Will Snider and Tucker Travins

Introduction

For the final project, we chose to use python to create a program which simulates a “Knight’s Tour”. What is a Knight’s Tour? This is a set of moves a chess knight can take on a chess board to cover every single square exactly once and only once, using only the valid “two in one direction, one in another” move which a knight uses in chess. Not only that, in certain situations, it is possible for the knight to complete its tour and end at a square from where it can jump back to the start square, a closed tour, or have a closed tour in which one corner square is omitted, a corner closed tour, an unrestricted tour is called “open”.

The scope of our project was to find a tour for any sized chess board in which one exists in linear time, with the user choosing both dimensions of the board. Also, the user can choose “closed”, “open”, or “corner” to create a path which follows the rules of said tour. We wanted our program to create a path for the knight which was stored inside a list, where each item in the list is the square which the knight will move to. In order to actually show what the path looks like, we made a print function which shows the board and which move number corresponded to each square, and also an animation that tracks the knight as it travels.

The reason this topic interested us is because we both play chess, and chess has been very popular among people our age, partially due to the Netflix series “The Queen’s Gambit”, a show centered around a chess prodigy. The knight’s tour was originally researched by Euler, on a standard 8x8 chessboard and other small boards in the 1700’s. As time has passed, this problem has interested many mathematicians, and as technology grew, there were new ways to create and solve the knight’s tour using programming, AI, and algorithms. The knight’s tour can be reduced to finding a hamiltonian path or cycle, an NP complete problem. Chess boards will at a minimum contain dozens of squares and hundreds of edges when constructing the graph, this makes solving the problem particularly difficult with brute force methodology such as breadth/depth first searches.

The Algorithm

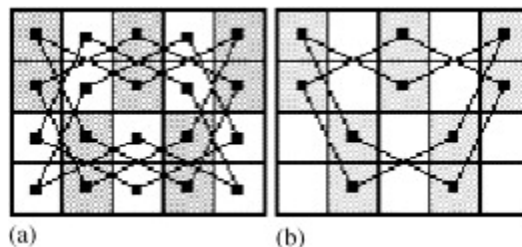
Initially a heuristic approach was taken, in which a standard recursive depth first search of the graph is done with the choice of which move to take being determined by a heuristic. For knights tours, the Warnsdorff’s heuristic is commonly used, this orders the moves by which one has the least following moves, second order tie breaking was implemented in which ties were decided by the sum of each moves available at the next layer down, this often allows boards to be solved in linear time. This approach worked reasonably well, allowing boards up to a couple thousand squares to be solved reasonably quickly. This heuristic doesn’t specifically look for closed tours and thus some modifications had to be made to find them, specifically forcing the

last square to have the start square as one of its edges, to further improve success, the starting square is chosen from the center as the heuristic searches “outward”, this increased the odds of ending near the start. This was a good start allowing us to solve closed boards up to 30x30 reasonably fast and open tours up to 50x50 until recursion depth issues were reached. To solve the recursion issue, implementing the recursion manually through a stack allowed open tours up to 160x160 to be solved, this was later reverted to recursive as we didn’t need large boards for the final algorithm and the code looked much nicer on the eyes. This approach can be found in `HeuristicApproach.py`.

As a side note the reason for this approach breaking down on large boards is actually quite interesting. On larger boards, the tie breaking will, more often than not, still result in ties and so “wrong” moves will be made early on. Further levels of tie breaking can be used, but there also exist perfectly symmetrical positions in which no amount of tie breaking will help and at a certain point the amount of levels looked into will become extremely inefficient and will practically be running a breadth first search every move. Multiple proposals to improve the heuristic exist, yet none work for every board, and none allow for closed boards in any other way than being an improved DFS.

From here we wanted to implement traversal of a cubic chessboard in which the surfaces are the boards. For a multitude of reasons, our current heuristic approach wasn’t viable for this. We wanted to produce much larger boards and being able to stitch together all the faces, required the ability to generate much larger closed tours and other specific kinds of tours that will be explained later. This new algorithm ended up taking much longer than expected and thus we weren’t able to implement the cubic boards; however, as far as we can tell, this is the only runnable program that can compute open and closed tours of arbitrarily sized square and nonsquare boards in linear time that we could find. The algorithm itself was developed by Shun-Shii Lin and Chung-Liang Wei in 2005 [\[1\]](#) as a further improvement on an algorithm created by Ian Parberry which was able to find tours on square boards. In the paper, closed and open tours are separated, however they can be combined with only one extra case and so that is how ours is implemented in code.

