Maximizing Candy Collection: A Graph-Based Dynamic Programming Approach

Esther Moran, Evan Li, Peilin Wu, Elizabeth Bai
Carnegie Mellon University

Dec 15, 2025

Abstract

This paper examines the decline of traditional trick-or-treating through the lens of optimization and community engagement. By modeling trick-or-treating as a dynamic programming problem on graph structures, we explore how route efficiency and candy yield could influence participation. We propose and compare one-dimensional (1D) and two-dimensional (2D) models—both directed and undirected—demonstrating that a Directed Acyclic Graph (DAG) model is computationally scalable, while an Undirected Graph model better captures realistic neighborhood movement. We conclude by discussing how this approach may inform community revitalization and broader applications in logistics and outreach planning.

1. Introduction

Over recent years, the number of children who participate in trick-or-treating has steadily declined. While the causes are debated—ranging from safety concerns to changing weather patterns and shifting community engagement—the loss of this tradition represents more than the disappearance of candy collecting. Trick-or-treating fosters social development, creativity, and local connectedness. It encourages children to engage offline, build confidence through self-expression, and interact with their community in a uniquely positive context.

Reinvigorating interest in trick-or-treating may therefore have social value. Current efforts to attract participants—such as decorative displays or themed lighting—are effective in drawing existing trick-or-treaters but do little to reach those who have opted out. Our project hypothesizes that increasing the expected candy payoff per outing could encourage participation among these non-participants. In essence, if the expected "wage" (candy) increases, the "labor supply" (participation) should rise—a behavioral analogy inspired by economic theory.

To test this hypothesis, we designed a computational model that identifies routes maximizing expected candy collection subject to time, distance, and capacity constraints. We first construct a 1D algorithm to model a single street, and then extend to 2D networks represented as directed or undirected graphs.

2. Data Collection and Assumptions

2.1. Data Collection

Our initial dataset focuses on Bennington Avenue, a residential street near Carnegie Mellon University. We focused on houses along one side of the street and recorded the inter-house distances using Google Map. Corresponding home price estimates were obtained from publicly available Zillow data.

Walking time between houses was calculated by assuming an average human walking speed of 4 feet per second. Additionally, we assumed a constant 5-minute waiting period at each house to represent the time required for candy collection.

To model expected candy yield per house, we introduced a heuristic that wealthier households tend to distribute more candy. While this assumption is a simplification, it serves as a practical and quantifiable proxy for modeling purposes. We mapped each home's price to an expected candy value using a logistic S-shaped function:

$$c(x) = 2 + \frac{8}{1 + e^{-(x-\mu)/k}}$$

where μ denotes the mean home price across all selected houses, and k is a tunable parameter that controls the steepness of the curve. This function bounds the expected candy between 2 and 10 units per house, allowing for smooth, non-linear scaling with respect to wealth.

2.2. Model Constraints

The optimization model is governed by three dynamic resource constraints: time, distance, and carrying capacity. Each constraint evolves as the trick-or-treater progresses through the neighborhood and directly influences the set of feasible future actions.

Let T, D, and H denote the remaining time, distance, and candy-carrying capacity, respectively. The time budget T decreases after each transition according to the walking time between two houses and, if applicable, an additional fixed waiting time $w(i)$ incurred when collecting candy at house i. The distance budget D is reduced by the edge length $d(i, j)$ corresponding to the distance traveled between houses i and j. The carrying ca-

pacity H represents the total amount of candy that can be held at any point in time; it is decremented by the expected candy yield $c(i)$ each time candy is collected at house i.

Together, these evolving quantities define the feasible region of the dynamic program. A transition from house i to house j is considered valid only if the updated state satisfies $T > 0$, $D > 0$, and $H > 0$. Once any of these limits is exhausted, the recursion terminates, ensuring that all generated routes remain physically and temporally plausible.

3. One-Dimensional Model (1D Formulation)

Our initial model considers a single street of houses arranged sequentially. This simplified setting allows us to establish the foundation for dynamic programming logic before extending to multi-directional neighborhoods.

In this formulation, each house r represents a decision point where the trick-or-treater must decide whether to stop and collect candy or continue walking. The objective is to maximize the total expected candy collected under the three constraints: time remaining , distance remaining, and carrying capacity.

