Safe and Efficient Multi-Stage Closed-Loop Planning for Snake Game

George Chustz

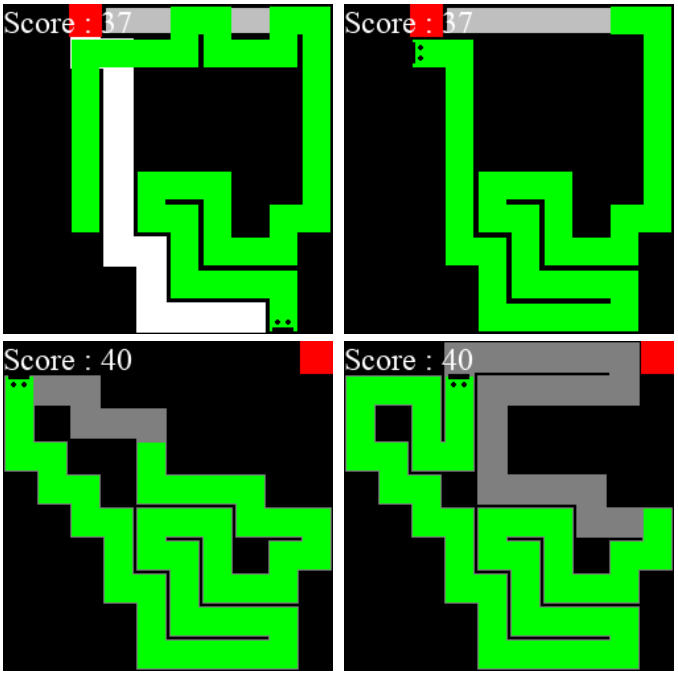

Fig. 1. Screenshots from the Multi-Stage Closed-Loop Planning implementation. The upper two images are from the beginning and end of a stage #1 planning cycle for the left and right images, respectively. For stage #1, the white path is the head-to-goal path and the light gray is the return path to the propagated tail position to form a closed-loop path. The lower two images are from the start and end of a stage #2 planning cycle for the left and right images, respectively. For stage #2 the closed-loop path is dark gray.

***Abstract*—In this work, the potentially impractical and time-wasting motion planning problem of the popular Snake Game is approached in a thorough manner to give it the precise treatment it deserves. This planning problem is formulated rigorously, and definitions of optimality and snake safety are presented and discussed. Two exact safe and optimal solutions are described for demonstrating complexity. The A* discrete optimal planner is implemented to find head-to-goal paths to illustrate the shortcomings of not considering future safety in this dynamic problem. A computationally efficient and safe multi-stage path planning implementation is detailed to advance the field of snake path planning and enhance the general merriment of the reader.**

***Index Terms*—Snake, Game, Planning, Algorithms, paper.**

I. Introduction

SNAKE GAME is a lasting cultural artifact from an earlier era of computing. The game, while simple in concept, has captivated multiple generations, and has a simple enough implementation that it was spread across a vast array of devices, including the iconic Nokia 6110. The aspirations of filling the play space with a perfectly wrapped virtual snake has eluded many people due to their lack of forward planning capabilities and poor reaction times resulting in many early "Game Over" scenarios.

Thankfully, there is a small group of individuals who have contributed algorithmic solutions for this problem which eliminates the need for the human-in-the-loop. This work adds another contribution to this narrow and important body of work in the form of a novel multi-staged planning implementation shown in Fig. 1 and providing rigorous detail to the problem formulations and derived definitions. This work is structured as follows:

1) Introduction: The remainder of this section covers the related work, theoretical background to provide context for this work, and contributions of this work.
2) Problem Formulation: Details the rules and corresponding C-Space representation of this problem, along with definitions of safety, optimality, and reachability for this planning context.
3) Optimal and Safe Planning Implementations Under the Connectivity Constraints: Two exact solutions are detailed and discussed, and the complexity of the exact solutions is related to similar problems in games and graph theory.
4) Discrete Optimal Planning Algorithm A* Discussion: The shortcomings for a simple A* implementation are shown, which is likely the most common first approach at this problem.
5) Multi-Stage Closed-Loop Planner: The safe and computationally efficient planning implementation is presented and detailed as the main contribution of this work.
6) Conclusion: Conclusions and future work opportunities are detailed.

