תרגיל בית חכם

האינטרנט של הדברים בתעשייה

אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

בחינת שיפורים בצריכת
החשמל השכונתית
בשכונת הבתים החכמים
בבאר שבע

מגישות:
313360489
314963810

תוכן עניינים

המצב הקיים

שכונת הבתים החכמים בבאר שבע משתמשת כיום באלגוריתמי IOT לחילוף מידע בין הבתים על מנת לצרוך חשמל בצורה חכמה. הסוכנים המפעילים את המכשירים בבתים מריצים אלגוריתם שמעניק השמה רנדומלית למכשירים בבית, ובוחנים את צריכת החשמל השכונתית.
במידה וצריכת החשמל השכונתית נמוכה מהקיימת, הסוכן עובר לפתרון החדש.
האלגוריתם הנ"ל מקיים את תכונת החמדנות, ועלול להביא את השכונה לצריכה מינימלית לוקאלית ולא אופטימלית בשל Exploration מועט. בפרוייקט זה נבחן את השיפורים האפשריים בפתרון הבעיה באמצעות אלגוריתם DSA עבור 2 אפשרויות:

1. מימוש האלגוריתם עבור בתים סוכנים – כל בית בוחן 10 פתרונות, בכל פתרון מחליט השמות רנדומליות למכשיריו ומקבל השמות עבור הבתים האחרים. הוא בוחר בפתרון הטוב ביותר מבין ההשמות ומשווה למצב הקיים. אם הפתרון החדש טוב יותר (צריכת החשמל השכונתית נמוכה יותר מהקיימת), הוא עובר לפתרון זה בהסתברות 0.7.
2. מימוש האלגוריתם עבור בתים ומכשירים סוכנים – כל מכשיר בכל בית בוחן 100 פתרונות, בכל פתרון הוא מקבל את השמות המכשירים האחרים בבית ובוחר השמה מועדפת עבורו. במידה והשמה זו טובה יותר (צריכת החשמל השכונתית נמוכה יותר מהקיימת), הוא עובר להשמה זו בהסתברות 0.7. לבסוף הבית מעביר למכשירים ולבתים השכנים את ההשמות שנבחרו.

נבחן את השינויים האפשריים באלגוריתם באמצעות הרצת שכונה בת 7 בתים. נבדוק את האפשרויות הנ"ל ונוסיף להן שיפורים נוספים על מנת להגדיל ככל הניתן את איכות הפתרון והבאת השכונה לצריכה מינימלית בחשמל ככל הניתן.

בחירת מדדים לשיפור

כדי להכריע מי מבין החלופות היא הטובה ביותר, נריץ 50 איטרציות של האלגוריתמים הקיימים והמשופרים בסימולטור, ונבחן שניים מהמדדים העומדים לרשותינו:

1. Total Grade Per Iteration –ציון כללי ממוצע על פני כל הבעיות בניסוי לכל אחד מהאלגוריתמים שבניסוי עבור כל איטרציה. החלופה הטובה ביותר היא זו שתניב ציון נמוך יותר בצריכת החשמל.
2. Average Run Time Per Iteration – מדד זמן הריצה הממוצע על פני כל הבעיות בניסוי, לכל אחד מהאלגוריתמים שבניסוי עבור כל איטרציה. מכיוון שהאלגוריתמים לא שלמים, מרחב החיפוש גדול מאוד ומתקיים Trade Off בין זמן הריצה לאיכות הפתרון, לכן נבדוק באמצעות הגרפים האם האלגוריתמים מתכנסים לפתרון לאחר 50 איטרציות.

