Resource-gathering algorithms in the game of Starcraft

Resource-Gathering Algorithms in the Game of StarCraft

Martin L.M. Rooijackers and Mark H.M. Winands
Games & AI Group, Department of Data Science and Knowledge Engineering
Maastricht University, Maastricht, The Netherlands
Email: mlm.rooijackers@student.maastrichtuniversity.nl, m.winands@maastrichtuniversity.nl

***Abstract*—*StarCraft* is a real-time strategy game, which has a large state space, and commonly features two opposing players, capable of acting simultaneously. One of the aspects of the game is resource gathering. Each agent playing *StarCraft* has to gather minerals from nearby mineral field in order to produce more units. The more resources can be gathered, the larger the army is to attack the opponent and win the game. We present five algorithms that can be used for resource gathering for an intelligent agent playing the game of *StarCraft: Brood War*. The results reveal that improving the scheduling or the pathfinding improve the resource gathering considerably.**

I. Introduction

StarCraft is a popular real-time strategy (RTS) game for testing AI. One of the aspects of StarCraft is resource gathering. Christensen et al. [1] showed that there is a correlation between spending resources and winning the game. Therefore, to increase the chance of a *StarCraft* agent winning the game, it is useful to increase its resource-gathering rate. Since the field of RTS AI is still relatively new [2], there has not been much study on the factors involved in the resource-gathering rate. Previous work by Wintermute et al. [3] has studied the effects of collision avoidance on the resource gathering rate in RTS games. Christensen et al. [1] showed that using a scheduling algorithm can also improve the resource-gathering rate.

In StarCraft, worker units gather the mineral resources [4]. The built-in resource-gathering algorithm used for mineral gathering in the game StarCraft is not optimal [1]. In this paper we look at several different resource-gathering algorithms and evaluate them based on how many minerals are gathered in an experimental setup.

This paper is structured as follows. First, Section II gives the problem definition of the resource-gathering task. Next, Section III describes five resource-gathering algorithms. Subsequently, Section IV presents the experimental results. Finally, Section V draws conclusions from the results and outlines future research.

II. Problem Definition

This section follows closely the problem definition that Christensen et al. [1] used for their resource-gathering algorithm.

As input, there is a set of worker units $A = \{a_i | i = 1, \ldots, n\}$ and a set of mineral fields $M = \{m_j | j = 1, \ldots, l\}$ in a two-dimensional Euclidean space [1]. Each mineral field m_j has a certain amount of minerals $r_j \in Z^+$. The goal of each resource-gathering algorithm presented in this paper is to gather as many minerals within a given time. We define T to be the total time that is required to perform every gathering. A resource-gathering algorithm selects a subset $S \subseteq G$ of gathering tasks $G = A \times M$ for each worker unit such that the total amount of minerals $R = \sum_{j=1}^{l} r_j$ is gathered in a minimal time T.

The worker units have a fixed maximum capacity for carrying mineral r_k. They collect $r_a = min(r_k, r_j)$ from a mineral field before returning to a resource depot D to unload their minerals [1]. The time required for the actual gathering is a constant C. For each mineral field m_j there is exactly one mineral-field queue Q_j. It consists of a set of worker units that collect minerals from this mineral field. A mineral-field queue is defined as a totally ordered set of worker units $Q_j = \{a_h \in A | h = 1, \ldots, z_j\}$, $\forall_{a_h} \in Q_j \implies a_h \notin Q_k$ where $j \neq k$ such that each worker unit is assigned to at most one mineral-field queue at a time. When a worker unit has finished collecting minerals, it is removed from the queue. The first element $a_1 \in Q_j$ is the worker unit that can use the mineral field first. A gathering task $s_{i,j}$ is completed in a round-trip time $t_{i,j}$ after worker unit a_i travels from D to a mineral field m_j, possibly waits there, collects resources and goes back to D. Formula 1 gives the calculation of the round-trip time [1],

$$t_{i,j} = tt^i_{D,j} + max[0, rt_j - tt^i_{D,j}] + C + tt^i_{j,D} \qquad (1)$$

where $tt^i_{D,j}$ is the time required for the worker unit a_i to travel from depot D to the mineral field m_j. Variable rt_j is the remaining time for mineral field m_j to become available after all preceding worker units have finished their work in queue Q_j. Therefore, the waiting time of a worker unit a_i is equal to the remaining time for the queue to become empty, rt_j, minus the time that has already been spent traveling. The constant collecting time C is added along with the travel time $tt^i_{j,D}$ for returning to the deposit.

