

DAアルゴリズムを使った救急隊員と病院の優先順序付き1対多マッチングについて

横須賀市立横須賀総合高等学校 コンピュータ部 2年 大月 誉貴

開発の経緯

高校1年時に自宅の事故で救急搬送された際、救急車の到着後搬送先が決まるまで30分以上を要した。地元横須賀市消防局が公表している「消防年報(令和5年版)」において、令和4年の救急車到着から病院収容までの平均所要時間は33.8分を要している(表2)(参考:全国平均36.9分)病状が回復後、時間を要する原因を消防局に問い合わせたところ、以下の問題が判明した。またヒアリングを進めながら救急隊員は凄惨な現場に立会うストレスに加え搬送先が決まらない焦りや重なりメンタルが疲弊している事実を知った。以上の現状から救急隊員の負担を軽減するため自分に何かできないか考えた結果、救急隊員(患者)と病院双方が納得する結果を導き出し加えて搬送時間を短縮するシステムの開発を思い立った。(救急搬送に関する問題)

- 救急出動件数の増加(表1)
令和4年は平成25年と比較すると1.2倍。
- 現場到着時間と病院収容時間の延伸(表2)
出動件数増加により現場到着時間が延伸。
救急隊員が病院と個別交渉で搬送先を決めるため病院収容時間が延伸。
- 搬送数と空病床の全体把握ができない
消防局が搬送数と空病床数を一元管理していないため全体把握ができない。
- 患者が必要とする病院に搬送されない(表3)
軽症者の割合が45%と高く傷病程度と必要な病院のマッチングが難しい。
大病院は重症者を希望するが軽症者が搬送される等病院側が不満を持つ。

過去10年間の救急出動件数(表1) (単位:件)

H25	H26	H27	H28	H29	H30	R元	R2	R3	R4
23,129	22,696	22,960	23,004	26,956	27,865	27,598	24,307	24,947	28,044

過去5年間の救急関係所要時間(表2) (単位:分)

	H30	R元	R2	R3	R4
現場到着平均所要時間 A	7.8	7.7	7.2	7.3	8.0
病院収容平均所要時間 B	40.4	40.2	38.9	38.9	41.8
現場到着から病院収容時間 B-A	32.6	32.5	31.7	31.6	33.8

傷病程度別搬送人数割合(表3)

	死亡	重症	中等症	軽症	その他	合計
人数(人)	381	1,448	11,672	11,036	80	24,617
割合(%)	1.5	5.9	47.4	44.9	0.3	100

出典:令和5年版 横須賀市消防局 消防年報 抜粋一部加工

システム構築方法の検討

システム構築する際に採用するアルゴリズムについて、ゲーム理論の応用分野であるマッチング理論のうち、仮受け入れを繰り返し最適なマッチングを導き出すDA(Deferred Acceptance)アルゴリズム(以下、DA)を軸に他の方法と比較検討した。

(検討1) AI との比較
DAは結果を導き出す過程が明確であるが、AIは判断根拠がブラックボックス化されているため救急隊員(患者)と病院双方が結果に納得できない可能性がある。また学習データの内容によりバイアスがかかる可能性がある。救急現場では判断の透明性が重要であるためDAが望ましい。

(検討2) TTC (Top Trading Cycle) との比較
DAは安定性(他の選択肢を考える必要がない状態)と耐戦略性(本当の希望を隠しても得をしない状態)を満たすが、TTCは効率性(自分の希望を最大限反映する)と耐戦略性(満たすアルゴリズムである)を満たす。

救急現場では個人の希望より全体最適が重要であるためDAが望ましい。検討の結果、救急隊員(患者)と病院双方が後で不満を持たないことが重要であると考え、DAを採用する。システム開発はPythonで行う。

消防局へのプレゼンテーション

消防局に依頼し、DAを用いた救急隊員(患者)と病院双方が納得できるマッチングについて、動作イメージとPythonの開発環境のプレゼンテーションをさせていただいた。

消防局からは、「救急隊員は生体情報等複数の要素を勘案して病院を選定しており、同様の作業をシステムで代替することは緊急を要する現場では入力に時間がかかり困難である。しかし、多数の傷病者が発生した場合のトリアージにおいては生体情報等の収集を限定的にするため、入力項目を簡潔にしたシステムであれば提案内容を生かせる。」とご指摘いただいた。ご指摘を踏まえ、GUIで入力可能なPythonをベースとしたトリアージ発生時のマッチングシステムを開発した。