מימוש אלגוריתם DSA ושיפורו

מימוש האלגוריתם עבור אפשרות 1:

- נוסיף 10 איטרציות להגרלת 10 השמות רנדומליות למכשירים בבית- אמנם צעד זה יגדיל את הזמן הריצה הכולל של האלגוריתם, אך הוא יכול לסייע לנו למצוא פתרונות טובים יותר של השמות בבית מאשר הרצת איטרציה בודדת.
- כדי לשמור כל פתרון אפשרי יצרנו אובייקט "IterData", המאחסן את צריכת החשמל (randSched) ואת השמות המכשירים (randomSchedForAllProps) עבור כל איטרציה (מפורט בנספח תיעוד הקוד). בסיום 10 האיטרציות נשלוף את הפתרון הטוב ביותר ואותו נשווה למצב הקיים.
- הוספת הסתברות למעבר להשמה האלטרנטיבית הטובה ביותר – נבחר בהסתברות P=0.7 למעבר, על מנת להימנע מנקודת ה-Face Transition שמביאה לפתרון גרוע. בהסתברות המשלימה נשאיר את ההשמה הקודמת של הבית.

שיפור האלגוריתם עבור אפשרות 1:

- השמה רנדומלית חכמה- נשנה את הפונקציה הנתונה `smartChoiceForProp()` : (נספח לתיעוד הקוד). נוסיף מבנה נתונים מסוג רשימה "explored", המאחסן השמות רנדומליות שהאלגוריתם ביקר בהן. כדי להגדיל את מרחב החיפוש האפשרי, נבדוק עבור כל השמה רנדומלית של מכשיר האם היא כבר נבחרה עבורו באיטרציה קודמת. אם כן, נגריל השמה אחרת עד אשר נמצא השמה שלא נבחרה בעבר. אם לא, נבחר בהשמה זו ובהסתברות 0.5 נכניס אותה לרשימה (כלומר בהסתברות 0.5 לא נבחר בהשמה זו שוב).
- הוספת הסתברות לאחסון- תעזור לנו לשלוח את המכשיר למרחבי חיפוש אחרים. יהיו מקרים בהם נרצה לאפשר לו לחזור למקומות שביקר בהם, כדי לבדוק האם שינוי השמות במכשירים האחרים עשוי להביא אותנו לפתרונות טובים יותר.
שיפור זה מבוסס על עיקרון ה-Tabu search, הנותן דגש לתכונת ה-Exploration לעומת ה-Exploitation עליו מתבסס האלגוריתם החמדני במצב כיום.

מימוש האלגוריתם עבור אפשרות 2:

- בפונקציה `improveSchedule()`, שינינו את מספר האיטרציות בלולאה ל-100 איטרציות. בכל איטרציה מכשיר בוחר בהשמה המועדפת עליו.
- כדי שכל מכשיר יוכל לבחור בהשמה המועדפת עליו, התבצעו שינויים בפונקציה הקיימת `chooseTicks()`. (נספח תיעוד הקוד). נעבור על רשימת ההשמות האפשריות עבור הסוכן, ונחשב את צריכת החשמל השכונתית עבור השמה זו. במידה וההשמה משפרת את צריכת החשמל בשכונה, נשמור אותה כהשמה המועדפת עליו עד כה. לבסוף, נבחר בהשמה כהשמה הטובה ביותר, ואותה נשווה למצב הקיים.
- הוספת הסתברות למעבר להשמה המועדפת – נבחר בהסתברות P=0.7 למעבר, על מנת להימנע מנקודת ה-Face Transition שמביאה לפתרון גרוע. בהסתברות המשלימה נשאיר את ההשמה הקודמת של המכשיר.