By using mineral-field queues, it is possible to determine the best mineral field to send worker units in order to minimize the time required for the worker unit to return to D with an amount of minerals [1]. Formula 2 defines the remaining time for a worker unit h in queue Q_j to finish gathering a mineral field m_j. Variable c refers to the remaining collection time of the first worker unit in the queue, $a_1 \in Q_j$ where $0 \leq c \leq C$.

$$rt^h_j = \begin{cases} tt^h_{D,j} + c_h & \text{for } h = 1 \\ rt^{h-1}_j + max[0, tt^h_{D,j} - rt^{h-1}_j] + C & \text{for } h \neq 0 \end{cases} \qquad (2)$$

The required time for worker units ahead of a given worker unit for finishing the collection of minerals from a field is included in the recursive definition of rt_j^h. The total time of Q_j is therefore $rt_j = rt_j^{z_j}$, meaning the time required for the last worker unit in Q_j to finish gathering the resource. The time required for a worker unit to finish its gathering is dependent on the time required for the preceding worker unit, as gathering cannot occur before the preceding worker unit has finished its gathering. The time required for the first worker unit in the queue, rt_j^1, does not depend on any preceding worker unit as there are none.

In StarCraft a worker unit starts gathering when it sends an action to the game engine that gives a gather command for a particular mineral field. In order for a worker unit to stay at that field it has to continuously send that action.

III. Resource Gathering

Whenever a worker unit is created or has returned mineral to a resource depot, it uses a resource-gathering algorithm to determine the next resource location to gather resources from. By default, the StarCraft engine tells the worker unit to go to the previous mineral field location. The idea of a resource-gathering algorithm is that it calculates the next resource location that the worker should go to in order to maximize the total amount of resources gathered.

We discuss five different resource-gathering algorithms[1] in this section: Built-in (III-A), Mineral-lock (III-B), Queue Based Scheduling (III-C), Co-operative pathfinding (III-D), and Co-operative pathfinding + Queue (III-E).

A. Built-in

The first algorithm called Built-in uses the default way that StarCraft handles worker units. The algorithm is only used for worker units that are not assigned to a mineral field. If upon arrival the mineral field is blocked, the worker unit is unassigned from that mineral field and the Built-in is called. If all mineral fields are blocked, then the worker unit waits at the closest mineral field.

The pseudocode can be found in Algorithm 1. This algorithm makes use of a function dist($worker, mineral$) that gives the Euclidean distance between a worker and a mineral.

Strong points:

- Low CPU requirements.
- Low number of actions needed.

For each worker unit a, one only has to iterate over all mineral fields M twice in the worst case, when assigning a worker unit. Since the total number of worker units is usually between double or triple the number of mineral fields ($2M \leq A \leq 3M$), the total running time is low. Compared to other algorithms whose running time is linearly affected by the total number of other worker units, Built-in has the lowest CPU requirement of all algorithms tested.

[1]The implementation of each algorithm can be found at: github.com/MartinRooijackers/LetaBot/tree/master/Research/MineralGatheringAlgorithm

Algorithm 1 Built-in

```
procedure BUILT-IN(Q, M)
    Input: Q_1, ..., Q_|M|: resource site queues
    Input: M: resource sites
    for all a_i ∉ ⋃_{j=0}^{|M|} Q_j do
        best ← 0
        distance ← ∞
        for all x ∈ 1...|M| do
            if |Q_x| = 0 then
                if dist(a_i, M_x) < distance then
                    distance ← dist(a_i, M_x)
                    best ← x
                end if
            end if
        end for
        if best ≠ 0 then
            Q_best ← Q_best ∪ {a_i}
        else
            for all x ∈ 1...|M| do
                if |Q_x| ≠ 0 then
                    if dist(a_i, M_x) < distance then
                        distance ← dist(a_i, M_x)
                        best ← x
                    end if
                end if
            end for
            Q_best ← Q_best ∪ {a_i}
        end if
    end for
end procedure
```

Because Built-in only has to send one action to start the gathering process, the total number of actions performed by this algorithm is also low. This is useful since the game StarCraft does not allow unlimited actions when playing multi-player (it stops processing commands when the network buffer is full).

Weak points:

- Sub-optimal resource assignment. Resulting in a low amount of resources gathered.

Built-in does not gather many minerals due to the fact that the worker units do not (directly) take the actions of other worker units into account. This is because of that a mineral field only allows one worker unit to gather minerals from it at a time. An example of a sub-optimal schedule would be the case where two worker units go to the same mineral field even though another mineral field will become available soon to allow both worker units to gather at different mineral fields.

B. Mineral-lock

The second algorithm (Mineral-lock) is implemented in several StarCraft agents (e.g., UAlbertaBot, Orikata). This algorithm evenly distributes the worker units over all the mineral fields part of the base.

The pseudocode can be found in Algorithm 2. This algorithm also stores for each worker which mineral field it is assigned to in $a_{i,mineral}$. By default, $a_{i,mineral}$ is equal to 0 to indicate that no mineral field has been selected yet.