Each house r has an expected candy yield $c(v_r)$, a walking time $t_{r,r+1}$ and distance $d_{r,r+1}$ to the next house, and a waiting time w_r (assumed constant at 5 minutes). At every house r, the participant chooses between collecting candy or skipping the stop. The dynamic programming recurrence is therefore defined as:

$$f(r, T, H, D) = \max \begin{Bmatrix} c(v_r) + f(r+1,\ T - t_{r,r+1} - w_r,\ H - c(v_r),\ D - d_{r,r+1}), \\ f(r+1,\ T - t_{r,r+1},\ H,\ D - d_{r,r+1}) \end{Bmatrix}.$$

$$f(r, T, H, D) = 0 \quad \text{if } T \leq 0,\ H \leq 0,\ D \leq 0, \text{ or if } r \text{ exceeds the last house.}$$

Implementation provided in Appendix A.1

3.1. Results

Along Bennington Avenue, we tested different configurations of maximum distance, time, and holding capacity (code provided in Appendix). The results are as follows:

Table 1: Candies Gained with Different T, D, H Configurations

Time (s)	Distance (ft)	Holding Capacity	Candies Gained
500	150	20	16.681
400	100	20	11.158
300	100	30	11.158

We note that these choices of time, distance, and holding capacity are unrealistic and that we would expect larger numbers on all three variables. However, we only taking data over one street, which is also much smaller than a realistic trick-or-treating route. Hence, we modified our values for time, distance, and holding capacity as above to ensure that they do have an effect on our outcomes. We further note that our program did not restrict the outcome to positive integers, hence the decimal values above. Thus, we simply took the floor of our results to represent the total amount of candies collected.

4. Two-Dimensional Model (2D Formulation)

While the 1D approach proved effective for a linear street, it could not fully represent the interconnected nature of real neighborhoods. Most trick-or-treating routes involve intersections, branching paths, and the possibility of returning to previously visited locations. To address this, we extended our formulation into a two-dimensional model using graph theory.

4.1. General Graph Recurrence

The two-dimensional formulation naturally extends the one-dimensional case by allowing each house to be connected to multiple neighboring houses. This structure more accurately reflects real neighborhoods, where participants can choose among several possible routes at each intersection. At each step in the recursion, the model considers all possible neighboring nodes that can be reached next, while accounting for the time, distance, and capacity constraints that limit further movement.

The recursive formulation implicitly captures both the act of stopping to collect candy and the alternative of walking past a house. This trade-off is formalized in the following recurrence:

$$f(i,T,H,D) = \max_{j \in N(i)} \begin{Bmatrix} c(i) + f\big(j,\ T - t(i,j) - w(i),\ H - c(i),\ D - d(i,j)\big), \\ f\big(j,\ T - t(i,j),\ H,\ D - d(i,j)\big) \end{Bmatrix}.$$

$$f(i,T,H,D) = 0 \ \text{ if } \ T \le 0 \text{ or } H \le 0 \text{ or } D \le 0$$

The first term represents the case in which the participant collects candy at house i before moving on to j, while the second term represents passing the house without stopping. By evaluating all possible transitions recursively, the model identifies the route that maximizes total expected candy while respecting the participant's remaining time, capacity, and allowable travel distance.

4.2. DAG Subcase: Acyclic Neighborhoods

We next consider a constrained variant of the general graph formulation in which the neighborhood topology is acyclic. In this setting, the graph can be represented as a collection of trees or forests, where each edge connects distinct nodes without forming cycles.

The recurrence relation for the DAG formulation remains identical to that of the general model:

$$f(i,T,H,D) = \max_{j \in N(i)} \begin{Bmatrix} c(i) + f\big(j,\ T - t(i,j) - w(i),\ H - c(i),\ D - d(i,j)\big), \\ f\big(j,\ T - t(i,j),\ H,\ D - d(i,j)\big) \end{Bmatrix}.$$

$$f(i,T,H,D) = 0 \ \text{ if } \ T \leq 0 \text{ or } H \leq 0 \text{ or } D \leq 0$$

This structure guarantees a single forward evaluation path for each node, avoiding redundant subcalls and enabling a consistent definition of optimal substructure across the network. The DAG-based formulation thus provides a mathematically well-defined foundation for computing maximum expected candy yield in environments with non-cyclic connectivity.

4.3. Undirected Graph: Richer Movement and Realistic Exploration

In contrast to the DAG model, we also consider the fully undirected graph case, where each house may be revisited and movement is allowed freely in both directions. This formulation aims to more accurately reflect real-world neighborhoods, where children may choose to backtrack or take winding paths in search of better candy hauls.

However, this flexibility comes at a computational cost. Allowing cycles introduces the risk of revisiting the same house multiple times, potentially creating infinite loops. To address this, we explicitly maintain a `visited` set during our recursive search. Each time a house is visited, it is marked in the set, and all future recursive calls avoid returning to any node already in `visited`.

