Agente de Limpieza

Rodrigo Daniel Pino Trueba C412

Prestaciones

- Modular y fácilmente extensible
- Utiliza ejecución en paralelo
- Desarrollado utilizando TDD

Tablero

Representación

El tablero es una lista enlazada (`linked list`) con los objetos del tablero y posición, cuando el tablero esta vacío es una lista vacía. Esto trae la ventaja que no es necesario recorrer el tablero entero para buscar un elemento, mientras que indexar es $O(n)$ donde n es la cantidad de elementos en el tablero.

El tablero es una estructura de datos con un tamaño máximo de columna y fila y con operaciones de insertar y eliminar.

Objetos

El tablero esta compuesto por Obstáculos `Obstacles`, Suciedad `Dirt`, Corral `Crib`, Niño, `Kid` y Robot de la Casa `Robot`.

El Robot de la Casa es diferente a los demás objetos pues puede cargar un objeto en sus manos. Se estableció de esta manera con el objetivo de tener bien definida la relación Robot - Niño cuando estos se encuentran en una misma casilla. El Robot carga al Niño o este se encuentra en el piso.

Reglas

Los únicos objetos que se mueven por "voluntad propia" son los Niños y los Robots, los demás son estáticos, excepto por los obstáculos que puede moverse si un Niño los empuja.

El **Robot** de la Casa tiene definida las operaciones: Agarrar `Grab` donde toma un niño cuando tiene los brazos vacíos, Mover `Move`, Soltar `Drop` donde deposita el Niño en sus brazos en una casilla limpia o un corral y Limpiar `Clean` cuando se encuentra sobre una casillas sucia.

Cuando un Robot carga a un Niño solo puede moverse `Move`, o depositarlo `Drop`, otras acciones no están permitidas. Un Robot no puede moverse por encima de un Corral con un Niño dentro.

Los **Niños** tienen definidas las acciones: Mover `Move` y Empujar `Push`. Esta última es similar a mover excepto que obliga a un obstáculo (o una cadena de estos) a cambiar de posición (siempre que se pueda).

Los Niños se mueven de manera aleatoria o pueden no moverse. Cuando se mueven pueden generar suciedad a su alrededor. Se calcula la cantidad de suciedad dependiendo de la cantidad de Niños inmediatamente adyacentes a él. La suciedad se genera alrededor o en su posición de antes de moverse.

Agente

Representación

Los agentes `Agent` toman control sobre los Robots en el tablero y mueven dicho Robot en pos de una Tarea que deseen realizar. Las Tareas Asignadas `Assign Task` de los agente consisten en un objetivo, un objeto del tablero, específicamente una casilla sucia o un corral; y una secuencia de acciones a seguir para lograr dicho objetivo.

El comportamiento de los agentes esta definido por el tipo de agente que son, precavidos `Cautious` o atrevidos `Bold`. La diferencia entre estos radica en que los cautos cada vez que ocurre la variación sobre el tablero, vuelven a realizar una repartición de las tareas mientras que los atrevidos mantienen su objetivo hasta que no lo puedan lograr.

Otro aspecto que define sobre el comportamiento de los agentes es el algoritmo de búsqueda.

Algoritmo de Búsqueda

Se utiliza como algoritmo de búsqueda a `A*` por su generalidad y flexibilidad para cambiar fácilmente su resultado de mejor camino modificando la heurística por la que se guía. Cambiar la función de costo de `A*` implica que con sólo unas líneas de código se puede crear múltiples comportamientos de los agentes sin necesidad de alterar el mecanismos interno. En la iteración actual del proyecto hay tres heurísticas definidas:

- `ShortestPath` donde realizar cualquier acción aumenta el costo en 1, es equivalente al `BFS`. Se obtiene un agente que realiza siempre la tarea más cercana.
- `Balance` donde las acciones de guardar los niños en corrales son bien recompensadas y limpiar la tierra en el camino no incluye en un aumento de costo. También se recompensa si en su camino separa los clúster de niños de tamaño mayor o igual que dos y penaliza si depositar a un niño provoca la creación de un clúster. El agente que utiliza `Balance` prioriza guardar niños que se encuentran junto a otros en corrales mientras limpia la tierra en su camino (siempre que no este cargando un niño).
- `BalanceCrib` es una variación de `Balance` con mayor precisión sobre en que corral se deposita a los niños. Recompensa más a medida que el corral este más alejado de una casilla vacía. Para esto en cada iteración del tablero se realiza un `BFS` sobre los corrales donde se calcula el beneficio de dejarlo en cada uno. Un agente guiado por esta función de costo tiene el mismo comportamiento que uno utilizando `Balance` excepto que evita bloquear el acceso a los corrales interiores. (Un corral deja de ser accesible si está rodeado por corrales con niños en su interior).