Algorithm 2 Mineral-lock

```
1: procedure MINERAL-LOCK(Q, M, D)
2:     Input: Q_1, ..., Q_l: resource site queues
3:     Input: M: resource sites
4:     Input: D: resource depot
5:     for all a_i ∉ ⋃_{j=0}^{|M|} Q_j do
6:         minWorker ← ∞
7:         for all x ∈ 1...|M| do
8:             if |Q_x| < minWorker then
9:                 minWorker ← |Q_x|
10:            end if
11:        end for
12:        minTravelTime ← ∞
13:        choice ← 0
14:        for all x ∈ 1...|M| do
15:            if (tt^i_{x,D}+tt^i_{D,x}) < minTravelTime∧|Q_x| =
   minWorker then
16:                minTravelTime ← (tt^i_{x,D} + tt^i_{D,x})
17:                choice ← x
18:            end if
19:        end for
20:        a_{i,mineral} ← choice
21:        Q_choice ← Q_choice ∪ {a_i}
22:    end for
23:    for all a_i ∈ A do
24:        if a_{i,mineral} ≠ 0 then
25:            gather(a_i, a_{i,mineral})
26:        end if
27:    end for
28: end procedure
```

Strong points:

- Low CPU requirements.

For each worker unit, one only has to iterate over all mineral fields twice. Once to check how many workers are already assigned to it, and the second time to calculate which of the potential mineral fields is closest to the resource depot. This brings the running time for each worker unit to $2|M| + |A|$. Since the total number of worker units is usually between double or triple the number of mineral fields ($2|M| \leq |A| \leq 3|M|$), there is in practice not much more CPU required than the Built-in algorithm.

Weak points:

- High number of actions required.
- Not an optimal resource setup (but still better than Build in). Resulting in a low number of resources gathered.

Since Mineral-lock has to send an action every time that the worker unit moves away from its assigned mineral field (i.e., when it has to wait for other worker units), the total number of actions performed by this algorithm tends to be high. This is a problem since the game StarCraft does not allow unlimited actions when playing multi-player, because it stops processing commands when the network buffer is full.

The worker assignment is not optimal since there is a possibility that a worker unit would have been better off going to a different mineral field for one round since that mineral field would be available earlier. This is caused by the fact that this algorithm does not take the schedule of other worker units into account.

C. Queue Based Scheduling

The third algorithm called Queue Based Scheduling is an approach proposed by Christensen et al. [1]. It attaches queues to each mineral field. After a worker unit has delivered its resources to the depot, the algorithm determines the queue that allows the worker unit to start harvesting as fast as possible [1]. It is a greedy approach, as it does not take into account other worker units arriving after the current worker unit.

The algorithm relies on two functions. The main function is in Algorithm 4. This function checks for each worker unit if it is assigned to a queue. If not, then it will pick the queue where the worker unit gets to mine the fastest. Calculating how long it will take for the worker unit to be able to start gathering minerals is done in Algorithm 3. This algorithm, called *Work*, checks at what position a worker unit is in the queue. It uses that to determine how much work needs to be done before the worker unit can gather from the mineral field associated with queue Q.

Algorithm 3 Work

```
1: procedure WORK(Q, a)
2:     w ← 0                                  ▷ workload of the queue
3:     p ← position of a in Q
4:     if p = 0 then
5:         return tt^i_{D,j} + C
6:     end if
7:     w ← w+Work(Q, a_{p-1})
8:     if w > tt^i_{D,j} then
9:         return w + C
10:    else
11:        return tt^i_{D,j} + C
12:    end if
13: end procedure
```

Algorithm 4 Queue

```
1: procedure QUEUE(Q, M, D)
2:     Input: Q_1, ..., Q_{|M|}: resource site queues
3:     Input: M: resource sites
4:     Input: D: resource depot
5:     for all a_i ∉ ⋃_{j=0}^{|M|} Q_j do
6:         time ← ∞
7:         for all x ∈ 1...|M| do
8:             A ← Q_x ∪ {a_i}
9:             if Work(A, a_i) + tt^i_{x,D} < time then
10:                time ← Work(A, a_i) + tt^i_{x,D}
11:                best ← x
12:            end if
13:        end for
14:        Q_best ← Q_best ∪ {a_i}
15:    end for
16: end procedure
```

Strong points:

- Good approximation of the optimal resource setup.

Because Algorithm 4 takes into account the schedule of other worker units, it produces schedules that are closer to being optimal than the Mineral-lock and the Built-in algorithm. The scheduling is performed greedy, so it still remains an approximation, and thus is not the optimal schedule.

Weak points:

- Medium CPU requirements.
- Not always an optimal resource setup. Total number of resources gathered could be improved.
- Medium number of actions needed.