システム設定条件と動作イメージ 【プレゼンテーションの様子】

(条件) 患者は10人。救急隊員は患者を赤(重篤)、黄(重症)、緑(軽傷)に分け赤、黄、緑の順で受付順に附番する。また、患者毎に病院の嗜好を持つ。病院はA、B、Cの3病院。受付順位は全病院赤、黄、緑で番号の若い順から受入る統一ルール。各病院受入人数に制限。配分 仮決定 再配分を繰り返し最適なマッチングを実現。

(動作イメージ)

1回目処理: 救急隊員(患者)と病院のマッチングを開始。患者は赤、黄、緑の順で受付順に付番される。病院はA、B、Cの3病院。受付順位は全病院赤、黄、緑で番号の若い順から受入る統一ルール。各病院受入人数に制限。配分 仮決定 再配分を繰り返し最適なマッチングを実現。

2回目処理: 仮決定された患者と病院のマッチングを確認。患者は赤、黄、緑の順で受付順に付番される。病院はA、B、Cの3病院。受付順位は全病院赤、黄、緑で番号の若い順から受入る統一ルール。各病院受入人数に制限。配分 仮決定 再配分を繰り返し最適なマッチングを実現。

3回目処理: 仮決定された患者と病院のマッチングを確認。患者は赤、黄、緑の順で受付順に付番される。病院はA、B、Cの3病院。受付順位は全病院赤、黄、緑で番号の若い順から受入る統一ルール。各病院受入人数に制限。配分 仮決定 再配分を繰り返し最適なマッチングを実現。

4回目処理: 仮決定された患者と病院のマッチングを確認。患者は赤、黄、緑の順で受付順に付番される。病院はA、B、Cの3病院。受付順位は全病院赤、黄、緑で番号の若い順から受入る統一ルール。各病院受入人数に制限。配分 仮決定 再配分を繰り返し最適なマッチングを実現。

プログラムの説明

入力画面(GUI)

- 患者数(チェック欄)
- 重症度
- 希望病院
- 各病院の色毎の受入人数を入力し、
- 計算ボタンを押すと
- 算出結果が表示される。

算出結果
A病院が受入れる重篤患者(赤)は1、3 重症患者(黄)は7 軽症患者(緑)は8
B病院が受入れる重篤患者(赤)は2 重症患者(黄)は5 軽症患者(緑)は10
C病院が受入れる重篤患者(赤)は4 重症患者(黄)は6 軽症患者(緑)は9

プログラム

```

def main():
    # 患者リストの初期設定
    patients = [{"id": 1, "severity": "Red", "hospital": "A"}, {"id": 2, "severity": "Red", "hospital": "B"}, {"id": 3, "severity": "Red", "hospital": "C"}, {"id": 4, "severity": "Yellow", "hospital": "A"}, {"id": 5, "severity": "Yellow", "hospital": "B"}, {"id": 6, "severity": "Yellow", "hospital": "C"}, {"id": 7, "severity": "Green", "hospital": "A"}, {"id": 8, "severity": "Green", "hospital": "B"}, {"id": 9, "severity": "Green", "hospital": "C"}, {"id": 10, "severity": "Green", "hospital": "A"}]

    # 病院の初期設定
    hospitals = [{"name": "A", "capacity": 10}, {"name": "B", "capacity": 10}, {"name": "C", "capacity": 10}]]
    
```

1-4行目:システムに必要なモジュールのインポートや変数の初期設定をしている。
5-9行目:患者の重症度を表す定義Severityを設定している。重症度は上から赤黄緑の順に123と順位をつけ赤の1が最も重症度が高い。10-28行目:Kanjyaクラスで患者についての情報(患者の名前、病院の希望順、重症度)を整理し、仮の病院決定の結果の判定、また重症度によって色も設定している。
28-54行目:Hospitalクラスで病院についての情報(病の名前前、重症度ごとの定員、患者の希望順)を整理し、重症度に基づいて患者を仮の患者リストに追加し、また追加する際に、定員が満たされていない場合は患者を仮リストに追加し、満たされている場合は希望順に基づいて既存の患者と比較して仮リストに追加するか判断する。