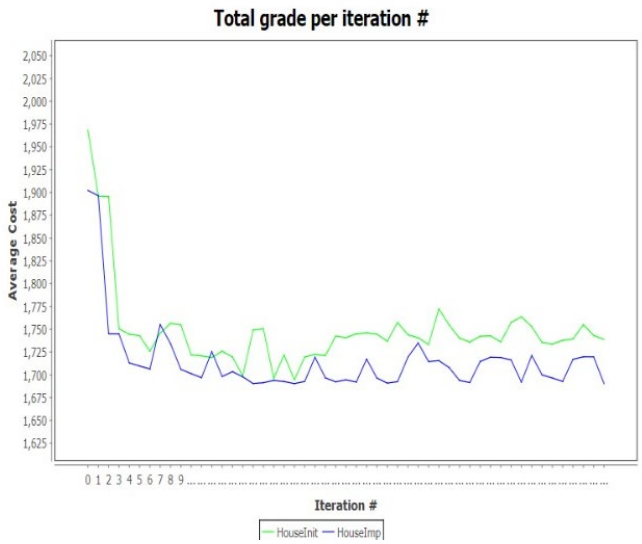
שיפור האלגוריתם עבור אפשרות 2:

- כאשר מכשיר בוחר בהשמה המועדפת עליו, הוא מקיים את תכונת החמדנות. כדי לנסות ולשפר את המדדים שנבחרו בפרוייקט, נוסיף בשיפור זה את התנאי הבא: בהסתברות 0.7 המכשיר אכן בוחר בהשמה המועדפת עליו כפי שצויין מעלה, **ובהסתברות המשלימה המכשיר בוחר השמה רנדומלית חכמה** (כמו בשיפור אלגוריתם 1). לבסוף, נשווה את ההשמה הנבחרת עם המצב הקיים, ונעבור אליה בהסתברות 0.7 אם היא טובה יותר. (נספח תיעוד הקוד)

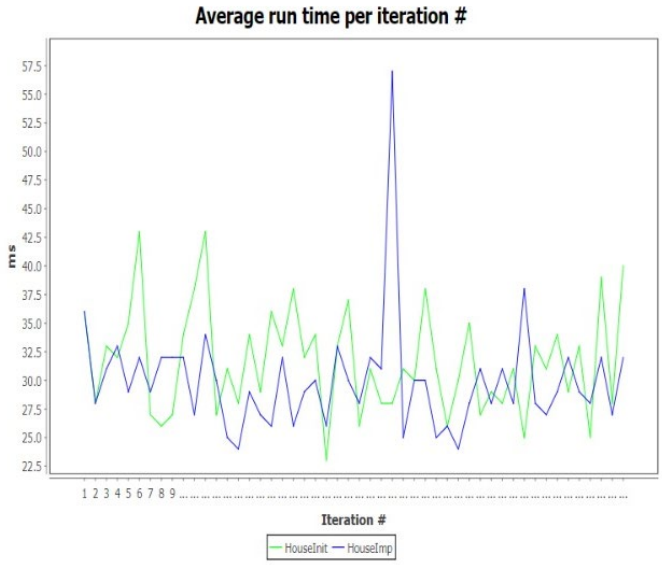
בחינת הפלטים

תחילה נשווה בין האלגוריתם המכיל את השיפורים שהצענו לעומת האלגוריתם שמממש את ההוראות בלבד.

עבור אפשרות House – 1:

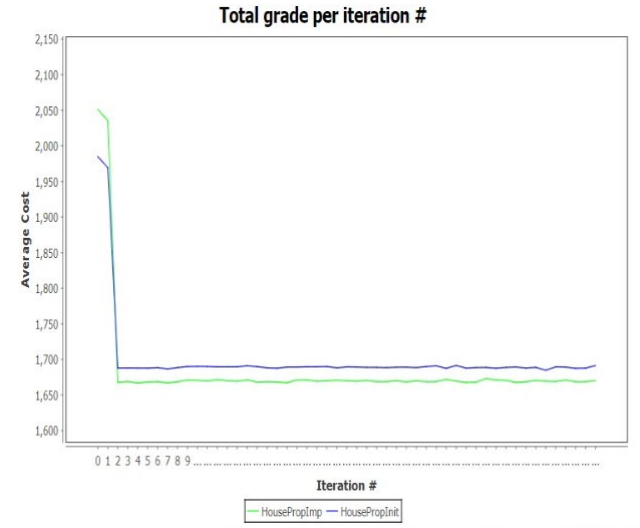


במדד זה, ציון צריכת החשמל הממוצע עבור האלגוריתם ששיפרנו (HouseImp בכחול) הינו **נמוך יותר** ועל כן טוב יותר מהאלגוריתם שמומש בעזרת ההוראות. במספר איטרציות גבוה השיפור הוא משמעותי. על כן הבחירה החכמה עם רשימת ה- explored עדיפה על פני הבחירה הרנדומלית.