When calculating the next mineral field assignment, this algorithm has to calculate the total amount of work needed for each mineral field, which is related to the total number of worker units already assigned to the mineral field. Unlike the Mineral-lock algorithm, which only takes into account the distance from the resource depot to the mineral field, the queue algorithm has to calculate the distance for each worker unit.

This queue based approach is also greedy due to the fact that it only looks at the present situation when assigning a worker unit. It does not take into account any other worker units that are about to return their resource. Therefore it can be the case that the next worker unit will receive a sub-optimal mineral assignment due to the greedy scheduling of the worker unit that was scheduled before it. The pathfinding to the mineral field is also not optimal in this algorithm, because this queue algorithm uses the default pathfinding to get the worker units to the minerals.

Since this algorithm needs to send an action every time that the worker unit moves away from its assigned mineral field (i.e., when it has to wait for other worker units), the total number of actions performed by this algorithm tends to be high. However, since it has a more optimal schedule than the Mineral-lock, this occurs less often.

D. Co-operative Pathfinding

In StarCraft Brood War, the path that the worker units take to the mineral field is not optimal. In this section we propose two techniques to improve the pathfinding of worker units. The goal of both techniques is to reduce the time that it takes to travel to a mineral field. Reducing the time to travel from the mineral field to the resource depot is left as future research.

Both techniques are based on the fact that worker units slow down when they approach the mineral field that they are assigned to. Instead of going directly to the mineral fields, both techniques send the worker unit to a position slightly beyond the mineral field and change direction when the worker unit is close enough to its assigned mineral field. Both techniques rely on knowing the mineral orientation, which is explained in Subsection III-D1.

The first technique is called *MineralPath* and uses a mineral field further away as the intermediate position. This has the advantage of the worker unit not needing extra collision avoidance, because worker units that go to any mineral field will have their collision detection temporarily disabled. This technique is described in Subsection III-D2.

The second technique is *CalculatePath* and uses empty space beyond the mineral field as the intermediate position. Since the worker is now no longer moving towards a mineral field, we also use a method for collision avoidance. This technique is described in Subsection III-D3.

In the following subsections we describe how to determine what orientation each mineral field has in relation to the resource depot. We also explain each technique individually.

1) Mineral Orientation: Determining how the minerals are positioned in relation to the resource depot is important for both techniques. Algorithm 5 determines how the worker units approach a mineral field. The algorithm uses the maximum distance of a mineral field belonging to a command center (i.e., 10 tiles of 32 pixels). There are four orientations for a mineral field: left(W), right(E), up(N), down(S) (see also Figure 1).

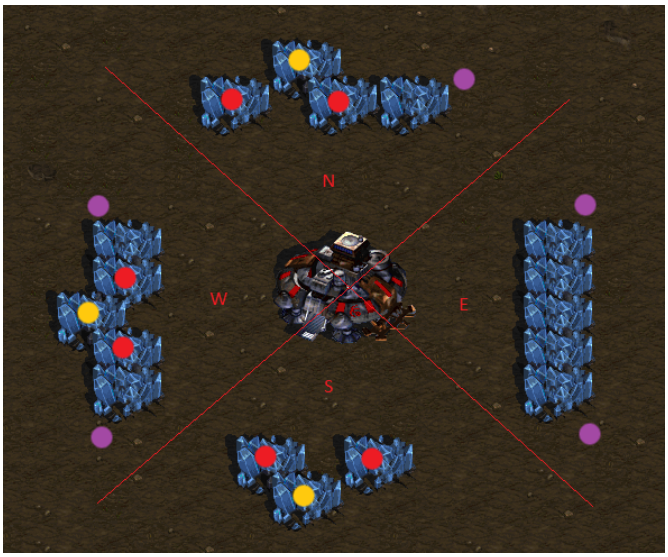

Fig. 1. Example of a Mineral Orientation calculation. The minerals are divided into four sections. The red circles indicate the mineral fields that are used to perform the MineralPath technique with the help of the nearby mineral field with a yellow circle. The purple circles indicate the location of the empty-path trick.

2) MineralPath: The mineral-path technique relies on the fact that a worker unit usually slows down when approaching the mineral field it wants to harvest. Instead of going to that mineral field, the worker unit first moves to a mineral field further away, then changes direction once it is close to the mineral field it wants to harvest. This is shown in Figure 1, with the red circles indicating the mineral fields where the mineral-path technique can be performed using the yellow mineral fields as way points.

Algorithm 6 is called on each mineral field to determine if it has a mineral-path technique. If so, the algorithm returns the mineral field that can be used for the mineral-path technique.