A. Related Work

Recent developments in snake game planning implementations are largely documented in online personal blog posts [1] and online videos [2], [3]. First, Tapsell's project to control snake game on an actual Nokia 6110 screen includes an interesting planning implementation. The implementation begins with an initial Hamiltonian Curve which is using a fixed grid size given by the screen's resolution constraints. The implementation treats the nodes in the Hamiltonian Cycle as a 1D array, which is then used to search for opportunities to "shortcut" sections of the cycle to achieve the goal faster [1]. The algorithmic efficiency of this approach is remarkable; however,

one trade off is that short cutting the Hamiltonian Cycle in this manner leads to blocked off regions which the snake cannot quickly access in subsequent moves. There have been many attempts at using the discrete optimal planning algorithm A* [4] to achieve efficient planning implementations, and one of the most popular examples of this is an implementation from "Code Bullet" on Youtube [2]. This uses standard A* optimal planning in the beginning when there is high connectivity of the free space, and, once certain game progression criteria are met, the planner's costs are inverted such that it attempts to fill as much space as possible before capturing the goals [2]. Additionally, [2] presents the self-obstacle of the snake body for A* dynamically such that the planner can account for the future movement of the snake while executing the plan. Inspired by [2], Haidet publishes a vastly different planning implementation that exploits Hamiltonian Cycles by applying cuts and splices to the cycle to create new Hamiltonian Cycles which achieve the goal with fewer moves [3]. As the planar Hamiltonian Cycle problem is NP-Complete [5], it is necessary for Haidet to build a reduced set of valid modifications to the cycle which could be applied quickly instead of finding new Hamiltonian Cycles for every goal. These related works serve as the primary inspiration for this work to further development in snake game planning implementations.

B. Graph Connectivity and Hamiltonian Cycles

The approaches detailed in the following sections are reliant on the connectivity of graphs and make use of Hamiltonian cycles for this problem. Graph theory is a mature area of mathematics and the resulting graph properties, operations, and representations are heavily used in computer science problems. Within this planning problem, the reachability between nodes in sets is critical for the efficiency and safety of planning implementations, and node-to-node reachability is discussed in Section II. Additionally, the existence of and complexity of finding Hamiltonian Cycles within graphs is a critical component to some planning implementations. Both of these concepts will be explored further in the following sections.

C. Application: Coverage Planning

The set of real world applications for this family of snake game planning implementations is rather small, but various styles of coverage planning are clear examples. The family of coverage planners are focused on determining efficient paths which cover a space of interest while avoiding obstacles [6]. The classic example of a coverage planning implementation is a consumer robot vacuum. The formulation of the snake game planning problem could be represented as a coverage planning problem with a limited memory of visited locations or with targets of immediate interest. To extend the popular example, this problem is analogous to a robot vacuum that is given an ever-growing queue of localized messes. The objective of this robot vacuum would be to get to the current target mess location as quickly as possible while avoiding previously cleaned regions if able; thus while traversing from objective to objective the space gets incrementally cleaner.

D. Contributions

This work exhibits the following contributions for the planning problem:

1) A novel, safe, and efficient multi-stage closed-loop planning implementation for Snake Game which outperforms humans is presented and included in the included code base.
2) A thorough C-Space representation of the planning problem as a grid-graph using sets and sequences is detailed.
3) Definitions of safety, optimality, and reachability concepts which are useful in this problem formulation are discussed.
4) The complexity of the exact solutions are discussed using analogies to similar problems.

II. Problem Formulation

The Snake Game Problem is a seemingly simple but deceptively complex planning problem where the player is responsible for navigating the virtual snake through a two-dimensional (2D) fixed-size grid world towards the current goal.

A. Rules of the Game

The game environment or "world" consists as a fixed-size two dimensional grid of tiles which will be referred to as nodes and are represented by (x, y) pairs. Each node is connected to four other nodes along the two grid dimensions, even the out of bounds nodes beyond the fixed-grid size. The snake is represented by a connected sequence of nodes which are a subset of the nodes within the environment. The snake's head occupies the newest node in the sequence. Navigation happens in discrete game steps where the snake head will move one node in a commanded direction of the player if the commanded direction is not directly opposite of the previous direction moved (except if the snake is only one node long). If no movement or an invalid movement is selected, the snake head will advance forward by one node in the previously selected direction. When moving, the node occupancies composing the sequence representing the snake are shifted to the subsequent node's former position to follow the snake's head. In other words, if the snake head moves from $(1, 2)$ to $(2, 2)$ then the node preceding the snake's head will move from it's previous position of $(1, 1)$ to $(1, 2)$. At the achievement of each goal, the snake's size is extended by one component by adding a linkage where the tail was on the preceding step. Each goal is assumed to be sampled uniformly random from the set of unoccupied nodes.