As a basis let’s discuss certain unique kinds of tours that will be important, a double loop knights tour is a tour consisting of two closed tours that do not overlap. The only board where this will be important is 4x5, said board is shown below.



[\[1\]](#)

The next unique tour is a structured tour, a structured tour is a tour in which each corner has a unique pattern. For the algorithm, a fully structured tour is never needed, only partially structured tours, when a structured tour is mentioned from here on, it will refer to a partially structured tour. An example of which is shown below, specifically those two corner moves will have to be present in the path for it to be considered partially structured.

10
Give the second side length
10
enter the type of tour, eg. closed, open, corner
closed

14	17	46	37	44	19	96	23	42	21
47	36	15	18	97	38	43	20	7	24
16	13	98	45	74	93	88	95	22	41
35	48	73	92	99	90	39	78	25	80
12	85	50	75	72	87	94	89	40	57
49	34	67	86	91	0	77	56	81	26
66	11	84	51	76	71	82	1	58	55
33	8	63	68	83	52	61	70	27	2
10	65	6	31	62	69	4	29	54	59
7	32	9	64	5	30	53	60	3	28

Lastly a stretched tour is a structured open tour in which the start square is the top left corner and the end square is on either side of the start. An example is shown below. These only exist when $n*m$ is even.

0	99	28	35	2	79	40	9	4	7
29	34	1	78	59	36	3	6	41	10
76	27	98	37	80	61	58	39	8	5
33	30	77	60	97	38	81	44	11	42
26	75	32	93	82	85	62	57	64	45
31	94	53	86	55	96	69	84	43	12
52	25	74	95	92	83	56	63	46	65
73	22	91	54	87	70	89	68	13	16
24	51	20	71	90	49	18	15	66	47
21	72	23	50	19	88	67	48	17	14

For reasons that will be discussed later, stretched boards will be required a lot, and often with the same sizes. This allowed for some optimization, using our heuristic approach, which by

nature of the heuristic will always produce structured corners where possible, we are able to generate a catalog of $n*m$ stretched boards for $4 \leq n \leq 10$ and $4 \leq m \leq 10$. Certain boards, specifically some where n or m are 4, don't exist and so aren't an issue when not generated. A select few others required some manipulation of other tours to generate, such as rotations and mirroring. To generate stretched tours, a few modifications to the original approach were made, as getting a specific end square, especially one so close to the start, which is in a corner, is even more challenging than a normal closed tour. Further pruning of the move tree was required, first removing the end square from all possible moves except the second to last, and second removing all the squares that can jump to the last square when less than half the board has been traversed. This gave us all the required boards.

Constructing Tours

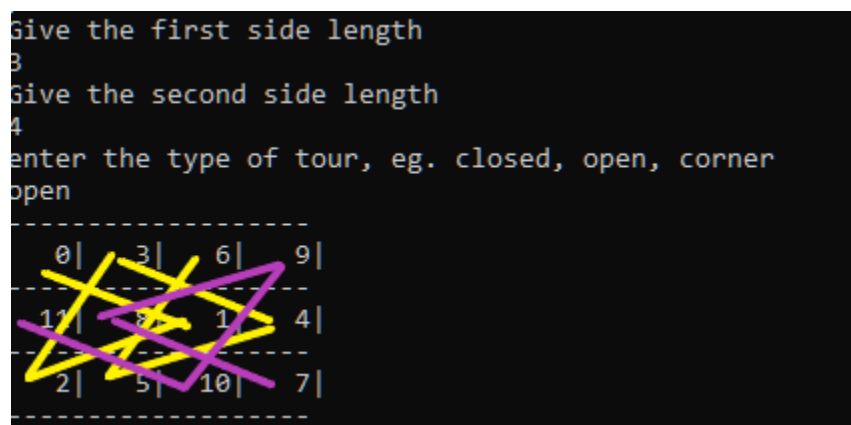
From here on n will always be less than m , the function will always be called when the dimensions are sorted so that this is the case, recovering the opposite dimensions simply requires a rotation plus a mirror depending on what the required outcome is. There are five cases to be considered when constructing knights tours.

Case One: $n \leq 10$ and $m \leq 10$

For this case, if a stretched tour is requested, the catalog will be referenced, all other tours will be found by calling the heuristic approach, cataloging all other tours is inefficient as, at most, only one will be required regardless of board size.