3) CalculatePath: Besides the previous path technique that uses another mineral field, the CalculatePath technique uses the empty space beyond the mineral field. This is called the

Algorithm 5 MineralOrientation

```
1: procedure ISLEFT(a, b, c)        ▷ Check if a point in a 2d plane is left of the line ab
2:     return ((b_x − a_x)·(c_y − a_y) − (b_y − a_y)·(c_x − a_x)) ≥ 0
3: end procedure
4: procedure MINERALORIENTATION(D, mineral)
5:     Input: D: resource depot
6:     Input: mineral: mineral field to calculate a path for
7:     TopLeft ← (D_x − 320, D_y − 320)
8:     BottomRight ← (D_x + 320, D_y + 320)
9:     BottomLeft ← (D_x − 320, D_y + 320)
10:    TopRight ← (D_x + 320, D_y − 320)
11:    isUpLeft ← false
12:    isBotLeft ← false            ▷ Short for bottom left
13:    if isLeft(TopLeft, BottomRight, mineral) then
14:        isBotLeft ← true
15:    end if
16:    if isLeft( BottomLeft, TopRight, mineral) then
17:        isUpLeft ← true
18:    end if
19:    if isBotLeft = true ∧ isUpLeft = false then
20:        return down              ▷ below the resource depot
21:    end if
22:    if isBotLeft = true ∧ isUpLeft = true then
23:        return left              ▷ to the left of the resource depot
24:    end if
25:    if isBotLeft = false ∧ isUpLeft = true then
26:        return up                ▷ to the above of resource depot
27:    end if
28:    if isBotLeft = false ∧ isUpLeft = false then
29:        return right             ▷ right the resource depot
30:    end if
31: end procedure
```

Algorithm 6 MineralPath

```
1: procedure MINERALPATH(D, M, mineral)
2:     Input: D: resource depot
3:     Input: M: resource sites
4:     Input: mineral: mineral field to calculate a path for
5:     Orientation ← MineralOrientation(D, mineral)
6:     for all m ∈ M \ {mineral} do
7:         if dist(m, mineral) < 64 then
8:             if Orientation = left then
9:                 if mineral_x > m_x then
10:                    return m
11:                end if
12:            end if
13:            if Orientation = right then
14:                if mineral_x < m_x then
15:                    return m
16:                end if
17:            end if
18:            if Orientation = up then
19:                if mineral_y > m_y then
20:                    return m
21:                end if
22:            end if
23:            if Orientation = down then
24:                if mineral_y < m_y then
25:                    return m
26:                end if
27:            end if
28:        end if
29:    end for
30:    return NULL                  ▷ no mineral field found for trick
31: end procedure
```

"empty space trick". Algorithm 7 describes this technique. An example is given in Figure 1.

For this the worker units do have to take into account other worker units, since a move command causes the worker unit to lose its ability to float over other units. Therefore, it avoids collisions. This is done by mapping all the tiles that are 3 tiles away from a worker unit as inaccessible. If a worker unit wants to perform this trick, it will only try to do so when it is not on an inaccessible tile (see also Algorithm 8, lines 30-36).

4) CoopPath: With the mineral path and empty space known for each mineral, the algorithm CoopPath can improve the path for each worker unit assigned to a mineral field where that is possible (see also Figure 2). In terms of scheduling, this pathfinding algorithm uses the same approach as Mineral-lock. In case the mineral field can both use the mineral-path technique and the empty space trick, the worker unit will prefer the former. This algorithm uses the function TileDist(a, b) (from the BWAPI), which gives the distance in tiles between a and b.

Strong points:

- Reduced travel time.

The main advantage to using Co-operative Pathfinding is that it reduces the travel time to a mineral field. Because of the lower time spend traveling, the worker units can deliver more minerals per minute compared to Built-in.

Weak points:

- High CPU requirements.
- Not always an optimal resource setup. Total number of resources gathered could be improved.
- Large number of actions needed.

Because of the need to calculate the path, and the position of other worker units to avoid collisions, this algorithm requires more CPU than Built-in and Mineral-lock. This is mainly due to the collision avoidance for CalculatePath. However, the results of the path calculations can be stored, and since most StarCraft maps have only 2 minerals where the empty space trick is possible, the collision avoidance will only need to be called once each frame.

Although the travel time is reduced, the scheduling is still not optimal. Therefore the total amount of minerals gathered from this technique is not optimal.

Since Co-operative Pathfinding has to send an action every time that the worker unit moves away from its assigned mineral field or its generated path, the total number of performed actions tends to be high. However, if the technique is not possible for the given mineral field, the algorithm will behave like Mineral-lock in terms of actions needed.