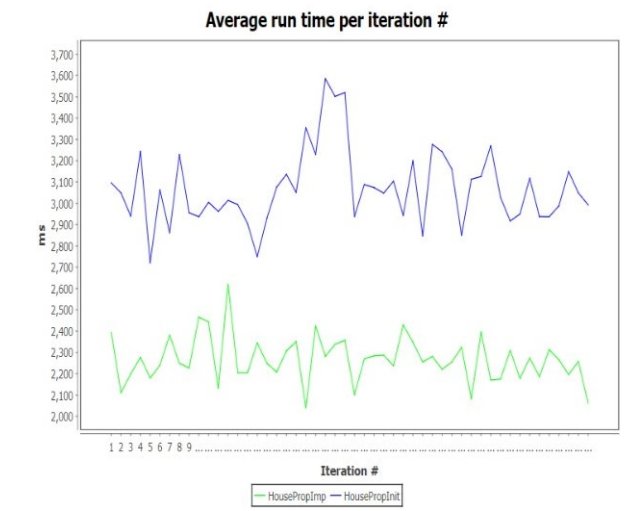


ניתן לראות כי שיפרנו את המדד הראשון, ושיפורים אלו לרוב לא גזלו זמן ריצה ארוך יותר (למעט מספר תצפיות חריגות). לעיתים, העקרונות שיישמנו בשיפורים (exploration) אף **תרמו לזמן ריצה קצר יותר.**

עבור אפשרות 1 – HouseProp:



במדד זה, הציון עבור האלגוריתם ששיפרנו (HousePropImp בירוק) הינו **נמוך יותר** במעט מאשר האלגוריתם שמומש בעזרת ההוראות.

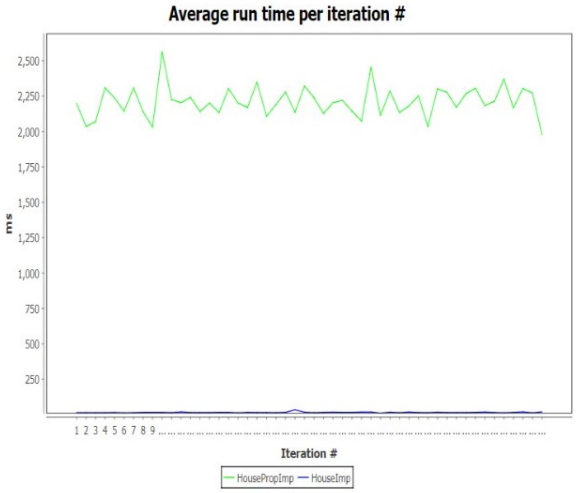


במדד זמן הריצה הממוצע ניכר כי האלגוריתם המשופר (בירוק) מניב **זמן ריצה קצר באופן משמעותי.** על כן למרות שבמדד הראשון השיפורים מינוריים, הם משתלמים לנו בשל זמן ריצה קצר בהרבה.

כעת נרצה להשוות בין שתי האפשרויות הניתנות לנו:

השוואה בין House (1) ו-HouseProp (2):

ראשית, נבחין כי צריכת החשמל השכונתית נמוכה יותר באפשרות (2).
כמו כן, אפשרות (2) בעלת תנודתיות מועטה שמתבטאת ברגישות מועטה לרעשים. ניתן לראות שבהתכנסות ל-50 איטרציות הגרף חלק כמעט לחלוטין.