This snake construction is implemented using a list representing the sequence of nodes covered by the snake's body where the front node is covered by the tail at index 0 and the last node is covered by the head at the maximum index. When the snake advances one node, the new node occupied by the snake's head is appended to the end of the sequence. If there is no goal reached that movement, the first node in the sequence is removed. The sequence is reindexed to start

at 0 at the new tail. This has the equivalent behavior as the shifting described in the previous paragraph.

The game is won when the entirety of the grid world is covered with the snake's body. The game is lost when the snake's head exits the bounds of of the grid world or the snake's head new node is already occupied by any other node of the new snake body. In practice, this means for any given valid game configuration, the safe actions are limited to navigating to a free node, goal node, or the node that is currently occupied by the tail of the snake which will be freed when the head moves to it. Notably, it is also crucial to ensure that there are enough free spaces beyond goals such that self collision after the snake grows does not occur. Additionally, a non-standard failure case is added where there is an upper bound to the number of moves that are allowed between goals such that the plan cannot cycle forever.

B. C-Space Representation

For a formal definition, there is a heavy focus on the accumulation of steps for the construction of the various configuration space (C-Space) sets where the superscript k means "at the k-th step" enumerated from 0 which is the starting configuration. The tail node and head node at step k are T^k and H^k, respectively. The subscript i denotes the index within the snake sequence which a general node N_i^k occupies at the k-th step. The snake's length in nodes is n. From this, it is apparent that $T^k = N_0^k$, $H^k = N_{n-1}^k$, and the snake propagation forward from step k to $k+1$ is

$$\{T^{k+1}, ..., H^{k+1}\} = \{N_1^k, ..., H^k\} \cup \{H^{k+1}\}$$

if the goal is not reached at H^{k+1} and the order is respected and

$$\{T^{k+1}, ..., H^{k+1}\} = \{T^k, ..., H^k\} \cup \{H^{k+1}\}$$

if the goal node G^k is reached at H^{k+1}. Additionally, the length at the next step n^{k+1} is incremented by one $n^k + 1$ if a goal is reached or otherwise remains equal to n^k. For simplicity, the sequence (or totally-ordered set) of nodes that comprise the snake at step k will be referenced as $S^k = \{T^k, N_1^k, ..., N_{n-2}^k, H^k\}$. For notation compactness, we will refer to S^k and paths as sequences and use set theory notation to describe operations upon them while maintaining the left-to-right ordering of elements therein.

The world is represented as a bipartite grid graph with a fixed size, $G(V, E)$. The bipartite structure is not fully utilized in any one planning phase, but it is important to recognize that traversal is bidirectional between connected vertices in this world and is only restricted by the single rule above. The vertices in V are equivalent to the aforementioned nodes and defined by the Cartesian product of every row and column index in the grid $V = \{(x, y) | \forall (x, y) \in X \times Y\}$ where $X = \{0, 1, 2, ..., n_x - 1\}$ and $Y = \{0, 1, 2, ..., n_y - 1\}$ where n_x and n_y are the number of rows and columns, respectively. The edges are defined as pairs in the node's adjacency set $(x, y) \to (x', y')$ where $(x', y') \in Adj((x, y))$; formally:

$$E = \{(x, y) \to (x', y') | \forall (x', y') \in Adj((x, y)), \forall (x, y) \in X \times Y\}$$

The adjacency set $Adj(\bullet)$ of a vertex (x, y) is equivalent to the set

$$Adj(x, y) = \{(x+1, y), (x-1, y), (x, y+1), (x, y-1)\}$$

which corresponds to the directions right, left, down, and up, respectively, as rendered and including the out-of-bounds regions.