Algorithm 7 CalculatePath

procedure CALCULATEPATH($D, M, mineral$)
  Input: D: resource depot
  Input: M: resource sites
  Input: $mineral$: mineral field to calculate a path for
  $Orientation \leftarrow$ MineralOrientation($D, mineral$)
  $OuterLeft \leftarrow false$ ▷ check if the other mineral field blocks the outer left or right
  $TotalAdjacent \leftarrow 0$ ▷ the technique only works with at most one mineral field adjacent
  for all $m \in M \setminus \{mineral\}$ **do**
    if dist$(m, mineral) < 64$ **then**
      $TotalAdjacent \leftarrow TotalAdjacent + 1$
      if $Orientation = left$ **then**
        if $mineral_y > m_y$ **then**
          $OuterLeft \leftarrow true$
        end if
      end if
      if $Orientation = right$ **then**
        if $mineral_y < m_y$ **then**
          $OuterLeft \leftarrow true$
        end if
      end if
      if $Orientation = up$ **then**
        if $mineral_x < m_x$ **then**
          $OuterLeft \leftarrow true$
        end if
      end if
      if $Orientation = down$ **then**
        if $mineral_x < m_x$ **then**
          $OuterLeft \leftarrow true$
        end if
      end if
    end if
  end for
  if $TotalAdjacent > 1$ **then**
    return $NULL$ ▷ This technique is not possible for the given mineral field
  end if
  if $Orientation = left$ **then**
    if $OuterLeft = false$ **then**
      return $(mineral_x - 32, mineral_y - 32)$
    else
      return $(mineral_x - 32, mineral_y + 32)$
    end if
  end if
  if $Orientation = right$ **then**
    if $OuterLeft = false$ **then**
      return $(mineral_x + 32, mineral_y + 32)$
    else
      return $(mineral_x + 32, mineral_y - 32)$
    end if
  end if
  if $Orientation = up$ **then**
    if $OuterLeft = false$ **then**
      return $(mineral_x + 32, mineral_y - 32)$
    else
      return $(mineral_x - 32, mineral_y - 32)$
    end if
  end if
  if $Orientation = down$ **then**
    if $OuterLeft = false$ **then**
      return $(mineral_x + 32, mineral_y + 32)$
    else
      return $(mineral_x - 32, mineral_y + 32)$
    end if
  end if
end procedure

Algorithm 8 CoopPath

procedure COOPPATH(Q, M, D)
  Input: $Q_1, \ldots, Q_{|M|}$: resource site queues
  Input: M: resource sites
  Input: D: resource depot
  for all $a_i \notin \bigcup_{j=0}^{|M|} Q_j$ **do**
    $minWorker \leftarrow \infty$
    for all $x \in 1 \ldots |M|$ **do**
      if $|Q_x| < minWorker$ **then**
        $minWorker \leftarrow |Q_x|$
      end if
    end for
    $minTravelTime \leftarrow \infty$
    $choice \leftarrow 0$
    for all $x \in 1 \ldots |M|$ **do**
      if $(tt^i_{x,D} + tt^i_{D,x}) < minTravelTime \wedge |Q_x| = minWorker$ **then**
        $minTravelTime \leftarrow (tt^i_{x,D} + tt^i_{D,x})$
        $choice \leftarrow x$
      end if
    end for
    $a_{i_{mineral}} \leftarrow choice$
    $Q_{choice} \leftarrow Q_{choice} \cup \{a_i\}$
    if MineralPath$(D, M, choice) \neq NULL$ **then**
      $Q_{choice,mp} \leftarrow$ MineralPath$(D, M, choice)$
    else
      if CalculatePath$(D, M, choice) \neq NULL$ **then**
        $Q_{choice,sp} \leftarrow$ CalculatePath$(D, M, choice)$
      end if
    end if
  end for
  for all $a_i \in \bigcup_{j=0}^{|M|} Q_j$ **do**
    for all $a \in A \setminus \{a_i\}$ **do**
      if TileDist$(a, a_i) < 3$ **then**
        $Q_{j,sp} \leftarrow NULL$
      end if
    end for
  end for
end procedure

E. Co-operative Pathfinding + Queue

The final algorithm is the combination of the Co-operative Pathfinding and Queue Based Scheduling. Queue Based Scheduling is used for the scheduling, and the Co-operative Pathfinding is used for moving the worker units towards the mineral fields. The pseudocode is given in Algorithm 9.

Strong points:

- Reduced travel time.
- Good approximation of the optimal resource setup.
- High resource gathering rate.

The combination of the Co-operative Pathfinding and the Queue Based Scheduling ensures that both the travel time and the scheduling can be improved. This is because of the scheduling and the pathfinding are executed independent of each other. Therefore the combination of both algorithms does not have a negative effect on the overall minerals gathered.

Fig. 2. Example of co-operative pathfinding with collision avoidance indicated by the red dots. The numbers on the mineral fields indicate travel time. The purple circle indicates the location of the empty space trick.