El algoritmo se utiliza para buscar los objetivos de los agentes y una versión modificada para hacer una búsqueda general de todos los objetivos a los que puede alcanzar en el tablero.

Distribución de Tareas

Cuando hay solo un agente en el tablero es trivial la distribución, se toma la tarea de menor costo.

Cuando se tienen varios agentes es necesario una organización para lograr una optimización de los agentes y su tiempo. Este problema se traduce a asignar a cada agente tareas de manera tal que sea mínimo el costo total de la realización conjunta. Además se garantiza que cada tarea sea solo realizada por un solo agente y un agente solo realice una tarea.

Para resolver el problema se utilizó programación dinámica, donde la idea general es calcular iterativamente la mejor solución utilizando un sólo agente, la mejor utilizando dos, hasta llegar a la mejor solución utilizandolos todos.

Asumiendo que *m* son el total de agentes y *n* el total de tareas, donde cada agente es capaz de llegar al menos a una de las *n* tareas:

Sea c_{kij} el costo de realizar la tarea i con el agente j donde $k-1$ agentes ya ha sido asignados.

Sea $cost(i,j)$ la función de costo de realizar la tarea i con el agente j.

En el caso base con un solo agente:

$c_{1ij} = cost(i,j)$

Sea C_k la matriz de costos utilizando k agentes.

C_1 al ser un solo agente y conocerse los valores $cost(i,j)$ se define como:

$$C_1 = \begin{matrix} c_{111} & c_{112} & \dots & c_{11m} \\ c_{121} & c_{122} & \dots & c_{12m} \\ \dots & \dots & \dots & \dots \\ c_{1n1} & c_{1n2} & \dots & c_{1nm} \end{matrix}$$

Luego calcular C_k requiere calcular todos sus valores interiores, donde el costo de cada valor esta definido por:

$$c_{kij} = cost(i,j) + \text{minCost } (i, j, C_{k-1})$$

La fórmula se traduce en:

Utilizando k agentes donde el agente i realiza la tarea j buscar la mejor manera de utilizar $k-1$ agentes donde ninguno realice la tarea i ni sea el agente j.

La función minCost es la que se encarga de buscar el menor valor de la iteración pasada y que no ocurran colisiones, como que a un agente se le asignen dos tareas, o una tarea le sea asignada a dos agentes

El algoritmo necesita calcular m matrices con n filas y m columnas cada una con un costo de $n*m$. En cada casilla es necesario explorar la matriz anterior, que tiene un costo igual de $n*m$. Luego la complejidad temporal es $O(m*(nm*(nm))) = O(m(nm)^2) = O(n^2m^3)$

Calcular la mejor distribución de tareas requiere la creación de m matrices de n filas y m columnas dando una complejidad espacial de $O(nm^2)$

Es posible que no se puedan resolver todas las tareas con todos los agentes, en ese caso es necesario solo un subconjunto de estos. Cuando esto ocurre C_m tiene solo valores inválidos, luego se busca la mejor solución utilizando $m-1$ agentes en C_{m-1} y así sucesivamente hasta dar con una matriz C_k con $k >= 1$ que tenga valores válidos.

Simulación

La simulación es un ciclo donde en cada iteración se llama a la función `simAgent` que se encarga de mover los agentes en el tablero. En las iteraciones donde se ejecuta variación aleatoria, se llama a la función `simWorld` que mueve a los niños y genera suciedad según corresponda. `simWorld` es siempre llamado después de `simAgent`.

La simulación se detiene cuando todos los Niños se encuentran en los corrales o en los brazos de los Robots. Cuando la casa llega al 60% de la suciedad también se detiene la simulación.

Cuando `simAgent` se ejecuta, se realiza la asignación, distribución y ejecución de las de tareas según corresponda por agente.

La asignación y ejecución de tareas son realizada por los agentes en paralelo haciendo uso del módulo `Control.Parallel`. Las implementaciones secuenciales se mantienen con fines educativos y de comparación.