This change ensures correctness while still enabling exploration of cyclic paths. The recurrence mirrors the general graph formulation, but now with the added condition that $j \notin V$, the set that contains all visited houses:

$$f(i,T,H,D,V) = \max_{j \in N(i) \setminus V} \begin{Bmatrix} c(i) + f\big(j,\ T - t(i,j) - w(i),\ H - c(i),\ D - d(i,j),\ V \cup \{i\}\big), \\ f\big(j,\ T - t(i,j),\ H,\ D - d(i,j),\ V \cup \{i\}\big) \end{Bmatrix}.$$

$$f(i,T,H,D,V) = 0 \ \text{ if } \ T \leq 0 \text{ or } H \leq 0 \text{ or } D \leq 0$$

Implementation provided in Appendix A.2

4.4. Experimental Setup

To evaluate our two-dimensional formulations under controlled conditions, we generated synthetic neighborhood instances using Monte Carlo simulation. For each trial, we constructed a grid-based neighborhood and sampled node and edge attributes stochastically to produce diverse test cases while keeping the overall structure fixed.

For the undirected setting, we begin with a standard $m \times m$ grid graph in which each node is connected to its four-neighborhood when applicable. Edge distances $d(i, j)$ and travel times $t(i, j)$ were sampled from bounded distributions centered around realistic neighborhood values (e.g., 150 ft per segment and 40 seconds per segment), introducing variability while keeping magnitudes consistent across trials.

To test the DAG formulation on comparable instances, we additionally derived a directed acyclic version from the same underlying grid by imposing a consistent edge orientation (e.g., only allowing transitions north and east relative to the starting corner). This orientation was used only for experimental evaluation to ensure acyclicity and does not represent a modeling claim about real neighborhoods.

All experiments were repeated across multiple Monte Carlo trials to account for randomness in $c(i)$, $d(i, j)$, and $t(i, j)$. All simulation code, solvers, and timing scripts were implemented in Python and executed in Google Colab; complete implementation details and experiment scripts are provided in Appendix A.3

4.5. Results

Table 2: DAG vs. Undirected Graph Comparison

Grid	Model	Max Candy	Runtime (s)	Observation
4×4	DAG	21.9	0.001	Fast, efficient
4×4	Undirected	23.8	1.8	Slightly higher candy yield
5×5	DAG	26.8	0.03	Scales efficiently
5×5	Undirected	27.0	161.8	Realistic but costly

The DAG model completed near-instantly, producing optimal paths with modest candy gains. The undirected model collected marginally more candy but required two to three orders of magnitude longer computation time. This result supports the conclusion that the DAG model scales better for large neighborhoods, whereas the undirected version is more suitable for small, realistic simulations.

The DAG approach provides a scalable and computationally efficient framework for large datasets. It can serve as a near-optimal approximation for real-world trick-or-treating when computational resources are limited. In contrast, the undirected model captures realistic street layouts and path flexibility but quickly becomes infeasible as the neighborhood size grows due to combinatorial explosion.

5. Future Work and Applications

5.1. Future Extensions

Some of the shortcomings of our model include its storage and processing requirements. As our worst case time is polynomial time, and it includes the necessity of n2 size matrix to store the adjacencies, this is not the quickest algorithm. We hope that future researchers may pick up our study with intent to streamline or provide an alternative that reduces the time and space requirements to run.

Due to time constraints, we were unable to implement certain more elaborate expansions to this problem, specifically in the possibility of adding more realistic mechanics and constraints. For example, the possibility of different neighborhoods and alternate transportation methods such as cars or bikes. Some other more recent concerns about trick-or-treating, such as unsafe treats given out, could also be accounted for through our heuristic function for estimating house candy value.

Furthermore, in the future, some broader avenues to explore would be how the heuristic function we're using to determine the candy harvest from houses may impact the pathing. Our assumption that the expected amount of candy is a function of the price of the house is very lacking. If we had more time we would have used other factors, such as house decoration, trick or treating and collecting the data in real time, surveying houses to see how much candy they plan to give out, and surveying past trick or treaters to see how much candy houses they have visited before have given out.

5.2. Other Applications

This problem formulation focuses on maximizing Halloween candy, but it also is fairly applicable or can extrapolate to other similarly critical issues. For example, when fundraising or outreaching door to door, our program should be able to give the optimal route to raise the most donations or garner support for an important cause.

Some other important applications of our problem include optimal campaigning locations to visit, which is critical for electing candidates into power; the most efficient trip guides

for vacations; resource gathering, and the variety of adjacent and subsidiary problems.