It is critical to accurately and efficiently represent the C-space regions as the following sections require some of the following to make their plans effectively. There are two subsets of the total C-Space denoted C which is invariant across all steps, the free region C_{free}^k which can be safely used at step k written and the obstacle region C_{obs}^k which would result in a failure or "game over" state. The union of these sets is equivalent to the total C-space at each step k

$$C = C_{free}^k \cup C_{obs}^k$$

with no intersections

$$C_{free}^k \cap C_{obs}^k = \emptyset$$

and subsequently

$$C_{free}^k = C \setminus C_{obs}^k$$

which is what will the primary tool for rejecting head-adjacent nodes in the obstacle region. The obstacle region has two disjoint contributors, the out-of-bounds (OOB) region C_{OOB}^k and the sequence S^k of the nodes occupied by the snake at step k. It is important to note that both of these are trivial to check and propagate forward for steps $k + j$.

This chosen notation allows that the nodes within S^k are not in C_{obs}^{k+j} if their index $i < j$. Assuming that all of the paths planned do not self-intersect, this allows for precise calculation of the necessary path segment cost required to overcome nodes in S^k when planning forward from H^k simply by referring to their index i. This enables a simple dynamic treatment of C_{obs}^k and is trivial to see for the tail of the snake at index 0. If a path node adjacent to the current snake tail has cost greater than or equal to 0, the the current tail of the snake can be occupied by the head of the snake in the next turn from that node.

Valid paths will be denoted as

$$\pi_{[k,k+j]} = \{(x_k, y_k), (x_{k+1}, y_{k+1}), ..., (x_{k+j}, y_{k+j})\}$$

where

$$(x_{p+1}, y_{p+1}) \in Adj((x_p, y_p)) \setminus C_{obs}^{p+1}, \forall p \in [k, k+j-1]$$

and

$$(x_p, y_p) \neq (x_q, y_q), \forall p, q \in [k, k+j], p \neq q$$

An entire plan (or full valid game) is the sequence of paths from the start to finish of the game

$$\Pi = \cup_{g=0}^{n_{goals}-1} \pi_g \supset \cup_{g=0}^{n_{goals}-1} \{G^g\}$$

and $G^g \in \pi_g$ for $g = 0, 1, ..., n_{goals} - 1$ where the goal index g is the game step range $[k, k+j]$ for the g-th goal. In other words, goal node G^g is the singular goal node during the game steps $k, k+1, ..., k+j$.

C. Optimality

In this planning problem, optimality of paths is specifically focused on their length, which will be the cost measure for this work. The cost of a path is simply calculated by determined by counting the quantity of nodes in the path. An optimal path from H^k to G^k is a path $\pi^\star_{[k,k+j]}$ where there are no other paths between H^k and G^k which have lesser costs. Path $\pi^\star_{[k,k+j]}$ is not necessarily unique. It is also important to note that the minimum total game path $\Pi^\star$ is not necessarily the union of individual optimal paths. Since the goals are randomly sampled from C^k_{free}, $\Pi^\star$ will receive no treatment in our planning approaches, but consideration will be taken for future reaching future goal nodes. Additionally, the dymanic treatment of the snake obstacle in section II-B can be exploited to efficiently find optimal paths.

D. Safety

The definition of safety is a broad one, and specific targeted subsets of this definition will be used heavily in the upcoming sections. A game has a safe total game path if and only if each H^k is not in the obstacle set except the head $C^{k-1}_{obs} \setminus H^k$ for all k game steps in the game.

There are many, many paths that allow for safe planning, but there is one important subset of safe plans that are guaranteed safe which will be the cornerstone of this work, cyclic paths. Cyclic paths $\pi^\circ_{[k,k+j]}$ are paths which create a closed loop by enforcing that the path begins at a node in $Adj(H^k)$, optionally navigates the graph returning to $Adj(T^k)$, and traverses S^k in order. A cyclic path can be repeated indefinitely and the H^k will never enter C^k_{obs}. Usually optimal paths $\pi^\star$ and cyclical paths π° are mutually exclusive. However, a considerable amount of the Multi-Stage Path Planning will exploit goal optimal paths that are subsets of a cyclic path back to the projected tail position $\pi^\star \subset \pi^\circ$. An important caveat to the safety of cyclic path are the small subset of cases when goals are generated within the cyclic path and there are not enough free nodes in $\pi^\circ_{[k,k+j]}$ to accommodate the increases in size. The measure of the $|\pi^\circ_{[k,k+j]} \cap C^k_{free}|$ is called the *margin* within the path and will be used to avoid failures caused by this caveat.