Usar paralelismo aumenta proporcionalmente la velocidad del programa de acuerdo a la cantidad de núcleos físicos disponibles en el `cpu`.

Cuando se ejecuta `simWorld` se analiza por cada niño su movimiento, dirección y suciedad creada.

Resultados obtenidos

Después de varias pruebas se llegó a la conclusión que lo que más afecta el comportamiento de los agentes es la heurística con el que funciona. El tipo del agente, si atrevido o cauto tuvo impacto notable solo con `ShortestPath`.

En el tablero 0, un ejemplo muy básico todos son bastante efectivos y tienen igual resultado.

En el tablero 1, los agentes con `ShortestPath` nunca terminan pues aunque logran mantener el tablero bajo control, los niños ensucian a la par y las misiones en los corrales no se realizan por su lejanía, en cambio con `Balance` siempre se logra guardar a los niños en las cunas.

En el tablero 2 los agentes que utilizan `ShortestPath` y `Cautious` se demoran alrededor de 700 turnos, mientras que con `Bold` lo hacen en menos de la mitad, con una variación baja de 1 turno. Esta diferencia entre los tipos de agente queda reducida cuando se utiliza `Balance` donde ambos resultados están sobre los 70 turnos.

En el tablero 3 y 33 comparando entre `Balance` y `BalanceCrib` esta última es el que mejor se desempeña, priorizando las corrales interiores. No obstante no es perfecta por que a veces se le quedan niños fueras. `ShortestPath` tampoco logra un buen resultado y termina bloqueando las cunas.

Luego el peso del resultado cae sobre el tipo de heurística que se utilice. El tipo de agente tiene muy poco efecto, aunque se nota una mejora con agentes atrevidos en ambientes altamente variables contra agentes cautos cuando se utilizó `ShortestPath`.

Interacción con el Proyecto

Tecnologías utilizadas

El proyecto fue desarrollado utilizando `stack-2.7.3` con `ghc 9.0.2-nightly`.

Compilar y ejecutar

Requiere tener instalado `Stack`

Compilar el proyecto

```
make install
```

Ejecutar el proyecto

```
make run
```

Compilar y ejecutar las pruebas

```
make unitest
```

Añadir nuevos tableros

Es fácil añadir nuevos tableros a los ya existentes. En `src/BoardBox.hs` en la función `boardSelect` basta con añadir una nueva rama con un identificador distinto a las otras. Se pueden definir todos los obstáculos del tablero haciendo uso de la función `make` y `makeMany`. Un ejemplo:

```
...
    -- Devuelve un unico objeto Robot con los brazos vacios en la posicion (3, 3)
let robot = make (Robot Nothing) (3, 3)
    -- Devuelve una lista con los objetos Dirt en las posiciones (5, 5) y (3, 3)
    dirt = makeMany Dirt [(5, 5) ,(3, 3)]
    rows = 6
    cols = 8
    objects = robot : dirt
-- worldInit crea el tablero y añade los objetos
in worldInit rows cols objects
```

Añadir nuevas heurísticas

En `src/Agent/Logic/Pathfinding/PathCalculation` es donde se encuentras las heurísticas definidas y donde se deben implementar las adiciones.

Para añadir una nueva heurística se debe:

- Definir un nuevo constructor a `PathCalcType` que represente el nombre de la heurística
- Añadir la función de calculo, esta tiene como datos a utilizar el `Board` actual, el costo del camino hasta ahora, el tipo de movimiento y un posible objetivo `Maybe Object`.
- (Opcional) En caso de que en cada llamado a la heurística haya que repetir las mismas operaciones sobre el tablero, estos se pueden pre-computar definiendo una nueva rama en la función `fillValues`

Cambiar parámetros de la simulación

El código en `app/Main.hs` es el encargado de ejecutar el proyecto y donde se establecen los parámetros iniciales. Se deben alterar los parámetros cuando se llama a la función `runSimulation`

```
runSimulation <boardNum> <changeTime> <agentType> <heuristic>
```

- `boardNum` selecciona un tablero de los defindios en `src/BoardBox.hs`.
- `changeTime` selecciona cada cuanto tiempo varia el tablero.
- `agentType` determina el tipo de agente si cauto `Cautious` o atrevido `Bold`.

- `heuristic` determina sobre las acciones del agente y la division de tareas, hasta el momento están definidas:
    - `ShortestPath`
    - `Balance`
    - `BalanceCrib`