6. Conclusion

Through dynamic programming, we modeled optimal candy collection routes that account for time, distance, and carrying constraints. Both 1D and 2D models yield flexible frameworks for future adaptations. Our results suggest that while undirected models capture realism, DAG-based methods are the most practical for large datasets and real-time simulations. Ultimately, this work highlights how even a simple Halloween tradition can illustrate core principles of optimization, decision theory, and computational trade-offs.

A. Code Appendix

A.1. 1-D Formulation Code

```python
def maximize_candy_1d(houses, r, T, H, D):
    """
    f(r, T, H, D) = max { c(x) + f(r+1, T-tr, H-c(x), D-dr) }
    Parameters:
        houses : dict of houses keyed by r
        r      : current house index
        T      : time remaining (seconds)
        H      : candy capacity remaining
        D      : distance remaining (feet)
    """

    # Base case
    if r not in houses or T <= 0 or H <= 0 or D <= 0:
        return 0

    # Retrieve house info
    house = houses[r]
    c_x = house["expected value"]      # expected candr
    dr = house["distance (ft)"]        # distance cost directly from CSV
    tr = dr/4 + 5                      # time cost

    # Option 1: skip this house
    skip = maximize_candy_1d(houses, r + 1, T, H, D)

    # Option 2: take this house
    take = 0
    if T >= tr and D >= dr and H >= c_x:
        take = c_x + maximize_candy_1d(houses, r + 1, T - tr, H - c_x, D -
        ↪ dr)
    if T >= tr and D >= dr and H >= c_x:
        take = c_x + maximize_candy_1d(houses, r + 1, T - tr, H - c_x, D -
        ↪ dr)

    return max(skip, take)
```

A.2. 2D Grid Experiments (DAG vs. Undirected)

```python
def f_dag(i, T, H, D, graph, candy):
    if T <= 0 or H <= 0 or D <= 0:
        return 0
    best = 0
    wait_time = 300  # 5 minutes to pick
    for (j, dist_ij, time_ij) in graph.get(i, []):
        # Pick candy at i
        if T >= time_ij + wait_time and D >= dist_ij and H >= candy[i]:
            best = max(best, candy[i] + f_dag(j, T - time_ij - wait_time,
            ↪  H - candy[i], D - dist_ij, graph, candy))
        # Skip candy at i
        if T >= time_ij and D >= dist_ij:
            best = max(best, f_dag(j, T - time_ij, H, D - dist_ij, graph,
            ↪  candy))
    return best

def f_undir(i, T, H, D, graph, candy, visited):
    if T <= 0 or H <= 0 or D <= 0:
        return 0
    visited = visited | {i}          # mark current as visited (no
    ↪  revisits)
    best = 0
    wait_time = 300                  # 5 minutes to pick

    for (j, dist_ij, time_ij) in graph.get(i, []):
        if j in visited:
            continue

        # Pick candy at i
        if T >= time_ij + wait_time and D >= dist_ij and H >= candy[i]:
            take = candy[i] + f_undir(j, T - time_ij - wait_time, H -
            ↪  candy[i], D - dist_ij, graph, candy, visited)
            best = max(best, take)

        # Skip candy at i
        if T >= time_ij and D >= dist_ij:
            skip = f_undir(j, T - time_ij, H, D - dist_ij, graph, candy,
            ↪  visited)
            best = max(best, skip)

    return best
```

A.3. Grid Setup Functions

```python
def build_grid_candy(rows, cols):
    """Candy increases modestly toward NE; deterministic."""
    candy = {}
    for r in range(rows):
        for c in range(cols):
            node = r * cols + c + 1
            candy[node] = round(rng.uniform(0, 7), 1)
    return candy

def build_graphs(rows, cols, step_ft=150, step_sec=40):
    """Return DAG (east+north) and Undirected (4-neighbor grid)."""
    graph_dag, graph_undir = {}, {}
    for r in range(rows):
        for c in range(cols):
            node = r * cols + c + 1
            dag_edges, undir_edges = [], []

            # DAG: only east and north
            if c + 1 < cols:
                dag_edges.append((r * cols + (c + 1) + 1, step_ft,
                ↪ step_sec))
            if r + 1 < rows:
                dag_edges.append(((r + 1) * cols + c + 1, step_ft,
                ↪ step_sec))
            graph_dag[node] = dag_edges

            # Undirected: all 4 directions
            for dr, dc in [(0,1),(1,0),(-1,0),(0,-1)]:
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols:
                    undir_edges.append((nr * cols + nc + 1, step_ft,
                    ↪ step_sec))
            graph_undir[node] = undir_edges

    return graph_dag, graph_undir
```