Case Two: $n == 3$ and ($m \geq 10$ for closed, or ($m == 4$ or $m \geq 7$ for open))

A $3*k$ board with the desired type will be constructed where $k = (m - 7) \bmod 4 + 7$ for open or $k = (m - 9) \bmod 4 + 9$ for any other type. The remaining squares will be evenly separated into $3*4$ stretched boards. The $3*4$ stretched boards can be separated into the two paths shown below.



From here the $3*k$ board can be stitched together with the rest along the structured corners as shown below. Yellow represents the path going forward and purple represents going back, as can be seen it alternates between both of the above paths going forwards and the reverse of the paths going backwards.

10	13	16	3	0	5	22	19	58	55	52	25	28	31	34	49	46	43	40	37
15	2	11	8	17	20	59	6	23	26	57	54	51	48	29	32	35	38	45	42
12	9	14	1	4	7	18	21	56	53	24	27	30	33	50	47	44	41	36	39

Doing this works for arbitrarily long boards, finding the $3*k$ path can essentially be considered constant time and the rest of the construction consists of doing two list operations per 12 squares.

Case Three: $n == 4$ and $m > 10$

This special case only pertains to open boards as closed and corner closed do not exist on $4*m$ boards. Using the double loop board discussed above, we can construct boards in a similar fashion to $3*m$ boards, this time a $4*k$ stretched board will be requested and mirrored across the n axis so as to have the start and end at the top right corner, where $k = (m - 6) \bmod 5 + 6$. From here we will find both the forward and backward path at the same time by iterating through the double loop boards and constructing the paths as shown below.

43	32	49	36	47	30	51	26	59	20	65	0	79	6	63	12	77	8	71	14
40	37	42	31	50	27	56	23	52	25	60	5	64	1	78	7	72	13	76	9
33	44	39	48	35	46	29	54	21	58	19	66	3	62	17	68	11	74	15	70
38	41	34	45	28	55	22	57	24	53	4	61	18	67	2	73	16	69	10	75

The path also follows a similar alternating structure as the previous case, to better understand how this case works, it's much easier to see if you run the visualiser on a $4*30$ board. This case is slightly less efficient than the previous, requiring 4 list operations per 20 squares.

Case Four: $n \leq 10$ and $m > 10$

This case pertains to all boards, here the board will be split into two, $n*m_1$ and $n*m_2$, where $m_1 = (m//4) * 2 + m \bmod 2$ and $m_2 = m - m_1$. This partitioning guarantees that only m_1 could possibly be odd and will only be so if m is also odd. From here we can recursively call this function to find a stretched $n*m_2$ board and whatever is requested for $n*m_1$. The partitioning guarantees that at most one board could be odd and the rest even, guaranteeing that we will be able to construct stretched tours from the rest. Below shows how the tours are stitched together.

Give the first side length
5
Give the second side length
31
enter the type of tour, eg. closed, open, corner
open

154	1	146	141	152	129	6	9	30	37	32	23	20	101	122	119	98	91	96	105	108	80	46	49	70	77	72	63	60	79	82
147	140	153	2	7	142	150	128	33	22	13	38	121	24	40	95	106	115	90	47	104	103	88	73	62	53	78	81	64	59	
134	137	0	145	130	5	8	29	10	31	36	21	16	123	120	99	118	97	92	107	112	45	48	69	50	71	76	61	56	83	80
139	148	135	132	3	150	143	34	127	12	27	14	125	18	25	94	41	116	101	114	43	110	103	74	87	52	67	54	85	58	65
136	133	138	149	144	131	4	11	28	35	126	17	26	15	124	117	100	93	42	111	102	113	44	51	68	75	86	57	66	55	84

Case 5: $n > 10$ and $m > 10$

Here we can separate the board much the same way as the previous case but this time into four different boards that are generated recursively. Following an identical partitioning rule, only the top left board will be unable to be stretched, this allows us to construct any tour we want in a similar way to the previous case. The stitching together of boards is shown below.

Give the first side length
25
Give the second side length
25
enter the type of tour, eg. closed, open, corner
open