Although both the scheduling and the pathfinding are not optimal, combined they increase the gather rate more than all the other algorithms discussed. It is possible to improve a mineral gathering algorithm in terms of both pathfinding and scheduling. Regarding pathfinding, it is sometimes the case that the path from the mineral field to the resource depot can be improved. Regarding scheduling, the queue based algorithm uses a greedy scheduling that does not plan ahead.

Weak points:

- High CPU requirements.
- Large number of actions needed.

This algorithm requires both the CPU time for the Co-operative pathfinding and the queue based algorithm. Therefore, it has the highest CPU requirement of all the algorithms discussed.

Since this combined algorithm has to send an action every time that the worker unit moves away from its assigned mineral field or its generated path, the total number of actions performed by this algorithm tends to be high. Although the greedy scheduling might schedule less (or more) worker units to mineral fields that use one of the path techniques, in practice the actions needed are roughly the same as the Co-operative Pathfinding since two worker units can usually saturate a mineral field that uses a path technique. Thus both the standard co-operative pathfinding and the version with a queue based scheduler have not much difference in actions required.

IV. Experiments

In the experiments of this paper, we use the top right part of the map "Astral Balance" as the initial setup. This setup includes the resource depot, four worker units, eight mineral fields and a gas geyser. At the start of the experiment, each worker unit is assigned a mineral field based on the current algorithm that is evaluated in this experiment. After a worker unit has returned its mineral back to the resource depot, it is reassigned to a mineral field using the same algorithm. During this process the resource depot should constantly build new worker units until it has 18 worker units. Since the initial setup only allows a supply of 10 worker units, a worker unit has to be pulled away from gathering minerals to build a supply depot at a certain moment. In the experiments, we pull one worker away from minerals when we have exactly nine worker units. Once this worker unit has finished building a supply depot, it is reassigned to gathering minerals using the algorithm that is evaluated. The experiment ends when 8000 frames have passed. In each frame, we record how many minerals have been gathered by the worker units. In the case of the queue based algorithms, we first pre-calculate the travel time to each mineral field before we start the experiment. This experimental setup is almost like the one used in Christensen et al. [1]. The difference is that we run the experiment where the algorithm builds the supply depot once there are nine worker units, since this is a more common occurrence in most build orders.

The results of the total minerals gathered can be found in Table I, their standard deviations are given in Table II. The time an algorithm took each 1000 frame steps is given in Table III. The tables reveal that the more computationally expensive algorithms appear to gather more minerals. The high computation time from the Co-operative Pathfinding is due to checking collisions for all worker units. Tests with

Algorithm 9 QueueAndCoopPath

procedure QueueAndCoopPath(Q, M, D)
  Input: $Q_1, \ldots, Q_{|M|}$: resource site queues
  Input: M: resource sites
  Input: D: resource depot
  for all $x \in 1 \ldots |M|$ **do**
    $m \leftarrow Q_{x,mineral}$
    if MineralPath(D, M, m) $\neq NULL$ **then**
      $Q_{x,mp} \leftarrow$MineralPath(D, M, m)
    else
      if CalculatePath(D, M, m) $\neq NULL$ **then**
        $Q_{x,sp} \leftarrow$ CalculatePath(D, M, m)
      end if
    end if
  end for
  $time \leftarrow \infty$
  for all $a_i \notin \bigcup_{j=0}^{|M|} Q_j$ **do** ▷ Every worker not assigned to a mineral
    for all $x \in 1 \ldots |M|$ **do**
      $A \leftarrow Q_x \cup \{a_i\}$
      if Work(A, a_i) + $tt^i_{x,D} < time$ **then**
        $time \leftarrow$ Work(A, a_i) + $tt^i_{x,D}$ ▷ Function Work from Algorithm 3
        $best \leftarrow x$
      end if
    end for
    $Q_{best} \leftarrow Q_{best} \cup \{a_i\}$
  end for
  for all $a_i \in \bigcup_{j=0}^{|M|} Q_j$ **do**
    for all $a \in A \setminus \{a_i\}$ **do**
      if TileDist(a, a_i) < 3 **then**
        $Q_{j,sp} \leftarrow NULL$
      end if
    end for
  end for
end procedure

TABLE I. TOTAL NUMBER OF MINERALS GATHERED IN TOTAL PER 1000 FRAMES. AVERAGE OF 20 RUNS.

Algorithm/Frames	1000	2000	3000	4000	5000	6000	7000	8000
Built-in	250	570	1010	1550	2158	2776	3398	4016
Mineral-lock	250	594	1066	1687	2408	3131	3860	4571
Queue	253	584	1066	1678	2405	3153	3904	4659
Co-op path	258	601	1084	1711	2444	3187	3927	4666
Co-op path + Queue	255	597	1090	1710	2431	3180	3931	4678
MineralPath	258	601	1086	1707	2443	3172	3909	4641

TABLE II. STANDARD DEVIATION IN MINERALS GATHERED PER 1000 FRAME STEP. STANDARD DEVIATION OF 20 RUNS.