56~72行目:matching関数で患者と病院のマッチングを行う。
まず患者のリストをコピーし、一時的に使用するプールリストを作成する。次にプールリストに入っている患者の情報を、各患者の希望順に病院へ割り当てる。病院側から拒否された患者は再びプールリストへ戻り次の希望順の病院を割り当てる。この作業を繰り返し、最終的にマッチング結果を算出する。

```

def matching(patients, hospitals):
    # 患者のリストをコピーし、プールリストを作成
    pool = patients.copy()

    # プールリストに入っている患者の情報を、各患者の希望順に病院へ割り当てる
    for patient in pool:
        # 患者の希望順に病院を割り当てる
        for hospital in hospitals:
            # 病院の定員が空いている場合、患者を仮リストに追加
            if hospital.capacity > 0:
                hospital.capacity -= 1
                patient.hospital = hospital.name
                break

    # マッチング結果を算出
    result = {}
    for hospital in hospitals:
        result[hospital.name] = []
        for patient in pool:
            if patient.hospital == hospital.name:
                result[hospital.name].append(patient.id)

    return result
    
```

```

def gui():
    # Tkinterを用いたGUI化するためのプログラム
    root = tk.Tk()
    root.title("救急現場マッチングシステム")
    root.geometry("800x600")

    # 重症度の変数
    severity_var = tk.StringVar(value="RED")
    preference_var = tk.StringVar(value="pref")
    hospital_pref_var = tk.StringVar(value="A")
    capacity_var = tk.StringVar(value="10")

    # 患者リストの初期設定
    patients = [{"id": 1, "severity": "Red", "hospital": "A"}, {"id": 2, "severity": "Red", "hospital": "B"}, {"id": 3, "severity": "Red", "hospital": "C"}, {"id": 4, "severity": "Yellow", "hospital": "A"}, {"id": 5, "severity": "Yellow", "hospital": "B"}, {"id": 6, "severity": "Yellow", "hospital": "C"}, {"id": 7, "severity": "Green", "hospital": "A"}, {"id": 8, "severity": "Green", "hospital": "B"}, {"id": 9, "severity": "Green", "hospital": "C"}, {"id": 10, "severity": "Green", "hospital": "A"}]

    # 病院の初期設定
    hospitals = [{"name": "A", "capacity": 10}, {"name": "B", "capacity": 10}, {"name": "C", "capacity": 10}]]
    
```

```

tk.Label(root, text="患者の重症度(スペース区切り):").grid(row=0, column=0, columnspan=7, padx=(8, 2))
row += 1
for hospital in hospitals:
    tk.Label(root, text=f"病院 {hospital.name}").grid(row=0, column=0, sticky="w", padx=2, pady=1)
    tk.Entry(root, textvariable=capacity_var(hospital.name)).grid(row=0, column=1, columnspan=2, sticky="we", padx=2, pady=1)
    row += 1
# REDの定員人数
tk.Label(root, text="定員人数(RED)").grid(row=0, column=0, padx=2, pady=1)
tk.Entry(root, textvariable=capacity_var(hospital.name)[RED]).grid(row=0, column=1, padx=2, pady=1)
row += 1
# YELLOWの定員人数
tk.Label(root, text="定員人数(YELLOW)").grid(row=0, column=0, padx=2, pady=1)
tk.Entry(root, textvariable=capacity_var(hospital.name)[YELLOW]).grid(row=0, column=1, padx=2, pady=1)
row += 1
# GREENの定員人数
tk.Label(root, text="定員人数(GREEN)").grid(row=0, column=0, padx=2, pady=1)
tk.Entry(root, textvariable=capacity_var(hospital.name)[GREEN]).grid(row=0, column=1, width=3, padx=2, pady=1)
row += 1
button = tk.Button(root, text="計算", bg="pink")
button.grid(row=0, column=0, columnspan=7)
root.mainloop()
    
```

算出結果
A病院が受入れる重篤患者(赤)は1、3 重症患者(黄)は7 軽症患者(緑)は8
B病院が受入れる重篤患者(赤)は2 重症患者(黄)は5 軽症患者(緑)は10
C病院が受入れる重篤患者(赤)は4 重症患者(黄)は6 軽症患者(緑)は9

結果の検証
算出結果は安定性と耐戦略性を満たし、患者と病院双方が納得する結果を得られる。また、結果は瞬時に表示されるため搬送時間が短縮される。患者数、病院数を増やしても算出時間は変わらないため、多くの傷病者が発生した場合でも対応可能。

完成したシステムは横須賀市消防局で検証をお願いする予定である。

課題と今後の展望

システム設計時の設定を1セットの配分が完了後次の配分を行うと、新規患者や空病床の増減をリアルタイムで反映できない。今後はリアルタイムで更新するアルゴリズムを研究しシステムを改善したい。
本システムは災害時に様々な支援物資を複数の避難所に適正配分する作業に適用できる。今後は災害対策、情報セキュリティ、国際競争等、様々な危機を実践的に学べる分野に進み、今回の研究を基に課題解決に向けた研究を続けたい。
謝辞:参考文献今回のシステム作成にあたり、説明の機会を設けていただき、貴重なご意見をいただいた横須賀市消防局、電気通信大学若崎研究室の皆様へ深く感謝申し上げます。(参考文献) 安定マッチングの数理とアルゴリズム 宮崎修一著 ゲーム理論とマッチング 栗野盛光著 天才たちの未来予測 小島武仁ほか著