עם זאת, זמן הריצה של אפשרות (2) גבוה בהרבה מזמן הריצה של אפשרות (1), מכיוון שמספר האיטרציות הפנימיות בכל אפשרות הוא שונה (100 לעומת 10).

מסקנות

בשתי האפשרויות העומדות לרשותינו, השיפורים שביצענו באלגוריתמים מניבים פתרונות טובים יותר. ניתן להסיק כי איזון בין עקרונות ה-exploration (ע"י מתן אקראיות למודל המונעת מאיתנו להיכנס למינימום לוקאלי) וה-exploitation, ממזערים את צריכת החשמל השכונתית וכן מקטינים את זמן הריצה (לעיתים באופן משמעותי).

לדעתנו, תמיד מתקיים Trade Off בין איכות הפתרון לזמן הריצה, ועל כן הבחירה בין שתי האפשרויות תלויה בצורך הלקוח. **מכיוון שמטרת העל בפרוייקט הינה למזער את צריכת החשמל השכונתית, נבחר ב-HouseProp, המבטיח התכנסות טובה יותר ורעש נמוך יותר** (שונות נמוכה).

נספחים

תיעוד קוד

מחלקת iterData

```
class iterData // each object contains array of consumption, and map of (prop, assignment)
{
    private double[] iterationSched;
    private Map<PropertyWithData, Set<Integer>> currentSchedAllProps;
    public iterData(double[] iterationSched, Map<PropertyWithData, Set<Integer>> currentSchedAllProps) {
        this.iterationSched = iterationSched;
        this.currentSchedAllProps = currentSchedAllProps;
    }

    // getters
    public double[] getIterationSched() {
        return this.iterationSched;
    }

    public Map<PropertyWithData, Set<Integer>> getCurrentSchedAllProps(){
        return this.currentSchedAllProps;
    }
}
```

שינוי הפונקציה `smartChoiceForProp()`

```
private Set<Integer> smartChoiceForProp(PropertyWithData prop, List<Set<Integer>> explored) {
    List<Set<Integer>> allSubsets = propToSubsetsMap.get(prop);
    int index = drawRandomNum(0, allSubsets.size() - 1); //pick a random assgiment
    //int counter = 0;
    if (allSubsets == null || allSubsets.isEmpty())
        return new HashSet<>(0);

    do {
        if (!explored.contains(allSubsets.get(index)) || explored == null || explored.isEmpty()) {
            if (flipCoin(0.5f))
                explored.add(allSubsets.get(index));
        }//if
        else index = drawRandomNum(0, allSubsets.size() - 1); //pick a random assgiment
    }
    while(!explored.contains(allSubsets.get(index)));
    return allSubsets.get(index);
}
```

כפי שהזכרנו בגוף הדו"ח, הגדלנו את ה-exploration ע"י בחירה בהשמות אלטרנטיביות שלא נבחרו בעבר. בחירה זו מתבצעת ב-50% מהמקרים על מנת לאפשר למכשיר לבחור השמה זו בעתיד.