E. Node-to-Node Reachability

Reachability between two nodes in the graph is an extremely important property which is used heavily in the following sections. Two nodes are reachable when there exists a path through the verteces of their graph to connect them. For this work, we consider two notions of reachability:

- Weak N-N'-reachable: $\exists \pi \Rightarrow N, N' \in \pi$ which means there exists a valid path π from N to N'.
- Strong N-N'-reachable: $\exists \pi \Rightarrow N, N' \in \pi \subseteq C^k_{free}$ which means there exists a valid path π from N to N' and the entire path is within C^k_{free}.

Two important notes are that strong N-N'-reachable implies weak N-N'-reachable and is easier to calculate, and that cyclic plans enforce weak H^k-T^k-reachability.

F. Connectivity Constraints

Another important consideration for optimality and safety is the connectivity of C^k_{free}. If C^k_{free} is not fully connected, then the planning algorithm risks wasting game steps waiting for a viable path into the goal connected region of C^k_{free}. In this respect, it would be worth choosing a path with higher cost that guarantees that all of C^{k+j}_{free} is connected after finishing the path. This constraint will be referred to as the **connectivity constraint**, and the objective of this is to reduce the overall cost of Π and provide safety guarantees.

This leads to two extensions of the reachability measures in the previous subsection:

- Weak connectivity constraint: A set $A \subseteq C$ is weakly connected if $N_i, N_j \in A$ is weak N_i-N_j-reachable for all $i \neq j$.
- Strong connectivity constraint: A set $A \subseteq C$ is strongly connected if $N_i, N_j \in A$ is strong N_i-N_j reachable for $i \neq j$.

Similarly, the strong connectivity constraint satisfies the weak connectivity constraint and is similarly easy to calculate. For the implementations in this paper, we will rely exclusively on the strong connectivity constraint applied to C^k_{free}, **referred to generally throughout this work as *the* connectivity constraint.**

This leads tangentially into an interesting set of results, which is $\exists N \in Adj(T^k) \Rightarrow N \in H^k \cup C^k_{free}$. This comes naturally as as the tail increments forward with game steps, either T^{k-1} is converted to a node in C^k_{free} or H^k and $T^{k-1} \in Adj(T^k)$. We use this result to assert that for a valid full game path Π requires that C^k_{free} is weakly connected for all k. Additionally, $\exists N \in Adj(H^k) \Rightarrow N \in T^k \cup C^k_{free} \forall (k, \Pi)$, meaning that for a full valid game, the head is always able to move into the C-free or to the space the tail was previously occupied. It is clear from the above that every Π requires that C^k_{free} always satisfies the weak connectivity constraint for all k and safety requires this constraint. Strong connectivity of C^k_{free} at k when a goal is achieved allows satisfaction of this constraint with less computation and guarantees strong H^k-G^k-reachability for all k.

III. Optimal and Safe Planning Implementations Under the Connectivity Constraints

There are two optimal and safe planning formulations which do not violate the connectivity constraint that will be considered in this section, an exact tree search and a Hamiltonian cycle approach. Both of which will have safety guarantees and are optimal under the connectivity constraint; however, the formulations described here are not practical, especially for large grid worlds. This section serves to demonstrate that the complexity of exact planning in this problem formulation is comparable to similar games such as chess.

A. Exact Search

The exact search involves the enumeration of a directed Tree-Graph $G(V, E)$ in which every walk along the graph either ends at the goal or at a dead end. The Vertices and

Edges sets are enumerated by propagating forward from the starting node, the snake head H^k. Each edge is from a source vertex representing a path $\pi = \{H^k, N^{k+1}, ..., N\} \in E$ up to node $N = (x, y)$ and directs to a node in its adjacency set $N' = (x', y')$ such that $N' \notin \pi$ which gives us the set of Vertices $V = \{(\pi \rightarrow \pi \cup N') | N' \in Adj(N) \cup C^N_{free} \backslash \pi\}$ where C^N_{free} is the C_{free} when N would be added. The graph G is enumerated until the goal is reached $G^k \in Adj(N) \cup C^N_{free} \notin \pi$ or a dead end is encountered $Adj(N) \cup C^N_{free} \backslash \pi = \emptyset$ for that branch. Once the graph is enumerated, the lengths of the paths can be counted by the number of nodes in the leaf vertices, and subsequently, all paths that meet the following criteria can be pruned in this order of priority:

1) The path hits a dead end.
2) The path bisects C^k_{free}.
3) The path has a cost that is higher than the minimum path cost after #1 and #2.