624	11	608	215	622	313	8	15	182	199	202	179	184	23	112	165	168	109	114	31	42	95	98	39	44
607	222	623	12	9	216	63	212	203	180	183	198	201	178	169	110	113	164	161	108	99	40	43	94	97
4	609	10	221	214	7	14	181	16	211	200	185	22	111	24	177	166	115	30	41	32	107	96	45	38
223	606	5	0	13	620	217	206	195	204	191	210	197	172	161	170	121	176	163	102	91	100	51	106	93
610	3	220	225	6	617	614	17	190	207	196	21	186	25	120	173	162	29	116	33	50	103	92	37	46
605	224	1	612	615	218	619	194	205	188	19	192	209	60	171	118	27	122	175	90	101	48	35	52	105
9	611	226	219	618	613	616	189	18	193	208	187	20	119	26	123	174	117	28	49	34	53	104	47	36
227	604	537	598	229	586	545	550	561	584	563	572	559	124	159	128	143	150	153	54	89	58	73	80	83
538	595	228	587	546	597	23	585	570	553	560	583	564	135	144	125	152	129	142	65	74	55	82	59	72
591	536	603	596	599	544	549	554	551	562	571	558	573	158	127	136	149	154	151	88	57	66	79	84	81
594	539	592	547	588	231	600	569	578	575	552	565	582	137	134	145	126	141	130	67	64	75	56	71	60
535	590	541	602	233	548	543	576	555	580	567	574	557	146	157	132	139	148	155	76	87	62	69	78	85
540	593	234	589	542	601	232	579	568	577	556	581	566	133	138	147	156	131	140	63	68	77	86	61	70
235	534	323	528	237	516	331	336	347	514	349	358	345	502	491	360	489	480	493	444	455	478	457	466	453
324	525	236	517	332	527	238	315	356	339	346	513	310	359	482	499	492	361	483	479	464	447	454	477	458
521	322	533	526	529	330	335	340	337	348	357	344	503	498	501	490	481	494	443	448	445	456	465	452	467
524	325	522	333	518	239	530	355	508	505	338	351	512	483	366	441	500	487	362	463	472	469	446	459	476
321	520	327	532	241	334	329	506	341	510	353	504	343	440	497	364	485	442	495	470	449	474	461	468	451
326	523	242	519	328	531	240	509	354	507	342	511	352	365	484	367	496	363	486	473	462	471	450	475	460
243	320	253	314	245	302	261	266	277	300	279	288	275	368	439	412	423	370	435	406	395	372	393	384	397
254	311	244	303	262	313	246	301	286	269	276	299	280	413	424	369	436	419	426	371	386	403	396	373	392
307	252	319	312	315	260	265	270	267	278	287	274	289	438	411	430	421	434	407	402	405	394	385	398	383
310	255	308	263	304	247	316	285	294	291	268	281	298	425	414	437	418	429	420	387	378	381	404	391	374
251	306	257	318	249	264	259	292	271	296	283	290	273	410	431	416	427	408	433	380	401	376	389	382	399
256	309	250	305	258	317	248	295	284	293	272	297	282	415	426	409	432	417	428	377	388	379	400	375	390

Board solved in 0.000470 seconds

For these last two cases, you can think of the board as being a smaller board where each square is one of the smaller boards except with each requiring 2-4 operations when they are visited rather than just one, as we methodically jump from board to board the required time to construct the tour is linear.

What's Next?

From the boards we can generate, we would be able to construct surface area tours fairly easily for cubes that are bigger than roughly $3*3*4$ by using the same concepts from the above algorithm. The cube would be separated into $l*(2m+2n)$, $n*m$, $n*m$ boards where $l \leq n \leq m$. This guarantees that the middle partition is even and thus can be closed, if $n*m$ is even, we can create two stretched tours and combine at the corners much like above. If $n*m$ is odd we can do some clever manipulation around the corners of the boards to generate tours with the $n*m$ boards being corner closed. Smaller cubic boards would either have to be cataloged or generated in some other way. Improving the current program could be done through multithreading in the last two cases, however in terms of speed, the print function seems to be the largest bottleneck on memory and time.

Conclusion

As far as accomplishing our goals of finding knight's tours and displaying that, we certainly accomplished them. The tours can be found extremely quickly, even with very large boards.. They can be displayed in the python terminal using ASCII characters for any size, and can be simulated as long as neither side is more than 30 long, due to the large window size required, and the time it takes to simulate. We achieved everything we wanted with this project, strengthened Discrete Math knowledge, Python expertise, and even some graphic processing.

Running the Program

Make sure to have pygame installed, the program will try to do so through pip if you don't, however it's likely best that you install it yourself. To run our project, inside the submitted folder, run driver.py and follow the instructions in the terminal.

References

- [1] Shun-Shii Lin, Chung-Liang Wei
Optimal algorithms for constructing knight's tours on arbitrary $n \times m$ chessboards,
Discrete Applied Mathematics, Volume 146, Issue 3, 2005, Pages 219-232,
(<https://www.sciencedirect.com/science/article/pii/S0166218X04003488>)