שינוי הפונקציה `chooseTicks()`

עבור כל השמה של מכשיר נבדוק מי המועדפת עליו, כלומר מי מניבה את צריכת החשמל השכונתית הנמוכה ביותר:

```
////////////////////// new one ////////////////////
double newGrade = 0;
List<Set<Integer>> allSubsets = propToSubsetsMap.get(prop); // all possible subsets of hours (ticks) for prop
if (allSubsets == null || allSubsets.isEmpty()) { // when no ticks exists
    this.nextTicks = new HashSet<>(0); // as pickRandom function
    this.currSched = helper.cloneArray(iterationPowerConsumption); // in order to not lose info
    this.updateNeighbors(); // update consumption of the agent's neighbors because of my change
    this.allScheds.add(this.currSched);
    newGrade = calcImproveOptionGrade(this.currSched, this.allScheds); // calculate improve neighborhood grade
    this.allScheds.remove(this.currSched);
}
else { // ticks exists
    Set <Integer> tempTicks = new HashSet<Integer>();
    double tempGrade = 0;
    double bestGrade = 0; // this will be the favorite assignment
    // for all assignments of an agent, the favorite one will be chosen
    for(int i= 0; i < allSubsets.size(); i++) {
        tempTicks = allSubsets.get(i);
        this.currSched = helper.cloneArray(iterationPowerConsumption); // in order to not lose info
        this.updateNeighbors(); // update consumption of the agent's neighbors because of my change
        this.insertMyTicks(tempTicks); // insert my new temp ticks

        // calc the grade
        this.allScheds.add(this.currSched);
        tempGrade = calcImproveOptionGrade(this.currSched, this.allScheds); // calculate improve neighborhood grade
        this.allScheds.remove(this.currSched);
        if (tempGrade < bestGrade || i == 0) { // temp is better than best grade we saved
            bestGrade = tempGrade;
            this.nextTicks = tempTicks;
        }
    }
    newGrade = bestGrade; // checking best assignment with previous assignment
}
```

לאחר שנמצא את ההשמה המועדפת על מכשיר, נבחר בה ונשווה אותה למצב הקיים כאן:

```
// decision: we compare both assignments using their grades
// you can use flipCoin(0.8f) as a boolean probability function
if (newGrade < oldGrade && flipCoin(0.7f)){ // minimum
    this.toChange = true;
}
```

הוספת הסתברות להשמה מועדפת על מכשיר

```
if (flipCoin(0.7f)) {
    if (allSubsets == null || allSubsets.isEmpty()) { // when no ticks exists
        this.nextTicks = new HashSet<>(0); // as pickRandom function
        this.currSched = helper.cloneArray(iterationPowerConsumption); // in order to not lose info
        this.updateNeighbors(); // update consumption of the agent's neighbors because of my change
        this.allScheds.add(this.currSched);
        newGrade = calcImproveOptionGrade(this.currSched, this.allScheds); // calculate improved grade
        this.allScheds.remove(this.currSched);
    }
    else { // ticks exists
        Set <Integer> tempTicks = new HashSet<Integer>();
        double tempGrade = 0;
        double bestGrade = 0; // this will be the favorite assignment
        // for all assignments of an agent, the favorite one will be chosen
        for(int i= 0; i < allSubsets.size(); i++) {
            tempTicks = allSubsets.get(i);
            this.currSched = helper.cloneArray(iterationPowerConsumption); // in order to not lose info
            this.updateNeighbors(); // update consumption of the agent's neighbors because of my change
            this.insertMyTicks(tempTicks); // insert my new temp ticks

            // calc the grade
            this.allScheds.add(this.currSched);
            tempGrade = calcImproveOptionGrade(this.currSched, this.allScheds); // calculate improved grade
            this.allScheds.remove(this.currSched);
            if (tempGrade < bestGrade || i == 0) { // temp is better than best grade we saved
                bestGrade = tempGrade;
                this.nextTicks = tempTicks;
            }
        }
        newGrade = bestGrade; // checking best assignment with previous assignment
    }

else {
    this.nextTicks = smartChoiceForProp(this.prop, explored); // in this case we choose smart set of ticks
    this.currSched = helper.cloneArray(iterationPowerConsumption); // in order to not lose info
    this.updateNeighbors(); // update consumption of the agent's neighbors because of my change
    this.insertMyTicks(this.nextTicks); // insert my new ticks
    // calc the grade
    this.allScheds.add(this.currSched);
    newGrade = calcImproveOptionGrade(this.currSched, this.allScheds); // calculate improve neighborhood grade
    this.allScheds.remove(this.currSched);
}

// decision: we compare both assignments using their grades
// you can use flipCoin(0.8f) as a boolean probability function
if (newGrade < oldGrade && flipCoin(0.7f)){ // minimum
    this.toChange = true;
}
```