If this approach is adopted for every planning cycle, it is clear that the connectivity constraint will be enforced when each goal is reached, such that when a new goal is sampled randomly from C^{k+j}_{free}, it is guaranteed to be reachable by the snake head. This enforcement additionally guarantees that Safety is maintained, and the exact tree search pruning ensures that Optimality under Connectivity Constraints is maintained.

It is clear to see that the growth of this Tree-Graph is exponential with respect to the depth n. At every expansion, at least three children nodes are created from their respective parents representing the three possible directions from the parents most recent node. An analogy can be drawn to similar combinatorial games such as chess, which have been proven to be PSPACE-Complete for reasonable generalizations of the $N \times N$ board [7] and require EXPTIME for perfect strategies [8]. A rigorous proof will not be pursued in this work, but the comparison of the exact search for the snake game to similarly complex games motivates the need for inexact implementations as the execution of such a search would exceed any reasonable time requirement or the storage of look-up tables containing exact solutions would exceed any reasonable memory allocation.

B. Hamiltonian Cycle Planning

Hamiltonian Cycle Planning is an interesting method to solve this problem; a closed-loop path is considered which passes once through all of C^k_{free} and S^k, in order, without violating connectivity within S^k. This path is defined as $\pi^{H\circ,k} = \pi^{\circ,k} \cup S^k$ such that $|\pi^{H\circ,k}| = |C^k|$. This assumes that a Hamilton Cycle is possible in the bipartite grid graph G. Any $\pi^{H\circ,k}$ is guaranteed safe under the connectivity constraint and to complete the game, but it would be extremely uninteresting.

However, it is possible to convert one Hamilton Cycle in our graph G to another using valid cuts and splices between sets of nodes. If the Hamiltonian Cycle is transformed such that it includes an optimal cost path under the connectivity constraint, then a safe plan is maintained with this qualified optimality. This is similar to the approach taken by [3]; however, to find an optimal path in all cases while maintaining the Hamiltonian Cycle it is necessary to consider a large subset of the possible modifications to the current cycle such that the approach in [3] will not suffice. This necessitates rebuilding significant portions of the Hamiltonian Cycle after each goal, which is another Hamiltonian Cycle search problem if the existing segments are treated as nodes, which is proven to be NP-Complete for triply connected planar graphs [5]. This complexity of constructing plans as Hamiltonian Cycles again motivates the value of inexact implementations.

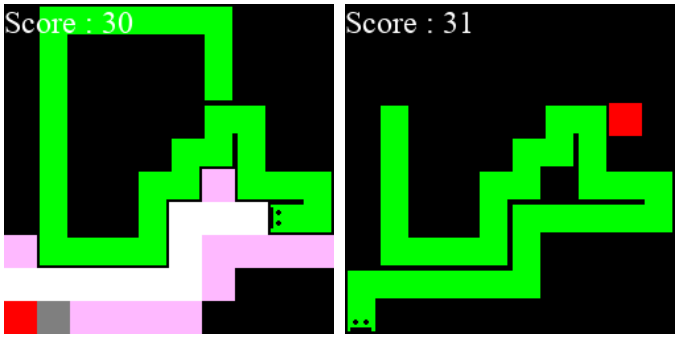

Fig. 2. Screenshots from the simple A* planning implementation. The left and right images are from the beginning of the planning cycle and the "Game Over" state at the end of the planning cycle, respectively. The white, dark gray, and blue tiles are the path, expanded nodes (most of which is hidden by the path), and the nodes in the priority queue of A* algorithm, respectively, at instant the plan was found. This figure illustrates the shortcomings of simply using A* for path planning in the Snake Game context as it is very likely to trap itself in a manner that no feasible future paths to goals can be found. In this case the snake continued off the game board after 248 moves because no valid path was found.