Algorithm/Frames	1000	2000	3000	4000	5000	6000	7000	8000
Built-in	0.0	0.0	7.3	18.3	24.6	32.3	35.3	32.3
Mineral-lock	0.0	0.0	3.6	3.8	11.3	8.7	8.3	11.2
Queue	4.0	7.7	7.3	9.5	10.2	19.4	27.3	33.7
Co-op path	0.0	1.8	3.9	7.0	7.0	8.7	14.0	10.7
Co-op path + Queue	5.9	9.2	9.7	12.3	13.8	16.7	24.1	21.5
MineralPath	0	1.8	4.0	2.9	7.0	12.5	11.7	14.3

TABLE III. COMPUTATION TIME (MS.) PER 1000 FRAME STEP. AVERAGE OF 20 RUNS.

Algorithm/Frames	1000	2000	3000	4000	5000	6000	7000	8000
Built-in	1	2	3	3	5	5	6	7
Mineral-lock	2	5	9	14	20	25	31	39
Queue	7	22	45	81	132	197	275	367
Co-op path	2372	4784	7184	9571	11993	14390	16823	19271
Co-op path + Queue	2682	5436	8376	11471	14725	17972	21261	24494
MineralPath	7	16	31	46	64	80	97	114

only using the MineralPath part of the algorithm confirm this (see last row Table III). In practice it would only be necessary to check collisions with other worker units using the CalculatePath, since other worker units rarely enter the path given by this algorithm. We also observe that Built-in has a high variance. This is because a worker unit goes to a nearby mineral field if the current one is occupied, which may differ. Letting the worker units wait as is done in Mineral-lock gathers more than 14% than Built-in. Finally, it should be noted that applying scheduling or using better pathfinding can improve the resource gathering rate significantly. Combining both techniques appears to give a small edge.

A. Discussion

In bot tournaments (like the CIG StarCraft AI tournament), most bots use the Built-in algorithm. In general, the top bots (e.g., UALBERTABOT) use the Mineral-lock algorithm. For our bot LETABOT, winner of SCCAI 2014-2016, we use only the MineralPath part of the Co-operative algorithm. The tables reveal that this variant has a nice trade-off between CPU time and mineral gathered. Queue Based Scheduling is not incorporated in the LETABOT because it relies on a good approximation of the travel time to a mineral field. The Euclidean distance usually is not a good approximation for this. Storing travel data is an alternative, which was used in the experiment, but it relies on many runs. Pre-calculating the map is only possible if the map pool is known beforehand.

V. CONCLUSIONS & FUTURE RESEARCH

In this paper, we have presented five algorithms, which can be used for gathering resources in StarCraft. Improving the pathfinding has the most increase in minerals gathered, but using scheduling increases the minerals gathered significantly as well. In general, the more CPU intensive algorithms can gather more minerals, indicating a trade-off between CPU time and the mineral gathering rate.

For future research, one possibility to increase the mineral gathering rate even more is to use (limited) depth-first search instead of a greedy algorithm for scheduling. By looking ahead, it is possible to schedule the worker units better at the expense of more CPU. In terms of the pathfinding, the route back to the resource depot can be optimized as well. When a worker unit goes back to the resource depot, it tends to slow down as well when approaching the resource depot.

REFERENCES

[1] D. Christensen, H. O. Hansen, J. P. C. Hernandez, L. Juul-Jensen, K. Kastaniegaard, and Y. Zeng, "A data-driven approach for resource gathering in real-time strategy games," in *Agents and Data Mining Interaction*, ser. Lecture Notes in Computer Science, L. Cao, A. L. C. Bazzan, A. L. Symeonidis, V. I. Gorodetsky, G. Weiss, and P. S. Yu, Eds., vol. 7103. Springer, 2012, pp. 304–315.

[2] M. Buro, "ORTS A Hack-Free RTS Game Environment," in *Computers and Games (CG 2002)*, ser. Lecture Notes in Computer Science, J. Schaeffer, M. Müller, and Y. Björnsson, Eds., vol. 2883. Springer Berlin Heidelberg, 2003, pp. 280–291.

[3] S. Wintermute, J. Xu, and J. E. Laird, "SORTS: A human-level approach to real-time strategy AI," in *The Third AIIDE Conference*, 2007, pp. 55–60.

[4] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft," *Computation Intelligence and AI in Games, IEEE Transactions on*, vol. 5, no. 4, pp. 293–311, 2013.