IV. Discrete Optimal Planning Algorithm A* Discussion

A* is an extremely popular algorithm which is guaranteed to be optimal if the heuristic used is admissible [4], and is a intuitive first consideration for this problem as it involves graph search from a start vertex to a goal vertex. In fact, many use a simple A* implementation as a first attempt or as a component of their broader algorithm [2], [3]. However, simple A* does not guarantee safety and frequently causes self-entrapment as shown in Fig. 2. The act of simply applying A* from H^k to G^k does not guarantee weak C^k_{free} connectivity which causes the self-entrapment; however it does return $\pi^{\star,[k,k+j]}$, the optimal path from H^k to G^k over j steps. Additionally, A* determines the feasibility of the path planning query by either returning a valid path or returning nothing. These two results will be used in the next section. Finally, an implementation of simple A* planning is included with the code base for demonstration purposes.

V. Multi-Stage Closed-Loop Planner

In the previous sections, it is shown that a complete game Π requires weak C^k_{free} connectivity for all k and that cyclic paths enforce strong H^k-G^k-reachability, yet a simple implementation using A* to plan from H^k to G^k is not sufficient to guarantee the weak connectivity constraint of C^k_{free}. However, the optimality of the paths generated from A* is a property which would be beneficial when $|C^k_{free}| \gg |S^k|$ particularly

in the early and middle game. Another useful property of A* is that it provides an infeasibility result when it cannot find it's goal from the starting node given the obstacles. Lastly, A* is computationally fast, especially when a feasible path exists.

This leads to the proposed multi-stage closed-loop implementation of this work, which has stages to address each phase of the game: beginning, middle, and end. During the beginning, there is relatively little within the C_{obs}^k set granting high, if not complete, connectivity of C_{free}^k. This motivates the heavy usage of the A* algorithm detailed in Stage #1 below to quickly find $\pi^\star$ and a candidate π°. During the middle and end portions, it becomes important to create closed loop paths which have far greater coverage of C_{free}^k to both block potentially inconvenient future goal positions and maintain safe traversal and expand stage #1 options by increasing the margin by **wasting time**, which is the motivation for the cheeky first sentence of the abstract. This motivates the greedy π° path changes in stage #2. However, pursuing these greedy changes inevitably leads to a path which has stable nodes within C_{free}^k that are not able to be reached without an exact solution. This leads to the optimization step in optional stage #3 which attempts to convert the path into a Hamiltonian Cycle by searching for changes between the head and the tail. After such a cycle is implemented, the snake will follow the path generated in stage #3 until the game is won. This is referred to as the **Multi-Stage Closed-Loop Planner** or the **planner** and contains three distinct stages when planning:

1) From H^k to G^k, A* is used to plan a $\pi^{\star,[k,k+\alpha]}$, if one exists. The snake sequence S^k is then projected forward using $\pi^{\star,[k,k+\alpha]}$ to a candidate snake of ${S'}^{k+\alpha}$. A* is again used to plan a $\pi^{\star,[k+\alpha,k+\alpha+\beta]}$ from ${H'}^{k+\alpha}$ to ${T'}^{k+\alpha}$, if one exists and this time considering the candidate snake as a static obstacle. If all of this is feasible, then it has quickly been determined that $\exists \pi^{\circ,[k+\alpha,k+\alpha+\beta+|{S'}^{k+\alpha}|]} \Leftarrow \pi^{\star,[k+\alpha,k+\alpha+\beta]} \cup {S'}^{k+\alpha}$ which is a safe closed-loop path to adopt after the goal is achieved. The plan $\pi^{\star,[k,k+\alpha]}$ can be executed without re-planning. If any of the above are infeasible, proceed to 2 immediately.
2) Adopt the previously determined safe closed-loop path created in the previous planning iteration $\pi^{\circ,[k-\Delta,k]}$. Perform a walk along $\pi^{\circ,[k-\Delta,k]}$ starting from the snake head and identify any simple cost-increasing, C_{free}^k connectivity increasing, goal capture, or space filling changes available make to the path while maintaining a safe closed-loop path, implement the first valid change in a greedy manner and adopt the new closed-loop plan $\pi^{\circ,k+}$ and try to re-plan with #1 again on the next game step. If no changes are found, lock the current path while still attempting #1 until #1 is successful. Optionally go to #3 when no further changes can be made using #2 when the Snake completes an entire loop of the path without triggering #1.
3) At this stage, a space filling path has been devised which covers most of the grid; however, since #2 is implemented in a greedy manner, there will likely be opportunities to increase the cost further using a Hamiltonian cycle. Once this stage is triggered, it is assumed that the snake is long enough that it is no longer valuable to pursue #1 or #2 and disrupt the near-Hamiltonian Cycle. For this construction, the segment of the closed loop between the head and current tail position is taken. If there are C_{free}^k nodes adjacent to this path segment, a backtracking depth-first search is used to identify if a series of moves could be taken to either (i) increase the length of the path segment safely or (ii) increase the connectivity of C_{free}^k safely without increasing the cost. Additionally, consideration is taken for changing the path to achieve a reachable goal if there is enough margin. If such a segment is found, it is adopted. #3 is repeated until the entire game board is filled and the snake will simply follow the cycle until game is won, assuming a cycle exists.

This planner is extremely fast and reliable with stages #1 and #2, shown in Fig. 1, easily outperforming humans in total number of moves and computation speed for reasonable board sizes (e.g. 30×30 tiles). With only stages #1 and #2 active, the snake is able to quickly acquire many goals in the beginning of the game, and handily switches between stage #1 and #2 until it reaches a safe and stable path which has no more available greedy changes to acquire goals or increase margin. Stage #3 is greatly impacted by the margin in both computational cost and feasibility of new solutions. If there is a large margin, solutions will take an infeasible amount of time to calculate superior paths; however, not enough margin does not allow for potential changes to be made to the path; thus, potential improvements are considered in the conclusion section. If stage #3 is triggered at an appropriate time, it is capable to make changes that increase the cost of the path. Figures of stage #3 are not present as this timing was not achieved as of the writing of this work. The multi-stage implementation is included with the code base.

VI. Conclusion

This work contains multiple contributions to the narrow field of Snake Game path planning. The primary contribution is the Multi-Stage Closed-Loop Planner implemented in Section V which takes inspiration from various previous works mentioned in Section I-A and utilizes various properties discussed in Section II. This planner leverages the fast nature of A* and greedy search algorithms to plan feasible paths to goals quickly or maintain the safe existing closed-loop path while increasing margins at a pace unmatched by humans. Another contribution from this work is the Problem Formulation which builds a detailed C-Space representation of the game in Section II-B, from which important definitions of valid safe paths, optimality, reachability, and connectivity are built to be used in the following sections and future works. Next, the complexity of exact solutions to the planning problem is presented by formulating two exact planning implementations and drawing analogies to known complexity results to motivate the need for inexact solutions such as the one presented in this work. Additionally, the Python implementations of the Multi-Stage Closed-Loop planner and the simple A* planner are released along with this manuscript for public access.

The primary avenue to improve this work is to improve the transition from Stage #2 to #3 and rework Stage #3 to more intelligently transition into Hamiltonian Cycle such as transitioning into a boustrophedon structure instead of attempting a costly search. There is also opportunity for further analysis on the cumulative cost of planning optimally for each goal as done in this work's implementation versus planning specifically to maintain C_{free}^{k} connectivity and determining if there is an advantage for one methodology over another.

REFERENCES

[1] J. Tapsell. Nokia 6110 part 3 - algorithms. [Online]. Available: https://johnflux.com/2015/05/02/nokia-6110-part-3-algorithms/

[2] E. C. Bullet". I created a perfect snake a.i. Youtube. [Online]. Available: https://www.youtube.com/watch?v=tjQIO1rqTBE

[3] B. Haidet. How to win snake: The unkillable snake ai. Youtube. [Online]. Available: https://www.youtube.com/watch?v=TOpBcfbAgPg

[4] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[5] M. R. Garey, D. S. Johnson, and R. E. Tarjan, "The planar hamiltonian circuit problem is np-complete," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 704–714, 1976. [Online]. Available: https://doi.org/10.1137/0205049

[6] E. Galceran and M. Carreras, "A survey on coverage path planning for robotics," *Robotics and Autonomous Systems*, vol. 61, no. 12, pp. 1258–1276, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092188901300167X

[7] J. A. Storer, "On the complexity of chess," *Journal of Computer and System Sciences*, vol. 27, no. 1, pp. 77–100, 1983. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022000083900302

[8] A. S. Fraenkel and D. Lichtenstein, "Computing a perfect strategy for n × n chess requires time exponential in n," *Journal of Combinatorial Theory, Series A*, vol. 31, no. 2, pp. 199–214, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0097316581900169